1

PM-1 3 1/2" x 4" PHOTOGRAPHIC MICROCOPY TARGET
NBS 1010a ANSI/ISO #2 EQUIVALENT

PRECISIONSM RESOLUTION TARGETS

National Library
of Canada

Acquisitions and
Bibliographic Services Branch

395 Wellington Street
Ottawa, Ontario
K1A 0N4

Bibliothèque nationale
du Canada

Direction des acquisitions et
des services bibliographiques

395, rue Wellington
Ottawa (Ontario)
K1A 0N4

Your file / Votre référence

Our file / Notre référence

NOTICE

The quality of this microform is heavily dependent upon the quality of the original thesis submitted for microfilming. Every effort has been made to ensure the highest quality of reproduction possible.

If pages are missing, contact the university which granted the degree.

Some pages may have indistinct print especially if the original pages were typed with a poor typewriter ribbon or if the university sent us an inferior photocopy.

Reproduction in full or in part of this microform is governed by the Canadian Copyright Act, R.S.C. 1970, c. C-30, and subsequent amendments.

AVIS

La qualité de cette microforme dépend grandement de la qualité de la thèse soumise au microfilmage. Nous avons tout fait pour assurer une qualité supérieure de reproduction.

S'il manque des pages, veuillez communiquer avec l'université qui a conféré le grade.

La qualité d'impression de certaines pages peut laisser à désirer, surtout si les pages originales ont été dactylographiées à l'aide d'un ruban usé ou si l'université nous a fait parvenir une photocopie de qualité inférieure.

La reproduction, même partielle, de cette microforme est soumise à la Loi canadienne sur le droit d'auteur, SRC 1970, c. C-30, et ses amendements subséquents.

AN EXPLORATION OF TURING MACHINE BASED COMPLEXITY

by

HASSAN MASUM

A thesis submitted to the Faculty of Graduate Studies and Research in
partial fulfilment of the requirements for the degree of Master of Science.

Department of Mathematics
Carleton University
Ottawa, Ontario
July 31, 1995

© 1995 by Hassan Masum

National Library
of Canada

Acquisitions and
Bibliographic Services Branch

395 Wellington Street
Ottawa, Ontario
K1A 0N4

Bibliothèque nationale
du Canada

Direction des acquisitions et
des services bibliographiques

395, rue Wellington
Ottawa (Ontario)
K1A 0N4

Your file Votre référence

Our file Notre référence

The author has granted an irrevocable non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of his/her thesis by any means and in any form or format, making this thesis available to interested persons.

L'auteur a accordé une licence irrévocable et non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de sa thèse de quelque manière et sous quelque forme que ce soit pour mettre des exemplaires de cette thèse à la disposition des personnes intéressées.

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

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

ISBN 0-612-09010-8

Canada

Name HASSAN MASUM

Dissertation Abstracts International is arranged by broad, general subject categories. Please select the one subject which most nearly describes the content of your dissertation. Enter the corresponding four-digit code in the spaces provided.

COMPUTER SCIENCE

SUBJECT TERM

0984

U-M-I

SUBJECT CODE

Subject Categories

THE HUMANITIES AND SOCIAL SCIENCES

COMMUNICATIONS AND THE ARTS

Architecture 0729
Art History 0377
Cinema 0900
Dance 0378
Fine Arts 0357
Information Science 0723
Journalism 0391
Library Science 0399
Mass Communications 0708
Music 0413
Speech Communication 0459
Theater 0465

EDUCATION

General 0515
Administration 0514
Adult and Continuing 0516
Agricultural 0517
Art 0273
Bilingual and Multicultural 0282
Business 0688
Community College 0275
Curriculum and Instruction 0727
Early Childhood 0518
Elementary 0524
Finance 0277
Guidance and Counseling 0519
Health 0680
Higher 0745
History of 0520
Home Economics 0278
Industrial 0521
Language and Literature 0279
Mathematics 0280
Music 0522
Philosophy of 0998
Physical 0523

Psychology 0525
Reading 0535
Religious 0527
Sciences 0714
Secondary 0533
Social Sciences 0534
Sociology of 0340
Special 0529
Teacher Training 0530
Technology 0710
Tests and Measurements 0288
Vocational 0747

LANGUAGE, LITERATURE AND LINGUISTICS

Language 0679
General 0289
Ancient 0290
Linguistics 0291
Modern 0401
Literature 0294
General 0295
Classical 0297
Comparative 0298
Medieval 0316
Modern 0298
African 0316
American 0591
Asian 0305
Canadian (English) 0352
Canadian (French) 0355
English 0592
Germanic 0311
Latin American 0312
Middle Eastern 0315
Romance 0313
Slavic and East European 0314

PHILOSOPHY, RELIGION AND THEOLOGY

Philosophy 0422
Religion 0318
General 0321
Biblical Studies 0319
Clergy 0320
History of 0322
Philosophy of 0469
Theology 0323

SOCIAL SCIENCES

American Studies 0324
Anthropology 0326
Archaeology 0327
Cultural 0310
Physical 0272
Business Administration 0770
General 0454
Accounting 0338
Banking 0385
Management 0501
Marketing 0503
Canadian Studies 0505
Economics 0508
General 0509
Agricultural 0510
Commerce-Business 0511
Finance 0358
History 0366
Labor 0351
Theory 0578
Folklore 0578
Geography 0578
Gerontology 0578
History 0578
General 0578

Ancient 0579
Medieval 0581
Modern 0582
Black 0328
African 0331
Asia, Australia and Oceania 0332
Canadian 0334
European 0335
Latin American 0336
Middle Eastern 0333
United States 0337
History of Science 0585
Law 0398
Political Science 0615
General 0616
International Law and 0617
Relations 0814
Public Administration 0452
Recreation 0626
Social Work 0627
Sociology 0938
General 0631
Criminology and Penology 0628
Demography 0629
Ethnic and Racial Studies 0630
Individual and Family 0700
Studies 0344
Industrial and Labor 0709
Relations 0999
Public and Social Welfare 0453
Social Structure and 0453
Development 0453
Theory and Methods 0453
Transportation 0453
Urban and Regional Planning 0453
Women's Studies 0453

THE SCIENCES AND ENGINEERING

BIOLOGICAL SCIENCES

Agriculture 0473
General 0285
Agronomy 0475
Animal Culture and 0476
Nutrition 0359
Animal Pathology 0478
Food Science and 0479
Technology 0480
Forestry and Wildlife 0817
Plant Culture 0777
Plant Pathology 0746
Plant Physiology 0306
Range Management 0287
Wood Technology 0308
Biology 0309
General 0379
Anatomy 0329
Biostatistics 0353
Botany 0369
Cell 0793
Ecology 0410
Entomology 0307
Genetics 0317
Limnology 0416
Microbiology 0433
Molecular 0821
Neuroscience 0778
Oceanography 0472
Physiology 0786
Radiation 0760
Veterinary Science 0786
Zoology 0760
Biophysics 0786
General 0760
Medical 0760

EARTH SCIENCES

Biogeochemistry 0425
Geochemistry 0996

Geodesy 0370
Geology 0372
Geophysics 0373
Hydrology 0388
Mineralogy 0411
Paleobotany 0345
Paleoecology 0426
Paleontology 0418
Paleozoology 0985
Palynology 0427
Physical Geography 0368
Physical Oceanography 0415

HEALTH AND ENVIRONMENTAL SCIENCES

Environmental Sciences 0768
Health Sciences 0566
General 0300
Audiology 0992
Chemotherapy 0567
Dentistry 0350
Education 0769
Hospital Management 0758
Human Development 0982
Immunology 0564
Medicine and Surgery 0347
Mental Health 0569
Nursing 0570
Nutrition 0380
Obstetrics and Gynecology 0354
Occupational Health and 0381
Therapy 0571
Ophthalmology 0419
Pathology 0572
Pharmacology 0382
Pharmacy 0573
Physical Therapy 0574
Public Health 0575
Radiology 0575
Recreation 0575

Speech Pathology 0460
Toxicology 0383
Home Economics 0386

PHYSICAL SCIENCES

Pure Sciences 0485
Chemistry 0749
General 0486
Agricultural 0487
Analytical 0488
Biochemistry 0738
Inorganic 0490
Nuclear 0491
Organic 0494
Pharmaceutical 0495
Physical 0754
Polymer 0405
Radiation 0605
Mathematics 0986
Physics 0606
General 0608
Acoustics 0748
Astronomy and 0607
Astrophysics 0798
Atmospheric Science 0759
Atomic 0609
Electronics and Electricity 0610
Elementary Particles and 0752
High Energy 0756
Fluid and Plasma 0611
Molecular 0463
Nuclear 0346
Optics 0984
Radiation 0984
Solid State 0984
Statistics 0984
Applied Sciences 0984
Applied Mechanics 0984
Computer Science 0984

Engineering 0537
General 0538
Aerospace 0539
Agricultural 0540
Automotive 0541
Biomedical 0542
Chemical 0543
Civil 0544
Electronics and Electrical 0348
Heat and Thermodynamics 0545
Hydraulic 0546
Industrial 0547
Marine 0794
Materials Science 0548
Mechanical 0743
Metallurgy 0551
Mining 0552
Nuclear 0549
Packaging 0765
Petroleum 0554
Sanitary and Municipal 0790
System Science 0428
Geotechnology 0796
Operations Research 0795
Plastics Technology 0994
Textile Technology 0994

PSYCHOLOGY

General 0621
Behavioral 0384
Clinical 0622
Developmental 0620
Experimental 0623
Industrial 0624
Personality 0625
Physiological 0989
Psychobiology 0349
Psychometrics 0632
Social 0451

probabilities are invariant; maybe there's a simple formula relating expected time/space scales for complexity development with randomness level. In other words, greater random noise just forces the patterns out to a larger error-correcting version that takes (much) longer to arise. [Wuensche] has some preliminary data for cellular automata ("CA").

Is there a CA analogue to the information theory concept of entropy, where very high error rates imply a much larger (space or time) scale is needed?

- NOTE that "space" and "time" are post-conditioned by the organisms/computational systems that evolve! i.e. a person the size of a galaxy doesn't care how inefficient or slow the embedding system is at a fine-grained level--all that matters is the size and speed of the constituent "computational units" relative to the person. But once computation evolves or is created at any level, perhaps it pushes the evolution of new computational units of smaller size and greater speed, until the ultimate limits of the system are reached. This suggests that the evolution of computational units may be a qualitative dividing line in a system.

Can we define "pre and post Turing" stages in evolution? (Possible crossover points between these stages are when a Turing Machine "wants" to optimize itself, or when a universal Turing Machine evolves in a CA.)

- We may consider parameters measuring the "bubbliness" of cellular automata, meaning the percentage of i) random transitions, or ii) transitions which change the state of a cell.

ABSTRACT

In this thesis, we first ask a series of unsolved motivating questions dealing with the definition of complexity and the evolution of complex systems. We survey several Turing Machine based measures of complexity, and explore their shortcomings with reference to the intuitive notion of complexity; Computational Complexity, Kolmogorov Complexity, and Logical Depth are among the measures looked at. With these measures of complexity in mind, we discuss several developing areas of general interest, including Parallel and Emergent Computation, Cellular Automata, Genetic Algorithms, and Biological Evolution. New ideas and avenues of research are proposed throughout the thesis.

ACKNOWLEDGEMENTS

I would like to extend my sincere gratitude to:

- My thesis advisor, Dr Don Dawson, for encouragement and stimulating discussions.
- The many wonderful people in the Department of Mathematics and Statistics of Carleton University, for providing a congenial environment in which to learn.
- My family, for patience, love and support.

TABLE OF CONTENTS

INTRODUCTION AND IDEA FUGUE	1
What Conditions are Necessary for a System to Evolve?	3
How Can we Measure "Complexity"?	7
What is the Importance of Hierarchy and Decomposition?	10
How Can Massively Parallel Activities be Analyzed?	13
What is the Importance of "Intelligent Algorithms"?	14
What Well-Defined Fields Deal with these Questions?	15
COMPUTABILITY	17
The Turing Machine	17
Classical Computability	23
Formal Languages and Grammars	29
Resource-Bounded Computability Theory and Loop Languages	32
The Busy Beaver Problem	39
COMPUTATIONAL COMPLEXITY	44
Approximability	49
Randomized Complexity Classes	53
Leaf Languages	58
Cryptography and Zero-Knowledge Proofs	61
Speculations on ϵ -Extensions	66

INFORMATION THEORY	69
Basics	69
Uniquely Decodable and Prefix Codes	72
Optimal Coding and Entropy	76
Universal Codes	80
KOLMOGOROV COMPLEXITY	83
Basics	83
Proofs Using Kolmogorov Complexity	90
Prefix Complexity	92
Universal Measures and Induction	95
Ω --The Ultimate Oracle	97
Resource-Bounded Kolmogorov Complexity	101
Connections between Information Theory and Kolmogorov Complexity	108
MORE TURING MACHINE BASED COMPLEXITY MEASURES	111
Logical Depth	111
Sophistication Complexity	120
Stochastic Complexity	125
Statistical Complexity	130

FURTHER IDEAS ON COMPLEXITY MEASURES	135
Abstraction From KC	135
Decomposability	140
Intelligence	145
Practical Usefulness of Complexity Measures	148
 DISTRIBUTED COMPLEXITY AND THE EVOLUTION OF	
INTELLIGENCE	150
Decomposition and Hierarchy	151
(Neg)entropy, Ecologies, Life and (Co)evolution	157
Parallel and Emergent Computation	165
Cellular Automata	172
Generalized Intelligence	178
Metahuman Institutions	189
 RECAP	195
 REFERENCES	197

INTRODUCTION AND IDEA FUGUE

What is complexity? How can complex systems be

defined

measured

predicted

optimized

controlled

built

and evolved (in the form of intelligent algorithms)?

The word "complex" is originally from the Latin "com" (together) and "plectere" (to twist); it is itself a symbol with an exceptionally large set of properties. Consider:

...ABSTRUSE, ADDLING, ADVANCED, AMBIGUOUS, BAFFLING, BEWILDERING, CHAOTIC, CIRCUITOUS, CLOAKED, CLOUDY, CLUTTERED, COMPLICATED, COMPOSITE, CONCEALED, CONFOUNDING, CONFUSING, CONTRADICTIONARY, CONVOLUTED, CRYPTIC, DAZING, DETAILED, DIFFICULT, DISORDERED, DISORGANIZED, DIVERSIFIED, ECLECTIC, ELABORATE, ENIGMATIC, ENTANGLED, FLUSTERING, FOGGY, HAZY, HIDDEN, INTRICATE, INVOLVED, JUMBLED, KNOTTY, LABYRINTHINE, MANY-SIDED, MIXED-UP, MUDDLED, MULTI-FACETED, MULTIFARIOUS, MYSTERIOUS, OBSCURE, PARADOXICAL,

PERPLEXING, PERTURBING, PUZZLING, RAMIFIED, RECONDITE, SCREENED, SECRET, SHROUDED, SNARLED, SOPHISTICATED, TANGLED, TECHNICAL, TRICKY, TURBULENT, TWISTED, UNCONTROLLABLE, VEILED, WILD...

As is suggested from the many synonyms, "complexity" encompasses within its scope an infinity of types of nonunderstanding, of which we will examine but a handful. One might more profitably define "simplicity" as the properties possessed by easy problems, and complexity as the negation of this set.

In this thesis, we'll first start things off by asking some "Big Questions" to set the stage. Following this motivational material, we look rigorously at several well-established theories of Turing Machine based complexity. The Turing Machine is the most comprehensive, deep, and accessible model of computation extant, and its associated theories allow many ideas involving "complexity" to be profitably discussed.

A sampling of more recent Turing Machine related complexity measures is then presented, followed by a similar sampling of ideas from the author. In the final section, the preceding ideas are extended in a semiformal manner to several developing areas, which the author believes are of fundamental importance to an understanding of the complexity of ourselves, our creations, and our environment.

What Conditions are Necessary for a System to Evolve?

- By "evolution of a system" we mean the increase in apparent complexity of the activity in a system. (Of course, this demands a metric for complexity.) What properties of a system could we use to predict the particular maximal level of complexity the system will probably reach--or to predict that the system will increase in complexity forever?

- A first requirement: storage of learned behavior must last longer than the time needed to make use of it to get more. Any complex patterns that arise build on previous less complex patterns; the information stored in a particular pattern should not dissipate before it has been made use of.

This suggests an upper bound on the amount of random activity desirable in a system. On the other hand, consider information theory and noisy channels; codes can be devised to deal with arbitrary amounts of noise, at the expense of transmission speed (and they can transmit within epsilon of the theoretical channel capacity). It seems probable that higher-dimensional analogues of these could be used to keep information stable over a large noisy area.

However, the likelihood of such analogues spontaneously developing seems extremely low for high randomness. Conjecture: the graph of amount of randomness versus likelihood of complex development is bell-shaped and skewed to the left, with the maximal value dependent on the dimensionality/topology of the space.

On the third hand, perhaps if we look at larger space and time scales, the

probabilities are invariant; maybe there's a simple formula relating expected time/space scales for complexity development with randomness level. In other words, greater random noise just forces the patterns out to a larger error-correcting version that takes (much) longer to arise. [Wuensche] has some preliminary data for cellular automata ("CA").

Is there a CA analogue to the information theory concept of entropy, where very high error rates imply a much larger (space or time) scale is needed?

- NOTE that "space" and "time" are post-conditioned by the organisms/computational systems that evolve! i.e. a person the size of a galaxy doesn't care how inefficient or slow the embedding system is at a fine-grained level--all that matters is the size and speed of the constituent "computational units" relative to the person. But once computation evolves or is created at any level, perhaps it pushes the evolution of new computational units of smaller size and greater speed, until the ultimate limits of the system are reached. This suggests that the evolution of computational units may be a qualitative dividing line in a system.

Can we define "pre and post Turing" stages in evolution? (Possible crossover points between these stages are when a Turing Machine "wants" to optimize itself, or when a universal Turing Machine evolves in a CA.)

- We may consider parameters measuring the "bubbliness" of cellular automata, meaning the percentage of i) random transitions, or ii) transitions which change the state of a cell.

What happens with lower or higher values of these parameters? They're interesting because the parameters can be calculated directly from the specification of the CA, and don't need to be measured; any results involving them give a way of predicting some of the behavior of a system just from seeing its rules. What other such parameters can profitably be defined?

- Do evolvable/survivable systems possess interesting characteristics? It seems that ecosystems and other survivable systems (i.e. systems which robustly maintain a high level of complex patterns) are quite atypical; since the systems work, they must have some nice characteristics--which ones? (Remember, the "surroundings" or environment of a system may (co)evolve along with the "inhabitants".)

If the survival of inhabitants helps the system survive AND if the system is subject to evolutionary pressures, then the system is pressured into being predictable and comfortable. Plant life, which forms most of the system within which animals live or die, furnishes many excellent examples; bees, for instance, find a tailor-made food source in nectar, and in return assist the plant's reproductive process by pollination. Contrast this with comets and volcanoes, which derive rather less use from presently-evolved species.

- How do we factor in purpose, direction, goal-seeking, and cooperation in organisms and societies? (Consider the Prisoner's Dilemma, Game Theory, psychology, and political science for diverse viewpoints.) Certainly evolution in sentience requires pressure (eventually self-generated and named desire) to self-complexify--how is such goal-seeking

behavior sustained on the upper level of a system when the lower level holons have their own smaller, contradictory goals? (What do cells get out of being part of humans? Or humans out of being part of a society?) When does the upper level work toward the long-term benefit of the lower levels?

Also consider the paradox between the seeming power of small self-reproducing units (or replicators) and the inefficiency of large organisms. A single microorganism could reproduce to fill the world in a few days, given the right conditions.

If replicators are so good at outcompeting macroscopic life, why don't bacteria rule the earth? There must be something inefficient about small simple lifeforms. But wait--large lifeforms are just cooperating conglomerations of small ones! Remarkably, then, it would appear that the small ones gain a competitive advantage by being linked together, from the implicit algorithmic power of the higher level organism to reduce the hostility of the environment; this advantage must be sufficient to outweigh the loss in efficiency.

How Can we Measure "Complexity"?

Information theory and Kolmogorov Complexity provide ways of measuring the minimal descriptive length of objects. Does "descriptive length" correlate precisely with "complexity"? How does time evolution under a specified set of evolution equations fit in? Conway's Life is very simple to describe (a description of a CA and a couple of rules) yet we might not be able to compress an evolved configuration into anything shorter than (Life rules + initial configuration + # of steps to get evolved configuration). This suggests that definitions of space and time bounded description length are needed.

- Some initial complexity definitions (of the complexity of an object):

- 1) Expected time to evolve under given initial conditions and dynamic rules.

Objections: how would one deal with Transhuman AIs (5.0001 GYears) versus humans (5.0000 GYears); the first should be far more complex. Or microwave ovens (which require humans as a precursor but are presumably far less complex)? As well, computation can simultaneously be taking place at many different levels of a system, so a consistent definition of "time" is difficult to give (i.e. a nanomachine or microorganism year represents far more activity than a human one). Perhaps time should be defined in terms of primitive logical operations? With the advent of self-reproducing organisms, humans, and computers, the number of implicit high-level logical operations increased tremendously at each step (as did, perhaps, the ceiling on the highest level of logical operation currently performable).

2) Ease of prediction, both space and time bounded. This measure could also be recursively bound by ease of prediction; how hard is it to guess how hard it is to predict this time sequence? This leads to a set of smaller and smaller systems, each of which can approximately simulate (some aspect) of the larger one--a common situation with most measures of complexity. How does this dynamic measure compare to the minimum length description of the sequence?

3) Ease of control, i.e. making a desired change take place. (But how do we define "change" appropriately? An unending question--but given a particular way of noticing change, we can measure it, and then "ease of control" means "computational resources necessary to cause a desired change in this measurement".)

- We would probably like micron and galaxy size computers to have the same complexity; we could thus define time for each by scaling to the time required for a single logical operation or transition rule. Or we could use the complexity measures above; objects with a similar size rule kernel to describe, predict or control should be defined as having similar complexity regardless of size or other properties. Stars are simple!

- To be satisfying, we would want intelligence to correspond to high complexity. (Since a satisfactory definition of intelligence itself is not yet available, this may be problematic!) Can we consider imagination as a tool for reducing the expected time for evolution of high complexity objects?

- For every mathematical technique or algorithm that solves some problem for a set of instances, we can define a "problem space" of the instances (defined by suitable parameters) versus the measure of success achieved in solving the problem for that instance.

What is the structure of "algorithmic problem space"; what good classifications of problems exist? P-NP was a first step; we need to go further and discuss such ideas as epsilon-approximability and mental "complexity". Some divisions: linear-nonlinear, degrees of freedom, approximability, predictability, # of dimensions, degree of irreversibility.

- Since we are interested in using mathematical techniques to discuss complexity, it might be of use to characterize complexity in metamathematics. How can promising mathematical axiom spaces be characterized? Is there some measure on such mathematical spaces which is greater for spaces that generate "good" theories? (A "good" theory is one that gives the greatest number of interesting and/or useful results with the least amount of work.)

What is the Importance of Hierarchy and Decomposition?

- How can we formulate measures of information and complexity that deal with hierarchical "levels"? Presumably as long as the topmost levels of systems 1, 2, and 3 behave exactly the same as (say) machine A, all systems will have the same complexity; however, this would only be true if the behavior was in fact exact under all possible events, so if the topmost level could somehow be even slightly disassembled, it would have greater complexity. (Of course, an algorithm with bounded computational resources would use machine A as an approximation for all three systems if the approximation was "good enough" for its purposes.)

In a hierarchy of levels we might want the predictability of a level relative to the level below it to be roughly constant across all levels. It also seems likely that survivable systems share the property that upper levels actively stabilize lower levels. The desirability of both these properties lies in their ability to simplify a possibly infinite hierarchy so that bounded computational resources suffice to deal with the hierarchy at any particular level. (Note that in nature, the relative time scales of levels (quantum to individual to biosphere) become slower. Is there a "scaling law" that gives a roughly constant value for the difference between time (and space) scales at two adjacent levels? It would be very interesting if a system showed qualitative changes in behavior in a way such that "zooming out" by a constant factor of space and time showed the same apparent complexity.)

A perfect fractal is an excellent example of an object presenting an infinite number

of detailed levels that nevertheless has a simple construction procedure. A stochastic fractal, although seemingly far more complex, is constructively formed from a perfect fractal by just adding a couple of knobs which are spun at each level of construction.

- The basic reason why the division between linear and nonlinear problems is such a fundamental one is that linear problems can be decomposed. Considering top-down control of a system, then, under what conditions or for which systems do we need an arbitrarily large amount of i) information about the state of the system at lower levels, and/or ii) computing power, and/or iii) amount of "force" and/or "work" available to be exerted on the lower levels? How about if we have control units at lower levels?

- If there is friction/randomness at lower levels, what are the implications for control at upper levels and overall stability of the system?

- In order to bound the difficulty of our task, we must be able to build or analyze complex systems from simple ones. To use the watchmaker parable in [Dawkins], the likelihood of successfully constructing an operational watch is far greater if each component is built and tested in turn than if the entire watch is built at once. Structured/modular programming principles give another direct illustration of this idea.

The essential question is: what sorts of systems are amenable to such construction, and what sorts of construction methods work on these systems? [Kauffman] suggests one simple principle based on investigations of Boolean networks: each part

should impinge on relatively few other parts during operation of the system. In the parlance of geneticists (and GA-ists), the system should have low "epistasis".

- What is a simple yet universal description language that can be used for interrelationships between parts of a system? The use of such a language would be to be able to describe the major interrelating parts of a system (organs of a body, subsolutions of a TSP) and the rough effects changing one part has on the other parts.

How Can Massively Parallel Activities be Analyzed?

- Population biology and ecology both deal with the behavior of large numbers of interacting life forms; how do such systems differ from typical sequential computation?
- How do large numbers of simple algorithms combine to compute very complex functions? (Cellular automata and ant colonies are two good examples.)
- How do we model "computation in the system", when instead of algorithms acting on objects, the algorithms are the objects? In many systems (e.g. brains and connectionist models) there is no single locus of computation; the algorithm operates in a distributed, diffuse manner. How can such distributed computation be analyzed?
- How does a parallel computation differ from a sequential one in which computation jumps between different subalgorithms? For what class of problems can cooperating processors perform better than a single processor? Do asynchronous and synchronous parallel computations "converge" in terms of their computational power at finer time granularity?

What is the Importance of "Intelligent Algorithms"?

- What is intelligence and how can it be formally measured in a computational sense?

How many essentially different computational abilities are embodied in the term?

- Many algorithms have been developed that claim to "learn" or to model some aspect of intelligence. What are the abstract properties shared by these "algorithms of intelligence"?

- Can a parametrized framework be constructed which contains all the "algorithms of intelligence" developed thus far as data points? If so, we can search the framework for new types of "intelligence", as well as exploring mappings between parameter space and problems well-solved by the corresponding algorithms.

Where would human and other biological intelligence fit into this framework?

- One of the most exciting properties of general learning algorithms is that the algorithms are embodied in software and can be modified: a general learning algorithm can improve its performance over time. What are some frameworks which allow efficient unbounded increase in "intelligence"? What is the simplest such framework, in the sense of ease of design?

What Well-Defined Fields Deal with these Questions?

Many fields of research have the study of some form of complexity at their core. Each of these domains presents a different perspective, and a different set of mental tools.

Good general overviews of some ideas related to those in this thesis are [Casti], [Cohen and Stewart], [Kelly], [Mainzer], and [Waldrop]. The World Wide Web also provided extremely useful information; the Principia Cybernetica and Extropians Web sites are two good starting points.

The following more specific fields and subfields are some in which the author has found useful sources of ideas, algorithms and knowledge:

Learning Algorithms

Genetic Algorithms - Neural Networks - Machine Learning - Artificial Life

Biology

Evolutionary Biology - Population Biology - Neurobiology - Immune Systems - Morphogenesis

Economics

Free Market Theory - Generalized Currencies

Optimization

**Linear Optimization - Nonlinear Optimization - Combinatorial Optimization -
Heuristics - Simulated Annealing**

Turing Machine Based Complexity

**Computational Complexity - Structural Complexity - Computability - Theory
of Automata and Languages - Kolmogorov Complexity**

**Information Theory - Category Theory - Logic - Stochastic Complexity - Statistics
- Induction - Pattern Recognition - Control Theory - Game Theory - Chaos -
Cellular Automata - Fractals**

COMPUTABILITY

The Turing Machine

The central problem of classical computability is to explore the set of problems ("computable problems") for which solutions can be found via a Turing Machine ("TM"). A TM is an abstract model of computation that is simple but nevertheless as powerful as all other known models of sequential computation (with some possible caveats as mentioned later).

The basic TM model is very simple, consisting only of an infinite tape, a read/write head, and a finite set of program instructions P embodied in a control mechanism. The tape consists of an infinite number of discrete cells, each of which contains a symbol from the tape alphabet; one of these symbols is identified as the blank symbol. (Without loss of generality, we will assume the nonblank symbols to be 0 or 1). The control mechanism can be in any of a finite number of states; two of the states are distinguished, namely the start state and the halt state.

Time proceeds in discrete steps, with the read/write head executing the current instruction from P on each step. At time $t=0$ (before the computation begins) some finite number of adjacent cells on the tape may already be nonblank; in this case, the values in the nonblank cells are called the input. The machine is in the start state, with the head

at the leftmost nonblank symbol.

During each step of the computation, the head reads the symbol directly under it, and then performs the following operation:

- 1) Writes a 0, 1, or blank to the current cell, and
- 2) Moves the head one cell left or right, or keeps it stationary.

The instructions P which control the head can then be represented as quadruples of the form

$(CURRENT_STATE, CURRENT_SYMBOL, OPERATION, NEXT_STATE);$

where the elements of the quadruple have the following interpretation:

$CURRENT_STATE$: the state the machine is currently in.

$CURRENT_SYMBOL$: the symbol under the head.

$OPERATION$: the operation for the head to perform if the machine reads $CURRENT_SYMBOL$ while in state $CURRENT_STATE$.

$NEXTSTATE$: the state to be switched to after the operation is complete.

If the current state and symbol of the TM are ever such that there is no corresponding quadruplet, operation ceases and the TM halts. In this case, the maximal binary string (bordered by blanks) of which some cell is being scanned when the machine halts is the output. For an example of a small TM program with interesting output, see

the section on the Busy Beaver Problem.

A deterministic TM is defined as one in which no two quadruples have the same first two elements; the operation to be performed is thus uniquely specified at every stage. (Nondeterministic TMs are those for which this condition fails; see the section on Computational Complexity for an example of the use of these machines.)

The usefulness of defining problem spaces which are decidable by this class of machines is succinctly expressed by Church's Thesis, which states that a TM can be constructed to solve any "effectively computable problem", where an effectively computable problem is one for which a finite algorithm can be specified which will always give the correct solution. (A great deal of credence has been lent to this assertion by the development of many other models of computation (such as lambda calculus, Post systems, and recursive functions) which have been shown to be equivalent to TMs. In particular, a digital computer (with access to unbounded time and memory) can solve precisely the same class of problems that a TM does.)

Despite the plausibility of Church's Thesis, there is no proof of its universal validity, and in fact it may be the case that problems solvable by physical computers cannot be solved by a TM. Thus the applicability of Computability Theory to the physical universe depends on a "Physical Church's Thesis" (which many people believe does not hold).

There are a countably infinite number of TMs: a computable bijection exists between the natural numbers and the set of TMs. Such a bijection can be found by encoding the instructions for a TM as a list of integers, and then encoding the list as a single integer (obviously both encodings need to be 1-1, with corresponding 1-1 decodings). A ranking of this resulting list of integers completes the bijection; TMs can thus be effectively enumerated.

Using this bijection, a Universal Turing Machine ("UTM") can be constructed; such a UTM can simulate the action of any TM on arbitrary input i , on being given the encoding of the TM and i as input. Essentially, the rules of operation of a TM as described above are encoded into the UTM; the logical operation of a TM along with its environment can then be duplicated in an isomorphic manner by the UTM.

These UTMs are of fundamental importance in Computability Theory and its extensions, as will be seen. The ability to simulate any machine allows results for the reference UTM to be universally applicable to the operations of all machines.

Notice that the input to the UTM is both data and program; the program description is read in and then used to direct the operation of the UTM. Since the program description is in software (on the infinite read/write tape) instead of hardware (the control mechanism), the intriguing possibility of self-examining and self-modifying TMs presents itself.

It is natural to look at extensions of TMs: 1-tape versus n-tape and 1-dimensional versus n-dimensional TMs. Although the class of functions computed is the same in all cases (since a machine of each type can simulate a machine of any other type), differences appear when space and time bounds are added. However, these differences are relatively minor; a particular TM is usually fixed for analysis with the understanding that time-dependent results may differ by a small factor.

Other possible extensions are: sequential versus parallel, synchronous versus asynchronous, continuous versus discrete, and the incorporation of randomness and quantum computation. The theory for all these cases is being actively developed. We will take a brief look at parallel computation in the last portion of the thesis.

Randomness itself is a deep concept, and far from fully understood. As we will see later, if deterministic procedures such as those involved in cryptography can create pseudorandom sequences of any desired "degree of randomness", then "true randomness" is not necessary. Otherwise, a probabilistic TM such as that discussed in the section on Statistical Complexity must have access to an external physical source generating highly random bits. (Proving that such physical sources really are random is extremely difficult, except perhaps in the case of devices measuring fundamental physical processes such as radioactive decay.)

Quantum computation offers perhaps the most intriguing potential; computers are

being developed that can incorporate quantum processes as a fundamental part of their operation, as explained in [Zurek]. The development of these machines (still very much in progress) has led to the "Quantum Church-Turing Thesis": every physically realizable algorithm is representable as a program for a quantum computer. The resolution of this question is essentially dependent on a formulation of a "Grand Unified Theory" of the fundamental laws of physics, along with an understanding of the computational properties of these laws.

Classical Computability

The phrase "Computability Theory" is usually an implicit abbreviation for "resource-unbounded Computability Theory"; no limits are placed on the computation time or storage space available to the TM during its operation. Thus, even if a problem falls into one of the computable categories, it will often be impossible to solve practically.

The classical categorizations of problems in resource-unbounded Computability Theory are:

- Uncomputable
- Computable
 - Recursively Enumerable
 - Recursive
 - Primitive Recursive

where the categories are in order of increasing feasibility. In the Computable domain, each category is a subset of the preceding category.

An uncomputable set is simply a set for which no TM exists that will output only those elements belonging to the set.

Recursively enumerable sets are those for which a program exists to generate all elements (in some unknown order). They are also called partially decidable sets; the

element-generating program for a set constitutes a decision procedure which will halt in a finite amount of time if an element belongs to the set, but continue forever without halting if the element does not belong to the set.

Recursive (or decidable) sets are easily defined as those for which an effective decision procedure (i.e. a TM) exists to decide whether or not an arbitrary element belongs to the set; this decision procedure is guaranteed to halt with the correct answer in a finite amount of time. A recursively enumerable set whose complement is also recursively enumerable is thus recursive.

Primitive recursive sets are a subclass of recursive sets; they are the sets whose characteristic function is a primitive recursive function. (The characteristic function C_S of a set S is defined by $C_S(x) = 1$ if x belongs to S , and 0 otherwise.)

The primitive recursive functions can be characterized in a natural computational manner via loop languages, as we shall soon see. However, the original definition from recursive function theory defines primitive recursive functions as those which can be built up by composition of certain basic functions. The base functions are:

$$\text{SUCCESSOR}(x) = x+1$$

$$\text{ZERO}(x) = 0$$

$$\text{PROJECTION}_j(x_1, x_2, \dots, x_n) = x_j$$

where x is a nonnegative integer. The other primitive recursive functions can be defined

inductively by the following two methods:

COMPOSITION: If $f, g_1, \dots, g_m$ are primitive recursive, then so is the function h defined by

$$h(x_1, \dots, x_n) = f(g_1(x_1, \dots, x_n), \dots, g_m(x_1, \dots, x_n))$$

PRIMITIVE RECURSION: If f and g are primitive recursive, then so is the function h defined by

$$h(0, x_2, \dots, x_n) = f(x_2, \dots, x_n)$$

$$h(j+1, x_2, \dots, x_n) = g(j, h(j, x_2, \dots, x_n), x_2, \dots, x_n)$$

Most common recursive functions are also primitive recursive.

The partial recursive functions can be obtained from the base functions by replacing "primitive recursive" with "partial recursive" in the two methods above, and adding another induction method:

MINIMIZATION: If f is partial recursive, then so is the function h defined by:

$$h(x_1, \dots, x_n) = \{\min y : f(x_1, \dots, x_n, y) = 0\}$$

Note that h may not be defined on all inputs (if no appropriate y exists).

If h is a total function and defined on all inputs, it is referred to as a recursive function. The recursive sets defined earlier are those whose characteristic function is a recursive function, while the recursively enumerable sets are those whose semi-characteristic function is a partial recursive function. (The semi-characteristic function of S is 1 if x is in S , and undefined otherwise.) It is a nontrivial result that a set is

recursively enumerable iff it is the domain or range of a unary partial recursive function.

(Significantly, the idea of recursive functions as given above and the idea of computable sets as those whose characteristic function is computable by a TM were developed independently and then shown to be equivalent. The proof shows how each method can "simulate" the base elements of the other method.)

Godel's Incompleteness Theorem and the undecidability of the Halting Problem are two results which place fundamental bounds on the set of problems which are computable. Godel's Theorem tells us that, in any sufficiently powerful formal system (more precisely, any system which is powerful enough to contain Peano Arithmetic) there are statements which are true but cannot be proven within the system. (This can be proved in many ways; an interesting and unusual proof will be presented in the section on Kolmogorov Complexity.)

The Halting Problem asks whether or not there exists an algorithm (call it H) which can tell in advance if an arbitrary TM will stop on an arbitrary input. Notice the power that H would give us: we would have a recursive algorithm for all recursively enumerable sets. As one might expect, this is too good to be true; it is the case that no such computable algorithm H exists, and hence the Halting Problem is not decidable. (Note that the Halting Problem is, however, partially decidable--simply running the program on the desired input gives a recursively enumerable decision procedure.)

TRADITIONAL PROOF: Suppose a computable algorithm for the Halting Problem exists; by the definition of "computable", this algorithm can be implemented as a TM H . Recalling that the number of Turing Machines is countably infinite, this TM must appear in a listing of all TMs; we shall now derive a contradiction.

Consider the new TM H' derived from H as follows:

$H'(n)$ halts if $T_n(n)$ does not halt

does not halt if $T_n(n)$ halts

(where $T_n(n)$ denotes the behavior of the n^{th} TM on input n , and the halting decisions are made by H).

By construction, H' differs from every TM in its behavior on at least one input (namely, the input n for TM T_n). Thus, H' cannot be a TM. But it must be a TM since by its definition it is computable iff H is computable, and H was assumed computable. Hence there is a contradiction, and H is not computable after all. ☹

This diagonalization method which relies on the countability of TMs is used in many computability proofs. The author has adapted the following more direct proof from a puzzle which asks why a machine that can infallibly predict the future cannot logically exist:

ALTERNATIVE PROOF: Suppose a computable algorithm H for the Halting Problem existed (encoded as a TM). If H can tell whether any TM halts on an arbitrary input, it can certainly tell whether a TM halts when given an encoding of itself as input. We can

thus define a new TM H' which requires only the encoding T of a TM as input, and halts iff T does not halt when given its own encoding as input. (H' simply copies the encoding of T to use as both TM and input for H .)

Consider then the behavior of H' when given the encoding of itself as input. By definition, when H' is given its encoding as input, it halts iff it does not halt when given its encoding as input--a contradiction.

Of course, particular TMs may be shown to stop on certain inputs, or even for all inputs--but any method for proving halting will only be applicable on a subset of all TMs. This "undecidability" result is often used to show various limits on machine capabilities, by showing that the existence of a certain capability would imply the decidability of the Halting Problem.

Another very general undecidability result is Rice's Theorem. This states that it is impossible to find a single algorithm which decides whether a given arbitrary TM belongs to any nonempty proper subset of the set of computable functions. There is thus no completely general method for deciding program equivalence.

Formal Languages and Grammars

Formal languages are an alternative characterization of computation. They are equivalent to TMs in terms of their computational abilities, but sometimes provide more convenient and natural descriptions.

An alphabet A is any finite collection of symbols. A word (or string) $W = w_1w_2...w_n$ on A of length n is a concatenation of n symbols from A . The set of all words on A is denoted by A^* ; any subset of A^* is a language on A .

A grammar G consists of an alphabet A , together with a finite set of rewrite rules $\{I_1 \rightarrow F_1, I_2 \rightarrow F_2, \dots, I_n \rightarrow F_n\}$. The effect of a rewrite rule is to change the string xI_iy to xF_iy . A is partitioned into two sets called the variables and the terminals; one of the variables, S , is distinguished as the "start symbol".

Denoting by T the set of terminals, the language $L(G)$ generated by G can be defined as:

$$L(G) = \{u \in T^* : S \text{ produces } u \text{ using } G\text{'s rewrite rules}\}$$

where a production is a finite sequence of applications of G 's rewrite rules. $L(G)$ thus consists of all those strings (or words) consisting entirely of terminals which can be produced from the start symbol S .

Grammars, TMs, and recursive function theory all give equivalent computation classes. Given a language L , the following conditions are equivalent:

- L is recursively enumerable
- for some grammar G , $L = L(G)$
- L is accepted by a deterministic TM
- L is accepted by a nondeterministic TM

(A TM is said to accept a language L if L consists of all those words which, when given as input to the TM, cause it to eventually halt. Note that nondeterministic and deterministic computers have the same computational power; a deterministic computer can simulate a nondeterministic one with an exponential slowdown in time, by dovetailing all the computation paths that the nondeterministic computer executes.)

If we restrict the types of rewrite rules that can be used, then we get more restricted classes of languages. These restricted languages can equivalently be described by placing constraints on TMs; restricted languages correspond to limited models of computation.

There is an infinite lattice of such computational models (or equivalently classes of languages). The Chomsky hierarchy is a well-known part of this lattice: it consists of the regular, context-free, context-sensitive, and recursively enumerable languages (where each language is a proper superset of the preceding one). A comprehensive introduction to formal languages can be found in [Davis, Sigal and Weyuker].

It is a powerful point in favor of Church's Thesis that very natural classes of languages turn out to be equivalent to natural computational models. Languages, the automata that accept them, and the problems that generate them are all equivalent. We can thus use whichever model is most convenient to the problem at hand. The choice may be compared to using different computer languages while programming; each language will have strengths and weaknesses in areas such as ease of representation and speed of operation, but they all solve the same class of problems.

Measures of grammatical complexity can be defined. Using such measures provides a natural model in which to study problems such as induction and natural language understanding. We will not discuss the grammatical approach further here; for an interesting introduction to research in this area, see [Abiteboul and Shamir], and [Carrasco and Oncina].

Resource-Bounded Computability Theory and Loop Languages

The classes of uncomputable and computable functions discussed previously assume an unlimited availability of time and space; they do not measure the resources required to solve a problem.

As a first formalization of such a measurement of resources, consider the Blum axioms. Recalling that TM programs correspond to partial recursive functions and can be ordered, we associate to each program P_i a partial recursive function F_i such that:

- 1) $\text{Domain}(P_i) = \text{Domain}(F_i)$.
- 2) Determining whether $F_i(x) < y$ is computable.

The sequence of functions F_i is called a complexity measure, and each individual function is called a complexity function. Letting $F_i(x)$ equal the amount of time or space used by the program P_i in computing its output from the input x gives two natural, widely used complexity measures that will be discussed in the section on Computational Complexity.

The first axiom says that each complexity function is defined exactly when its program is; the second says that it is a computable problem to decide whether the complexity function falls within a given bound, for a given input.

It is to be emphasized that these complexity functions are associated with programs for computing functions and not directly to the functions themselves, since every program is equivalent to infinitely many other programs (and it is uncomputable to find these programs, or to decide if two given programs compute the same function).

The known consequences of the Blum axioms by themselves are relatively abstruse and not relevant to the present discussion; some elementary ones can be seen in [Bridges]. They are useful primarily as a framework within which more specific complexity measures can be defined and explored, as we shall do later on.

Let us now examine an actual computational language and its computational capacity, as explored in [Calude]: the LOOP language. It contains the following instructions:

$X = 0$

$X = Y$

$X = X + 1$

LOOP X

END

Here X and Y represent variables which can contain any natural number. The first three instructions are straightforward, with the effect of assigning the value on the right hand side to X; the last two indicate the repeated execution X times of whatever statements are between the LOOP and END instructions. (Changing the LOOP variable inside the loop

is permitted, but does not change the number of times the loop is executed.)

The set L of loop programs is defined recursively by:

- all of the above instructions are in L
- if P and Q are programs in L , then so is the program

P

Q

- if P is a program in L , then so is the program

LOOP X

P

END

We also allow the special labels INPUT X and OUTPUT X ; these must be at the beginning and end of the program respectively, and specify the variables containing the input and output values. They are not counted as instructions, as their purpose is only to indicate which variables contain input and output values.

Here are two sample programs; the first computes $X + Y$, the second computes the predecessor function (0 if X equals 0; $X - 1$ otherwise):

INPUT X, Y

LOOP X

$Y = Y + 1$

```
END  
  
OUTPUT Y  
  
INPUT X  
  
Y = 0  
  
Z = 0  
  
LOOP X  
  
    Y = Z  
  
    Z = Z + 1  
  
END  
  
OUTPUT Y
```

As these programs show, this is quite a natural language. It is surprising but true that the class of functions computed by programs in L is identical with the class of primitive recursive functions defined earlier. (The proof consists of defining the "base elements" of each class in terms of the other class and then using induction to prove equivalence for all functions.)

Let's now look at the properties of L itself. A natural way to classify programs in L is to express L as the union of an infinite hierarchy of classes L_i , where L_i is the set of loop programs having at most i nested LOOP statements. Both of the programs given above are in $L_1 - L_0$.

Clearly each L_i is contained in L_{i+1} ; this hierarchy is in fact proper. To see this, first define the function $t_n(x)$ recursively by:

$t_0(x) = 1$ if $x = 0$, 2 if $x = 1$, and $x + 2$ otherwise;

$t_{n+1}(x) = 1$ if $x = 0$, and $t_n(t_n(\dots t_n(1)\dots))$ otherwise, where t_n is composed x times.

Note that (except for $x = 0$) $t_1(x) = 2x$, $t_2(x) = 2^x$, and $t_3(x)$ is an exponential tower of x 2 's.

It is then the case that the loop program T_n computing the function t_n is contained in $L_n - L_{n-1}$, for $n \geq 1$. A counting argument shows that T_n cannot be in L_{n-1} . Assuming T_n is in L_n , the following program in L_{n+1} computes T_{n+1} :

```

INPUT X
Y = 0
Y = Y + 1
LOOP X
    Y =  $T_n(Y)$ 
END
OUTPUT Y

```

It only remains to find a program in L_1 for the base case t_1 , which can be done by modifying the above program: simply replace the statement " $Y = T_n(Y)$ " by the two statements " $X = X + 1$ " and " $Y = X$ ".

L can be extended in a natural way by adding the following two types of

instructions:

IF $X \neq 0$ GO TO Label

Label

where "Label" denotes any of an infinite number of distinct labels. The effect of the new instructions is to allow testing of the value of any variable; if the value is zero, program execution transfers to the statement following the "Label" statement.

Denote by *UL* (for "universal loop language") the class of all programs that can be made by adding the new instructions. While programs in *L* compute the class of primitive recursive functions, programs in its superset *UL* compute the broader class of partial recursive functions (i.e. computable functions).

We can easily define a useful complexity measure *TIME* for *UL*, by associating with each program and input the number of instructions executed before the program halts on the given input. (If a program never halts, the number of instructions is defined to be infinite. *UL* programs can easily be effectively numbered, and their running time is clearly computable; the Blum axioms are thus satisfied.)

Using *TIME*, we can state the following interesting theorem which is discussed in [Calude]:

The function *f* is computed by a program in L_n iff there is a $P \in UL$ which

computes f and $\text{TIME}(P(x)) \leq t_4(x)$.

This implies the non-feasibility of computing non-primitive recursive functions. Even primitive recursive functions computed by programs in $L_n - L_{n-1}$ for $n \geq 4$ are infeasible. This is so since $\text{TIME}(P(x)) \geq t_3(x)$ for such programs (recall that t_3 represents an exponential tower of 2's!).

The "feasible" primitive recursive functions that belong to L_3 are called the elementary functions. These functions include almost all commonly used mathematical functions; they can be characterized in a very natural way in a manner similar to primitive recursive functions, as shown in [Cutland].

As a final note, recall that the problem of deciding whether two general programs are equivalent is undecidable. What is the corresponding situation for the loop hierarchy? It turns out that the class of functions computed by L_1 (also called the class of simple functions) is the highest level in the loop hierarchy where program equivalence is decidable.

The Busy Beaver Problem

To illustrate the unreasonable complexity and difficulty of analysis of even small algorithms, let's look at a fascinating puzzle that captures the very essence of "measuring complexity": the Busy Beaver Problem.

The problem is simply stated: How does the function $BB(n)$ behave, where $BB(n)$ is the maximal (finite) number of 1's output by any n -state (2-symbol) Turing Machine on a null input?

By definition, machines which never halt are excluded; since there are a finite number of machines with n states, the question is well-defined. The first four values of the function are:

$$BB(1) = 1$$

$$BB(2) = 4$$

$$BB(3) = 6$$

$$BB(4) = 13$$

Here is the program for a 3-state Busy Beaver: (Note that HALT is not counted as a separate state. We are also using 0 as the blank symbol, and the tape initially contains all 0's.)

	Input Symbol	
State	0	1
S_1	1,R, S_2	1,L, S_3
S_2	1,L, S_1	1,R, S_2
S_3	1,L, S_2	1,HALT

And here is the action of this Busy Beaver on an initially blank tape; the subscripts indicate the current state of the machine and the symbol it is reading:

n	Status of tape after n steps					
0	0	0	0	0_1	0	0
1	0	0	0	1	0_2	0
2	0	0	0	1_1	1	0
3	0	0	0_3	1	1	0
4	0	0_2	1	1	1	0
5	0_1	1	1	1	1	0
6	1	1_2	1	1	1	0

7	1	1	1_2	1	1	0
8	1	1	1	1_2	1	0
9	1	1	1	1	1_2	0
10	1	1	1	1	1	0_2
11	1	1	1	1	1_1	1
12	1	1	1	1_3	1	1
13	1	1	1	1	1_{HALT}	1

We can see that even for a mere 3-state machine, the behavior is nontrivial!

From the first four values for $BB(n)$ given above, one might suppose that the function grows fairly slowly. In fact, for such an intuitively defined function its growth is nothing short of spectacular:

$$BB(5) \geq 4\,098$$

$$BB(6) \geq 95\,524\,079$$

and $BB(12) \geq 6 * \text{POWERTOWER}(4096, 166)^4$, where $\text{POWERTOWER}(a, y)$ denotes an exponent stack of y a 's.

Such awe-inspiring growth makes the following result less surprising:

THEOREM: $BB(n)$ grows faster than any computable function, and is thus itself uncomputable.

PROOF (by contradiction, by the author): Suppose $BB(n)$ was computable (i.e. a fixed size program outputs $BB(n)$ when fed n as input). Then consider the following program:

```

LOOP  $BB(n)$ 
    print a "1"
END
print a "1"

```

This program can be translated into a fixed size TM T with input n and a fixed number $|T|$ of states.

We can generate a new program T' from T which prints the same number of 1's on a null input, by prepending to T $\lceil \lg n \rceil$ states which print $\lceil \lg n \rceil$ 1's and a constant number c of states that raise 2 to the power of 1's printed. These initial states of T' generate at least n 1's on the tape that can then be used as input to T (which forms the later states of T'). Choosing n large enough that $\lceil \lg n \rceil + |T| + c$ (i.e. the number of states of T') is less than n , we have a contradiction since T' with less than n states prints one more 1 than $BB(n)$. ☹

Instead of measuring the number of 1's output, one can measure the number of "shifts" or times that the read/write head moves; the corresponding Busy Beaver function then defines the slowest n -state program.

Many questions are suggested by $BB(n)$. Note that the Busy Beaver function corresponds to a worst-case complexity measure; what properties does the average n -state program possess? What is the connection with Ramsey Theory, where there are similar blow-ups in size? What is the complexity of calculating BB or similar functions? What properties do "typical programs" (which could mean either random programs or programs used in practical situations) have, and in which of these properties (besides execution time) do they differ from very slow programs? And a final unexpected relationship introduced in [Michel]: how does BB relate to the function on natural numbers defined by $T(2n)=n$ and $T(2n-1)=3n-1$, which appears to eventually reach the steady state cycle $(1,2,1,2,...)$ when repeatedly iterated for any starting value (but is not yet proven to do so)?

It has been suggested quite seriously that the value of $BB(6)$ may never be known. The reasoning behind this depressing projection is that calculating a value of the Busy Beaver function may be as difficult as solving the Halting Problem for all machines of the size in question; unfortunately, TMs of quite modest size are sufficient to encode extremely difficult questions.

COMPUTATIONAL COMPLEXITY

The problem with using classical computability classes to differentiate easy and difficult problems is that no account is taken of the amount of space and/or time required to actually compute the output. The computability of a function is thus a necessary but far from sufficient condition for the function to be "practically computable".

By Church's Thesis, any "effectively computable" problem is computable by a TM. It is therefore clearly of interest to classify problems computable by a TM on the basis of the amount of time or space required for the computation, or by other constraints; such classes will give an indication of the amount of computational resources the problem might require to solve. The study of problems and the classes they fall into is the province of Computational Complexity theory; [Papadimitriou] is an excellent introduction. (Structural Complexity, which we will not discuss here, is a somewhat different approach that emphasizes abstract structures and reducibilities of problem classes; the field is summarized in [Balcazar, Diaz and Gabarro 88 & 90].)

We will be using the concept of a nondeterministic TM in the following discussion; as defined before, this is a TM which may have more than one permissible action in a given configuration. Such a TM is said to solve a problem under constraint C if the shortest permissible sequence of operations for which the solution will be found satisfies C. (The constraint C could be a time or space bound, or some other form of

restriction.)

To determine the maximum possible amount of resources required to solve a problem, the worst possible case is imagined, and any other case is proved not to be harder to solve than this worst case; such a characterization is called "worst-case analysis". Problems can be grouped together according to their worst-case time or space requirements into "complexity classes". (Note that this is only one possible way of categorizing problems (albeit a very natural one). We will see others shortly.)

Some useful worst-case complexity classes:

$\text{DTIME}(t(n))$: languages accepted by a deterministic Turing Machine in time $O(t(n))$.

$\text{NTIME}(t(n))$: languages accepted by a nondeterministic Turing Machine in time $O(t(n))$.

$\text{DSpace}(s(n))$: languages accepted by a deterministic Turing Machine in space $O(s(n))$.

$\text{NSpace}(s(n))$: languages accepted by a nondeterministic Turing Machine in space $O(s(n))$.

P : union of $\text{DTIME}(n^c)$ for all c . Consists of those problems whose worst-case solution time is bounded by some polynomial.

NP : union of $\text{NTIME}(n^c)$ for all c . Consists of problems for which a correct solution can be verified in worst-case time bounded by a polynomial.

PSPACE : union of $\text{DSpace}(n^c)$ for all c . Consists of those problems whose worst-case

solution space is bounded by some polynomial. (The space required is defined as the minimal size finite tape on which the computation can be carried out.)

(NPSPACE can be defined analogously, but is in fact equivalent to PSPACE.)

As an example of a nondeterministic complexity class, consider the SAT problem: decide if a given n -variable Boolean expression in conjunctive normal form has a satisfying truth assignment. (A Boolean expression in conjunctive normal form is one which is the logical AND of a set of clauses, each of which is the logical OR of a set of (possibly negated) Boolean variables. An example is $(A \text{ OR } \sim B \text{ OR } C) \text{ AND } (B \text{ OR } \sim C)$, where $\sim X$ denotes NOT X ; note that such expressions are easily evaluated, since every clause must be satisfied. Every Boolean expression can be transformed into an equivalent one in conjunctive normal form.)

The problem of deciding whether there is an assignment of the values TRUE and FALSE to each of the variables in an n -variable Boolean expression which makes the whole expression TRUE is in $\text{NTIME}(n^2)$. A nondeterministic machine generating all truth assignments can check each one to see if it satisfies the expression, and if any such truth assignment exists the SAT instance is decided positively.

Notice the distinction between P and NP. P contains problems which can be solved in polynomial time, while NP contains problems for which any solution has a "certificate" which verifies the solution in polynomial time. (This certificate can be

thought of as encoding an accepting computation of the nondeterministic machine; by "fixing" one computation path out of all the possible computations, a deterministic machine can execute this computation and verify that it does indeed solve the problem.)

It is the case that $P \subseteq NP \subseteq PSPACE$; whether either or both of the inclusions are proper is a well-known open question. There is a vast array of problems which are known to be in NP but have not been shown to belong to P.

Note the difference between decision and function problems. The definitions above are for classes of algorithms which decide languages, i.e. which give yes/no answers to questions. One may, however, want a functional answer: what is the minimum of a given function over its domain? Usually (but by no means always), a decision algorithm can be used to give an efficient answer to a functional problem. One method is by simply repeating the algorithm with different values and doing binary search, which amounts to asking for the bits of the answer successively and entails only a logarithmic slowdown.

An oracle TM is a TM T which has access to a special oracle O (denoted T^O). At any computation step, T can ask whether a string on its work tape belongs to the language O . A language A is then said to be polynomially reducible to B if there is an oracle TM T^B that accepts A in polynomial time.

A problem (language) H is NP-hard if a general solution method for the problem would allow a solution to every problem in NP with no more than a polynomial loss of time; in other words, if a polynomial time oracle TM N^H exists for every problem in NP. An NP-hard problem which belongs to NP is called NP-complete; a version of the SAT problem mentioned before was the first shown to be NP-complete. It is sometimes thought that problems in NP are either NP-complete or in P; in fact, if NP is not equal to P, then NP contains problems which are neither NP-complete nor in P.

Problems shown to be NP-hard are usually considered to be "intractable", i.e. unsolvable exactly except for small instances; this is so since no polynomial-time algorithm has been found for any NP-hard problem. (By the definition of NP-hard, the existence of such an algorithm would imply tractability for all problems in NP.)

Note that all complexity classes defined so far have been worst-case ones; the algorithms are guaranteed to perform under the constraint for all possible instances of the problem. While this kind of guarantee is reassuring to have, it may be difficult or impossible to find. It will also often be of interest to know how an algorithm performs for "most" problems, or for problems of a certain category.

Approximability

As NP-complete problems are unsolvable exactly in feasible amounts of time, one naturally wonders whether it is possible to find faster algorithms which will find "approximate" solutions. We can thus consider extending the notion of computability to measuring the complexity of approximations to a function, as opposed to the exact function. From a philosophical point of view, this is necessary for finding practical solutions to problems which are difficult or impossible to solve exactly (or indeed deciding whether such practical solutions exist).

A first step toward formalization of approximate computability is the notion of an ϵ -approximation algorithm. This is an algorithm where, for every problem instance x , the value $SOL(x)$ of the solution found satisfies:

$$|SOL(x) - OPT(x)| / \max\{SOL(x), OPT(x)\} \leq \epsilon$$

(where $OPT(x)$ is the value of the optimal solution for the problem instance x , and all solutions are assumed positive).

Note that the condition to be satisfied is valid for minimization and maximization problems, and is scaled by $\max\{\text{solution found}, \text{optimal solution}\}$ so that ϵ -approximation algorithms for a fixed ϵ and different problems will all be guaranteed to return a solution

with error at most ϵ . Valid values of ϵ are $0 \leq \epsilon \leq 1$; a value of 0 indicates an exact algorithm, which can be explored via the exact complexity classes discussed earlier.

Also note that, although the output of our algorithm may be a real number, the TM will actually only give the answer to a certain number of digits of accuracy. We can fix the desired accuracy and then compare algorithms; increasing the number of digits required will increase the time of execution of the algorithm by a low-order polynomial (possibly linear, depending on the algorithm).

For any problem for which no polynomial-time exact solution method is known, it will be of interest to characterize the polynomial-time ϵ -approximation schemes known. With this in mind, we can define the approximation threshold of a problem X as the greatest lower bound of all $\epsilon > 0$ such that a polynomial-time ϵ -approximation algorithm exists for X .

The approximation threshold can equal 0 (an arbitrarily close solution can be found), 1 (virtually no approximation is possible), or any value in between. Note that, if $P = NP$, every optimization problem in NP has approximation threshold 0.

To illustrate this notion for a well-known problem, we'll examine the Travelling Salesman Problem (TSP). Unfortunately, this problem is quite intractable in terms of approximation, as the following theorem presented in [Papadimitriou] shows:

THEOREM: The TSP has approximation threshold 1, unless $P = NP$.

PROOF: We'll show that an approximation threshold below 1 for the TSP implies a polynomial-time exact algorithm for the Hamilton Cycle problem ("HC") (recall that this is the problem of deciding whether or not a graph contains a cycle which visits every vertex exactly once). Since it is well known that HC is NP-complete, this gives a polynomial-time exact algorithm for every NP problem, which implies $P = NP$.

Assume the TSP has approximation threshold less than 1. This implies the existence of a polynomial-time ϵ -approximation algorithm for the TSP for $\epsilon < 1$.

The polynomial-time exact algorithm for HC for some graph $G = (V, E)$ starts by constructing an instance of the TSP from G . This instance has $|V|$ cities, with the distance between each pair of cities being 1 if the corresponding vertices in G have an edge between them, and $|V|/1-\epsilon$ otherwise.

The polynomial time ϵ -approximation algorithm is then applied to this TSP instance. If the resulting solution has length $|V|$, then only unit-length edges were used, and G has a Hamilton cycle. Otherwise, at least one edge of length $|V|/1-\epsilon$ was used, and this length is a strict lower bound for the length of the solution found.

Since the algorithm used was ϵ -approximate, the actual optimum tour must have length at least $1-\epsilon$ times the length of the solution found. Thus the optimum tour has length strictly greater than $|V|$, and G has no Hamilton cycle.

We have thus created a polynomial-time exact algorithm for HC, which is only possible if $P = NP$, and the theorem is proved. $\odot$

It is natural to wonder if there is some analogue of the worst-case exact complexity classes (such as P and NP) for approximation problems. Is there some natural characterization of problems that are easy or hard to approximate?

A promising result in this direction is the development of the class MAXSNP, which consists of maximization problems which can be expressed in a certain subset of existential second-order logic. (The interest in this type of logic is motivated by Fagin's Theorem, which states that all languages in NP are reducible to a graph-theoretic property which can be expressed in existential second-order logic.) All optimization problems in the class MAXSNP have ϵ -approximation schemes with $\epsilon < 1$ (which may or may not be polynomial-time).

Interestingly, it turns out that MAXSNP-complete problems have polynomial-time approximation schemes iff $P = NP$. Hence, this class gives many natural problems which are almost certainly not approximable.

Randomized Complexity Classes

We come now to the idea of randomized complexity classes. Consider the converse of ϵ -approximation algorithms. Instead of an algorithm which always gives an answer to within a certain fixed approximation, consider algorithms which, within a certain fixed probability, find a solution to a given decision problem. This is the Monte Carlo approach; do a large number of trials with an algorithm which "probably" gives the correct answer, and then the conclusion reached from many trials will almost certainly be correct.

In order to formally incorporate a probabilistic feature into our complexity classes, we'll use nondeterminism, which allows the equivalent of coin-flipping. To standardize and simplify bookkeeping, we will deal with precise nondeterministic Turing Machines; these are nondeterministic TMs for which all computations halt after the same number of computation steps. Very little is lost by restricting attention to such TMs; if a TM exists which decides a language L in time (or space) $f(n)$, then there is a precise TM which decides L in time (or space) $O(f(n))$.

Define a polynomial Monte Carlo TM for a language L to be a precise nondeterministic TM with all computations halting in time $p(x)$ (a polynomial in the input x), such that the following is true: If $x \in L$, then at least half the computations halt with a "yes" answer; if $x \notin L$, then all computations halt with a "no" answer.

(Informally, for every time we want to flip a coin we define two transitions, which tell the actions to be taken on the two possible outcomes of the coin flip. At this stage, the nondeterministic computation will "branch" into two computations. For all other steps of the computation two transitions are also defined, but these two transitions do not change the state of the machine--they are merely for the purpose of counting the number of computations.)

Note that the fraction of computations halting with a "yes" can be any fixed k with $0 < k < 1$; a different value of k will change the time required to get the same probability of a false negative by a constant time factor. (To see this, note that repeating the algorithm r times makes the probability of false negatives k^r ; for this to equal $1/2$, we choose the expected number of repetitions $r = -1/\lg k$.) Random choices in the computation which require a choice between more than two alternatives can be decomposed into binary choices (for the bits of the required quantity).

The class of all languages with polynomial Monte Carlo TMs is called RP (randomized polynomial time). Since NP is the class for which there is a polynomial nondeterministic TM, RP is contained in NP; RP contains P since every polynomial deterministic TM is also a polynomial Monte Carlo TM.

RP is an interesting complexity class. Given a precise nondeterministic TM, the problem of deciding whether or not it is a polynomial Monte Carlo TM for some

language is in fact undecidable; there is no general way to tell if such a TM will either accept by majority or reject unanimously for all inputs. Such classes are called semantic classes. This is in contrast with classes such as P and NP, where a given precise nondeterministic TM can easily be checked for inclusion in the class. Another way to distinguish semantic and syntactic classes is the unified treatment of complexity classes given by leaf languages, which will be discussed later.

The complement of RP (denoted coRP) can be defined analogously to RP; it is the class of problems which have a Monte Carlo algorithm with a bounded probability of false positives but no false negatives.

The intersection of RP with coRP (denoted ZPP) is a nice class; these problems have two Monte Carlo algorithms, one with no false positives and one with no false negatives. By defining a new algorithm which consists of running both algorithms concurrently, a definitive positive or negative answer will occur sooner or later (if the probability of false positives and negatives is $1/2$, running both algorithms n times will give a definitive answer with probability $1 - 2^{-n}$). This type of algorithm is called a Las Vegas algorithm; the name ZPP comes from "Polynomial randomized algorithm with Zero Probability of error". It turns out that deciding primality is in ZPP.

A broader complexity class which contains all of the classes RP, co-RP, and ZPP is PP. This class contains those languages for which there is a precise nondeterministic

polynomially bounded TM such that, for all inputs, the input belongs to the language iff more than half the computations of the TM on that input halt with answer "yes". (In other words, the TM decides the language "by a majority".) Note that NP is contained in PP.

Although PP looks nice, it is not useful as it stands because many languages in PP cannot be solved efficiently. The TM for an input may have only slightly (perhaps even a constant number) more accepting computations than rejecting computations. Such a TM can't be used to efficiently decide the membership of an input; there is no feasible method of deciding whether there are more accepting or rejecting computations. To see this, consider the following corollary of the well-known Chernoff Bound for Bernoulli random variables:

If $p\{X=1\} = .5 + c$ for the binary random variable X , then the probability of the sum of n independent outcomes of X being less than $n/2$ is at most $\exp(-c^2n/6)$.

This implies that setting the number n of outcomes roughly equal to c^{-2} will suffice to determine whether the bias c is positive or negative. However, the "bias" c in the class PP can be as low as $2^{-\text{TIME}(|x|)}$, where $\text{TIME}(|x|)$ is the number of steps every computation takes on the input x ; an exponential number of outcomes would be required in this case.

PP thus needs to be modified to eliminate the possibility of an exponential number of trials for a high probability answer. This can be done quite simply by raising the probability of a right answer to be strictly greater than $1/2$; this gives the class BPP ("Bounded Probability of error").

Formally, BPP contains those languages for which a precise nondeterministic polynomially bounded TM exists with the following property: there exists k with $1/2 < k < 1$, such that for all inputs that belong to the language, at least a fraction k of the computations on each input accept; conversely, for all inputs that don't belong to the language, at least a fraction k of the computations on each input reject. Note the inclusion of RP in BPP, and BPP in PP; whether BPP is in NP is open.

BPP is a very natural semantic complexity class, and widely accepted as a good model of "realistically solvable problems". It contains those problems for which an algorithm exists that will almost certainly solve the problem in polynomial time.

Leaf Languages

The plethora of complexity classes defined so far suggests looking for a unified framework within which to describe them. Let's examine one such framework.

Consider a nondeterministic polynomially bounded TM N , which solves a decision problem; standardize it as before, so that it is precise (halts after precisely the same number of steps $p(n)$ for every input of length n) and has exactly two choices at each computational step (order them as the first and second choice). When given x as input, where $|x| = n$, N produces a computation tree of depth $p(n)$; by the conditions on N , this is a complete binary tree with $2^{p(n)}$ leaves. Each leaf will be labelled 1 or 0 ("yes" or "no" respectively) depending on the outcome of the associated computation.

Since the choices were ordered, the leaves are ordered as well, and the overall computation of N on any particular input x can be associated with a binary string in $\{0,1\}^{2^{p(|x|)}}$. Call this binary string $N(x)$; it is simply the list of all outputs of the nondeterministic algorithm.

Let a leaf language be a language L from $\{0,1\}^*$; let A and R ("Accepting" and "Rejecting") be two disjoint leaf languages. Any two such languages A and R can be used to define a complexity class C , as follows: let $C(A,R)$ be the class of all languages L such that there exists an N (as above) for which $x \in L$ iff $N(x) \in A$, and $x \notin L$ iff $N(x) \in R$.

As examples of the unifying power of this framework, consider the following language definitions of some complexity classes we've already discussed (let $1(x)$ and $0(x)$ denote the number of ones and zeroes in the string x , respectively):

$$\text{P:} \quad A = 1^* \quad R = 0^*$$

(A nondeterministic TM for a problem in P is guaranteed to give a definitive answer in polynomial time. All computation paths proceed to the correct "yes" or "no" answer.)

$$\text{NP:} \quad A = \{0,1\}^* 1 \{0,1\}^* \quad R = 0^*$$

(Every problem in NP has a "certificate"; if the correct answer is "yes" and the right guesses are made (i.e. the right computation path is followed), then a "yes" answer can be found in polynomial time. A "no" answer is always decided correctly in polynomial time.)

$$\text{RP:} \quad A = \{x \in \{0,1\}^* : 1(x) > 0(x)\} \\ R = 0^*$$

(If the correct answer is "yes", the algorithm gives more "yes" than "no" answers. "No" answers are always decided correctly in polynomial time.)

$$\text{BPP:} \quad A = \{x \in \{0,1\}^* : 1(x) > k0(x)\} \quad 1/2 < k < 1 \\ R = \{x \in \{0,1\}^* : 0(x) > k1(x)\}$$

(If the correct answer is "yes", the algorithm should give strictly more "yes" answers than "no" answers, and conversely if the correct answer is "no".)

Note that this formalism suggests a definition of the syntactic and semantic classes mentioned before; syntactic classes are those for which R is the complement of A . (Although the (A,R) pair given for the syntactic class P is not complementary, there is a less intuitive pair for P which is complementary; for further details on leaf language formalisms and their applications, see [Bovet, Crescenzi and Silvestri].)

Looking at these definitions, it is interesting to note the use of formal languages inside the complexity class definitions. One wonders how the properties of grammars in the definitions relate to the properties of the classes themselves.

Cryptography and Zero-Knowledge Proofs

As we will see in later sections, dealing with randomness is a central problem in attempting to formulate measures of complexity; the lack of compressibility and pattern in most sequences has many deep consequences. However, in some situations the generation of randomness is desirable; one such situation is to generate sequences for use in randomized algorithms. Another, where one wishes to reversibly transform a given sequence so that it appears random, is found in the theory of cryptography.

The basic problem which cryptography attempts to solve is that in which two parties (traditionally referred to as Alice and Bob) wish to send messages to each other. The messages may be intercepted by a hostile third party; Alice and Bob thus agree beforehand on an encoding transformation ("encryption") to be applied to all messages. This encoding is such that any party without the inverse transformation should find it computationally infeasible to calculate the decoding method ("decrypt the message"). Of course, Alice and Bob should easily be able to decrypt each other's messages.

(Note that the problems in this section are different from those discussed before. Previously, we had a single decision or function to be attacked and solved by an algorithm. Now, we are looking at a more general case where a "protocol" must be devised to meet certain given conditions. The protocol may of course involve decision or function algorithms of the type discussed before, but it is a solution to a dynamic

situation as opposed to a static one.)

Clearly, not any transformation will do. The ideal cryptographic function f will have the following properties:

- f is 1-to-1, and does not increase or decrease the length of the input by more than a polynomial.
- f can be computed in polynomial time.
- f^{-1} should not be "feasibly computable". (Note that, instead of the usual worst-case requirement, this should take the form of requiring that polynomial-time inversions should be possible only on a vanishingly small set of inputs.)
- There is a function (the "decryption key") which, if known, allows an easy (i.e. polynomial-time) calculation of f^{-1} .

Such a function f is called a trapdoor function; computation is easy one way, but very difficult the other way unless one has the "key". (Functions that satisfy the first three conditions, but not necessarily the fourth, are called one-way functions.)

The well-known RSA cryptosystem uses such a trapdoor function. Alice creates a pair (e,d) of encrypting and decrypting integers (this can be done quickly). The encryption key is made public, and corresponds to the function f above; encrypting is quick. Alice can decrypt quickly knowing d , but for anyone else the decryption process is computationally infeasible.

The security of the RSA cryptosystem and others similar to it is of great practical importance, as these cryptosystems are used to ensure the security of most electronically transmitted data. However, this issue is also highly relevant for exploring randomness and related concepts. The ability to invertibly hide information (i.e. to make it appear random while still retaining the ability to recover the original information) implies the ability to efficiently and deterministically create a pseudorandom sequence, which appears random to resource-bounded tests (as mentioned later in the section on resource-bounded Kolmogorov Complexity)

Another intriguing idea that cryptography opens up is that of zero-knowledge proofs. Consider problems in the class NP. By definition, a solution to a particular instance of an NP problem can be verified as such in polynomial time; solutions in NP may be difficult to find but they are easy to verify once found.

Suppose, then, that Alice has found a solution to an NP-complete problem. Recall that these are the most difficult problems in NP; a polynomial time solution method for an NP-complete problem implies a polynomial time solution method for all problems in NP. Alice can therefore be justifiably proud; she may wish to let Bob know of her achievement. One way of doing this is simply to show her solution to Bob, who can then verify it as a solution in polynomial time.

However, this would allow Bob to show the solution to others and claim the credit

for himself, if he wished. To guard against this possibility, Alice would like to convince Bob beyond a reasonable doubt that she has found a solution, without actually divulging her solution. Such a "proof" method is called a zero-knowledge proof.

To illustrate the idea, consider the NP-complete problem of 3-coloring: coloring each of the vertices v_i of a graph with one of three colors, so that no two adjacent vertices have the same color. (Denote the color for vertex v_i by c_i .) We can then use a cryptosystem such as the RSA cryptosystem to implement the zero-knowledge protocol, as follows.

First, Alice and Bob agree on the graph to be colored. Alice then permutes the colors (i.e. changes each c_i to $\text{PERM}(c_i)$, where PERM is a random permutation) and generates a separate encryption-decryption pair (e_i, d_i) for each vertex. She encrypts each $\text{PERM}(c_i)$ to get the encryption $f(e_i, \text{PERM}(c_i))$, and then gives Bob e_i and $f(e_i, \text{PERM}(c_i))$ for all vertices i .

Bob now picks an edge of the graph at random, and asks if the corresponding vertices (v_i, v_j) have different colors. Alice replies by revealing the decryption keys d_i and d_j , which allows Bob to compute $\text{PERM}(c_i)$ (i.e. $f^{-1}(d_i, f(e_i, \text{PERM}(c_i)))$) and $\text{PERM}(c_j)$ and compare them for equality.

Suppose that Alice does not have a valid 3-coloring; since each edge is chosen at

random out of $|E|$ edges, Bob will discover that $\text{PERM}(c_i)$ and $\text{PERM}(c_j)$ are different with probability $1/|E|$. Repeating the process for k independent rounds will thus allow Bob to uncover an invalid 3-coloring with probability $1 - (1 - 1/|E|)^k$, which rapidly approaches 1.

This probabilistic proof-method is indeed "zero-knowledge". At each round, Bob receives only some cryptographic keys, the encrypted colors (which contain no information if the cryptosystem is secure), and two permuted colors. Since the permutation and encryption-decryption pairs are changed every round, no information about the actual colors is given away. Of course, the validity of the entire method hinges on the existence of unbreakable cryptosystems; although intensive effort has failed to yield a solution method for systems such as RSA, no cryptosystem has yet been proved unconditionally secure.

It is quite remarkable that solutions to difficult problems can be verified without revealing any information about the solution. This result can be generalized; using reduction, it is shown in [Goldreich, Micali, and Wigderson] that the above zero-knowledge proof for the NP-complete problem of 3-coloring implies that every problem in NP has a zero-knowledge proof.

Speculations on ϵ -Extensions

We've looked at algorithms which are guaranteed to give an approximate solution for optimization problems, as well as Monte Carlo algorithms. We can extend this line of reasoning and look for algorithms which are only guaranteed to give us approximate solutions some guaranteed percentage of the time.

Many situations only require algorithms to work to a certain desired degree of accuracy, with given probability. This immediately leads to the following proposed classes:

ϵ -P: An algorithm exists which is guaranteed to find a solution within a factor ϵ of optimal in polynomial time. This is essentially the same as the ϵ -approximable problem class discussed earlier.

ϵ^2 -P: An algorithm exists which is guaranteed to find a solution within a factor ϵ_1 of optimal in polynomial time, with probability $1 - \epsilon_2$. This is similar to the notion of PAC (probably approximately correct) learning in machine learning theory.

Higher levels can be nonuniquely defined by letting further ϵ values define broader notions of the required closeness of the solution found to the optimal solution; instead of being within a strict factor of the optimal solution, the solution found may be only "polynomially close", or may only approximate some subset of the solution.

Of course, there is an infinite lattice here--but how useful would the upper levels be? The author conjectures that after levels 3 or 4 the categories would be essentially useless in any practical sense, since i) intuitively it would require an extremely deep understanding of the problem space to be able to guarantee performance through several levels of abstraction, and ii) the categories would probably not be stable (see below). Are there any natural problems in higher levels of the lattice?

The intent of the ϵ hierarchy is to allow classification of algorithms that only give "approximate information, some of the time, for part of the problem, perhaps..."; the principle is that, in practice, any (true) information is better than none.

The author conjectures that, for any "reasonably complex" system, it will in general not be possible to prove which of these classes an algorithm belongs to. Furthermore, if such systems are non-stationary then the algorithm will probably continually change classes along with ϵ values. Thus, perhaps the only reasonable expectation is to continually evaluate the performance of the algorithm and approximate the correct class from this.

More generally, we can consider classes which express some other factor than solution time, such as predictability, controllability, and so on. It would be very interesting and useful if these problem classes turned out to be largely redundant, by being reducible or equivalent to time solvability classes and thus relatively easy to

characterize. Factors such as predictability, controllability, decomposability, and "representation in an easy-to-understand form" are the core issues with which we continually grapple in solving problems.

Once these other problem classes have been defined, it would be interesting to cross-fertilize the different types of factors, in a manner analogous to the ϵ^1 -P classes. For instance, one could define a class of feedback systems with a polynomial-time algorithm controlling the system to within an accuracy A , with this control algorithm being decomposable by a factor B . Combining such a class with a good average-case performance guarantee would yield algorithms that are ideal in several important respects.

INFORMATION THEORY

Basics

Consider a message being sent between two parties which is composed of symbols from a fixed alphabet. Without considering in any way the meaning of the message, is it possible to do anything useful using only the relative frequencies of occurrence of each symbol? As it turns out, the answer is yes; the frequencies alone are sufficient to define encodings which compress the message and protect it from corruption by noise.

The central quantity of measurement in Information Theory is entropy; this is a measurement of uncertainty defined on discrete random variables. For a discrete random variable X , let the probability mass function be defined by $p(x) = \Pr\{X=x\}$, for $x \in A$. Then the entropy $H(X)$ of X is defined by:

$$H(X) = -\sum_{x \in A} p(x) \log p(x)$$

(Usually the logarithm is taken to the base 2 and H is measured in bits.)

$H(X)$ will be shown to be the average description length necessary to specify an outcome of X .

The joint entropy $H(X,Y)$ and conditional entropy $H(Y|X)$ of a pair of random variables can also be defined. These quantities have the natural property that $H(X,Y) = H(X) + H(Y|X)$; the joint entropy of two random variables is the entropy of the first plus the conditional entropy of the second given the first. Formally, they are given by:

$$H(X, Y) = - \sum_{x \in A} \sum_{y \in B} p(x, y) \log p(x, y)$$

$$H(Y|X) = \sum_{x \in A} p(x) H(Y|X=x)$$

The mutual information $I(X;Y)$ is a natural next quantity to define:

$$I(X;Y) = H(X) - H(X|Y)$$

or the reduction in entropy of the random variable X given knowledge of Y . From the fact that $H(X,Y) = H(X) + H(Y|X)$, it follows that:

$$I(X;Y) = H(X) + H(Y) - H(X,Y), \text{ and}$$

$$I(X;X) = H(X).$$

Finally, relative entropy (also called "Kullback-Leibler distance") measures the "distance" between two probability distributions. (Note that although it is always nonnegative and 0 only when the distributions are equal, it doesn't satisfy the triangle inequality and is not symmetric. Thus, it is not a formal metric.)

As we will show in the subsection on Optimal Coding, relative entropy has a natural interpretation as the number of bits in excess of $H(p)$ needed to describe an

outcome under p , if we use the distribution q instead of the distribution p for encoding. (This may happen if, for example, the distribution q is the best available approximation for p .) Relative entropy can be defined as:

$$D(p||q) = \sum p(x) \log \frac{p(x)}{q(x)}$$

Uniquely Decodable and Prefix Codes

Given a string S consisting of elements x from A , we may wish to compress S . In order to do this, we encode the elements of S with a code C .

The code C for X is a mapping from A (the alphabet for X) to D^* (the set of finite strings from a D -ary alphabet). For each $x \in A$, let $C(x)$ be the codeword corresponding to x and $l(x)$ be the length of $C(x)$. The expected length $L(C)$ of the code C is then given by:

$$L(C) = \sum_{x \in A} p(x) l(x)$$

When choosing a code for a message, there are several desirable criteria. Clearly, each codeword should be distinct to allow for unique decoding. However, by itself, this is not enough; when decoding the encoded string $C(S)$, there may be multiple ways to choose where one codeword ends and the next begins, and thus multiple decodings may be possible. Codes for which this does not occur and the decoding is always unique are called uniquely decodable.

When decoding a uniquely decodable code, it may be the case that one may have to look past a particular codeword to decode it (indeed, the entire encoded string might have to be examined to decode just the first codeword). A very useful class of codes is the class for which this never occurs; the encoded string $C(S)$ can be decoded symbol by

symbol as soon as the end of each codeword is reached, and no backtracking is ever necessary. These are the prefix (or instantaneous) codes; they can be defined by the property that no codeword is a prefix of any other codeword. (An obvious but inefficient way to create a prefix code is to reserve one symbol as a punctuating mark, so that codewords are always separated by the punctuating symbol.)

Given X and $p(x)$, we may wish to make a prefix code. The limitation on codeword lengths is given by the Kraft inequality:

THEOREM: For any prefix code over an alphabet of size D , the codeword lengths $l_1, l_2, \dots, l_m$ satisfy the Kraft inequality:

$$\sum_{i=1}^m D^{-l_i} \leq 1$$

Conversely, given any $l_1, l_2, \dots, l_m$ satisfying the Kraft inequality, a prefix code with these lengths for the codewords can be constructed.

PROOF: Let the D -ary alphabet be $\{0, 1, \dots, D-1\}$. Representing codeword i of length $l = l_i$ as $y_1 y_2 \dots y_l$, we can consider the real number corresponding to it:

$$0.y_1 y_2 \dots y_l = \sum_{j=1}^l y_j D^{-j}$$

Associate with this codeword the interval

$$[0.y_1y_2\dots y_i, 0.y_1y_2\dots y_i + 1/D^i)$$

i.e. the set of all real numbers whose D -ary expansion begins with $0.y_1y_2\dots y_i$.

Thus, each codeword has corresponding to it a subinterval of the unit interval $[0,1)$. By the prefix condition, all of these subintervals are disjoint; hence the sum of the lengths of the intervals is at most 1. Since the length of the subinterval corresponding to codeword i is the same as term i of the sum, the first part of the theorem is proved.

To get the converse, first reorder the indexing of the lengths so that $l_1 \leq l_2 \leq \dots \leq l_m$. Starting from the left end of $[0,1)$, choose disjoint adjacent subintervals of length D^{-k} for $k = 1, 2, \dots, l_m$. By construction, the right end of interval j will be

$$\sum_{i=1}^j D^{-l_i}$$

for $1 \leq j \leq m$. Since we are given that the lengths satisfy the Kraft inequality, it will be possible to choose adjacent subintervals corresponding to all lengths; we then assign codeword i to be the D -ary representation of the left end of interval i , and the proof is complete. $\odot$

One might expect that other uniquely decodable codes, which are not prefix codes, might give better compression results than prefix codes in some cases. In fact, the Kraft

inequality above also holds for uniquely decodable codes; thus, for every uniquely decodable code, a prefix code can be constructed which has the same codeword lengths. For this reason, we can restrict our attention to the class of prefix codes.

Optimal Coding and Entropy

Consider now the problem of finding a prefix code C for a distribution X with minimal expected length $L(C)$. A lower bound for $L(C)$ is given by the following theorem:

THEOREM: The expected length $L(C)$ of a D -ary prefix code for X is bounded below by the entropy $H_D(X)$.

PROOF: We first define r_i and c as follows, and then show the difference $L(C) - H_D(X)$ is nonnegative:

$$\begin{aligned}
 c &= \sum_i D^{-l_i} & r_i &= \frac{D^{-l_i}}{c} \\
 L(C) - H_D(X) &= \sum_i p_i l_i + \sum_i p_i \log_D p_i \\
 &= -\sum_i p_i \log_D D^{-l_i} + \sum_i p_i \log_D p_i \\
 &= \sum_i (p_i \log_D p_i - p_i \log_D c r_i) \\
 &= -\log_D c + \sum_i p_i \log_D \frac{p_i}{r_i} \\
 &= \log_D \frac{1}{c} + D(p||r) \\
 &\geq 0
 \end{aligned}$$

or the limit superior as n approaches infinity of $C(S_n|n) / n$ (which always has a value between 0 and 1). Infinite strings need not have infinite C , e.g. (0000...) or the digits of π which can be generated by finite programs. Note that, seemingly paradoxically, a finite length string can have greater C than an infinite length string! For example, consider 0^n , the string consisting of n 0's. Clearly $C(0^n)$ is roughly $C(n)$, which can take on arbitrarily large values depending on n ; however, an infinite string of 0's can be generated by a small program. Note that although $C(0^n)$ is approximately $C(n)$, $C(0^n|n)$ is a small constant; here is a simple example of how conditional Kolmogorov Complexity can be used to "remove" the descriptive complexity of some feature of a string.

One problem with using Kolmogorov Complexity as a measure for "intuitive complexity" is that of randomness. While a homogeneous string has minimal C , a random string will have maximal C ; however, pure randomness certainly does not accord with the intuitive notion of a complicated structure. C can thus be viewed as measuring "algorithmic randomness".

We can therefore say that low C indicates the absence of complexity, but medium to high C does not necessarily indicate its presence. Very high C indicates almost complete algorithmic randomness, i.e. the absence of "intuitive complexity"; in fact, having high C is an excellent definition of randomness (but not perfect, as we shall see later). We will later explore several complexity measures which get around this problem of assigning high complexity to a random sequence.

sequence of random variables, and stationarity implies that the joint distribution of any subset of variables is invariant with respect to shifts in the index.) The entropy $H(P)$ of the stationary stochastic process P can be defined as:

$$H(P) = \lim_{n \rightarrow \infty} \frac{H(X_1, X_2, \dots, X_n)}{n}$$

Then, using the increasing block length code defined previously, $L(C)$ approaches $H(P)$. Note that $H(P)$ can thus be thought of as the expected description length of an outcome from the process P .

An optimal prefix code (called a Huffman code) can be found using the following remarkably simple technique:

- Assign to each probability $p_1, \dots, p_n$ a node with a value equal to the probability; this will be the lowest level (level $k+1$) of a tree. (We will be reducing the number of nodes by $D-1$ at each stage, so if the number of probabilities is not of the form $1 + k(D-1)$ then add sufficient dummy probabilities with probability 0 to achieve this form.)
- Take the D nodes with the smallest values in level $k+1$, and join them by edges to a parent node at the next higher level of the tree (level k). Give this parent node a value equal to the sum of the values of its D child nodes. Join each of the

other nodes at level $k+1$ to a parent node at level k with a value equal to its child node. Move to level k and repeat. After a total of k steps, the tree will be completely constructed.

- Now backtrack, moving down the tree from the root node at level 1. First assign each of the D available symbols to one of the D nodes at level 2. Then, moving down level by level, consider each node which has D child nodes; assign to each of these child nodes the code of its parent plus one of the D symbols. (Nodes with only one child node simply copy their code to their child node.) After k steps, the nodes at level $k+1$ will each have codes assigned to them; this is the Huffman code.

Note the similarity to the construction in the proof of the Kraft inequality.

Universal Codes

The codes discussed thus far require the probability distribution of the symbols to be known ahead of time. In a more general situation, we would like to be able to find coding schemes that asymptotically approach the maximal efficiency, for a fixed (but unknown) probability distribution. Such a coding scheme, in which the average length per symbol of the encoding is asymptotically equal to the entropy rate of the source for any stationary source, is called a universal code.

A simple example of a universal code (which is proved to be universal in [Cover]) is the widely-used Lempel-Ziv algorithm:

- Starting from the current location in the source sequence, read in the minimal size string S of length n that has not appeared thus far.
- Denoting the last symbol of S by s , encode S with the concatenation of (the location of its maximal proper prefix S' in the current ordered list of prefixes) and (the value of s).
- Update the ordered list of prefixes by storing the encoding of S in it, and repeat the process until the end of the source sequence is reached.

For example, the sequence 101101011... would be separated into 1,0,11,01,011,... and encoded as (0,1), (0,0), (1,1), (10,1), (100,1), and so on. The first number in each pair indicates the position of the maximal prefix, in the list of prefixes that has been formed up to that point. Note that later code words will be longer than earlier ones.

Universal codes are extremely important in theory and in practice. When presented with a large amount of data to be compressed, the "true probability distribution" will often be unknown, due to i) lack of knowledge of the data generation machinery, and ii) lack of understanding of how the data generation machinery works (i.e. what types of sequences it will typically generate).

Algorithms that "adapt to the data" are thus essential; a universal code is a first step toward such adaptive algorithms, one that uses a fixed method based only on observed frequency regularities to build a growing model of patterns in the data. We will expand on this idea later.

Two important extensions of information theory deal with finding codes for nonstationary distributions and finding codes that will work under noisy conditions. How does one recognize when the nature of a source sequence has changed, and how does one separate data from noise? These are fundamental questions applicable to all theories of pattern recognition.

Information theory is a well-established field with proven uses in many areas, and as we have seen the theory provides insights into more general ideas relating to complexity. However, it is important to note that the entire theory deals with the compression and abstraction possible using only the probability distribution of symbols in the data. It is as if one were only allowed to describe or listen to Beethoven by using a table containing the frequency of occurrence of each note; only a very limited form of description or understanding would be possible.

Let us, then, move on to a theory which combines ideas from information theory and computational complexity. Instead of only dealing with fixed probability distributions, our encodings will be done using the powerful algorithmic formalization of the TM.

KOLMOGOROV COMPLEXITY

Basics

Any function computed by a TM has many TM representations, corresponding to different algorithms for computing the same function. Correspondingly, any object (i.e. output string) can be produced as output by many different TMs on differing inputs. The shortest such TM-input pair (in the sense of description length) that produces the desired output gives us one measure of how complicated the function is.

This shortest description length is called Kolmogorov Complexity ("C"); see [Li and Vitanyi] for a comprehensive introduction. Note that the description length includes the length of the input (the parameters and initial data) as well as the length of the program (the algorithmic process itself); by the previous discussion on UTMs, both of these can be combined into one input to a UTM.

(It is necessary to take into account both lengths, as the aggregate of program and input data is what describes the output string. Otherwise, for example, the shortest possible copying program copying any input would yield the same string as output, and all strings (except those few producible as the output of TMs shorter than the minimal copying program) would be defined to have the same small value of C--not a very useful complexity measure!)

Formally, for a partial recursive function f , we can define the complexity:

$$C_f(x|y) = \min\{l(p) : f(\langle p, y \rangle) = x\}$$

where $l(p)$ is the length of p , and $\langle p, y \rangle$ denotes a bijective encoding function mapping the pair (p, y) to some natural number. (If no p exists such that $f(\langle p, y \rangle) = x$, $C(x|y)$ is defined to be infinity.)

If f is the function computed by a UTM, the above defines the conditional Kolmogorov Complexity of x given y . The Kolmogorov Complexity of a single string x is just the conditional Kolmogorov Complexity of x given the empty string. Note that the Kolmogorov Complexity will be finite as long as x and y are themselves computable. (Infinite Kolmogorov Complexity is equivalent to uncomputability.)

Since every UTM U can simulate (using space c_U) a constant size print program that simply copies the input to the output, an upper bound for $C_U(x)$ is $l(x) + c_U$.

The fact that a UTM can simulate any other TM (including another UTM) implies that C is an invariant measure across description methods, up to an additive constant. More precisely, given two UTMs U_a and U_b , U_a can simulate U_b when given its encoding of fixed length c_{ab} ; similarly, U_b can simulate U_a given an encoding of length c_{ba} . Thus, if any string S has a minimal description using U_a of length l , S can also be described by U_b using space at most $l + c_{ba}$, and vice versa.

So translating between UTMs for C adds only a fixed constant depending on the UTMs. For short descriptions, the constant will be an important factor; however, as description length increases, the contribution of the constant term for translation between UTMs will go to 0. We can thus fix a single UTM U' to use, and write $C(x)$ for $C_{U'}(x)$; the results we get will then all be true up to an additive constant which depends only on the translation constants between UTMs.

Note that every S has a UTM U for which $C_U(S) = 0$, but this conundrum is easily solved. If $C_{U'}(S)$ is high, the fact that $C_U(S) = 0$ implies that $C_{U'}(\text{PROG}(U))$ must itself be as complex relative to U' as S is (where $\text{PROG}(U)$ is the program listing for U). Thus if S is assigned very low C by one UTM and high C by another, the first UTM must contain a lot of information about S in its definition; obviously a UTM tailor-made to produce S will assign it low complexity.

C is thus an intrinsic property of each object. With any general description method, functions with low values of C can be described succinctly, and functions with high values of C cannot. (Note that the area of applicability of C is that set of functions which can be described by Turing Machines; Church's Thesis is thus implicitly being used here, and the theory makes no claim about the properties of any object without a Turing Machine representation.)

Strings with infinite C can be differentiated by looking at the "average C per bit",

or the limit superior as n approaches infinity of $C(S_n|n) / n$ (which always has a value between 0 and 1). Infinite strings need not have infinite C , e.g. (0000...) or the digits of π which can be generated by finite programs. Note that, seemingly paradoxically, a finite length string can have greater C than an infinite length string! For example, consider 0^n , the string consisting of n 0's. Clearly $C(0^n)$ is roughly $C(n)$, which can take on arbitrarily large values depending on n ; however, an infinite string of 0's can be generated by a small program. Note that although $C(0^n)$ is approximately $C(n)$, $C(0^n|n)$ is a small constant; here is a simple example of how conditional Kolmogorov Complexity can be used to "remove" the descriptive complexity of some feature of a string.

One problem with using Kolmogorov Complexity as a measure for "intuitive complexity" is that of randomness. While a homogeneous string has minimal C , a random string will have maximal C ; however, pure randomness certainly does not accord with the intuitive notion of a complicated structure. C can thus be viewed as measuring "algorithmic randomness".

We can therefore say that low C indicates the absence of complexity, but medium to high C does not necessarily indicate its presence. Very high C indicates almost complete algorithmic randomness, i.e. the absence of "intuitive complexity"; in fact, having high C is an excellent definition of randomness (but not perfect, as we shall see later). We will later explore several complexity measures which get around this problem of assigning high complexity to a random sequence.

Since every string S can be produced by a simple copying machine which copies its input to its output, we have the upper bound $C(S) < l(S) + c$ for some small positive integer c . In fact, it is the case that most strings have C very close to the maximal value, a fact formalized in the following Incompressibility Theorem:

THEOREM: For each fixed T and positive integer c , every finite set A of size m has at least $m(1-2^{-c})+1$ members S with $C(S|T) \geq \log m - c$.

PROOF: Simply count the short descriptions: the number of binary programs (program-data inputs to a UTM) of length less than $\log m - c$ is $2^{\log m - c} - 1 = m2^{-c} - 1$.

The number of members of A described by these programs is at most the number of programs. Therefore, at least $m - m2^{-c} + 1$ members of A have binary programs of length at least $\log m - c$. ☺

If A is the set of strings of length n , its cardinality is 2^n , and a fraction of at least $1-2^{-k}$ of these strings have $C \geq n-k$; most strings are almost incompressible (or equivalently random).

Considering the binary string x as corresponding to the natural number it represents, how does the complexity function $C(x)$ behave? By the Incompressibility Theorem, $C(x)$ is close to $\log x$ for most values of x . It is also true that the greatest monotonically increasing function bounding C from below is unbounded, but increases

2

PM-1 3½"x4" PHOTOGRAPHIC MICROCOPY TARGET
NBS 1010a ANSI/ISO #2 EQUIVALENT

PRECISIONSM RESOLUTION TARGETS

slower than any computable function. (We can see this informally by noting that $C(BB(x)) = C(x) + O(1)$; since $BB(x)$ grows faster than any computable function, the result follows.)

Is C itself a computable function? No, as the following theorem shows:

THEOREM: $C(x)$ is not partial recursive. Worse yet, no partial recursive function $f(x)$ exists which is defined on an infinite domain and equal to $C(x)$ over its whole domain.

PROOF: Suppose by way of contradiction that such a partial recursive $f(x)$ exists. Its domain is (by definition) an infinite recursively enumerable set; every such set contains an infinite recursive subset (as mentioned in the section on Classical Computability).

Take one such subset and call it A . Define a new function $MIN(m)$, such that:

$$MIN(m) = \min\{x : C(x) \geq m, x \in A\}$$

(Recall that we fix a particular UTM that we use to measure C ; the absence of a subscript indicates that the complexity is being taken relative to this reference UTM.) The function MIN is total recursive, since $C(x) = f(x)$ on A ; it takes on arbitrarily large values, by the monotonically increasing greatest lower bound on C mentioned previously. By definition of MIN , $C(MIN(m)) \geq m$.

However, since our reference UTM can simulate any other TM with a constant

size program, $C(\text{MIN}(m)) \leq C_{\text{MIN}}(\text{MIN}(m)) + C(\text{PROG}(\text{MIN}))$. Note that $C_{\text{MIN}}(x)$ denotes the minimal size input to the function MIN that will return x ; although the function MIN may not be equivalent to a UTM, we will only use strings x for which $C_{\text{MIN}}(x)$ is defined. $\text{PROG}(\text{MIN})$ is the program listing for the function MIN.

Now $C_{\text{MIN}}(\text{MIN}(m)) \leq l(m) = \log m + 1$, since giving m as input to min will return $\text{MIN}(m)$. (The length of m is an upper bound for the description length of $\text{MIN}(m)$ with respect to the function MIN.) Combining the two inequalities for $C(\text{MIN}(m))$ gives $m \leq \log m + O(1)$, which is false for large enough m . $\odot$

If C can't be computed exactly, is it possible to approximate C ? A straightforward approach is to run all programs of length at most $l(x) + c$ (the maximum possible length of a shortest program for x , with c a UTM-dependent constant) in tandem, and define $f(t, x)$ as the shortest such program that has halted with output x after t steps. This is clearly computable, but is of questionable practical use.

Proofs Using Kolmogorov Complexity

To make the notions presented thus far more intuitive, here are two Kolmogorov Complexity proofs of historic mathematical theorems, adapted from [Balcazar, Diaz and Gabarro 90]:

THEOREM: There are an infinite number of primes.

PROOF: Assume by way of contradiction that only k primes exist. Since each natural number n is uniquely defined by its prime factorization, we can represent n by this list of k primes along with k exponents. The list of primes is fixed and of constant size, and each of the exponents is at most $\lg n$ and can be stored in $\lg \lg n$ bits.

A program of fixed size (including the fixed size list of primes), along with the exponents of total size at most $k \lg \lg n$, is therefore sufficient to generate n . But by the Incompressibility Theorem, many integers can't be compressed to significantly less than n bits for large enough n . The contradiction establishes the theorem. ☺

THEOREM (Godel's Incompleteness Theorem): A consistent formal system F which produces a recursively enumerable set of theorems and contains the concepts of Universal Turing Machines and computations, such as one containing Peano Arithmetic, is incomplete; F contains statements which are true but can't be proved within it.

PROOF: Consider the program (constructed inside F) which enumerates theorems of F ; such a program exists since the theorems of F are recursively enumerable. Pick a constant c much larger than the size of this program. Many statements (expressed as words in the alphabet of F) have Kolmogorov Complexity higher than c , by the Incompressibility Theorem; this fact cannot be proved by F for any such statement.

To see this, assume that F implies " $C(s) > c$ " for some statements s , and consider the s with the lexicographically smallest proof. The following program P will generate s : enumerate the proofs of theorems of F , and output the first theorem which proves " $C(s') > c$ " for some s' .

If P halts, then for sufficiently large c P is shorter than c and " $C(s') > c$ " is not true since $C(s') \leq |P|$. This would imply the inconsistency of F in contradiction to the assumption of its consistency; hence P cannot halt. But then the assumption that F implies " $C(s) > c$ " for some s is untrue, even though there are such statements s ; F is therefore incomplete. ☹

Notice that the last proof uses the Berry paradox: does "the smallest positive integer specified in more words than this phrase in quotes contains" exist? The proof also gives a somewhat constructive insight to the meaning of Godel's Theorem, by showing that one class of "true but unprovable statements" is the class containing objects for which no sufficiently succinct description exists.

Prefix Complexity

We now define a new variant of Kolmogorov Complexity: prefix complexity, which we will denote by $KC(x)$. Here we restrict the set of shortest program-data combinations to be one which is prefix-free; the minimum description length will be taken with respect to prefix TMs whose domain is prefix-free.

A prefix TM can be viewed as one which has a one-way input tape, a one-way output tape, and a two-way work tape. At each step, the TM can read the next input symbol, write the next output symbol, or perform a computation step on the work tape. Clearly, such a TM only computes prefix functions; if the TM halts on input x , it will also halt on input xy for any y (since the machine reads its input sequentially and halts after reading the last symbol of x).

Prefix Kolmogorov Complexity KC is then measured with respect to some fixed prefix UTM U . The machine U is itself a prefix TM which can simulate all other prefix TMs. (Note that whenever prefix complexity KC is discussed, the UTM is always a prefix UTM.)

The encoding of program-data combinations into a prefix-free set can be any prefix code; a normal UTM can be made into a prefix UTM by including a constant-length subroutine which decodes the input before processing it (and halts if the input is

not a valid prefix-encoding). As mentioned earlier in the section on Information Theory, one simple prefix code can be defined by reserving one symbol as a punctuation mark. Extending this idea, a series of asymptotically more and more efficient prefix-codes can be defined:

$$E_0(x) = 1^*0$$

$$E_i(x) = E_{i-1}(l(x))x \text{ (for } i > 0)$$

So $E_1(x) = 1^{l(x)}0x$, and has length $2l(x) + 1$; $E_2(x) = E_1(l(x))x$ with length $l(x) + 2l(l(x)) + 1$, and so on. (The first code is called self-delimiting, since it explicitly gives the length of each codeword.)

There are several problems with C , most of which are fixed by the new measure KC :

- C is not subadditive; the best bound obtainable for $C(xy)$ is:

$$C(xy) \leq C(x) + C(y) + O(\log(\min(C(x), C(y))))$$

since it is necessary to specify a way to tell where x ends and y begins. In contrast, we have $KC(xy) \leq KC(x) + KC(y) + O(1)$.

- It would be desirable to have $C(xy) \geq C(x)$; this is not the case. Thus, appending to a string can make the string shorter to describe. (Consider for example $C(0^m)$ and $C(0^{m+k})$, with $C(m+k) \ll C(m)$.) Unfortunately, this is the case for KC as well.

- The notion of a universal prior probability is useful for constructing universal induction algorithms, and for guessing the probability that a given string will occur in "nature". However, the complexity measure C doesn't allow such a prior probability (as will be seen in the next section).

Have we lost anything in going from C to KC ? Not much, as the following theorem shows.

THEOREM: KC and C are asymptotically equivalent; within an additive constant it is true that:

$$C(x|y) \leq KC(x|y) \leq C(x|y) + 2 \log(C(x|y))$$

PROOF: The first inequality is clear since every prefix UTM is also a general UTM.

For the second inequality, recall that if the reference machine used for C computes x from $\langle y, p \rangle$, $C(x|y) = l(p)$. We can therefore prepend a self-delimiting encoding of $l(p)$ to p in order to get an appropriate prefix machine. Using the prefix code introduced earlier, we see that $E_1(l(p))$ has length $2l(l(p)) + 1$; since $l(k) < \log(k) + 2$ and $l(p)$ is equal by definition to $C(x|y)$, the theorem is proved. (A sharper upper bound can be obtained by using more efficient prefix codes instead of E_1 .)

Universal Measures and Induction

As foreshadowed in the previous section, Kolmogorov Complexity can be used to define a type of "universal prior probability".

When flipping a coin, why would a sequence of 1000 consecutive tails be considered unlikely to have occurred purely by chance? A random sequence S of coin flips of length n has a probability of 2^{-n} . However, if we consider S as arising from a random algorithmic process (i.e. as the output of a UTM whose input is provided by a sequence of fair coin flips) then the probability of S is bounded below by $2^{-KC(S)}$. For sequences of low Kolmogorov Complexity, therefore, generation by an algorithmic process will be much more likely. (Note that the Incompressibility Theorem tells us that there are relatively few sequences of this type.)

Assuming that programs are generated by random coin flips, one might try to define the probability of occurrence of the string S as:

$$P(S) = \sum_{p: U(p)=S} 2^{-l(p)}$$

where we sum over all programs p for which U computes S . This doesn't work; since every S has a TM which outputs S on all inputs, $P(S)$ is infinite for all S .

However, if we restrict the UTM U to be a prefix UTM, then the sum of $2^{-l(p)}$ over

all p converges to at most 1, by the Kraft Inequality. Thus we have a notion of "universal a priori probability", usually called algorithmic probability; the probability of occurrence of a string S is the probability that S occurs as output when a prefix program generated by coin flips is run.

Note that if we consider only the shortest (not necessarily prefix) program instead of all programs, we have $P(S) = 2^{-C(S)}$; this still diverges when summed over S , since $C(S)$ is close to $\log S$ for almost all S and thus $P(S)$ is close to $1/S$. But replacing $C(S)$ by $KC(S)$ does give a convergent series, by the Kraft Inequality which applies since we're encoding the integers by a prefix-free set of programs (or codewords). The quantity $2^{-KC(S)}$ is a lower bound on the algorithmic probability given in the previous paragraph (since this quantity is one term of the sum which defines the algorithmic probability).

Algorithmic probability is the probability for a particular sequence to arise from a random algorithmic process; picture a monkey programming at random on a keyboard connected to a TM compiler. It provides a "universal prior" that can be used in Inductive and PAC learning, as well as for generating plausible finite automata to describe processes.

Ω --The Ultimate Oracle

We'll look now at the fascinating number Ω , first proposed by Gregory Chaitin (who discusses it at length in [Chaitin 87 and 92]. It is defined by:

$$\Omega = \sum_{p: U(p) \text{ halts}} 2^{-l(p)}$$

where U is our fixed prefix UTM. Note that the quantity is well defined (in particular, the sum does not diverge) since the set of programs forms a prefix code; by the Kraft Inequality, the sum is thus at most 1.

Ω is a weighted sum of the prefix representation of all halting programs; it is also the probability that U halts if its program is given by a sequence of fair coin flips. Naturally enough, it is sometimes called the "halting probability".

THEOREM: Given the first n bits of Ω , we can solve the Halting Problem for any program q of length at most n .

PROOF: Letting Ω_n denote the first n bits of Ω , we have that:

$$\Omega_n \leq \Omega < \Omega_n + 2^{-n}$$

With this fact in mind, we generate an approximation to Ω_n --call it Ω_n' , and assign to it the value 0 to begin with. We then start U running on all programs, from shorter ones

to longer ones. (This can be done by executing the first step of the first program, then the first step of the second program and the second step of the first program, and so on; this is called "dovetailing".)

Every time a program p halts, we add $2^{l(p)}$ to Ω_n' . Eventually, Ω_n' will exceed Ω_n (this is so since $\Omega_n < \Omega$, and our sequence of successive values for Ω_n' is monotonically increasing with least upper bound Ω). At this stage, if q has not already halted, it never will. To see this, note that if it did halt it would add a factor of at least 2^n to Ω' , which would make Ω' greater than Ω --a contradiction. $\odot$

(By storing $2^{n+1}-1$ bits, this process actually solves the halting problem for all programs of at most n bits. These stored bits (which form an initial segment of the so-called Halting Set) are another representation of a solution to the Halting Problem, requiring exponentially more bits than Ω but giving a solution to any particular program immediately.)

Ω is also an incompressible real number--it has maximum Kolmogorov Complexity.

THEOREM: There exists a constant c for which $KC(\Omega_n) > n-c$ for all n .

PROOF: From the previous theorem, we can use Ω_n to generate all programs of length at most n for which U halts. For every x which is not an output of any of these halting

programs, it must be the case that $KC(x) > n$.

We can therefore define a program HC for which $HC(\Omega_n) = x_n$, with $KC(x_n) > n$. (HC could, for example, output the least x as defined in the previous paragraph; since (given Ω_n) the procedure for doing this as outlined above is computable, HC can be expressed as a program.)

Since HC itself is a program of constant length c , we can generate x_n by a program of length $KC(\Omega_n) + c$, which implies $KC(\Omega_n) + c \geq KC(x_n) > n$; subtracting c from the leftmost and rightmost term, the proof is complete. ☺

The ability to decide halting problems implies that Ω_n can be used to decide the truth of any finitely refutable or provable theorem that can be expressed in at most n bits. For example, given a TM that looks for counterexamples to conjectures such as the Goldbach conjecture ("every even number is the sum of two primes"), the truth or falsity of the conjecture can be determined as long as the TM is of length at most n bits.

Unfortunately, since $KC(\Omega)$ is infinite, Ω is uncomputable--not surprising, for such a seemingly omnipotent number. And even if we did know Ω_n , the process described above for deciding the Halting Problem for programs of length at most n is essentially the fastest possible. The time taken by this process will be at least as great as that for running the slowest n -bit program--a Busy Beaver function which grows faster than any computable function.

As a seeming puzzle, note that the Busy Beaver function is also incompressible and thus has maximal Kolmogorov Complexity. This is true almost by definition of the function; if a Busy Beaver program was compressible, instructions could be added to the compressed program to give a program which executes for a longer time than the original Busy Beaver program--a contradiction. So the programs for calculating both $BB(n)$ and Ω_n have maximal Kolmogorov Complexity, which we have equated with "randomness". But in what sense can programs which calculate such profound quantities be said to be random?

A partial answer comes from realizing that we have already seen such random programs that compute important functions. In fact, every minimal program is itself almost incompressible--otherwise we could run the compressed version of the minimal program to get the minimal program as output, and then feed this output back into the UTM. (This will involve some fixed overhead which will become insignificant with increasing program size.)

Thus, every function's corresponding shortest program is random in the sense of having no regularities or compressibilities. Contrasting such programs with the programs we usually write to calculate functions suggests that minimal programs may as hard to analyze and construct as Busy Beaver programs.

Resource-Bounded Kolmogorov Complexity

One obvious problem with applying Kolmogorov Complexity to realizable systems is that the minimal machine used to measure complexity has no restrictions on usage of space or time. Adding restrictions in either or both of these leads to the notion of resource-bounded Kolmogorov Complexity as discussed in [Li and Vitanyi].

We may, for example, wish to find the shortest possible Turing Machine that will compute a given function in time at most T , and/or space at most S . This is intimately related to Computational Complexity theory, which deals with finding such space or time bounds for language recognition and search problems. (See the previous section on Computational Complexity for details.) In the first case, we minimize the description length of the algorithm while holding maximal time and space constant; in the second case, we minimize time (and sometimes space) while placing no constraints on the description length of the algorithm.

Although invariance theorems for space and time bounded Kolmogorov Complexity hold, they are weaker and hold only up to a logarithmic constant. As one would expect, they are also uncomputable.

It turns out that the machinery of minimal length TMs can be used to provide alternative and qualitatively different characterizations for many properties in

Computational Complexity and Structural Complexity. One interesting notion that connects all these areas is that of instance complexity. Instead of dealing with the worst-case complexity for an entire class of problems (as with P and NP), particular instances of the problem can be evaluated for their difficulty. One can then talk about the hard instances of a language.

More formally, let the t -bounded instance complexity of the string x with respect to the set A be denoted by $ic^t(x:A)$. It represents the length of the shortest program which decides whether x is in A , in time $t(l(x))$. For all strings y other than x , the program can give a "don't know" answer (but if the program gives a definite answer for the membership of y in A , the answer must be correct and given in time $t(l(y))$). If no such program exists, the instance complexity is infinite.

As with time-bounded Kolmogorov Complexity, an invariance theorem between TMs holds; there is a UTM such that:

$$ic^{t'}(x:A) \leq ic^t(x:A) + c$$

where $t' = ct(n) \log t(n) + c$ is the time bound on the UTM, t is the time bound on any other given TM, and c is a constant depending only on the two TMs. Thus instance complexity can be taken with respect to this UTM, and is invariant up to the above factors.

THEOREM: A is in P iff a polynomial P and constant c exist such that $ic'(x:A) \leq c$ for all x .

PROOF: ($\Rightarrow$) If A is in P , then by definition of P there is a polynomial time program deciding membership in the set A . This program is of constant size and decides whether or not x is in A for all x .

($\Leftarrow$) By assumption, for all strings x there is a program of size at most c which decides in polynomial time whether x is in A . Since the number of such programs is bounded, the program A -DECIDE which simulates all of these programs is therefore itself of constant size and runs in polynomial time. A -DECIDE can decide if x is in A by running all its constituent programs and accepting x iff one of these programs accepts x . Thus the set A is in P . ◻

The notion of instance complexity can also be used to give nice characterizations of polynomial complexity cores and self-reducibility, as well as a simple criterion for the equivalence of NP and P , as discussed in [Ko, Orponen, Schoning and Watanabe].

Here are two other interesting ideas involving resource-bounded Kolmogorov Complexity, both of which are related to cryptography:

POTENTIAL: It would be nice to be able to define some notion of "mathematical potential" which corresponds to the idea of potential in chemistry. A mathematical object

with "low potential energy" could be transformed to one with high potential energy only with the input of large amounts of energy (i.e. computation). Such a notion which is applicable to the problem of integer factorization can be defined using a variant of Kolmogorov Complexity.

The age of a string is roughly the time it would take for a UTM to generate the string, when simulating all programs lexicographically in a dovetailing manner. The logarithm of the age gives the K_t complexity of a string. Formally, for the first p such that $U(p) = x$ after t steps when simulating all programs in order of increasing length, $K_t(x) = \min\{l(p) + \log t\}$.

Using K_t complexity, we say that a string x is k -potent relative to y if k is the least positive integer for which $K_t(x|y) \leq k \log l(x)$. (If y is the empty string, x is simply said to be k -potent.) The lower k is, the greater the probability for a k -potent string to "arise quickly".

A function f mapping strings to strings is called inflating iff for all integers k , $f(x)$ is not k -potent relative to x for infinitely many x . (Roughly speaking, f is inflating iff the expected time to generate $f(x)$ from x using the dovetailing simulation is superexponential in the length of x for infinitely many x .)

The following remarkable result (which is a corollary of a more general theorem)

can now be stated: The integer factoring function f is in P iff f is not inflating. Regarding a low value for k -potency as "high potential energy", f is in P iff the factors of a number infinitely often have high potential relative to their product.

The membership of f in P is of tremendous practical significance; many public-key cryptographic systems (such as the RSA system mentioned earlier) would be breakable if f was indeed in P .

P-RANDOMNESS AND CRYPTOGRAPHY: Just as high Kolmogorov Complexity corresponds to "absolute randomness", high time or space bounded Kolmogorov Complexity corresponds to "practical randomness".

If we define Pspace-pseudorandom sequences as those that appear random to all tests for nonrandomness that use polynomial space, then high space-bounded Kolmogorov Complexity is sufficient for a sequence to be Pspace-pseudorandom. (Such a sequence must also be Ptime-pseudorandom, since all tests using polynomial time cannot use more than polynomial space.)

The effective generation of pseudorandom sequences would have strong implications for the simulation of randomness and for cryptography. We can phrase the notion of effective generation as a question of the existence of pseudorandom generators; these generators are deterministic programs that expand a randomly selected seed into a

much longer pseudorandom sequence.

The first construction of a valid pseudorandom generator was made under the assumption that exponentiation modulo a prime is a one-way function (as defined earlier in the section on Cryptography and Zero-Knowledge Proofs). (If this assumption does not hold, then the RSA cryptosystem would have a polynomial-time algorithm.) A much more general result has recently been proved: one-way functions exist iff pseudorandom generators do. (For an accessible discussion, see [Goldreich, Krawczyk and Luby].)

After reading so much about the theory of Kolmogorov Complexity, one might very well wish to tie the theory in with the broader notion of the best resource unbounded or (more reasonably) bounded KC machine that can be found with given space/time computational resources. In other words, how can we actually find low-KC TMs in practice?

This is of obvious interest, as it would allow us to both minimize computational resources used as well as learn abstract properties of the problem from its minimal algorithm.

Unfortunately, finding low-KC TMs is almost the canonical problem of learning: to be able to find the most compact description of a process. As such, the generality of methods for reducing KC will probably vary inversely with their effectiveness.

Note that even defining "the minimal KC machine for function F that can be found

with given computational resource bounds" is difficult. We would need to take into account:

- the time and space available to the algorithm
- the particular function F
- the particular algorithm used; obviously algorithms for lowering KC will vary greatly in effectiveness; how is the difficulty of finding these algorithms or the "quality of prior knowledge" embodied in each algorithm to be measured in a few parameters?

This last requirement is really the key one. Any sort of reasonable measure should depend on only a few parameters; parametrizing "the effectiveness of a learning algorithm" is probably arbitrarily difficult in a general universe. Many of the problems involved can be found in the literature on finding minimal finite state machines for a process, and on algorithmic learning in general as surveyed in [Hutchinson].

Connections between Information Theory and Kolmogorov Complexity

Note that classical information theory uses only the information contained in $p(x)$ (the probability mass function of X); it gives the average description length for a sequence drawn from $p(x)$. Contrast this with Kolmogorov Complexity, which uses all possible information about a particular sequence x to define an optimal compressed description. The first gives the average maximal compression possible in choosing an object from an ensemble of other objects, given their relative frequencies; the second gives the maximal compression for a specific object, and is thus an intrinsic property of the object.

For example, encoding the first 2^n digits in the binary expansion of π would take about 2^n bits, using any optimal prefix code. (This assumes that π is a normal number, i.e. one in which every finite sequence of bits occurs with the same probability.) However, the same digits could be compressed into $(KC(n) + \text{a constant})$ bits, by using a constant size program to expand π and giving it a compressed description of n as input.

Intuitively, one might guess that the entropy is an upper bound for the complexity of a string, since we should be able to incorporate any regularities (including the frequency regularities measured by entropy) into our minimal program. This is in fact the case, and we can formalize this intuition with the following theorem:

THEOREM: Let $x = y_1 y_2 \dots y_m$ be a finite binary string, where each y_i represents a binary substring of length r . Denoting 2^r by R , let the frequency of binary subsequence k ($k = 1..R$) of length r be denoted by $p_k = |\{i : y_i = k\}| / m$.

Then, up to the additive constant of the reference UTM,

$$C(x) \leq m(H + e(m)) \quad (H = -\sum_{k=1}^R p_k \log p_k)$$

with $e(m) = 2Rl(m)/m$. (Note that $e(m)$ approaches 0 as m approaches infinity, for fixed r .)

PROOF: In order to construct x , it is sufficient to be given the number of occurrences $s_k = p_k m$ of the binary subsequence k in x ($k = 1..R$), along with the ranking j of x in the ordered set of all strings having the given values of s_k . Thus x can be reconstructed from $s_1, \dots, s_R, j$.

Using the self-delimiting code E_1 mentioned in the section on prefix complexity, we therefore have $C(x) \leq 2l(s_1) + \dots + 2l(s_R) + l(j)$. Note that since each $s_k \leq m$, we can bound j and $C(x)$ as follows:

$$j \leq \binom{m}{s_1, \dots, s_R}$$

$$C(x) \leq 2^{r+1} l(m) + l\left(\binom{m}{s_1, \dots, s_R}\right)$$

Using Stirling's approximation on the multinomial coefficient along with the definition of entropy proves the theorem. □

Thus, for fixed alphabet sizes (i.e. fixed r), $C(x)$ has an asymptotic upper bound of the entropy of the sequence. Of course, $C(x)$ could be much lower. However, the above theorem can be generalized to show that if the probability distribution $p(x)$ is computable, then the entropy of the distribution and the expected complexity of a sequence drawn from the distribution are asymptotically equal. This is what one would intuitively expect from knowing the fact that most sequences have near-maximal complexity; $C(x)$ will only be lower than the entropy when there are non-frequency regularities to exploit, and such regularities are usually not present.

MORE TURING MACHINE BASED COMPLEXITY MEASURES

In this section we'll look at other measures that have been suggested for capturing the presence of complexity. Different properties are measured by each one; a combination of these and other complexity measures will be necessary to fully describe the intuitive notion of complexity.

Logical Depth

Resource-bounded Kolmogorov Complexity measures the length of the minimal size program that will compute a string S under given restrictions of time and/or space. We can turn this definition around and ask: what is the length of time the (resource-unbounded) minimal program for S takes to compute S ?

This is a satisfying start toward quantifying the amount of time (i.e computational effort) required to generate S . By the invariance properties of KC, it's an intrinsic property of the string (up to a constant).

The main problem with this approach (as foreshadowed by the results on the ill-behavedness of Resource-bounded KC in the previous section) is that there may be a program which is only slightly longer than the minimal program but which runs much faster. Ideally we should sum the probability of occurrence of each algorithm times the

length of time it takes to compute the string in question; this would give us the "expected length" of computation that it took to create the string.

Such a measure, called logical depth, has been proposed by [Bennett]. It incorporates a significance level approach: a string's s -significant depth is the least time required to compute it by a program that can be compressed by at most s bits.

"...information per se is not a good measure of message value. For example, a typical sequence of coin tosses has high information content but little value; an ephemeris, giving the positions of the moon and planets every day for a hundred years, has no more information than the equations of motion and initial conditions from which it was calculated, but saves its owner the effort of recalculating these positions...the value of a message is the amount of mathematical or other work plausibly done by its originator, which its receiver is saved from having to repeat.

Of course, the receiver of a message does not know exactly how it originated; it might even have been produced by coin tossing. However, the receiver of an obviously nonrandom message, such as the first million bits of π , would reject this 'null' hypothesis, on the grounds that it entails nearly a million bits worth of ad-hoc assumptions, and would favor an alternative hypothesis that the message originated from some mechanism for computing π . The plausible work involved in creating a message,

then, is the amount of work required to derive it from a hypothetical cause involving no unnecessary, ad-hoc assumptions."

Definition: Letting A and B be strings, and s a significance parameter, the depth of A relative to B at significance level s (denoted $D_s(A/B)$) is:

$$\min\{\text{TIME}(U(p,B)) : (|p| - \text{KC}(A/B) < s) \text{ and } (U(p,B) = A)\}$$

where

$U(p,B)$ is the output of the prefix UTM U simulating program p on input B , and $\text{TIME}(U(p,B))$ is the running time for $U(p)$ on input B .

(If B is the empty string, we can simply refer to the depth $D_s(A)$ of A .)

(Note that since UTMs can simulate each other efficiently, we can make depth more or less machine-independent. For any two UTMs, there exist constants a,b,c such that, for any t , strings whose $(s+c)$ -significant depth is at least $at+b$ on one machine will have s -significant depth of at least t on the other.)

For a given significance level s , a string is called t -deep if its depth is greater than t , and t -shallow otherwise. (The t may be omitted when t is small and the property under discussion does not depend upon the exact value of t .)

In order for depth to characterize the notion of "amount of work necessary to produce a string", it must be hard to produce deep strings from shallow ones; a random process should be capable of such a production with only low probability. This is indeed the case, as the following "Slow Growth Law" states:

THEOREM: Given a string A as input and two significance parameters s and t with $t > s$, a random program (generated by coin tosses) has probability less than $2^{-(t-s)+O(1)}$ of generating an output whose t -significant depth is greater than the s -significant depth of A plus the run time of the program plus $O(1)$.

More formally, there exist positive constants c and d such that for all strings A , and all positive s, t with $t > s$, the prefix set Q defined by

$$\{q : D_t(U(q, A)) > D_s(A) + \text{TIME}(U(q, A)) + c\}$$

has measure less than $2^{-(t-s)+d}$.

PROOF: Let

p be an s -incompressible program computing A in time $D_s(A)$,

r be a "concatenating prefix", so that $U(rpq, A) = U(q, U(p, A))$, and

Q be the prefix set in the statement of the theorem, where the constant c is large enough for the concatenation process to be done.

For any $q \in Q$, rpq computes $U(q, A)$ in time less than its t -significant depth, by definition of q . Thus rpq is compressible by at least t bits.

The sum of the algorithmic probabilities P for all possible rpq is therefore:

$$\sum_{q \in Q} P(rpq) < \sum_{q \in Q} 2^{-|rpq|+\epsilon} = 2^{-|r|-|p|+\epsilon} \mu(Q)$$

where the "measure" of the set of prefix programs Q is defined as:

$$\mu(Q) = \sum_{q \in Q} 2^{-|q|}$$

(Note that the "measure" of all programs being considered is guaranteed to have a sum at most 1 by the Kraft Inequality, since all programs are prefix programs; in fact, we can and will take the measure of all prefix programs to be exactly 1, as justified by Bennett.)

On the other hand, since p can be recovered from rpq (by deleting r and reading in and executing pq until the program halts, when exactly p will have been read) the algorithmic probability of p is at least as great as the sum of algorithmic probabilities of prefix programs of the form $crpq$ (where c is a constant size prefix which indicates how to recover p from $crpq$):

$$P(p) > \mu(Q) 2^{-|r|-|p|+\epsilon-O(1)}$$

Now recall that $KC(p)$ is equal up to an additive constant to the negative log of its algorithmic probability. Combining this with the s -incompressibility of p , we have:

$$P(p) < 2^{-(|p|-s+O(1))}$$

and combining this with the previous equations,

$$\mu(Q) < 2^{-(t-s)+O(1)}$$

which proves the theorem. ⊙

An analogous result holds for the slow growth of $D_s(A/B)$.

Relative depth measures the effort (amount of time) necessary to construct one string from another. Analogously to relative Kolmogorov Complexity, it also allows the difficulty of constructing the content of a string to be separated from the difficulty of making a string of a certain length. For example, $D_s(0^n | n)$ is $O(n)$ --this is essentially the minimum possible depth, as it takes at least n time steps to print out a string of length n . However, $D_s(0^n)$ can be extremely large if programs for producing n are very slow. Just as with relative Kolmogorov Complexity, other information than the length (such as substrings or statistical properties) can also be "factored out".

Random (incompressible) sequences R are also shallow; the fact that R has near-maximal C implies that a simple print program (again running in $O(n)$) will be only marginally larger than any other program for R . (Of course, this program will be very large even though it runs quickly.)

This shows, for example, that the number Ω is shallow at some significance level

s! This may seem paradoxical, since (as mentioned previously) production of Ω is a very time-consuming process. The reason for the shallowness is that Ω can't be distinguished from a random sequence; even though there is an extremely slow process to generate it, it could just as plausibly have been created by a print program. (For large enough significance levels everything is shallow.)

How about deep objects? As one might suspect, the Halting Set is deep. It can be used to speed up slow computations, and thus to calculate deep objects; by the Slow Growth Law, it is thus deep itself with probability 1.

Letting $B(n)$ be the largest finite number of steps a prefix program of at most n bits takes to complete its computation (called the shift Busy Beaver function), we can easily show that $B(n)$ is an approximate upper bound on the depth of finite strings. Consider a given string A of length n , and a significance parameter s ; suppose that A is computed by a prefix program P of size $n-s+O(\log n)$. (The $O(\log n)$ is the overhead for the self-delimiting prefix description.) Then the s -significant depth $D_s(A)$ measures the speed of the fastest program of size at most $n+O(\log n)$ generating A , which obviously takes at most as many computation steps as P does. And by the definition of $B(n)$, P takes at most $B(n-s+O(\log n))$ computation steps.

The interpretation of the Slow Growth Law is that there is a universal notion of the "amount of work" that a string contains, under the assumption that strings are

produced by random programs. For a system where the programs have any sort of adaptive capability, this assumption will be invalid; the random distribution of programs that the system is seeded with will quickly become nonrandom as programs learn and better programs proliferate.

Does some analogue of the Slow Growth Law hold even under exploration by purposeful programs? If there is a particular deep string we would like to generate, to what extent can the intrinsic depth be overcome by our skill and knowledge in choosing effective algorithms? This question suggests a notion of knowledge-dependent depth: instead of weighting the running time of generating programs by their length, we would have to weight by the "simplicity" of the program. (The rationale for this is that "simpler" programs to generate the desired string are more likely to be found during our search, and so their running time should be given greater weight when averaging over the methods of generating the string that we are likely to come up with.)

Further objections to logical depth as a measure of complexity:

i) Perhaps the "quality" of computation involved in generating strings should be more important than the "quantity". In other words, instead of weighting programs by their running time, weight them by some other quality of the program or the computation of the program.

ii) Weighting 1 n -bit program with 2^k $(n+k)$ -bit programs seems ad hoc.

iii) This weighting is induced by the universal measure m , which is analogous to a uniform probability measure; as noted before, most general universes will have nonuniform measures. (How nonuniform, one wonders?)

Sophistication Complexity

Recall that a problem with using Kolmogorov Complexity to measure "intuitive complexity" is that it assigns a high value to random strings. Logical depth overcomes this by showing that random strings are probably ones that were generated quickly, using a small amount of "computational effort". Another possible way around the randomness problem, discussed in [Koppel 88], is to consider the description of a string S as having two parts: one part describes the "structure" of S , and the other identifies S from among all the strings belonging to the class described in the first part. The sophistication of S is the size of the first part.

For example, consider three binary strings: a random string, a string in which each odd bit is random but each even bit is 0, and the string consisting of alternating 0's and 1's. The first has no structure, the second is partly characterized by the structure of the even bits, and the third is completely specified by its structure. Note that under this definition, a random string has low sophistication, as it has no structure.

How can we make this notion of "structure" more precise? Recalling that for Kolmogorov Complexity the length of the minimal program (which was an input to a UTM) incorporated both the length of the program and the length of the input data, we can give one formalization of "structure" by measuring the length of the minimal possible "program" portion. (This is the portion of the input to the UTM that allows the UTM to

simulate the desired function.) Thus KC can be redefined as:

$$KC(S) = \min\{|P| + |D| : U(P,D) = S\}$$

where S is the output when the UTM U runs program P on input D.

And the c-sophistication of S will then be:

$$SOPH_c(S) = \min\{|P| : \exists D \text{ such that } U(P,D) = S \text{ and } |P| + |D| \leq KC(S) + c\}$$

Here the parameter c gives the number of bits by which the size of the minimal program-data combination can be exceeded. If the constant is unimportant, one can simply talk of the sophistication of S. (Recall that KC(S) returns the minimal input to a UTM, which consists of a single prefix-free string encoding both the program and the data.)

Thus, we are looking at all program-data combinations that can be input to a UTM to give S as output, of size at most c bits more than the smallest such combination. Among all combinations which qualify as input, the one which has the smallest size program is identified and the size of this program measures the "algorithmic kernel", or sophistication, of the string.

The justification for choosing the minimal size program is that this program represents the smallest set of regularities or patterns that will describe the data--in essence, a variant of Occam's Razor. (Of course, such properties as decomposability could provide other criteria of simplicity, if they could be measured.)

Surprisingly, Koppel has shown that slightly modified versions of sophistication and logical depth are equivalent up to an additive constant. Thus, the two notions of "expected time to generate X " and "expected length of the generating program for X " are fundamentally similar.

Formally, sophistication complexity can be extended to an infinite string S by looking at longer and longer initial segments of S . If there is a fixed finite size program P that can generate these increasing length segments when given an increasing amount of data as input, P is said to be a compression program for S , which is then said to be describable. (This is equivalent to saying that describable strings have the property that the sequence of values for the c -sophistication of increasing length initial segments is bounded above for some constant c .) Otherwise, the infinite string S has infinite sophistication complexity and is said to be transcendent. (The name is motivated by the fact that the algorithmic generating process for S in the latter case has unbounded length.)

Note that describable infinite strings will have an infinite number of compression programs. Intuitively, larger values of c may allow superfluous (or at least less compact) processes in the compression program, such as dummy statements or the incorporation of a more complex process than is needed for the particular string. However, as shown in [Koppel 91], all of the compression programs for an infinite string S are equivalent in the limit; they yield a converging set of possible continuations of increasing length initial segments of S , as well as equivalent predictions of the most likely next symbol. (For a

given compression program P , the probability of a particular substring SUB following a given initial segment SEG is the number of data inputs to P which yield the string $SEG + SUB$, divided by the number of data inputs to P which yield the string $SEG +$ an arbitrary substring.)

As an objection to sophistication complexity, note that the program-data distinction is somewhat artificial; in a way, sophistication just shuffles the randomness from the data to the program! If we have a "random incompressible program" that does nothing much, the algorithm may be very large but the output will be "effectively random". What have we gained?

Presumably, large random programs arise less often than large random data. Just a few algorithms running can make a great mess of confusing data, and it's certainly of use to be able to separate out the minimal algorithmic description of a situation.

How can we go further? One idea is to eliminate the binary program-data distinction with a continuum. In nature, all the programs (except possibly for the laws of physics) are also data which are actable upon by other programs. So perhaps the frequency of use as a program versus use as data (measured as the number of read/write accesses) would be an interesting measure.

But we have a problem of deciding whether, say, the number of read accesses to a portion of a program tape in a UTM is proportional to "structural importance". We could try poking and tweaking the program and seeing how much it changes, but now we're back to general "evaluation of complexity" strategies.

Second idea: the program for a given string in the definition of sophistication is, by construction, almost incompressible--i.e. random. What would the sophistication of this program be? It seems that this recursive sophistication could give a sequence of decreasing length programs, each of which generates the previous one given appropriate data; for how many steps would such sequences typically decrease for? What types of interesting strings would have a large length decreasing sequence; could looking at the programs corresponding to the sequence in reverse order yield insights into "bootstrap programs" which generate more and more sophisticated versions of themselves?

Third idea: large random programs are argued to be unlikely to occur by recourse to the universal measure m , which tells the probability of a program coming about by a random series of "coin flips". This would be reasonable, under the assumption that coin flips were how the program originated--which is usually false. Clearly, then, it's of interest to look at the complexity of the originating process; what if program A is likely to be generated from program B which is likely from m ? How does the sophistication of A making B making S carry on to B?

One point that seems important is: whatever qualities sophistication or other complexity measures check in data may be just as valid and interesting in programs themselves--and programs for programs (metaprograms), and so on.

Stochastic Complexity

When applying statistical methods to find regularities in data, the general procedure is as follows:

- Pick a particular model class which the data will be fitted to, i.e. a particular set of assumptions that will be made about the data.
- Using the consequences of these assumptions (i.e. the theory of the model class) and the data values, find the specific model or set of parameter values which is "closest" to the data with respect to some criterion.
- If the derived model is not sufficiently "close" to the data then begin anew from the first step.

There are several questions one might have when looking at this procedure:

- How is the model class to be picked?
- What criteria should be used to measure "closeness"?
- Given a closeness measure, how does one decide if the derived model is sufficiently close to the data? In a practical sense, this translates to asking:
 - How does one decide if the assumptions of the theory capture some property of the data-generating machinery?
 - Can a better model be found given the computational resources available?

- How can the utility of different models be compared in a standard fashion? In particular:

- How can the assumptions that comprise the model class be critically examined in light of the performance of the model, so that inferior assumptions can be dropped and new ones proposed?
- Any data can be explained by a model with sufficiently many assumptions; how can the complexity of the model itself be taken into account?

These are all deep questions, and together they articulate questions to be answered to understand the process of problem-solving and learning. We'll look at two approaches which attempt to formalize some of these answers.

Consider first the last question above: how can the complexity of a model be taken into account when deciding how well the model explains the data? One possible answer has already been foreshadowed in the discussion of likely explanations for the algorithmic generation of data strings: simply take the shortest model as the best one. We can then measure the complexity of data by its Stochastic Complexity ([Rissanen]); this is the length of the shortest model which generates the data.

More formally, given data D and a model class to be used to describe D , the Stochastic Complexity of D is that model from the model class which minimizes the sum of the length of the description of the model and the length of the data when encoded by

the model.

The Stochastic Complexity of D differs from the Kolmogorov Complexity of D in the following crucial respects:

- The model class is restricted--arbitrary programs are not permitted. This is done so that the Stochastic Complexity will be computable or at least easily approximable (in contrast with Kolmogorov Complexity).
- The focus is on finite size data strings D , and finding the actual shortest model. Thus, the asymptotic invariance theorem for Kolmogorov Complexity is of little use.
- Different model classes will yield different values for the Stochastic Complexity of D . Since finding the model class giving minimum D is uncomputable, prior knowledge and heuristics are used to guess good model classes.

The first difference is particularly important. While Stochastic Complexity is a useful metaphor, it is also intended to be a practical technique; restrictions on the allowable models, precision of data storage, and other such factors must therefore be made.

Even then, it will often be the case that there is no known method for finding the global minimum; as with all areas of optimization, heuristics and approximation techniques are then used. The Stochastic Complexity in this case serves as a traditional

"goodness-of-fit criterion" to be minimized, with the advantage that procedures developed to minimize the criterion serve the purpose of simplifying the model as well as the data.

As with Kolmogorov Complexity, the justification for choosing models with low Stochastic Complexity as "good" models is that, by virtue of their succinctness, they must have maximally compressed the data and found all exploitable regularities. Such an explanation can be expected to be the most general one possible. This approach is very common in science and mathematics in general, where reduction of a body of knowledge to its constituent parts or axioms indicates that the fundamental reason and description for the knowledge has been found.

As an example, let's consider the case where D consists of n data points, $(x_1, y_1), \dots, (x_n, y_n)$. We will restrict the model class to be that of polynomials (of any degree less than n)--note that this decision is "heuristic". A degree- n polynomial can be found that will fit all data points exactly, but if the data points are noisy samples from a lower-degree polynomial, we would prefer to find the latter model. We will also let the description of the x_i be part of the assumed model class.

Any $k-1$ degree polynomial can be described by k values each of precision d bits. Noting that we need to specify k and d as well, the polynomial can be described in $kd + O(\log kd)$ bits.

To describe the noise, we assume that each y_i is the outcome of a normal random variable with mean $f(x)$ and constant variance. Since the probability of obtaining y_i is of the order of $\exp(-(f(x_i) - y_i)^2)$, we can use a prefix code to encode y_i in $s(f(x_i) - y_i)^2$ bits, where s is a scaling constant. (Here we are just encoding the exponent in a manner similar to that used in the proof of the Kraft Inequality--the length of code words of the prefix code corresponds to the negative log of probability in the probability distribution.)

Adding together all the descriptions, we get that the quantity to be minimized is:

$$kd + s \sum_{i=1}^k (f(x_i) - y_i)^2$$

(where the $O(\log kd)$ term is relatively small and ignored). Finding the polynomial which gives the least value for this quantity will give the stochastic complexity of D relative to the described model class.

Statistical Complexity

Although Stochastic Complexity extends traditional statistics by taking the complexity of the model into account, it does not give much information on the process of actually deciding on appropriate model classes. In [Crutchfield], a framework for inductive reasoning and model building has been proposed, called hierarchical ϵ -machine reconstruction.

The reader should note that this framework is not meant to be a magic algorithm for universal induction. It is a mental model with which to think about the process of finding structure and computational models for processes. For most of the steps in the model, no deterministic algorithm is known (or probably even possible); building a theory of suboptimal procedures for these steps is an ongoing task.

The central quantity in this approach is called Statistical Complexity. It is roughly equivalent to Kolmogorov Complexity with "randomness given for free"; the minimal description is initially searched for among less powerful computational models than a TM in the hope that such a model suffices to describe the data. More formally, define a Probabilistic Turing Machine (PTM) as a UTM with access to an external source of random bits. Then (in the case where a full-fledged TM is necessary to describe the data) the Statistical Complexity SC of x is the size of the minimal program for a PTM that is most likely to give x as output. Note that even a trivial PTM which generates totally

random output can generate x ; the "most likely" condition forces the machine found to incorporate all (and only) those rules which give a better explanation of the data.

The SC can also be thought of as the amount of information that must be stored to make optimal forecasts of the future of the data sequence (i.e. forecasts with an error rate equal to the entropy). An infinite sequence whose length-conditional KC is finite must be specified by a finite deterministic process; SC will therefore be of special interest when the length-conditional KC of a sequence is infinite. Note that any such sequence that can be "effectively described" (such as "compose a piece of music with note frequencies and durations drawn from a normal distribution") must consist of a finite specification or model which uses a source of randomness to get the infinite KC; the finite specification or model is what SC attempts to measure.

SC gives values close to zero for both highly ordered and highly random sequences; the strings with maximal SC are those whose data-generating machinery contains large numbers of irreducible deterministic and stochastic computational elements. For a maximally complex sequence (i.e. one that cannot be modelled by anything less than a TM), SC is asymptotically the difference between a sequence's KC and entropy. (As suggested by the material in the section on Connections between Information Theory and Kolmogorov Complexity, the KC of a sequence drawn from a stationary stochastic process approaches its entropy rate; such a sequence will therefore have finite SC.)

Given a data stream arising from some process, the goal of the computational agent can be thought of as building a model that can help it predict the data. This model, in turn, can be represented as i) a set of states $S = \{S_1, S_2, \dots, S_n\}$ the agent can be in, and ii) a mapping T between these states. The pair (S, T) is referred to as an ϵ -machine, and specifies the data-predicting machinery.

For all times that the agent is in a particular state S_i , it has the same knowledge about the future of the data stream; being in S_i suffices to make the future conditionally independent of the past (as far as the agent can tell).

The types of mappings T between states that will be permissible will be those corresponding to different formal computational models, such as finite automata or context-free grammars. (There is an infinite partial ordering of such computational models; practically, some finite subset of these with properties appropriate to the sequences being analyzed would be used.) Of course, it would be desirable to use the least powerful computational model which is adequate to describe the process, as this corresponds to a "simplest description" (note that all such computational models can be written as PTM programs). When comparing Statistical Complexities, therefore, both the value of the SC and the model which the SC is taken with respect to should be considered.

The reconstruction procedure for finding the SC relative to an appropriate

computational model for given data works as follows:

- Start by assuming the data is random: the model equals the data.
- Given a particular model class (i.e. class of automaton) find a model in the class that fits the data. If such a model can be found that is finite, stop--an explanation has been found.
- If the length of the model that describes the data increases without bound (practically speaking, if it approaches the computational resources available) then "innovate" by enlarging the model class to a more powerful one in the appropriate computation hierarchy.

Create this more powerful model from the previous one by looking for patterns of transitions between groups of states in the previous model; these groups of states in the previous model become states in the current model. (The state-grouping step may not be enough to find a new computational level; however, it will hopefully reduce the necessary search space by a substantial amount.)

Essentially, the procedure tries to find the minimal model at the least powerful computational level which gives a finite description for the nonrandom portion of the data stream. The power of the model found will increase with the computational resources available, until the "true" model is found. In a competitive resource-bounded environment, a less-powerful model which uses less resources may well prevail; compare humans and insects.

The computation hierarchy referred to is a hierarchy of language classes. Starting with a verbatim description, the procedure then tries to find finite descriptions of the data stream in the form of trees, then finite automata, and so on up the lattice to more and more powerful computational systems.

The procedure outlined above for finding Statistical Complexity is a guideline only--not a precise algorithm. Methods of finding formal grammars in a particular data class that fit a data stream are explored in the area of Grammatical Induction. The problem of finding a Finite State Machine to describe a process is one special case that has been the focus of much study, and that first suggested many of the ideas mentioned above.

FURTHER IDEAS ON COMPLEXITY MEASURES

Abstraction From KC

It would seem that qualities such as epistasis, decomposability, well-conditionedness, and so on could all form complexity measures. What would we gain from putting these in the framework of TMs? Well, the same as we got from the formalization of KC--by casting the principle of "minimum description length" in this framework, we get use of a formal description language that applies to all algorithmic processes.

Here is an interesting generalization of KC, which may be quite universal. The definition of KC itself is the shortest program-data input to a UTM that outputs a desired object. (For ease of notation, we will equivalently talk of the program-data input that computes a desired function, whose output is the desired object.) But noting that each program can be represented as a bit string itself, we could rephrase as:

"Consider the class P of programs which compute the function F .

Define a new TM LENGTH which, when given a member of P as input, returns the length of that member of P .

We can then define $KC(F)$ as $\{\min(\text{LENGTH}(p) : p \in P)\}$."

This gives an insight into why Kolmogorov Complexity is not computable; the

characteristic function of the class P of programs over which the minimization is to be done is itself not computable (as suggested by Rice's Theorem).

Now, if we replace the TM LENGTH in the above definition by another TM which computes a different property of bit strings, we have defined a different complexity class:

"Consider the class P of programs which compute the function F .

Define a new TM PROPERTY which, when given a member of P as input, returns the value for PROPERTY of that member of P .

We can then define $KC_{\text{PROPERTY}}(F)$ as $\{\min(\text{PROPERTY}(p) : p \in P)\}$."

All we need now is algorithmic representations of TMs DECOMPOSABLE, PREDICTABLE, CONTROLLABLE...

Note that finding these algorithmic representations will be a major task, in contrast to Kolmogorov Complexity where LENGTH is a trivial TM. So just having these representations would in itself be of great interest as it would allow us to evaluate properties of particular programs.

The task of finding the program minimizing such a property for a function would of course be an additional extremely difficult problem (for both Kolmogorov Complexity and generalized complexities). How can we find algorithms for the automatic generation of programs which compute the same function as the input program but have reduced

complexity measure? (Some extremely important algorithms for doing this that are already intensively used are in the area of compiler technology--optimizing compilers (speed/length) and decompilers (understandability for reverse engineering) find simpler programs for different complexity measures).

It is interesting to try to formulate this idea with the Blum axioms and syntactic/semantic computational complexity measures in mind.

The Blum axioms simply associate a computable value with every program; LENGTH is one such value. Kolmogorov Complexity is calculated via an uncomputable procedure which looks through all the LENGTH values for equivalent programs and picks the smallest one; the generalization from the Blum axioms is the grouping into equivalence classes of programs (undecidable, from Rice's Theorem).

In the Kolmogorov Complexity generalizations, the value associated with each program may itself be uncomputable. Semantic computational complexity classes also have this property; membership in the class is not computable in general for a nondeterministic program.

It seems like UTM invariance would often hold for generalized Kolmogorov Complexity measures: since UTMs can simulate each other, changing universal machines would change the complexity by at most the complexity of the UTMs. And most complexity measures would give low complexity for a UTM (which is quite a simple machine to describe and understand).

For many measures, a suitably defined Incompressibility Theorem might also hold; this would depend on showing that there are relatively few programs that are simple relative to the complexity measure.

We can also extend Kolmogorov Complexity in a natural way via oracles: Statistical Complexity with respect to the maximally powerful computational class of TMs, for example, is just KC relative to random oracles. This process of taking subalgorithms for problems that we know how to solve well and turning them into new "primitive procedures" which can be used to build the next level of knowledge is essentially how cumulative knowledge works, as well as how our minds learn. We build on what has gone before; once we know how to do something, it has low cost and problems easy relative to it become easy overall.

How about KC using restricted models of computation? This echoes Crutchfield's Statistical Complexity; we want to find the most restricted computational level such that the (findable) KC is finite/reasonably small. Question: for natural computational problems, as we successively restrict the KC to smaller and smaller "natural" computational classes, how does it grow? If the value for the restricted KC ultimately becomes infinite, is the penultimate value reasonably small?

Another extension is to consider the class of programs computing not the exact function but an "approximation" to the function. In this case, a program would fall in the

class to be examined if it computed a function "sufficiently close" (with respect to some defined "function distance" metric) to the desired target function. Note that each program will often approximate many other programs; however, the measure is well-defined since for each program P and each function distance metric, all other programs either will or will not be within the desired distance of P .

It would be interesting if some natural function distance metrics allowed a "basis set" of programs to be computed for each given distance d , so that every program was "within a distance d " (with respect to the function it computed) of only one program in this basis set. This would allow approximate understanding of arbitrary programs with respect to the properties captured by the basis set. In fact, such a basis set is easy to define for the basic version of KC: simply take the basis set to consist of one incompressible program for each of the lengths k , $2k$, $3k$, and so on, and the function distance between two functions to be the difference in length between the minimal programs for the functions. This example and results such as Rice's Theorem suggest that, while such function distance metrics may be definable, any nontrivial ones will make the problem of finding the closest basis set program for a given program uncomputable in general.

Decomposability

As an example of a somewhat different KC measure, consider the shortest TM required to predict the whole of an object from a significantly smaller part of it. Since many different wholes are possible from a part, this will have to be a probability; the TM will have to guess the most reasonable completion from the part of the pattern it can see. This is like getting a theory from data--a generalized form of statistical inference. As the size of the part given to the TM increased, so would the accuracy of the prediction; if a plateau occurred where increasing size of input did not increase accuracy, this might indicate that the size where the plateau started was that of a natural "subpattern" of the overall pattern.

Let's try to formulate an appropriate algorithmic representation for decomposability. What is the minimal size TM (or resource-bounded KC TM) to "tie together" two parts of a whole? In other words, given a TM ORIG, what is the minimal size TM MIN such that MIN can make some two parts of the ORIG function behave the same (output the same results on the same input) as ORIG? (As a consequence of the definition MIN will be incompressible, since otherwise a compressed MIN would be shorter than MIN and perform its function.)

Take as an example the case of ORIG representing a sum of two functions f and g , each of which is assumed to be incompressible (i.e. to have maximal KC); MIN would then simply be an adder in TM format.

Actually, this is an upper bound for MIN since some incompressible f and g will be compressible relative to each other. For example, let f be an incompressible binary function and simply define $g(x) = \text{NOT}(f(x))$; then $f+g$ is 1 for all x . We need to word our decomposition definition with care so that intuitive subprograms are treated as such.

More formally, let $KC(f) \geq |f|-c$, $KC(g) \geq |g|-c$ for some constant which is small relative to $|f|$ and $|g|$. Let it be true that $KC(f|g) \geq KC(f)-c$ and $KC(g|f) \geq KC(g)-c$; this says that f and g are "independent", so that being able to do the function g does not make doing f any easier. (Hopefully it is in general possible to find f and g so that this condition is true.) Then if we define $h(d_1, d_2) = f(d_1) + g(d_2)$, $\text{MIN}(h) = +$ and h is decomposable to (say) half its size.

Note that even though $KC(h)$ is roughly $|f| + |g|$, we have decomposed it to parts, none of which has complexity greater than $\max(|f|, |g|)$. This will not always be the case; sometimes our function h may be such that no decomposition is possible. Can such examples be found? One problem with using a counting argument to show existence is that programs are tweaked a bit when decomposed (i.e. h is not just f concatenated with g).

There is no reason to stop with a simple decomposition into two parts. Picturing the computation as a digraph, the given example corresponds to a two-level tree with one parent and two child nodes, with arcs taking the output from the functions f and g (the child nodes) to the function h (the adder). We could have any number of nodes at the second level; more generally, we could consider decompositions into a form corresponding to an arbitrary digraph, in which case the problem of understanding the

digraph itself comes recursively into play.

Recall the example of "chunking"; large routines become new primitives. This suggests KC where each unit of length measurement is itself a big black box doing something predictable. There is also a tie-in to cognitive science where successively more complex concepts are treated as chunks: the number of chunks we can handle stays relatively constant, but the apparent complexity of concepts we can deal with increases as our chunking depth increases. (Note that data and algorithms are both chunked!).

Patterns, levels...the complexity we see depends on the level at which we examine the algorithm. We can understand the atomic steps of any algorithm (e.g. the effect of the basic TM commands) with a well-defined and reasonably concise language. One would think that even multisteps of 2 to 5 atomic steps could probably be combined--but no--the Busy Beaver problem shows otherwise! We don't understand some 5-chunks, where one restrictive definition of "understand" is "will it halt"; others could be "what will it output", "how many times will a given instruction be executed", and so on. Of course, a "random" 5-chunk may not be understandable because there is nothing to understand--but a 5-chunk with some special property like BB(5) is not "random" in the intuitive sense.

In general, we might want approximate decomposability, and we would require MIN to fall in some appropriate approximating class for ORIG. Note the similarity between program decomposition and object-oriented programming, where subalgorithms

are "encapsulated" and function independently, with program-specific information being passed between them.

The purpose of decomposing is to get simpler subparts, each of which can be understood; the subparts can then hopefully be combined in a simple manner to understand the overall process. However, some programs will not be decomposable. (In fact, perhaps this is true of almost all programs, by analogy to the Incompressibility Theorem for KC.)

We can fix some D and say that an algorithm is D -decomposable iff we can find a decomposition of the algorithm such that the maximal complexity of any subcomponent or of any "tying-together component" is at most D . Then the author conjectures that human minds are constructed such that problems that are not D -decomposable for some relatively small value of D are effectively insoluble by human minds alone.

Justification is the finiteness of our minds and the experimental findings from cognitive science showing a maximum of 7-10 chunks that can simultaneously be in "operational memory". If this is the case, then D is a crucial bottleneck to improved cognition--how much larger would the class of thinkable concepts become if we could hold 15 (or 50) chunks at once? What are some canonical problems that would be doable (e.g. SAT / FSM / Busy Beaver understanding) with expanded D -decomposability?

As an aside, note that a combination human-computer team could solve D -indecomposable problems if the sections of complexity greater than D could be farmed

off to the computer. This suggests formulating classes of D-indecomposable problems which computers can solve but humans can't (these classes will change every few months or so...).

The Busy Beaver problem illustrates how some tiny algorithms are tough to analyze. Solving the Busy Beaver problem for size n gets you the first n bits of Ω , which as we have seen suffices to decide all problems expressible in size n . The author conjectures that the tractability size of BB is a sort of complexity constant UB for each "computational universe"; problems not UB-decomposable aren't exactly (or even practically?) solvable. ("Computational universe" could for example be a CA--a 1-dimensional CA would have a bigger BB size, while a system with more complex atomic rules would have a smaller BB size.)

Intelligence

"Intelligence" is a property that is notoriously difficult to define. One measure might be the ability to guess the internal program of a TM from its external actions; another might be the ability to predict its future actions from its internal program. For each particular TM, this could be defined as the rate of increase in prediction ability given a stream of input-output data pairs.

General prediction ability might be defined as the degree of predictive success with each TM averaged over the probability distribution of expected encounter frequency with the TM. If the TMs that we encountered were generated by a random procedure ("a monkey typing on a TM compiler") then the ability would just be a sum of the probability of success for each TM weighted by the algorithmic probability of the TM. However, the physical universe's probability distribution (and that of most computational universes) is highly nonuniform; as well, if good guessers get more prevalent (due to their survival advantage), then we have (co)evolution and the probability of success will constantly change.

Let's try to formalize this a bit more. Consider the problem of predicting some statistic of a dynamic pattern of an evolving system. Difficulty of prediction is not in itself a measure of complexity; a random sequence is unpredictable. But on the other hand, random sequences have no "structure", and are thus in this respect utterly predictable--with the machinery of statistics, we can tell tons about uniform or IID variable behavior. (It's usually easy to outwit randomness!) So the difficulty of

predicting (some set of) statistics could serve as a measure; difficulty could be measured by size of machine, and would again have resource-unbounded and bounded versions.

For actual implementation, one could have a bunch of common small algorithms continuously checking their environment (which would be a big messy data set) to see if they can get nontrivial predictions. (Common algorithms could include those looking for repetition, clustering about an average value, or exponential growth; any empirically common property that could be checked by a small program.) Ideally one would have a "basis set" of small TMs, weighted for prevalence and performance in the current environment. As with biological life, this leads to the potential for arms races and parasites.

We can then turn this around and measure the accuracy of the predictor algorithm in computing or predicting statistics. How well does the predictor perform over predicting the statistics of a large number of configurations? Presumably, a predictor would have to capture the essence of the reasoning behind a statistic to perform well with a variety of configurations; if a sufficient variety of statistics were used to test the predictor, this would measure its general-purpose prediction capability, which encompasses much of what we mean by "intelligence".

Note that there will be great difficulties in i) choosing appropriate statistics (what statistical properties of the universe are essential to know to survive in it?), and ii)

choosing appropriate configurations to test on. Organisms don't generally get faced with totally random configurations; the greatest part of their interactional universe is highly ordered--and arguably the most important/difficult part is the prediction of other predictors. Thus the ensemble of configurations on which predictors will be tested should be chosen to adequately represent a "typical environment"--a highly nonuniform probability distribution, and one that changes as its component predictors change.

Two other questions involving "learning" suggest themselves:

- 1) What happens when the system is examining itself and trying to judge how well it (or a subpart of it) is doing?
- 2) How can an environment implicitly judge the "intelligence" of an algorithm, and strive for improvement of the algorithm?

Genetic Algorithms and Evolutionary Computation are two heuristic methods which start to answer these questions; we'll discuss them briefly later.

Practical Usefulness of Complexity Measures

When we deal with complexity measures it is natural to wonder about questions of usefulness. When describing a system, the effective approximation of a complexity measure C is a necessary condition for the measure to be of operational use. (Even better would be effective exact solutions, but many of the measures discussed thus far are not even computable, let alone falling in some nice class like P .)

We would thus ideally like C to fall into an appropriate approximable computational complexity class. It may, however, be more realistic to look at statistics of C , or sampling of C on small subsets of the object or function X in question; we would then need the weaker requirement that C be guaranteed to give us some nonzero amount of information about X . This guarantee could take the form of one of the ϵ - n classes proposed before, or some other general complexity class. In order to be useful, the guarantee would have to include some property of interest, depending on the problem to be solved.

Will we always have tradeoffs between the generality of our problem classes and the power of algorithms for them?

Some other ideas that would be interesting to look into:

- Application of complexity measures to each other.

- Non-stationary systems and adaptation.
- Actual universal measures versus "random" (uniform) universal measures.
- Dynamic complexity measures.
- Some effectively approximable complexity classes.
- A picture of the universe as a collection of many computational loci, each of which has very restricted computing power.

Although the problems to be solved seem overwhelming, observing our physical universe gives cause for optimism. Usually knowing a few "dominant parameters" seems to suffice to put us in control.

Indeed, as discussed in [Cohen and Stewart], it is amazing that the universe is simple enough for us to manipulate and control. On the other hand, the "Anthropomorphic Principle" suggests that our guesses of what is "probable" are strongly conditioned by the fact that we did evolve; if the universe were not simple enough for us to manipulate and understand, we wouldn't be here wondering about it.

DISTRIBUTED COMPLEXITY AND THE EVOLUTION OF

INTELLIGENCE

In this final section, we'll take a brief look at many different ideas which extend the material discussed earlier.

While the Turing Machine based computational complexity paradigm has yielded immense insights, it's not the only level at which complexity can or should be studied. Why? Two reasons: inexact discussion at higher levels may yield more useful information (chemistry and biology are just "higher-level physics", but useful nonetheless); Church's Thesis may not hold.

There is also the extension to be made from sequential to parallel. We need to adopt a "decentralized mindset" to find solutions which involve the interaction of many computational nodes. How can simple computational units combine their efforts to solve complex problems--especially when there is no central "overseer" coordinating the problem-solving tasks?

And, of course, the primal Big Question remains: how does an initially simple system evolve patterns of increasing complexity?

Decomposition and Hierarchy

A key strategy in problem solving and making complex situations tractable is to use decomposition. If a big problem can be split into a set of smaller problems whose solutions can be recombined to get a reasonably good solution for the big problem, then:

- The computational effort can be split between many minds or computational units.
- A single mind can think about different parts of the problem individually, solving each one in turn.
- Any piece of the problem can be solved in isolation without needing to keep the influences of each subsolution on the entire solution in mind.
- Subsolutions can be reused if the same subproblem occurs in other problems.

The presence of these conditions creates an extremely favorable environment for cooperation in problem solving and discovery of general problem-solving knowledge. Linear systems, of course, satisfy all these conditions and are the best possible situation one could hope for.

An abstract structure that is amenable to decomposition and has been shown to encompass a wide class of optimization problems is that of matroids ([Papadimitriou and Steiglitz]). Define a subset system (E, S) as a finite set E along with a collection S of subsets of E . The elements of S are called independent, i.e. they are closed under inclusion: if S_1 belongs to S and S_2 is a subset of S_1 , then S_2 belongs to S . We can then

define the combinatorial optimization problem for the subset system (E, S) as follows: given a nonnegative weight for each element e of E , find an independent subset (i.e. an element of S) having maximum total weight.

Many common problems can be formulated in this framework, such as the Maximum Weight Matching Problem, a variant of the Travelling Salesman Problem, and the Maximum Weight Forest Problem (which is essentially equivalent to the Minimum Spanning Tree problem). As can be seen by the inclusion of a TSP-like problem in the list, some of these problems are intractable (as far as is currently known). However, there is a special class of subset systems which can be optimized easily.

Recall that the greedy algorithm consists of simply adding to the partial solution, at each stage of the optimization, the maximal weight remaining element which yields an element of S . Then a subset system is said to be a matroid iff the greedy algorithm yields an optimal solution for any instance of its combinatorial optimization problem. Of the three problems listed above, only the Maximum Weight Forest problem has a matroid structure. The Maximum Weight Forest (i.e. maximum weight acyclic collection of edges of a graph) can be found by simply repeatedly adding the highest weight edge that still yields a forest. (Note that by inverting the weights this problem can be shown to be essentially equivalent to the better-known Minimum Spanning Tree problem, for which the greedy algorithm can also be used.)

There are alternative characterizations of a matroid. Letting $M = (E, S)$ be a subset system, the following are equivalent:

- M is a matroid.
- If S_p and S_{p+1} of cardinality p and $p+1$ respectively belong to S , then $\exists e \in S_{p+1} - S_p$ such that $S_p + e$ is in S .
- If E' is a subset of E and I_1 and I_2 are maximal independent subsets of E' , then $|I_1| = |I_2|$.

The last property will be familiar from linear algebra (from which matroid theory can be formulated as a generalization).

Extending the basic notion of a matroid, some problems (such as the Bipartite Matching problem) can be thought of as the intersection of two matroids. A somewhat involved algorithm will solve these problems in time $O(|E|^3)$, i.e. in polynomial time. However, the intersection of three matroids is very likely not polynomial-time solvable; the existence of such an algorithm would give a polynomial-time solution method for the Directed Hamilton Path problem, which is NP-complete.

Matroid theory offers one characterization of problems that are decomposable; the greedy algorithm requires a constant amount of computational effort at each stage, no matter how large the problem is. What are some other universal axes along which to measure the decomposability of problem spaces?

One such axis can be found from a common heuristic method. Denote by k -OPT the optimization algorithm where, at each stage, all solutions within a "distance" of k from the current solution are checked and the one with maximal value becomes the new current solution. (A "distance" of k could mean, for example, differing by at most k edges.) The greater the average value of k necessary to find the optimum is, the less decomposable the problem (since a large value of k implies that local improvements do not suffice to solve the problem). Another is the number of epistatic connections, i.e. the average number of other variables which each variable depends on. (Perhaps a better approximate measure would be to count only variables for which a large change in value usually causes a large change in the state of the overall system.)

Instead of talking about the problem space directly, we could look at the algorithms used to solve the problems. As mentioned before, one of the primary virtues of a decomposable problem is that it can be split into simple subproblems that can be combined in a simple manner. This suggests the following ideas:

- Classify problems by the amount of workspace ("memory") that suffices to solve them.
- Measure the KC of the algorithm necessary to solve a problem. For example, what is the KC of the optimal (or a good) algorithm for the TSP? Note the difference from the way in which KC is traditionally applied; normally, we are looking for the minimal size program to generate a particular bit string, instead of the minimal size program that generates a certain function or suitable

approximation thereof. This is like asking for the KC of a program which solves a problem in a certain computational complexity class. If a bounded description suffices, the problem is doable in that class. (This also suggests extensions like the minimal size program to learn data, to function in an environment, and so on.)

There are many ideas from geometry that are applicable in the context of decomposability; simplicial complexes are a simple example. For n objects with some binary relationship defined for each pair, the corresponding simplicial complex is an n -dimensional polytope where vertices i and j are connected by an edge iff the corresponding objects have a "true" value for the relationship. Vertices with more edges incident to them correspond to objects which are related to many other objects, and which can thus be expected to be important members of the overall pattern (as well as being difficult to change without affecting many other parts of the pattern). Versions of such "connectivity graphs" are very useful in "data mining", which seeks to find relationships and patterns in large databases; the visual patterns of connected components make it easy to spot clusters of related objects.

Along with decomposability, hierarchy is also an important idea for making sense of complex systems. Empirically, many large systems (such as animals, galaxies, and organizations) seem to be organized into distinguishable "levels". The crucial property shared by these systems is that their behavior at a certain level (or resolution) can be largely understood in terms of the behavior of components at the next lower level. This

allows an approximate understanding of the behavior of each level to be gained with constant computational activity; it also promotes stability and ease of control. The fact that biological systems in particular are strongly hierarchical (with suborganisms at each level functioning fairly independently) is a strong hint that the processes of adaptation and evolution find the aforementioned benefits of hierarchy to be indispensable.

If the hierarchy is such that every level is composed of the same type of elements, then even an infinite number of levels may be tractable. An excellent case in point is that of fractals, whose "self-similarity" property allows many analytic results about infinitely detailed objects to be derived.

The traditional idea of a hierarchy corresponds to an inverted tree, where the children of each vertex correspond to the lower-level constituents of an object. But this sort of perfect separability is clearly a very special case; often the components of an object may overlap or be at different levels. How can the idea of a hierarchy be extended?

One proposal is the idea of hyperstructures which has recently been introduced in [Baas]. The hyperstructure incorporates definitions for levels, n^{th} order properties of interactions between these levels, and adaptation; it includes the traditional hierarchy and the hypergraph as special cases.

(Neg)entropy, Ecologies, Life and (Co)evolution

No discussion of complexity measures could be complete without mentioning entropy and biology. Let's look briefly at some avenues of exploration in these areas. Keep in mind that biological processes create and maintain the substrate in which our most complex systems live; as the only such robust and evolving complex system in existence, the biosphere deserves intense study (as well as dedicated protection, at least until its operation is fully understood and reproducible).

We have already explored entropy in the context of information theory. A related quantity with the same name is an essential idea in thermodynamics and statistical mechanics. It was derived in response to the empirical formulation of the classical Laws of Thermodynamics:

- 1) The energy of an isolated system is constant over time.
- 2) Heat cannot be transformed into work with 100% efficiency.

Together these laws rule out any sort of perpetual motion machine. (The first law was later generalized to a conservation law for mass-energy, by the mass-energy equivalence shown by Einstein).

The thermodynamic version of entropy essentially measures the inefficiency in the second law above, i.e. the amount of heat that is "wasted". For example, consider a thermodynamic system such as a steam engine. It is possible for a complete cycle of the

engine to return to the starting state without loss of energy if no work is done by the engine; this is called a "reversible process".

Any work done by the engine (such as pushing a piston) inevitably entails some inefficiency, so that when the engine returns to its original point in the cycle the sum of its energy and the energy used to perform useful work is less than the original amount; this irreversible loss of energy is measured by the thermodynamic entropy.

Two problems with thermodynamic entropy suggest themselves: finding a convincing theoretical explanation for the validity of this empirically derived concept, and explaining why the Second Law implies irreversibility of processes that do work when all the equations of physics seem to be time-reversible. Boltzmann addressed both of these with his formulation of statistical mechanics, which explained the macroscopic quantity of the entropy of a system in terms of the mechanics of the microscopic molecules comprising the system.

Essentially, a macrostate of a system (such as a certain temperature) can be realized in many different specific ways (corresponding to all the different combinations of velocities of molecules which yield the given temperature), each of which is called a microstate. With W denoting the number of microstates for a given macrostate, the Boltzmann entropy H is then given by $H = k \ln W$ (where k is the Boltzmann constant).

This formulation explains why so many everyday processes seem to be irreversible, e.g. why mixing hot and cold water in a pot gives a lukewarm result, but the lukewarm water never spontaneously differentiates itself into hot and cold segments. The number of microstates (and hence entropy) corresponding to a mixed (lukewarm) state is far greater than the number of microstates for a differentiated state; assuming "random" physical interactions, it will therefore be far more likely for the system to go from differentiated to lukewarm than vice versa, and to stay in that macrostate once it is reached.

The Second Law is thus true with overwhelming probability (though there may be extremely infrequent fluctuations). Considering also the possible gain or loss of entropy from the external environment of a system, the Second Law can be reformulated: "Entropy production inside a system is greater than or equal to the entropy loss to the system's surroundings".

Boltzmann entropy is thus a measure of the "nonuniqueness" of a system; low entropy systems are in very unusual states. (It is often claimed that Boltzmann entropy measures "disorder"; this is not at all clear, since "order" is almost as nebulous and multifaceted as "complexity".)

As an intriguing possible application involving the entropy concept, consider that from our discussion above a reversible system can operate without dissipating heat.

Surprisingly, realistic models of reversible computation have been devised; these hold the promise of building extremely small scale computational elements, breaking through the heat dissipation barrier that is a major constraint on present-day computer design and allowing the development of nanotechnology (nanometer-scale engineering). This possibility is discussed in [Zurek], along with a formulation of the entropy concept which uses Kolmogorov Complexity.

From the time the concept of entropy was first devised, many people have explored the connection between it and the evolution of life. The main paradox is that the Second Law seems to imply that disordered states will arise with probability one; how, then, can we explain the presence of so many ordered structures in the universe (most notably our own biosphere)? The answer seems to be that most structures, such as the Earth, are not closed systems; there is a constant flow of energy into and out of the system. Thus, the Second Law applies to isolated systems and (perhaps) to our universe as a whole; it is not applicable to subparts of isolated systems which receive and transmit energy to the rest of the system.

Such open systems which have both internal and external sources of entropy production maintain their structure by dissipation and consumption of energy. They have been called "dissipative systems"; see [Nicolis and Prigogine]. Many nonliving examples of such systems (such as clouds, solitons, vortices and solar systems) have been explored.

In addition to further exploring such inanimate systems, an issue which is of great interest is to extend such theories to try to answer questions involving the emergence of life:

- What are some plausible mechanisms for the first reproductive processes to have arisen?
- How could such initially simple reproductive processes have increased their complexity over time?
- How probable are the above two operations (the arising of life and its increase in complexity) in different surroundings? (Do most planets have some form of life on them? How about stars?)
- Are there simple parameters explaining the probabilities of the previous question? (Can we estimate a few quantities for a system and plug them into a formula which returns the estimated time to intelligence?)

Until the discovery of extraterrestrial life, exploring life on Earth is one of only two current options for learning universal principles underlying the evolution of highly complex systems. (The other is the development of generalized computational intelligence as is being explored in the field of Artificial Life, which we'll discuss briefly later.)

A fascinating introduction to themes of biological evolution can be found in [Cohen and Stewart] and [Dawkins]. The author believes the traditional Darwinian view

of evolution is probably comparable to the Greek theory of epicycles; correct in its most basic form, and extendable in a convoluted way to approximate more complex cases, but soon to be superseded by a more generally applicable and correct theory. (See [Kelly] for an accessible argument in favor of this view.) Some dynamical results on evolution as an optimization process are in [Kauffman]. Understanding the utility of features in traditional Darwinian evolution such as gradient descent and sexual evolution is being studied intensively in computational learning theory.

One central subgoal in explaining evolution is to find a satisfactory framework in which to talk about coevolution. Formally, we can say coevolution is occurring when part of the definition of the fitness of organism A involves the fitness of organism B, and vice versa; when both these conditions are met, an evolutionary "arms race" often takes place where an increase in fitness of one organism prompts the other to improve itself, and so on in a feedback loop. (For an example of coevolution, see the beginning of the discussion on Genetic Algorithms in the section on Generalized Intelligence below.)

The definition of coevolution raises two general questions. First, how do we define exactly what the coevolving entities are? Even in the case of biological organisms, the presence of hierarchies of living systems (cells, organs, animals, societies) makes the problem very pertinent. In a general context, one could consider any heterogeneous group of these--including even the inanimate environment--to be the "organism" that is coevolving. The only requirement is that the formal definition above be satisfied, i.e. that

the "organism" be coherent enough that a consistent fitness measure can be defined on it. When considering coevolution in a computational process, the problem is even more severe, since the "computational organism" may not be predefined. The most challenging part of understanding the evolution of biological or computational processes may be to decompose them meaningfully into suborganisms. Note the strong relationship between decomposability and coevolution.

Second, how is a more general coevolutionary situation to be modelled? The case with two organisms is of course the simplest possible; population biologists and ecologists struggle with "food webs" showing interrelationships between dozens or hundreds of species. Some sort of hierarchical structure such as a dynamic hypergraph might provide an appropriate conceptual framework.

Note that many ecological features can be understood as special cases of coevolution. An example is parasitism, where one organism increases its own fitness by "stealing" the fitness of another organism. If the parasite is dependent on the host, then the two can be considered as a system where one part raises its fitness by lowering the fitness of the other part. The parasite will generally not weaken the host too severely unless it can easily find a new host, as this would impact on its own fitness. (This explains the relative paucity of truly virulent diseases, as they tend to "burn themselves out".) Contrast this with symbiosis, when both parties raise each other's fitness.

Understanding coevolution may point the way to a broader theory which explains the reason for the hierarchical patterns of life. Why do multicellular organisms exist? How can they survive the virulence of the attacks which our immune systems are constantly exposed to, and the reproductive and adaptive efficiency of small organisms such as bacteria and viruses? The loss in efficiency of cooperating colonies of organisms (e.g. multicellular organisms) must be more than counterbalanced by the improved resources (both algorithmic and physical) that they can bring to bear on achieving their goals.

Another fundamental reason for the decomposability of organisms into smaller and smaller functional parts is simply that it is much faster to use already existing subsolutions to make a solution than to design a new solution entirely from scratch. Explaining why and when this is true is a very deep problem--not just in evolutionary biology, but in any situation in which learning or design takes place.

Parallel and Emergent Computation

Parallel computation has recently become an active area of study. The primary motivation for studying parallel computation is to increase the speed of computation; if many processors can cooperate on solving a problem, the running time of algorithms can be speeded up tremendously. (Indeed, it currently appears that there are fundamental limits on the speed of computation that a single processor can achieve, such as a minimum size for computational elements. Once these barriers are approached, the only avenue that remains to increase processing speed is to use many processors in tandem.)

Although parallelizing algorithms can provide increased speed, for many difficult problems this speedup will not be enough to make them tractable. Assuming that NP is not equal to P, any NP-hard problem will require an exponential amount of time to solve via a sequential algorithm. A sequential computer can simulate a parallel one, so $(\text{sequential time}) = (\text{parallel time}) \times (\text{\# of processors operating in parallel})$; this implies that a parallel algorithm for an NP-hard problem would require exponential parallel time or an exponential number of processors, neither of which is feasible.

This suggests the sequential complexity class P as a class of problems that one might profitably try to find parallel algorithms for. A fundamental question immediately springs up: which algorithms in P are parallelizable? Do parallel versions of all algorithms in P exist, or are some of these algorithms inherently sequential?

It currently appears that the latter situation holds; there is a class of problems in P (called P-complete problems) such that no parallel algorithm has been found for any of them, and a parallel algorithm for any problem would imply a parallel algorithm for all problems in P. As we will see shortly, this question can be formulated in terms analogous to the $P \stackrel{?}{=} NP$ question.

In order to study parallel computational complexity, a model analogous to the Turing Machine must be defined that incorporates parallelism. One such model is the PRAM, which consists of a sequence of n RAMs (where each Random Access Machine is a model of computation equivalent to the TM but with more natural programming instructions, similar to machine language). Each RAM contains its own program, which runs concurrently with all others.

There is a common memory space which can be read from and written to by all RAMs. Since the RAMs run their programs independently, it is possible that more than one RAM will try to write to the same location at the same time step; in this case, a convention (such as allowing the RAM with the lowest number to succeed) is used to determine which write attempt is successful. A read or write operation takes one time step. (This is unrealistically powerful as any implementable computer would require at least logarithmic time in the worst case to access an arbitrary memory location.)

The number of RAMs used is a function $f(a,b)$ of the number a of input integers

as well as their total length b . (This is done so that the degree of parallelism may be tailored to the size of the problem. Again, it is infeasible to require an arbitrarily large number of processors to be available on an actual computer; however, a computer with a fixed number of processors can simulate one with more processors fairly efficiently.)

Although the PRAM seems overly powerful, it is useful for three main reasons (besides its simplicity). First, the power of the PRAM seems to be an upper bound for that of physically implementable computers; problems that a PRAM can't do efficiently are probably unparallelizable even in principle. Second, the PRAM has been shown to be roughly equivalent to other models of computation which are seemingly more realistic, such as Boolean circuits. Third, it has been found in practice that most PRAM algorithms can be translated into algorithms for real computers without too great a loss in efficiency.

The parallel complexity class NC can now be defined: it consists of those problems which have a PRAM algorithm which takes time $O(\lg^a n)$ and uses $O(n^b)$ processors for some fixed integers a and b (where n is the size of the input, which is usually proportional to the number of input integers). In other words, the parallel algorithm for the PRAM must execute in polylogarithmic time on a polynomial number of processors. (The restriction to a polynomial number of processors is to avoid absurd cases where an exponential or infinite number of processors can be used to solve a problem in constant time.)

The original motivation for the definition of NC was that many natural problems in the sequential complexity class P were found to have polylogarithmic execution time when parallelized. Noting that $\lg^a n$ is asymptotically less than n for fixed a , and that a sequential computer can simulate a PRAM in time proportional to the number of RAMs in the PRAM, it is clear that NC belongs to P. The question, then, is whether NC is equivalent to P. Remarkably, the definition of NC appears to have been the correct one; despite intense effort, the equivalence of NC and P is an open question.

If NC and P are indeed equivalent, then every problem in P (i.e. every problem for which a "feasible" sequential algorithm exists) can be massively parallelized. Analogously to the P vs NP question, the problems which are least likely to be in NC and thus parallelizable are those which are complete for the class P. (A problem A is P-complete if a polynomial-time solution for A can be used to define a polynomial solution to every problem in P; the presence of such a problem A in NC implies that every problem in P is also in NC.)

The following is a selection of P-complete problems (see [Greenlaw, Hoover and Ruzzo] for a comprehensive reference):

- Max Flow
- Linear Programming
- Gaussian Elimination
- deciding whether a Context-Free Grammar gives rise to an empty language

- deciding whether a graph has an induced subgraph of vertex connectivity at least k (where vertex connectivity is the minimum number of vertices whose deletion disconnects the graph)

Note that, in contrast to the sequential case, there is no known generally effective method for efficiently converting a decision problem into the corresponding search problem (e.g. actually finding the induced subgraph in the last problem above). (Recall that the method of binary search outlined in the section on Computational Complexity is inherently sequential; each computational step directly affects the next computational step to be taken.)

Although the class NC given above is not nearly as well-accepted a definition of efficient parallel computation as P is of efficient sequential computation, the wide variety of natural problems that fall in it lend credence to this belief. In fact, there is another fascinating result involving NC called the "parallel computation thesis"; this states that parallel time is roughly equivalent to sequential space. In the context of NC, this conjecture can be formalized and proven: it can be shown that any problem which is in NC (and can thus be solved in polylogarithmic parallel time) can be solved by a sequential TM in polylogarithmic space.

As a final note on parallel algorithms, consider the following: even when fast parallel algorithms are available for a problem, the best known parallel algorithm often bears little resemblance to the best sequential algorithm for the problem. It is not simply

a matter of having many processors perform a subset of the sequential algorithm; the parallel algorithm often decomposes the problem in fundamentally different and counterintuitive ways. One major reason is that an efficient parallel algorithm must ensure that all processors are constantly busy, and that solutions to subproblems are found just before they are needed as input to other subalgorithms.

Emergent computation refers to computation where the algorithm is not explicitly specified in advance. Instead, a simple set of rules is given which results in a complexity of behavior which, to an independent observer ignorant of the actual rules used, would appear to have been caused by a very complex set of rules. These rules are usually executed in parallel by many interacting processors. (See [Forrest] for an overview of many research efforts along this line; [Kelly] is a highly readable informal introduction.)

An excellent example is cellular automata, which will be discussed in the next section; an infinite grid of processors performing extremely simple computations gives an unpredictable, evolving configuration of mesmerizing complexity that appears almost "alive". In the classic book "Godel, Escher, Bach", the ant colony is vividly described as another instance of such emergence ([Hofstadter 79]).

Consider also the current paradigm of particle physics, which sees the current state of the universe as the result of time-evolution of an initially simple state under a small set of universally-applicable rules. Although the "rules of the universe" are specified at

the level of particles, with the passage of time robust computational structures have evolved on levels many orders of magnitude larger than that of the individual particle. What is there in the initial "rules of the universe" that would tell a pre-Big-Bang observer of the potential inherent in the system about to be unleashed?

Another characteristic of emergent computation, as suggested by the previous examples and the section on Decomposition and Hierarchy, is that the computation occurs at many "levels". Measuring or detecting higher level computation and describing when it is likely to arise are very interesting questions. This is an especially interesting question when the algorithm doing the detection is itself part of the computational system (such as humans explaining societies or the universe).

At first, the sequential TM model which has a well-defined input and output seems quite different from emergent computation, where the computational activity perceived depends on the intelligence of the observer. Aside from possible inherent differences between sequential and parallel computation as discussed previously, this may be largely illusory. A "more intelligent" observer can discern patterns in the operation of a TM, and is thus capable of perceiving its computation in terms of higher-level operations; people think about programs at a much higher level than individual statements. (Crutchfield essentially uses the ability to discern patterns in a static record of such computation as his measure of complexity, as discussed in the previous section on Statistical Complexity.)

Cellular Automata

As an example of the evolution of complex behavior from simple rules, cellular automata are second to none. First introduced by Ulam and von Neumann in the 1940's, the field was relatively quiescent for a long period; with the advent of widespread computing power, they have re-emerged as a very active research area.

A cellular automaton ("CA") consists of a discrete lattice of cells, each of which has one of k possible values assigned to it. A neighborhood is defined for each CA, which is simply some set of cells relative to the cell whose neighborhood is being taken. For example, there are two neighborhoods that are used especially often in rectangular 2-dimensional CAs:

C	C	C			C
C	C	C		C	C
C	C	C			C

These are the Moore and von Neumann neighborhoods respectively. In both cases, all cells are in the neighborhood of the center cell.

The value of each cell is updated synchronously in discrete time. The value of a cell at time $t+1$ is a fixed function (called the update rule) of the values of its

neighborhood cells at time t . Usually the update rule is deterministic, but it can be stochastic as well.

Of the set of states in which a cell can be, one is usually distinguished as being the quiescent state; if the neighborhood of a cell contains only quiescent states, the cell will be quiescent after the update as well. At any time, the states of all cells are collectively referred to as the configuration of the CA.

Note from the definition that the rules governing a CA are stationary with respect to space and time; the same update rule applies everywhere. Since the neighborhood is always finite (and usually quite small) the CA possesses localized dynamics.

In many respects, CAs are the antithesis of TMs. The computation of a CA is completely parallel versus the completely sequential case of a TM, so that a CA can (theoretically) update a configuration of infinite size in unit time. The CA incorporates a minimal set of rules, leaving the bulk of the complexity to arise in the "workspace"; in contrast, the TM incorporates arbitrarily difficult functions in its program.

CAs with very simple rules can display extremely complex behavior, such as the "Life" CA invented by John Horton Conway. Most nontrivial CAs are capable of embodying a UTM, and can thus calculate arbitrary computable functions. It is thus not

surprising that an analogue of Rice's Theorem (from recursive function theory) has been shown to hold for the limit sets of CAs. (Incidentally, the proof of this analogue and the proof of the original Rice's Theorem are quite dissimilar and do not seem to be reducible to one another.)

Defining the limit set of a CA as the set of all those configurations which can arise after arbitrarily long computations, the following has been shown in [Kari]: every nontrivial property of the limit sets of CAs is undecidable, even if only 1-dimensional CAs are considered. (A nontrivial property is one which is not either true for all CA limit sets or false for all CA limit sets.) Thus, for any nontrivial property, no computable algorithm exists that can decide if the limit set of an arbitrary given CA has that property. Interestingly, the proof shows that the property of nilpotency--deciding whether all configurations of a CA eventually reach the same limiting configuration--is the easiest such property, in the sense that a computable algorithm for any nontrivial property would imply a computable algorithm for nilpotency.

To illustrate the algorithmic power of CAs with simple rules, consider the problem of counting connected sets of "on" pixels in a monochrome image. A traditional sequential task to perform this task would be quite involved. In contrast, a simple CA can be used to solve the problem ([Smith]). Let the value of a pixel be 0 or 1 according to whether the pixel is off or on, and denote the value of pixel (i,j) by $a_{i,j}$ (where i measures horizontal distance and j vertical distance from the lower left corner of the

image). Defining the step function $S(x)$ on nonnegative integers as 0 if x is 0 and 1 otherwise, the CA update rule can be written as:

$$a_{i,j} \rightarrow S(S(a_{i,j-1} + a_{i,j} + a_{i+1,j} - 1) + S(a_{i,j} + a_{i+1,j-1} - 1))$$

This update rule is executed in parallel for all cells. It has the effect of "shrinking" connected components until only a single isolated pixel remains at the upper left corner; at this point, a counter is incremented and the pixel removed. Every connected component will have been counted after at most $\max(i) + \max(j)$ steps.

Many measures on CAs have been devised to estimate their complexity. Two of the simplest ones are the lambda parameter and the Z parameter; the lambda parameter measures the proportion of neighborhood configurations which do not map to the quiescent state, and the Z parameter gives the probability that a given cell in the possible preimages of a given configuration is uniquely determined. Both are discussed in [Wuensche] and explored in the context of 1-dimensional CAs.

In [Wolfram] four basic classes on CAs are defined, according to their asymptotic behavior: limit point, limit cycles, chaotic aperiodic dynamics, and nonstationary localized structures. (The last class contains those CAs which generate robust and coherent patterns, corresponding to the intuitive notion of "self-generated complexity".) This book contains most of Wolfram's work on CAs, and is an excellent wide-ranging introduction to many fundamental problems in the field.

Another interesting measure which the author has thought of would be an analogue of Kolmogorov Complexity for the CA. Formally, for any given non-quiescent configuration, this "CA KC" would be the minimal size preimage that generates the state A corresponding time-bounded CA KC can be defined in the obvious manner. (The "size" of a configuration could be measured as the number of non-quiescent cells or as the area in cells of an enclosing rectangular hyperprism.)

How does this contrast with the "TM KC"? Instead of expressing functions in terms of a standard computer language, we are treating each CA as a separate dynamical system; the CA KC of a given configuration for a given CA is thus an exact quantity. The logical next step is to define an "absolute" CA KC for a given configuration across all CAs. This corresponds to finding a CA dimension and update rule that gives a "simple" or small description of the given configuration; in other words, a plausible parallel algorithm for generating the configuration. Such an "absolute" CA KC requires construction of a "Universal Cellular Automaton" (UCA) which can simulate any CA; this would allow an invariance theorem such as in the usual TA KC.

Note that there are reversible CAs which are capable of universal computation. In such a CA, deciding whether the CA KC is less than a given bound is a recursively enumerable (but not necessarily recursive) problem. This is so since we can just run the CA in reverse and keep track of the minimal size configuration that arises. As per the requirements for a problem to be recursively enumerable, this method will tell us in finite

time if the CA KC is below the given bound but may continue forever otherwise.

As an example of a configuration with nontrivial low CA KC, consider the well-known "Acorn" configuration in Conway's Life CA; the Acorn is a configuration of 5 "on" cells which grows to a phenomenally large (but finite) number of "on" cells. Could there be any conceivable way of guessing that this large and complex result grew from such a tiny "seed"?

An important construction requirement for any complex system is the "persistence of information"; information must last longer than the time required to make use of it. This suggests the question: can a noisy CA (i.e. one with a given probability of random updates to a cell) simulate a quiet one at a larger scale? Some form of "error-correction" is required, such as exists in coding theory.

A first step in this direction has been taken in [Gacs and Reif], where an arbitrary unreliable 1-dimensional CA is shown to be efficiently and reliably simulable by a 3-dimensional CA. However, this construction uses simple repetition to achieve the error correction, as well as additional dimensions; of greater interest would be an error-correcting CA with no loss of dimensions and an emergent error-correction scheme which is not explicitly specified by the CA update rule. This line of thought suggests the larger question of exploration of the general principles used by robust structures to maintain their coherence in the presence of random or hostile noise.

Generalized Intelligence

It could be argued that any reasonable measure of complexity should assign an extremely high value to systems which display intelligence. But "intelligence" is notoriously difficult to define precisely; in its usual human substrate, it is also difficult to experiment with and modify. Recently, many formal algorithms have been developed which seem to capture some properties of "intelligence". They display the capabilities of generalizing principles from data, adapting to changing situations, and even progressively increasing their computational abilities.

As an example, we will look at Genetic Algorithms, which are computational procedures which borrow their basic principles from natural genetics (i.e. the genetics of living organisms). The idea behind them is to follow the same ideas that nature has used so successfully; evolution has produced highly complex organisms which are capable of surviving in a variety of dynamic and challenging environments.

Consider an example drawn from natural selection: a population consisting of lions and deer. Some of the deer will be better equipped to escape the lions (through being faster, having better eyesight or hearing, and so on) and will therefore tend to survive longer; such deer will have a greater chance of producing offspring. The progeny of each pair of deer will inherit properties from both of their parents; on the average, they will be more fit than the previous generation, since their parents had enough survival

value to last until breeding time. There will also be occasional random mutations which will introduce some variability to the population. We thus expect that, due to this classical "survival of the fittest" principle, the deer will become more and more fit.

Of course, a similar process is also happening with the lions, since the evasive improvements of the deer will allow them to escape the slower or less determined carnivores. This type of escalating "arms race" occurs in many diverse situations involving direct competition.

Keeping this theme in the background, we can now formalize the properties that a genetic algorithm ("GA") for a particular problem must have:

- a genetic representation for solutions to the problem
- a method for generating an initial population of solutions
- a fitness function which evaluates and ranks the quality of the solutions
- genetic operators which alter the representation of the offspring during the reproductive stage
- values for parameters such as the population size and the probability of applying genetic operators

Consider a GA for Axelrod's well-known Prisoner's Dilemma, which is simple to state but difficult to analyze. Two prisoners are accused of a crime and held in two

separate cells, unable to communicate with each other. Each has the choice of confessing evidence against the other in the hope of receiving clemency; giving evidence is termed "defection", and staying silent "cooperation".

If only one prisoner defects, that prisoner is set free and given a large reward, and the other is executed. If both defect, they are both imprisoned. If neither defects (i.e. they both cooperate) they are both set free. Thus, for each prisoner, the choice of defection always yields a higher reward than the choice of cooperation; however, if both defect then they do worse than if both had cooperated.

We can consider repeating this procedure many times; in this case, the action of each prisoner will be based on the previous actions of the other prisoner. This is called the "Iterated Prisoner's Dilemma" and can be played as a game between two players, where during each turn each of the players chooses to defect or cooperate and then receives a reward accordingly:

<u>Player 1</u>	<u>Player 2</u>	<u>R₁</u>	<u>R₂</u>
Defect	Defect	1	1
Defect	Cooperate	5	0
Cooperate	Defect	0	5
Cooperate	Cooperate	3	3

In order to use a GA to find good strategies, we will need to keep a population of players, each of which has a particular strategy. These strategies will initially be chosen at random. At each iteration (or turn) afterward, the players will play the game with each other; the ones which accumulate greater rewards will be favored for survival and mating. When two players mate, the new player created will have a strategy which is combined from its parents' strategies via the genetic operators.

The essential parts of the genetic algorithm may be set up as follows:

Genetic representation: We can use a simple deterministic set of strategies; the current move will be based strictly on the previous three moves. Since each move has four possible outcomes, this gives a total of $4 \times 4 \times 4 = 64$ possible strategy histories.

For each of these histories the algorithm will either defect or cooperate; this set of actions can be represented as a string of 64 bits with a 0 in a position indicating the algorithm chooses defection if the corresponding history just occurred, and a 1 indicating the algorithm chooses cooperation. An additional $1 + 4 + (4 \times 4) = 21$ bits can be used to determine the strategy at the beginning of a game, for a total of 85 bits. Our population is thus represented by a set of 85-bit strings.

Initial population: A set of random 85-bit strings.

Fitness function: During each round, we can let each player play with every other player for a certain number of rounds, and keep track of the cumulative score of each player. The higher this score is, the more "fit" the strategy.

Genetic operators: The two standard genetic operators are crossover and mutation. Many other possibilities exist, however.

If we have two players P_1 and P_2 represented by bit strings, the crossover operator applied to them chooses a position at random, as follows:

$$P_1 = A.B \qquad P_2 = C.D$$

where A,B,C,D represent partial strings and the dot represents the position chosen at random. Two new players can then be generated by exchanging the partial strings:

$$P_3 = A.D \qquad P_4 = C.B$$

The crossover operator thus reorders subsections from old players to make new ones. In this way, portions of effective strategies are combined to explore new alternatives.

The mutation operator introduces some random variability into the population. It randomly changes some of the bits of a player to make a new player.

Values for parameters: These are arbitrary; different values for the population size, probability of genetic operators and other parameters will affect the strategies

that are found.

We have now specified the complete genetic algorithm. When this algorithm is run, our initial random population of strategies competes and produces increasingly better strategies. We expect that after many generations the strategies in the population will be of very high quality; experimentation verifies our intuition.

Why does the GA work? The two reproduction operators provide the answer. Mutation allows for random forays into new areas of the search space, while crossover allows good subsolutions to be combined to yield a more highly fit individual. There are many intuitive descriptions of why the GA works in results such as the Schema Theorem, which states that under certain assumptions subparts of above-average fitness will tend to increase in prevalence; [Michalewicz] provides a lucid introduction with many applications.

It is important to note that the GA only works well on a small class of problems. (Characterizing this class is a very interesting open problem.) This is not surprising; in [Belew and Hart] it is shown that the problem that the GA attempts to solve is NP-hard, even if the requirement is relaxed from finding the global maximum to finding a value guaranteed to be close to the maximum. Some qualities of problem spaces that make them difficult for the GA to solve are known, such as high epistasis. (If the overall fitness of an organism depends on the values of a set of characteristics, then epistasis is

the average number of characteristics whose values affect the fitness contribution of a single characteristic. Epistasis thus measures nonlinearity; zero epistasis corresponds to the case where the overall fitness of an organism is just a sum of values of its characteristics.)

It may seem that the algorithm as presented is too simple to solve any real problems. This is usually the case; to apply the basic GA model above, the following domain-dependent work must be done:

- Tweaking the parameters.
- Creating tailor-made reproduction operators. For example, the crossover operator given above cannot be used when applying the GA to find TSP solutions, since it would not generate valid tours.
- Finding features of the problem that can be used to define a good search space. This consists of choosing the properties of a solution that each place in the bit string will represent. In general, discovering important features of solutions is an extremely difficult problem (not just in the GA but in all learning algorithms).

There are also many extensions of the GA model which incorporate additional heuristics; a recent survey is [Proceedings 5th ICGA]. Building the ability to solve problems hierarchically into the GA is one particularly interesting extension, for reasons discussed earlier in the section on Decomposition and Hierarchy.

3

OF/DE

3

PM-1 3 1/2" x 4" PHOTOGRAPHIC MICROCOPY TARGET
NBS 1010a ANSI/ISO #2 EQUIVALENT

PRECISIONSM RESOLUTION TARGETS

One crucial feature that the GA does not incorporate is the ability to evolve its own operation. The framework needs to be extended so that both the solutions found and the method used to find solutions become more effective over time. Creating such an efficient open-ended learning algorithm is essentially the Holy Grail of Artificial Intelligence.

Note that the reason GAs (only one of an infinity of possible methods of distributed computation) seem intuitively to be important is that they're almost the least complex evolutionary method imaginable. The strategy used is so compact in description that no element can be removed while still retaining a viable algorithm (except for crossover, in which case a simple random search algorithm results).

They also incorporate the least possible feedback from the environment. If the individual does something wrong, it dies if its fitness is below a certain threshold and doesn't if its fitness is above the threshold. There are no clues as to what was done wrong, or how wrong it was--thus, the "teacher" (the environment) is giving the least amount of information possible to the "student" (the algorithm). Causating factors (i.e. which actions were helpful and which harmful) all have to be reverse-analyzed from this one ultimately blunt binary lesson.

Also note that GAs are a (very) simplified implementation of the operating rules of biological evolution. Since we and most of the rest of the complexity of our

environment appear to have arisen through operation of these biological rules, the experimental evidence is certainly interesting enough to warrant their investigation.

A very interesting application of the GA is in Genetic Programming ("GP"), where the "individuals" that are genetically optimized are themselves actual programs ([Koza 92 and 94]). The fitness function by which programs are now being measured is their capability to perform some given task. Programs that perform well are altered and recombined, and the new programs then tested, until some fitness level is reached.

There is an interesting connection between GP and the previously considered extended KC measures for estimating simplicity, speed, decomposability, and other such properties. When the programs in GP are evaluated, they are considered as "black boxes"; only their ability to solve the assigned task affects their fitness. But we may well wish to consider qualities of the black box itself; if a short program performs as well as a longer program, the former is clearly to be preferred.

If estimators could be developed for the size, speed, and decomposability of a program (or function), they could play an important part in measuring the fitness of programs. Just as the Stochastic and Statistical Complexity approaches stress, the complexity of the model used to solve a problem should be minimized while its performance is maximized; these two conflicting constraints amount to tradeoffs between "elegance and generality" and "problem-specific performance".

Perhaps the most exciting aspect of GP is that programs of arbitrary complexity can theoretically be evolved, as opposed to most learning paradigms where the solutions are found from a restricted computational class of models. The problem, of course, is that the speed of evolution in the basic GP model is prohibitive for quite small sizes of programs. There is again a tradeoff: adding such methods as mutation, recombination, and the collapsing of subprocedures into new primitives seems to increase speed of evolution with respect to a smaller and smaller class of problems. The challenge is to abstract the most generally applicable principles of learning into a "seed" program, that can bootstrap itself indefinitely by finding better and better learning methods.

Many other learning algorithms are being explored; see [Hutchinson] for a wide-ranging introduction. We have discussed many aspects of learning peripherally in earlier sections, since measuring the difficulty or complexity of a problem is a large step toward solving it; one highly relevant link is the rigorous theory of machine learning being developed which uses computational complexity as its basis ([Natarajan]).

Parallelism is an essential feature of biological brains. The individual computational elements of our brains (i.e. neurons) operate at very slow speeds, but the cooperation of billions of such elements produces a "neural computer" that still outperforms the fastest supercomputers on many cognitive tasks. In [Minsky], an analogy is drawn between the mind and a society; the brain is parallel not only at the "atomic" level of neurons, but also at the level of cognitive "subminds" which each focus on

certain tasks and cooperate to produce the illusion of a single "I".

The importance of "algorithms of intelligence" is that they formalize parts of what we think goes on in our brains. Another productive method of exploring thought is to try to formulate theories of how brains operate directly. The "biology" of thought processes is explored in the field of cognitive science, while the "chemistry" of thought is explored in neuroscience.

If life is a property only of patterns and interaction in material instead of the material itself, then it is not restricted to the usual carbon-based biological building materials. Any substrate capable of supporting the necessary patterns can contain life--and mind.

The nascent but rapidly growing endeavor of Artificial Life takes this proposition as an axiom, and is using many of the ideas discussed in this thesis to try to construct lifelike patterns (see [Levy], [Langton 90 92 and 94] and [Brooks and Maes]). As another consequence of this proposition, the "patterns" of intelligence may take place at the level of groups of organisms; in [Stock], the idea of human society as a metaorganism is explored.

Metahuman Institutions

Economics. Law. Politics. Business.

These are the glue that bind together our civilization--the institutions that collectively generate the rules under which we all operate. Notice that they share a common theme: reconciling the inevitable clash between individual goals. None of them can be decomposed; situations involving one or two individuals are qualitatively different from those considered at the usual level of discussion. They form a "metaknowledge" whose ideal purpose is to more efficiently facilitate "the greatest good for the greatest number".

The Prisoner's Dilemma that we saw in the section on Generalized Intelligence has been widely used as a metaphor for the evolution of cooperation. In order to sustain cooperation, it is necessary for the individual to be willing to suffer a short-term loss in exchange for long-term gain. As the systems in which organisms operate grow more complex, it requires greater algorithmic power (called "superrationality" in [Hofstadter 85]) to realize that an action which seems disadvantageous can actually be beneficial in the long run. The required level of thinking may include consideration of the goals and strategies of other organisms, which leads to an interesting infinite regress if they are also superrational.

Besides expanding the time within which payback for sacrifice is required, organisms also display altruistic behavior when their definition of "self" is enlarged. An individual may consider itself part of a larger group, composed either of particular individuals or of a set of ideas which represents a class of individuals; in this case, the individual relaxes the requirement of payback to include payback to the group.

Intuitively, a greater presence of superrationality and an enlarged sense of self in a population correlates with more cooperation in the population. This in turn will increase the quality of the theoretical global maximum compromise that can be achieved for the population.

The "metaknowledge" embodied in law, politics and economics exists at many different levels (individuals, groups, groups of groups and so on). One of the basic challenges facing our civilization is to improve the quality of our metaknowledge, and hence of the institutions and methods we use for generating such metaknowledge.

As an example of the issues involved, consider economics. One of the most interesting and longstanding debates is over the merits of free markets versus planned economies; the twentieth century has generated a great deal of evidence in favor of the former. There are two basic problems facing a planned economy: the bounded rationality (or algorithmic power) that the planners have available to form strategies to meet their goals, and the lack of data that the planners have about the goals of the population at

large and the condition of factors influencing the economies. To this may be added an important third problem which makes the design of any human-run algorithm much more difficult: the tendency of individuals to pursue their own goals instead of working toward their assigned goals.

The first two problems are present in any system in which a small fraction of the total computational resources plan and control the entire system. Free markets alleviate these problems by allowing all individuals some latitude to pursue their own goals; the third problem is at least ameliorated, since the free market system takes as a given the desire for individual gain. The "parallel emergent computation" thus formed--the famous "invisible hand"--has usually proven to be more robust, flexible, inventive and efficient than the alternative; [Rothschild] uses biological metaphors in a compelling explanation of the emergent computational properties of economies. Understanding when and why free markets are better parallel algorithms than planned systems is a major challenge.

For instance, how much "structure" (in the form of laws, government regulations, and nonmonetary institutions) will produce a free market that gives the greatest benefit to the greatest number of its participants? Constraining markets (via techniques such as environmental protection, arbitration rules, and labor laws) may actually have the effect of providing more freedom of action to market participants, by freeing up energy that would otherwise have been spent in nonconstructive (or non-superrational) activity.

The "tragedy of the commons", or the tendency for individuals to overuse or destroy common public goods, is an economic analogue of the Prisoner's Dilemma. The name refers to an illustrative scenario in which herdspeople graze their livestock in a common pasture; each herdsman has a strong incentive to allow his or her personal livestock to graze more and become better-fed, and a very weak incentive not to. The end result is that the pasture is overgrazed and all are worse off than if they had refrained from shortsighted behavior. The problem is again seen to be a lack of superrationality.

One very important issue is simply to find tools that measure how well an economy is doing, as discussed in [Henderson]. The archaic use of GNP as a measure of the health of an economy is equivalent to using the total number of instructions processed by a parallel algorithm as a measure of how well the parallel algorithm is performing its assigned task. It would not make sense to count computational activity that is useless or even counterproductive as a gain; we do not want processors simply to be busy, we want them to be constructively busy.

It is thus illogical to consider only economic activity; what must be measured and maximized is constructive economic activity. This of course requires some concept of constructive activity which is reasonably widely accepted; a first step is the recent creation of goal-based formulations that value activities increasing literacy rates, lifespan, health levels, income relative to the cost of basic goods, and so on.

A common task facing metahuman systems is to incentivize properly. What seems to occur again and again throughout different societal institutions is that a decision-making agent assigns subtasks to different subagents, which if completed will contribute toward the overall task. However, it is natural for individuals and subsystems to pursue their own immediate incentives; the subtasks take on a life of their own after they have outlived their usefulness. As well, subagents often attempt to maximize individual gain.

The solution is to match the subsystem incentives with the overall ones, via such tools as performance bonuses and accountability measures. Note that this problem also arises whenever designing a computational algorithm that uses subalgorithms to get partial results; the subalgorithms must be chosen to give relevant and constructive results, and modified or changed entirely when the computational situation changes. The difficulty of solving the problem in both the computational and metahuman instances will be a measure of the "control complexity" of the instance (where "control complexity" refers to the computational resources required to devise a strategy to cause a system to give a desired response, as well as to the amount of "force" applied to the system in the course of the devised strategy).

Another common problem involves the lack of self-monitoring and repair. Rules and actions within many metahuman institutions are often made in response to crisis situations, without a clear understanding of the impact of the action on other parts of the system in question. Deficits are allowed to grow unchecked, until the huge accumulated

debt is collectively noticed and triggers a frenzy of indiscriminate cost-cutting. Habitats which do not encourage creative growth and learning generate unemployment and crime, which is then dealt with through massive drains on the productive power of society. Our (irreplaceable) environment is stressed in a thousand different ways, and given temporary respite only after some natural disaster warns of a greater cataclysm to come.

Biological systems manage to avoid such disasters by employing a host of different mechanisms which ceaselessly patrol the body in parallel; illness and injury are quickly noticed and dealt with. A system without this sort of continuous negative feedback will require much greater energy to maintain, as problems are detected and corrected only at the last minute (or later). Of course, simply recognizing the existence of a problem or abnormal situation can be a difficult task in a complex metahuman system with different contending subgoals; it is even more challenging to understand the operation of the system well enough to be able to make corrective changes which will not have undesirable side effects.

Building metahuman institutions and metaknowledge is particularly relevant in the case of knowledge-producing organizations, such as universities. In a wider sense, the generation of expert learners and environments to support them is one of the most urgent tasks of our society, as is eloquently explained in [Bereiter and Scardamalia]. We need to understand and create distributed systems which can pool their efforts to create a whole greater than the apparent sum of its parts.

RECAP

After having examined so many differing formalizations of "complexity", one may well wish for a larger framework within which to place complexity measures. What are some directions that could move us closer to the "parametrized space of measures of complexity" mentioned as a goal in the Introduction?

Of the measures we have examined, the framework with the largest scope is probably that of Computational Complexity. How could one expand or modify this framework to encompass the other measures mentioned, as well as to allow more fuzzy and intuitive ideas of complexity to be quantified (even if only approximately)?

One important distinction is that between explicit and implicit complexity measures. An example of the former is a standard Computational Complexity class such as NP where an algorithm is proved to have a clearly definable property; the latter could be some sort of statistical performance average of an algorithm in a series of difficult situations. Usually the former will be more desirable; however, when proof is not possible, the latter type of algorithms can provide very useful insight (as the Genetic Algorithm and other such learning algorithms have shown). One expects that over time the best algorithms known for a problem will move from the latter class to the former class, as salient features and simplifications of a problem lead to better understanding.

A second related distinction is that between semantic and syntactic computational properties for which algorithms might be desired. Clearly minimizing time or space for an algorithm is an important concern; these are properties which are easy to define and observe during a computation. For more important and general properties, however, the process of definition itself is extremely difficult.

In the most general sense, a complexity measure might be thought of as a measure of difficulty. This demands the specification of i) a problem of interest, as well as ii) a way of measuring how well a proposed solution solves the problem. The usefulness of such a complexity measure would be measured by the applicability of solutions to the problem of interest, as well as the ease of finding such solutions. The incorporation of all such complexity measures into a useful framework would seem to demand a solution to the general problem of "knowledge representation"; a formal knowledge specification system that defined static and algorithmic knowledge in a reasonably intuitive way would give a common language for defining properties of such knowledge.

This thesis has surveyed many ideas, and pointed the way to the future interdisciplinary "sciences of complexity". Our challenge is to explore the not-well-understood, via explanation of traditional sequential processes and understanding of simple parallel processes that generate complex, interesting, and evolving behavior. As we generate ever-increasing complexity in our personal and collective environments, we have no choice but to take this next step in our quest to make sense of the universe.

REFERENCES

- Serge Abiteboul and Eli Shamir (Eds). Automata, Languages and Programming: ICALP 94. Springer-Verlag, Berlin; 1994.
- Michael A Arbib. Brains, Machines and Mathematics (2nd Ed). Springer-Verlag, New York; 1987.
- Nils A Baas. "Emergence, Hierarchies and Hyperstructures," in Artificial Life III; Addison-Wesley, New York; 1994.
- Jose Luis Balcazar, Josep Diaz, and Joaquim Gabarro. Structural Complexity I. Springer-Verlag, Berlin; 1988.
- Jose Luis Balcazar, Josep Diaz, and Joaquim Gabarro. Structural Complexity II. Springer-Verlag, Berlin; 1990.
- Richard K Belew and William E Hart. "Optimizing an Arbitrary Function is Hard for the Genetic Algorithm," in Proceedings of the 4th ICGA; Morgan Kaufmann, San Mateo; 1991.

Charles H Bennett. "Logical Depth and Physical Complexity", pp 227-257 in The Universal Turing Machine: A Half-Century Survey; R Herken (ed), Oxford University Press New York; 1988.

Carl Bereiter and Marlene Scardamalia. Surpassing Ourselves: An Inquiry into the Nature and Implications of Expertise. Open Court, Chicago; 1993.

D P Bovet, P Crescenzi, and R Silvestri. "A Uniform Approach to Define Complexity Classes," pp 263-283 in Theoretical Computer Science; Vol 104, 1992.

Douglas S Bridges. Computability: A Mathematical Sketchbook. Springer-Verlag, New York; 1994.

Rodney A Brooks and Pattie Maes. Artificial Life IV. MIT Press, Cambridge; 1994.

Cristian Calude. Theories of Computational Complexity. Elsevier Science Publishers, Amsterdam; 1988.

Rafael C Carrasco and Jose Oncina (Eds). Grammatical Inference and Applications: ICGI-94. Springer-Verlag, Berlin; 1994.

John L Casti. Reality Rules: Picturing the World in Mathematics (2 Vols). Wiley, New

York; 1992.

John L Casti. Complexification: Explaining a Paradoxical World Through the Science of Surprise. HarperCollins Publishers, New York; 1994.

Gregory Chaitin. Algorithmic Information Theory. Cambridge University Press, Cambridge; 1987.

Gregory Chaitin. Information-Theoretic Incompleteness. World Scientific Publishing Company, Singapore; 1992.

Jack Cohen and Ian Stewart. The Collapse of Chaos. Viking Penguin, New York; 1994.

Thomas M Cover and Joy A Thomas. Elements of Information Theory. John Wiley and Sons, Toronto; 1991.

James P Crutchfield. "The Calculi of Emergence: Computation, Dynamics and Induction," pp 11-54 in Physica D; Vol 75, 1994.

Nigel Cutland. Computability: An Introduction to Recursive Function Theory. Cambridge University Press, Cambridge, England; 1980.

Mihaly Csikszentmihalyi. The Evolving Self. HarperCollins Publishers, New York; 1993.

Martin D Davis, Ron Sigal, and Elaine J Weyuker. Computability, Complexity, and Languages (2nd Edition). Academic Press, San Diego; 1994.

Richard Dawkins. The Blind Watchmaker. Longman Scientific and Technical, Essex; 1986.

Stephanie Forrest (Ed). Emergent Computation. MIT Press, Cambridge; 1991.

Peter Gacs and John Reif. "A Simple Three-Dimensional Real-Time Reliable Cellular Array", pp 125-147 in Journal of Computer and System Sciences; Vol 36, 1988.

O Goldreich, S Micali, and A Wigderson. "Proofs that Yield Nothing but their Validity, and a Methodology of Cryptographic Protocol Design," pp 174-187 in Proc 27th IEEE Symposium on the Foundations of Computer Science; 1986.

O Goldreich, H Krawczyk, and M Luby. "On the Existence of Pseudorandom Generators", pp 1163-1175 in SIAM Journal of Computing; Vol 22, 1993.

R Greenlaw, H J Hoover, and W L Ruzzo. A Compendium of Problems Complete for

P. Oxford University Press, New York; 1993.

Hazel Henderson. Paradigms in Progress: Life Beyond Economics. Knowledge Systems, Indianapolis; 1991.

Douglas Hofstadter. Godel, Escher, Bach: An Eternal Golden Braid. Basic Books, New York; 1979.

Douglas Hofstadter. Metamagical Themas. Basic Books, New York; 1985.

John H Holland. Adaptation in Natural and Artificial Systems. MIT Press, Cambridge; 1992.

Alan Hutchinson. Algorithmic Learning. Oxford University Press, New York; 1994.

Jarkko Kari. "Rice's Theorem for the Limit Sets of Cellular Automata", pp 229-254 in Theoretical Computer Science; Vol 127, 1994.

Stuart A Kauffman. The Origins of Order. Oxford University Press, New York; 1993.

Kevin Kelly. Out of Control: The Rise of Neo-Biological Civilization. Addison-Wesley, New York; 1994.

K Ko, P Orponen, U Schoning, and O Watanabe. "Instance Complexity", pp 96-121 in Journal of the ACM; Vol 41, 1994.

Moshe Koppel. "Structure", pp 435-452 in The Universal Turing Machine: A Half-Century Survey; R Herken (ed), Oxford University Press, New York; 1988.

Moshe Koppel. "An Almost Machine-Independent Theory of Program-Length Complexity, Sophistication, and Induction", pp 23-33 in Information Sciences; Vol 56, 1991.

John Koza. Genetic Programming. MIT Press, Cambridge; 1992.

John Koza. Genetic Programming II. MIT Press, Cambridge; 1994.

Christopher G Langton. Artificial Life: Proceedings of an Interdisciplinary Workshop on the Synthesis and Simulation of Living Systems. Addison-Wesley, New York; 1989.

Christopher G Langton et al. Artificial Life II. Addison-Wesley, New York; 1991.

Christopher G Langton. Artificial Life III. Addison-Wesley, New York; 1994.

Stephen Levy. Artificial Life: The Quest for a New Creation. Pantheon Books, New York; 1992.

Ming Li and Paul Vitanyi. An Introduction to Kolmogorov Complexity and its Applications. Springer-Verlag, New York; 1993.

Klaus Mainzer. Thinking in Complexity: The Complex Dynamics of Matter, Mind and Mankind. Springer-Verlag, New York; 1994.

Zbigniew Michalewicz. Genetic Algorithms + Data Structures = Evolution Programs (2nd Edition). Springer-Verlag, New York; 1994.

Pascal Michel. "Busy Beaver Computations and Collatz-like Problems", pp 351-367 in Archive for Mathematical Logic; Vol 32, 1993.

Marvin Minsky. Society of Mind. Simon, New York; 1986.

Balas K Natarajan. Machine Learning: A Theoretical Approach. Morgan Kaufmann, San Mateo; 1991.

G Nicolis and I Prigogine. Exploring Complexity: An Introduction. W H Freeman, New York; 1989.

Christos H Papadimitriou. Computational Complexity. Addison-Wesley, New York; 1994.

Christos H Papadimitriou and Kenneth Steiglitz. Combinatorial Optimization: Algorithms and Complexity. Prentice-Hall, New Jersey; 1982.

Proceedings of the Fifth International Conference on Genetic Algorithms. Morgan Kaufmann, San Mateo; 1993.

Jorma Rissanen. Stochastic Complexity in Statistical Enquiry. World Scientific Publishing Company, Singapore; 1989.

Michael Rothschild. Bionomics: Economy as Ecosystem. Henry Holt and Company, New York; 1992.

Justin R Smith. The Design and Analysis of Parallel Algorithms. Oxford University Press, New York; 1993.

Gregory Stock. Metaman. Simon and Schuster, New York; 1993.

M Mitchell Waldrop. Complexity: The Emerging Science at the Edge of Order and Chaos. Simon and Schuster, New York; 1992.

Stephen Wolfram. Cellular Automata and Complexity. Addison-Wesley, New York; 1994.

Andy Wuensche. Global Dynamics of Cellular Automata. Addison-Wesley, New York; 1992.

Wojciech H Zurek, Ed. Complexity, Entropy and the Physics of Information. Addison-Wesley, New York; 1990.

END

0 5 | 0 6 | 9 | 6

FIN