Decomposition and Optimization in Near-Hierarchical Boolean Function Systems

By
Hassan Masum

A thesis submitted to
the Faculty of Graduate Studies and Research
in partial fulfillment of
the requirements for the degree of
Doctor of Philosophy

Ottawa-Carleton Institute for Computer Science
School of Computer Science
Carleton University
Ottawa, Ontario

December 18, 2002

© copyright 2003, Hassan Masum

National Library
of Canada

Bibliothèque nationale
du Canada

Acquisitions and
Bibliographic Services

Acquisisitons et
services bibliographiques

395 Wellington Street
Ottawa ON K1A 0N4
Canada

395, rue Wellington
Ottawa ON K1A 0N4
Canada

Your file Votre référence

ISBN: 0-612-83522-7

Our file Notre référence

ISBN: 0-612-83522-7

The author has granted a non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of this thesis in microform, paper or electronic formats.

L'auteur a accordé une licence non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de cette thèse sous la forme de microfiche/film, de reproduction sur papier ou sur format électronique.

The author retains ownership of the copyright in this thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without the author's permission.

L'auteur conserve la propriété du droit d'auteur qui protège cette thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

Canada

The undersigned hereby recommend to
the Faculty of Graduate Studies and Research
acceptance of the thesis

**DECOMPOSITION AND OPTIMIZATION IN NEAR-HIERARCHICAL
BOOLEAN FUNCTION SYSTEMS**

Submitted by
Hassan Masum, B.Sc., M.Sc.

in partial fulfilment of the requirements
for the degree of Doctor of Philosophy

Director, School of Computer Science

Thesis Supervisor

External Examiner

Carleton University
February, 2003

Abstract

This thesis contributes to research in near-hierarchical systems and Boolean Satisfaction in four main areas. First, work in hierarchical systems across a range of disciplines is surveyed, with a focus on nested hierarchies. A mathematical model related to divide-and-conquer is developed to provide intuition and demonstrate the utility of decompositional methods. Second, we suggest the "Neartree" - a tree-like structure with multiple parents allowed at nodes, but with the numbers of both parents and children bounded for computational tractability - as a useful framework for modeling and optimizing near-hierarchical systems. Properties of Neartrees viewed as data structures are discussed, as well as algorithms for working with Neartrees viewed as meta-models tying together associated sub-models at each node. Third, we define "hSAT" - an NP-complete hierarchical version of Boolean Satisfaction - and use it as the node-level problem in a Neartree in a series of experiments. These experiments both provide an implemented testbed to illustrate and validate the use of Neartrees as computational tools, and give new results for properties of hierarchical Boolean Satisfaction. In addition, Minimal Information Sets are motivated as a practical problem isomorphic to the maximization version of hSAT. Last, we suggest a series of steps forward for future work, including dynamic hierarchical clustering, a minimax version of hSAT, and potential applications in game-theoretic modeling, valuation, and distributed decision-making.

Acknowledgments

*The ancient Masters were profound and subtle.
Their wisdom was unfathomable.
There is no way to describe it:
all we can describe is their appearance.*

*They were careful
as someone crossing an iced-over stream.
Alert as a warrior in enemy territory.
Courteous as a guest.
Fluid as melting ice.
Shapable as a block of wood.
Receptive as a valley.
Clear as a glass of water.*

*Do you have the patience to wait
until your mud settles and the water is clear?
Can you remain unmoving
until the right action arises by itself?*

*The Master doesn't seek fulfilment.
Not seeking, not expecting,
she is present, and can welcome all things.
- Lao-tzu*

Thanks to all those who assisted in my academic vision quest:

Franz Oppacher for constant inspiration. Franz gave me space to grow, and his outlook and supervision style suited me perfectly. He has assembled a fantastic group of students around him over the years, many of whom I've learned from and become friends with. His combination of competence, insight, and wit made him an ideal supervisor.

My Thesis Committee:

- George Carmody has been a long-time ALEC contributor, and many genetics discussions with him inspired an interest in bioinformatics and genomics.
- Stan Matwin's Machine Learning course at the University of Ottawa was a useful experience, and I enjoyed reading Mitchell's *Machine Learning* for his comprehensive exam and getting pointers from him now and again.
- Thanks also to Dwight Deugo for helping to keep me grounded in my original over-optimistic thesis proposal.
- Thomas Homer-Dixon's book *The Ingenuity Gap* crystallized an important facet of "what's wrong with the world" for me. I thank him for agreeing to be external examiner, and undertaking to understand a thesis more interdisciplinary than most.

I would also like to acknowledge the help of Danny Krizanc, John Oommen, Carleton University, and Canada's National Research Council over the years.

Mark Wineberg and Steffen Christensen are both colleagues and friends, who never failed to help out and stimulate. I learned more from each of them than from any term's worth of courses I've taken.

Dan McShea gave helpful comments on the introductory hierarchy chapter, and pointed out references in the biological and philosophical literature. Insight was also gained from Alexander Chislenko (general inspiration), Robin Hanson (Idea Futures), Eric Baum (manifesto on "agoric AI"), and many members of Carleton's ALEC discussion group (general stimulation).

At a deeper level, we are all indebted to thousands of thinkers through the ages. And great music makers from Afro-Celt Sound System to Mozart propelled me forward through many a late night of programming, theorizing, and writing.

Hyder Masum provided a quiet example of what sustained hard work can achieve, and Zahra Masum reminded me that intellectuals should consider reproducing themselves.

Most of all, I thank **My Parents**, Arif and Aziza Masum. Their countless sacrifices and encouraging acts through the decades form the foundation behind all my work.

Table of Contents

Abstract	iii
Acknowledgments	iv
Table of Contents	vii
List of Tables	xiii
List of Figures	xiv
1. Introduction: Decomposition and Valuation	1
1.1 Contributions of the Thesis	1
1.2 A Motivating Analogy: The Black Box Boss	5
2. Decomposition and Hierarchies	8
2.1 Methods of Recursive Decomposition	8
Why Decomposition?	
Where and When Hierarchies	
Information Hierarchies: Taxonomies and Perspectives	
Hierarchy Theory	
2.2 Computational and Mathematical Connections	18
Trees	
Divide-and-Conquer, Dynamic Programming, and Clustering	
Software Hierarchies: Specialization and Composition	

Graphical Models

Distributed Computation, Reinforcement Learning, and Hierarchical EC

Multilevel Models, Lattice Theory, and Factoring

3. A Mathematical Model	28
3.1 Hierarchical Decomposition: Optimizing Parts vs Wholes	28
Goal: Math and a Model	
The Base Case: Multilevel Factorial Search Spaces	
Adding Overhead and Exponential Search Spaces	
Polynomial Search Spaces and Beyond	
3.2 Extensions and Consequences	42
Recurrence Equations and Extensions to the Model	
Applicability, and Extracting the Neartree Heuristic	
4. Neartrees: Generalizing Hierarchies	48
4.1 The Algorithmic Framework: Hierarchical Neartrees	48
Specification and Data Structures	
Discussion and Examples of Hierarchical Neartrees	
Hierarchical Neartrees and Flow Neartrees	
Similar Previous Work	
Operationalizing the Neartree	
4.2 Theory for Neartrees	57
Basic Recursive Unit: Node with Uplinks and Downlinks	
General Algorithmic Properties of Neartrees	
Optimizing Parts vs Wholes: The Neartree Case	
Hierarchical Kolmogorov Complexity	
Requirements and Goals for Neartree Applications	

5. hSAT: Going beyond SAT	69
5.1 Generalizing SAT	69
SAT in a Nutshell	
From SAT to Hierarchical SAT (hSAT)	
Top-down vs Bottom-up Specifications and Solution Methods	
5.2 Theory and Applications of hSAT	79
3-hSAT and 2-hSAT are NP-Complete	
SAT and hSAT Solution Heuristics	
Applications of SAT and hSAT	
6. hSAT: Implementation	89
6.1 Parameters and Program Flow	89
Program Functionality and Parameters	
Algorithms	
6.2 Data Structures and Optimization Methods	92
Data Structures and Class Structure	
Top-Down, Bottom-Up, and SAT optimization Pseudocode	
7. hSAT: Experiments	102
7.1 Experimental Setup	103
Test Cases: Benchmarks and Random Formula Generation	
Key Parameters, and Adaptive Parameter-Space Exploration	
7.2 Experiments in SAT and WalkSAT	107
Experiment: A First and Second Look at Traditional SAT	
Experiments: Tuning the Number of Restarts in WalkSAT	
Experiments: Tuning Computational Resources in WalkSAT	
7.3 Experiments in hSAT	121
Experiment: Two-Level hSAT	
Experiments: Three-Level and Four-Level hSAT	

8. Multiple Parents	127
8.1 Beyond Trees	127
Generating Multiple-Parent Neartrees	
Conflicting Parent Requests, and Parent Models of Children	
Multiple-Parent Heuristics and Caching	
8.2 Experiments: Multiple Parents	133
8.3 Multiple-Parent Models	137
Distributing Choice through Levels	
Steps Toward a Schema Theorem for Neartrees	
9. Clarifying Hierarchy	143
9.1 Beyond SAT: A Framework	143
State vs Value - A Framework	
Homogeneous, Heterogeneous, and Multiobjective Goal States	
Multiresolution Views and Other Dimensions of Differentiation	
9.2 hSAT minus SAT equals h	149
9.3 Phase Transitions	151
Experiments: Phase Transitions	
Mathematical Insights into Phase Transitions for hSAT	
9.4 h-MAX-SAT and Minimal Information Sets	156
Experiment: MAX-SAT	
Minimizing Information Overload through Hypergraph Vertex Covers	
10. Data to Neartrees, Dynamic hClustering, and DisSATisfaction ..	165
10.1 Noticing Neartrees	165
How Hierarchical?	
Experiment: Expanding hSAT into SAT	

Evolving hSATs and Neartrees	
Empirical Evidence for Neartrees, and Statistical Considerations	
10.2 Creating Neartrees through Clustering	174
Clustering in a Nutshell	
Hierarchical Clustering	
Further Methods, and Neartree-Specific Clustering	
10.3 Dynamic Clustering and Judging with Neartrees	185
Goals for a Dynamic Neartree Clustering Algorithm	
Distributed Evolving Neartrees: Ontological Order from Chaos	
Applications and Model Extensions	
10.4 Valuation	191
Valuing a Portfolio, or a Planet	
Risk Analysis and DisSATisfaction	
Strategies for Risk-Averse and Adversarial Cases	
DisSAT Theory, and Reliable Networks	
11. Long-Term Themes, and Conclusion	201
11.1 Autonomous Nodes	201
Competition vs Cooperation	
A Simple Model for Explaining Hierarchy Formation	
11.2 Algorithmic Valuation and Meritocratic Algorithms	207
Local vs Global Value	
Scaling Up Fitness Evaluation	
The Design Space of Distributed Meritocratic Algorithms	
11.3 Neartree Networks and TOOL	214
Nearthree Networks, Multiobjectivity, and Shared Reality	
TOOL: The Open Opinion Layer	
11.4 Conclusions	221

Glossary 224

References 226

List of Tables

Table 1... Neartree Notation	50
Table 2... Components of a Node	57
Table 3... Neartree Parameters	89
Table 4... Extrapolation for required resources in 1-level WalkSAT	120
Table 5... Multiple-Parent Neartree Sizes	129
Table 6... Elements of Hierarchical SAT	144
Table 7... Hierarchical TSP	146
Table 8... Meritocratic Algorithm Design Space: Markets vs EC	212

List of Figures

Figure 1...	Recursive subdivision of a problem	30
Figure 2...	A Neartree	48
Figure 3...	Paths to the root, and the information compression barrier	62
Figure 4...	Hierarchical Boolean Satisfaction	74
Figure 5...	Internal structure of an HNode	93
Figure 6...	A first, flawed look at SAT	108
Figure 7...	A better look at 1-level SAT	111
Figure 8...	Tuning Restarts in WalkSAT ($c = 256$)	113
Figure 9...	Tuning Restarts in WalkSAT ($c = 64$)	114
Figure 10...	Resource usage in WalkSAT ($c = 64$)	117
Figure 11...	Resource usage in WalkSAT ($c = 256$)	118
Figure 12...	Resource usage in WalkSAT ($c = 1024$)	120
Figure 13...	Two-level hSAT	123
Figure 14...	Three-level hSAT	125
Figure 15...	Four-level hSAT	126
Figure 16...	Two-parent, two-level hSAT	134
Figure 17...	Multiparent, two-level hSAT	136
Figure 18...	Single-level SAT phase transition	152
Figure 19...	Two-level hSAT phase transition	153
Figure 20...	Two-level h-MAX-SAT	159

1. Introduction: Decomposition and Valuation

“The competent programmer is fully aware of the strictly limited size of his own skull.”

-EW Dijkstra

1.1 Contributions of the Thesis

This thesis has **three goals**. First, to understand the phenomenon of hierarchy through a mathematical model, and through high-level analysis informed by interdisciplinary and historical research. Second, to develop and experimentally test a framework for computationally modeling large-scale hierarchical problems, which we call the “Neartree”. Third, to propose and experiment with a specific computational testbed for experimenting with large-scale hierarchies, which we dub “hierarchical SAT” (or hSAT). As a bonus, we suggest many specific areas for further development of theory and algorithms - in both computer science and other fields.

How can we understand large distributed systems? One method is to cluster similar objects together into categories, and **recursively group** these categories into hierarchies. This allows irrelevant details to be ignored, and an appropriate level of system detail to be used. From an algorithmic point of view, such simplification is necessary due to bounded resources; we summarize the ways in which this point has been recognized and exploited in many fields.

Multilevel divide-and-conquer is at the heart of many recursive algorithms in computer science. We derive a **mathematical model** which provides upper bounds on the potential speedup of a general class of hierarchical algorithms. This is in a sense the “best-case scenario”, where we can decompose a problem into independently-solvable subparts and easily recombine their solutions.

We define a general data structure - the **Neartree** - which can be used to represent and

analyze certain classes of large data sets in a way that is relatively comprehensible by humans. Essentially, a Neartree is a close relative of a tree, where nodes have up to p parent nodes and optionally a bound c on the number of child nodes. Additionally, we usually require a “compositional” property, discussed later. Algorithmic properties, potential applications, and connections with previous work are discussed.

Now that we have defined the Neartree, what can we do with it? To a large extent, this depends on the user’s goals and problem domain: data structures are only a starting point, that can be used in multifarious ways. Three important goals are:

1. **Optimizing** large static systems, by finding an optimal configuration of elements.
2. **Valuing** large systems (static or dynamic), by intelligently combining valuations of subparts.
3. **“Pessimizing”** large dynamic systems, by using a minimax-type approach of minimizing the worst-case loss.

We focus primarily on the first of these goals, while discussing the other two as well. Our test bed is a new algorithmic problem: **near-hierarchical SAT**. In contrast to traditional SAT, we define hSAT as a hierarchical (or near-hierarchical) analogue of SAT, so that variables can themselves be recursively composed of sub-variables which are combined via some boolean formula. Traditional SAT is discussed, and the development of hSAT naturally motivated; the problem is proved to be NP-complete.

We have created a program which implements the Neartree idea, which is the focus of the next sections. Our program provides a modular and extensible framework for research into near-hierarchical systems; the Java **implementation** is discussed and **core algorithms** explained.

At each node of the Neartree in a run of our program, hSAT is the specific problem we

use to explore the properties of both Neartrees and hSAT. We define several experimental goals, and run corresponding **experimental series**. First, it is important to understand the node-level implementation of our SAT solver (a variant of WalkSAT). We run several experiments that give some insight into how the solver operates, pointing out an intriguing sigmoid-curve behavior of performance vs $\log(\text{resources})$ in the process.

The initial experiments make clear the necessity for different approaches past an infeasibility boundary; we thus turn to **experiments with hSAT**. Properties of hSAT vs single-level SAT are demonstrated through a series of several experiments, showing in particular how solution probabilities vary with independent variables like the number of levels, number of children per node, and difficulty of node-level problems.

It is also shown how an hSAT problem can be “**expanded**” into an **equivalent SAT** problem - an expansion which yields exponentially-sized SAT instances. In exploring this analogue of one of the most fundamental theoretical problems of computer science, we learn about both generalized-SAT and the properties of the Neartree.

Although hierarchies are very useful, there are many cases where near-hierarchies are more suitable. **Multiple parents** are the key distinction, and introduce a host of conceptual and practical difficulties. Hence we explore how hSAT changes with multiple parent nodes. The issue is explored both experimentally and theoretically.

After intuition has been built up from theory and experiments, we return to the basic idea of a hierarchy, and clarify several subtle issues. **Phase transitions** are a central algorithmic property of SAT; their occurrence in hSAT is investigated. Other types of problems aside from those derived from propositional logic can of course be used in the Neartree framework; we demonstrate one such problem by defining a **hierarchical version of Weighted Vertex Cover on Hypergraphs**, showing how it can be translated into a weighted hSAT representation, and pointing out how it naturally models the problem of

deciding which subfields of a category of knowledge to learn in order to achieve understanding or competence.

A key issue in the practical use of Neartrees is the generation of **Near**trees **from data**. Data may or may not have an underlying hierarchical representation; metrics are proposed which measure how hierarchical a set of interrelated objects are. We discuss metrics for measuring “how close to a Neartree” an object is, suggest the use of evolutionary computation for evolving Neartrees and hSATs, and investigate the potential use of clustering algorithms as one method for automating the creation of Neartrees. One application of wide interest and utility is clustering dynamic data points; we suggest that a dynamic Neartree is a useful way of representing such data.

Another practical application that we explore is to use Neartrees for **computational valuation**, to store a decomposed representation of the value of different parts of a composite product. This algorithmic product could for instance be an abstract space representing some set of possible solutions. As well, we explore the idea of **minimax valuation**. The intuition here is that real life is ever-changing and full of “unknown unknowns”, so it’s often best to plan defensively by putting your system into states where likely neighboring states are not too bad.

We close off the thesis with a look forward at several broad areas for further development. Extending nodes to have **autonomous goals** may allow the modeling of many key real-life difficulties, such as political competition and game-theoretic dilemmas. If nodes are not “computational slaves”, but rather each have their own independent goals, then the development of any higher-level strategy must take the satisfaction of these lower-level goals into account.

A related consideration is the question of “metavaluation”. What do the algorithmic properties of markets and EC have in common? We contend that they are two particular

instantiations of the same idea: create a population of locally-optimizing actors, and set up mechanisms to encourage the emergence of feedback structures each of which rewards a distinct type of “good” behavior. (Here “good” is defined as a desirable set of values for some set of observable system variables.) This creates a dynamic adaptive structure that, over time, directs attention toward better parts of solution space. However, to date both these areas have developed independently. We claim that both areas are in fact instances of a more general meta-algorithm, and describe the “**meritocratic algorithm**” design space within which these two instances live.

One can consider going beyond a single Neartree, to a **Nearthree Network** which represents a set of distinct opinions. By combining multiple Neartrees, we get the idea of an overlaid Neartree structure, with overlapping Neartrees forming a Neartree Network. We suggest that focusing on these overlapping portions could highlight thought-flows, areas of consensus, and crucial points of difference. Further, we point out the connection to a general framework for distributing, sharing, and valuing opinions that we have previously proposed, called “**TOOL: The Open Opinion Layer**”.

Along the way, connections with related fields are discussed, as are mathematical and computational methods. The overarching theme is that of adaptive optimization and valuation through decomposition. Our ideas are validated by conceptual taxonomies, mathematical modeling, and an implemented algorithm. We also design and validate a set of metarules and heuristics which are broadly applicable. Enjoy the journey!

1.2 A Motivating Analogy: The Black Box Boss

Let’s consider an analogy to gain intuition and explain some of the background goals behind this thesis. In our analogy, we would like ways of extracting “maximum value” from a set of black box evaluations, knowing as little as possible about the black boxes. (Defining “value” is highly nontrivial, as we’ll discuss later in the thesis - for now a typical

optimization problem furnishes an example, by defining value as progress toward the optimum state.)

Consider a model in which we have a collection of Black Boxes (BBs), each of which performs some function which is complex relative to its inputs and outputs. In order to coordinate the activity of the BBs, a “manager algorithm” is desirable; we’ll call this the **Black Box Boss (BBB)**. The role of the BBB is to evaluate and direct the overall activity of the BBs which it oversees, e.g. to influence their goal states and decide how much progress each is making toward some larger problem. Our goal, then, is to think of rules of thumb for the BBB which are relatively context-insensitive - and then to validate some of these rules of thumb through experimentation and theoretical justification.

The Black Box Boss will use **heuristics** like these:

- Every BB gets a fair shot at the problem (adequate resources and information).
- We focus on good BBs, sampling the others decreasingly (selection).
- We watch BBs and pick out promising ones (prediction).
- We look beyond fluctuations to average performance (statistics).
- We specify only a few interface variables for BBs (choosing parameters).
- The interaction strength and type between BBs should be estimated (epistasis).
- Don’t let the BBs get stuck in ruts (convergence and diversity).
- Competition (driving coevolution) and cooperation both have their benefits.

A good BBB is like a generic system evaluator, which can also “gently manage” the activities of its BBs by varying their environment based on performance parameters. In order for the BBB idea to be truly useful, a specific one should work for a large class of underlying problems. Thus **the BBB itself is the object of interest**, that is evolved and improved over successive generations. (It may be that the effort involved in setting up a BBB only becomes useful when the underlying problem is relatively large-scale and long-

term - current human society and technology presents such problems in abundance.)

Now consider the case where there are many BBs (hundreds to trillions). In this case, we need some way of organizing them. One such way, which is the subject of much of this thesis, is to use a **(near-)hierarchical control structure** to organize and direct them. (Another might be to use markets, reputations, and fitness functions to filter them - a point we return to in the last chapter.)

This suggests other heuristics that the BBB might consider:

- Organize a near-hierarchy in a recursive way, so that each node can consider just a few other nodes but the overall group solves a shared problem.
- Calculate how much resource expenditure (i.e. time, information, effort) can be saved, by chunking BBs into small groups and aggregating groups.
- If trying to optimize a near-hierarchy, simultaneously consider what higher-level nodes want from lower-level nodes, and what lower-level nodes have proven capable of doing - interpolate between ideal solutions and demonstrated capabilities.
- When moving from a hierarchy to a near-hierarchy, consider how to resolve conflicting goals from multiple parents, and how and when to reuse past solutions.
- Mediating information flows - an “economy of merit”.
- Multiple-resolution levels of summary and analysis.

In a sense, much of this thesis describes an early-stage BBB for near-hierarchical systems, that uses decomposition and valuation to understand the interaction of BBs in an algorithmic environment. We also make a start in researching BB management (as outlined above), and finally incentive engineering (moving beyond “management” to subgoal construction and evaluation).

2. Decomposition and Hierarchies

“Nothing is particularly hard if you divide it into small jobs.” - Henry Ford

2.1 Methods of Recursive Decomposition

Why Decomposition?

When dealing with large problem spaces, it is essential to find ways to **decompose them into simpler subproblems**. Information overload, bounded rationality, and combinatorial explosions are factors limiting a single computationally-bounded entity from being able to compute everything about a complex world. In such cases, task breakdown is a ubiquitous strategy.

There are many advantages to decomposition:

- **Speed.** Given superlinear increase in problem difficulty with problem size, a modular approach will be faster as long as the extra time involved to implement modularity (i.e. setting up and recombining the modules) is not too large.
- **Adaptiveness.** Most problems in the real world are not static. Since current states of problems are highly correlated with previous states, the same usually holds true of solutions, and so a modular approach allows solving only the changed portions of a problem. Design and evolution of solutions is also easier when a pool of modular subsolutions to subproblems already exists.
- **Comprehensibility.** For human users and designers, a modular solution approach is essential in comprehending solution methods.
- **Knowledge Representation.** Each decomposition method gives a “filter” through which to look at other problems, which may help in stripping away irrelevant detail.

[Dijkstra 1972] is a historic paper which deals with cognitive limitations of programmers

and the increasing complexity of the software design process, and suggests general heuristics for dealing with software complexity. Dijkstra suggests that designing systems with **hierarchies and factorability in mind** is a key heuristic:

“...the only problems we can really solve in a satisfactory manner are those that finally admit a nicely factored solution. At first sight this view of human limitations may strike you as a rather depressing view of our predicament, but I don't feel it that way, on the contrary! The best way to learn to live with our limitations is to know them. By the time that we are sufficiently modest to try factored solutions only, because the other efforts escape our intellectual grip, we shall do our utmost best to avoid all those interfaces impairing our ability to factor the system in a helpful way. And I cannot but expect that this will repeatedly lead to the discovery that an initially intractable problem can be factored after all.

....Hierarchical systems seem to have the property that something considered as an undivided entity on one level, is considered as a composite object on the next lower level of greater detail; as a result the natural grain of space or time that is applicable at each level decreases by an order of magnitude when we shift our attention from one level to the next lower one. We understand walls in terms of bricks, bricks in terms of crystals, crystals in terms of molecules etc. As a result the number of levels that can be distinguished meaningfully in a hierarchical system is kind of proportional to the logarithm of the ratio between the largest and the smallest grain, and therefore, unless this ratio is very large, we cannot expect many levels. In computer programming our basic building block has an associated time grain of less than a microsecond, but our program may take hours of computation time. I do not know of any other technology covering a ratio of 10^{10} or more; the computer, by virtue of its fantastic speed, seems to be the first to provide us with an environment where highly hierarchical artefacts are both possible and necessary.”

We will therefore discuss a selection of the most important algorithmic methods for problem decomposition. Our focus is on recursive methods - those which can be used to decompose a problem on multiple levels, where the process at each level is similar (modulo parameter variation).

Where and When Hierarchies

What is a hierarchy? A strict hierarchy can be modeled as a **tree**, which immediately gives us access to the large set of algorithms developed for dealing with trees. However, as we will see later other definitions can be used, e.g. allowing multiple parents or imposing structure on the set of allowable children.

One important distinction is between nested and non-nested hierarchies. A **nested** or **compositional hierarchy** is one where each successive level forms a more detailed representation of the previous level. A parent node in a nested / compositional hierarchy is semantically and computationally composed of its child nodes.

In contrast, in many non-nested hierarchies a tree-like data structure just happens to describe e.g. the computational or information flow in a system, without implying composition or containment. “**Flow hierarchy**” is perhaps a good term to use for these types of non-nested hierarchies, where the edges between nodes indicate a flow of information or power.

In addition to compositional and flow hierarchies, we can distinguish a third natural type: the **classification hierarchy**. Levels in a classification hierarchy denote more general categories moving toward the root, and more specific categories moving away from it; examples include the classical taxonomy of biological species, postal addresses, and object type hierarchies.

(Note the overlap between classification and compositional hierarchies - it is natural in many cases to consider a general category as being "composed of" or described by the collection of its subtypes. To the degree to which this can be done, a classification hierarchy is also a compositional hierarchy - e.g. there might be a case for this dual typing in the case of postal addresses.)

Biological systems are one of the domains where hierarchy is most naturally present. Biological taxa generate a set of hierarchies used to classify biological organisms, e.g. [Hall 2000]. In [GordonR 1999], a hierarchical structure to DNA is postulated, which permits systematic interaction with the developing embryo. (Note that the phylogenetic trees representing the genealogy of species, although they may seem like compositional hierarchies at first glance, are really examples of what we define as “flow hierarchies” and as such are distinct from compositional or nested hierarchies.)

In an individual organism, hierarchy is present in the multiple levels of structure, from cell components to organ systems; [Nehaniv 1999] discusses models for computational morphogenesis and hierarchical complexity in biology. It has been postulated for many decades - since at least the writings of Herbert Spencer in the late 19th century - that there is a long-term trend toward increasing degrees of hierarchies in biological organisms. In [McShea 2001], this hypothesis is explored empirically: first a scale of degree of hierarchy is defined, which can be thought of at its coarser resolution as the number of nested levels present in an organism. Extracts from the fossil record (i.e. reconstructed organisms) are then graphed with respect to this scale. It seems clear from this evidence that there is indeed a **long-term trend toward deeper biological hierarchies**, over the time scale of life on Earth. (Note that this trend is an increase in the maximum degree of hierarchy present across all organisms at a given point in time - not necessarily an increase in the mean level of hierarchy present in the population as a whole.)

Ecologists think of populations, communities (i.e. groups of different species), ecosystems and the entire biosphere as a hierarchy. These are more a heuristic partitioning of reality than necessarily a true ontological view. Even though there are no self-labeled levels in the biosphere, using such levels as an approximation helps to frame questions and interpret data. Everyone acknowledges that “interactions are the essence”, i.e. that the actual pattern of inter-species interactions is the true underlying model that must be kept in mind. However, defining hierarchical levels assists useful progress in understanding the otherwise overwhelming complexity of ecosystems.

[Salthe 1993] distinguishes between scalar and specification hierarchies, which are roughly analogous to what we call compositional and classification hierarchies respectively. Talking from the point of view of biological development (but bringing in many other fields), Salthe suggests that scalar hierarchies are distinguished by scales of time or space, or possibly some other observable quantity; more inclusive objects are at larger scales, operate more slowly, and can have as components objects of smaller scale. In contrast, in specification hierarchies the more inclusive classes are naturally described in more general terms, with less inclusive objects being specialized types of more general ones - analogous to type or is-a hierarchies in programming.

Organizations are often run on a hierarchical basis. While strict hierarchies have perhaps become less common over time, several different hierarchies representing functions from management authority to financial decision-making to political power are still present in most organizations and societies. [Klein & Kozlowski 2000] survey multilevel theory, research, and methods in organizations; an example of the type of issue discussed is performance measurement, where one wishes to assess the contribution of an individual to group success (or indeed the contribution of a group to the success of a larger group of which it is part).

Perhaps the largest-scale conception of hierarchical organization is the idea of “society as superorganism”. Since antiquity, isolated scholars have noted the similarity between a body composed of symbiotic parts with distinct functions and a society composed of symbiotic individuals with distinct contributions to the commonweal, speculating that societies formed some dimly-sensed higher organism. In the 20th century, this idea was developed by numerous thinkers including Pierre Teilhard de Chardin, H.G. Wells, Joël de Rosnay, Gregory Stock, and Alexander Chislenko; for an overview see [WebGlobalBrain].

Gregory Stock names the superorganism “Metaman”. As he explains the concept in [Stock 1993]:

“Metaman is that part of humanity, its creations, and its activities that is interdependent—joined together by trade, communications, and travel. At the moment, the superorganism is primarily the world's industrialized countries and the urban areas in developing lands, but it is growing and spreading rapidly into the rural regions of the third world that are as yet peripheral to it.

....Countless nonbiological innovations from radio telescopes to X-ray machines have enormously extended human senses, and this is only a hint of the vast sensory capabilities of Metaman. It observes the growth of a country's population, watches the concentration of carbon dioxide in the atmosphere rise, and traces the shape of an oil deposit miles beneath the ground.... As each of us responds individually to fragments of this huge stream of information, our responses couple and merge to create the large-scale adjustments that constitute Metaman's integrated response to information.”

To what extent is this idea of society as an emergent computational hierarchy writ large a useful model, and to what extent a non-falsifiable flight of fancy? That there exist coherent decision-making organizations and social computational patterns collectively surpassing detailed grasp by any single human is likely the case. Beyond this, time will tell.

Other domains where nested hierarchies appear include:

- Geographic Information Systems.
- Virtual reality representation.
- Economic modeling - region, country, sector, company, line of business.
- Network design, e.g. [Richa et al 2001].
- Valuation - e.g. indicators and accounting subdivisions.
- Societal organization - see [Pooler 2000] for thoughts on a Canadian perspective.
- Large-scale structure of matter in the universe.
- Time itself - we deal with multiple nested time scales on a daily basis.

When do these hierarchies provide a useful model? One criterion is that they should form a **more compact model** of the system than is possible without using hierarchies. (This is conceptually related to the theory of Kolmogorov Complexity, which classifies objects by

the length of their minimal-length description; see [Li & Vitanyi 1997] for a detailed exposition). Similarly, a hierarchical description might be more efficient in terms of computational resources, adaptability, or controllability. If a system can be described or predicted in a more compact or accurate way using a hierarchical model, then that model is a useful representation of the system.

(We will revisit this idea after introducing hierarchical SAT, when we will see that one advantage of hierarchical representations is that they allow specification and solution of problems which would otherwise be prohibitively large. One consequence of these representations is that problems specified using them can be not just NP-complete but PSPACE-complete in a natural way.)

Information Hierarchies: Taxonomies and Perspectives

Forming hierarchies is a standard way of dealing with large and complex environments. Choosing the appropriate level of the hierarchy allows selection of abstraction and detail level; algorithmic properties to be discussed later imply a large degree of independence between nodes and levels of the hierarchy.

Examples of information hierarchies include **information organization** schemes like the Dewey Decimal and Library of Congress systems, ontologies for Web resources, and **file systems**. Current file systems are usually organized in a tree structure (although this is slowly changing); this gives access to any piece of data in time logarithmic in the total amount of data stored, if the next step in the path to the data is known at each level. Information ontologies like online directories are also largely hierarchical, with a good deal of variation; for example, it is common to have multiple parent categories of a given category, corresponding to different ways of subdividing the information resources.

[Svenonius 2000] discusses hierarchies from the perspective of information organization. Several kinds of hierarchies are distinguished, each of which has seen substantial use by

organizers and seekers of information:

- **Genus-species hierarchies** are essentially just classification or inheritance hierarchies. In information organization, two key properties of these hierarchies are “mutual exclusivity” (nothing belongs to more than one class) and “exhaustivity” (everything belongs to a class). In software hierarchies, the second property is true of a fully object-oriented language, while the first is true as long as there is no multiple inheritance.
- **Perspective hierarchies** are more general types of subdivision: each non-leaf node can be subdivided into child nodes by any distinguishing property on the child nodes (e.g. a library might divide Programming into Programming - Genetic, Programming - History, Programming - Languages, Programming - Mathematical, and so on). While there is usually just one inheritance hierarchy in a domain, there can be as many perspective hierarchies as there are points of view from which to view the domain. (These are related to our previous definition of flow hierarchies, if one considers each node as aggregating or separating flows of human attention up and down in the hierarchy respectively.)
- **Composition hierarchies** (whole-part relationships). These are similar to compositional hierarchies as previously defined. They can sometimes be distinguished from genus-species hierarchies by the fact that their relationships are more empirical than logical - e.g. the language LISP is a type of programming language by definition, but the particular libraries that are part of LISP may change.

Perspective hierarchies are usually more appropriate to domains where concepts are polymorphic and otherwise not well-defined, while genus-species hierarchies are suitable for precisely-defined and stable categories. (Svenonius also distinguishes topic-subtopic hierarchies and region-subregion hierarchies. For our purposes, we will regard these as particular applications of the three basic types above.)

Graphs are often used to **visualize hierarchies**. Choosing the right visualization method

for a large hierarchy is a nontrivial task, with a large effect on usability and perceptual relevance; [Battista et al 1999] discusses algorithms for graph layout. There has been some work on graph layout algorithms for application specifically to large hierarchical graphs, e.g. [Eades & Feng 1997]. (We use the layered approach in our illustrations, while our software uses a more complex approach to accommodate the large lower layers of the digraph.)

Hierarchy Theory

Perhaps surprisingly, there has been relatively little systematic exploration of large-scale hierarchical systems as a field of study; although many areas of research deal with hierarchical phenomena, the actual properties and mathematical techniques applicable to a large hierarchical system are usually not the area of focus. One exception is “Hierarchy Theory” as developed by researchers in general systems theory; [Ahl & Allen 1996] discusses some of the basic ideas.

Although the approach of the general systems version of Hierarchy Theory to date appears to be relatively sparse in quantitative tools, there are still useful qualitative concepts and insights from a computational perspective for anyone wishing to build models of hierarchical systems. We quote below from “A Summary of the Principles of Hierarchy Theory”, online at [WebHierarchy]:

Hierarchy: in mathematical terms, it is a partially ordered set. In less austere terms, a hierarchy is a collection of parts with ordered asymmetric relationships inside a whole. That is to say, upper levels are above lower levels, and the relationship upwards is asymmetric with the relationships downwards.

Hierarchical levels: levels are populated by entities whose properties characterize the level in question. A given entity may belong to any number of levels, depending on the criteria used to link levels above and below. For example, an individual human being may be a member of the level i) human, ii) primate, iii) organism or iv) host of a parasite, depending on the relationship of the level in question to those above and below.

Level of organization: this type of level fits into its hierarchy by virtue of a set of

definitions that lock the level in question to those above and below. For example, a biological population level is an aggregate of entities from the organism level of organization, but it is only so by definition. There is no particular scale involved in the population level of organization, in that some organisms are larger than some populations, as in the case of skin parasites.

Level of observation: this type of level fits into its hierarchy by virtue of relative scaling considerations. For example, the host of a skin parasite represents the context for the population of parasites; it is a landscape, even though the host may be seen as belonging to a level of organization, organism, that is lower than the collection of parasites, a population.

The criterion for observation: when a system is observed, there are two separate considerations. One is the spatiotemporal scale at which the observations are made. The other is the criterion for observation, which defines the system in the foreground away from all the rest in the background. The criterion for observation uses the types of parts and their relationships to each other to characterize the system in the foreground. If criteria for observation are linked together in an asymmetric fashion, then the criteria lead to levels of organization. Otherwise, criteria for observation merely generate isolated classes.

The ordering of levels: there are several criteria whereby other levels reside above lower levels. These criteria often run in parallel, but sometimes only one or a few of them apply. Upper levels are above lower levels by virtue of: 1) being the context of, 2) offering constraint to, 3) behaving more slowly at a lower frequency than, 4) being populated by entities with greater integrity and higher bond strength than, and 5), containing and being made of - lower levels.

Nested and non-nested hierarchies: nested hierarchies involve levels which consist of, and contain, lower levels. Non-nested hierarchies are more general in that the requirement of containment of lower levels is relaxed. For example, an army consists of a collection of soldiers and is made up of them. Thus an army is a nested hierarchy. On the other hand, the general at the top of a military command does not consist of his soldiers and so the military command is a non-nested hierarchy with regard to the soldiers in the army. Pecking orders and food chains are also non-nested hierarchies.

....Arthur Koestler in his "Ghost in The Machine" referred to the notion of holon, which means an entity in a hierarchy that is at once a whole and at the same time a part. Thus a holon at once operates as a quasi-autonomous whole that integrates its parts, while working to integrate itself into an upper level purpose or role. The lower level answers the question "How?" and the upper level answers the question, "So what?"

2.2 Computational and Mathematical Connections

Trees

Trees are a fundamental data structure in computer science. As a **mathematical representation of hierarchies**, they are also fundamental to classification and analysis of information. One crucial property is that every node aside from the root has exactly one parent node; this implies that subtrees of any two distinct nodes at the same height in the tree are themselves distinct, which in turn often implies that these subtrees can be treated independently for computational purposes.

Balanced trees (with low variation across all leaf nodes in the path length from root to leaf node) have height logarithmic in the total number of nodes. Many efficient implementations like red-black trees exist that maintain a near-balanced state with low cost per insertion or deletion of an element into the tree; see [Knuth 1997].

If a large set of objects is partitioned into subsets, and these subsets are then joined to make larger units, the result is a tree-like structure where non-leaf nodes represent subsets of objects. In this structure, any two non-leaf nodes have the property that

$$A \cap B = \emptyset, \text{ or } A \cap B \in \{A, B\}$$

(i.e. either they are disjoint, or one is a subset of the other). Hence each leaf node represents an atomic object, and each non-leaf node represents the class of objects corresponding to the leaf nodes which are its descendants. This structure is called an **n-tree** in [GordonA 1999]; its graphical representation can be thought of as an unordered rooted tree where all non-leaf nodes other than the root have degree at least three.

One can extend the idea of an n-tree to associate a value (or “height”) with each non-leaf

node; this associates a metric with each subset. Often the values themselves are not important, but only the relative rank of nodes; in this case, one speaks of a “**ranked tree**”. The difference between a ranked tree and an n-tree is that the non-leaf nodes of a ranked tree can be compared (with respect to a metric which denotes size, importance, etc) even if one is not a descendant of the other, whereas only such non-leaf nodes can be compared in a non-ranked tree.

Divide-and-Conquer, Dynamic Programming, and Clustering

Divide-and-conquer is a common method in algorithm design. The input problem to an algorithm is **split into smaller subproblems**, the algorithm is called recursively on these subproblems, and the subproblems are combined to get a solution to the original problem. Examples include sorting algorithms like Quicksort and merge sort, and many geometric algorithms.

The running time of such an algorithm can be described by a **recurrence equation**: following [Cormen et al 2001], suppose $T(n)$ is the time to solve a problem of size n , each problem is divided into a subproblems of size $1/b$ times the original, and it takes $D(n)$ time to divide a problem into subproblems and $C(n)$ time to combine the solutions. Then the recurrence describing the solution time is:

$$T(n) = a T(n/b) + D(n) + C(n)$$

(We will discuss a similar but more detailed mathematical model later in the thesis, as well as solutions to that model and to the recurrence equation above.)

Dynamic programming can be viewed as a variation of the divide-and-conquer paradigm, where the **subproblems share sub-subproblems**. More generally, there may be many levels of sub-sub-sub...-subproblems; the crucial point is that they form a layered DAG

(directed acyclic graph), where the size of the layers does not grow “too fast” as one works backward. In this case, a solution can be found by solving all the problems at the last level, and then working forward level by level, optimizing each level given the already-computed optimum of previous levels at each step. Examples include the longest common subsequence and longest increasing subsequence problems, and many operations research problems.

Clustering deals with dividing data points in some metric space into **similarity classes**, so that each subset contains data points that are close to one another; this process can yield an implied hierarchical partitioning of the original data. Clustering methods will be discussed in detail in a later section, when we discuss constructing Neartrees from data and propose a new algorithm for dynamic hierarchical clustering.

Software Hierarchies: Specialization and Composition

In object-oriented approaches to programming, there are (at least) two basic approaches to forming object hierarchies: “is-a” and “has-a”. Is-a hierarchies formalize a **functional hierarchy**:

If node Child is a child of node Parent, Then Child is-a particular type of Parent.

For example, Rectangle is-a Quadrilateral; we start with a general object at the root and differentiate into more specific subtypes going down the tree. This is the usual scenario in object-oriented programming. (We note in passing the similarity to concept and lexical hierarchies such as WordNet where hypernyms and hyponyms refer to more general and more specific categories respectively. In the case where an is-a hierarchy is used for knowledge representation, one could do statistics on leaf nodes and then propagate these statistics upward to empirically characterize the parent categories.)

In contrast, has-a hierarchies formalize **compositional hierarchies**:

If node Child is a child of node Parent, Then Parent has-a subcomponent of type Child.

For example, Computer has-a CPU. This type of semantics is discussed in component-based software engineering, e.g. [Heineman & Councill 2001]. Eventually, the hope is that design and maintenance of large software systems can be simplified through reuse and interconnection of specialized subcomponents; difficulties with the approach include context dependency of subcomponents, inter-component communication protocols, and ontology design of the “space of software objects” so useful fine-grained components can be found easily.

The issue of decomposing software into separate modules is fundamental to software engineering. Since the complexity of programming and debugging software tends to increase more than linearly with the size of the software, decomposition becomes crucial (as our later model demonstrates). Typically, the trick is to **define each module in a “maximally separable” way**, i.e. so that the interface between different modules is as simple as possible. We will later see that a similar criterion holds for defining nodes in hierarchical structures.

Has-a hierarchies are usually less clearly present than is-a hierarchies, since the extent to which an object uses subobjects is partly dependent on our frame of reference. For instance, if A and B are two separate objects such that B occasionally passes vital information to A but they otherwise do not interact, A has-a B in only a loose sense. (Knuth talks about this briefly in [Knuth 1997], in defining subroutines vs coroutines.)

In the absence of a formal definition, perhaps one could think of has-a as a spectrum, with greater degrees when the “mutual information” between A and B is high, and with B being called and instantiated only by A. One might also quantify “emergent behavior” by

considering the complexity of specifying A relative to the complexity of specifying its subcomponents or called modules.

We will largely be using compositional hierarchies in the theory and system we develop. However, there will be aspects of functional hierarchies, e.g. in valuing an object by combining valuations under possible future states (which could be considered “temporal specializations” of the parent object).

In [Baldwin & Clark 1999], the ideas of modularity and subcomponents are explored with respect to the computer industry. The authors suggest that a key factor in the unprecedented growth of computer-related industries was the implicit design principle that **complex products be built from small independently-designed subsystems**, which could be built and tested separately while being composable into a functional whole. They give four principles of modular systems:

1. Create **nested hierarchical structures**.
2. Define **independent components** within an integrated architecture.
3. **Partition design information** into hidden and visible subsets. (In object-oriented programming terms, expose a small API and keep most code private.)
4. Invest in **clear interfaces** and **effective module-level tests**.

These heuristics allowed cognitively-limited humans to collectively transcend their limitations. As they say:

“...Conscious, well-founded divisions of effort and knowledge made it possible for human beings of individually-limited capacity to design and produce ever more complex machines and programs...the resulting modular designs could change in unpredicted, yet coordinated ways. They could be improved via a decentralized, value-seeking process, which we have called ‘design evolution’.”

Graphical Models

Graphical models (also known as **probabilistic networks**) are a class of methods based on graph theory, probability theory, and the mathematics of inference. They are useful for specifying causal links and conditional independence between a large set of variables. The undirected graph version is known as a Markov network or Markov random field; it is an undirected graph (with each node representing a variable) such that the state of each node is independent of all non-adjacent nodes, given the states of all adjacent nodes. Bayesian Networks (or belief networks) are a generalization using directed graphs.

[Pearl 2000] explains both of these formalisms, and explains their use in deciding causality, understanding statistics, and designing computational models of knowledge. Although not necessarily recursively defined, such networks can be highly modular; in fact, this modularity property (which is a result of the conditional independence of any node from most other nodes in the model) is one of the main advantages of using them. Graphical models would likely be defined hierarchically for large problem spaces.

The relevance of Bayesian networks to decomposability is in their **compact representation of joint probability distributions and causal relationships**, and the efficient algorithms that have been developed to perform inference of probabilities in simple classes of such networks (the inference problem for general networks is NP-hard). Bayesian networks provide a framework for reasoning about causal relationships in a mathematical way, in situations too complex to deal with without formal methods.

The **modular** nature of the networks is largely due to the notion of separating subsets of variables. Intuitively,

Given three subsets of variables $\{X, Y, B\}$ in a network,

B separates X from Y if B blocks every path in the network from X to Y

- and in this case X is conditionally independent of Y given B .

Note also that the typical Bayesian network contains a number of directed edges which are a low-order polynomial of the number of nodes, while a fully-general joint PDF would require exponential size to specify in non-functional form. Learning Bayesian networks from data is still very much an open area.

Graphical models have a more general utility, as a common underlying language to **describe many multivariate probabilistic systems**; examples as mentioned in [WebGraphicalModels] include mixture models, factor analysis, hidden Markov models, auto-regressive systems, Kalman filters and Ising models. This category of models represents mathematical processes graphically to aid in comprehension and analysis.

Distributed Computation, Reinforcement Learning, and Hierarchical EC

Distributed computation throws multiple problem-solvers at the same problem. Their co-operation, and different experiences in different parts of the “algorithmic landscape” of the same problem - usually combined with some meta-algorithm, explicit or implicit, to channel their behavior - often lead to more effective and adaptive solution methods. A complex solution method may therefore sometimes be replaced by **multiple interlinked simple solution** methods, with the centralization of interlinking varying on a spectrum from P2P to SIMD parallel machines.

If each agent (i.e. goal-seeking entity) is responsible for solving only part of the overall problem, **functional specialization** allows the emergence of a complex interlinking structure where each agent can focus on what it does best without needing to consider the overall problem state. A well-defined problem usually contains a specified goal to accomplish. If that goal can be split into multiple subgoals, then the subgoals can be solved individually to make progress toward an overall solution.

Similarly, if agents have a particular objective that they are working toward, it may be of use to split this objective into multiple objectives and then use **multiobjective optimization** methods, as surveyed in [Ehrgott & Gandibleux 2002]. This can increase flexibility and make it easier to understand tradeoffs between multiple objectives and constraints; in multilevel problem decomposition, entities at a given level are often distinguished by different subgoals.

Reinforcement learning is a simple yet powerful method for learning how to act in a dynamic environment. The goal of reinforcement learning is to learn a policy, i.e. a strategy defining which actions are taken in response to a given environmental state. Each potential action that can be taken is associated with a specified reward. One key issue in reinforcement learning can then be viewed as learning a value function on {environment state, action} pairs: the value of an action at any point in time is just the **expected long-term reward** if that action is taken. [Sutton & Barto 1998] overview the field with this value function idea in mind.

For large problem spaces, one approach being investigated is Hierarchical Reinforcement Learning, e.g. [Dietterich 2000]. Although many current models assume a near-Markov process and relatively simple problem domains, the area may be applicable to more general problem domains in the future. Note that Hierarchical Reinforcement Learning is an example where the hierarchy learned is one of actions as opposed to static configurations.

Hierarchies have been a topic of interest for some time in Evolutionary Computation, although their effective use is still in its infancy. The idea of ADFs - Automatically Defined Functions - was explored in [Koza 1994] and many works since; the basic idea is to track the evolution of programs to solve a given problem, and **chunk reused modules into coherent higher-level entities**. A more recent approach is the Probabilistic Model-Building Genetic Algorithm, which learns statistics on the search space which are used to guide subsequent optimization. This approach has been extended to hierarchical domains in [Pelikan 2002].

In the related field of Artificial Life, perhaps the place where hierarchy-like ideas have been explored to the greatest extent is in the creation of **multicellular computational organisms**. It has been recognized for some time that modularity (as is present in biological organisms in the form of cells and other such structures) is likely essential for developing complex life-like programs. A recent thesis, [Schmutter 2002], surveys this work in combination with the related mechanism of gene regulation. Regulatory genetic networks can be viewed as a flow hierarchy of genetic functionality (as opposed to a more easily visualizable spatial or object hierarchy); their potential use in evolutionary computation is discussed in [Masum & Oppacher 2001].

Multilevel Models, Lattice Theory, and Factoring

[Raudenbush & Bryk 2002] and [Goldstein 2002] discuss **multilevel statistical models**. This relatively new field of statistical modeling attempts to explicitly model the hierarchical structure of a system, in contrast to previous single-level models. Some basic background and the HLM software can be found online at [WebHLM], from which we quote:

“Behavioral and social data commonly have a nested structure. For example, if repeated observations are collected on a set of individuals and the measurement occasions are not identical for all persons, the multiple observations are properly conceived as nested within persons. Each person might also be nested within some organizational unit such as a school or workplace. These organizational units may in turn be nested within a geographical location such as a community, state, or country. Within the hierarchical linear model, each of the levels in the data structure (e.g., repeated observations within persons, persons within communities, communities within states) is formally represented by its own sub-model. Each sub-model represents the structural relations occurring at that level and the residual variability at that level.”

Lattice theory is a mathematical subfield dealing with **partially ordered sets**, that is related to the study of hierarchies. A partially ordered set (or poset) is simply a set with a specified partial ordering defined on it; this can be visualized as a DAG whose nodes are the set elements and whose directed edges define the ordering.

The meet (or greatest lower bound) of any two elements p and q is the greatest element lying below both of them, if it exists. Similarly, the join (or least upper bound) of p and q is the least element lying above both of them, if it exists. A lattice is then defined as a poset in which every element has both a meet and a join. [Davey & Priestley 2002] provide an introduction to lattice theory and ordered sets with applications.

The idea of **factoring** is ubiquitous in algebra and elsewhere, as a way of decomposition into a representation by a minimal common set of elements. Examples include linear independence, principal components analysis, and singular value decomposition.

Much of statistics deals with parameter estimation: calculating the true value of some property based on assumptions about the generating process and imperfect observations. Statistical tools allow us to use our models in the real world or in complex simulations; sample uses are quantifying observables, extrapolating trends, and estimating confidence levels. For decomposition and modularization, tests for statistical independence are key tools to decide when parts of a problem may be treated independently.

This concludes our survey of relevant streams of thought regarding decomposition and hierarchies. Hierarchical models have been used in many fields, due to their descriptive and computational advantages. While there are a growing number of complementary algorithms and social structures designed to operate in a distributed fashion, the design of useful hierarchies is (and will likely remain) key for many endeavors.

In the next section, we focus on a mathematical model explaining one aspect of the computational advantages of hierarchies.

3. A Mathematical Model

“There is no branch of mathematics, however abstract, which may not someday be applied to the phenomena of the real world.” - Nicolai Lobachevsky

3.1 Hierarchical Decomposition: Optimizing Parts vs Wholes

Goal: Math and a Model

We aim to create mathematical models which capture intuitive and experimental ideas on decomposition, valuation, and recombination. The first model, “Optimizing Parts vs Wholes,” formalizes notions on the desirability of increasing the resolution at which selection operates; the potential speedup from problem decomposition is derived under various assumptions. (We discuss later additional mathematical models which specialize the general discussion in this chapter to our Neartree model.)

Our intended problem domain is decomposition of large problem spaces, with dynamic independent solutions of the components of the problem spaces. For example, many different search strategies may operate in parallel in some large domain, each with their own strategies to solve different components of the domain. We might then like to characterize an overall guiding algorithm which lets strategies proceed independently in components and later recombines these solutions for an overall solution to the original large problem space. How much faster is this strategy than searching the original large problem space directly?

(While the general idea is similar to divide-and-conquer, there are several important differences in our formulation, as we discuss later. A human situation satisfying this model would be a large knowledge space, where each person has different strategies to solve his or her own specialized area, and there are enough generalists around that the niche solutions can be fitted together into an overall solution.)

The Base Case: Multilevel Factorial Search Spaces

Let's consider a mathematical model of optimizing parts of a problem vs optimizing the problem as a whole. This model will illustrate the varying benefits of decomposing a problem under different assumptions. We will make optimistic assumptions and derive upper bounds on the benefits to be gained from problem decomposition, and then explore the relaxation of these assumptions.

Simplifying assumptions in all cases:

- The difficulty of solving any problem instance is directly proportional to the number of potential solutions considered by the algorithm; we will refer to the size of the space searched by the algorithm as the search space, although it may be all or only a part of the total search space. (This subset of the search space which is considered by the algorithm is some superlinear function of the size of the problem input; we will derive different results by varying this function.)
- There is some algorithm for decomposing a problem instance into smaller instances, and then (once the smaller instances have been optimized) recombining the small solutions into a solution for the original larger problem.

Although this model is applicable to any system fulfilling the relatively few assumptions given above, it may help in understanding the mathematics to keep a specific problem in mind. Any hierarchically decomposable problem will do; for example, exact clustering or a fractal TSP (defined later), where the problem would be near-exponential if we just tried all possibilities without considering or perceiving the intermediate higher-order patterns.

Note that the goal of our decomposition will be to split an overall problem into subproblems, while simultaneously equalizing the sizes of the subproblems. (As we show, this approach is generally best where the solution time grows superlinearly with problem size.) In Figure 1, we graphically depict the general idea - the overall problem is the outer

rectangle, sub-problems are triangles, and sub-sub-problems are the small circles.

Figure 1...Recursive subdivision of a problem

Of course, decomposition into subproblems assumes that the problem has multiple scales at which finding solutions makes sense; however, the assumption is somewhat weaker than in the divide-and-conquer case, where one recursively splits a problem into typically a small constant number of smaller subproblems, all the way down to a small base case.

Case Fact1: Factorial Search Space, 1 level

Let's look now at a case based on the TSP, in the case where **the search space is factorial in the number of nodes**. For an n -node TSP, there are $n!$ permutations of the nodes to consider; denoting the number of cases in this 1-level problem by FactNum1, we have :

$$\begin{aligned} \text{FactNum1} &= n! \\ &\approx (n/e)^n (2\pi n)^{1/2} \quad (\text{by Stirling's approximation for } n!) \end{aligned}$$

$$\begin{aligned}
\text{So } \ln(\text{FactNum1}) &\cong n \ln n - n \ln e + 1/2 \ln n \\
&\cong n (\ln n - 1) \quad (\text{asymptotically}) \\
&\cong n \ln n
\end{aligned}$$

(We've taken logarithms to facilitate comparison between otherwise unwieldy quantities, on an order-of-magnitude basis - which is sufficient to determine the solution up to a constant factor.) Therefore our measure of the problem difficulty turns out to be about $n \ln n$.

Recall that we've assumed that problem difficulty is proportional to the size of the subset of the search space which is searched by the algorithm - and this size is equal to FactNum1 in this case. Since we'll be concerned with measuring relative sizes of search spaces under differing assumptions, it suffices to compare logarithms of the quantities of interest.

Case Fact2: Factorial Search Space, 2 level

Now decompose the graph into smaller clusters. How many? In the absence of any other information, let's set the number of clusters equal to the size of each cluster; this gives $n^{1/2}$ clusters, each containing $n^{1/2}$ nodes. (Note that this is a huge number of clusters relative to typical divide-and-conquer constructions, which usually assume a fixed and small constant number of divisions at each step.)

We're now going to see how many cases we would need to consider, under the previously-mentioned simplifying assumption that we can optimize each cluster separately - and then do one more optimization treating each optimized cluster as a node. Calling the number of cases in this 2-level problem FactNum2, we get:

$$\begin{aligned}
\text{FactNum2} &= (n^{1/2}) (n^{1/2})! + (n^{1/2})! \quad (\text{i.e. } n^{1/2} \text{ small and 1 big cluster}) \\
&= (1 + n^{1/2}) (n^{1/2})!
\end{aligned}$$

$$\begin{aligned}
\text{And } \ln(\text{FactNum2}) &= \ln((1 + n^{1/2}) (n^{1/2})!) \\
&= \ln(1 + n^{1/2}) + \ln((n^{1/2})!) \\
&\cong 1/2 \ln n + (n^{1/2}) (\ln(n^{1/2}) - 1) \text{ (Stirling's again)} \\
&= 1/2 \ln n + 1/2 (n^{1/2}) \ln n - (n^{1/2}) \\
&\cong (n^{1/2}) (1/2 \ln n - 1) \\
&\cong 1/2 (n^{1/2}) \ln n
\end{aligned}$$

For comparison, recall that $\ln(\text{FactNum1})$ is approximately $n \ln n$.

Now we can use our calculated difficulty measure to get a sense of how much effort we save by using a 2-level problem formulation. In particular, what we would like to know is this: if we can solve an n -node problem with a 1-step method, how much more can we solve with a 2-step method? To approximate this, we fix N as the 1-step problem size in FactNum1 , and then solve for our unknown number of nodes k in FactNum2 in terms of N . Setting the logs of FactNum1 and FactNum2 equal (using the logs instead of the actual case numbers is more convenient and still correct up to an approximation factor), we get:

$$1/2 (k^{1/2}) \ln k = N \ln N$$

By intuition we guess that the solution is $k = N^2$, and verify through substitution:

$$\begin{aligned}
1/2 ((N^2)^{1/2}) \ln (N^2) &= 1/2 N^2 \ln N \\
&= N \ln N
\end{aligned}$$

And so we find that in Case2 we can solve quadratically larger problems than in Case1. To make matters concrete, for a graph problem like the TSP, the problem difficulty measure for a 1000-node problem in Case1 is equivalent to that for a 1000000-node problem in Case2.

Case FactD: Factorial Search Space, D level

Let's consider now what happens in Case3, Case4, ..., CaseD (for Depth), by deriving a **general formula for the performance gain for a D-level algorithm**. In this case, the n nodes are first partitioned into lowest level clusters containing $n^{1/D}$ nodes; there are then D levels of aggregation, in each of which $n^{1/D}$ clusters of a given level are aggregated to form a cluster of the next highest level.

Using the same factorial function to get the search space size for each cluster, we get:

$$\begin{aligned} \text{FactNumD} &= (n^{(D-1)/D}) (n^{1/D})! + (n^{(D-2)/D}) (n^{1/D})! + \dots + (n^{1/D}) (n^{1/D})! + (n^{1/D})! \\ &= (1 + n^{1/D} + n^{2/D} + \dots + n^{(D-1)/D}) (n^{1/D})! \\ &= (1 + n^{1/D} + (n^{1/D})^2 + \dots + (n^{1/D})^{(D-1)}) (n^{1/D})! \end{aligned}$$

Let's put an upper bound on the size of the first term. Set $y = n^{1/D}$, and note that by our model assumptions $y \geq 2$ (since the smallest size cluster certainly contains at least two elements):

$$\begin{aligned} \text{So first term} &\leq 1 + y + \dots + y^{D-2} + y^{D-1} \\ &\leq 2^{-(D-1)} y^{D-1} + 2^{-(D-2)} y^{D-1} + \dots + 2^{-1} y^{D-1} + y^{D-1} \\ &< 2 y^{D-1} \end{aligned}$$

Using this inequality in the previous expression for FactNumD, we get:

$$\begin{aligned} \text{FactNumD} &< 2 (n^{(D-1)/D}) (n^{1/D})! \\ &\cong (n^{(D-1)/D}) (n^{1/D})! \quad (\text{we don't care about constant factors}) \end{aligned}$$

$$\begin{aligned} \ln(\text{FactNumD}) &\cong \ln((n^{(D-1)/D}) (n^{1/D})!) \\ &\cong (D-1) \ln n^{1/D} + (n^{1/D} + 1/2) \ln n^{1/D} - n^{1/D} \end{aligned}$$

$$\begin{aligned}
&= ((D - 1 + n^{1/D} + 1/2)/D) \ln n - n^{1/D} \\
&\cong (n^{1/D}/D) \ln n - n^{1/D} \\
&= n^{1/D} ((\ln n)/D - 1) \\
&\cong (1/D) n^{1/D} \ln n
\end{aligned}$$

As with FactNum2, we now set the logs of FactNumD and FactNum1 equal, fixing the problem size in FactNum1 as N and solving for the equivalent problem size k in FactNumD:

$$(1/D) k^{1/D} \ln k = N \ln N$$

Now try $k = N^D$:

$$\begin{aligned}
(1/D) (N^D)^{1/D} \ln N^D &= (1/D) N^D \ln N \\
&= N \ln N
\end{aligned}$$

Our educated guess works, and so we have derived a general formula for the D-level case. Given computational resources sufficient to solve an N-node case using 1 level, we can with those same computational resources solve an N^D -node case using D levels.

Note that since the cluster size is at least two, $D \geq \lg N$, and therefore in terms of N the best possible speedup is a factor of $N^{\lg N}$. These are generous upper bounds, showing a large potential speedup from problem decomposition.

Adding Overhead and Exponential Search Spaces

Realistically speaking, the problem decomposition and recombination steps will themselves take some time to do. Let's incorporate this overhead into our general formula for the D-level case.

Case FactDPoly: Factorial Search Space, D level, Poly Overhead

A realistic generalization would be that **decomposition and recombination takes time polynomial in the number of clusters**. Recall our previous formula:

$$\text{FactNumD} = (1 + n^{1/D} + n^{2/D} + \dots + n^{(D-1)/D}) (n^{1/D})!$$

Here the first term in the product at right counts the number of clusters at all levels, while the second term represents the amount of work per cluster. To incorporate a polynomial amount of overhead, we need to add another term where the “overhead polynomial” $f()$ is applied to each of the summands in the first term:

$$\text{TotalD} = \text{FactNumD} + \text{OverheadD}$$

$$\text{OverheadD} = (f(1) + f(n^{1/D}) + f(n^{2/D}) + \dots + f(n^{(D-1)/D}))$$

(Note that, since all clusters have the same number of next-lower-level objects by assumption, we’re assuming that OverheadD increases as the number of clusters at a given level increases, but is independent of the level to which clusters belong.) Now, the overhead only becomes significant if OverheadD is not $o(\text{FactNumD})$. By comparing OverheadD and FactNumD, we see that if the overhead polynomial $f()$ is linear, the largest summands in the two terms are $c(n^{(D-1)/D})$ and $n^{(D-1)/D} (n^{1/D})!$ respectively, where c is constant. Therefore, linear overhead in decomposition and recombination does not change the previous results.

How about if $f()$ is a polynomial of degree greater than one? In this case, let f be the degree of $f()$. Going through a similar sequence of steps as our calculation of FactNumD, we get:

$$\begin{aligned} \text{OverheadD} &\cong (1 + n^{f/D} + n^{2f/D} + \dots + n^{f(D-1)/D}) \\ &= (1 + n^{f/D} + (n^{f/D})^2 + \dots + (n^{f/D})^{(D-1)}) \end{aligned}$$

For an upper bound, set

$$\begin{aligned}
 y &= n^{f/D} \\
 &= (n^{1/d})^f \\
 (n^{1/d})^f &\geq 2^f > 2 \quad (\text{since } n^{1/d} \geq 2 \text{ and } f > 1)
 \end{aligned}$$

and so using the same inequality as in FactNumD we get:

$$\begin{aligned}
 1 + y + \dots + y^{D-2} + y^{D-1} &< 2 y^{D-1}, \text{ and so} \\
 \text{OverheadD} &\leq (n^{f/D})^{(D-1)} \\
 \ln(\text{TotalD}) &= \ln(\text{FactNumD} + \text{OverheadD}) \\
 &\leq \ln((n^{(D-1)/D}) (n^{1/D})! + (n^{f/D})^{(D-1)}) \\
 &= \ln((n^{(D-1)/D}) ((n^{1/D})! + n^f))
 \end{aligned}$$

Comparing with the expression for $\ln(\text{FactNumD})$, we see that the expression is identical except for the additional factor of n^f on the right. Therefore, TotalD will become appreciably larger than FactNumD when $n^f \geq (n^{1/D})!$; treating D and n as fixed, we can solve for f as follows:

$$\begin{aligned}
 n^f &= (n^{1/D})! \\
 \log_n n^f &= \log_n (n^{1/D})! \\
 &\approx (n^{1/D} + 1/2) \log_n n^{1/D} - n^{1/D} \log_n e \\
 &= (n^{1/D} + 1/2) 1/D - n^{1/D} \log_n e \\
 &\approx 1/D n^{1/D} \quad (\text{assuming } 1/D \gg \log_n e)
 \end{aligned}$$

So the overhead becomes significant when the degree of the overhead polynomial $f \geq 1/D n^{1/D}$.

These are generous bounds for overhead; any P-time algorithm where f is less than the size of the smallest cluster divided by the number of levels will do. As an example, consider a 10^{12} node network being solved with a 6-level algorithm; in this case the overhead polynomial can have any degree less than $10^{12/6}/4 = 25$. (However, we'll soon see that the bound is less with more realistic bounds on algorithm capabilities.)

Case ExpD: Exponential Search Space, D level

Now let's consider a case where the **search space is “only” exponential** in the size of the generating set, i.e. instead of a problem instance of size n generating a search space of size $n!$, suppose it generates a search space of size only c^n with c a constant. (This is quite typical of many problems that need to search through all possible subsets of a set.) How does this change the results above?

In this case, the overhead-free running time for a D-level algorithm can be calculated as follows:

$$\begin{aligned}
 \text{ExpNumD} &= (n^{(D-1)/D}) c^{n^{1/D}} + (n^{(D-2)/D}) c^{n^{1/D}} + \dots + (n^{1/D}) c^{n^{1/D}} \\
 &\quad + c^{n^{1/D}} \\
 &= (1 + n^{1/D} + n^{2/D} + \dots + n^{(D-1)/D}) c^{n^{1/D}} \\
 &= (1 + n^{1/D} + (n^{1/D})^2 + \dots + (n^{1/D})^{(D-1)}) c^{n^{1/D}} \\
 &< 2 (n^{1/D})^{(D-1)} c^{n^{1/D}}
 \end{aligned}$$

Contrast this with ExpNum1, which by assumption is simply c^n .

As before, we now fix c , D , N , and let k be a variable number of nodes to solve for that will make ExpNumD equal to ExpNum1:

$$c^N \cong (k^{1/D})^{(D-1)} c^{k^{1/D}} \quad (\text{dropping the constant } 2)$$

$$\begin{aligned}
N &\approx ((D-1)/D) \log_c k + k^{1/D} \\
&\approx \log_c k + k^{1/D}
\end{aligned}$$

This requires a bit of analysis to interpret. If on the right the first term is o (the second term), then the solution $k = N^D$ works and this case behaves the same as the previous case where the search space was given by the larger factorial function instead of an exponential function. But when will the first term be small relative to the second term? Well, we can take c to be fixed for any given problem, and then the relative sizes depend on the value of D that we choose; the constraint $D \leq \lg k$ implies that the second term can have least value $k^{1/\lg k}$.

If we keep increasing D to keep the cluster size constant with increasing problem sizes, then eventually the first term (the logarithm of the problem size) will become larger than the second. However, for small values of c this will not happen for quite a while, and if the cluster sizes increase logarithmically it will not happen at all. Essentially, if the cluster size is larger than the log of the problem size, we will get the nice behavior as in the factorial space where we can solve an N^D size problem in the same time as the 1-level algorithm can solve an N size problem.

Case ExpDPoly: Exponential Search Space, D level, Poly Overhead

Now we consider the exponential case with polynomial overhead. Analogously to the FactDPoly case, we can write:

$$\begin{aligned}
\text{TotalD} &= \text{ExpNumD} + \text{OverheadD} \\
\text{ExpNumD} &\approx 2 (n^{1/D})^{(D-1)} c^{n^{1/D}} \\
\text{OverheadD} &= (f(1) + f(n^{1/D}) + f(n^{2/D}) + \dots + f(n^{(D-1)/D})) \\
&\approx (n^{f/D})^{(D-1)}
\end{aligned}$$

$$\ln(\text{TotalD}) = \ln(\text{ExpNumD} + \text{OverheadD})$$

$$\begin{aligned}
&\cong \ln((n^{1/D})^{(D-1)} c^{n^{1/D}} + (n^{f/D})^{(D-1)}) \\
&= \ln((n^{(D-1)/D}) (c^{n^{1/D}} + n^f))
\end{aligned}$$

TotalD will become appreciably larger than ExpNumD when $n^f \geq c^{n^{1/D}}$; treating D and n as fixed, we can solve for f as follows:

$$\begin{aligned}
n^f &= c^{n^{1/D}} \\
\log_c n^f &= \log_c (c^{n^{1/D}}) \\
f \log_c n &= n^{1/D}
\end{aligned}$$

So the overhead becomes significant when the degree of the overhead polynomial $f \geq n^{1/D} / \log_c n$. This is a greater restriction than for the FactDPoly case: consider the same example of a 10^{12} node network being solved with a 6-level algorithm, with the constant c set equal to 2 for simplicity. In this case the overhead polynomial can have any degree less than $10^{12/6} / (12 * \lg 10) \approx 10^2 / (12 * 3.322) \approx 2.5$. The logarithmic factor in the denominator will be significant for many feasible values of n and D.

Polynomial Search Spaces and Beyond

Finally, we consider a more realistic assumption where problem instance solution difficulty is not directly proportional to the size of the search space for that instance. In particular, it may be possible to find an “acceptable” solution by using effort only $o(\text{search space size})$; this is in fact the assumption behind most heuristic methods.

Therefore, it would be of interest to experiment with different assumptions of the amount of the total search space size that needs to be searched (e.g. logarithmic, polylogarithmic, sublinear polynomial). We’ll look specifically at the case where the algorithmic effort required is a polylogarithmic function of an exponentially-sized total search space, i.e. where the algorithm’s **search space is polynomial** in the problem size (as measured by e.g. number

of nodes in a graph).

Let's call the polynomial degree of the algorithm p . Then we can go through the same process as before and calculate the speedup that we get from a D -level algorithm, both with and without overhead.

Case PolyD: Polynomial Search Space, D level

In the first case, we have no overhead, and so we get:

$$\begin{aligned}
 \text{PolyNumD} &= (n^{(D-1)/D}) (n^{1/D})^p + (n^{(D-2)/D}) (n^{1/D})^p + \dots + (n^{1/D}) (n^{1/D})^p + (n^{1/D})^p \\
 &= (1 + n^{1/D} + n^{2/D} + \dots + n^{(D-1)/D}) (n^{1/D})^p \\
 &= (1 + n^{1/D} + (n^{1/D})^2 + \dots + (n^{1/D})^{(D-1)}) (n^{1/D})^p \\
 &< 2 (n^{1/D})^{(D-1)} (n^{1/D})^p \\
 &= 2 n^{(D-1+p)/D}
 \end{aligned}$$

(Notice the difference in functional form from previous cases.) Setting PolyNum1 equal to PolyNumD, we get:

$$\begin{aligned}
 N^p &\cong k^{(D-1+p)/D} \\
 k &\cong N^{pD/(p+D-1)}
 \end{aligned}$$

To analyze this expression and find the qualitative behavior of k , we consider the exponent $pD/(p+D-1)$ on the right:

- If $p \gg D$, then the exponent is approximately equal to $pD/p = D$ and $k \approx N^D$, just as with the previous exponential search space cases FACT and EXP.
- If $p = D$, the exponent is $p^2/(2p-1) \approx p/2 = D/2$, so in this case the speedup is substantially less.

- Finally, if $D \gg p$, the exponent approaches $Dp/D = p$ from below as D increases, and the speedup is only equal to the degree of the polynomial p . (D does not have to be that much greater than p , e.g. if $D = 10p$ then the speedup will be around $0.9p$, within 10% of optimum.)

So if our algorithm finds an “acceptable” solution in low-degree polynomial time, then no matter how many levels D we use we can’t speed up by more than the (low) degree of this polynomial. This confirms our intuition that the hierarchical decompose-and-recombine paradigm is of limited benefit where a “good-enough” P -time approximation algorithm already exists.

Case PolyDPoly: Polynomial Search Space, D level, Poly Overhead

In the last case we have a polynomial approximation algorithm with polynomial decomposition overhead.

$$\begin{aligned}
 \text{TotalD} &= \text{PolyNumD} + \text{OverheadD} \\
 \text{PolyNumD} &\cong n^{(D-1+p)/D} \\
 \text{OverheadD} &= (f(1) + f(n^{1/D}) + f(n^{2/D}) + \dots + f(n^{(D-1)/D})) \\
 &\cong (n^{f/D})^{(D-1)} + (n^{f/D})^{(D-2)} + \dots \quad (\text{using just 1st dominating term of } f) \\
 &\cong (n^{f/D})^D / (n^{f/D}) \\
 &= n^{f(D-1)/D}
 \end{aligned}$$

Comparing exponents, we see that the overhead dominates when:

$$\begin{aligned}
 f(D-1) &> D-1+p, \text{ or} \\
 f &> 1 + p/(D-1)
 \end{aligned}$$

From the analysis of case PolyD, there is no point in choosing D more than $c * p$, where c

is a fairly small constant factor (say 10 for concreteness). We see then that there is a tradeoff with choosing $D \approx p$ being a dividing line: if $D < p$, the overhead can be quadratic or greater, while $D > p$ means that the overhead should not too far from linear.

3.2 Extensions and Consequences

Recurrence Equations and Extensions to the Model

There are many real-life cases where we might not be able to form a multilevel hierarchy, but might still be able to partition the problem space into **multiple independent subproblems**. In this case, a P -part problem decomposition with no overhead would take time $P \cdot \text{Time}(N/P)$, where $\text{Time}()$ is the computation time to find an acceptable solution in terms of the size of the problem.

This is similar to the usual divide-and-conquer paradigm for algorithms, where a problem instance is recursively split into smaller instances. In e.g. the classic reference [Cormen et al 2001], a solution is shown in the “Master Theorem” for most cases of the recurrence:

$$\text{Time}(N) = a \cdot \text{Time}(N/P) + \text{Recomb}(N)$$

Here $\text{Time}(k)$ is the time to solve a problem of size k , where k is not search space size but instance specification size, i.e. the number of nodes in a TSP. $\text{Recomb}(k)$ is the time to partition and recombine the subproblems.

(Note that all the computation at non-leaf nodes of the recursion tree is accounted for by this factor. One difference between this case and the case we were discussing in the previous section is that, in our previous case, there are computations involving $\text{Time}()$ at all levels of the recursion tree. In the previous case, we do not just reduce a large problem to a set of small problems of the same type, but also recombine these problems using additional

problems of the same type, along perhaps with additional overhead factors.)

The three cases covered are:

- If $\text{Recomb}(N) = O(N^{\log_p a - \epsilon})$, then $\text{Time}(N) = \Theta(N^{\log_p a})$.
- If $\text{Recomb}(N) = \Theta(N^{\log_p a})$, then $\text{Time}(N) = \Theta(N^{\log_p a} \lg N)$.
- If $\text{Recomb}(N) = \Omega(N^{\log_p a + \epsilon})$, and $a \text{Recomb}(N/P) \leq c \text{Recomb}(N)$ (for some constant c and sufficiently large N), then $\text{Time}(N) = \Theta(\text{Recomb}(N))$.

So when framed in this way, the solution time is at least $N^{\log_p a}$ (and sometimes $\text{Recomb}(N)$ if recombination takes longer than solving each problem). Note that another difference from the previous section is that this formulation applies only to polynomial-time solutions for $\text{Time}(N)$; if $\text{Recomb}(N)$ is small enough to be in the first case mentioned above, then $\text{Time}(N) = \Theta(N^k)$ implies $a = P^k$ in the recurrence equation for $\text{Time}(N)$.

We note that the overhead-free case discussed earlier can also be represented as a recurrence equation. Letting $T(n, D)$ be the time to solve an instance of size n using D levels, we have:

$$T(n, D) = n^{1/D} T(n^{(D-1)/D}, D-1) + T(n^{1/D}, 1)$$

The idea is that to solve a problem of size n , we split it into $n^{1/D}$ problems at the next lower level, solve each of these, and then combine the solutions.

We can solve this recurrence more easily by making a variable substitution, letting $b = n^{1/D}$:

$$\begin{aligned} T(b^D, D) &= b T(b^{D-1}, D-1) + T(b, 1) \\ &= T(b, 1) + b T(b^{D-1}, D-1) \end{aligned}$$

$$\begin{aligned}
&= T(b, 1) + b (T(b, 1) + b T(b^{D-2}, D-2)) \\
&= \dots \\
&= T(b, 1) + b T(b, 1) + b^2 T(b, 1) + \dots + b^{D-1} T(b, 1) \\
&= T(b, 1) (1 + b + b^2 + \dots + b^{D-1}) \\
&= T(b, 1) (b^D - 1) / (b - 1) \\
&= T(n^{1/D}, 1) (n - 1) / (n^{1/D} - 1)
\end{aligned}$$

which gives additional insight into the relationship between a single-level solution and a D-level solution. (Note that, although this recurrence formulation is easier in this case than the previously-used method of explicit calculation and comparison of times for each level, the previous method is more general and easily extends to the case where solution times are heterogeneous across levels or nodes.)

We note some additional differences between hierarchical decomposition and divide-and-conquer. D&C usually has a very fine granularity of base-case problem size; hierarchical decomposition may have a natural cluster or subproblem size, and a natural number of levels. In divide-and-conquer the recursion tree should be followed to the bottom; in hierarchical decomposition, the problem can still make sense even if we ignore some number of bottom levels, since we then just have a less-detailed version of the problem.

Extensions to the model suggest themselves:

- Problem decomposition may not be balanced, so that some subproblems are larger than others. The general consequence of this will be to reduce the efficacy of the method, since the solution time is superlinear in the size of each problem. Similarly, there could be uncertainty about time required; cascading uncertainties imply wide performance ranges.
- Each subproblem solution could have a different value (either known or unknown) if solved; this would constitute a weighted form of the basic problem.

- Real problems often have natural levels of granularity; often there is a grain size below which splitting isn't possible without the complexity of between-parts interactions overwhelming the complexity of within-parts interactions.
- Do there exist complexity classes for hierarchical problems, analogous to P and NP?
- Is there a relationship between hierarchical decomposition and problem spaces that are “near-greedy” (e.g. matroids)?

Going from the model to the generation of hierarchical search methods will require modeling search methods (using e.g. Monte Carlo Markov Chain, as discussed in [Aldous & Propp 1998]), to see how different assumptions on local search effectiveness affect the expected amount of solution space that needs to be sampled to get an “acceptable” solution.

Applicability, and Extracting the Neartree Heuristic

Let's consider where this recursive decomposition mathematics could be applied. Essentially, the problem must have these properties:

- The computational complexity of the problem increases rapidly with problem size, i.e. more than low-order polynomial. (Otherwise we could just solve the overall problem directly.)
- There should be a reasonable way to recursively subdivide the problem into “independent components”, such that each can be optimized separately.
- There should be a reasonable way to recombine the subsolutions into an overall solution that is “good enough”, i.e. that meets the required approximation bounds.

An example that will be explored in detail later is “hierarchical SAT”: a hierarchical layering of variables, each of which has a truth value that can be estimated from a function of at most k lower-level variables. Other potential problem classes include high-dimensional clustering, “fractal TSP”, information organization, and multilevel spatial analysis and simulation. Two good sources for NP-hard approximation problems and algorithms are

[Hochbaum 1997] and [Ausiello et al 1999].

To apply the method, one needs to create a heuristic divide-and-conquer algorithm which follows this paradigm and performs well for problem classes of interest. It will be of interest to compare the theoretical upper bound given by mathematical models like the one in this chapter to actual performance of heuristic divide-and-conquer algorithms.

As we have demonstrated, divide-and-conquer decomposition is a powerful heuristic to subdivide and solve large problems. What makes the method powerful is the restriction at each level to considering just a few lower-level components. This can be generalized past trees: consider multilevel recursive networks.

Let's represent the tree implicitly generated by divide and conquer as a directed tree, with all links pointing toward the root. Each link shows information flow from a node to an upper-level node; nodes with multiple incoming links combine lower-level results and pass the combined result up to the next level. (We use the common convention that the root node is on top of the tree, and child nodes are below their parent.)

A first thought on generalizing is to bound the number of upward links from every node, i.e. the number of links from any node to its parents. Note that this parameter has a value of 1 for the original divide-and-conquer tree. This directed tree generalization has the benefit of bounding the number of upper-level nodes for which a given lower-level node needs to be consulted.

For a hierarchical problem, this sort of bounding is important, since otherwise it becomes more and more difficult to predict the effect of changing a lower-level node. A second parameter can bound the number of downward links, i.e. the number of nodes which need to be consulted to construct each upper-level node.

We will call this directed tree generalization a **Neartree**. Neartrees with low values for

the two specified parameters intuitively seem easy to understand, and nicely solvable with a dynamic programming style of approach. We show evidence to support this later.

4. Neartrees: Generalizing Hierarchies

"A mind stretched to a new idea never goes back to its original dimension."

- Oliver Wendell Holmes

4.1 The Algorithmic Framework: Hierarchical Neartrees

Specification and Data Structures

The basic Neartree can be pictured as a **tree with the parental restriction relaxed** and a restriction on children imposed. (Recall that in an ordinary tree, each node can have an arbitrarily high number of child nodes, and each node aside from the root has exactly one parent node.)

We define several sets of Neartrees. The most specific set is the (c,p)-Nearthree, where

Figure 2...A Neartree

each node can have at most c child nodes, and each node (aside from the root) can have up to p parent nodes. Hence a $(c,1)$ -Neartree is simply an ordinary tree with the fanout at each node bounded by c , and a $(*,1)$ -Neartree is an ordinary tree with no fanout bound. A $(*,p)$ -Neartree is a strict generalization of a tree.

We may speak of a c -Neartree, as shorthand for a $(c,*)$ -Neartree, which has no limit on the number of parent nodes a given node may have. (Note that this set is not strictly a superset of the tree class, due to the c bound. However, for any set of trees with bounded fanout c , a c -Neartree is a strict superset.)

The key elements of a Neartree are **Nodes**. Nodes have a state which define their current configuration, and a value which defines their contribution to some user-specified function of the overall Neartree. Each node contains a set of objects and properties, as described later; these determine how the node's state and value varies as a function of its children's states and values. (While the Neartree itself treats nodes as relatively opaque objects - they communicate and are described via relatively little information compared to their internal structure - an external system using a Neartree could have each node representing an arbitrarily complex system, as long as each node follows the interface and semantics we define).

The **nodes are arranged in successive layers**, with a designated root node in the first layer. Some nodes in adjacent layers have a child-parent relationship, implying both an information transmission pathway and compositional relationship between the node pair (i.e. parents and children communicate, and a parent is "made up of" its children - though it may be more than the sum of its parts). In a (c,p) -hNeartree, each node can have at most c child nodes, and may influence up to p nodes at the next level above.

We will use a shorthand notation to refer to Neartrees, the levels in a Neartree, and the nodes in a level. Every node has a label of the form $_{\text{LevelNumber}}\mathbf{H}_{\text{NodeNumber}}$ so e.g. $_1\mathbf{H}_2$ denotes the node with ID 2 (which is the third node, since numbering starts at 0) in level 1 in a

Neartree.

(Note that the notation we have developed can equally well refer to a strict hierarchy, i.e. tree - hence the H notation may be useful when discussing hierarchies in general.)

Table 1...Neartree Notation	
Notation	Meaning
H	A particular Neartree under discussion will be referred to by H - this is meant to mnemonically suggest “hierarchy”.
${}_jH$	The root node is level ${}_0H$, and each successive level ${}_1H$, ${}_2H$, ... is the set of nodes which are the child of some node at the previous level. (We restrict all paths from the root to any child node to have equal length, so a node can only link to other nodes which are one level above or below. There is then no ambiguity about the level to which a node belongs.)
$ {}_jH $	Size of level ${}_jH$, i.e. the number of nodes in ${}_jH$.
${}_{i..j}H$	The set of Neartree levels from ${}_iH$ to ${}_jH$ inclusive.
L	Maximum level number of a Neartree. The Neartree levels then range from ${}_0H$ to ${}_LH$.
${}_jH_k$	Node k within a given level ${}_jH$ will be referred to as ${}_jH_k$. Hence level j consists of the nodes ${}_jH_0$, ${}_jH_1$,
${}_0H_0$	Distinguished root node .
${}^+_jH_k$	The set of children of node ${}_jH_k$. (As a mnemonic, note that the “+” for children corresponds to increasing a level number by 1, e.g. ${}^+_0H_0 = {}_1H$.)
${}^-_jH_k$	The set of parents of node ${}_jH_k$. If we want to refer to ancestors or descendants more than one level away, the number of levels can be placed after the + or - sign; e.g. ${}^{-2}_2H_k = {}_0H_0$ (for any valid k).
$@({}_jH_k)$	The state of node ${}_jH_k$. Mnemonic - the state ${}_jH_k$ is @ (at).
$?@({}_jH_k)$	The goal state of node ${}_jH_k$. (What state does ${}_jH_k$ want to be at?)
$\$({}_jH_k)$	Value of node ${}_jH_k$. (Problem-dependent - see hSAT chapters.)

Several other components of a Neartree:

- **Value** (evaluational algorithm). A method for judging child nodes on one or more criteria. Each node judges only its immediate children, to preserve computational tractability. The criteria should therefore be “transitive” in the sense that judgements can meaningfully be aggregated and passed up through the Neartree.
- **Combiner**. Each node has a combiner, which uses some combinational algorithm to combine incoming information from inbound child links. This method can range in complexity from simple weighted linear combinations to a complex adaptive algorithm. Generally, the combinational algorithm will be the same at most nodes; however, it could have differing node-specific parameter values, or (in the most general case) different algorithmic form at different node levels or nodes.
- **Nested Hierarchy**. In each (c,p)-hNearthree each lower level is a partition of the level above it. If $p=1$, this generates a traditional hierarchy; if $p>1$, this generates a more general structure with upper level nodes composed of overlapping combinations of lower-level nodes, as in many information taxonomies. (Note that this partitioning restriction is a subset of the general non-hierarchical Neartree case.)

Discussion and Examples of Hierarchical Neartrees

The hNearthree is essentially a representation of a hierarchy (or near-hierarchy for a (c,p)-hNearthree with $p > 1$), in which **each lower-level node is “part of” its parent node**. More precisely, the lower-level nodes along with the “combinational algorithm” of the upper-level node constitute a good representation of the upper-level node, or good solution to the problem implicit in the upper-level node.

This is a crucial point. It means that, starting from the root, each successive level of the hNearthree is a successively more detailed representation of the problem space. Typically we can ignore lower levels if we want to get a quick and approximate understanding of the problem, and then consider them as our analytical and computational resources increase. Note that these lower levels could be “more detailed” in many semantic senses: a finer

spatial resolution, a larger set of potential future states in time, or a finer partitioning of some abstract space.

Note that along with the parameters c and p , there are several other quantities to measure that affect the properties of the hNeartree: size of combinational algorithm, size of total incoming information, possible size of any “extra” information from the environment (e.g. random bits for a partially-randomized algorithm), fidelity of representation of upper-level node in terms of lower-level node, and so on.

One important advantage of a (c,p) -hNeartree is that it gives a human-comprehensible decomposition of a problem space - since hierarchies are natural and familiar in many everyday situations, framing a problem as a Neartree gives a natural structure. Neartrees are also designed to be cognitively simple in the sense that any particular node is only decomposed into at most c lower-level nodes, and is only itself used by at most p upper-level nodes.

Sample problems:

- General hierarchical clustering. Nodes represent clusters, links represent inclusion.
- Graph optimization problems (i.e. hierarchical TSP, weighted graph clustering).
- Information taxonomies (nodes represent categories).
- Multilevel spatial algorithms, where nodes represent a region of space and the subnodes are a partition of their parent node (e.g. quadtrees, octrees).

Hierarchical Neartrees and Flow Neartrees

We can distinguish two contrasting formulations of Neartrees: hierarchical Neartrees (“hNeartrees”) and Flow Neartrees. Both use the same underlying data structure defining inter-node connectivity, of a tree-like object with multiple parents allowed; they are distinguished by different semantics on the relationship of a node to its child and parent

nodes. Unless otherwise specified, we will be focusing on hierarchical Neartrees through the rest of the thesis (and will use “Nearthree” to imply a hierarchical Neartree). However, the idea of a Flow Neartree is important for both conceptual clarity and future work.

A Flow Neartree is simply a hierarchical Neartree with the hierarchy condition removed. This is what we referred to as a non-nested hierarchy previously, where upper-level nodes are not composed of their children but rather are merely functionally related to them. In an hNearthree, each node is a sort of coarser-resolution view of its children, and is specified by the incoming information from its links, along with its Combiner. However, in the more general Neartree case, additional exogenous information is needed to specify the node. (We use “Flow” to match our earlier terminology for hierarchies. Flow Neartrees could also be thought of as POV-Nearthrees, where POV stands for “Point of View” - emphasizing that such a Neartree is both a representation of information streams converging on an agent, and created according to the usually fallible information of a single entity.)

For instance, if the Neartree represents a decision problem where each node is a human, then a particular node will make decisions based not only on its “incoming information from others” and its analysis algorithm of that incoming information, but also on other information particular to that node. (A specific example of such a case would be command or organizational hierarchies, where a node is related to but clearly contains much more than its children.) In contrast, in an hNearthree case such as clustering, once we have received the incoming information and processed it with the combinational algorithm, we have completely specified the upper-level node.

In a Flow Neartree, each lower-level node is a “causative factor” for the parent node. The parent node contains extrinsic information exclusive of the incoming information and combinational algorithm; the size of this extrinsic information as compared to the incoming information may be a useful measure of how far the Neartree departs from being an hNearthree.

Sample problems:

- The single-POV “human decision problem”: I decide on an action, based on input from a few sources, each of which in turn get input from a few (possibly overlapping) sources, and so on. (In contrast to most NP-hard problems, it is nontrivial to formalize this into a testable algorithmic problem and to get real-world instance data.)
- Information networks, where subnodes are relevant to their parent nodes but do not necessarily compose them (e.g. spreading activation networks of concepts).
- Many graphical models and Bayesian networks.
- Power structures and command hierarchies.

Similar Previous Work

There are some previous data structures which bear a resemblance to Neartrees. However, none is quite the same:

- A balanced tree. An example would be a subdivision of a spatial region or future time periods by equal-size increments; a uniform level of detail implies a balanced tree.
- An unbalanced tree. (This is what is typically meant by “hierarchy”.) Applications include taxonomies of all sorts (especially in information organization), inheritance hierarchies (e.g. subclassing), and graph visualization.
- Pyramids. Defined in [GordonA 1999], these are slightly modified trees: the nodes in each level are totally ordered, and each node is allowed to have either one parent or two adjacent parents. They are thus a particular form of hNeartree.
- Concept / Inheritance Hierarchies. These may in fact form Neartrees, or it may be desirable to write or transform code in such a way that Neartree-like properties exist, i.e. limiting the number of objects that a single object inherits from or references. However, the underlying structure of the hierarchy itself does not seem to have been a focus of interest in this area.
- Decision Trees. They implicitly partition the data space into yes-no subsets via a

monothetic divisive algorithm, and typically the fanout at each node is 2 or 3. The tree is really a compact representation of a decision process; a single path is followed given any input, to find the leaf node encoding the output action.

- Computational Geometry (quadtrees and image-recognition trees). Quadtrees and octrees are compact representations of spatial structures.
- Selfridge's Pandemonium. This was an early hierarchical perceptron type method, in which specialized subagents recognize a specific characteristic of input, and pass their individual judgements on to an overall decision agent. A recent paper using this idea with more modern neural networks is [Smieja 1996].
- Neural Networks. Neartrees have some similarity to feedforward neural networks. Differences include: i) bounds on parents and children, ii) a single final decision node, iii) each node can use many different mechanisms, and iv) neural nets typically do not decompose a domain (though there is ongoing research in modular neural networks).
- Bayesian Networks, while they do decompose domains, are typically neither Neartrees nor hierarchical. However, they do offer insights in probabilistic decomposition, and in opinion pooling methods as in [Pennock & Wellman 2002].

We see that the Neartree and its variants are indeed new data structures.

Operationalizing the Neartree

Let's summarize the basic structure and goals of a Neartree implementation. The domain of interest is recursively subdivided into a series of node levels; each node has a set of values along with links to parent(s) and children. The overall structure represents a near-hierarchy of valuations (or some other property), where upper levels are less-detailed models of the same structure than lower levels. (Note that although upper levels are less detailed, they may be more accurate models, if the lower levels have a lot of errors or take too much time to compute).

Consider the functions that the Neartree could potentially be used for:

- **Static Optimization:** We may wish to find the best configuration of node states in a given static problem space (e.g. combinatorial optimization problems).
- **Dynamic Optimization:** If the external environment is changing, we may wish to find a good configuration of node states, while realizing that what constitutes a “good state” may change in the future.
- **“Pessimization”:** If we are risk-averse, or if the downside is bad enough, it may be more important to prevent the worst case. Here we want to “satisfice”, and find a solution which is good enough in all salient respects while minimizing the danger of a bad case occurring in the future.
- **Focus:** Highlighting valuable or dangerous subparts, critical paths, and so on.
- **Measurement and Assessment:** By aggregating values in subNeartrees, the system provides an objective and general metastructure for function evaluation in (near) hierarchical structures (e.g. hierarchical EC).
- **Representation:** Decomposing a problem into a Neartree format, specifying links and the distinguished point of view of the root node, and defining the criteria and valuation methods which are important could itself be a valuable exercise for a complicated problem domain. In essence, this is a heuristic for representing and manipulating a domain-specific theory, for a single cognitively-bounded agent.
- **“Process optimization”:** [Baum 1998] discusses “agoric AI”. As opposed to optimizing a configuration, a more challenging problem is to optimize behavior in a changing environment, by explicitly valuing sets of different possible actions. Baum’s approach is related to reinforcement learning, and focuses on representing compact strategies as sets of economically-driven agents. Since some valuations have a Neartree-like structure, this might provide a testbed for higher-level cooperative structures to emerge from lower-level valuation and selfishness.

With these functions in mind, let’s now turn to algorithms and applications for Neartrees.

4.2 Theory for Neartrees

Basic Recursive Unit: Node with Uplinks and Downlinks

Table 2...Components of a Node	
Node	A single basic unit; nodes are combined to make a Neartree. A node only knows about and communicates with itself, its parents, and its children.
Root	Distinguished node at root of Neartree. Usually the ultimate source of resources, control, and direction, and master locus of decision-making and evaluation. (Note however that all non-leaf nodes will typically also allocate resources and make decisions about their subtrees.)
Downlink	Performs input from children, via getting observations from them. Downlinks are weighted so the amount of attention paid to each child can be changed. Downlinks also can optionally perform output along downlinks by telling their child nodes to do something: allocate resources, perform problem-specific action, and so on.
Uplink	Passes information up to the various parent nodes (if all nodes have only one uplink, the Neartree becomes a tree); optionally receives resources and commands. No control commands or resources can be passed up an uplink. (Note that downlinks and uplinks are the same underlying data structure; we use the separate terms when looking at a node-specific point of view.)
Combiner	Every node with child nodes has a combiner between itself and its child nodes; input from all child nodes feeds into the combiner, which filters and compresses the information into a vector value. Essentially, the combiner takes all the information from child nodes, compresses it using both an internal model of how the child nodes fit together and parameter values for quantities such as degree of approximation required, and passes the compressed information into its associated node. (Clearly, this combiner could potentially be very complex. One challenge of implementing Neartrees in a particular domain is to specify combiners that are both computationally tractable, and that capture sufficient metrics about child nodes to enable competent decision-making at a node.)

General Algorithmic Properties of Neartrees

Since Neartrees are a generalization of trees, we inherit a lot of nice algorithms from the world of data structures in the special case where Neartrees are strict trees (i.e. $p=1$). Specific algorithms of interest include searching, updating, and balancing; see [Cormen et al 2001] and [Knuth 1997] for theory and examples.

For the general case where $p>1$, the algorithms used depend to some extent on the application for which the Neartree is to be used. General heuristics that we have found useful include breaking the least number of parent dependencies, and solving higher levels preferentially. We discuss algorithms in detail later in our application section.

Considering the effect of increasing each of the parameters c , p , L of a Neartree gives insight into the role each plays:

- By increasing c , we increase the complexity of the Combiner at a node. This increases the complexity of calculating $@(jH_k)$, by increasing the difficulty of computing the Combiner function. It also increases the difficulty of finding good target states to assign to child nodes, i.e. doing our desired process (optimization, pessimization, or other) on the Combiner function.
- By increasing p , we increase the constraints on the possible target states $?@(jH_k)$ of a node, since we are increasing the chance of the parents $?_jH_k$ requesting their common child node jH_k to be in different states. This suggests that increasing p implies that most parents must request the same state for children that they have in common - an observation we verify experimentally later. Many familiar systems display such behavior, with subcomponents designed to meet a single set of goals such that the goals are useful for most components which would want to use that subcomponent.
- Increasing L has potentially conflicting effects. We may both increase efficiency of the problem solving process (as shown in the Optimizing Parts vs Wholes section), and increase the lossiness between the root and leaf nodes (since in many real problems there

is delay, approximation, or communication cost involved in each additional level - an issue explored in e.g. control theory).

We can look at the effect of varying c , p , and L from another point of view: since we usually have a fixed amount of computational resources available, what has to change if we want to try to solve a problem with a higher value for a given parameter?

A homogeneous (c,p) -Neartree (i.e. one whose c and p parameters are similar across levels) will have a characteristic “level growth exponent”. Consider both c and p : since each node has at most c children feeding information into its combiner, the number of nodes can grow by at most a factor of c at each successively lower level. However, depending on the degree of overlap between children, a Neartree with each node having c children may have as few as c nodes at the next lower level (in the limiting case where all nodes at some level have the same nodes as children). (Note that this implies that a Neartree can actually have the number of nodes at the next lower level drop if the overlap is high enough.)

We can formalize this as follows: c and p are upper bounds for the number of children and parents respectively of any node. However, the actual number of children and parent nodes may be less than this upper bound; denote the mean number of children and parent nodes across all nodes by c_{avg} and p_{avg} respectively. Assuming for the moment that the variance in the actual numbers of children and parents is low (i.e. that most nodes have close to c_{avg} children and p_{avg} parents), we see that

$$|_{j+1}H| \cong |_jH| * (c_{avg} / p_{avg})$$

The quantity in the rightmost term is exactly the level growth exponent we are searching for; we thus define

$$l_{avg} = c_{avg} / p_{avg}$$

For homogenous Neartrees, l_{avg} characterizes the growth in size between successive levels.

(Heterogeneous Neartrees have parameters that vary across levels, but a similar if more complex analysis could still be done if the characteristics of each level are known. Note for example that if l is drawn from the same PDF across levels and there are a substantial number of levels, then the distribution of $|lH|$ will be approximately lognormal - a consequence of the Central Limit Theorem.)

Another important characteristic of a Neartree is the number of paths from any given node to the root node. Why is this important? Recall that a tree has just one path from any node to the root node, which can trivially be found by recursively adding the parent node of the current node to the path. This makes trees very simple in many cases - there is never any question about which subtree a node belongs to, or about the “path of control” from the root to the node.

However, in Neartrees the situation is more complex. Since a node ${}_kH_i$ may have many parents, and each of those parents could have many parents, the number of ancestor nodes which could be affected by a change in $@({}_kH_i)$ may grow very quickly as one moves up toward the root. This means that the effect of changing a single node may propagate widely through the Neartree as a whole - in contrast to the tree case where the effect of changing a node is more limited in scope.

How many more paths will there typically be from a given node ${}_kH_i$? Let’s first consider the number of ancestors of ${}_kH_i$. Clearly the quantity p_{avg} which has already been introduced tells us a great deal - an “average node” ${}_kH_i$ will have about

$$p_{avg} \text{ ancestors in } {}_{k-1}H,$$

$$p_{avg}^2 \text{ ancestors in } {}_{k-2}H,$$

and so on; thus the number of ancestors at each preceding level will increase exponentially until we reach some level S for which

$$| {}^{-(k-S)}_k H_i | \cong | {}_S H |$$

(Note that, at this point, the ancestors of our starting node include most or all of ${}_S H$, and hence a change in our original node affects the whole of ${}_S H$ and all levels above.) From then on, the number of ancestors at each level will drop exponentially as we approach the root node, by a rate l_{avg} per level, until the root node is reached; this is so since the size of the levels as a whole has that property, by our previous discussion.

Further analysis reveals more detail. We can also find a possible guideline for constructing Neartrees with a reasonable number of paths to the root - where “reasonable” will for the moment be taken to mean $O(|{}_k H|)$ for any level ${}_k H$. (Why? Well, this seems like a reasonable upper bound, since it implies that the total number of information or control paths from ${}_k H$ to the root must be at most quadratic in the size of ${}_k H$. A more detailed analysis may be possible if more is known about a specific Neartree, using e.g. binomial coefficients for the parents at each level, and estimating the distribution of sizes or probabilities of parent sets.)

So, let’s define $N = |{}_k H|$. Then the number of levels k between ${}_k H$ and the root is about $\log_{l_{avg}} N$.

Now we need to count the number of paths to the root from an “average node” in ${}_k H$. Note that the number of paths grows by a factor of p_{avg} at each level - and unlike the number of distinct ancestors discussed above, the number of paths continues to grow past ${}^{-(k-S)}_k H$, where $(k-S)$ is the number of levels until the ancestor set of ${}_k H$ includes almost all of ${}_S H$.

Figure 3...Paths to the root, and the information compression barrier

(However, past sH , the paths will have much more overlap. In a sense, sH seems like a sort of “information compression barrier”, past which the effect of any part of kH is similar across all ancestor nodes. An analogy would be to imagine your family tree - as you go back through the generations, the number of ancestors will on the average keep growing. But at some point, your ancestors will include most or all of the humans alive in that epoch, and you might consider yourself “equally related” to ancestors before then.)

So, the number of paths to the root is p_{avg}^k - but since we saw above that $k = \log_{l_{avg}} N$, this gives us an explicit formula, given l_{avg} and N . Now, under what conditions will this be $O(N)$? Setting the number of paths equal to N :

$$N = p_{avg}^k$$

$$\log_{p_{avg}} N = k$$

$$\text{But } k = \log_{l_{avg}} N,$$

$$\log_{p_{avg}} N = \log_{l_{avg}} N, \text{ and so}$$

$$p_{avg} = l_{avg}$$

$$\text{But } l_{avg} = c_{avg} / p_{avg} \text{ (by definition of } l_{avg}), \text{ and hence}$$

$$p_{avg} = c_{avg} / p_{avg}$$

$$p_{avg} = c_{avg}^{1/2}$$

which gives us a condition that a Neartree should satisfy, to have “relatively few” paths to the root from most nodes.

Optimizing Parts vs Wholes: The Neartree Case

Let’s consider extending our previous mathematical model “Optimizing Parts vs Wholes” from a tree structure to a (c,p)-hNearthree. In this case, we can still compute subparts independently, and use each level’s solutions as inputs to the next level above. However, a subpart can be used by two or more parent-level parts.

How does the total computation done change? One way of characterizing it is through l_{avg} as defined in the previous section. Using the same methodology as in the previous “Optimizing Parts vs Wholes” section, we set n equal to the total number of nodes and L equal to the maximum level number of the Neartree. We then have the relation:

$$n = 1 + l_{avg} + l_{avg}^2 \dots + l_{avg}^L = (l_{avg}^{L+1} - 1) / (l_{avg} - 1)$$

Holding n and c_{avg} constant, what happens as we modify the structure away from a tree, i.e. increase p_{avg} from 1? We see l_{avg} will drop, and hence the number of levels L must rise.

Alternatively, if L and c_{avg} are held constant originally, then the total number of nodes n must fall - this intuitively plausible result implies that the case where one moves from a tree to a Neartree means reusing subsolutions, and hence reducing the overall number of cases to be solved.

Finally, if n and L are held constant originally, then l_{avg} must remain constant, and so c_{avg} must increase in direct proportion to the increase in p_{avg} . However, recall that our $f()$ function - the function describing how much work needs to be done to solve a problem of given size - is superlinear in the size of the input. Since c_{avg} measures the average size of a subproblem, increasing this function will drastically increase the amount of work that needs to be done overall.

(We may also consider the perhaps unusual case where $p_{avg} > c_{avg}$, i.e. if relatively few building blocks are combined in many ways. Then we have $l_{avg} < 1$; clearly this can only happen if some number of levels below the root node have an $l_{avg} > 1$, so that there are sufficiently many parent nodes for the nodes at lower levels to have $p_{avg} > c_{avg}$. So in this case, there are (at least) two regimes of l_{avg} : one initial regime down from the root where it is greater than one, and another at lower levels where it may become less than one. If we can transform a Neartree or problem to this sort of layout, the increased reuse of lower-level solutions will be advantageous in the current model that we are considering - although not necessarily in some of the more sophisticated models we describe later.)

Note that, for problems with a quickly-growing (e.g. at least high-order polynomial) $f()$, if we relax the assumption that all nodes are close to the average values for c , then the variation in $|^+_j H_k|$ may dominate the difficulty - just one subproblem with this quantity well above c_{avg} could dominate the total solution time. This is a fairly complex case where the performance may be approximately determined by the degree of balancing in parents and the consequent maximum subcomponent size. One could find a relationship between the maximum acceptable variance of c , the number of nodes n , and the order of growth of $f()$ using e.g. Chernoff bounds.

Hierarchical Kolmogorov Complexity

Kolmogorov complexity is a different computational perspective than the time, space, and structural measures we have been discussing up to now. The basic idea of Kolmogorov complexity (also called algorithmic complexity) is to define a measure on objects according to the shortest program that generates them; due to the existence of a Universal Turing Machine which can emulate any programming language, this measure is well-defined up to an additive constant. Variants of the basic notion exist, e.g. bounding the space or time available to the program for reconstructing the original object; see [Li & Vitanyi 1997] for a comprehensive overview.

Kolmogorov complexity appears to be a useful way of characterizing objects defined as near-hierarchies. Suppose that we have an algorithmic problem P whose instances are specified with length at most $P_{\max}$, from which we construct a near-hierarchical extension through recursive specification, so that its description can be specified as a near-hierarchy H . If non-leaf nodes introduce no additional computation, then this near-hierarchy can be written as an equivalent single-level specification FlatH , which may be exponentially longer.

To see this, let each leaf node ${}_l H_k$ correspond to an instance of P . If we assume that the Neartree is complete in the sense that each node has the maximum number of children c , then each node in ${}_{L-1}H$ can then be written out as a problem of length $cP_{\max}$, each node in ${}_{L-2}H$ can be specified with length $c^2P_{\max}$, and so on; the root node ${}_0 H_0$ (which represents the entire problem) can then be specified with length $c^L P_{\max}$. This shows that FlatH (i.e. writing the near-hierarchical version in terms of P) is exponentially larger than a single non-hierarchical problem instance P .

However, the Neartree specification H itself may have much less length. Counting the number of nodes in H gives an approximation to this length:

$$|H| = 1 + l_{avg} + l_{avg}^2 \dots + l_{avg}^L = (l_{avg}^{L+1} - 1) / (l_{avg} - 1)$$

So if we take the ratio $|H| / |\text{FlatH}|$, we get (assuming as usual that l has low variance)

$$\begin{aligned} |H| / |\text{FlatH}| &= (l_{avg}^{L+1} - 1) / (l_{avg} - 1) c^L \\ &\cong (c / p)^{L+1} / (c^{L+1} / p) \\ &= (1/p)^L \end{aligned}$$

What does this tell us? $|\text{FlatH}|$ is the length of writing down H without using the space-saving representation of a near-hierarchy - but we managed to write it in a much shorter form, using just $(1/p)^L$ as much space. From a basic result of Kolmogorov complexity, most objects are not compressible. Combining this fact with our result above, we see that a near-hierarchical specification represents an object with relatively low Kolmogorov complexity.

Requirements and Goals for Neartree Applications

When and why would we use Neartrees?

As data structures, Neartrees can be used whenever problems contain data that is suitably structured in a recursively clustered way. However, hierarchical Neartrees as used to represent nested hierarchical systems are more specific - we use them when the problem domain has a naturally near-hierarchical nature, such that subproblems and levels corresponding to nodes of a Neartree can be coherently defined. Several additional conditions seem desirable:

- Nodes should have measurable individual characteristics, e.g. one should be able to reliably observe or estimate the parameters in each node's Combiner function, along with the external environment which perturbs or directs the node.
- The subproblems represented by each node should be relatively stable over the time

period which the Neartree is computing.

- Bounds for c and p should be low for most applications, to capture computational speedup.
- Subnodes of a given node should operate independently of their sibling nodes at each iteration. (This is a relatively strong restriction, that could be loosened in future work. Note that some sibling interaction is implicitly defined in the Combiner algorithm.)

One goal of developing the Neartree framework was to design a data structure and protocol which could meaningfully provide some representational power for very large problems - large in space, time, or other properties. (Other such computational methods include Bayesian networks, UML, and component-based software.) As we have seen and will see again later, a near-hierarchy is very efficient computationally, and there are some natural large-scale problems which seem representable in this manner. For such large-scale problems, we would also like:

- “Persistent hierarchy”: Node definitions and valuation methods should be stable for long periods of time, so that Neartrees can be built collaboratively over time, and different people can swap individual nodes while retaining the knowledge implicit in the rest of the Neartree. Similarly, levels themselves should be stable.
- Subproblems may be of independent interest as problems of longstanding research and operational interest in their own right. Combining solutions to subproblems in a Neartree can then leverage expertise in different domains, to solve larger-scale problems that may be beyond the capacity of any single discipline or approach.

What is the use of Neartrees, and of the hierarchical problem-solving framework in general?

Simply **describing a problem hierarchically** has a great deal of value, since it implies that a problem has been decomposed into relatively independent subparts. This allows each subpart to independently progress in theoretical and experimental terms. It also fosters

ontological coherence - an accurate near-hierarchical description, where accurate, provides structure to a knowledge or problem space in ways we discuss throughout this thesis.

Once a system has successfully been modeled hierarchically, the most obvious task is to **find an optimal state** for the system.

Given a set of stochastic processes driving the time evolution of the system, the model can be used for **risk analysis and scenario exploration**. Similarly, one can find key nodes and weak points of the system, by finding which system deltas would have maximal impact on the system state or value.

With a hierarchical representation, one can choose to use just some subset of the levels below the root - resulting in an **efficient higher-level approximation** of the problem. This represents a tradeoff between level of detail and resource requirements; a compact representation is usually easier to learn and faster to use.

Given a valuation function from states to values that is valid at each node, and a function that combines these values going up the Neartree, one can **value the system state**. In general, knowing the value of a system (with respect to one's value function) is essential to making conscious progress. Being able to assign partial values to subsets of the system is even more powerful, as it lets one move toward desired system states independently in different system subsets.

If there is any choice in how to construct a system, one may try varying models and **explore architectural properties**, to find the one with best properties. For example, one can characterize the difficulty of optimizing (or even satisficing) the system, as a function of key system difficulty parameters.

5. hSAT: Going beyond SAT

Without changing our patterns of thought, we will not be able to solve the problems we created with our current patterns of thought. - Albert Einstein

5.1 Generalizing SAT

We've discussed several facets of system decomposition and analysis, developed a mathematical model to show the conditions under which decomposition is beneficial, and specified the Neartree data structure and algorithmic framework. Now we proceed to the next logical step: implementing the Neartree framework to analyze a specific application domain. The domain we choose is hierarchical SAT.

SAT and the generalizations we have developed are of both theoretical and practical interest. The goal is to create a controlled “microworld”, within which external confounding factors are minimized and the properties and utility of our methods can be clearly demonstrated. (This microworld approach is often preferred for ease of analysis, as the absence of a large and complex external environment increases confidence that results observed are a consequence of the system and algorithms specified by the experimenter, and not just a side effect of some unknown property of the external system. See [Thompson 2000] or the agent-based simulation literature for further details.)

SAT in a Nutshell

The SAT problem is well-known in Computer Science, as the first problem to be proved NP-complete. It is at the core of much interesting work in theory and applications; see [Du et al 1997] for a survey. As Stephen Cook says in that volume:

“The first, and one of the simplest, of the many problems which have been shown to be NP-complete, SAT holds a central position in the study of computational complexity. [...Many] practical problems are NP-hard and may be transformed

efficiently to SAT, or have component problems which can be. Thus, a good SAT algorithm would likely have considerable utility. Since it seems improbable that a polynomial time algorithm will be found, 'good' might mean performing well on average, or with high probability, or on a class of "interesting" inputs. The hard inputs for a class of algorithms characterize the limitations of those algorithms, and point up where additional research is needed."

In ordinary SAT, one constructs a boolean formula with variables connected by $\vee$ (OR) and $\wedge$ (AND) operators, and then seeks an assignment of truth values to boolean variables in the formula such that the whole formula evaluates to T (true). Unless $P = NP$, this has exponential worst-case time complexity.

Typically the formula will be expressed in CNF (Conjunctive Normal Form), where a k-CNF formula is a conjunction of disjunctive clauses, with each disjunctive clause having at most k terms. An example of a 3-CNF formula is:

$$(A \vee B \vee D) \wedge (A \vee C \vee D) \wedge (B \vee C \vee D) \wedge (A \vee B \vee C) \wedge (B \vee C \vee D)$$

This formula has a satisfying assignment: (T, F, T, F).

k-CNF-SAT is one of the canonical NP-complete problems, for $k \geq 3$. 2-SAT is polynomially solvable; [Bang-Jensen 2001] (Ch 1.10, p 35) shows that 2-SAT is in P - in fact, it can be solved in time $O(\# \text{ of clauses})$. A problem class that interpolates between the P 2-SAT and NP-complete 3-SAT is $(2+p)$ -SAT, where a fraction $(1-p)$ of the clauses are 2-SAT clauses and the remaining fraction p of the clauses are 3-SAT clauses.

SAT can be generalized to a class of similar Boolean constraint satisfaction problems. Examples include Not-All-Equal-SAT and 1-in-3-SAT, where the names describe the required truth values of the literals in each clause; both of these are also NP-complete. A general family of such constraint satisfaction problems can be defined; remarkably, a theorem of Schaefer shows that this family has the dichotomy property that an easily defined subset is in P while all the rest of the family is NP-complete. A summary of this result and

related work can be found in [Creignou et al 2001].

The ratio of clauses to variables is a key parameter in random boolean formulae, often called the density of the formula. It has been shown that as the density increases, there is a sharp phase transition at some key value - different and increasing for each value of k , the number of variables per clause - where almost all boolean formulae with values below the transition are satisfiable, and almost all boolean formulae with values above the transition are unsatisfiable.

For $k=3$, experimental work suggests the parameter is around 4.3; an upper bound of 4.598 has been proven in [Kirov et al 1996]. An easy informal proof (discovered independently many times, see e.g. Cook's paper in [Du et al 1997]) demonstrates the weaker upper bound of 5.19:

- Fix the ratio of clauses to variables R , and generate a random formula.
- Now any individual clause has a $7/8$ probability of being satisfied by a particular fixed assignment, and hence that assignment satisfies the whole formula with probability $(7/8)^{Rn}$.
- But there are 2^n such assignments, and therefore the expected number of assignments satisfying all Rn clauses of this n -variable formula will be $2^n (7/8)^{Rn}$.
- This quantity (and the probability of being satisfiable) will approach 0 as n approaches infinity when $R > \log_{8/7} 2 = 5.190893...$

Similarly, one can explore how running time changes with fixed density and increasing formula size; further discussion and experimental exploration of this phenomenon can be found in [Coarfa et al 2001]. The structure of SAT solution space is currently the focus of much research; see for example [Slaney & Walsh 2002] for phase transition behavior in optimization versions of SAT.

From SAT to Hierarchical SAT (hSAT)

How would one extend the basic SAT idea? The first step is to seek some hierarchical or compositional extension. In the problems we have considered thus far, it has been possible to decompose a problem into “more detailed” subparts; consideration of these more detailed subparts, along with interaction between subparts as defined by the link structure, allows us to find better solutions (at the cost of greater computation). The natural analogue of this in a boolean formula would be to have a multilevel boolean formulation - we’ll call it hSAT for hierarchical SAT.

The only previous work that we are aware of in this area is a sparse series of papers through several decades on compact specifications, of which a recent example is [Marathe et al 1998] (which contains further references to previous work in the area). They explore compact specifications for several problems, include graph and satisfiability-type problems. These specifications are essentially algorithmic descriptions of problem structure, which can be expanded via prespecified rules into the full structure; they demonstrate PSPACE-completeness of many natural problems specified in this way. (This property is partly due to the exponential increase of the expanded specification size as compared to the compact specification which appears to happen only if lower-level nodes are used more than once in upper-level combiners - e.g. it doesn’t seem to happen for hTSP, defined later.) Another paper along these lines is [Condon & Agarwal 1997], on Hierarchical MAX-SAT. However, our framework appears to be more general than these, as well as including experimental work.

At the highest level, one simply has a small case of ordinary SAT, with a few variables to assign truth values to and an understanding of how those variables interact (as represented by a boolean formula). However, each of these variables represents a composite structure, which can be broken down into a set of subvariables with its own boolean formula, and so on recursively. The idea is that a decision-making entity at a given level can solve a reasonably-sized problem given the solutions of the next lower level, but a full analysis of

the problem requires combining many evaluations at different levels.

Example: Using the Neartree nomenclature for nodes, let's denote the variables by $\{ {}_1V_0, {}_1V_1, {}_1V_2, \dots \}$. For the formulae, we'll initially take a very simple case with the formula at each node being the OR of all child variables of that node (so that the formula at each node evaluates to T if any of its child variables are T). For simplicity, we'll let each node's formula have exactly three variables (so that the hSAT variable structure is just a complete ternary tree, going down however many levels we want).

Then ${}_1V_0, {}_1V_1, {}_1V_2$ are the highest-level variables - we set their parent, the root node ${}_0V_0$, to be the node whose truth value represents the truth value of the whole formula (i.e. the formula which determines its truth value from its children's value is just the OR of all children). The level-1 solution is just to check these three variables to see if any are T - what could be simpler? But ${}_1V_1$ itself contains three variables, i.e. ${}_1V_1 = \{ {}_2V_3, {}_2V_4, {}_2V_5 \}$. If we're considering the level-2 solution, we would need to check these three variables to see if any are T, and only in this case would ${}_1V_1$ be T. The extension to more levels and more complex formulae is clear - although see the later section on top-down vs bottom-up solution methods for subtle properties that need to be defined once one thinks about the solution process.

As discussed in the Neartree section, upper-level nodes use a Combiner to synthesize a value based on the input values of the lower-level nodes, which in this hSAT realization of the Neartree paradigm is a boolean formula. This Combiner function could be similar for most nodes, or more heterogeneous. We will usually be assuming that the Combiner is a different boolean formula for each node, but drawn from the same specified parameterized PDF. We use formulas instead of truth tables because formulas are far more compact, and because the ratio of clauses to variables forms an objective way of parameterizing problem difficulty that has been used by many researchers.

Figure 4...Hierarchical Boolean Satisfaction

In order to complete the specification of the problem, we need to choose the goal of the algorithm, a fitness function to compare the algorithm's performance across problem instances, and the method of coming up with particular problem instances (preferably parameterized via natural characteristics, so a meaningful cross-section of different and increasingly difficult instances can be tested):

- The goal is simple: to find a truth assignment on the leaf nodes, such that the overall tree evaluates to the desired state (which we will without loss of generality specify as T, i.e. ${}^?@({}_0H_0) = T$).
- The fitness function could be the time required to find the satisfying assignment, or (for

partial solutions) some function of the number of satisfied clauses.

- Instances could be chosen as random boolean formulae, with parameters being e.g. the fan-out, number of levels, and ratio of number of clauses to variables. This assumes a homogeneous structure for the Neartree, i.e. with fanout being constant across nodes and levels; relaxing this assumption would require additional parameters describing how the formula's structure varies across the Neartree.

Top-down vs Bottom-up Specifications and Solution Methods

Very well...we have specified the problem definition, and the evaluation of any particular assignment of truth values. Or have we? As it happens, there are some subtleties in defining how a hierarchical system's state should be self-consistently defined.

At the most general level (i.e. without considering problem- or instance-specific knowledge), several state-specification methods exist in the static case, each of which induces a natural solution scheme:

Fixed-target. The straightforward way of recursively defining truth values would be to have a series of formulae fixed for each node, and assign various truth values to the leaf nodes only. In this case, the upper level nodes “auto-generate” their values level by level based on these lowest-level values. The problem with this approach is that the leaf nodes are exploring blindly - knowing nothing about what is happening at the levels above, this is almost like random search.

Can we fix this by having each node tell its children when it has reached a desired state, so that its children can stop exploring? No, because if the goal is to get the overall tree into a desired state (i.e. to solve $?@_0(H_0) = \text{desired state}$), then all other nodes may need to explore many configurations to find this state, and in the general case no partial optimization is possible until the final state has been reached.

To see this, note that if $@(^+_j H_k)$ evolve to be equal to the states requested by $_j H_k$, then $@(_j H_k)$ will be equal to $^?@(_j H_k)$. But $^?@(_j H_k)$ - and hence $@(_j H_k)$ and $@(^+_j H_k)$ - may have to change later to accommodate the parents' desired states $^?@(^+_j H_k)$. Similar reasoning applies further on up the Neartree.

Perhaps there are some cases where all nodes at all levels of the overall Neartree are “separable”, so that we can use this approach, since each level could be optimized individually. But with most SAT problems this would seem to be far from the case.

Top-down. Characterizing the set of satisfying truth values at a given level, and then going down level by level to find the values at each lower level that would produce required values at the next level above. “I know my own target state $^?@(_j H_k)$...now which states should my children be in to get me into my target state? Let me guesstimate that - then I can tell my children their target states $^?@(^+_j H_k)$.”

This presupposes an estimation method, that allows good guesses about what should work at upper levels before actually being able to implement the guessed solution. In fact, this is what usually happens in real systems - we make models of a system, which let us make an educated guess as to good potential states for our system.

In the Neartree as opposed to tree case, this requires a “target collision resolution protocol”: if two parent nodes provide a common child $_j H_k$ with two different target states, the child node needs some method to reduce the given target state set to a single target state $^?@(_j H_k)$. For example, a simple protocol would be to choose the target state that is requested by the greatest number of parents, and break ties randomly.

Bottom-up. Alternatively, one could go up level by level, optimizing nodes level by level and tracking which nodes could be evaluated to both T and F so that an overall decision could be made at the root node. “By experimenting with my child node states $@(^+_j H_k)$, I know which states $@(_j H_k)$ I have the option of getting into, and the corresponding child state

configurations - let me pass that information up to my parents, so they know which states they can get into.”

This requires keeping a “library” of past solutions for each subnode. In the boolean-state case, one would store two representative canonical solutions if both outcomes are possible, or only one solution if all attempts to solve that subnode ended with the same outcome. In a case where a node $_jH_k$ has a wider variety of possible states, one would want to keep a larger solution cache at that node - equal to the size of the range of the function $@(_jH_k)$. As well, in the case where the environment is dynamic, one may wish to keep multiple solutions for each state at each node $_jH_k$, corresponding to alternative solutions that one could fall back to if e.g. $@(_j^+H_k)$ is unexpectedly changed (by the environment, or by the child itself).

“*Cyclic*”. It will often be desirable to combine the top-down and bottom-up methods. Why? Several factors, including dynamic environments invalidating previously-found solutions, multiple parents requesting different target states from common children, and limited computational resources forcing node-level computations to be approximations instead of exact solutions.

A cyclic solution method would bubble up information about what can be done, bubble down information about what should be done, and repeat. (Clearly this could be done more locally or asynchronously than the level of the entire Neartree, at the cost of additional complexity in controlling and understanding the process.)

(Note that in the simplest case, once we specify the leaf (lowest-level) truth value assignments, all upper level nodes “auto-generate” their values level by level based on these lowest-level values. However, in a more general case changing the child values of a node shouldn’t necessarily change the value of the node itself - rather, it may change the range of options available to the node. The key parameter to be distinguished here is how much choice is available at a node itself, given its child values; we discuss this issue later.)

In general, we would expect the top-down scheme to require less effort than the bottom-up scheme, since i) one could stop each lower-level node computation as soon as a configuration creating the desired value had been achieved, instead of continuing until configurations creating all values had been achieved; ii) the existence of a particular desired value may create a “solution gradient”, especially for cases where the functions have more possible values than just booleans. However, there are cases where the effort required by the two would be equal to within a constant factor, e.g. where the required value at the upper levels was present at only one configuration at each lower level. The top-down scheme should never require more effort than the bottom-up scheme, since one can use a bottom-up solution to solve a given top-down problem.

How much slower would the top-down and bottom-up schemes work in the hnTree case? The bottom-up scheme is much the same: for each node, the only relevant metric determining solution time is the number of child nodes, so the overall computation time will be roughly the same as for a strict hTree case with the same number of nodes. The top-down scheme could potentially be much slower, since many lower-level nodes might need to fulfil conflicting directives from different parents; note that this could lead to a “no solution possible” result in the strict hnSAT problem, but would introduce additional degrees of freedom to be managed in the relaxed hnSAT problem where satisfiability is some function aside from AND of the satisfied children, e.g. the proportion of satisfied children.

We also need to specify success metrics for partial solutions. If only some nodes are satisfied in a solution, how can we quantify the value of that solution? A simple method is to just count the number of satisfied nodes; one can alternatively count and maximize the number of satisfied clauses in BooleanCombiners, which is exactly the MAX-SAT case.

5.2 Theory and Applications of hSAT

3-hSAT and 2-hSAT are NP-Complete

In this section, we derive theoretical results for several simple cases of hSAT. The fact that even these straightforward cases are NP-hard points to the necessity of using heuristic approaches for hSAT.

We prove our results for “up-hSAT” : a problem instance consists of specifying truth values for all leaf nodes, along with formulae for calculating upper-level truth values in terms of their child nodes. Given such an instance, the upper level nodes are “forced” by their children, level by level, until a truth value for the root node is derived; the hSAT instance is T if its derived root node truth value is T.

Given this definition of up-hSAT, we show that 3-up-hSAT (where each node is restricted to have at most three children) is NP-complete. The proof is through reduction from 3-CNF-SAT, which is well-known to be NP-complete, e.g. [Cormen et al 2001].

Proof: 3-up-hSAT is NP-complete

Consider any instance Foo of 3-CNF-SAT. We need to show a polynomial-time method of creating a corresponding instance $hFoo$ of 3-up-hSAT, and transforming a solution back to a solution for the 3-CNF-SAT instance Foo .

Now Foo consists of the conjunction of an arbitrary but finite number of clauses $C_0 \dots C_N$, each of which is a disjunction of at most three variables. So make the bottom level ${}_L H$ of $hFoo$ a set of nodes corresponding 1-to-1 with the Foo variables. (We start numbering the levels with L , and reduce the level numbers by 1 as we work up toward the root ${}_0 H$, in order to stay consistent with the notation for level numbering. The L in ${}_L H$ is thus a constant

whose value is not known yet, but which can be deduced once the reduction procedure is understood for a particular instance.)

At the next-higher level $_{L-1}H$ of $hFoo$, create a node for each clause C_i in Foo , with child nodes being the variables in C_i , and truth value calculated as the disjunction of the child nodes. (Remember, for hSAT we can set the derived truth value of any non-leaf node to be any desired boolean formula of its children.) Note that consideration of the number of possible distinct clauses of 3 variables yields

$$\begin{aligned} |_{L-1}H| &\leq C(|_L H|, 3) * 2^3 \\ &< k |_L H|^3 \text{ (for some constant } k) \end{aligned}$$

where $C(|_L H|, 3)$ is the number of distinct sets of three variables, and the second the number of ways of assigning triplets of (negated, non-negated) states for these three variables.

At this point, if it were not for the restriction to 3 children per node, we could create a single node at $_{L-2}H$ connected to all $_{L-1}H$ nodes, and set that $_{L-2}H$ node to have derived truth value equal to the conjunction of all $_{L-1}H$ node truth values. However, the 3-child restriction does exist...but getting around it is easy enough. We just create a number of levels equal to $\lceil |_{L-1}H| / 3 \rceil$. Node k at each of these levels is connected to children nodes $3k$, $3k+1$, and $3k+2$ at the next lower level, and its truth value derived from the conjunction of these three children. (The last node in each level may be connected to fewer than three children, if the total number of nodes at the next lower level is not divisible by 3.)

It is clear that this new constructed 3-up-hSAT instance $hFoo$ is equivalent to the original 3-CNF-SAT instance Foo . To conclude the proof, we need to ensure that the time taken for the whole process is polynomial, and that a solution to $hFoo$ implies a solution to Foo . Each node of $hFoo$ and its corresponding boolean combination function (i.e. its truth value as a function of its children's truth values) is clearly constructible in constant time. The number

of nodes constructed is bounded above by

$$\begin{aligned}
\# \text{ of nodes} &= |L|H| + |L_{L-1,0}|H| \\
&\leq |L|H| + |L_{L-1}|H| (1 + 1/3 + 1/9 + 1/27 + \dots) \\
&= |L|H| + |L_{L-1}|H| * (3/2) \\
&< k |L|H|^3 \text{ (for some constant } k)
\end{aligned}$$

So the construction of $hFoo$ takes polynomial time.

(A similar gadget can be used to restrict parents per node to 3 as well. Nodes with more than 3 parents can “distribute” their values to these parents through a sequence of fanout nodes, each limited to 3 parents. We omit the details.)

The evaluation process to determine the truth value of $hFoo$ takes time a constant times the number of nodes, and hence is also polynomial time. Finally, $hFoo$ is T iff Foo is T. We’re done.

One might intuitively think that going from 3-CNF-SAT to 2-CNF-SAT does not change the time complexity for solution; however, as we have already mentioned this turns out not to be the case, and 2-SAT is in P. Given this result, it could be that 2-up-hSAT is also solvable in low-order P time, or in P but solvable with a higher time complexity, or that it is NP-complete like 3-up-hSAT. We show that the last option is in fact the case.

Proof: 2-up-hSAT is NP-complete

We show a reduction via 3-CNF-SAT.

The proof is very similar to the 3-up-hSAT case, with two modifications. First, in going from the lowest level $|L|H$ (where nodes represent variables) to the higher level where the

nodes represent clauses, we insert an additional level. So in this case, we have two nodes at level L_1H for each clause C_i ; the first node is the disjunction of the first two variables in C_i , and the second node is the unmodified truth value of the third variable in C_i . Then each of these pairs of nodes at L_1H has the same parent node at L_2H , with the parent node being the disjunction of its two children. (The point of this is to break the 3-variable disjunction C_i into two 2-variable disjunctions; it works because $A \vee B \vee C = (A \vee B) \vee C$.)

The second modification is that every level higher than L_2H is the conjunction of two child nodes (instead of three as in the original proof). This increases the total number of nodes by a constant factor, since each higher level has $1/2$ the number of nodes as the previous level instead of $1/3$, but that constant factor does not matter for establishing NP-completeness.

The remainder of the proof is as before. Since 3-CNF-SAT can be solved by a corresponding 2-up-hSAT instance, and 3-CNF-SAT is NP-complete, we reach the perhaps surprising conclusion that 2-up-hSAT is also NP-complete.

Note that the reduction process which shows that hSAT is NP-complete only tells us that some instances of hSAT are difficult. It does not say anything about the average-case complexity, nor about approximability - both of which are important in practical applications. However, proving NP-completeness is usually a good first step in characterizing a problem's difficulty. There is ongoing work in understanding the behavior of average boolean formulae, as discussed earlier.

This work also seems to imply that the flat SAT problems that an hSAT problem is equivalent to will have low KC; tree-formulae are sparse in the equivalent flat formulae that they expand to. So as we discuss later, one application of hSAT might be to reverse the process: given a flat formula, find a closest approximating tree-formula. These latter tree-formulae should be a compact and more efficient encoding of the flat formula, albeit often

quite lossy; maybe only some flat formulae will have useful tree approximations. One might tune the compression via number of levels. It seems difficult to find these, as one might have to find repeated subformulae, or use some decision tree like algorithm to pick the most discriminating few variables for each level...this is definitely NP-hard by reduction to e.g. graph or formula isomorphism.

SAT and hSAT Solution Heuristics

There arises now the tricky question of actually finding a satisfying assignment of values for a given hSAT. Given our proofs that even 2-hSAT is NP-complete, we need to use some sort of heuristic methods for practical solutions.

First, let's survey heuristic solution methods for the traditional SAT problem. (Our survey is based on [Gent & Walsh 1999]; we'll assume the problem is in CNF unless otherwise specified.). Then we'll extend these heuristics and combine them with the general framework discussed already, to come up with solution heuristics for our dynamic hnSAT case.

There are a large number of algorithm variants for solving SAT. The complete algorithms (i.e. those guaranteed to find a solution) all search the tree of possible assignments of values to variables. One well-known procedure, called DPLL after its authors, works by using rules such as the following to check all variable assignment cases and to simplify the formula where possible:

```
DPLL( $\Phi$ ){
  Sat: if  $\Phi$  is empty
    return satisfiable
  Tautology: if clause  $c$  is tautologous
    return DPLL( $\Phi - c$ )
```

```

Unit Propagation: if  $\Phi$  contains a unit clause  $\{l\}$ 
    return DPLL( $\Phi$  with  $l$  assigned True)
Pure Literal Deletion: if  $\Phi$  contains  $l$  but not  $\neg l$ ,
    return DPLL ( $\Phi$  with  $l$  assigned True)
Split: if DPLL ( $\Phi$  with  $l$  assigned True) is satisfiable
    return satisfiable
    else return DPLL ( $\Phi$  with  $l$  assigned False)
}

```

Many refinements to this basic procedure exist. Note that there is a choice of which literal to split on; several heuristics have been suggested to choose the literal most likely to lead to a satisfying assignment, e.g. MOM's heuristic (pick the one occurring Most Often in the Minimal size clauses).

Since a search of all possible assignments to a formula can take exponential time, heuristic methods for SAT have been explored. Many of these are essentially local stochastic hill-climbers, most of which share a similar idea:

```

Given a formula  $\Phi$ , randomly generate a truth assignment  $\alpha$ 
do {
    if  $\alpha$  satisfies  $\Phi$ , return  $\alpha$ 
    choose a variable  $v$  using some selection heuristic
    reverse the value of  $v$  in  $\alpha$ 
} while (time is not up)
return failure to find a satisfying assignment

```

Specific algorithms include:

- Gent's greedy algorithm: Assign a variable the truth value that satisfies the greatest number of clauses that contain it.
- GSAT: Start with a random assignment. Change any variable assignment which gives maximal increase (possibly zero) in the number of satisfied clauses.
- HSAT: Like GSAT, but if there is a choice of variable assignments to make which all give the maximal satisfied clause increase, pick the one changed least recently. (If two or more variables have not yet been flipped, use an arbitrary fixed ordering to choose.)
- Other modifications of GSAT include focusing attention on the hardest clauses, flipping with some probability a random variable in an unsatisfied clause (even if it decreases the number of satisfied clauses overall), and adding restarts. One well-known variant that performs well is WalkSAT.

Hybrid algorithms combining different approaches also exist, e.g. combining heuristic and complete methods, or using evolutionary computation to evolve effective heuristic methods as in [Fukunaga 2002].

Now, how can these methods be adapted to our hnSAT case? We will focus on heuristic methods, since their generality implies that they will be more suited to the variety of different circumstances in which we will be applying hnSAT. Empirically, they also perform much faster than complete methods on many problems (and in fact seem to be the only practical method for solving very large problems).

At the level of a single node, using variants of WalkSAT and GSAT seems the best approach, and this is the approach we have taken. Essentially we do stochastic hill-climbing at each node, by flipping variable assignments; we add several heuristics to improve performance.

The basic WalkSAT algorithm is as follows:

M = formula to solve, p = probability of greedy move


```

for tryNum = 1 to maxTries {
    A = random truth assignment to variables of M
    for flipNum = 1 to maxFlips {
        if A satisfies M, return true
        else pick a random clause C that is unsatisfied by A
        with probability p {
            v = the variable in C that breaks the fewest number of other clauses
        }
        else {
            v = a random variable in C
        }
        flip v in the truth assignment A
    }
}
return false

```

There are many variations of the basic algorithm. Heuristics include tracking which clauses were modified in the recent past, choosing the variable to flip based on additional criteria, looking at multi-flips (i.e. flipping more than one variable at once), and hybrid algorithms which add aspects of a complete solver like DPLL. Choosing good values for parameters such as maxTries, maxFlips, and p can be essential for high performance of the algorithm; [Patterson & Kautz 2001] looks at a self-tuning implementation.

Applications of SAT and hSAT

On a theoretical level, SAT is the canonical NP-complete problem. On a practical level, SAT has many applications. We mention some from [Du et al 1997] and [Gent & Walsh 1999]:

- Graph coloring. A k-coloring on an n-node graph can be encoded with nk variables, with

e.g. a clause for each edge and additional clauses for graph properties such as ensuring every node is assigned at least one color.

- Quasigroup existence. Quasigroups (or Latin squares) are $n \times n$ multiplication tables with each element appearing exactly once in each row and column; they model experimental design problems such as tournament scheduling and drug tests. Several open problems have been solved by encoding into SAT.
- Circuit design and verification. Many circuit problems can be easily mapped into SAT; logic gates are directly translated into boolean formulae, and the state of a wire (i.e. current on or off) is simply a boolean variable. These encodings have been used for fault diagnosis and circuit synthesis.
- Steiner trees. A network Steiner tree is specified by an undirected graph with weighted edges, and a subset of the graph's nodes called Steiner nodes; the problem is to find a minimum-weight subtree of the graph which includes the Steiner nodes. This has many applications in routing and network design. Encodings into SAT have been solved to yield solutions competitive with the best specialized algorithms.
- Other applications include scheduling, planning, diagnosis, channel assignment in cellular networks, and circuit partitioning.

More applications can be found in SAT conference proceedings, or at several websites devoted to SAT applications and benchmarks.

How about applications for hSAT in particular? At the current time, we are in a situation similar to the identification of SAT as a crucial theoretical problem: we have specified hSAT and given some evidence that it is theoretically important, but it will take time for practical applications to be worked out. Nevertheless, we can suggest several problem domains to which hSAT could be applicable:

- We have already discussed potential applications of general Neartrees - nhSAT could be used for some of these, as it is a specific instance of the more general Neartree framework.

- Just as SAT has been used to solve several suitably-encoded NP-complete problems, nhSAT might be used to solve encoded problems from an analogous hierarchical complexity class. E.g. hierarchical graph coloring, a graph problem, bin packing. We specify one such problem in detail later, when suggesting the idea of “Minimal Information Sets”.
- Circuits are often designed in a modular manner, with lower-level components being combined to make larger circuits. Thus one could use an hSAT representation for such modular circuits - with each SAT problem representing an individual circuit at the appropriate level.
- It might be possible to represent a large software project as a Neartree, with each node being a functional module, and links representing inclusion or functional dependency. Then an hSAT description might encode desired properties of each module, e.g. conditions for each module to be successful in terms of lower-level modules. Of course, large-scale software project management uses critical path analysis, dependency charts, and time estimates to manage the time to completion. However, an explicitly near-hierarchical approach to software metrics and management could provide an additional method of empirical analysis.
- [Mali and Kambhampati 1998] have looked at encoding hierarchical tree networks in Boolean formulae. Such networks are a recursive way of specifying a model for a problem-solving process, which might work better or be easier to understand if they could be restricted to Neartrees.

We also discuss later several natural generalizations of hSAT, and the applications to which these generalizations could be suitable.

6. hSAT: Implementation

“What I cannot build, I do not understand.” - Richard Feynman

6.1 Parameters and Program Flow

We have implemented the ideas discussed in the preceding chapters in a Java framework. In this section, we detail the structure of the program, and some design choices made.

Program Functionality and Parameters

Table 3...Neartree Parameters		
Parameter	Bounds	Discussion
N - Total # of nodes	100 000 (usually 100 to 10 000)	Space used is approximately linear in the number of nodes.. Time cannot be bounded from the number of nodes alone, but also depends on other parameters like number of levels, c, p, and time to solve each subproblem.
c	1024	Additional restriction to not violate the bound on N. E.g. choosing $c = 4$ children per node implies $L \leq 8$, leading to $4^8 = 64K$ nodes at the lowest level and $\leq 86K$ nodes total. (More children imply fewer levels: $c = 8$ and $L = 5$ levels implies $8^5 = 32K$ lowest-level nodes and $<40K$ overall.)
L	10	
p	32	If $p > c$, the first few levels will actually have fewer than p parents, due to shortage of nodes at previous levels.
Links	from c and p	Bounded by (c, p) constraints. Since each node can have no more than c children and p parents, the total number of inter-node links is then at most $(\# \text{ nodes}) * (c + p)$.
Resource Bounds	varies	Computation time is bounded by either max real time per run, or max search operations per run.
Runs	varies with other parameters	Increasing the number of runs at a particular set of parameter values increases the statistical accuracy, but also increases computation time. ($4x \text{ runs} \leq 2x \text{ accuracy}$.)

Algorithms

Algorithms implemented are as follows:

- Each node $_jH_k$ belongs to a specified level j , and has a unique assigned ID k in that level.
- Nodes have a **current state** $@(_jH_k)$, and a **target state** $?@(_jH_k)$ derived from requests given by parent(s). For a leaf node, $@(_LH_k)$ is chosen by that node, taking into consideration the target state requests from parents. For a non-leaf node, $@(_jH_k)$ ($j < L$) is a function of the derived state from $@(^+_jH_k)$ (and optionally additional choice at the node itself). For example, in our hSAT example where leaf nodes choose boolean states, the state of a non-leaf node is a boolean function of its child states.
- We use a **cyclic solution method** incorporating elements of both top-down and bottom-up solution methods.
- Having non-leaf node states be completely determined by their child states removes any room for action on the part of non-leaf nodes. Therefore, we let non-leaf nodes **set the target states** of their children - note that this does not lead to trivial solutions since desirable states will interfere between nodes! As well, nodes can optionally have some additional direct degree of influence on their states. So, a node can choose which of the state combinations of its children it prefers, as well as add some influence to its children's influence. (In future work, non-leaf nodes might also choose which children they want, intercommunicate, etc.)
- Given a particular Neartree specification, we can **search for a solution** (e.g. set of states of all nodes that makes the Neartree's overall value closest to our desired value).

A node's **expressed state** $@(_jH_k)$ is largely a function of $@(^+_jH_k)$; in the current case with the Combiner function being a Boolean formula, $@(_jH_k)$ is simply the Combiner output with $@(^+_jH_k)$ as inputs to the Combiner (i.e. with $@(^+_jH_k)$ being the truth values in the Combiner's boolean formula). However, this is a 0-1 function with a node reaching a state of T only if all of its child nodes have their desired states $?@(^+_jH_k)$ (or if $@(^+_jH_k)$ happens to satisfy a

solution of $_jH_k$'s Combiner other than the one that $_jH_k$ asked for - as may happen if there are many satisfying states for that Combiner).

In addition to state, a node also possesses a **Value** $\$(H_k)$, which is a measure of the fitness of that node. This fitness is calculated with respect to how closely $@(_jH_k)$ matches its target(s). Clearly, one might wish to extend this measure, so that a node's value represents not just how well its current state matches with its target, but what its expected contribution to the fitness of the overall Neartree is, in the long term. (In other words, a node value should also include its contribution to other nodes' values, as well as the estimated future contributions.) We will have more to say about this value measure later.

In the hSAT case, we can measure the progress toward a node becoming T (or F, if that is its desired state) by using a MAX-SAT-like metric: measuring how far the fraction of satisfied clauses differs from 1 or 0 respectively. (Note the asymmetry of outcomes - the function mapping fraction of satisfied clauses to value of formula is a step function at 1 in the regular SAT case, while it becomes a line of slope 1 in the fraction of satisfied clauses case.) This is the current measure used.

Another approach is setting $\$(H_k) = |\{n : n \sqsupseteq _jH_k \text{ and } @(n) = ?@(_jH_k)\}|$, i.e. equal to the number of children in a satisfied state. This implies each child node will always be attributed some fixed positive value for being in a satisfied state. (We could generalize this by measuring some distance function $D(@(H_k), ?@(_jH_k))$, and weighting D by the empirically-measured difficulty or importance to $@(_jH_k)$ of $_jH_k$ reaching $?@(_jH_k)$. One could compose this with some moving-average decay function, to add weight to the time history of a node's state with respect to its parent nodes.)

To measure the progress toward solution (or quality of a current partial solution) of a Neartree as a whole, one can either specify weights for each level and node, or use some default heuristic to assign weights. Usually, having good values in a upper-level node is worth more to the overall Neartree's value than having similar values in a lower-level node;

this property is common across most hierarchies in both algorithmic and real-life situations. Two default heuristics are:

- Uniform: Each level jH has equal weight, divided evenly between all jH_k .
- Potential: $\text{Value}(jH) = \sum_k \text{Value}(jH_k)$...the higher the value is, the more capacity jH “has to work with”; children being in good states gives a parent more power.

We have currently implemented the first heuristic.

6.2 Data Structures and Optimization Methods

Data Structures and Class Structure

Our program is written in Java. Let’s define the important classes and methods that our program uses.

Nodes have **State** and **Value**. The State is an arbitrarily complex data structure which is problem-dependent; each node has a given state, derived from the states of its children and its **Combiner**. In order to decide how useful each state is, nodes also have Values derived from the past history of States; the Value is simply a scalar real number, or a vector of such numbers if one is measuring multiple properties. State represents the particular current configuration; Value represents the value of that configuration, either from external rewards or from an internal judgement.

Each node stores its children as a resizable array. While this makes adding and removing children more costly, we make this design choice to favor speed, since children will be added or removed rarely (if at all) relative to the number of times they are accessed. (For an application which needed very frequent changes, the option exists of using a balanced tree style of data structure if rebalancing is required.) However, the speed sacrifice

is not too bad, since the maximum length of a chain of links from the root node to a leaf node is logarithmic in the number of nodes in the Neartree, while the maximum size of a single child array is c .

Figure 5...Internal structure of an HNode

```

class Node // this is an inner class of Neartree
class LeafNode extends Node
class HNode extends Node
class RootNode extends HNode

```

(Node has three subclasses: LeafNode, HNode, and RootNode. Each is specialized for its position in the Neartree. We show the methods together below for simplicity.)


```

state // this node's current state

value /** a function only of the state and not position within Neartree, e.g.
        Neartree.Value(this.state) */

childNodes/** between 0 and c child nodes that compose this node. Note that this
            sort of recursive structure gives access time proportional to height of the
            tree - 1 method call per level, 1 array lookup to get appropriate child. */

parentNodes // nodes up to which this node passes information (between 0 and p).

TypeOfCombiner combiner/** can swap in appropriate combiner for childValues
                            (Combiners are split off into a separate class) */

public Value valueMyself() // could be called by self, or by parents
    state = combiner.Eval(childValues) // first, we are what our children are
    state.modify()// next, we may be able to exert some additional choice on state
    value = Neartree.Value(state) // evaluate my state
    return value

decideTargetState()
    /** based on accumulated parent requests, decide which state to aim for. If there
        are multiple incompatible requests, use a conflict resolution protocol - currently
        weighted majority rule. */

setChildrenTargetRequests()
    /** given my Combiner, solve the Combiner to find some satisfying set of hypothetical

```

Child states. Once this set is found, request children to take these states. (This is often the most computation-intensive part.) */

monitorChildNodes()

/** observe value patterns from child nodes; if low value, tell subNodes to work harder; if one node stuck, give it more resources or work around it */

listenToParents()

get state target requests from parents

(optionally, get resources from parentNodes based on value provided)

class Neartree // represents a Neartree as a whole, and contains all Nodes

RootNode /** distinguished Node; we set the goals for the whole Neartree by telling the Root to do something, and it recursively directs its children */

nodes[][] 2-D array, storing all nodes directly. (Note that nodes are also linked to each other via their parent and child links, which are used to pass requests and status; we use the array references for speedy access.)

fanout, fanin, L, etc // structure of this Neartree

StateView view // separate class storing consolidated status of all nodes, and statistics on Neartree; view is what is graphically represented

Also methods to give each node a timeslice in turn, synchronize / evaluate status, and

call director methods that each node uses in its timeslice (Optionally a timeMultiplier, with upper levels being a constant factor slower than next lower ones in action or analysis - the rationale is it takes time to observe and adapt.)

```
class Combiner    /** knows State types, and has a model for combining childStates into
                    an overall State. (Combiners are central to control flow and evaluation
                    and operate on Node States; one can subclass Combiner to any
                    particular evaluation function on nodes, as we have done with
                    BooleanCombiner for the SAT problem.). */
```

```
Eval(nodeStates) /** combines the states nodeStates, via a model which knows about
                    State types and how a node is approximated by its childNodes. Each
                    node has its own Combiner. Across the Neartree, Combiners can be
                    different samples from the same problem class, or even from entirely
                    different problem classes. */
```

```
Solve() /** search for some set of child states which satisfies this Combiner. Often (as
          in our implemented hSAT version) there will be multiple search methods, each
          with tunable parameters. This allows searching through the space of solution
          methods, for effective solutions. */
```

```
// other current and future functions: check goodness of approximation, evaluate at a
resolution matching available resources, track past valuations
```

```
class BooleanCombiner extends Combiner
```

```
size, ClauseRatio /** formula parameters */
```

generate(params) */* create a Boolean 3-CNF-SAT formula, sampled randomly from the distribution given by params (i.e. with given size, clauseRatio, etc)*

load(DIMACS file) */* load a 3-CNF-SAT formula in DIMACS SAT format. This allows comparison with benchmarks, and interfacing with external generators of problems from specific problem classes. */*

Eval(childStates) */* given a formula that the combiner knows about - return true/false according to the formula being satisfied by childStates. If doing MAX-SAT, return fraction of clauses satisfied. */*

class Value

/ Some valuation of the “goodness of a node’s state.” This compressed information about a node can be nominal, ordinal, discrete, real, vector, or a general data type. The main requirement is that one can compare different Values. */*

class State

class StateVector

class StateVectorWithID extends StateVector

/ State stores a single state, while StateVector stores a set of states, representing for example the states of a node’s parents or children. StateVectorWithID is if we want to store who each state belongs to, or more interestingly track the history of a state (e.g. # of time steps that this state has had its current value). */*

```

class StateView
class ValueFrame

```

```

/** Summary of states of entire Neartree. Every k ticks, the Neartree examines itself
and passes detailed information to StateView, which calculates statistics of node
States and Values, and values for the Neartree as a whole. This summary information
is then optionally passed to a graphical output method (ValueFrame) for human
observation and analysis. */

```

Top-Down, Bottom-Up, and SAT optimization Pseudocode

There are several key optimization algorithms that are basic to the program as a whole. In this section, we describe them via pseudocode. (Our implementation uses both top-down and bottom-up optimization, in a “cyclic” procedure; here we separate them for clarity.)

Top-Down Optimization

```

Nearthree nTree;
BFTraversal nTreeTraverse; // a breadth-first traversal of all non-leaf nodes, going down
                           from the root

// Assign reasonable targets
for each node in nTreeTraverse {

    /** first resolve a target value for myself: this is externally assigned at the root node,
    and otherwise whatever parents ask if they all ask for the same target, or some
    function of parents' requests if they ask for different targets */

```

```

set target state targState; // externally assigned at root

/** search through candidate child states for combination giving targState, using local
search or other method */

do {
    generate potential vector childStates;
    candidateState = evaluate(childStates);
} while ((candidateState < targState) && (timeIsNotUp))

/** ask each child to set its target as the corresponding state from childStates; if a
child receives two or more different target requests, it will return false to those it
doesn't currently intend to fulfil */

for each child {
    boolean tryTarget = child.tellTarget(childStates(child))
    if (!tryTarget) mark child as needing different target next time
}

} // next node in traversal

evaluate nTree; // this is an upward evaluation from leaf nodes, which gives the actual
states and value at each node once all targets have been assigned

```

Bottom-up Optimization

```

Neartree nTree;

BFTraversal nTreeTraverse; // a breadth-first traversal of all non-leaf nodes, going up

```

from the next-to-last level (i.e. skipping leaf nodes)

```
// Assign reasonable targets
for each node in nTreeTraverse {

    // look for a vector childStates satisfying each possible value
    for each stateType not yet found {
        set target value targState = stateType; // externally assigned at root

        // search through reachable candidate child states for combination giving targState
        do {
            generate potential childStates;
            candidateState = evaluate childStates;
        } while ((candidateState != targState) && (timeIsNotUp))
        if (candidateState == targState)
            myStateMap.add(candidateState, childStates);
    }

    tell parent myStateMap.keys() // e.g. tell parent which States I can do
} // next node
```

at root, choose which possible State to take

if no acceptable State, go back and search some more

There are additional components to the algorithm such as caching and multiple-parent heuristics, which will be discussed in the text as they are naturally motivated by experiments. As well, there is the node-level problem to solve:

SatSolve // node-level Combiner solution method, implemented as methods and classes inside BooleanCombiner

```
vars.init() // initial random assignment of children's states
value = Formula(vars) // initial evaluation of variable assignments (# satisfied clauses)

do {
    candidateVars = vars.mutate() // search via WalkSAT (smart variable flips)
    candidateValue = Formula(candidateVars)
    if ((candidateValue < value) || (float randF > randFlipProbability)) {
        vars = candidateVars
        value = candidateValue
    }
} while (value not good enough)
```

This completes our high-level survey of the algorithms and data structures implemented. Note that, although we use a BooleanCombiner at each node in order to implement hSAT, the framework is more general than that. By writing a different subclass of Combiner and replacing a few other code modules, one can implement other similar near-hierarchical problems. Naturally, a good solution heuristic for the node-level problem needs to be included in the new subclass of Combiner - but the solution code which directs inter-node activity remains valid for different Combiners, as long as they respect the data types and method signatures used in the Node subclasses.

Next, we look at details of our experimental setup, and at the results of several experiments.

7. hSAT: Experiments

“To measure is to know.” - Ernst Werner von Siemens

In this chapter, we look at a series of experiments on hSAT.

Keep in mind that our goal is twofold: to understand hSAT, and to use hSAT as a testbed to understand Neartrees, and hierarchical problem-solving in general. Perhaps this is an opportune moment, therefore, to review why we will be using SAT as the basic problem to be solved at each node:

- Speed: Remember, we want to solve not just one problem, but hundreds or thousands of non-trivial problems in parallel. If one wants to do statistics and explore a variety of parameter settings, hSAT lies on the outer boundary of what is currently feasible computationally.
- SAT is complex enough to be useful, as the applications we’ve listed show.
- Our hSAT model seems quite a natural model of hierarchical systems, and a natural extension of a well-studied non-hierarchical problem. We propose it as a minimal set of axioms for capturing important aspects of hierarchical systems in an unambiguous computational model.
- Synthetic models are transparent, which makes it easier to see what works and figure out why. If they can be parameterized by difficulty, so much the better.

Although the SAT language is simple, it is complex enough to possess many properties of more realistic systems. An example is "backbones", which are the set of variables which must be assigned to a specific value in every satisfying assignment - [Achlioptas et al 2000] show a phase transition seems to occur as the fraction of backbone variables goes from 100% to 0, with the hardest problems (in the sense of taking the most time to solve satisfiable instances) occurring in the middle of this range. [Slaney & Walsh 2001] show that the notion is more generally applicable to other problems in approximation and optimization as

well. The study of SAT still likely holds many surprises in store.

7.1 Experimental Setup

Test Cases: Benchmarks and Random Formula Generation

There are several choices for specifications and classes of SAT problems:

- Easy SAT classes like 2-SAT. These are classes for which a reasonably fast polynomial time algorithm already exists - hence we already know how to solve them.
- Ratio of clauses to number of variables (i.e. formula density). This is a commonly used measure of problem difficulty for randomly-generated SAT formulae, and one that we will use.
- More complex specifications of hard SAT problems, that can better tune the difficulty level. While these would be nice to have, there is no general agreement on such specifications yet. (Note that backbones are a property requiring complete knowledge of the solution set to measure - as opposed to clause-to-variable ratio, which is a syntactic property that is easy to measure.)
- Using benchmark problems for the local SAT problem to be solved at each node. (Many such benchmarks exist, e.g. see the SAT2002 proceedings.) We have tried using some benchmark problems as well - specifically, test cases from the widely-used DIMACS benchmark set.

Note that generating random satisfiable instances is nontrivial. The straightforward way to generate random CNF formulae with a given density is to simply pick three distinct variables at random for each clause and assign each a random polarity (negated or un-negated). However, the phase transition phenomenon implies that such random CNF formulae are almost all unsatisfiable above some characteristic density, thought to be around 4.3 (and proven to be less than 4.6). Although one can filter the set of such formulae by

generating many instances and keeping only those which can be proved to be satisfiable via some complete SAT-solution method like DPLL, the difficulty of solving larger problems implies that this can't be done for very large formulae.

One might therefore wish to generate random formulae which are difficult (have high density) but are known to be satisfiable. A natural method for doing so is to generate “forced formulae”: pick a random assignment of truth values to variables, and then generate the appropriate number of random clauses, rejecting those which do not satisfy the variable assignment. While this does generate a satisfiable formula with the appropriate number of clauses, these formulae have been shown to be easier to solve than those generated by filtering.

This problem of systematic generation of hard test instances is a major open problem in the constraint satisfaction community, in which slow progress is being made such as the formulae derived from mathematical functions of [Achlioptas et al 2000]. In future work, we would like to develop at least one other hSAT problem parameter family - preferably representing a domain of practical interest.

Key Parameters, and Adaptive Parameter-Space Exploration

Our Neartree SAT explorer has many parameters which can be varied. As well, it can be used for several distinct purposes - and potentially linked to sources of variation or reward, which themselves are variable models. In order to draw useful conclusions from this experimentation, therefore, we need to i) clearly specify the “dimensions of variation” of our model, including an approximate assessment of their importance and suggestion of some natural values for the dimensions to take, and ii) run the model on a representative set of parameters, to get an idea of its behavior under different circumstances and from that behavior draw more general conclusions.

With respect to the Neartree structure, we have several straightforward variables from

which to start parametrically specifying classes:

- c - bound on number of children per node.
- p - bound on number of parents per node.
- L - maximum level number of Neartree.
- Formula density - This is a standard parameterization of SAT problem difficulty for CNF formulae, which we use as our node-level problem difficulty measure.

The dependent variables that we measure include quality of solution and time to find a solution. “Quality of solution” is measured in two basic ways: percentage of formulae found satisfiable for the SAT case, and closeness to satisfiability for the MAX-SAT case.

Since we are sampling from a stochastic distribution, it is essential to keep issues of statistical accuracy in mind. In particular, increasing the number of samples taken will decrease the size of the confidence intervals, and hence increase the accuracy of and confidence in results. Note that, for heavy-tailed phenomena, i.e. those for which the probability density drops off relatively slowly away from the mean value, it is often better to use the median rather than the mean as a statistic; see [Gomes et al 2000] for discussion of such phenomena in SAT. (We use the mean since our tails are truncated via a time bound.)

Typically we will be varying one independent variable and checking the effect on a dependent variable. Sometimes it is of interest to guesstimate the value of the independent variable past which the dependent variable has converged, e.g. solution quality with increasing computational resources. If we were computing a deterministic function, it would be a simple matter to define some convergence interval, and check when the progressive increases (or decreases in the case of minimization) of the dependent variable as the independent variable increases fell below some threshold value.

However, there are two complicating factors in our case. First, since we are taking

statistics over an ensemble of functions generated from a probabilistic distribution at each parameter vector value, there is some error involved (which is typically a function of the number of samples taken, although other factors also play a part as we will see). Second, due to the hierarchical nature of our problem domain and the consequent rapid increase in solution time required as problem difficulty increases, it is highly desirable from a practical point of view to minimize the number of samples and parameter vector values taken for difficult problem instances, especially if the dependent value of interest has already converged modulo stochastic error.

We deal with this issue in three ways. First, a hands-on approach to data exploration was taken - we looked at the data sets intermittently as they were being generated, over a period ranging from hours to a week depending on the data set. One can often see patterns and infer appropriate parameter vector values to try this way, along with catching subtle program errors. Second, we implemented a simple test for stochastic convergence: a sliding window of size 3 was taken over the last 3 data points (d_1, d_2, d_3), and whenever both d_2 and d_3 were less than or equal to d_1 (in the case of maximizing the dependent variable), a flag was set in the program indicating probable convergence. This works because past some independent variable value $t_{\max}$, the dependent variable D has reached some maximum m in the underlying process, and past that point changes only due to stochastic fluctuations:

$$d(t) = D_{\max} + \text{error}(t), \text{ if } t > t_{\max}$$

(where t is the independent variable). One could get a more accurate test by using a larger window size, but because samples become rapidly more costly as t increases, this is a good compromise in practice; since samples are independent, one can always go back and sample at higher values of t after examining the automatically-generated data set, if it is felt to be necessary (as we did a few times). So we have a simple method of doing some adaptive exploration in parameter space automatically - every experiment is given a cutoff time, and when the time taken gets too large or convergence appears to have been reached, the program can automatically move on to the next parameter setting.

Third, since samples are independent trials, hSAT is ideally suited to “sampling exploration”. We usually did a “vertical sweep” across the graphs we generated - first taking all the samples for the first independent variable value t_1 , then taking all the samples for the next independent variable value t_2 , and so on. This is a good strategy if one wants to get statistically valid results, as shown in all the graphs in this thesis. However, an alternative strategy is useful for real-time exploration (or as close to real-time as one can get when optimizing thousands of functions simultaneously). This is the “horizontal sweep” - take one sample for all the independent variable values t_i , then take the next sample for all independent values t_i , and so on, graphing the result every k samples. Although the results will not reach statistical validity for a long time (since accuracy increases as only the square root of the number of samples), it is emotionally satisfying to see even approximate results quickly, and one can often rapidly spot anomalies and see the general shape of the results - a boon for exploratory work.

7.2 Experiments in SAT and WalkSAT

Experiment: A First and Second Look at Traditional SAT

To validate the SAT solver, we first try it on a distribution of randomly-chosen 1-level problems (i.e. traditional SAT problems). The problems are parameterized by the clause ratio (higher means more difficult, since each clause is a constraint).

```

for  $c = 16, 64, 256, 1024$  {
  for clauseRatio = 1 to 5 step 0.25 {
    for 100 random instances {
      spend 10 seconds solving this instance
      record SAT or unSAT (i.e. whether satisfying assignment was found)
    }
    record sample success probability for this value of  $c$  and clauseRatio
  }
}

```

}
}

There are several features to note in this experiment. First of all, it is quite evident that there is a transition in the likelihood of solvability of formulae as the clause ratio increases; this likelihood goes from near 1 to near 0 in a relatively short span. One can also see that this transition gets sharper as the number of variables increases. Both of these are well-known (although incompletely understood) phenomena in 3-SAT.

Figure 6...A first, flawed look at SAT

Next, there are two sources of experimental error in the graph as given: the error due to sampling a limited set of formulae at each given length, and the error from possibly not being able to solve a formula that is really satisfiable. The first source of error, sample error, can be analyzed statistically by considering each instance as an independent binomial trial. (This is justified since each problem instance is drawn independently, and since a problem instance from the set of 3-SAT problems with a given length and clause ratio has a well-

defined probability of being satisfiable.)

It is well-known that the mean μ and variance σ^2 of a binomial distribution (n, p) (i.e. with probability p of success at each trial and n trials) are np and $np(1-p)$ respectively. It's clear that σ^2 is maximal when $p = 1/2$. In this case, and with $n = 100$ since we did 100 trials per parameter vector value in this first experiment, $\sigma^2 = 25$ and $\sigma = 5$.

Since the binomial distribution rapidly approaches a normal distribution for large values, a 95% confidence interval for the true value of our observed data points is the sample mean $\pm 2\sigma$, or $\pm 10\%$ in the scale of the y-axis of the graph given. (This is for those values of CR and c for which the probability of success is around 50%; the confidence interval gets smaller to either side, since σ decreases as p moves away from $1/2$.) Thus we need to be careful about drawing inferences from experiments if the pattern seen is minor with respect to the confidence interval - a lesson which will be kept in mind in all future graphs.

It is straightforward to derive a formula for the number of samples that should be taken, in order to achieve a desired standard deviation. Setting V as the desired standard deviation (expressed as a fraction of the total number of sample), we can take the formula for variance of a binomial distribution and divide by the total number of samples to express an identity, and then solve for the required number of samples:

$$\begin{aligned} V &= \text{variance}^{1/2} / \# \text{ of samples} \\ &= (np(1-p))^{1/2} / n \\ &= (n/4)^{1/2} / n \quad (\text{for the worst case where } p = 1/2) \\ V^2 n^2 &= n/4 \\ n &= 1 / (4V^2) \end{aligned}$$

Hence we have e.g. $V = 1\%$ implies $n = 2500$.

The second source of error comes about because the method we use is heuristic in nature.

Although it is much faster on the average than many complete methods, the lack of a specific time bound means one can never be 100% sure that more computation will not solve the problem. (This puts it in the computational complexity class RP.) To check how much of a factor this was in these experiments, an additional data point was calculated: $c = 1024$, $CR = 4.25$, with 60 seconds instead of 10 seconds per run, and 1000 runs instead of 100. (Note that 1000 runs and $p \approx 1/2$ leads to $\sigma \approx 250^{1/2} \approx 16$, for a 95% confidence interval of just over $\pm 3\%$.)

The result of this calculation was a success rate of 42.5% - plotted on the graph as an “x”. Compare this with the point directly below, which shows a success rate of 28% from using a shorter calculation time for the same number of variables. Clearly this suggests that, at least for formulae of the order of 1000 variables, calculation time is a significant issue. We therefore need to do experiments to get more insight into this effect, by holding all variables fixed except for calculation time - an issue explored in the next section.

Consider now the same graph, redone with 2500 samples per data point and a resourceBound of 2.

The greater number of samples smooths out most of the curves, as one would expect. It is also readily apparent that increasing the size of the formula results in a progressively sharper transition from solvability to unsolvability - a phenomenon we explore again later.

Figure 7...A better look at 1-level SAT

Experiments: Tuning the Number of Restarts in WalkSAT

This experiment and the next look at the solution method, to see how changing parameter values affects the solution efficacy. Parameters that one could tune include the computation bound for optimizing at each node, looking for a better target set for your child nodes vs trying to achieve those targets, breaking local minima, and so on. Since it is not feasible to test every combination of values, we will fix many of these at values that empirically seem to work well.

In this experiment, we tune the number of restarts in WalkSAT. The WalkSAT algorithm solves the SAT problem by doing a combination of a random walk with several

heuristics designed to preferentially choose variable changes likely to lead toward a solution. One key parameter in the algorithm is the number of restarts; once the maximum number of steps has been taken and a solution has not been found, one may wish to reinitialize the solution and try again, in the hope of exploring a different part of the solution space. This heuristic helps to avoid local optima.

We'll do this for 64 and 256 variables (i.e. $c = 64$ and 256 , $L = 1$). The clause ratio is fixed at 4.25 - this is in the middle of the transition region where about half of the problems are solvable and half are not, giving a good test region for experiments.

```
fix c = 64, 256; fix maximum resourceBound (i.e. # of solution steps)
for restarts = 1 to 8 {
  for 10 000 random instances with CR = 4.25 {
    spend resourceBound solving this instance
    record SAT or unSAT (i.e. whether satisfying assignment was found)
  }
  record sample success probability for this value of c and restarts
}
```

For the larger case of 256 variables, we see that the probability of success shows a definite rise at first, but then seems to fluctuate past restarts = 3 or 4. Note that 10 000 random instances gives $\sigma = 0.5\%$, so the results are quite accurate (within 1% of the data point shown, 95% of the time).

Figure 8...Tuning Restarts in WalkSAT ($c = 256$)

Keeping in mind that there is one underlying true probability of formulae of this size being satisfiable, we can interpret the results as follows: at first, adding restarts helps the algorithm to find more solutions. However, past 3 or 4, adding more restarts (and hence more computational resources) has no effect, meaning that the algorithm has converged to optimal performance within experimental error.

If this was not the case, we would expect to see a continued rise in success probability, instead of a seeming fluctuation about some value. Clearly, adding more resources can never make an algorithm's expected performance worse - at least not this algorithm in RP, where success comes whenever an optimum is first reached. Therefore, if the success probability falls with increasing resources, this can only be due to sampling error, since all other variables are controlled for. When sampling error overwhelms the effect of adding more resources, it means that that resource level is the largest one for which we can observe statistically valid results.

Here we see a similar experiment, for $c = 64$. The smaller experiment size results in a faster running time, allowing us to try the experiment with 10 000 samples at two different resourceBound levels. Note that the maximum computation time required for each sample also increases linearly with the number of restarts.

Figure 9...Tuning Restarts in WalkSAT ($c = 64$)

(Recall that resourceBound is a measure of the amount of computational resources available for each individual sample, i.e. the resources available to try to solve the same problem between restarts. We will discuss precisely what the resourceBound levels mean in the next section; for now, note that a higher r value means more resources, so $r = -4$ represents more resources than $r = -5$ - twice as many resources, in fact.)

Here we see again a definite increase in solution probability at first, which is more pronounced at the lower resourceBound level. However, again the fluctuation effect appears - past 4 or 5 restarts, the solution probability fluctuates, showing that we have reached the success limit and are now in the regime of experimental error.

If we compare the series for $r = -4$ with that for $r = -5$, it is clear that at every point on the x-axis, the former series has a higher solution probability - this is as we would expect, given that the former series represents a solution series with greater computational resources.

Experiments: Tuning Computational Resources in WalkSAT

Now we look at the actual computational resources used by each experiment.

There are two candidates for measuring resources used: a bound on clock or CPU time, and a bound on the number of basic algorithmic operations performed. While the former is more convenient to use experimentally since it makes the time for each experiment straightforward to predict, it has the disadvantage of not being portable: the same amount of CPU time on two machines of differing power represents different computational resources.

We therefore use the second method for bounding computational resources in most cases. The specific algorithmic bounds are on i) the number of transitions between candidate formulae done in WalkSAT, and ii) the number of cycles of the overall Neartree algorithm (where each cycle represents an optimization at each non-leaf node, along with ancillary and Neartree-wide operations). The second bound is only necessary for hSAT formulae with $L > 1$, so we discuss it further in the hSAT sections.

In order to measure the number of transitions meaningfully, we chose to use a logarithmic scale for resourceBound. This facilitates comparison of results across a wide range of possible bounds. The scale we use has each successive unit interval representing a doubling in the number of transitions allowed. "0" on the scale is $2^{16} = 65536$ flips, a default we used in some experiments. Numbers below or above 0 then represent powers of two below or above 16; for example, $r = -5$ represents $2^{16-5} = 2^{11} = 2048$ transitions.

We do the experiment for 64 and 256 variables (i.e. $c = 64$ and 256 , $L = 1$). As with the

experiments on restarts, the clause ratio is fixed at 4.25 which is empirically a good test region (since about half the formulae are solvable, and it's difficult to prove unsolvability in the region via complete methods). The number of restarts is fixed at 3, as from the previous section this gives a good compromise between solution efficacy and computation time.

```
fix c = 64; fix restarts = 3
for resourceBound = -13 to -1 step 0.5 {
  for 10 000 random instances with CR = 4.25 {
    spend resourceBound solving this instance
    record SAT or unSAT (i.e. whether satisfying assignment was found)
  }
  record sample success probability for this value of c and resourceBound
}
```

Here we have a very interesting result: the graph of solution success follows a very well-defined sigmoidal curve. Note that the resource bound on the x-axis is a logarithmic scale - if we were using a simple proportional scale, this sigmoidal shape would not be evident, since then the distance along the x-axis between the points shown here would double between every two points.

Figure 10...Resource usage in WalkSAT ($c = 64$)

The appearance of such a regular sigmoidal shape is certainly evidence for some definite algorithmic property at work - the results are statistically strong with a σ of 0.5%. We suspect that it may be explicable by considering WalkSAT as a sampling process in the space of all possible boolean formulae of a certain length. At first, only a small neighborhood of the random starting solution is seen. Then, there is a gradual increase in the rate up to some limiting value, which is maintained until a large fraction of the search space has been sampled. Finally, the rate increase slows, and the success rate asymptotically approaches some limiting value.

While this explanation seems plausible, future analysis is needed to determine why the curve takes this specific shape, showing such a regular sigmoid when the x-axis is logarithmic.


```

fix c = 256; fix restarts = 3
for resourceBound = -8 to 4 {
  for 2 500 random instances with CR = 4.25 {
    spend resourceBound solving this instance
    record SAT or unSAT (i.e. whether satisfying assignment was found)
  }
  record sample success probability for this value of c and resourceBound
}

```

For $c = 256$, we see a similar sigmoid curve as with $c = 64$, giving additional confidence that the results represent a consistent phenomenon in WalkSAT when increasing the resource bounds.

Figure 11...Resource usage in WalkSAT ($c = 256$)

Computational limits dictated that fewer samples were made than in the more tractable $c = 64$ case: - 2 500 vs 10 000, and resourceBound sampled at intervals of 1 instead of 0.5. Note that, since the scale is logarithmic, each successive point in this graph requires about twice as long to compute - and that type of exponential growth can't be sampled very far.

Also, going to $c = 256$ means that substantially higher values of resourceBound are necessary to reach the top performance value: $r = 1$ in the $c = 256$ case, versus about $r = -5$ in the $c = 64$ case. This represents a factor of $2^{1-(-5)} = 2^6 = 64$ times more computational steps required when c is increased by a factor of 4. Finally, each computational step takes longer to compute when c increases. The result of these factors is that $c = 256$ is an approximate feasible limit for statistically valid 1-level SAT computations in reasonable time on a PC.

If we go to e.g $c = 1024$ or 4096, what we see is the same kind of sigmoid curve as with lower values of c . However, the limit of infeasibility is reached while the curve is still rising (in contrast to lower values where the curve can adequately be solved until it converges). In the case shown here, we graph $c = 1024$, with 2500 samples per data point.

Remember - each successive incremental increase in the x-axis variable resourceBound implies a twofold increase in computational effort. At $r = 6$, the curve is still rising, and it will clearly take at least until $r = 7$ to reach convergence. This represents 2^{16+7} or 8 388 608 flips per try; with the value used of 5 restarts per sample, and 2500 samples per data point, this results in an estimate of at least 10^{11} variable flips to get a reliable data point for $r = 7$.

Figure 12...Resource usage in WalkSAT ($c = 1024$)

(With $c = 1024$, a PC circa 2002 might be able to generate and evaluate up to 10^5 flips per second, resulting in a time of a couple of weeks for this single data point. Obviously this can be improved by using more hardware or waiting a few years, but the exponential nature of the resource increase required limits the usefulness of this strategy.)

We can combine this estimate with the previous required resource values observed for the cases $c = 64$ and $c = 256$ to get an approximate formula:

Table 4...Extrapolation for required resources in 1-level WalkSAT				
c	$64 = 2^6$	$256 = 2^8$	$1024 = 2^{10}$	2^k
r	-5	1	7	$-5+3(k-6) = 3k-23$

This demonstrates the necessity of a more efficient approach - of which our hSAT model is an example, for an important subset of SAT problems. (For a complementary approach to showing the benefits of hSAT, see the section “Expanding hSAT into SAT”.)

7.3 Experiments in hSAT

Experiment: Two-Level hSAT

With this experiment series, we move from traditional SAT to hSAT. For the first experiment, we look at 2-level hSAT, since this is the simplest case beyond SAT. This will help build our intuition for further experiments.

Clearly, increasing either L or c makes an hSAT problem more difficult. Increasing c makes both node-level problems and the overall Neartree solution harder, while increasing L makes the overall Neartree solution harder, and also adds to the number of nodes. In fact, for a given value of c , a homogeneous Neartree will have c^L leaf nodes - so even in this two-level case, the total number of nodes in the Neartree (and hence memory requirements) grows quadratically with c .

We can fix the parameters of restarts and resourceBound to the values which we empirically found to work well in the previous experiments. For c up to 64, resourceBound = -5 is a good value; 3 restarts seems a good compromise value between speed and effectiveness.

Note that we need one additional parameter, which tells how many overall cyclic Neartree solution cycles are allowed. Recall that, in a problem with $L > 1$, we have a cyclic solution process where nodes first solve for desirable states of their child nodes and pass these desirable states down as requests, sweeping down through all levels. Then nodes choose values to take on which are passed up to parents in an upward sweep through the

Neartree (along with additional information).

We call each complete cycle a “tick”, and need to place a bound (“maxTicks”) on the number of ticks allowed. For this first experiment, we used maxTicks = 5.

fix $L = 2$, $restarts = 3$, $resourceBound = -5$, $maxTicks = 5$

for $c = 4, 8, 16, 32$ {

 for clauseRatio = 2 to 5 step 0.2 {

 for 2500 random instances {

 spend timeBound solving this instance

 record SAT or UNSAT

 }

 record median / mean / range of results

 }

}

Graph curves by clauseRatio, % SAT, and c

There are several points to note in this graph. First, as with the 1-level case, increasing c causes the transition (from satisfiable to unsatisfiable as clauseRatio increases) to become sharper. (It is not, however, clear whether this is really a phase transition in the sense that higher values of c approach a step function - this is an open question.)

Figure 13...Two-level hSAT

Second, as compared to the 1-level case, the values of clauseRatio for which the formulae become unsatisfiable are lower. For example, with $c = 16$ in the 1-level case, we have 50% of the samples satisfied at clauseRatio around 4.7 or 4.8; with $c = 16$ in the 2-level case, 50% of the samples are satisfied at clauseRatio around 3.9. We can partially understand this by noting that, in the problem formulation for this experiment, a node is satisfied only if all of its child nodes reach their target states; clearly this reduces the clauseRatio at which some nodes will not be satisfied.

Third, the computational resources required to deal with these Neartrees grow as a function which is quite sensitive to L . Computing the $c = 16$ curve for $L = 3$ will require a 4096 leaf node Neartree (and 4369 total nodes); for $L = 4$, 65 536 leaf nodes and 69 905 total

nodes. How does this square with our previous assertions that one of the advantages of using a hierarchical decomposition is supposed to be the savings in computational resources?

Well, although computational resources required for increasing L are substantial, they need to be kept in perspective. Computing a single 70 000 node problem, if that problem is the same difficulty level (i.e. as parameterized by `clauseRatio`) as all the individual 16-node problems in our $L = 4$ Neartree, will be vastly more difficult. (Of course, the reason that problems of such large sizes can be solved practically is that they have structure which makes them relatively easy, as compared to the set of all problems of that size.)

Experiments: Three-Level and Four-Level hSAT

We now go beyond two levels, to three and four level hSAT.

One can note several features in the three-level graph (Figure 14). Since the data points were sampled with 400 samples per point, the curves are noticeably less smooth than in the 2500 and 10000 sample cases. (400 samples were used for many of the multilevel and multiparent experiments, due to computational constraints.)

There still appears to be a phase transition, but it is taking place at a lower value of `clauseRatio` - perhaps around 3.5. This accords with our intuition that multilevel problems will pose more of a challenge.

Figure 14...Three-level hSAT

In fact, it may be a bit of a surprise that the algorithm works as well as it does at such a relatively high value for clauseRatio. Keep in mind that a 3-level problem is only considered solved if satisfactory solutions can be found for the BooleanCombiners at every one of the non-leaf nodes (and there are $32^0 + 32^1 + 32^2 = 1057$ such non-leaf nodes in the $c = 32$ case).

Consider now the four-level case - our upper limit due to computational feasibility.

Note that when one gets up to four and more levels, both the total number of nodes and the time to compute an overall result rise very rapidly as c increases by only a small amount. We were only able to investigate up to $c = 12$ due to this factor - while higher values of c up to 24 would have been feasible in terms of memory requirements on our 1 GB experimental machine, the time required became prohibitive past $c = 12$.

We again see that the clauseRatio value for which about 50% of the formulae are solvable appears to have dropped somewhat. This probably continues to drop as c increases, so the actual limiting value is likely somewhat lower (and needs further experimentation to establish).

Figure 15...Four-level hSAT

It still appears to be the case that three and four level problems give rise to solution curves with a clearly sigmoidal shape. While this may accord with intuition, it is not necessarily obvious that this should be the case, and these experiments are to our knowledge the first experimental confirmation of this general behavior for hSAT.

This concludes our initial experimental foray into the demesnes of hSAT. In succeeding chapters, we will build upon this work, and look at theoretical and practical connections.

8. Multiple Parents

“Do not let what you cannot do interfere with what you can do.” - John Wooden

8.1 Beyond Trees

When each node has but a single parent, the goal state to be achieved from the point of view of that node is simple: just try to get to whichever state your parent wants you to be in. However, with multiple parents a myriad of complications arise. To take the most apparent difficulty, consider a node with two or more parents. Such a node may receive requests from all parents to go to the same goal state, in which case the problem reduces to the single-parent case - or it may receive conflicting requests.

This latter scenario is of course where difficulties arise. If I’m asked to do seemingly conflicting tasks, how can I reconcile the requests in a maximally satisfactory manner?

We also consider related issues of constructing a multiple-parent Neartree, and of designing caching protocols to speed up computation. Finally, we explore modeling issues and take an initial mathematical foray into characterizing the behavior of multiple-parent systems.

Generating Multiple-Parent Neartrees

Generating the parent-child links for a complete Neartree with $p=1$ is easy - this is just an ordinary tree with fanout c . How do we generate the nodes and links for $p > 1$?

What we would like, from the definition, is for the following equality to hold:

$$p |_j H| = c |_{j-1} H|$$

i.e. the number of parents at level j should be equal to the number of children at level $j-1$. (This must be true, since links are symmetrical.)

This in turn implies that, if the c and p upper bounds were achieved at every node, we would have:

$$\begin{aligned} |_jH| / |_{j-1}H| &= c / p \\ |_jH| &= |_{j-1}H| c / p \end{aligned}$$

Two observations are in order. First, this cannot hold for the topmost pair of levels ($j=0$ and 1) if there is only one root node. (We are sticking with the single root node restriction in the current work, although multiple root nodes could make sense in extending Neartrees to deal with multiobjective optimization.) To deal with this, we simply override the Neartree's parameter p with a stricter upper bound at level 1 , and set $p=1$ at $_1H$ for all Neartrees.

Second, the quantity c / p may not be an integer. (This problem of course never arises for a tree, since $p=1$ in that case.) To deal with this, we modify the definition slightly by rounding down the quantity on the right:

$$|_jH| = \lfloor |_{j-1}H| c / p \rfloor$$

This has the effect of reducing slightly the level sizes from what might be achieved if c / p was integral; e.g. for $c=3$ and $p=2$, the sequence of level sizes is $(1, 3, 4, 6, 9, 13, \dots)$. We then generate exactly p parents for each node in $_jH$ for $j > 1$, by linking $_jH_k$ with the leftmost p nodes in $_{j-1}H$ which do not already have their maximum c children. With this scheme, every node at levels > 1 has exactly p parents, and every node at levels $< L$ has either c or $c-1$ children. This gives us a closest approximation to a “complete Neartree” (i.e. one which comes as close as possible to achieving its c and p upper bounds.)

For illustration purposes and to help develop a feel for the sizes of some multiple parent Neartrees, let's consider some examples:

Table 5...Multiple-Parent Neartree Sizes				
Level #	p=2, c=3	p=2, c=4	p=4, c=12	p=8, c=32
0	1	1	1	1
1	3	4	12	32
2	4	8	36	128
3	6	16	108	512
4	9	32	324	2048
5	13	64	972	8192
6	19	128	2916	32768
7	28	256	8748	131072
8	42	512	26244	524288

Conflicting Parent Requests, and Parent Models of Children

Multiple parents may generate conflicting target state requests for a common child. In this case, a **request collision resolution protocol** is necessary. This protocol decides to which (if any) parent requests a child should listen.

One straightforward heuristic is as follows:

```

When target requests from parents change, or are initialized {
    Set my target as the target requested by the greatest number of parents.
    Tell the other parents they're out of luck this time.
}

```

Several more sophisticated variants are possible. If parents have some measure of how much they desire a child to be in a certain state, they may attach a bid with their request; in this case, the heuristic above is changed to choose the target with the maximum total bids from parents. If target requests change frequently, a damping factor may be introduced so that the threshold for changing targets is greater than 50% of parents; this would prevent flickering behavior back and forth between target states, in some cases.

We have used the basic heuristic above in all of our multiple-parent experiments - without bids, but with damping in some cases.

In order to better decide what is reasonable to request of child nodes, parents should **model their child nodes' behavior**, to make better choices about which states to ask for. Possibilities include:

- Correlation between values of child states. (And more generally, Estimation of Distribution algorithms, where the general joint distribution is learned.)
- Frequency of change of child states.
- Speed of progress toward requested value.
- Incentive that child has to reach each state requested by at least one parent. (This might lead to bidding wars or thrashing, if parents try to top each other for a crucial child.)

These heuristics are like a sophisticated version of backtracking, optimized for the hierarchical case.

We have used some of these heuristics in our implementation. In particular, every parent keeps track of both the **state that its children are in**, and the **last requested goal state** that it asked of each child. These are stored in a data structure we call a “StateVector”.

Along with states, StateVector is extended to a subclass “StateVectorWithID”, which stores an associated integer for each state stored. This forms a computationally efficient way

of tracking the number of ticks since an event of interest took place. For the StateVectorWithID storing child states, the ID stores the number of ticks for which that child's state has remained unchanged. The parent can thus track which children seem to have converged in state, and which are still changing rapidly.

For the StateVectorWithID storing requested goal states for each child, the ID stores the number of ticks for which that child has not achieved its requested goal state. So, each parent of a child can request that child to try to achieve a specified goal state (T / F in the case of hSAT, but any state in general).

From the point of view of the child, it may receive several of these requests, and will have to choose which to listen to. But from the point of view of a parent, each child needs to be in a state that, together with all the parent's other children, forms a satisfying combination of states of the parent's BooleanCombiner.

So the parent is interested in seeing which children have **difficulty reaching their goal states**, in order that it may for example consider working around them (as discussed in the caching section). Thus the StateVectorWithID storing requested goal states for each child has an ID which is reset to 0 whenever the parent's request for that child changes, and incremented for each tick that the child is not in the parent's requested state.

Multiple-Parent Heuristics and Caching

Let's discuss how the overall control flow works with multiple parents.

In moving to the more complex multiple-parent case, fulfilling all requests becomes more difficult due to conflicts. Hence it may be more important to observe nodes and their empirical behavior, with regard to e.g. instability (frequency of state change), estimated room for improvement of value, and degree of contribution to value of upper-level nodes (both individually and in combination with other nodes).

Let's now consider caching schemes. A simple choice is "first-time cache", where a node caches just the first solution found for its BooleanCombiner, for each possible goal state. This empirically performs quite poorly. The reason is not hard to understand: children may not be able to reach the desired solution. First-time cache is a completely non-adaptive method in which parents don't observe their children to see how well the children are achieving their assigned target states.

We have explored two alternatives: **periodic-resolve cache** (where new solutions are searched for at specified intervals, if children have not managed to reach the old solutions), and **targeted-resolve cache** (where children which can't reach a requested states are tracked, and an attempt made to find solutions which accommodate those children).

Of these, periodic-resolve cache gives the best empirical performance. On several benchmark problems, it gives a percentage of formulae solved comparable to any other approach we tried, at a substantial speedup. Each level's nodes will wait several ticks to see if their children reach their targets; the number of ticks is a parameter, `SolveTickRatio`, that can be varied. If children are not all successful, the nodes search for new solutions and assign those. (Note that this naturally gives a time scaling between levels, with a solution reassignment at $\phi_i H$ happening about SolveTickRatio^i times less frequently than at $\phi_j H$.)

Targeted-resolve performed surprisingly poorly, for reasons we do not yet fully understand. It tracks which child nodes are having trouble reaching their assigned state, calculates an alternate BooleanCombiner with these children set to the value they have already reached (thus seeing if the other children can find an alternative solution that accommodates children having difficulty), and assigns the alternate BooleanCombiner's solution to all children, if found. Although its more sophisticated approach suggests that it should find solutions that the other approaches miss, it may be that by being too focused on optimizing it is missing out on stochastic exploration, which is necessary to find certain solutions serendipitously.

8.2 Experiments: Multiple Parents

Let's look now at the experiments on multiple parents.

There are several implementation issues that arises in multiparent formula generation. One, which is also important to a lesser extent in the single-parent case, is that it is necessary to interpolate the actual number of clauses in the BooleanCombiner of a node, given the values of c and $clauseRatio$ at that node. For non-integer values of the quantity ($c * clauseRatio$), this makes a difference in the outcome, particularly for small values of c - without interpolation, small values of c give a “step effect” where increasing $clauseRatio$ smoothly gives discontinuous jumps in the actual number of clauses in the BooleanCombiner.

But how does one add, say, $3/7$ of a clause to a Boolean formula? The approach we take is to use the fractional portion as a probability of rounding up or down, using the following algorithm:

```
given  $c$ ,  $clauseRatio$ 
 $rawNumClauses = c * clauseRatio$ 
 $randNum =$  a random floating-point number between 0 and 1
if ( $randNum < (rawNumClauses - \lfloor rawNumClauses \rfloor)$ )
     $numClauses = \lceil rawNumClauses \rceil$ 
else
     $numClauses = \lfloor rawNumClauses \rfloor$ 
```

It is easily verified experimentally that interpolation works and avoids discreteness artifacts. For example, taking a 2-level experiment with $c=8$ and a large number of samples (10000), increase the $clauseRatio$ from 2 to 3 by steps of size $1/16$. With interpolation on, one sees a smooth decrease in solution probability. With interpolation off, there is a

stepwise decrease in solution probability - with the probability corresponding to each successive overlapping pair of CR values ($2^{k/16}$, $2^{(k+1)/16}$) being alternately equal to within experimental error and taking a big downward jump.

Our first experiment looks at the simplest multiple parent case - two parents, two levels. (Clearly a 1-level problem cannot have multiple parents, since there is only one root node.)

Figure 16...Two-parent, two-level hSAT

For this experiment, we looked at a number of different values of c : 5 values of c from 4 to 64, going up by powers of two. We used 400 samples per point.

It is immediately apparent that behavior in this case is qualitatively different from the single-parent case. (Compare with the previous graph for two-level single-parent hSAT.) As the value of c increases, we see a dramatic drop in the clauseRatio values for which these problems are solvable. In contrast with the single-parent case, there does not appear to be

any phase transition value in multiple-parent cases; rather, it appears as if increasing the size of each subproblem (as represented by c) simply makes the overall problem more and more unlikely to be solved, without any lower bound on the difficulty of problems that can be solved.

This point can be noted by carefully looking at the values of `clauseRatio` shown on the x-axis, and comparing them with previous graphs. We started plotting data points at a `clauseRatio` of 0.2 - an extremely low value, which represents a light level of constraints on the possible values that children can take.

Since each clause in 3-CNF is a constraint on exactly 3 variables, a `clauseRatio` of 0.2 means that there are a total of $0.6 * c$ variables mentioned in all constraints, applied to the c variables - so almost half of the variables have no constraints whatsoever on them at this level! This phenomenon needs closer investigation. We conjecture that there may still be a phase transition value of `clauseRatio` in this case, but at a low value, i.e. at $1/3$ or a little below. The intuition is that formulae below this value are still solvable for all c , but as soon as there start to be a “significant” number of variables which are constrained by two or more parents, it may become very unlikely to satisfy all the constraints if there are a large number of children.

So, we see that even with only 2 parents, large problems become quite difficult to solve. (Note, however, that these experiments only assert lower bounds on the percentage of formulae at a given `clauseRatio` that are solvable - it may well be the case that our multiple-parent algorithms can be improved to yield better results at the cost of more computation, and in fact we are exploring ideas in this regard.)

Next, we look at extending the number of parents beyond 2. We fix c at 16, and let p take on the values 2, 4, and 8. In all cases, 400 samples were used per point.

The case for $p = 2$ gives a relatively smooth sigmoidal curve, albeit at lower values of

clauseRatio than we have become accustomed to from the single-parent case.

However, at $p = 4$ and $p = 8$ there is a surprise in store. The case $p = 4$ is similar to what

Figure 17...Multiparent, two-level hSAT

one might expect - note that the first data point is at clauseRatio = 0.2, and the curve does go up to reach 1 as the clauseRatio is decreased toward 0. But $p = 8$ actually appears to be better than $p = 4$. How can this be the case? Surely adding more parents should make problems more difficult to solve, and not easier?

Although the difference is slight, it is definite and has been verified through repeating the experiment. The feature may be due to one of two causes: first, it may be an unintended side effect of how we have chosen to construct or solve multiple-parent Neartrees, although we have not been able to determine how that could be the case.

Second, it may be a genuine feature of large numbers of parents. Recall that, as p increases with c held constant, l_{avg} will decrease. This means that with $p = 8$ as compared to lower levels of p , there are a much smaller number of leaf node variables. Perhaps having fewer of these variables to deal with somehow makes up for the greater number of constraints imposed by having more parents, even though each parent has the same number of children. In other words, each leaf variable is more constrained, but since there are far fewer of them it may be easier to go through the search space and find a satisfying assignment to all these variables. This phenomenon is certainly worthy of more research in the future.

8.3 Multiple-Parent Models

Theory can give us insight into the multiple parent case. We look first at semantic issues of modeling emergent upper-level nodes. Next we discuss a model of the multiple-parent problem formulation.

Distributing Choice through Levels

In this subsection, we briefly discuss some modeling issues raised by the multiple-parent case, which raise subtle but important semantic points for future researchers to consider when extending the Neartree framework from hSAT to other types of models.

Note that, as we have defined hSAT and nhSAT, the state of a non-leaf node is a function of its BooleanCombiner and of its childrens' states. (There is a great deal of computation and decision-making which happens at non-leaf nodes, including specifying targets for children - but no additional independent state variables.) However, one might wish to model the case where there are additional state variables at non-leaf nodes, which influence the state that the node is in.

A straightforward way of doing this would be to e.g. add 1 variable at every node, so that the Combiner is a function of the children plus the node's variable. One could extend this to having more than one variable at the node, which would model having more of a node's state decided by what a node chooses to do (versus what its children do). The essential distinction to make is, to what extent do upper-level nodes have a status as emergent coherent entities not easily describable in the same terms as their children?

This will be useful in more sophisticated models, where there is some non-obvious function which maps the decisions taken at a node to the expressed state at that node. In our current model, the choice of states at a leaf node or at child nodes immediately results in a value at parent nodes, from a formula which is transparently visible to the parent node. The formula itself is difficult to optimize, but it is at least known. But in real situations, this function will be far more hidden - choosing a configuration of components of a system can have a delayed or highly complex effect on the system as a whole, in a way that is not visible to the system itself. (Consider for example Hidden Markov Models, choice of an information ontology, organizational structure and reward, or many other real-world systems.)

An alternative formulation is to try having separate functions to be satisfied. Note that there is an asymmetry in hSAT in terms of how difficult it is to find a satisfying solution to a desired target state - for a given formula, it is far easier to find a variable assignment making the formula equal F than to find a T assignment. (It is, however, NP-complete to minimize the number of true clauses - just as to maximize the number of true clauses.)

So an alternative model has a separate boolean combiner that should be made true, if the target state is F. (Although this may seem like strange semantics, one should remember that we can use "true" and "false" to stand for any two separate target states - "quick" vs "complete", "comprehensible" vs "optimized", and so on.) Note that this is no longer equivalent to hSAT, though.

Steps Toward a Schema Theorem for Neartrees

Let's assume that each node has one of V possible goal states to work toward, and that the probability of satisfying any particular goal state is either constant or depends only on the state desired.

Then at a single node with 1 parent, denoting by d the probability of doing a request successfully:

$$P(\text{sat parent}) = d \text{ (if } p(\text{find a solution}) \text{ is independent of goal state desired)}$$

$$\text{or } = \prod_{i=1..V} (w_i d_i) \text{ (if } p(\text{find a solution}) \text{ is not independent)}$$

And with n parents, letting w_i be the probability that a parent wants the node's goal state to be i :

$$P(\text{sat all parents}) = \prod_{i=1..V} (w_i^n d_i)$$

$$P(\text{sat } k \text{ of } n \text{ parents}) = \prod_{i=1..V} (C(n, k) w_i^k (1-w_i)^{n-k} d_i) \text{ (where, for each } i, \text{ the non-} w_i \text{ choices}$$

must be restricted so that each other state w_j with $j \neq i$ is
chosen less often than w_i)

$$P(\text{sat } \geq k \text{ parents}) = \sum_{j=k..n} \prod_{i=1..V} (C(n, j) w_i^j (1-w_i)^{n-j} d_i) \text{ (i.e. sum on the previous line,}$$

summed from k to n)

$$= \prod_{i=1..V} d_i \sum_{j=k..n} (C(n, j) w_i^j (1-w_i)^{n-j})$$

So we have a numerical method of calculating the probability of satisfaction for multi-parent cases. Note that we have approximated part of the summands in the second and third case by binomial PDF terms, which can easily be calculated or approximated. These are really just the probability for each i that more parents will want i than any other choice, given that

all parents' requests are drawn from the same distribution - an unrealistic assumption in general, made for modeling tractability.

Often w_i is not independent across parents; different parents will often have correlated probabilities of wanting similar states. This is related to our later idea of distinguishing between homogeneous and heterogeneous goal states. A Neartree with homogeneous goal states has high positive correlation between desired states; in this case many of the terms in the sum above would vanish, since all parents would be looking for the same few goal states from children (i.e. "similar subcomponents").

Note also what this formula implies about solution probabilities - they seem to drastically decrease as the number of parents increases. In the simplest case where there are two possible states that parents desire equally often, w_1 and w_2 both equal $1/2$. Even in the best case where d_1 and d_2 equal 1, this implies that the likelihood of satisfying all n parents is only $2^{-(n-1)}$:

If all w_i are equal, w_i is $1/V$ for all i , and hence

$$\begin{aligned} P(\text{sat all parents}) &= \prod_{i=1..V} (1/V)^n d_i \\ &= V^{-(n-1)} \prod_{i=1..V} d_i \end{aligned}$$

In fact, we can formalize this by simplifying the above formulae under the additional assumption that all w_i are equal (to $1/V$):

$$P(\text{sat } k \text{ of } n \text{ parents}) \cong \prod_{i=1..V} (C(n, k) (1/V)^k (1-1/V)^{n-k} d_i)$$

$$\begin{aligned} P(\text{sat } \square k \text{ parents}) &= \prod_{j=k..n} \prod_{i=1..V} (C(n, j) w_i^j (1-w_i)^{n-j} d_i) \\ &= \prod_{i=1..V} d_i \prod_{j=k..n} (C(n, j) w_i^j (1-w_i)^{n-j}) \\ &= \prod_{i=1..V} d_i \prod_{j=k..n} (C(n, j) (1/V)^j (1-1/V)^{n-j}) \end{aligned}$$

Similarly, the probability that a given parent $_jH_k$ with $|+_jH_k| = c$ would be satisfied is only 2^{-c} , since each child tries to satisfy the parent's requested state for that child with probability only $1/2$. These facts imply that successful multiparent optimization is dependent upon parents having many highly correlated or common goals for their common children.

Now we can take the complementary perspective and look at whether $+_jH_k$ are likely to satisfy $_jH_k$. If $_jH_k$ has 1 child:

$$\begin{aligned} P(\text{sat by child}) &= \prod_{i=1..V} (p(\text{I ask child for goal } i) * p(\text{child satisfies goal } i)) \\ &= \prod_{i=1..V} (w_i * p(\text{child has goal } i \mid \text{my requested goal is } i) * d_i) \end{aligned}$$

But $p(\text{child has goal } i \mid \text{my requested goal is } i)$ depends on the parent conflict resolution protocol, distribution of number of parents, and distribution of parent goals. The last item we've assumed defined in our w_i definition. The number of parents we'll set to p_{avg} for tractability, keeping in mind that a more accurate analysis would look at the distribution of p and not just its average. And for the parent conflict resolution protocol, we'll use the majority-of-parents scheme, where the goal is set as what more parents want.

With no information about the PDF of goals of parents, it is difficult to evaluate $p(\text{child has goal } i \mid \text{my requested goal is } i)$. As a simplifying example, consider the case where parents have each possible state $1..V$ as goal with equal probability; then the term in question reduces to $1/V$ (since the state for which the number of parent requests is maximum is equally likely to be any of the possible states).

$$\begin{aligned} P(\text{sat by child}) &= \prod_{i=1..V} (w_i * p(\text{child has goal } i \mid \text{my requested goal is } i) * d_i) \\ &= \prod_{i=1..V} (w_i * (1/V) * d_i) \\ &= 1/V \prod_{i=1..V} (w_i * d_i) \\ &= 1/V \mathbf{w} \cdot \mathbf{d} \text{ (i.e. } 1/V \text{ times the dot product of } \mathbf{w} \text{ and } \mathbf{d}) \end{aligned}$$

One can similarly derive more complex formulae for being satisfied by at least k children.

Given a particular problem, this analysis could be carried further in several directions. For example, one could explicitly account for bounded resources at a node - with c children, at most an average of $1/c$ time (minus overhead for maintenance and planning) can be given to each for monitoring and optimization, and hence one could estimate solution probabilities given a time bound if it is possible to estimate $d_i(t)$ and $w_i(t)$. Considering the number of upward paths from a child may be a factor in estimating the likely number of conflicting requests a child is likely to receive.

Note that this model seems to show the near-impossibility of a node satisfying all of its parents under many conditions. (Of course, we made several strong assumptions, which need to be verified or disproved before applying the model to any specific problem.) So how do actual decision-making units manage to deal with conflicting requests in real life? In several ways:

- Accept that many parents will be unsatisfied - ignore them, and do the best job you can on the ones you decide to listen to.
- Maybe the states requested are not mutually exclusive - then you might try to optimize both simultaneously. (This is usually the case in multiobjective optimization situations, where several goals may be complementary.)
- Realize that requests from different sources are often positively correlated - and perhaps even search out correlated requests.
- Recognize your limitations, and listen to or belong to only a few parent nodes.

9. Clarifying Hierarchy

“A complex system that works is invariably found to have evolved from a simple system that works.” - John Gaule

9.1 Beyond SAT: A Framework

Now that we have seen how hierarchical SAT works in practice, it's time to examine some issues of computation and definition. There are a number of important points to address, some clear and some subtle. Being precise about how our hierarchical system works - and the “dimensions of variation” along which it could be adapted - will help us both computationally in the current endeavor and intellectually for future research.

State vs Value - A Framework

Recall from our description of the program that the elemental objects of a single node in a Neartree (as instantiated in our program) are as follows:

- **State:** A description of the node, from the point of view of that level. This can be considered a “summarizing statistic” that describes a node sufficiently to be all the information that a parent-level node needs; this is necessary computationally to avoid having to “recursively unpack” each composite node to its representation in terms of leaf nodes. E.g. true / false.
- **Combiner:** A function that has as inputs child states and has as output the current node's state. Thus the Combiner incorporates the “local environment” or problem to be solved by this node.
- **Goal state:** The state(s) that a node wants to get into.
- **Value:** Some estimate of the current value or worth of the node. E.g. distance from goal state, robustness, diversity, minmax distance from goal state. NB: Value is a separate computational layer from State - and the function for Value is dependent on the goals

and estimates of the user. Ideally, a node's Value should summarize its State for its parents.

- **Value Combiner (implicit):** There is an implicit method of incorporating child Values into a node's Value, although this is not usually a separate object.

So a node can take actions which change its own state, subject to constraints imposed by the problem. The value, however, is derived from the set of node states, goal states, and other Neartree parameters.

Let's make tables showing the essential elements of a Neartree, for several levels, in two different problems. This will point out the differences between using hierarchical Neartrees for SAT, and using them for other problems. First, hSAT:

Table 6...Elements of Hierarchical SAT			
	L (boolean variable)	L-1	L-2
State	T/F	T/F	T/F
Child States	-	T/F	T/F
Combiner	-	static CNF Boolean formula of child states	
Value	optional - # parent requests satisfied	% clauses SAT - % clauses requested-SAT (or other similar formula)	
Goal	dynamically assigned by parents; if >1 parent, do majority request minimize distance from my state to goal state assigned by parent(s)		

Next, consider a hierarchical TSP problem, which we define as follows: *Fractal TSP* is simply the TSP problem, but with the additional restriction that the cities to be toured are recursively clustered. The minimum intercluster distance D at each level should be large compared to the maximum intracluster diameter d ; this way we can hopefully guarantee that the best tour enters and leaves each cluster exactly once. If the number of nodes in a cluster is bounded by N , then a sufficient condition is that $(N-1)d < 2D$. That in turn seems to imply

that optimizing each cluster separately and then optimizing the between-clusters tour (and proceeding recursively) gives a good solution.

A simple first heuristic would be to use “breadth-first recursive k-NN”: in the first pass, split into k clusters and optimize between clusters; in the second pass, recursively optimize each sub-cluster (note that the sub-clusters can be optimized separately which may be useful if parallel or distributed hardware is available). This heuristic (and the fractal TSP method in general) seems like a reasonable and fast heuristic for solving TSP, although testing is obviously needed to establish solution quality and time properties - one might test or estimate the goodness by running a fast clustering algorithm and checking some clustering coefficient of the TSP problem, since clustered data should be more suitable for this approach. (Reclustering many times to improve initial clusters might happen in a more sophisticated approach.) This general approach might be especially valuable if the clusters are reasonably persistent but still change slowly, e.g. a TSP in a changing environment.

An illustrative domain where fractal TSP would make sense is construction of a “world tour”: one world, 200 countries, each of which has 5-100 regions, each of which has several cities and towns, each of which has a layout. This naturally gives rise to multiple parent cases, since regions come with overlapping borders, and one may use multiple salespeople and weight sales routes by importance (so multiple trips are possible). (Of course, different salespeople might have different estimated weights of the value of visiting each node, which induces another interesting problem. And they could have restrictions or rules as to what should happen - not just min length or max weight, but logic rules or restrictions, which makes a sort of CSP. But we digress...let us return to the matter at hand and leave this as an open problem for future algorithmic research.)

Now, how does the hierarchical TSP compare with hierarchical SAT? As it turns out, there are fundamental differences between the two problems, which materially influence both the solution strategies and the computational formulation used to define them:

Table 7...Hierarchical TSP			
	L (city)	L-1	L-2
State	position	predefined convex hull of child cities	
Child States	-	position	~convex hull
Combiner	-	dynamic tour (city ordering) Value: static inter-city distances	
Value	-	tour distance + sum of childrens' values	
Goal	-	static - minimize tour distance	

First, note that leaf nodes in the hTSP have a fixed “state” - in marked contrast to the hSAT where leaf nodes must choose their state based on parent requests. The choice has been moved into the Combiner function of parents, which changes the locus of decision-making. In a sense, the Combiner must be more dynamic than the hSAT case - not only must it optimize a tour among its child nodes, it may also need to choose its child nodes.

As well, the Value of a node is a better representation of its worth in the hTSP case than in the hSAT case. If we know how long a subtour is, we always know that decreasing the subtour length is good, if the child nodes stay the same - as opposed to the hSAT case where goals or requests at higher levels may radically change the goal at a given node. We discuss this issue at greater length next.

Homogeneous, Heterogeneous, and Multiobjective Goal States

By assumption in both hSAT and hTSP, the goal for each node is independent of its sibling nodes. (Sibling nodes do interact, but only through their parent Combiners.) However, we can distinguish one more kind of independence, where a node’s goal is independent of its parent and child nodes. We can therefore distinguish two common cases: nodes across levels have the same goals or have different goals.

In the first case, all nodes have the same goal. An example is hTSP above - every node is trying to minimize the tour distance of its children, and this does not change with changing parent state. Similarly, changing child states changes the value but not the state of a parent. Then each node can optimize its own subproblem relatively independently, since its target will not change.

In the second case, different nodes may have different goals. An example is hSAT - although the Combiner at a node stays the same, the desired State may change depending on the requests of parents. In such cases, one rational strategy is to put some effort into finding satisficing solutions for the range of goal states for which a parent might ask, and keeping a cache of past best solutions for each goal state - our implementation is discussed in the Multiple Parent chapter.

With multiple parents, or with a sophisticated node where parents want some mixture of goal states, a single node may be trying to fulfil multiple goals - this leads to a multiobjective optimization problem. A common strategy in such problems is to satisfice on all goals (e.g. by defining a minimum acceptable value for each objective, and then optimizing each objective in turn subject to the constraint that previously-optimized objectives don't fall below their minimum value).

A deep analytic consideration is the comparison of difficulty of reaching different goal states. For example, in hSAT it is usually easier for a formula to be falsified than to be made true; hence there is an asymmetry in the difficulty of the two possible goal states. A moment's reflection will suffice to suggest how much more complex such considerations can become in a more detailed problem; each node will have particular statistics on the difficulty of different state regimes, which will usually be dependent on other node's states.

We look briefly at this issue in a model proposed later in this chapter, where the difficulty of reaching each separate goal state is assigned to a different function. It is possible to model this issue in hSAT as well, by having two or more BooleanCombiners at

each node, with each such Combiner being applicable to a particular desired state. This allows natural parameterization of i) difficulty of reaching each state, as each state's Combiner can be chosen from a known distribution with desired difficulty level, and ii) common subproblems, since two or more Combiners at a state can be constructed with a specified percentage of common clauses. We leave exploration of this interesting model extension to future research.

Multiresolution Views and Other Dimensions of Differentiation

One advantage of using a hierarchical representation is that one can choose to use just some subset of levels below the root, resulting in a less accurate but more efficient approximate representation. This coarse-resolution version of the problem may be easier to optimize, understand, gather data for, and so on.

Note that, given some Neartree H , one can consider the sequence of approximating Neartrees: ${}_0H, {}_{0..1}H, {}_{0..2}H, \dots, {}_{0..L-1}H, {}_{0..L}H$ (where the last Neartree is equal to all of H). Since earlier Neartrees in the series are smaller - possibly exponentially smaller - it may be easier for a bounded-rationality agent to understand more of the world by having access to such approximating Neartrees. Instead of spending a huge amount of time learning a Neartree down to the leaves, the same computational effort might suffice to learn many different Neartrees at a lower level of detail; if the value to a given agent of learning an approximate version of a Neartree is close to the value to that agent of learning the whole Neartree, then this strategy will often pay off.

We note that if approximating Neartrees are of value in a domain, then the creation and maintenance of upper-levels nodes should be given resources commensurate with their importance. (Examples from the world of research include survey articles, books which attempt to synthesize several domains or lines of thought, long proofs or technical articles which have summarizing paragraphs and headings, and news sources with multiple levels of detail available.)

More speculatively, when dealing with multiple domains (and hence multiple Neartrees) it may be easier to tie them together or infer causality if approximating Neartrees are available. Otherwise the level of detail would be too great, and one might so to speak miss the forest for the trees.

9.2 hSAT minus SAT equals h

Carrying on in the vein of the previous section, we can consider simplifying and abstracting the hSAT model.

There are really two related but separable sub-models - the node-level problem (i.e. SAT) and the inter-node problem which determines how differing nodes interact. Consider then a case where the node-level problem is as simple as possible: just a probabilistic success or failure.

More precisely, we can define a PDF $S_{\text{node}}(t)$ of time vs success probability, indexed over nodes. The function S_{node} encapsulates in a simple model the likelihood of solving a node-level problem, given a specific amount of effort t spent trying to solve that problem.

Simple examples of such functions include:

- $S(t) = \text{constant}$. If the probability of solving is constant at every time step, the distribution of solution times is a geometric random variable.
- $S(t) = \text{Lognormal}$. The PDF of $\log(\text{solution time})$ is a (μ, σ^2) normal random variable.
- $S(t) = \text{Pareto}(\theta, a)$. This can be used to model the case where a minimum solution time a exists, with large solution times decreasing as a power function.
- $S(t) = \text{Uniform}(\min, \max)$. If all we know is the least and greatest solution time, a uniform distribution is a reasonable assumption.

The choice of such a function needs to be justified empirically relative to the problem at hand. It may therefore also be useful to model $S(t)$ as an interpolated histogram of observed solution times, which could yield more accurate values but is harder to model.

There is then a separate modeling component consisting of the hierarchical portion, with these parameters:

- c, p, L - Usual Neartree parameters
- (Potentially other parent-related parameters such as those discussed in the previous section - e.g. probability of requiring the same goal.)
- Computational parameters - `resourceBound`, # samples, etc.
- Goal - Find an optimal state (i.e solve the Neartree completely), optimization, pessimization, ...

What can we use this model for? One intriguing possibility is that it could model the empirical nhSAT work we have done - by running a single-node SAT problem, and then entering parameters observed from watching an hSAT, does the solution time (or quality vs time curve) given by the model match that observed in reality? If so, the model may be a good way of predicting hierarchical behavior for other systems as well.

In order to implement this in our hSAT program, it is necessary to change the SAT-specific portions of the code to the simpler model above. Fortunately, due to the modular architecture we have used, this is quite feasible - we only need to change the Combiner and search methods, and partition off a small amount of other SAT-specific code in the Node class.

This model and set of experiments could give a simple, natural, and easily extensible description of the solution power of hierarchical representations. Perhaps one could think of it as a kind of hierarchical version of statistics.

9.3 Phase Transitions

As we have discussed previously, one phenomenon in the SAT literature (and more generally in the constraint satisfaction community) which has attracted much interest is the phase transition phenomenon, where the probability of a formula being satisfiable exhibits a sharp transition from near 1 to near 0 at a specific clause ratio R .

Experiments: Phase Transitions

This experiment series looks at phase transition behavior for the SAT problem. First, we look at the phenomenon at the single-node level - this is done mostly to verify that the phenomenon occurs and thereby validate the program. Next, we look at the phenomenon at a Neartree-wide level - the first time this has been done, to the author's knowledge.

(Note that our definition of "phase transition" looks at the change in fraction of problems that are satisfied as is the usual case in the SAT community. See e.g. [Coarfa et al 2001] for a comparison of algorithms)

```
for increasing Neartree L and specified fanout {  
  for gradually increasing clauseRatio {  
    for several instances {  
      spend timeBound solving this instance  
      record % SAT (or alternatively % nodes nearly-SAT, or average node-value)  
    }  
    record mean / median / range of results  
  }  
}
```

Figure 18...Single-level SAT phase transition

First we verify the 1-level phase transition. This experiment was done with 400 samples per data point. Note the x-axis is a much smaller range than most previous experiments - all data points are from clauseRatio 3.5 to 4.5.

As expected, with 256 children we see a rather sharp transition from solvability to unsolvability - taking place over a range of about 0.4 in clauseRatio.

It is worth emphasizing the non-intuitive nature of such phase transitions. One might venture to hypothesize that very few people, without being told of the phenomenon ahead of time, would predict that almost all boolean formulae with clauseRatio 4 can be solved given sufficient computing power, while almost all such formulae with clauseRatio 4.5 cannot be solved. This should caution us to the likely existence of many such transitions

between solvability and intractability in other domains - a point we have made in a minor way with the demonstration of likely phase transitions in 2 and higher level hSAT.

Here we see a similar phenomenon demonstrated in 2-level hSAT. We again use 400 samples per point, but $c = 64$ this time.

Interestingly, despite the lower value of c (which generally means a broader range over

Figure 19...Two-level hSAT phase transition

which the phase transition happens in single-level SAT), the range appears to be quite similar for 2-level 64-child hSAT and 1-level 256-child SAT - about 0.4 in both cases. Determining the precise relationship between the width of the phase transition, number of variables, and number of levels is another open area for future research.

Mathematical Insights into Phase Transitions for hSAT

In this section, we derive partial results for the existence and properties of the phase transition phenomenon in hierarchical boolean formulae. (One cannot simply apply the results for ordinary SAT to hSAT, since it has a different structure.)

We'll try to derive an upper bound for the phase transition value in the hierarchical case. In all cases, we are making a strong assumption that all node-level problems are independent - this is an approximation which needs to be justified or disproved in future work. We will also assume that the Neartree is homogeneous, i.e. has a common value for c , p , and the clause ratio R .

A single node-level formula for any level jH_k with $j < L$ is just an ordinary case of SAT with some clause ratio R and number of variables c . In this case, the proof for the phase transition value carries over from ordinary SAT: each clause has a $7/8$ chance of being satisfied by a random assignment. The probability that a random assignment fulfils all clauses is thus $(7/8)^{cR}$. But since there are 2^c assignments, the expected number of assignments satisfying all clauses is $2^c (7/8)^{cR}$. This quantity (and the probability of being satisfiable) will approach 0 as n approaches infinity when $R > \log_{8/7} 2$. Hence:

$$p(jH_k \text{ UNSAT}) = 1 \text{ for } R > \log_{8/7} 2 = 5.190893\dots$$

Since the number of Neartree nodes which need to take on value T will only increase as the Neartree grows larger, the upper bound on the phase transition value for a single node carries over to the Neartree as a whole. Hence we can say that the phase transition value (if it exists) for R for any multilevel SAT is at most $\log_{8/7} 2 = 5.190893\dots$

However, it may be less; adding more nodes can only decrease the upper bound. Although (for $R < \log_{8/7} 2$) $p(\text{SAT})$ of each of the formulae individually approaches 1 as c

approaches infinity, the joint probability may not: since each is assumed independent, the probability of all being satisfied might in the worst case (i.e. with all children needing value T) be as low as the probability of a single node being satisfied raised to the number of nodes.

To take this further one needs to answer the question: what are the conditions on the solution probability of each node, that induce a phase transition with increasing numbers of nodes? We need some additional model element, since the ones we have so far seem to require that the solution probability at each node must approach 1 in the limit - one possibility is that a node might be content if some constant fraction of its children are satisfied, similar to MAX-SAT. Note also that phase transition phenomena have a potentially-important practical implication, as demonstrating mathematically the existence of non-intuitive “difficulty limits” beyond which problems become almost impossible to solve.

For an individual formula, $p(\text{SAT}) \approx \text{expected \# satisfying assignments}$ (see p644 of [Duf et al 1997]). A stronger inequality is possible from more advanced techniques (p645). However, the problem with using these is that using any value for R below the bound gives an expected number of satisfying assignments > 1 , and hence no bound for $p(\text{SAT})$.

Suppose we did have a formula for $p_{\text{Node-SAT}}(c, p, R)$. Then, under the assumption that nodes are independent, we have that:

$$p_{\text{NT-SAT}}(c, p, L, R) \approx p_{\text{Node-SAT}}(c, p, R)^{(|L \cup \dots \cup L| * \% \text{ nodes needing value T})}$$

The % of nodes needing value T might be around 50% by symmetry: the expected number of true and false truth assignments might be 50% each, since for each assignment there is an opposite assignment with all literals and truth values inverted. To be safe, we might choose 100% for a lower bound on $p_{\text{NT-SAT}}(c, p, L, R)$. (Note that $p(\text{finding an UNSAT assignment})$ is always 1, since we can just pick any clause and choose truth values to break

it.) We can estimate the other term on the right as follows:

$$\begin{aligned} |I_{0 \dots L-1} H| &\leq 1 + c + c^2/p + c^3/p^2 + \dots + c^{L-1}/p^{L-2} \\ &= 1 + c (k^{L-1} - 1) / (k-1) \quad (k = c / p) \end{aligned}$$

To derive $p_{\text{Node-SAT}}(c, p, R)$, one approach is through examining the number of satisfying assignments. We have from the proof above that, for a single node with c children, there are 2^c assignments, the probability that a random assignment fulfils all clauses is $(7/8)^{cR}$, and hence the expected # of assignments satisfying all clauses is $2^c (7/8)^{cR}$. What we want is probability(# of assignments satisfying all clauses is zero). But we can find this by considering each random assignment's probability of satisfying the formula as a binomial trial, and asking what the probability is of at least one successful binomial trial:

$$\begin{aligned} p_{\text{Node-SAT}}(c, p=1, R) &= p(\text{at least one successful trial}) \\ &= 1 - p(\text{all trials are failures}) \\ &= 1 - p(\text{one trial is a failure})^{\# \text{ of trials}} \\ &= 1 - (1 - (7/8)^{cR})^{2^c} \dots \end{aligned}$$

The case of multiple parents is more complex. The relevant parameter in this case is the probability that multiple parents ask for differing goals (in which case node satisfaction is impossible). We can simply assign this a probability that needs to be worked out in future research. (As we see in our multiple-parent model, this probability is a very relevant parameter for any multiple-parent situation, not just with boolean formulae; it can probably be measured through experiment or observation for many systems.)

9.4 h-MAX-SAT and Minimal Information Sets

Note that SAT is essentially a 0-1 problem at each node with no explicit intermediate fitness values to provide an optimization gradient (although as we have seen one can use

derived fitness values like the number of clauses satisfied).

Is it true that as the evaluation information at each node increases - nominal, ordinal, interval, ratio, multivalued, general model - an optimization method will be able to deduce more and more information about valuation gradients in the space, and hence increase its speed and the quality of solutions found? We look at h-MAX-SAT - the optimization version of hSAT - to get some insight.

Next, we close the chapter by pointing out an isomorphism between hierarchical MAX-SAT, hypergraph vertex covering, and efficiently learning a minimal set of information to understand a composite field of knowledge.

Experiment: MAX-SAT

This experiment series looks at how close Neartree solutions come to optimum. Instead of measuring the fraction of satisfied formulae, we define a measure on the quality of a Neartree solution, and use that as our dependent variable. The values of nodes are aggregated using the equal-levels method outlined in the algorithms chapter earlier.

Each node has a SAT problem as before. However, instead of an all-or-none approach, each node's value is set equal to the fraction of clauses satisfied, if the goal is T. For a goal of F (i.e. where one wants to falsify as many clauses as possible), valuation is more difficult since some minimum fraction of clauses must always be satisfied. For this case, we simply assign the maximum value as long as any clauses are falsified, which yields an optimistic but easy-to-compute result.

fix a solution method and corresponding parameters

for some Neartree sizes {

for several instances {


```

    spend timeBound solving this instance
    record solution quality, with respect to Neartree solution quality metric
  }
  record median / mean / range of results
}
}

```

Graph curves or surfaces by changing solution quality.

Compare solution quality variance across levels and nodes.

In order to do MAXSAT instead of SAT, we need to make several changes to our program. First we need to modify WalkSAT to store the corresponding variable assignment whenever a new minimum number of unsatisfied clauses is reached. We track and use this clause-satisfaction measure.

For weighted MAXSAT - we add a weight to each clause in a separate array, and use the dot product of the weight vector and clause satisfaction as the fitness (or value) of an assignment, instead of just the number of satisfied clauses. (We don't use this in the experiment below, but it will be useful for applications, and is mentioned in the subsequent section on minimal information sets.)

Finally, instead of tracking the number of formulae satisfied, we track the value of each formula. This value is defined as in the algorithms section - the value of a Neartree is an average of the value of each non-root level, the value of each level is the average of node values, and the value of each node is the fraction of clauses satisfied in its BooleanCombiner (for non-leaf nodes) or the fraction of parent requests satisfied (for leaf nodes).

We also tracked the minimum, maximum, 5 %ile, and 95%ile values to get a sense of the distribution of values (not shown).

Figure 20...Two-level h-MAX-SAT

These experiments were done with 2 levels, $c = 4, 8$, and 16 , and 400 samples per data point. All of the curves are for the single-parent case. (Note that the y-axis is reporting the value metric.)

We see that the behavior of MAX-SAT is quite different from that of SAT. Instead of the phase-transition behavior seen earlier, there is a smooth dropoff in the value metric as the clauseRatio increases. Even for values approaching 5, the value metric is still reasonably high. (Note though that it is somewhat skewed by our metric's inclusion of leaf node's fitness values - although this seems a good idea in the general case, it produces relatively high-values for $L=2$ in the single-parent case, since leaf nodes always get satisfied.)

Clearly MAX-SAT is representative of quite a different problem space. In accordance

with intuition, if it is satisfactory to leave a number of constraints unsatisfied, then one can deal with problems of a substantially higher difficulty value. It will be of interest in the future to examine more fully the shape of the curves for a broader range of L , c , and clauseRatio . The distribution of value across nodes is also of interest - are just a few nodes causing most of the difficulty, or is the value metric fairly well distributed across nodes? Clearly similar questions can be asked in the hSAT case. In both cases, real-world near-hierarchical applications - for which data sets have yet to be developed - may well have different characteristics from the parameterized families we have considered; however, these parameterized families form a well-specified base case for future research to build on.

One difficulty in running additional experiments with h-MAX-SAT is that one really needs to specify “how much is enough” in optimizing - there needs to be a specific bound where one can say a node-level problem has been sufficiently optimized. This issue does not arise in hSAT, since we always require a node to be completely optimized. It thus seems necessary to have a more completely specified algorithmic problem or a large set of real data to work with for h-MAX-SAT, as otherwise one runs the risk of making too many arbitrary assumptions.

It is noteworthy that our algorithm can deal with this h-MAX-SAT case with only a few minor modifications. We are interested in applying this program to more practical applications in the future, since MAX-SAT can be naturally used to model a variety of optimization problems.

Minimizing Information Overload through Hypergraph Vertex Covers

We now present an important problem that turns out to be naturally modelable with hSAT.

Consider the dilemma of an individual or group faced with becoming competent in a field of knowledge. What does it mean to “become competent”? Typically each field has

a set of subfields. Some of these subfields are essential. Others, however, may be part of a loose collection, where it's important to know some number of items from the collection, but not necessarily all items.

But we can model this situation with the "vertex cover on hypergraphs" model. In the vertex cover problem for graphs, a minimal covering subset of vertices must be chosen such that every edge is incident to at least one vertex in the cover.

Recall that a hypergraph is a natural extension of a graph, where an "edge" can connect more than two vertices (i.e. an edge can be any subset of vertices of the hypergraph). Now if the nodes in such a hypergraph represent categories or subfields, knowing a subfield corresponds to selecting a vertex.

Using hypergraphs instead of regular graphs permits us to model the case where three or more subfields (nodes) form a cluster where one needs to know only one of them to be considered competent in the problem represented by that cluster (or to know what is known about the cluster). So - finding the minimal amount of learning or research effort to understand a field under our model corresponds to finding a minimal vertex cover in a hypergraph. (Approximation algorithms for this problem are discussed in [Halperin 2000].)

Now, though, we can translate this problem to a CNF SAT problem! The transformation is straightforward, as follows:

- For each node i in the hypergraph, create a variable v_i . The variable v_i will be true or false according to whether node i is chosen or not chosen (which in turn corresponds to the subfield being learned or not learned).
- To each v_i , assign a weight w_i . This corresponds to the cost of learning v_i .
- For each edge in the hypergraph, create a clause consisting of the disjunction of the corresponding node variables. I.e. if an edge had the vertices v_1, v_4, v_6 , then the

corresponding clause is $(v_1 \vee v_4 \vee v_6)$.

- Now, consider the formula which is the conjunction of all clauses formed in the previous step. Clearly this formula is satisfiable, as there are no negated variables - one can always just set every variable true (which corresponds to choosing all nodes). But the *_minimum weighted_ vertex cover* corresponds to finding the least-cost subset of variables - i.e. least-cost subset of fields to learn - which "covers" all the hyper-edges. Such a covering subset means that one has sufficient background to understand all the clusters.

Great! We've shown that the Minimal Information Set problem is isomorphic to Hypergraph Vertex Cover. (Of course, we've made some simplifying assumptions for modeling tractability, such as thinking of each subfield as either learned or not learned.)

Now we can go further and relate this result to our hSAT formulation. Consider that subfields of knowledge are themselves often composed of many distinct sub-subfields in their own right - and in fact this process goes on for several levels if one starts with a broad or interdisciplinary problem. So let each node in the top-level problem (i.e. $_1H$ nodes in our parlance) itself represent another Hypergraph Vertex Cover problem, with all $_2H$ nodes in this problem (i.e. ^+_1H) representing subdivisions of the field corresponding to the $_1H$ node in question.

We can still consider only the top-level problem, and it remains valid. But we now have the option of considering what might have to be learned at the next, more detailed level in order to really understand the first level. As an example, consider "knowing computer science". Surely, every self-respecting computer scientist must know something about $(AI \vee \text{heuristic methods}) \wedge (\text{assembly language} \vee \text{compilers} \vee \text{lambda calculus})$.

But to know about AI, one must know about $(\text{Evolutionary Computation} \vee \text{Neural Networks}) \wedge (\text{Simulated Annealing} \vee \text{Fuzzy Logic} \vee \text{Neural Networks})$. Note the repetition

of Neural Networks - in general, there would be many such repetitions, since subfields often combine with many other fields to make different "knowledge subsets" that are useful for solving or understanding particular problems. Note also that the clauses would usually be weighted - with the weights representing how important each knowledge subset is, either to the person doing the learning or as a consensus of the field as a whole. Weights could also take into account time or other investment required to learn that knowledge subset.

One potential application of this algorithmic problem is the computation of **“Minimal Information Sets”**. Suppose you are faced with a large set of information to learn - many different disciplines, each of which has many subtopics, each of which in turn might be further subdivided, and so on.

Represent the information universe you are dealing with by a Neartree, and fix a particular node (i.e. topic) X. Then there are several natural questions to ask about X:

- What’s the least-cost subNeartree of X that you need to learn to be competent in X? How about to “know what’s known”? (Cost can be e.g. time required to learn, weighted by the reward for learning it.)
- Suppose you want to summarize X - for reporting news about it, or for generating summaries and multiresolution views. Can you leave out some of the levels adjacent to the leaf level of the Neartree when discussing X? Have you gotten the best “covering set” of the relevant points of view on X, given your resources? (Metrics include time, importance, and breadth of coverage.)
- Are there simple Combiners which usefully describe knowledge about X, as a function of knowledge of X’s child nodes?

Another important application area is scanning the news or literature; our increasingly limited time relative to what there is to know makes a hierarchical solution more and more necessary. Simply filtering most information is necessary but not sufficient - a near-hierarchical ontology (or some more complex knowledge structure) is necessary to put the

information we do get into a coherent framework.

We have shown a natural correspondence between Weighted Near-Hierarchical Hypergraph Vertex Cover, and Minimal Information Sets in multilevel domains. This seems like a natural candidate problem to illustrate the applicability of hSAT, to a very timely and important practical difficulty.

10. Data to Neartrees, Dynamic hClustering, and DisSATisfaction

“Genius is the ability to reduce the complicated to the simple.” - C W Ceran

10.1 Noticing Neartrees

How Hierarchical?

We can distinguish two distinct problem categories: i) those which are “naturally hierarchical”, or defined with a hierarchical specification, and ii) those which are defined in some non-hierarchical fashion, where a transformation can be made to put the problem in an equivalent hierarchical format. In the latter case, not every problem really has an underlying hierarchical representation - hence it becomes necessary to decide how close a data set’s current transformation is from being in the desired Neartree format.

If the hierarchical representation has been derived from a clustering method, one should note that most clustering methods will produce a clustering, regardless of whether or not the data really is clustered. Testing for clustering tendency is therefore necessary where general properties of the data are not known (and in fact such testing is usually worth doing even where the data is thought to be understood). Statistical methods can be used - typically by defining some appropriate statistics that capture the notion of clustering (e.g. intra-class vs inter-class similarity), and then comparing statistic values for the problem at hand with those from problem sets with known values and clustering tendencies.

Consider now problems which can be specified as Neartrees easily, e.g. our hSAT example. Given a general non-hierarchical problem - specified as a set of objects, each of which has internal computations and interacts with other objects - can it be converted to a hierarchical one, and if so how effectively? This is a highly non-trivial problem, as we

demonstrate by considering what would be involved in a general heuristic solution procedure:

- First, if the problem is not in the form of a set of interacting objects, one may need to define additional objects that encode the interactions. For example, a SAT problem has only boolean variables in its definition - one needs to come up with “composite variables” that encode the result of partial computations of the SAT formula, so that the single overall SAT computation is represented as a set of local computations.
- Now we have a set of objects which interact with each other. Draw a node for each object, and a directed arc between nodes for objects A and B if B sends output to A (the meaning of this output is problem dependent, but could represent B influencing or being a part of A). This step forms a DAG containing all variables and specifying their interactions.
- In general, it is necessary to compress this DAG by defining the right composite variables in the first step above. (Think of each such composite variable as a higher-level concept that succinctly encodes the state of a set of lower-level variables.) Additional compression may also be possible depending on what output is sent by nodes and what computation is performed at nodes.
- Given a compressed object relationship DAG, use a topological sort or other method to arrange it into layers. Now there is a partial ordering defined on nodes, and the partial order metric defines a level number.
- Define a metric measuring how close the layered DAGs is to being a Neartree. Then we can decide how far the layered DAG is from being “near enough to being a Neartree”. The Neartree-ness is dependent on the user to some extent; one will have to decide what values of L , c_{avg} , p_{avg} , c , p , etc are acceptable.

Consideration of the steps involved will give an idea of the magnitude of the task in the general case.

In the last step, the Neartree distance metric might be partially derived from tree distance metrics for phylogenetic trees, or from metrics measuring the goodness of clusterings as in [Theodoridis & Koutroumbas 1999]. We can define a sensible metric as follows: set the maximum acceptable c and p values, and penalize all nodes which go over them by some increasing function of the excess. Formally:

$$\text{nonNT}(c, p) = \text{average over all } j \text{ and } k \text{ of} \\ (\max(\text{LOSS}(l_j^+ H_k^-, p), \text{LOSS}(l_j^+ H_k^-, c)))$$

$$\begin{aligned} \text{LOSS}(\text{value}, \text{bound}) &= 0 && \dots \text{if } (\text{value} \leq \text{bound}) \\ &= (\text{value} - \text{bound}) / \text{value} && \dots \text{if } (\text{value} > \text{bound}) \end{aligned}$$

(In fact, for a specific given problem we would actually want to refine the method by weighting the excess in each variable differently. For example, exceeding the c bound is bad in proportion to the difficulty of the Combiner; exceeding p is bad in proportion to the likelihood of differing requirements of parents.)

How could one test this idea experimentally? One approach is to try transforming several formulae with a Neartree-generating heuristic, and then comparing the Neartree-ness metric of the transformed formula with the actual closeness to a Neartree as determined by human analysis. Formulae that could be used include the benchmark DIMACS formulae, real circuits, and finally handmade formulae parameterized by degree of hierarchy.

One could try this with a large number of formulae, and then measure the statistical correlation between the metric and the a priori Neartree-ness. It's also of interest to see if the metric is correlated with faster solution times or better solutions, which would provide

additional evidence that near-hierarchical problems can be more adequately solved. Finally, a Neartree-ness metric provides a possible element of a fitness function for Genetic Algorithms or Genetic Programming to use in the evolution of structured solutions and programs.

In the case of boolean formulae, we have implicitly seen in a previous section that a SAT problem of length n with an equivalent hSAT representation seems to be solvable more quickly than other formulae of the same length n , since the hSAT representation is much shorter and easier to solve. However, most SATs will not have such shorter representations; finding these representations is an interesting open problem. If one can find an approximate representation, that may also be of interest, e.g. a compact hierarchical hSAT that has a similar truth table with respect to its leaf inputs and root node output as a larger SAT problem.

A related method of finding the closest approximating tree for data is that used in decision tree learning, as explained in [Mitchell 1997]. There, the information gain is used to decide which attribute of a set of data should be used to partition the data set at each step; this measure is essentially the difference in entropy before and after the attribute in question is used. When post-pruning possibly-overfitted trees, similar measures can be used. This general method may be useful to create and prune Neartrees on data sets.

Notice that the previous formulation works only with binary relationships between parts of a problem - they are either related or not related. These binary relationships induce a graph, as discussed above. A natural extension is to allow inter-part relationships to be weighted, thereby inducing a weighted graph. In this case, we can use not only the connectivity but also the “degree of flow” between nodes, in deciding how similar to a Neartree a given data set is.

A simple algorithm in this case might be “Nearthree imposition on a weighted digraph”: given a weighted digraph, form the “closest approximating Neartree” by e.g. removing

edges. This closest approximating Neartree is analogous to finding the decomposition of a problem space that is closest to being hierarchical, and as such represents a simplest decomposition of a problem.

A related metric could quantify the degree of nested-ness. Given the node-level function $f_j(H_k)$ - which could be computation, description, prediction, etc - we would like the complexity of $f_j(H_k | \bigwedge_j H_k)$ to be small. In other words, children in a flow hierarchy are only inputs to their parent, which could be a totally separate process. Children in a compositional hierarchy are by contrast part of the description of their parent, so the parent consists of relatively little more given the child descriptions.

Experiment: Expanding hSAT into SAT

Given a specific instance of hSAT with multiple levels, we can convert it into an equivalent SAT problem. This vividly demonstrates the computational savings of the hSAT approach.

Converting an hSAT involves scanning the hierarchical representation level by level, replacing literals at a given level by their formula expressed in terms of literals at the next lower level. The general procedure is as follows:

```

EquivSat =  $\phi H_0$ .
for (levelNum = 0 to L-1) {
    tempSat = EquivSat
    for each literal  $_{levelNum}H_i$  in tempSat (scanning e.g. from left to right) {
        replace literal with “(“ + BooleanCombiner of  $_{levelNum}H_i$  + “)”
    }
    EquivSat = tempSat
}

```

output EquivSat

We have implemented this procedure. Typically, a relatively small hSAT expands to a large resulting equivalent SAT, and the size of this equivalent SAT grows very rapidly with the number of levels in the formula. For example, a solvable instance with $p=1$, $L=3$, and $c=12$ yields an hSAT of length 125K, with equivalent flatSAT of length 9288K. For a solvable instance with $p=1$, $L=3$, and $c=16$, hSAT is only modestly larger at a few hundred K, but the flatSAT representation is huge - somewhere over 100 megabytes. And these are relatively modest problem sizes...

It is clear from examination of the procedure that a given hSAT representation can be equivalent to a SAT formula that is exponential in the number of levels of the hSAT - which clearly demonstrates the infeasibility of solving an hSAT formula by converting it to these equivalent SAT formulae. (There may be some other method for finding efficient SAT representations for some hSAT formulae - this is an open question, as far as we know.)

We can think of this process as follows:

a compact hSAT hFoo --> a huge equivalent SAT Foo --?-> a compressed Foo

We know how to do the first conversion - the algorithm is given above. There may be algorithms for compressing Foo, which would give the second conversion (and which seems necessary for even attempting to solve equivalent SAT representations).

Now an interesting question: are there conversions from the second or third representation to the first? In other words, can we find an hSAT equivalent to a given SAT, or show that no such hSAT exists?

If so, this would give a method for solving SAT problems for which an equivalent hSAT

could be found. Note that such SAT problems will be sparse in the set of all SAT problems (a simple counting argument suffices to show this). It's interesting to speculate whether algorithms could be developed to find a "closest approximating hSAT" for a given SAT - that would give the hierarchical equivalent that is most akin to a given non-hierarchical problem.

Evolving hSATs and Neartrees

Up until now, we have been parameterizing the problems being solved by structural properties. However, these problems are being measured with respect to some metric in each experiment, e.g. time to solve, fraction solved, or quality of solution found.

It is therefore of interest to try to evolve problems which are either very easy or very hard with respect to the metrics we have defined - e.g. to search for hSAT instances which are solvable, yet which take the most computation time to solve. Similarly, one could try to evolve a Neartree that is a "closest hierarchical representation" of a given single-level formula.

```

for differing Neartree metrics and values {
  for several independent runs {
    spend timeBound evolving an hSAT instance with desired metric value
    record metric value found
  }
  record mean / median / range of results
}
Analyze properties of hSATs found.
```

An intriguing idea for carrying this method further is to use a co-evolutionary approach for problems like our DisSAT setup introduced later in this chapter. There would be two

populations of solutions being evolved - one trying to optimize an hSAT structure for some specified problem, and the other searching for weaknesses in the structure. A related approach is the evolution of robust structures, as in [Imamura et al 2002]; sorting networks have been used as a testbed, i.e. networks that have been evolved to sort input sequences of bits, given some noise process that may corrupt parts of the computation.

We note a connection with “generative genetic programming” - the evolution of computational structures, where each structure is specified implicitly via a compact representation that is executed to get the full structure. (In other words, the evolved genotype is a compact representation, which is executed to generate a substantially larger phenotype; the latter phenotype is what is actually evaluated in the fitness evaluation phase.) This line of work seems like a logical next step for research into the evolution of hierarchical structures; see [Hornby 2003] for detailed examples and discussion.

Empirical Evidence for Neartrees, and Statistical Considerations

Is it possible to find empirical evidence for Neartrees? This would consist of finding a domain of importance, which is i) subdivided into nodes on discrete levels, with ii) restrictions on the numbers of parents and children, and iii) each layer representing a successively more detailed partition of a problem space.

In the introductory chapter on decomposition and hierarchies, we have given anecdotal evidence for the existence and importance of hierarchical structures in many fields of study. We would suggest that it is a worthwhile research problem to define a small set of metrics, and survey data across these fields for specific large systems which exhibit definite Neartree-like parameter values in their structure.

Three specific suggested examples are:

1. Online ontologies, e.g. Yahoo!, dmoz.org, and Google’s version of dmoz. (Hypothesis -

all of these are Neartrees, where the Combiner function is related to human cognitive ability to understand categories. (This could be checked through getting statistics on the number of parent and child nodes in various categories. It would be interesting to note where and why nodes or topics are exceptions.)

2. File Structures, e.g. in PC and Server operating systems. Again, this is a testable hypothesis - one would, however need to distinguish between directories meant to be human-readable, and those primarily used by computational tasks.
3. Circuit Design. Ordered Binary Decision Diagrams are a tool for compactly specifying certain classes of SAT formulae, by representing them as compressed binary parse trees; they are widely used in model checking, e.g. [Clarke et al 1999]. VLSI circuit design in general makes ubiquitous use of modular circuit specifications.

An issue in practical applications is getting sufficient detailed data to validate a model - this is a problem common to applying any computational model, but is perhaps accentuated in the case of Neartrees by the fact that a near-hierarchy typically describes a system which is quite large in size. The users of a Neartree will need to expend effort in getting data of sufficient quality and quantity - there is no way around this.

Algorithmic problems provide a feasible way of generating large-enough synthetic data sets and problem spaces; they also have the advantage of being easy to understand, and transparent enough that one can see what works and why in solving such problems (which is why we have used them). Real-world domains for which large amounts of data are readily available include financial markets and information organization. However, such data does not usually come packaged into near-hierarchical formats, making the data acquisition problem a difficult task.

Statistical difficulties involve separating signal from noise, and determining causative factors. The latter may be particularly difficult in a large Neartree whose data is determined from observation; in a large system with many interacting parts, determining which factors caused a long-term event to happen can be extremely difficult. Complex causality is

especially problematic where good models of the underlying behavior do not exist, e.g. social, economic, and political behavior. (Bayesian network research seems a promising avenue for moving forward on this front; see [Pearl 2000].)

10.2 Creating Neartrees through Clustering

Assuming that a Neartree will be constructed by a human is a sensible approach in many cases - if the Neartree will be used for a truly large-scale problem, it is important that it be structured in a manner that respects many constraints both obvious and subtle, which a human can often do better than an algorithm.

However, there will also be many cases when algorithms will be useful to assist in the creation or autonomous generation of Neartrees. We therefore turn to the consideration of algorithms which could help in this endeavor. (This problem of generating Neartrees is distinct from the main focus of the thesis, which is the use of Neartrees assuming they have already been generated; the discussion in this chapter should therefore be viewed as an initial set of ideas for future research to build on.)

We need a generally applicable way of decomposing problem domains; this will form a preprocessing stage that will decompose data in a near-hierarchical fashion, or possibly show that no such decomposition is possible. One such method of problem decomposition is **clustering**, which is widely used in science and data analysis for understanding large data sets; applications include computational biology and recommender systems. This canonical method of unsupervised learning groups data items into “natural” similarity classes; properties of these classes can then be used to understand problem characteristics or find similar groups of data items.

If we view the idea of clustering in a more general way - grouping ideas and data into similarity classes - it is in fact a near-universal method for decomposing large problems into

manageable chunks; we cluster implicitly every time we focus on a task and choose only those variables and data that are relevant to the problem at hand. We will discuss the traditional clustering methods which operate on vectors, and then proceed to a more general framework for decomposition that will use some of these ideas. Our discussion of clustering methods is largely adapted from [Theodoridis & Koutroumbas 1999] and [GordonA 1999].

Clustering in a Nutshell

Clustering is the process of reducing a mountain of data to manageable piles. For cognitive and computational simplification, these piles must consist of “similar” items. (Similarity measures depend on the assumptions one brings to the data; different ontologies of the same data can result from different similarity measures, each of which may be equally valid for a particular usage domain.)

Basic clustering steps:

1. Preprocessing and feature selection. Most clustering models assume that data items are represented by n -dimensional feature vectors. This first step therefore involves choosing appropriate features, and doing preprocessing and feature extraction on data items to measure the values of the chosen feature set. One often chooses a subset of the features available, to reduce the dimensionality of the problem space. This step often requires a good deal of domain knowledge and data analysis.
2. Similarity measure. This is a function which takes two items or sets of data items as input, and returns as output a similarity measure between them. Item-item versions include the weighted l_{pq} norm (and its fuzzy version), the inner product, Hamming distance, Mahalanobis distance, and edit distance. Item-set versions use any item-item version as subroutines and include max / min / average distance; another approach looks at the distance from the item to the cluster representative of the set, where point representatives are chosen as the mean vector / mean center / median center of the set;

hyperplane or hyperspherical representatives of the set can also be used. Set-set versions include max / min / average distance, as well as item-item versions applied to the two set representatives.

3. Clustering algorithm. Clustering algorithms are general schemes which use particular similarity measures as subroutines. The particular choice of clustering algorithms depends on the desired properties of the final clustering, e.g. what are the relative importances of compactness, parsimony, and inclusiveness? Other considerations include the usual time and space complexity.
4. Result validation. Do the results make sense? If not, we may want to iterate back to some prior stage. It may also be useful to do a test of clustering tendency, to try to guess if clusters are present at all (since most clustering algorithms will produce some clusters regardless of whether or not natural clusters exist).
5. Result interpretation and application. Typical applications of clustering include data compression (via representing data samples by their cluster representative), hypothesis generation (looking for patterns in the clustering of data), hypothesis testing (e.g. verifying feature correlation or other data properties through a high degree of cluster formation), and prediction (once clusters have been characterized, new data items can be classified by the characteristics of the cluster to which they would belong).

Sequential clustering is the simplest type, with the basic sequential algorithm being as follows:

Start with one cluster formed by the first data item.

Define the dissimilarity threshold D_{dissim} and maximum number of clusters maxClusters ,

For each new item data_i {

```

    Find the cluster closestClusteri to datai (using any item-set distance measure).
    If ((distance(closestClusteri, datai) > Dissim) and (numClusters < maxClusters)), put
    datai in a new cluster; otherwise add datai to closestClusteri.
}

```

Dissim controls whether overly small or large clusters are found, while maxClusters (which is present to bound computational resources used) can result in too few clusters if set too low. The order in which data items are presented can also change the clustering found; heuristics can be used to constrain presentation order.

One can run the algorithm multiple times and infer a good value of Dissim from the clustering results. This can be done by choosing an appropriate range for Dissim, and running the algorithm several times for each incremental value of Dissim. Present the data in a different order each time, keeping track of the resulting number of clusters; estimate the number of clusters for this value of Dissim as either the mode, median, or mean of the set of cluster number value observations. When we graph numClusters(Dissim) vs Dissim, the largest flat region corresponds to the natural number of clusters; if multiple such regions exist, we may want to consider several that are within our desired range of cluster numbers and choose between them via some other criterion.

One drawback of the method as presented above is that initial data items are assigned to clusters before the final clustering is known. This can be overcome by using a two-stage algorithm: in the first stage, clusters are formed as above, but only one data item is assigned to each cluster (i.e. *data_i* is only assigned to a cluster if this would result in the formation of a new cluster); in the second stage, all unassigned data items are assigned to their closest cluster.

This method is very similar to the Nearest-Neighbor method used in machine learning, with the difference that in machine learning each data item has an associated class. A more

robust variant has been found to be k-Nearest-Neighbor, where an item is assigned to the most common class among its k nearest neighbors.

Hierarchical Clustering

Hierarchical clustering algorithms produce a hierarchy of nested clusterings. There are two main categories: agglomerative algorithms which start with N clusters of one data item each and join clusters to end with one clustering, and divisive algorithms which start with one clustering containing all N items and split clusters until finally N clusters of one data item each result.

The **general agglomerative algorithm** is as follows:

Make N clusters of one data item each.

For clusterNum = N-1 to 1 {

Check all pairs of clusters, and merge the two clusters that are most similar.

}

Two auxiliary data structures are commonly used: the pattern matrix (i.e. the N x N matrix of similarity values between data items), and the dendrogram, which is a graphical tree representation of which clusters get merged at which time step. Cutting the dendrogram at any given level results in the clustering for the corresponding time step.

There are two main categories of agglomerative algorithms:

1) **Matrix methods.** These methods use the similarity matrix, and update it at each cluster joining so that the matrix always represents the similarity between all current clusters. (At each step, two rows and columns for the old clusters are deleted, and one new row and column for the new cluster is added.) Time complexity is typically between $O(N^2)$ and

$O(N^3)$.

The key difference in the variants of matrix methods is the different update function used to specify similarity distances from the newly-formed cluster. Let $C_{\text{new}} = C_{\text{new1}} \cup C_{\text{new2}}$ (i.e. C_{new} is a newly-formed cluster made by joining C_{new1} and C_{new2}), and let C_{old} be any other old cluster. Then a general equation for the similarity measure Sim which includes many common variants as special cases is:

$$\begin{aligned} \text{Sim}(C_{\text{new}}, C_{\text{old}}) = & a_1 * \text{Sim}(C_{\text{new1}}, C_{\text{old}}) + a_2 * \text{Sim}(C_{\text{new2}}, C_{\text{old}}) \\ & + b * \text{Sim}(C_{\text{new1}}, C_{\text{new2}}) \\ & + c * | \text{Sim}(C_{\text{new1}}, C_{\text{old}}) - \text{Sim}(C_{\text{new2}}, C_{\text{old}}) | \end{aligned}$$

Letting n_{new1} and n_{new2} be the number of elements in C_{new1} and C_{new2} respectively, several common algorithms can be succinctly described as special cases of the equation above:

Single link: $a_1 = a_2 = 1/2, b = 0, c = -1/2$.

In this case, $\text{Sim}(C_{\text{new}}, C_{\text{old}})$ reduces to $\min(\text{Sim}(C_{\text{new1}}, C_{\text{old}}), \text{Sim}(C_{\text{new2}}, C_{\text{old}}))$. This tends to form isolated, elongated clusters.

Complete link: $a_1 = a_2 = 1/2, b = 0, c = 1/2$.

In this case, $\text{Sim}(C_{\text{new}}, C_{\text{old}})$ reduces to $\max(\text{Sim}(C_{\text{new1}}, C_{\text{old}}), \text{Sim}(C_{\text{new2}}, C_{\text{old}}))$. Complete link tends to form compact clusters, but they may not be as well-separated as single-link clusters.

Unweighted pair group average: $a_1 = a_2 = 1/2, b = c = 0$.

Here $\text{Sim}(C_{\text{new}}, C_{\text{old}})$ reduces to $1/2 (\text{Sim}(C_{\text{new1}}, C_{\text{old}}) + \text{Sim}(C_{\text{new2}}, C_{\text{old}}))$, i.e. the average of similarities between the newly joined clusters and the old cluster.

Weighted pair group average: $a_1 = n_{\text{new1}} / (n_{\text{new1}} + n_{\text{new2}}), a_2 = n_{\text{new2}} / (n_{\text{new1}} + n_{\text{new2}}), b = c = 0$.

Similar to unweighted pair group average, but weighted by the relative size of each of the two clusters which are merged.

Median: $a_1 = a_2 = 1/2$, $b = -1/4$, $c = 0$.

This can be thought of as a function of the unweighted average of the centroids of $C_{\text{new}1}$ and $C_{\text{new}2}$; it implicitly uses a Euclidean distance metric for similarity.

Single and complete link are two extreme cases, and many other compromise parameter values can be taken. One slightly different algorithm is the minimum variance algorithm; if we define the variance of each cluster as the sum of squared distances from the cluster mean, then the algorithm joins the two clusters which lead to the smallest increase of total variance of all clusters.

2) Graph methods. These methods use ideas from graph theory to choose clusters to join. A graph can be made from the data items by associating a node with each data item; weights on the links are then set equal to the similarity between data item pairs. This complete weighted graph can be simplified to an unweighted “threshold graph” by eliminating all links with similarities below a given threshold (choosing this threshold value is often problem-dependent).

Clusters are then merged based on maximizing the usual similarity measures, with the additional proviso that the threshold subgraph formed by the potential new cluster’s nodes possesses some graph property. Typical properties include:

- **Connectedness.** The constraint on merged clusters is that the subgraph be connected.
- **Completeness.** This is the graph theory version of “complete link”.
- **Intermediate constraints.** These can be some requirement about node connectivity, node degree, or edge connectivity.
- **MST (minimum spanning tree) algorithm.** The weighted MST of all nodes is formed,

and a hierarchical nested clustering sequence is derived by successively removing edges in ascending order of similarity.

Once we have our agglomerative hierarchy of clusterings, the best clustering can be chosen via domain-dependent preferences. Two such methods are i) look in the dendrogram for clusters with large lifetimes, and pick the clustering when all (or most) such clusters have formed as the “natural” clustering; ii) use heuristics such as ensuring that the within-cluster variance is less than the between-clusters variance.

Divisive algorithms are the converse of the agglomerative method, starting with one all-inclusive partition and then splitting partitions until all data items are separate. At each step, all possible partitions of clusters are checked to find the one which would yield two clusters of minimum similarity (since we want the most dissimilar sets of items to be in separate clusters as soon as possible).

Of course, checking all partitions of clusters is computationally infeasible, so in practice some prespecified algorithm is used to examine only “reasonable” partitions. One sample method of splitting $\text{Cluster}_{\text{old}}$ first moves the least similar item to $\text{Cluster}_{\text{new}}$; it then moves data items from $\text{Cluster}_{\text{old}}$ to $\text{Cluster}_{\text{new}}$ as long as their average similarity to $\text{Cluster}_{\text{new}}$ is greater than to $\text{Cluster}_{\text{old}}$ (moving the item with greatest average similarity to $\text{Cluster}_{\text{new}}$ each time), and then stops when no items from $\text{Cluster}_{\text{old}}$ meet the criterion.

Another way of reducing the partition space to examine is “monothetic algorithms” which split clusters based on only a single data item feature at each step. Although such methods can lead to 2^{NumSteps} clusters, they can be useful for approximate splittings or small-dimensional spaces. (Decision trees are a practical example of monothetic algorithms in action.)

Further Methods, and Neartree-Specific Clustering

Other clustering methods include fuzzy clustering (where cluster membership is a fuzzy instead of hard function), mixture decomposition schemes, Isodata, k-means, vector quantization, binary morphology, boundary detection, competitive learning, and function optimization clustering methods (using e.g. branch-and-bound, simulated annealing, EC) where some global fitness function is defined and maximized for clusterings.

For large data sets, it may help to group the population into subsets with similar goals, abilities, or problem-solving approaches, and then reduce the problem space by acting only on a representative from each subset. Practically speaking, k-means is a natural method for large-scale data sets - this is because its time complexity is linear, once the number of clusters k and number of passes has been fixed. (Multilevel hybrid algorithms have also been used with large data sets.)

The k-means algorithm is quite simple:

Choose k cluster centers at random.

do {

Assign each data item to the closest cluster.

Recompute the cluster center (through e.g. centroid of its data items).

} while (clusters have not converged, e.g. data reassignment still happening)

Isodata is a less efficient but better performing modification of k-means which adds splitting and merging. At each time step, high-variance clusters are split and pairs of clusters with centroids closer than a prespecified threshold are merged.

Kohonen maps (also called self-organizing maps) efficiently make 2D visual maps of data items. They are a version of competitive learning, in which k representatives are

presented with each data item in turn and the winner (i.e. closest representative) moves closer to that data item at some learning rate. Self-organizing maps have the additional property that other representatives in a neighborhood of the winner also move toward the presented data item, with the neighborhood shrinking over time; this tends to produce a more topologically ordered set of representatives.

As we've seen, clustering is a relatively assumption-free method for decomposing large data sets into similarity classes. Clustering can be used for more semantic spaces as well, as long as data items can be approximated by vectors and a suitable distance metric can be found.

We expect that clustering algorithms will be key in the **formation of Neartrees from data**. Two natural approaches are i) to specialize the hierarchical clustering algorithms to the case of Neartrees, or ii) to leave the specification of levels to a human or other algorithm, and use single-level clustering at each node to specify child nodes.

Metrics to determine the fitness of a created Neartree could be similar to those outlined in the previous section, "How Hierarchical?". Next and most challengingly, good adaptive heuristics are needed to find such good Neartrees. One expects this to be substantially more difficult than tree evolution algorithms (e.g. clustering hierarchy formation, decision trees), due to the additional links which destroy the tree property - as the number of additional links increases, the number of paths between lower and upper level nodes increases very rapidly.

Regulatory Genomic methods as suggested in [Masum and Oppacher 2001] are one potential method of creating Neartrees. They might provide a compact representation of phenotypic properties that can "locally differentiate" the tree based on local conditions.

The morphogenetic structure of organisms, both biological and artificial, can typically be represented via some recursive grammar - but the execution of a path on this grammar in turn defines a near-hierarchical graph whose nodes are atomic structural elements.

Relatively simple local heuristics can create long-range global order and structure in an organism, in a fashion that adapts to the environment in which it grows; examples can be found in [Coen 1999].

L-systems are similar algorithms that were developed as parallel string rewriting systems, where a set of rules are repeatedly applied to modify a string - which could represent a visual object, such as a fractal or a plant [Prusinkiewicz 1996]. A similar process could be applied to grow Neartrees, with the additional factor that the choice of rules to apply at each stage would depend on the "local environment" at each node. For example, a rule telling a node to differentiate and grow several children might be invoked when a node either gets too complex or is found to contain several identifiable subsets. Conversely, a group of nodes would give rise to a parent node when an observable salient property of the group can be modeled in terms of its constituent nodes.

In this case, the organisms typically are evolved against some fitness function from an external environment: beauty in the eye of humans, internal consistency, structural integrity, or performance in a competition. With these fitness functions and embryogenetic mechanisms, it may be possible that the Neartree structure can be "grown" from a small seed structure.

The basic operations in growing a Neartree will probably be expansions and contractions. An expansion occurs when a node differentiates into subnodes; contraction occurs when there is an adequate model for the value of the parent node without referring to its subnodes (i.e. one can directly predict or optimize the parent node without needing the subnode values). Note that these dynamic reconfigurations may be useful in any case where the Neartree structure evolves over the course of solving a problem, e.g. through learning a better decomposition. In this case, nodes may spend some time exploring their current local environment, which will affect differentiation and contraction strategies.

10.3 Dynamic Clustering and Judging with Neartrees

In this section we outline a general class of problems for which the Neartree framework seems suitable: dynamic clustering and judging.

We assume that the source process is clustered, with a hierarchical structure to the clustering; dynamic but with cluster membership changing relatively slowly; and contains inter-point communication which roughly matches cluster membership.

Goals for a Dynamic Neartree Clustering Algorithm

In practice, finding the absolute best clustering may not be important, and real-life clustering domains are often dynamic. There has been some recent research on approximability results and algorithms for dynamic clustering:

- [Aslam et al 2000] discusses the star algorithm, a simple clustering method for which the number of clusters doesn't have to be prespecified. [Alon et al 2000] describes a method for approximate clustering via sampling the data points, that gives clusterings with near-optimal cost of all but a small fraction of the points.
- [Procopiu 2001] surveys several definitions of clustering problems, with applications and approximation results. [Agarwal & Sharir 1998] surveys efficient algorithms for geometric optimization, including in particular clustering problems; [Agarwal & Sen 2001] is a similar effort focusing on randomized algorithms.
- [Charikar et al 1997] introduces incremental algorithms for dynamic clustering with provable performance - a sample application of dynamic clustering is topic detection and tracking. Theoretical success metrics for clustering often don't correspond well to our intuitive notions of what a "good" clustering is; [Kannan et al 2000] applies the graph-theoretic notions of expansion and conductance to "spectral clustering".

Let's consider generalizing traditional clustering methods, to outline new methods suited for combination with Neartrees. (We haven't come up with a detailed solution - instead, the intent of this section is to present a high-level view of how further development of Neartrees could solve an important problem.)

Such an algorithm should have several properties. First, it should be fast (since it must operate in large spaces), sensitive to the different topology of high-dimensional spaces, and (near-)hierarchical for reasons outlined elsewhere in this thesis. Preprocessing may be necessary - in case the problem as stated does not have data points in a metric space, the problem space and inferred metrics will have to be translated.

An incremental algorithm is necessary, so that if the problem is slightly changed, the algorithm can in most cases easily update the previous solution without needing to start over from scratch. Finally, the algorithm should be online, so it doesn't need to always run to completion, but instead iteratively improves the best solution so far and always has a candidate solution ready in case the user unexpectedly requests an immediate answer.

This clustering algorithm could be viewed as a preprocessing stage which creates a Neartree from raw data; without it, one would need to specify clusters (corresponding to nodes) for a Neartree problem ahead of time. However, in this latter case it would be more difficult to handle the dynamic situation - without the proposed dynamic clustering method that is intended for maintaining clusters when the data changes, one would need to update clusters manually, which could be very time-consuming.

Distributed Evolving Neartrees: Ontological Order from Chaos

Let's look now at theoretical and practical considerations in the proposed dynamic clustering algorithm. The essential problem is to cluster data points. However, we generalize the traditional clustering problem discussed in a previous section in the following ways:

- Data point location contains some error.
- Data point location is dynamic.
- Data points may contain additional attributes aside from just location in some metric space (e.g. representing “performance” of that data point on some test problem.)
- In addition to the points themselves, there is some inter-point “flow” (representing the flow of information, resources, control, etc).

So in addition to looking at static clusters, we’d like to efficiently track cluster movement, examine the inter-point flow, and characterize the aggregate performance of clusters.

Our goal is “Dynamic Near-Hierarchical Clustering under Uncertainty and Multiple Points Of View”. The basic implementation idea is to use a combination of hierarchical k-means, Self-Organizing Maps, and “cluster watching” (i.e. monitoring changes to clusters and reclustering data as appropriate) as a first approximate algorithm. Then, track data changes: if a data point moves but is still within its local cluster, do a re-organization at that cluster only, otherwise re-organize at progressively higher cluster levels as needed.

(In order to get a fast approximation to check at what level reclustering is necessary, use “blurring” of the data set via sampling or averaging, or use a moving average of deltas in overall mass distribution. Uncertainty in the position of a data item can be dealt with by adding jitter (i.e. random noise) with variance proportional to uncertainty, or blurring in the clustering.)

A modular approach suggests itself, with separate modules being as functionally independent as possible. We suggest the following set of modules as a starting point:

- **DataSupplier:** Provides a set of data points; provides their movement and uncertainty in the dynamic case.
- **Clusterer:** Queries the DataSupplier, and forms clusters based on replies - classifying and partitioning the data into relatively independent and coherent subparts.

- **FlowSeeker:** Forms and analyzes a flow graph, for interpoint flow and / or aggregate point movement. This will probably use the Clusterer's representation to form aggregate nodes, but may also query the DataSupplier if necessary.
- **TreeGrower:** Based on the Clusterer and FlowSeeker representations, grows a Neartree for the problem space. The Neartree is a compact representation which may point out important properties of the problem.
- **Judger:** Given qualities of the problem space as sensed through the Neartree, as well as derived qualities based on analysis of point information passed up through the Neartree, the Judger performs description, information aggregation, and decision-making. The main functions of the Judger include information filtering and focusing attention on productive parts of the problem / solution space. (Multiple adaptive Judges might exist.)

We look at the first few modules above in a little more detail:

DataSupplier takes data in from a data source and supplies it to the other modules. A synthetic source for test purposes could be a hierarchical process, where data points are both dynamic and have inter-point communication. Static parameters include:

- Number of levels.
- Branching factor per level.
- Number of points per cluster.
- Space characteristics (dimension, metric).
- Spread of clusters (PDF, variance).

Dynamic parameters include:

- Speed of top cluster.
- Relative speed increase of each lower level.
- Movement function (i.e. path taken by each point, noise, inter-point force function).

- Measurement error.
- Point-to-point flow function (as a big matrix for observed sources, or a function for synthetic sources).

The **Clusterer** forms candidate clusterings of the data, using e.g. variants of the algorithms discussed in the previous section on Clustering Algorithms. These clusterings are fed to the FlowSeeker and TreeGrower, which attempt to find persistent information flow and structures.

The **Flowseeker** module finds the main inter-point “flows” in the process. Flow algorithms from combinatorial optimization and graph theory will be useful. A simple calculation shows that, even if only two-point flows are allowed, the number of possible flows to monitor is quadratic in the number of points. Hence we expect to use three strategies in this stage:

- Resolution coarsening: Inter-point flows can be aggregated by e.g. octants, and the main inter-octant flows found; this process can be repeated recursively as long as only a few flows exist at each level.
- Randomization: If we imagine the inter-point communication as a Markov chain, a “communication packet” would spend most of its time in the high-frequency flows; hence we can simulate random packets and keep track of where most of them go.
- Focusing: The user may focus on only parts of the process, and the algorithm may focus only on clusters which are relatively large or well-defined.

The **TreeGrower** creates a Neartree representing the underlying data set, using algorithms such as those outlined in the previous section on Neartree-specific clustering.

Applications and Model Extensions

Potential applications of this scheme include:

- Recommender systems: clusters correspond to group or preference affinities and inter-point communication corresponds to communication between related objects or movement by a searcher through related objects.
- Information organization (e.g. file systems). At a longer time scale, building ontologies would be a very relevant application; such ontologies usually have a Neartree-like structure, for ease of generation and human comprehension. Two common and important examples are those generated by major search engines and those that have been developed over decades by the information science community. [Svenonius 2000] discusses foundational issues in this discipline.
- Bioinformatics: organism or gene-function taxonomies, where inter-point communication corresponds to e.g. observed inter-organism or gene product reactions.
- Financial market evaluation: economies are subdivided into sectors, subsectors, companies, lines of business, projects, and so on. There are also more abstract separations by creditworthiness, time frames, and many other properties.

Another interesting application would be to use this framework to control the exploration of some complex object space. For instance, in an EC system the genotypes would be the data points, and a hierarchical exploration through parameter space would result; constraining evolution to take place via clusters might increase the comprehensibility of the process.

In many dynamic situations, nodes are not isolated - resources, information, or activity may flow between nodes. Patterns of these flows can be used for cluster formation, since those nodes which interact strongly are likely to form a functional cluster. If both flow and an underlying metric space are present, a continuous extension of Neartrees might be possible (“Nearthree Flows”).

Real-life situations also have uncertainty: data points may not be perfectly known, due to measurement or perception errors. An optional feature could then be to add a distance-proportional “sensing error” to Neartree nodes: the closer nodes get to the points they represent or control, the more accurately they sense and operate on the points. The rationale behind this is that it is a more realistic representation of reality - we usually know more and can better control those objects which are “close” to us.

Usually large hierarchies have smaller spatial resolution and faster time resolution as one travels away from the root. Hence a useful metaphor for picturing dynamic clustering is a “**blurtree**” - a large hierarchy with upper levels relatively stable, lower levels changing more quickly, and the lowest levels being a blur of movement from the point of view of an upper-level observer. Since our nested hierarchical model implies that upper-level nodes will usually contain a coarse-resolution representation of the nodes below, if resources are limited one can rapidly operate with a large Neartree or blurtree by considering only the top few levels - scaling the computational accuracy to available resources.

10.4 Valuation

Valuing a Portfolio, or a Planet

Valuation and risk management are core functions in our civilization. What do we mean by “value”? Typically, the process used is to use a combination of theory and experiment to find a usual range of values for some scalar-valued parameter; this parameter describes some observable, repeatable, and in some sense “characteristic” property of the system being observed.

However, the process of parameter deduction, estimation, and inference is complicated by many factors. One is typified by multiobjective optimization: in many cases, a single parameter will not adequately describe the system. (One could define “adequately describe”

operationally: if some decision-maker is given the parameter values for several instances of the system, these parameter values are insufficient to decide which system instance will perform better on some measurable task. In other words, the parameter value is inadequate as a summary of the performance of the system instance.)

In such a case, multiple parameters are usually the next step up in sophistication, but here one runs into a problem of comparing separate instances with regard to these multiple parameters. At the end of the day, a decision may need to be made on which instance is “better”, and increasing numbers of system parameters make this increasingly difficult.

Adding valuation to our Neartree model as outlined in this section may be one answer to restricted cases of this problem. As the best-developed and tested theory of valuation on the planet, finance and economics have a lot to teach algorithm designers about the issues involved in valuation -and the possible strategies that can be used to value complex objects.

A typical approach to stochastic valuation is the use of Monte Carlo methods; see [Vose 2000] and [Thompson 2000]. In the simplest case, a current value is assigned as the equation $\sum_{s \in \{\text{Future States}\}} \text{Pr}(s) * \text{Value}(s)$; an entire industry has grown up around estimation of terms in such equations. (Note that stress testing of a hierarchical portfolio is a case where successive levels of the Neartree can represent partitions of successively later points in time, as well as the more natural partitions of greater spatial or detail resolutions.)

The valuation of an investment is an extremely complex affair - essentially, one tries to assign a current value to a probabilistic stream of future values. These future values are dependent on many factors, some of which can be estimated only imperfectly if at all.

In the domain of R&D valuation, [Boer 1999] discusses the theoretical and practical difficulties involved. Consider hierarchical portfolio stress-testing, where one constructs a portfolio of securities, R&D projects, and so on. The idea is to test its overall value under possible future scenarios, under various assumptions about i) internal synergies that make

the whole portfolio more (or less!) than the sum of its parts, and ii) external environment properties, e.g. testing the upside / downside given a plausible range and PDF for several key variables in the future.

We can generalize this to representing and dynamically valuing an economy, or even a planet, e.g. [Costanza 1997]. Advantages of such a large-scale valuation:

- Defining a relatively small set of metrics, and widely-agreed desired states in terms of those metrics, would provide an objective and quantitative goal to aim toward in the conduct of human affairs. Currently such a quantitative metric is widely used only in economics, and even that imperfectly.
- Understanding how achieving subgoals contributes to achieving overall desired planetary metrics will make valuing and acting toward those subgoals more probable.
- It is widely accepted that the relegation of social and environmental goods to unmeasured externalities by conventional economics is a primary cause of irrational policy-making. The field of environmental valuation attempts to quantify some of these variables, e.g. [O'Connor & Spash 1999]. Valuation of these goods - and incorporation of valuations into decision-making - will likely have a large multiplier effect in directing human activities toward more healthy and sustainable goals.
- Humans have a very small cognitive capacity, measured in terms of working memory, processing rate, etc. The proliferating conflicting constraints to be met in complex systems quickly overwhelm even the best-prepared brain, unless effort is made to tailor the problem-solving strategy to our cognitive limitations. Placing a planetary model in extra-somatic memory - as a transparent and decomposable model that exists as an object in our intellectual commons - allows us to focus on the individually-tractable task of solving small subgoals separately. It also allows collective and computational co-operation in reconciling large numbers of conflicting constraints, and aids mutual understanding of the assumptions being made in the decision-making process.

One approach to this valuing-the-planet problem is to represent and connect a set of sub-

valuations as a Neartree. Of course, there will be a variety of approaches to this problem, and many different Neartrees would be possible. To the extent that a particular node exists in multiple Neartrees with the same context (interface with children and parents), it represents a sub-problem whose importance is agreed upon by multiple parties.

How could values be aggregated, once the data is represented? The simplest algorithm might be to define the value of each node as a simple mathematical function of the values of its children. (This is the method often used to combine environmental or social indicators into composite indicators.) A somewhat more general method would be to represent each node value as some relatively simple function of its children's values - for example, a weighted boolean formula with each clause having a specified weight, or a linear programming problem. There has been some work along these lines in the field of Integrated Assessment, see [Abaza & Baranzini 2003]. (We see the Neartree-valuation approach as potentially complementary to simulation-based approaches. Although the latter give a much more detailed model that can be explored to infer causality and build intuition, Neartree valuation may provide a useful middle layer, which can summarize statistics and indicators of the full model for policy-makers and the public at large.)

An application of these ideas is the development of a "Human Composite Index", which would be a mutual fund or composite index incorporating all investable asset classes on our planet (and elsewhere once human civilization starts to spread). Portfolio theory (see e.g. [Reilly & Brown 2002]) suggests that risky assets which are not perfectly correlated can be combined to form a composite asset with reduced risk, and that this process encounters diminishing returns after a fairly modest number of assets. (A simple definition of risk in an asset is the unexpected variance of price with respect to time. Most of the time the semi-variance is a better measure, as it looks at only the variance due to negative outcomes - usually what is relevant, since unexpected bonuses are welcome.)

To the extent that this is the case, each asset in a maximally-diversified hierarchical portfolio containing all investable Earth assets could be naturally represented by a node in

a Neartree; the levels in this Neartree would be different scales of economic activity (e.g. country, sector, company). This Neartree would be a very interesting financial data structure, containing within it assumptions about the interrelationships of asset classes. Aside from the obvious financial utility should it turn out to be a useful representation, it could also be of help in economic planning and analysis.

Risk Analysis and DisSATisfaction

Quantifying uncertainty and risk was a crucial theoretical advance promoting large-scale and long-term human endeavors, as chronicled in [Bernstein 1998]. For example, the insurance industry is essentially a risk-sharing institution, which allows catastrophic risks to be mitigated via shared planning and up-front payments. Instead of hoping to avoid catastrophe, such occurrences can be converted to an estimated cost with much lower variance. (As an indirect benefit, actuarial statistics can be used to help identify high-risk activities and situations, which may then be modified or avoided through incentives; think of differentials in car insurance rates.)

Futures and option markets can similarly be used to hedge against uncertainty, making it possible for the buyer or seller of a commodity to limit future losses; [Hull 2002] is a comprehensive overview. Speculators are motivated to take positions in such markets by the potential profits if their risk-estimation and prediction skills are substantially better than average. Again, prices of futures and options often provide a valuable service by quantifying the massed opinions of many individuals.

Often our goal in finding a solution may not be to maximize the expected case, but to minimize the worst case. How can our hSAT framework be applied to minimizing risk in an uncertain future - or to an adversarial case, where an opponent may be searching for weakness?

Instead of optimizing an hSAT, we might more often be interested in strategies for

“pessimizing” DisSAT so as to bound loss. Analogous situations include credit risk and disaster prevention; pessimizing is a rational strategy if the future is uncertain enough and the downside is bad enough. This is an important issue in risk analysis; an accessible discussion is in [Dembo & Freeman 1998].

To analyze the fitness value of an interim or final solution, we may form a fitness estimate based on the value of different node subsets. There are several choices for our goal:

- Get the overall system (i.e. root node) into a desired state. Keep it in that state if the environment is dynamic; keep it as close as possible (or minimize the expected variance or expected maximum loss) if we are risk-averse or planning for the future.
- Each node gets a reward when its local SAT subproblem is solved, with higher rewards for upper-level nodes. More generally, different states have varying rewards, and we want to maximize rewards received (i.e. similar to reinforcement learning).
- Noise may change random values, or an adversary may perturb or clamp chosen nodes to chosen values. What’s the best strategy (measured as above, e.g. average state over time or worst state reached)?

Strategies for Risk-Averse and Adversarial Cases

In the risk-averse case one wants to maximize a changing position under assumptions about the dynamic process.

First, there are obvious methods to try when optimizing any dynamic as opposed to static problem (where “dynamic” includes combinational algorithms changing, certain nodes being clamped to externally-generated stochastic values, etc):

- Change the value function, to see how more and less strict value functions (e.g. with child success more or less necessary for parent success) influence solution time.
- Capture a value back to the root node, to estimate the value of the root under current time

/ space / state decomposition.

- Level slowdown: in real hierarchies, often time runs k times faster at each lower level. (In fact, this effect emerges naturally when we use periodic-resolve cache, since each node gives its children several ticks to reach their targets before itself retargeting.)
- Experiment with evaluation noise and misunderstanding of state...solutions should be robust with respect to small errors (“stochastically stable”).

Next, there is the risk-averse case where the goal is to find a good general strategy for minimizing expected or maximum loss over a sequence of positions. This is risk-assessment without a malicious adversary:

- Accurately value the expected value of a current Neartree (e.g. option valuation).
- Trade off current situation optimization vs states which reward less now but more in likely future states.
- Specify or learn an arbitrary PDF for negative outcomes based on experience, for assigning utility to outcomes (which in turn implies a risk measure). This is a better approach than typical statistical tests, which may require or assume a PDF from a specific mathematical class for analytical mathematical tractability.
- Guard key nodes; precompute alternatives, and switch to them if the value delta in minmax terms is high enough.

It's crucial to assign meaningful values at nodes; one can then estimate improvement in expected value per unit input, to help allocate resources. One specific valuation method is the minimum value in a given neighborhood of current state. (For hSAT, this could be all the states that can be reached by at most k flips of a truth value assignment.) Beyond this, one can look at statistics of the value that the node takes over time in a set of simulations.

The difference between the risk-averse and adversarial cases can easily be visualized by considering the branching tree of future possible states of our process, where each branch represents a stochastic event. In the risk-averse case, the branch taken is chosen randomly

from a prescribed PDF. However, in the adversarial case, the worst branch is always taken. (One can picture a malicious demon at each branch who always decides which branch you go onto. This is essentially the minmax idea used in lookahead search trees for games like chess, and in game theory.)

Finally, there is the adversarial case (e.g. someone wants you DisSATisfied):

- Noise (knock one out) - e.g. make $k\%$ of nodes “liars” which misreport their state. This models the case of imperfect or deliberately false information.
- Sabotage - an adversary chooses $k\%$ of nodes to change that will reduce your value as much as possible.
- Vulnerable graphs - e.g. a node with many parents means knocking that node out has a large effect.
- Critical path and key nodes: an adversary knocks out those nodes which are most difficult to satisfy, or have most estimated impact on overall outcome.
- Countermeasures - choose the configuration which is safest from noise or sabotage.
- Worth of avoiding bad states. Value states by median value, or current / likely / realized neighborhood (with differing estimators of the future, this can give different valuations on the same Neartree state, and these valuations could be evolved).

DisSAT Theory, and Reliable Networks

This framework could be modeled mathematically as a hierarchical process. If we can estimate e.g. the mean and variance of the time required to complete each subproblem, it should be possible to analytically derive estimates for the time required to complete the entire composite problem. One might wish to assume a normal distribution of solution times for analytical tractability. An even simpler approximation is to consider all the nodes as having some specified polynomial solution time, in which case our “Optimizing Parts vs Wholes” mathematical framework is applicable.

Related work includes hierarchical control theory in engineering, and the “Beer Game” and other simulations of hierarchical systems in business. There has been some work on Stochastic Boolean Satisfiability in [Littman et al 2001], and a related formulation of Stochastic Constraint Programming in [Walsh 2002]. Sorting networks and circuits have been evolved with reliability in mind in the field of Evolutionary Computation, as we discuss later.

In a now classic essay, “The Architecture of Complexity”, Herbert Simon in [Simon 1996] argues that modularization means failure is firewalled. He illustrates the point with the story of two watchmakers who are continually interrupted in their work: one restarts from scratch, while the other builds components modularly and so loses only work on the most recent component. However, this argument needs extension to the truly hierarchical case, where upper-level nodes have independent meaning - in this case, failure in an upper-level node, while still somewhat bounded, is more costly than lower-level nodes. In fact, the true effect of a failure depends on the connectivity of that particular model - which is bounded in the case of a Neartree.

We also note that there is a natural connection with the theory of network reliability and survivable networks, whose mathematical and algorithmic features are outlined in [Ball et al 1995]. In network reliability, a network consisting of nodes and edges is analyzed with respect to the probability that a set of nodes remains connected, when both nodes and edges are subject to failure with some specified PDF.

This naturally suggests the analysis of Neartree reliability, including questions like: how does the Neartree react if a single node is knocked out? How does the importance of adding redundancy to a node change with respect to its level, children, and weight? How much more robust are multiple-parent Neartrees? What is the optimum balance between the pressure to have more parents due to increased reliability, and the pressure to have less parents due to difficulty of satisfying all of them? Natural applications include designing robust communications, infrastructural, and social networks.

Finally, there has been a good deal of empirical and theoretical work into properties of large-scale networks - particularly the Internet, and what might be called the “SocialNet” (i.e. the network formed by human interactions in a particular domain). Relevant topics include the ideas of scale-free networks, hubs and authorities, and power-law distributions of properties like node degrees; see [Albert & Barabási 2002]. We expect this line of research to generate practical algorithms for designing robust network structures; specializing robust network algorithms to the case of structures designed for solvability and comprehensibility such as Neartrees may yield a fruitful synthesis.

11. Long-Term Themes, and Conclusion

“Social progress makes the well-being of all more and more the business of each.” -

Henry George

What can we do with hierarchies to help solve the many difficult problems our society faces? Hierarchies offer several advantages: relative cognitive simplicity, comprehensibility, explicability, persistence, and modularity. Features such as simplicity and explicability are particularly important for a technique to be applied by policy-makers; persistence and modularity ease long-term and distributed analysis of complex issues.

As is cogently discussed in [Homer-Dixon 2000], we as a species face an increasing gap in the supply of ingenuity relative to demand, where “ingenuity” is defined as ideas or algorithms that can be applied to solve practical technical and social problems. Note that application is key in this definition: it’s not enough just to come up with a method that would solve a problem, if the method never gets implemented.

In this final chapter, we look at longer-term applications which are relevant for practical policy development and analysis, as well as connections with other disciplines which may prove fruitful in the future. The underlying goals are twofold: to understand and extend our theoretical and experimental results through interplay with real-world problems, and to suggest applications in which our framework could be applied. We then conclude with a look back at what we’ve accomplished, and key open questions.

11.1 Autonomous Nodes

Competition vs Cooperation

We have been assuming that nodes always do what they are told, and have no individual agendas that cannot be superseded through requests from a parent node. However, in many

real-life situations this is not the case; indeed, the primary difficulty may not be solving an overall hierarchical problem so much as promoting co-operation between a myriad of agents with individual conflicting goals.

The theory of games, as introduced by Von Neumann and Morgenstern and elaborated through the last half-century, has as one of its canonical problems the determination of optimal strategies in 2-person perfect information games. The payoff matrix is stationary, and each cell contains two entries which specify the payoff to each player, given that each player made the move to which the row and column of that cell correspond.

This basic framework can be generalized in several different ways. Of particular interest are the case where the strategies of the players adapt with their experiences, as surveyed in [Gintis 2000], and the case where the game is “noisy” in various ways. There is also a theory of combinatorial games, in which play takes place in a geometric space with a well-defined topology (similar to many board and strategy games). [Young 1998] develops some of these issues into a theory of stochastically-stable equilibria of games, and their time to convergence.

Decision theory (and similar variants, notably utility theory in economics) is an attempt to formalize and quantify the factors involved in making decisions, and then (given the assumed formal model) to specify good algorithms for maximizing expected performance. In a similar fashion to game theory, agents are given utilities that they attach to various outcomes; a matrix model along with optional constraints then defines the search space in which agents attempt to maximize their utility. Many heuristics have been devised to solve such models, and much philosophical discussion has been devoted to their domain applicability and logical consistency; see for instance [Joyce 1999] for an exposition of subjective expected utility maximization as a foundation for rational decision making. As [Dawes 1988] shows, environmental uncertainty and human biases cause substantial deviations from this utility maximization heuristic in real observed situations.

A small amount of work has been done on explicitly hierarchical games (the American Mathematical Society even has a small subcategory for such games, “AMS 91A65”). Several authors have considered a Hierarchical Prisoner’s Dilemma; for example, [Fletcher & Zwick 2000] discuss an n-player PD with additional group structure imposed, making a simple hierarchical PD. They show that two parameters - gain in individual utility for defection and the dependence of utility on collective cooperation - largely determine model behavior, and are similar to within-group and between-group forces in multilevel selection theory.

We sketch an outline of how these issues might be modeled in a Neartree-like framework. This would deal with questions like: Do nodes within a level communicate? Is there direct competition or co-operation between nodes? If goals and problems are similar across nodes, can upper-level nodes monitor lower-level nodes for performance, extract the best performers, and “teach” the other lower-level nodes about best practices?

First, the following changes are made to nodes:

- Nodes have some individual utility on their own set of possible states - they may prefer being in some states more than others.
- Nodes care between 0 and 100% about fulfilling their parent requests (with the remainder of their goal being to maximize their state with respect to their own utility). The 100% case is the one we have discussed in this thesis - a typical “centralized dictatorship” of goals.
- The utility or value of a node depends to some extent on its child values - including how well child nodes are able to satisfy their personal utility on their own states. (So, for example, a child node being in a state it really hates to be in will count against the overall value of the Neartree.)

With these changes made, the goal is to maximize overall utility - with node-level constraints on their individual utility in mind. Since nodes will resist moving to a state

which they have a personal aversion to, the range of choices (or requests that a parent node can make) narrows considerably. If child nodes are intelligent enough to anticipate parent requests and / or the actions of siblings, the problem becomes more complex yet - incorporating elements from game theory, for instance.

Each subnode may individually wish to do well, with individual (but similar) goals. In this case, the nodes only put effort into “collective” problems to the extent that they are rewarded or forced. Our goal may be to find protocols maximizing productive collective problem solving.

Further, note that we have been assuming that there exists an objective valuation function for each node, given its state. But in fact this valuation function will often be extremely difficult to compute - as discussed in the section “Local vs Global Value”. Hence node valuation might be partly dependent on the opinions of other nodes. Similarly, node valuation of children might be explicitly dependent on the values of parents, with the intuition being that a low-value parent doesn’t allow its children to achieve their full potential.

There is a strong parallel with hierarchical selection in biology and game theory. As surveyed in [Keller 1999], a good deal of empirical and theoretical evidence suggests that selection pressures can operate on the structures that mediate conflict and co-operation between individuals, along with operating on the individuals themselves. This is an important factor in the evolution and maintenance of prosocial structures in many areas, in the face of seemingly strong incentives for individuals to ignore group interest.

A Simple Model for Explaining Hierarchy Formation

Is it possible to come up with simple models with few assumptions, that explain the formation of hierarchies in so many situations?

We suggest the answer is yes. In this section, we discuss a model which may help to illuminate one aspect of hierarchy formation. Our goal is to demonstrate the creation of hierarchical organization at several levels, with a minimal and natural set of assumptions.

The underlying idea is that agents derive some benefit from co-operation. This benefit represents positive returns to scale, e.g. cognitive simplification or mutual protection. However, as co-operative clusters grow larger and larger in size, there is some penalty that sets in past a certain point. The penalty represents negative returns to scale, such as the cost of enforcement or computation growing with group size (e.g. bounded rationality, trust, or limited social radius in low-trust societies a la [Fukuyama 1996]).

Nodes are the basic objects in the simulation:

- Nodes have some “power”. This represents good stuff of whatever variety.
- Nodes co-operate in varying-size clusters. (We’ll use “node” to refer to clusters of nodes as well as single nodes - which one will be clear from the context. If we want to make the distinction explicit, we’ll use “composite node” and “atomic node”).
- Each node’s goal is to maximize power. There is also a “size” and “cost” at each node. They start off with a size of 1, and the size aggregates as discussed below. Cost is a function purely of size - the rationale being that larger sizes have penalties.
- Thus, the “net Power” (= power - cost) is the expressed power, i.e. the excess of power over cost. This net Power is what the node uses as its fitness metric - both as a unit, and when deciding whether or not to join with another node.

Nodes can take only one action - choosing to join or not to join with a randomly-chosen partner. If two nodes join, they form a composite node (which may in turn go on to join with other atomic or composite nodes).

If a composite node `NewNode` is formed from `Node1` and `Node2`, then:

- $\text{Size}(\text{NewNode}) = \text{Size}(\text{Node1}) + \text{Size}(\text{Node2})$
- $\text{Power}(\text{NewNode}) = \text{PowerFn}(\text{Power}(\text{Node1}), \text{Power}(\text{Node2}))$
- $\text{Cost}(\text{NewNode}) = \text{CostFn}(\text{size}(\text{NewNode})) = \text{CostFn}(\text{Size}(\text{Node1}) + \text{Size}(\text{Node2}))$ (i.e. cost is a function only of the size of a node, and not of its power)
- PowerFn represents positive returns to scale: it's a superlinear function of the sum of the powers of the node's constituent nodes (modeling the benefits of co-operation).
- CostFn represents negative returns to scale: typically the cost of trust or computation grows with group size. Hypothesis: if this negative function grows slowly, we wind up with a winner-take-all case; if it grows quickly, there is more clustering.

The “scale” of a grouping is its size: the number of nodes or composite nodes. Co-operation between composite nodes is, after all, part of what makes real-world hierarchies attractive. They induce less of whatever CostFn represents (since nodes are chunked into composite nodes), but still have the same benefits (as represented by PowerFn). Of course, any rules that induce node linking will create composite nodes - to detect the presence of hierarchies, one has to look at the overall structure of the linking, e.g. the average height of nodes and the distribution of node sizes.

The basic idea is this: Suppose an atomic node and a composite node are considering linking, and a composite node is already at its maximum “natural size” (i.e. adding more atomic nodes will give $|\Delta(\text{cost})| > |\Delta(\text{netPower})|$). Since the composite node will get little or no positive returns to scale from adding the atomic node, it will choose not to.

But suppose two composite nodes consider linking. Then the cost increase remains the same as in the case above of adding an atomic node to a composite node. However, there is a big positive return to scale in netPower. So they may consider linking.

While the above idea seems natural, we have not yet been able to get hierarchies to form without adding artificial rules - like giving composite nodes the option of declaring themselves to be atomic nodes at a higher level once they reach a certain size. (See

Wimsatt's essay in [Matthen & Ware 1994] for a philosophical viewpoint on a variety of ontological and definitional issues of "levels" in complex systems; a simplified model of the kind we propose here will have to carefully distinguish the kind of levels and phenomena that are being demonstrated.) Building such a model is therefore still open for experimentation.

11.2 Algorithmic Valuation and Meritocratic Algorithms

Local vs Global Value

The valuation techniques used in most situations ultimately yield a metric describing some facet of the situation. But how about valuing parts of a situation, with respect to their contribution to the value of the overall situation?

Confounding factors that can make it difficult to see how a local value change affects global value include delay between cause and effect, complex causality, non-stationary environments, hidden or unknown factors, small or dispersed effects, and resource bounds on observation or calculation. In fact, the valuation problem is so difficult that in most real-life situations there are many competing valuation techniques and agents (which is positive to the extent that competition generates better evaluation methods).

The goal of developing better evaluation methods for local value is to make the actions that will increase global value more clear. To the extent this is done, incentives can be put in place to bias local problem-solvers toward increasing global value. Reinforcement learning is perhaps the computational paradigm that has articulated these problems most clearly: [Sutton & Barto 1998] explain how reinforcement learning aims to develop algorithms which can estimate the long-term reward of current action choices.

Note that this problem of allocating credit to good subsolutions is crucial to using

Algorithmic Learning on large-scale problems. As Marvin Minsky put it:

“It is my impression that many workers in the area of ‘self-organizing’ systems and ‘random neural nets’ do not feel the urgency of this problem. Suppose that one million decisions are involved in a complex task (such as winning a chess game). Could we assign to each decision one-millionth of the credit for the completed task?...For more complex problems, with decisions in hierarchies...and with increments small enough to assure probable convergence, the running times would become fantastic.” [Huberman 1988, p 171]

Although human markets at first glance appear to be single-level systems and thus unsuitable for assigning credit in hierarchical problems, a closer glance reveals some intriguing facts. The production of a particular good or service may involve a transaction chain several steps in length, where at each value-added step some product is bought by a firm, modified, and then resold for a higher price; there is an interesting, almost recursive loop here since every transaction step takes place in the same market.

Also note that the complete “value-addition” step actually involves many subsidiary steps (and indeed several other markets using such currencies as internal budgets, reputation, and political power) inside any given business. In fact, it seems that most of the computational intelligence involved in human markets takes place in these intra-business markets, in two linked ways: firstly by the problem-solving abilities and organization mechanisms used by intra-business participants to add value, and secondly by investors in the larger market who examine the intra-business mechanisms of potential investments to assess their expected future profitability.

Scaling Up Fitness Evaluation

While fitness evaluation is also used in other heuristics like simulated annealing, and neural nets, it is most advanced in reinforcement learning and EC (see [Sutton & Barto 1998] and [Baeck et al 2000] respectively) The common element is usually an iterated comparison of candidate solutions with a scalar or vector valued “measure of goodness”.

The Darwinian selection process is the kernel of most EC to date. We conjecture that straightforward selection-of-the-fittest is necessary but not sufficient, i.e. that in adaptive systems that deal with non-toy problems (such as economics, human creativity, and future large-scale EC implementations), there has to be another part to this kernel, and that this other part would be common to a large subset of problem domains. In [Masum & Oppacher 2001] this issue was explored with respect to introducing more mechanisms from molecular biology into EC. Here we consider advantages in evolutionary and hierarchical problem-solving of using a more sophisticated notion of valuation.

There's an analogy with economic theory. In standard economics there are various notions of "equilibrium": given self-interest maximizers and a static environment, they'll eventually converge to a steady state. As is well-known, the classical theory is often not an adequate model of the real world, since many assumptions don't apply to the observed behavior of real people. But even within the model, "eventual convergence" doesn't say anything about "time to convergence" - a factor which becomes important very rapidly as problem complexity scales up.

Similarly in EC, a simple Darwinian GA may "converge" to a reasonable solution in reasonable time, but that convergence time also increases rapidly with problem complexity. Hence we conclude that to actually use a GA-type algorithm for large problems, it's mandatory to have more structure to the solution algorithm. One general way to think about that structure is "incentive engineering": **pushing the selection “inside” large potential solutions via problem subdivision, subproblem analysis, and subproblem recombination**, in order to increase the resolution at which selection operates.

Applying this idea will clearly require a widely applicable method of subdividing large problems, along with a heuristic that can accurately combine valuations of individual subproblems. But in fact, these ideas are exactly those which we have been exploring in the rest of this thesis.

Fitness evaluation in EC can break down for a variety of reasons, which interestingly all have observed analogues in economics:

- Premature convergence at local optima. This is a truism of business: Adapt or die.
- Loss of diversity. A local or regional economy that is overfocused on some market sectors is vulnerable to changes in technology and demand; examples include many developing countries with commodity-based economies, and innumerable examples of regional over-specialization. The same holds true for a specific company.
- Too little or too much selection pressure. Too little selection pressure was one of the main reasons behind the economic inefficiency of command economies. Too much selection pressure corresponds to markets which are so competitive that players cannot afford to invest for the future; many examples of protected industries (e.g. in East Asia) have shown that sheltering a fledgling industry from direct market competition in initial stages is often a good strategy for entering new market sectors.
- Lack of perception of structure in solution space. If regularities are present but not realized, the algorithm searches a much larger solution space than necessary. Market research performs a similar function in product development.

It seems that the “valuation” of population members which is at the heart of fitness-based algorithms is too simplistic to scale well, as it does not consider much available and potentially important information. While this simplicity may be a virtue for small or structurally simple problem instances, it does not scale to large and heterogeneous problems, and does not even address co-ordinating valuations in the case where different members are optimizing different fitness functions (e.g. in the case of either multiobjective optimization or self-directed agents with differing goals).

As an analogy, consider application of a similar algorithm to human performance feedback. Would a football team work well if the only comments from the coach were checks paid out after each game, with amounts based solely on goals scored? How about paying professors proportionally to papers published each year, and replacing the least

productive 10% with new hires? And consider cognitive and social development in a child raised by parents who tally pluses and minuses over the course of a day, and only then give the child an aggregate reward or punishment. Remarkably, this method actually worked in Fogel's evolution of checkers players chronicled in [Fogel 2002], where evolving players received feedback only at the end of each game - but it's doubtful that it would provide a competent method in the far more complex domain of child-rearing...

While human situations are admittedly more complex than algorithmic ones, these examples nevertheless illustrate some pitfalls and inefficiencies in an overly nonadaptive and naive valuation approach. One general method of attack may be to generalize the fitness function and credit assignment routine, and use algorithmic money as a measure of fitness; [Baum 1998] is an intriguing manifesto in favor of this route.

The Design Space of Distributed Meritocratic Algorithms

What do the algorithmic properties of market-based computation and EC have in common? We contend that they are two particular computational instantiations of the same underlying idea: create a population of locally-optimizing actors, and set up feedback structures to reward actions which increase "value".

"Value" is the key measure, and is defined with respect to some normative variables of the system as a whole; examples would be minimizing a mathematical function or surviving in a specified environment in the case of EC, efficiently allocating limited resources or directing actor attention toward areas which increase global actor utility in the long term in the case of market-based computation, and maximizing utility in the case of agent-based finance simulations. As discussed in the earlier section, Local vs Global Value, the issue of deciding or inferring which local definitions of value will mesh to form a structure increasing global value is a deep one, and really at the heart of many tough computational problems (even in the restricted case where we assume co-operation among all agents or sub-algorithms solving the problem).

Given that the computational instantiations of markets and evolution are both particular instances of the same metavaluation algorithm, the next logical step is to understand the design space within which these two instances live. We therefore point out key axes along which such instances can differ, to define a design space:

Table 8...Meritocratic Algorithm Design Space: Markets vs EC		
	EC	Markets
Fitness Fn Heterogeneity	Low - Typically single objective, sometimes multiobjective.	Medium - Combination of multiobjective, group objectives, and individual objectives.
Individual Agent Sophistication	Low - simple variation and local search.	Medium/High - Complex investing strategies.
Persistence	Low - mostly one-off solutions.	High - Markets can last for centuries.
Time variation	Low/Medium - usually static problems, sometimes controllers.	Medium/High - constantly changing market conditions.
“Valuation Network”	Low - usually a single implied “judge” (fitness function).	High - Supplier chains, and many-to-many trading networks.

Although the above comparison may appear to put EC in a negative light, this is somewhat misleading. Real markets implicitly contain human intelligence as part of their “specification”, both as market participants (individual agents) and as market regulators (rule designers and snafu fixers). When computational markets have been implemented without the benefit of this human intelligence, they have typically shown more modest performance, e.g. [Arthur et al 1997] and many later papers on market instantiations.

There has also been very limited use of market mechanisms directly in problem-solving situations more complex than resource allocation, with [Baum & Durdanovic 2000] being a notable exception. On the other hand, EC has shown itself to be capable of tackling complex computational problems, as [Koza et al 1999] demonstrates in the context of Genetic Programming.

We close this section by coming full circle, linking the meritocratic algorithm framework back to Neartrees and hierarchies. The primary insight linking these two areas is this: in order to solve a complex problem, one will typically have to i) partition the complex problem into many simpler subproblems, each of which can be understood relatively independently; ii) model the interactions between these subproblems, as a separate computational structure; iii) rejoin subsolutions to form a solution to the whole problem.

This process has many ancillary benefits as discussed elsewhere in the thesis, such as the generation of subsolutions which can be used for problems other than the original problem, and a more robust and rapid solution methodology. Given a Neartree decomposition, the structure and valuations can themselves become models and data for decision-making. Studying decompositions will aid the study of constructing subsolutions with a “minimal API” so that they are maximally generalizable and reusable.

But in fact, I believe this same insight is crucial to the advance of Evolutionary Computation past its current stage (and also to a better understanding of evolutionary biology). Putting selection pressure only on overall performance in an environment gives a much poorer rate of useful information flow to an evolving organism than a more detailed evaluation - i.e. than an evaluation which talks about performance on separable aspects of an environment. This insight has been repeated by various researchers since at least Minsky’s 1961 paper “Steps Toward Artificial Intelligence” (reprinted in [Feigenbaum et al 1995]).

We thus see that the idea of “valuation” connects with near-hierarchical systems in two

main senses. First, Neartrees are natural data structures for representing and computing the valuation of systems - economic, environmental and other. Second, the general idea of a Neartree may prove to be a useful heuristic in combination with valuation functions in algorithmic learning. It is no accident that many tough computational and social problems are problems of assigning appropriate valuation...for correct valuation is at the heart of decision-making.

11.3 Neartree Networks and TOOL

The Neartree framework provides a distributed framework for specifying and controlling multiple lower-level problem-solving agents. With some general assumptions on the agents, we can meaningfully interconnect heterogeneous agents in a single persistent structure that allows for practical understanding of their overall performance, utilization of their information for decision-making, adaptation to changing environments, and feedback to the agents to express perceived utility.

We consider now extending this framework to multiple points of view. Next, we relate our work in Neartrees to a long-term idea on opinion layers that we have proposed.

Nearthree Networks, Multiobjectivity, and Shared Reality

In the Neartree scenarios, there is one distinguished root node (or “point of view”). However, there are many situations where no such distinguished point of view exists; all nodes may have their own goals and beliefs. To model this, we suggest the idea of a **Nearthree Network**: this consists of a set of nodes each of which may be the root of one or many separate Neartrees, or alternatively a set of overlaid Neartrees.

Each Neartree represents a different point of view. Overlaying all the Neartrees in a domain gives the Neartree Network. In the case where there are many common beliefs

among different nodes, many of the Neartrees will contain similar subNeartrees. Neartree Networks of concepts would be similar to semantic networks, with the important difference that they would be constrained so that outward paths from any node form a Neartree (where “outward paths” can be formed by e.g. taking any node as the root concept and using spreading activation to move outward to related concepts).

Canonical problems where Neartree Networks would be of use:

- Recommender systems, to aggregate opinions and make customized suggestions.
- Consensus finding. Each party’s differing point of view can be represented by a Neartree, with the set of all points of view being a Neartree network. Analyzing the network and comparing similarities and differences between individual Neartrees should point out crucial areas of difference, and the “Neartree Hamming distance” or minimal set of changes in belief necessary to transform one point of view to another.
- A decision-making metalayer. Nodes represent individual problem-solvers, links represent information flow; the goal is to effectively combine subsolutions.
- Filtering good ideas from bad, e.g. [Stiegler 1999].

Consensus trees and supertrees as discussed in [GordonA 1999] are practical methods for finding the intersection of multiple tree-like structures. Perhaps the distance between Neartrees in a Neartree Network (as measured by some form of tree or edit distance) can calculate an objective distance between worldviews.

One key issue that arises in many real-world problems is that of optimizing different and mutually conflicting problem goals. The typical solution is to take a weighted linear combination (or some slightly more complex function) of the function values in each problem dimension, and then use this derived scalar function as the quantity to be optimized. This approach, while having the advantage of simplicity, has many disadvantages: two main ones are the relative arbitrariness of the transformation and the large amount of information thrown away when the linear combination is taken.

There has been some research done into alternative multiobjective optimization schemes in the Operations Research and Evolutionary Computation communities. In EC, [WebEMO] catalogues virtually all work done in the area; while [Ehrgott & Gandibleux 2002] reviews work across several disciplines. [Van Veldhuizen & Lamont 2000] survey several different approaches, and categorize the approaches according to how users articulate their preferences:

- A Priori: User combines the different objectives into a single scalar cost function; this makes the problem single-objective before optimization.
- Progressive: User preference and optimization cycle back and forth. Partial preference information is provided, which results in an interim set of solutions for the user to consider; the user then has the choice of updating preferences, and so on.
- A Posteriori: User is given a Pareto-optimal set of solutions, and chooses from that set.

Multiobjectivity is an important method of problem decomposition: if we can break a single overall problem objective into multiple objectives, and gain some understanding into how those multiple objectives interrelate, we have effectively structured our search space and made it more amenable to a distributed approach. We can perhaps model the various ways in which a problem can be multiobjective as a functional network (let's call it an "objective network"): each node represents a particular objective function to be satisfied, and directed links between nodes indicate that the destination node's objective function uses the source node's objective function output as one input. In the case where the objective network contains multiple disconnected components, no single overall best solution can be found; in this case there are various strategies such as taking the Pareto-optimal frontier.

Another case of interest is where the objective network is a bipartite graph, with all directed links pointing from one relatively small level to the other larger level. This represents a situation where many differing objectives share a dependence on a small common set of objectives; it could be a good model of a large population with differing goals, where the goals are variations on common factors.

An alternative (and perhaps more complex) type of multiobjective optimization occurs when there are multiple “points of view”. In such a case, different agents in the problem space may have different and even directly conflicting points of view. The computational problem then bifurcates into two cases:

1. Individual Happiness: Calculating the best strategy for a particular agent to follow, given the preferences and likely strategies of other agents (e.g. game theory and decision theory).
2. Collective Happiness: If the “rules of the system” are not completely fixed, then there is some room for a meta-level agent to try to choose the particular rules and parameters so as to maximize some collective function of the happiness of the individual agents (optionally in combination with some “higher-level” policy goals). That’s what our “Black Box Boss” of the introduction would aim for.

The second case is “Computational Public Policy”, if you will - in the same vein as Bentham’s “Hedonic Calculus” (introduced in detail in [Bentham 1789]) which strove to quantify policy-making by choosing those actions leading to the greatest good for the greatest number.

The collective happiness case can again be separated into two cases: i) having a separate policy-level layer with separate agents whose goal is to maximize the collective happiness function, and ii) having no separate policy layer but rather an emergent peer-to-peer framework within which each agent maximizes some combination of personal and collective happiness (the latter because of altruism and / or enlightened self-interest). Note the similarity with many practical social situations; this is a very relevant problem in both computational and social domains.

The personal-happiness case has been extensively discussed in many domains, and is basically just optimization from a single point of view; hence we are more concerned with the collective-happiness case. Where policy goal functions are absent, the framework is

suitable for controlling a system where the welfare of the constituent agents is the prime goal, i.e. where each agent is a human user or proxy thereof.

On the other hand, when a policy goal function is present, we are satisfying the goals of individual users while simultaneously working toward goals which will either lead to long-term benefit for users (if we think we are better able to do so than users due to farsightedness or impartiality) or to benefit for the policy-level agent (if we are using this framework for e.g. computational maximization, where agents pursue their individual goals as part of a strategy to solve some overall problem). It may be advantageous in such collective-happiness situations to incorporate market-like mechanisms, to keep track of both individual and collective goal optimization progress. Note that in markets, each agent is usually simultaneously optimizing several different objectives, and different trading agents normally have differing sets of objectives - leading to a natural multiobjective optimization situation which might be naturally simulated with multiple computational currencies.

Now let's consider using multiobjectivity and multiple points of view with Neartrees. The basic Neartree has only a single point of ultimate decision: the root node toward which all links flow. There is a single decision or solution to be found, and the structure of the Neartree itself (i.e. the problem components which are chosen to be nodes, and the links which are chosen to be important functional relationships between nodes at different levels) is determined with this ultimate goal in mind.

However, in real life there is often no single privileged point of view. For instance, economies and multiuser programs both contain multiple actors with divergent goals, and typically the designers and regulators of the system are (at least in the ideal case) trying to optimize some overall welfare function for all actors. And in many research domains, different subgroups hold a number of theories or models as to how a system works and which parts of it are important. How can this sort of situation with multiple points of view be reconciled in an objective way?

One idea is to generalize, by combining multiple Neartrees. With intentional nodes, each of which has its own Neartree converging on it, we get the idea of an overlaid Neartree structure; to the extent that nodes share definitions of other nodes and an understanding of their interconnections, these Neartrees will overlap, forming a Neartree Network. (Efficient algorithms for storing such an overlaid network might be derived from “trie” data structures.) Typically, given rational actors of roughly similar abilities and goals, there should be a good deal of overlap between Neartrees; extracting these overlapping portions could provide good information on thought-flows, consensus, and crucial points of difference.

Markets may be characterized as one of the class of nodes with large outdegree, which also have large indegree - analysis and action routers. Perhaps markets should only be allowed as Neartree leaf nodes; since they usually contain far too many players for the root to track, they can be used as functional inputs for efficient resource sourcing. They might also be used in internal nodes, with the current k best performers becoming the k inbound links to the Neartree for that node. As we foreshadowed in our Valuation discussion, EC-style population competition and selection can be thought of as a similar kind of switching node which chooses a few best performers from a large set to become part of the Neartree.

Note that the decision scenario may be dynamic, in which case we have a blurtree. We can incorporate multiple human evaluations at each node to track node performance, e.g. [Hanson 1999]’s Idea Futures as node-based judgements against stated or imposed node goals. In essence, we are discussing the distributed evolving Neartree idea from the “Data into Neartrees” chapter writ large - one can picture an abstract Neartree Network aggregated from personal Neartrees as forming a noospheric ideaflow matrix.

TOOL: The Open Opinion Layer

A potential application of Neartree Networks is in algorithms and tools for predictive collaborative information evaluation. One instantiated relevant example which could be

adapted is “Idea Futures”, a new securities institution first proposed by Robin Hanson. It can be used to securitize certain types of ideas, and hence its prices become a predictive tool that can be used within a broader context. This could include measurement of the predictive capabilities of agents using the exchange, as any agent with statistically significant greater success level will have some proven predictive advantage in a transparent testing system.

In the Idea Futures exchange, players buy shares in one of k alternatives to a question. Examples of questions being traded are the winner of future political contests, the type of technology that will be most successful in some product area, and the resolution of open scientific questions by a given future date.

After the time specified in the question has elapsed, a specified judging agency decides which alternative the outcome falls into. Players then split the net pot (gross pot - market commission - judge commission = net pot), proportional to units of the alternative purchased. As in most futures exchanges, the price of units becomes more expensive as the alternative becomes more popular.

We see this as but one component technology of a broader internetworked mesh of opinion layers that will be formed over the 21st century. This opinion metalayer has been named “TOOL: The Open Opinion Layer” in [Masum 2002]. As we suggest in that article:

“Shared opinions drive society: what we read, how we vote, and where we shop are all heavily influenced by the choices of others. However, the cost in time and money to systematically share opinions remains high, while the actual performance history of opinion generators is often not tracked.

[We explore] the development of a distributed open opinion layer, which is given the generic name of TOOL. Similar to the evolution of network protocols as an underlying layer for many computational tasks, we suggest that TOOL has the potential to become a common substrate upon which many scientific, commercial, and social activities will be based.

Valuation decisions are ubiquitous in human interaction and thought itself. Incorporating information valuation into a computational layer will be as significant

a step forward as our current communication and information retrieval layers.

....From a broader societal point of view, TOOL could provide a social substrate that would quantify and encourage the production of useful ideas, honest behavior, and healthy communities.

Many of the complex social problems we face involve both informational and motivational constraints; the generation of models which synthesize facts across many domains or do not yield economic benefit for long periods of time seems to be systematically underinvested in, and one causative factor is the difficulty of capitalizing on being part of such a model-generation effort. At an individual level, the venture-capital industry has created an efficient mechanism to reward entrepreneurs for the creation of good product and service ideas, but no such efficient reward mechanism exists for the creation of more intangible or dispersed ideas, especially when they possess the characteristics of public goods. Inefficient proxies do exist, like promoting your idea as the "Next Big Thing" and going out on the lecture circuit, but such proxies skew incentives and aren't necessarily the most efficient way of idea dispersal.

At a collective level, consider that one way of ranking the intelligence of organisms is with respect to their effectiveness in modeling the external world. Humans are the undisputed leaders in Earth's biosphere - but in order to proceed beyond the limited power of a single mind, world-models need to be shared and collaboratively built, through the creation of a vast public space of dynamic distilled judgements.

....I believe that TOOL could eventually become a "credit allocation layer" for many human endeavors. One of the advantages of markets is their resource allocation and price discovery functions, implicitly performing many complex decisions in a largely distributed manner. The ability to directly generate and analyze robust opinions can be expected to amplify the computational power of our societal infrastructure. In many adaptive computational systems, a key function is the fitness or reward function, which provides feedback that directs the system toward better behavior. The ability to objectively reflect on our individual and aggregate behavior - and come to a consensus on what works and what doesn't - will profoundly increase our collective capability to solve the increasingly complex problems that we will be faced with in the future."

11.4 Conclusions

In this thesis, I have worked toward several **goals**:

- To develop models and data structures explaining computational hierarchies, and survey previous research in the field.
- To use these models and data structures to develop algorithms for hierarchical simulation and analysis of near-hierarchical systems.
- To define and explore “hierarchical SAT” - a hierarchical version of a fundamental problem in theoretical computer science.
- To suggest ways of moving forward in related issues including valuation, risk analysis, and generating Neartrees from data.
- To develop a coherent point of view that ties these and other ideas together, with the aim of providing useful tools and laying an intellectual foundation for future work in the area.

Alert readers will have noticed many specific questions and general avenues of exploration suggested by the experiments and theory in this thesis. Several of the more notable **open questions** include:

- What changes need to be made to match the theory developed with large-scale applications? How can Neartrees be implemented in a large-scale real-world domain?
- Can the theory be extended to the case where nodes have some individual goals, over and above those assigned by their parent nodes? What changes take place when intra-level node communication is added?
- How about tracking node dependencies and correlations directly, using Estimation of Distribution Algorithms as in e.g. [Pelikan 2002]?
- More mathematical insight always helps. Fractals are perhaps the paradigmatic example of recursive structures - can fractal mathematics be applied to hierarchies?
- Where and why do hierarchies break down? Mismatch with the real world, multiple decompositions depending on user’s purpose (i.e. too much context-dependence), nodes that are too fuzzy or dynamic, epistasis... What we can do about it?
- Our framework as defined naturally lends itself to a distributed implementation, with attendant advantages including greater computational power and a variety of approaches attacking the same problem. What additional protocols are then needed to ensure

properties like overall model consistency and goal synchronization?

- While research into hierarchical knowledge representation and reasoning such as [Lambrix 2000] deals with more expressive languages than the boolean functions we have explored, our domain has the advantage of being both solvable and amenable to some degree of analysis. Can the two frameworks be integrated, to apply our algorithms to representations which more directly specify real-world domains?

There is a great deal of work to be done in this field, both in better understanding behavior of near-hierarchical systems, and applying the insights gained to engineering and policy-making issues. We have only begun the process - and yet it is clear from our research that even the relatively simple nhSAT model we have used is far from being fully understood.

In order to understand and solve the thorny problems we face in analyzing and decomposing large complex systems, research into the nature of near-hierarchical systems is a necessary prerequisite. We hope that this thesis represents a step forward.

“Any intelligent fool can make things bigger, more complex, and more violent. It takes a touch of genius -- and a lot of courage -- to move in the opposite direction.”
-E. F. Schumacher

Glossary

“When I use a word, it means just what I choose it to mean - neither more nor less.” -

Lewis Carroll

API	Application Programming Interface - the “control panel” of a code module, that hides arbitrary complexity underneath.
blurtree	metaphor for a large dynamic hierarchy - upper levels stable, lower levels changing more quickly, and lowest levels being a blur of movement
CDF	Cumulative Distribution Function.
clauseRatio	Same as density, below.
CNF	Conjunctive Normal Form - a Boolean formula composed of the AND of a set of clauses, each of which is the OR of a set of (possibly negated) variables. Any Boolean formula can be put into CNF form.
CSP	Constraint Satisfaction Problem
DAG	Directed Acyclic Graph
density	The density of a boolean formula is the ratio of clauses to number of variables. Separating formulae by density makes apparent phase transitions in difficulty.
DISSAT	A “pessimizing” variant of SAT introduced in this thesis, where one wishes to find a maximally satisfying assignment of truth values to variables, subject to maximizing the estimated worst-case satisfaction amount given an observed process changing the formula or realized truth values.
EC	Evolutionary Computation.
F	False.
h	A prefix meaning “hierarchical” - hX is a hierarchical version of X.
hSAT	Hierarchical SAT - a hierarchical extension to the well-known SAT problem, introduced in this thesis.
$_jH$	Level number j of a Neartree (or hierarchy).
$_jH_k$	Node number k in level number j of a Neartree (or hierarchy).

${}^{-}_jH_k$	Parent set of ${}_jH_k$.
${}^{+}_jH_k$	Child set of ${}_jH_k$.
$@({}_jH_k)$	State of ${}_jH_k$.
$?@({}_jH_k)$	Goal state of ${}_jH_k$.
$\$_jH_k$	Value of ${}_jH_k$.
L	Maximum level number in a Neartree (where root has level 0).
l_{avg}	Level growth exponent of a Neartree.
level	Set of nodes in a Neartree that are at a the same distance from the root node; usually each successive level represents a more detailed view of a problem.
MAXSAT	Maximal SAT - an optimizing variant of SAT, where the goal is to find an assignment of truth values to variables causing the maximal possible number of clauses to evaluate to true.
Neartree	Read the thesis! :-)
nh	Shorthand for “near-hierarchical” - nhX is a near-hierarchical version of X.
PDF	Probability Distribution Function
POV	Point Of View.
R	Clause Ratio - an abbreviation with the same meaning as density.
RP	A computational complexity class, in which an algorithm is guaranteed to report either a correct answer or “don’t know” in polynomial time.
SAT	Satisfiability Problem - a canonical NP-complete decision problem. Given a Boolean formula (in e.g. 3-CNF form), the problem is to find an assignment of truth values to the variables which makes the formula evaluate to true.
SocialNet	Human social interactions viewed as a network - analogous to “Internet”.
SubNeartree	The new Neartree formed by a given node in a Neartree and its children.
T	True.
Technical	Font used to visually distinguish math and algorithms.
TSP	Traveling Salesman Problem.
$^{\wedge\wedge}$	Exponentiation, e.g. $4^{\wedge\wedge}(2^3) = 4^8 = 65536$. (Used in cases of double exponentiation only, for typographical clarity.)

References

“Knowledge is power.” - Thomas Hobbes

- [Abaza & Baranzini 2003] Hussein Abaza and Andrea Baranzini (Editors). *Implementing Sustainable Development: Integrated Assessment and Participatory Decision-Making Processes*. Edward Elgar, 2003.
- [Achlioptas et al 2000] D Achlioptas, C Gomes, H Kautz, and B Selman. Generating satisfiable problem instances. In *Proceedings of AAAI 2000*, 2000.
- [Agarwal & Sharir 1998] Pankaj Agarwal and Micha Sharir. Efficient Algorithms for Geometric Optimization. In *ACM Computing Surveys*, Vol.30 (1998), pp 412-458.
- [Ahl & Allen 1996] Valerie Ahl and TFH Allen. *Hierarchy Theory*. Columbia University Press, 1996.
- [Albert & Barabási 2002] Réka Albert and Albert-László Barabási. Statistical mechanics of complex networks. In *Reviews of Modern Physics*, Vol.74 (2002), pp 47-97.
- [Aldous & Propp 1998] David Aldous and James Propp. *Microsurveys in Discrete Probability*. AMS, 1998.
- [Alon et al 2000] Noga Alon, Seannie Dar, Michal Parnas, and Dana Ron. Testing of Clustering. Extended Abstract appears in *FOCS 2000*.
- [Anderson et al 1988] Philip Anderson et al (Editors). *The Economy as an Evolving Complex System*. Addison-Wesley, 1988.
- [Arthur et al 1997] Brian Arthur et al (Editors). *The Economy as an Evolving Complex System II*. Addison-Wesley, 1997.
- [Aslam et al 2000] Javed Aslam, Fred Reiss, and Daniela Rus. Scalable Information Organization. In *RIAO 2000*.
- [Atallah et al 1999] Mikhail Atallah (Editor). *Algorithms and Theory of Computation Handbook*. CRC Press, 1999.
- [Ausiello et al 1999] G Ausiello, P Crescenzi, G Gambosi, V Kann, A Marchetti-Spaccamela, and M Protasi. *Complexity and Approximation: Combinatorial Optimization Problems and Their Approximability Properties*. Springer, 1999.
- [Axelrod 1997] Robert Axelrod. *The Complexity of Cooperation: Agent-Based Models of*

- Competition and Collaboration*. Princeton University Press, 1997.
- [Baeck et al 2000] Thomas Baeck, David B. Fogel, and Zbigniew Michalewicz (Editors). *Evolutionary Computation 1 & 2*. Institute of Physics Publishers, 2000.
- [Baldwin & Clark 1999] Carliss Y Baldwin and Kim B Clark. *Design Rules: The Power of Modularity (Volume 1)*. MIT Press, 1999.
- [Ball et al 1995] MO Ball, TL Magnanti, CL Monma, GL Nemhauser (Editors). *Handbooks in Operations Research and Management Science, 7: Network Models*. North-Holland, 1995.
- [Bang-Jensen 2001] Joergen Bang-Jensen. *Digraphs: Theory, Algorithms and Applications*. Springer, 2001.
- [Battista et al 1999] Giuseppe di Battista, Peter Eades, Roberto Tamassia and Ioannis G Tollis. *Graph Drawing: Algorithms for the Visualization of Graphs*. Prentice-Hall, 1999.
- [Baum 1998] Eric B Baum. Manifesto for an Evolutionary Economics of Intelligence. In *Neural Networks and Machine Learning*, CM Bishop (Editor), Springer-Verlag, 1998; pp 285-344.
- [Baum & Durdanovic 2000] Eric B Baum and Igor Durdanovic. Evolution of Cooperative Problem Solving in an Artificial Economy. NEC Research Institute Technical Report TR-2000-027R; NECI, 2000.
- [Bentham 1789] Jeremy Bentham. *An Introduction to the Principles of Morals and Legislation*. 1789.
- [Bernstein 1998] Peter Bernstein. *Against the Gods: The Remarkable Story of Risk*. Wiley, 1998.
- [Boer 1999] F Peter Boer. *The Valuation of Technology: Business and Financial Issues in R&D*. Wiley, 1999.
- [Bomze et al 1999] IM Bomze, M Budinich, PM Pardalos, and M. Pelillo. The maximum clique problem. In *Handbook of Combinatorial Optimization (Supplement Volume A)*, DZ Du and PM Pardalos (Editors.), Kluwer Academic Publishers, 1999; pp. 1-74.
- [Charikar et al 1997] Moses Charikar, Chandra Chekuri, Tomas Feder, and Rajeev Motwani. Incremental Clustering and Dynamic Information Retrieval. In *STOC-97*, pp 626-635.

- [Clarke et al 1999] Edmund M Clarke Jr., Orna Grumberg, and Doron A Peled. *Model Checking*. MIT Press, 1999.
- [Clearwater 1996] Scott Clearwater (Editor). *Market-Based Control: A Paradigm for Distributed Resource Allocation*. World Scientific, 1996.
- [Coarfa et al 2001] Cristian Coarfa, Demetrios D. Demopoulos, Alfonso San Miguel Aguirre, Devika Subramanian, Moshe Y. Vardi. Random 3-SAT: The Plot Thickens. Full version of paper in *Proceedings of the International Conference on Constraint Programming, 2001*.
- [Coen 1999] Enrico Coen. *The Art of Genes: How Organisms Make Themselves*. Oxford University Press, 1999.
- [Cormen et al 2001] Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, and Clifford Stein. *Introduction to Algorithms (2nd ed)*. MIT Press, 2001.
- [Costanza 1997] Robert Costanza et al., The Value of the World's Ecosystem Services and Natural Capital. In *Nature*, Vol. 387 (1997), p. 259.
- [Creignou et al 2001] Nadia Creignou, Sanjeev Khanna, and Madhu Sudan. Complexity Classification of Boolean Constraint Satisfaction Problems. In *SIGACT News*, Vol.32 No.4 (December 2001); pp 24-33.
- [Davey & Priestley 2002] BA Davey and HA Priestley. *Introduction to Lattices and Order (2nd ed)*. Cambridge University Press, 2002.
- [Davies 1996] Glyn Davies. *A History of Money: From Ancient Times to the Present Day*. University of Wales Press, 1996.
- [Dawes 1988] Robyn M Dawes. *Rational Choice in an Uncertain World*. Harcourt Brace Jovanovich, 1988.
- [Deboeck & Kohonen 1998] Guido J Deboeck and Tuevo K Kohonen (Editors). *Visual Explorations in Finance (with Self-Organizing Maps)*. Springer-Verlag, 1998.
- [Dembo & Freeman 1998] Ron S Dembo and Andrew Freeman. *Seeing Tomorrow: Weighing Financial Risk in Everyday Life*. John Wiley & Sons, 1998.
- [Dietterich 2000] Thomas G Dietterich. Hierarchical Reinforcement Learning with the MAXQ Value Function Decomposition. In *Journal of Artificial Intelligence Research*, Vol.13 (2000), pp 227-303.
- [Dijkstra 1972] EW Dijkstra. The Humble Programmer. In *Communications of the ACM*,

Vol.15 No.10 (1972), pp 859–866.

- [Du et al 1997] Dingzhu Du, Jun Gu, and Panos Pardalos (Editors). *Satisfiability Problem : Theory and Applications*. American Mathematical Society, 1997.
- [Eades & Feng 1997] Peter Eades and Qingwen Feng, Multilevel Visualization of Clustered Graphs. In *Proceedings of the 4th International Symposium on Graph Drawing (GD '96)*, Springer-Verlag, 1997; pp 101-112.
- [Ehrgott & Gandibleux 2002] Matthias Ehrgott and Xavier Gandibleux (Editors). *Multiple Criteria Optimization: State of the Art Annotated Bibliographic Surveys*. Kluwer Academic Publishers, 2002.
- [Epstein & Axtell 1996] Joshua M Epstein and Robert L Axtell. *Growing Artificial Societies: Social Science from the Bottom Up*. MIT Press, 1996.
- [Feigenbaum et al 1995] Edward A Feigenbaum, Julian Feldman, and Paul Armer (Editors). *Computers and Thought*. AAAI Press, 1995.
- [Fletcher & Zwick 2000] JA Fletcher and M Zwick. Simpson's Paradox Can Emerge from the N-Player Prisoner's Dilemma: Implications for the Evolution of Altruistic Behavior. In *Proceedings of The World Congress of the Systems Sciences and ISSS 2000*.
- [Fogel 2002] David B Fogel. *Blondie24: Playing at the Edge of AI*. Morgan Kaufmann, 2002.
- [Foner 1999] Leonard Newton Foner. *Political Artifacts and Personal Privacy: The Yenta Multi-Agent Distributed Matchmaking System*. PhD Thesis, MIT Media Lab; 1999.
- [Foster & Kesselman 1998] Ian Foster and Carl Kesselman (Editors). *The Grid: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann, 1998.
- [Friedman & Rust 1993] Daniel Freedman and John Rust (Editors). *The Double Auction Market: Institutions, Theories, and Evidence*. Perseus, 1993.
- [Fukunaga 2002] Alex Fukunaga. Automated Discovery of Composite SAT Variable-Selection Heuristics. In *Proceedings of AAAI-2002*, Edmonton, Canada, 2002; pp.641-648.
- [Fukuyama 1996] Francis Fukuyama. *Trust: The Social Virtues and the Creation of Prosperity*. Free Press, 1996.
- [Gent & Walsh 1999] Ian Gent and Toby Walsh. The Search for Satisfaction. Article online at dream.dai.ed.ac.uk/group/tw/sat/sat-survey2.ps; accessed September 08, 2002.

- [Gintis 2000] Herbert Gintis. *Game Theory Evolving*. Princeton University Press, 2000.
- [Goldstein 2002] Harvey Goldstein. *Multilevel Statistical Models (3rd ed)*. Edward Arnold, 2002.
- [Gomes et al 2000] CP Gomes, B Selman, N Crato, and HA Kautz, Heavy-Tailed Phenomena in Satisfiability and Constraint Satisfaction Problems. In *Journal of Automated Reasoning*, Vol.24 Nos.1-2 (February 2000), pp. 67-100.
- [GordonA 1999] AD Gordon. *Classification (2nd ed)*. CRC Press, 1999.
- [GordonR 1999] Richard Gordon. *The Hierarchical Genome and Differentiation Waves: Novel Unification of Development, Genetics and Evolution*. World Scientific, 1999.
- [Hall 2000] Brian K Hall (Editor). *Homology: The Hierarchical Basis of Comparative Biology*. Academic Press, 2000.
- [Hall et al 1998] J Storrs Hall, Louis Steinberg and Brian D Davison. Combining agoric and genetic methods in stochastic design. In *Nanotechnology*, Vol.9 No.3 (September 1998), pp 274-284.
- [Hanson 1999] Robin Hanson. Decision Markets. In *IEEE Intelligent Systems*, Vol.14 No.3 (May/June 1999), pp 16-19.
- [Halperin 2000] E Halperin. Improved approximation algorithms for the vertex cover problem in graphs and hypergraphs. In *Proceedings of the 11th SODA (2000)*, pp 329-337.
- [Hawken et al 1999] Paul Hawken, Amory Lovins, and L Hunter Lovins. *Natural Capitalism*. Little Brown & Company, 1999.
- [Hayek 1960] F A Hayek. *The Constitution of Liberty*. University of Chicago Press, 1960.
- [Heineman & Councill 2001] George T Heineman and William T Councill. *Component-Based Software Engineering: Putting the Pieces Together*. Addison-Wesley, 2001.
- [Hemaspaandra & Selman 1997] Lane Hemaspaandra and Aan Selman. *Complexity Theory Retrospective II*. Springer, 1997.
- [Hochbaum 1997] Dorit S Hochbaum (Editor). *Approximation Algorithms for NP-Hard Problems*. PWS, 1997.
- [Hofstadter 1985] Douglas Hofstadter. *Metamagical Themas: Questing for the Essence of*

- Mind and Pattern*. Basic Books, 1985.
- [Holland 1999] John H Holland. *Emergence: From Chaos to Order*. Perseus Publishing, 1999.
- [Homer-Dixon 1999] Thomas Homer-Dixon. *Environment, Scarcity, and Violence*. Princeton University Press, 1999.
- [Homer-Dixon 2000] Thomas Homer-Dixon. *The Ingenuity Gap*. Knopf, 2000.
- [Hornby 2003] Gregory S Hornby. *Generative Representations for Evolutionary Design Automation*. Brandeis University Dept. of Computer Science, Ph.D. Thesis; 2003.
- [Huberman 1988] Bernardo Huberman (Editor). *The Ecology of Computation*. Elsevier Science, 1988.
- [Huberman et al 1999] Bernardo A Huberman, Matt Franklin, and Tad Hogg. Enhancing Privacy and Trust in Electronic Communities. In *EC-99: Proceedings of the ACM Conference on Electronic Commerce*, November 1999, pp 78-86.
- [Huberman, Lukose & Hogg 1997] BA Huberman, RM Lukose and T Hogg. An Economics Approach to Hard Computational Problems. In *Science*, Vol.275 (Jan 3 1997), pp 51-54.
- [Hull 2002] John C Hull. *Options, Futures, and Other Derivatives (5th ed)*. Prentice Hall, 2002.
- [Imamura et al 2002] K Imamura, RB Heckendorn, T Soule, and JA Foster. N-version genetic programming via fault masking. In *Genetic Programming: Proc. 5th European Conference, EuroGP 2002*; Lecture Notes in Computer Science 2278, Springer-Verlag, 2002 (pp 172-181).
- [Jain et al 1999] AK Jain, MN Murty, and PJ Flynn. Data clustering: a review. In *ACM Computing Surveys*, Vol.31 Issue 3 (September 1999), pp 264 - 323.
- [Joyce 1999] James M Joyce. *The Foundations of Causal Decision Theory*. Cambridge University Press, 1999.
- [Kanehisa 2000] Minoru Kanehisa. *Post-Genome Informatics*. Oxford University Press, 2000.
- [Kannan et al 2000] Ravi Kannan, Santosh Vempala and Adrian Vetta. On clusterings: good, bad and spectral. In *Proceedings of the 41st Foundations of Computer Science Conference*.

- [Keller 1999] Laurent Keller (Editor). *Levels of Selection in Evolution*. Princeton University Press, 1999.
- [Kirousis et al 1996] LM Kirousis, E Kranakis, and D Krizanc. Approximating the Unsatisfiability Threshold of Random Formulas. In *Proceedings of the 4th Annual European Symposium on Algorithms (ESA '96)*, pp 27-38, 1996.
- [Klein 1997] Daniel B Klein (Editor). *Reputation: Studies in the Voluntary Elicitation of Good Conduct*. University of Michigan Press, 1997.
- [Klein & Kozlowski 2000] Katherine J Klein and Steve WJ Kozlowski (Editors). *Multilevel Theory, Research, and Methods in Organizations: Foundations, Extensions, and New Directions*. John Wiley & Sons; 2000.
- [Knuth 1997] Donald E Knuth. *The Art of Computer Programming, Volume 1: Fundamental Algorithms (3rd ed)*. Addison-Wesley, 1997.
- [Koza 1994] John R Koza. *Genetic Programming II: Automatic Discovery of Reusable Programs*. MIT Press, 1994.
- [Koza et al 1999] John R Koza, Forrest H Bennett III, and David Andre. *Genetic Programming III: Automatic Programming and Automatic Circuit Synthesis*. Morgan Kaufmann, 1999.
- [Kuttner 1996] Robert Kuttner. *Everything for Sale: The Virtues and Limits of Markets*. Knopf, 1996.
- [Lambrix 2000] Patrick Lambrix. *Part-whole reasoning in an object-centered framework*. Springer, 2000.
- [Li & Vitanyi 1997] Ming Li and Paul Vitanyi. *An Introduction to Kolmogorov Complexity and Its Applications (2nd ed)*. Springer-Verlag, 1997.
- [Littman et al 2001] M.L. Littman, S.M. Majercik, and T. Pitassi. Stochastic Boolean Satisfiability. In *Journal of Automated Reasoning*, Vol.27 No.3 (October 2001), pp 251-296.
- [Luenberger 1997] David G Luenberger. *Investment Science*. Oxford University Press, 1997.
- [Mali & Kambhampati 1998] AD Mali and S Kambhampati. Encoding HTN Planning in Propositional Logic. In *Proceedings of the Fourth International Conference on Artificial Intelligence Planning Systems*, pp 190-198; AAAI Press, 1998.

- [Marathe et al 1998] M Marathe, H Hunt, R Stearns, and V Radhakrishnan. Approximation Algorithms for PSPACE-Hard Hierarchically and Periodically Specified Problems. In *SIAM Journal on Computing*, Vol.27 (1998), pp. 1237-1261.
- [Masum and Oppacher 2001] Hassan Masum and Franz Oppacher. Regulatory Networks and Genomic Algorithms. In *SCI 2001 / ISAS 2001*; July 2001, Orlando, USA.
- [Masum 2002] Hassan Masum. TOOL: The Open Opinion Layer. In *First Monday*, Vol.7 No.7 (July 2002); available online at firstmonday.org/issues/issue7_7/masum/.
- [Matthen & Ware 1994] Mohan Matthen and RX Ware. *Biology and Society: Reflections on Methodology*. University of Calgary Press, 1994.
- [McShea 2001] Daniel W McShea. The hierarchical structure of organisms: a scale and documentation of a trend in the maximum. In *Paleobiology*, Vol.27 No.2 (2001); pp 405–423.
- [Mitchell 1997] Tom M Mitchell. *Machine Learning*. McGraw-Hill, 1997.
- [Nehaniv 1999] Chrystopher L Nehaniv (Editor). *Mathematical and Computational Biology: Computational Morphogenesis, Hierarchical Complexity, and Digital Evolution*. American Mathematical Society, 1999.
- [Nowak & Sigmund 1998] MA Nowak and K Sigmund. The Dynamics of Indirect Reciprocity. In *Journal of Theoretical Biology*, Vol.194 No.4 (Oct 21 1998), pp 561-574.
- [O'Connor & Spash 1999] Martin O'Connor and Clive L Spash (Editors). *Valuation and the Environment : Theory, Method, and Practice*. Edward Elgar, 1999.
- [Patterson & Kautz 2001] Donald J Patterson and Henry Kautz. Auto-Walksat: A Self-Tuning Implementation of Walksat. In *Proceedings of SAT2001: Workshop on Theory and Application of Satisfiability Testing*.
- [Pearl 2000] Judea Pearl. *Causality: Models, Reasoning, and Inference*. Cambridge University Press, 2000.
- [Pelikan 2002] Martin Pelikan. *Bayesian Optimization Algorithm: From Single Level to Hierarchy*. Ph.D. thesis, University of Illinois at Urbana-Champaign; 2002.
- [Pennock & Wellman 2002] David M. Pennock and Michael P. Wellman. A market framework for pooling opinions. Submitted for publication; available online at <http://www.neci.nec.com/homepages/dpennock/papers/mfop-submit.ps>

- [Pooler 2000] James A Pooler. *Hierarchical Organization in Society: A Canadian Perspective*. Ashgate Publishing, 2000.
- [Prather 1995] Ronald E Prather. "Design and analysis of hierarchical software metrics." In *ACM Computing Surveys*; Volume 27 Issue 4 (December 1995), pp 497-518.
- [Procopiuc 2001] Cecilia M Procopiuc. *Clustering problems and their applications*. Survey paper, online at www.cs.duke.edu/~magda/; accessed September 08, 2002.
- [Prugh 1999] Thomas Prugh. *Natural Capital and Human Economic Survival (2nd ed)*. Lewis Publishers, 1999.
- [Prusinkiewicz 1996] Przemyslaw Prusinkiewicz. *The Algorithmic Beauty of Plants*. Springer-Verlag, 1996.
- [Rardin 1999] Ronald L Rardin. *Optimization in Operations Research*. Prentice-Hall, 1997.
- [Raudenbush & Bryk 2002] Stephen W Raudenbush and Anthony S Bryk. *Hierarchical Linear Models: Applications and Data Analysis Methods (2nd ed)*. Sage Publications, 2002.
- [Raymond 1999] Eric S Raymond. *The Cathedral and the Bazaar*. O'Reilly, 1999.
- [Reilly & Brown 2002] Frank K Reilly and Keith C Brown. *Investment Analysis and Portfolio Management (7th ed)*. South-Western College Publishing, 2002.
- [Resnick & Varian 1997] Paul Resnick and Hal R Varian. Recommender Systems. In *Communications of the ACM*, Vol.40 No.3 (March 1997), pp 56-58.
- [Richa et al 2001] AW Richa, K Obraczka, and A Sen. Application-Oriented Self-organizing Hierarchical Clustering in Dynamic Networks. In *Proceedings of 1st ACM Workshop on Principles of Mobile Computing (POMC)*, August 2001; pp 57-65.
- [de Rosnay 1995] Joël de Rosnay. *L'homme Symbiotique*. Seuil, 1995.
- [Rothschild 1990] Michael Rothschild. *Bionomics: Economy as Ecosystem*. Henry Holt, 1990.
- [Salthe 1993] Stanley N Salthe. *Development and Evolution: Complexity and Change in Biology*. MIT Press, 1993.
- [Schafer et al 1999] J Ben Schafer, Joseph Konstan, and John Riedl. Recommender Systems in E-Commerce. In *EC-99: Proceedings of the ACM Conference on Electronic*

Commerce, November 1999, pp 158-166.

- [Schmutter 2002] Peter Schmutter. Object-Oriented Ontogenetic Programming: Breeding Computer Programs that Work Like Multicellular Creatures. Diploma Thesis, University of Dortmund, 2002; available online at <http://www.ooop.org/publications/thesis/>.
- [Sen, Biswas and Debnath 2000] Sandip Sen, Anish Biswas, and Sandip Debnath. Believing others: Pros and Cons. In *Proceedings of the Fourth International Conference on Multiagent Systems*.
- [Shapiro & Varian 1998] Carl Shapiro and Hal Varian. *Information Rules*. Harvard Business School Press, 1998.
- [Simon 1996] Herbert Simon. *The Sciences of the Artificial (3rd edition)*. MIT Press, 1996.
- [Simon 1997] Herbert Simon. *Administrative Behavior: A Study of Decision-Making Processes in Administrative Organizations (4th ed)*. Free Press, 1997.
- [Sipser 1997] Michael Sipser. *Introduction to the Theory of Computation*. PWS Publishing Company, 1997.
- [Slaney & Walsh 2001] John Slaney and Toby Walsh. Backbones in optimization and approximation. In *Proceedings of IJCAI 2001*, 2001.
- [Slaney & Walsh 2002] John Slaney and Toby Walsh. Phase Transition Behavior: From Decision to Optimization. In *SAT-2002*.
- [Smieja 1996] Frank Smieja. The Pandemonium System of Reflective Agents. In *IEEE Transactions on Neural Networks*, Vol.7 No.1 (1996), pp 97-106.
- [Sowa 1998] John F Sowa. *Knowledge Representation; Logical, Philosophical, and Computational Foundations*. PWS, 1998.
- [Stiegler 1999] Marc Stiegler. *Earthweb*. Baen Books, 1999.
- [Stock 1993] Gregory Stock. *Metaman: The Merging of Humans and Machines into a Global Superorganism*. Simon & Schuster, 1993.
- [Sutton & Barto 1998] Richard S Sutton and Andrew G Barto. *Reinforcement Learning: An Introduction*. MIT Press, 1998.
- [Svenonius 2000] Elaine Svenonius. *The Intellectual Foundation of Information Organization*. MIT Press, 2000.

- [Theodoridis & Koutroumbas 1999] Sergios Theodoridis and Konstantinos Koutroumbas. *Pattern Recognition*. Academic Press, 1999.
- [Thompson 2000] James R Thompson. *Simulation: A Modeler's Approach*. Wiley, 2000.
- [Van Veldhuizen & Lamont 2000] David A Van Veldhuizen and Gary B Lamont. Multiobjective Evolutionary Algorithms: Analyzing the State-of-the-Art. In *Evolutionary Computation*, Vol.8 No.2 (Summer 2000), pp 125-147.
- [Vose 2000] David Vose. *Risk Analysis: A Quantitative Guide (2nd ed)*. John Wiley & Sons, 2000.
- [Walsh 2002] Toby Walsh. Stochastic Constraint Programming. In *Proceedings of ECAI-2002*, 2002.
- [Watts 1999] Duncan J Watts. *Small Worlds: The Dynamics of Networks Between Order and Randomness*. Princeton University Press, 1999.
- [WebGlobalBrain] The Social Superorganism and its Global Brain. Online at <http://pespmc1.vub.ac.be/SUPORGLI.html>; accessed November 4, 2002.
- [WebGraphicalModels] A Brief Introduction to Graphical Models and Bayesian Networks. Online at <http://www.cs.berkeley.edu/~murphyk/Bayes/bayes.html>; accessed November 4, 2002.
- [WebHierarchy] A Summary of the Principles of Hierarchy Theory. Online at <http://www.iss.org/hierarchy.htm>; accessed August 30, 2002.
- [WebHLM] Hierarchical Linear and Nonlinear Modeling. Online at <http://www.ssicentral.com/hlm/hlm.htm>; accessed September 08, 2002.
- [WebNPApproximability] A Compendium of NP Optimization Problems. Online at <http://www.nada.kth.se/~viggo/problemlist/compendium.html>; accessed September 18, 2002.
- [WebPostAutisticEconomics] Post-Autistic Economics. Online at <http://www.paecon.net/>; accessed September 18, 2002.
- [Wolpert & Tumer 1999] David Wolpert and Kagan Tumer. *An Introduction to Collective Intelligence*. Tech Report NASA-ARC-IC-99-63.
- [Yao & Darwen 1999] X Yao and PJ Darwen. How Important Is Your Reputation in a Multi-Agent Environment? In *Proceedings of the 1999 IEEE Conference on Systems, Man, and Cybernetics*, October 1999, pp.II-575 - II-580.

- [Young 1998] H Peyton Young. *Individual Strategy and Social Structure: An Evolutionary Theory of Institutions*. Princeton University Press, 1998.
- [Ygge 1998] Fredrik Ygge. *Market-Oriented Programming and its Application to Power Load Management*. PhD Thesis, Department of Computer Science, Lund University, Sweden; ISBN 91-628-3055-4, 1998.