A Working Hypothesis for General Intelligence

Eric Baum
Baum Research Enterprises
ebaum@fastmail.fm
http://whatisthought.com

Abstract. Humans can construct powerful mental programs for many domains never seen before. We address the questions of how this occurs, and how it could possibly be accomplished in software. Section one surveys a theory of natural understanding, as follows. One understands a domain when one has mental programs that can be executed to solve problems arising in the domain. Evolution created compact programs understanding domains posed by nature. According to an extrapolation of Occam's razor, a compact enough program solving enough problems drawn from a distribution can only be found if there is simple structure underlying the distribution and the program exploits this structure, in which case the program will generalize by solving most new problems drawn from the distribution. This picture has several important ramifications for attempts to develop Artificial General Intelligence (AGI), suggesting for example, that human intelligence is not in fact general, and that weak methods may not suffice to reproduce human abilities. Section 2 exemplifies this picture by discussing two particular thought processes, the mental program by which I solve levels in the game of Sokoban, and how I construct this mental program. A computer program under construction to implement my introspective picture of my mental program is based on a model of a particular mental module called Relevance Based Planning (RBP). Section 3 argues that programs to address new problems (such as my mental program to play Sokoban) can be constructed (both naturally and artificially) if and only if sufficient guidance is already present. It proposes a computational structure called a scaffold that guides rapid construction of understanding programs when confronted with new challenges.

Introduction

A striking phenomenon of human intelligence is that we understand problems, and can construct powerful solutions for problems we have never seen before. Section one surveys

a theory under which one understands a problem when one has mental programs that can solve it and many naturally occurring variations. Such programs are suggested to arise through discovering a sufficiently concise program that works on a sufficiently large sample of naturally presented problems. By a proposed extrapolation of Occam's razor, such a concise effective program would only exist if the world posing the problems had an underlying structure that the program exploits to solve the problems, and in that case it will generalize to solve many new problems generated by the same world. It is further argued that the concise Occam program leading to human and natural understanding is largely embodied in the genome, which programs development of a modular program in the brain. We build upon this modular program in constructing further mental programs.

Section 2 exemplifies this picture by discussing two particular thought processes, the mental program by which I solve levels in the game of Sokoban, and how I construct this mental program. A computer program under construction (with Tom Schaul) to implement an introspective picture of my mental program is based on a new approach I call Relevance Based Planning (RBP). RBP is suggested as a model of a human planning module. By invoking domain dependent objects, it exploits the underlying structure of its particular domain, which in the case of Sokoban involves exploiting 2-dimensional structure. Several other domain dependent modules, intended to reproduce introspection, are under development.

Section 3 proposes a computational structure called a scaffold that provides guidance on how to rapidly construct understanding programs when confronted with new challenges.

A general contrast between the point of view of this paper and much work in AGI or Human Level AI, is that this paper is focused on the problem of how one can construct programs that understand new problems and problem domains. Much current work is based on systems that are expected to run existing programs, and either avoid facing the issue of automatically constructing new code, or assume it can be done by weak methods. The theory presented here argues that understanding is based on Occam programs, which are computationally expensive to find, because one must solve a hard computational problem to construct them. The necessary programs, for example in Sokoban, seem to be rather complex. Accordingly, I suggest that human intelligence is not really "general", but rather is powerful only at exploiting a class of problems arising in the natural world, and problems solvable with modules arising from these. New programs solving a new domain can be feasibly constructed only when one already has much of the computational structures necessary to build them, so that the search for the new program is tractable. This contrasts, for example, with the assumption that intelligence is based on weak methods--methods not exploiting much knowledge [1]. Instead, I argue that understanding is based on having powerful domain dependent knowledge, and address the questions of how we

can build such knowledge, what form it takes in humans and what form it should take in attempts at building human-level AI, and how it guides search for new programs.

Natural Intelligence

Turing argued compellingly that whatever is happening in the brain, it could be simulated in detail by a computer running the right software. Thus thoughts must be isomorphic to some particular computations. Turing's thesis gives us a precise language which we can use to discuss and model thought, the language of computer programs. This thesis, however, left us with some puzzles. A first important one is: what about this particular code causes it to *understand*? A second important one is: given that complexity theory has indicated that many computations are inherently time consuming, how does the mind work so amazingly fast?

Computational learning theory has explained generalization as arising from Occam's razor. The most studied context is concept learning, where one sees a series of classified examples, and desires to learn a function that will predict correctly whether new examples are examples of the concept or not. Roughly speaking, one can show that if one presents examples drawn from some process, and finds a simple enough function classifying most examples in a large data set, it will also correctly classify most new examples drawn from the process on which it hadn't been been specifically trained, it will generalize. Results along these lines from three viewpoints (VC dimension, Minimum Description Length, and Bayesian probability) are surveyed in chapter 4 of [2]. Yet another branch of related results can be found in the literature on universal algorithms, cf [3,4].

Such approaches start by formalizing the notion of what is meant by ``simple''. One way is by program length, a shorter program (or alternatively, a shorter program that runs within a given time bound) is considered simpler.

A basic intuition behind these kinds of results is the following. If the process producing the data was not structured, the data would look random. No extremely simple program would then exist dealing correctly with the seen data. The fact that you are able to find a compact program correctly classifying massive data thus implies that the process providing the data actually had a compact structure, and the program was exploiting that structure to make predictions.

Such results have been known for decades, so why do we not yet have programs that classify concepts as well as people do? An answer is, we are unable to actually find a compact program (say a compact neural net) correctly classifying most interesting kinds of data, even for sets where people could classify. In fact, for a wide variety of sample

problems, it has been proved NP-hard to find such compact programs, which indicates there can be no fast solution. The reader may be familiar with the Zip utility, in common use to compress computer files. Zip embodies a simple algorithm that runs rapidly, but is largely oblivious to the file being compressed. Generalization, by contrast, requires extracting the simple underlying structure particular to a given data set thus achieving a much smaller compression, and can not be done rapidly in such a generic fashion. It is only after the point where data is compressed beyond what is easy or generic that the underlying structure becomes apparent and meaningful generalization begins, precisely because that is the point where one must be sensitive to specific, surprisingly compact structure of the particular process producing the data. When such compression can be accomplished in practice, it is typically done by some algorithm such as back-propagation that does extensive computation, gradually discovering a function having a form that exploits structure in the process producing the data.

The literature also contains results that say, roughly speaking, the only way learning is possible is through Occam's razor. Such no-go theorems are never air tight -- there's a history of other no-go theorems being evaded by some alternative that escaped conception-- but the intuition seems reasonable. A learner can always build a complex hypothesis that explains any data, so unless the space of possible hypotheses is highly constrained by some inductive bias, there is no reason why any particular hypothesis should generalize. Note that the constraint does not have to be shortness of program-- in fact evolution seems to use a form of early stopping rather than just building the shortest hypothesis.

Human thought and understanding, of course, seem to encompass much deeper abilities than simple concept classification. However, the basic Occam intuition behind such classification results can be straightforwardly extrapolated to a conjecture about a general program interacting with a complex world: If you find a short enough program that rapidly solves enough problems, that can only be because it exploits compact structure underlying the world, and then it will continue to solve many later problems presented by the same world.

My working hypothesis is that this exploitation of structure is what understanding is. Humans are based on a compact program that exploits the underlying structure of the world. So, for example, we can solve new problems that we were not explicitly evolved to solve, such as discovering higher mathematics, playing chess or managing corporations, because underlying our thought there is a compact program that was trained to solve numerous problems arising in the world, so many that it could only solve these by exploiting underlying structure in the world, and it thus generalizes to solve new problems that arise.

How does a very concise program deal with many problems? Experience and a number of arguments suggest, by code reuse. It can be so concise by having a modular structure, with modules exploiting various kinds of structure that can be put together in various ways to solve various problems. As evolution produced a compact program to deal with the world, it discovered such modules.

This may explain why thought is so metaphorical. Metaphor occurs when we reuse a module associated with one context, to exploit related structure in another context. So when we speak of buying, wasting, spending, or investing our time, we are reusing a computational module or modules useful for valuable resource management. When we speak of buttressing the foundations of our arguments, we are reusing modules useful for building concrete structures. And so on.

Note that the kind of exploitation of structure involved here is rather different than what we usually think of in simple classification or prediction problems. If we simply find a concise enough program (for example, a small enough neural net) correctly classifying data points (for example saying whether images show a chair or don't), it will generalize to classify new data points (e.g. images) drawn from the same process. But simply finding a compact description of structure can be a separate problem from exploiting compact structure. In the Traveling Salesman Problem, for example, we are handed a concise description of the problem, but it is still computationally hard to find a very short tour. Roughly speaking, to find the shortest tour, we will have to search through a number of possibilities exponential in the size of the description. The claim is that the world has structure that can be exploited to rapidly solve problems which arise, and that underlying our thought processes are modules that accomplish this. And for many problems, we accomplish it very fast indeed.

Consider, for example, the integers. That the integers have a compact structure is evident from the fact that all of their properties are determined by 5 axioms-- but beyond this you know algorithms that you can use to rapidly solve many problems involving them (for example, to determine if a 50 digit number is even). My working hypothesis is that our mathematical abilities arise from Occam's razor. Roughly speaking, we have these abilities because there is a real a priori structure underlying mathematics, and evolution discovered modules that exploit it, for example modules that know how to exploit the structure of Euclidean 2 and 3 space. By evolving such modules we were able to solve problems important to evolution such as navigating around the jungle, but such modules perforce generalize to higher problems.

Mathematical reasoning is one example of an ability that has arisen this way, but of course the collection of modules we use to understand the world extends far beyond, as seen for example from the explanation of metaphors above. My working hypothesis is that

each concept, basically each word in English, corresponds to a module (or at least, a method).

I expect that the mind looks like some incredibly elegantly written object-oriented code. That water is a liquid is presumably more or less represented by the class water having a superclass liquid, from which it inherits methods that allow you to mentally simulate it flowing, as well as solving various problems involving it. Experience indicates that such things are not readily coded into logic, especially not first order logic. Rather, I expect humans accomplish inference by using special procedures, that know how to do things like simulate liquid flows. My working hypothesis is that the brunt of inference by humans is not carried by modus ponens or some such general inference mechanism, it is in the various procedures that encode methods exploiting particular aspects of the structure of the world, such as the a priori structure of Euclidean 2 space or 3 space, the physics of liquids and hard objects and gases and so on. Thought is too fast to involve extensive searching over proofs, rather it is put together by executing powerful procedures encoded into a library. I expect words are more or less labels for code modules (roughly speaking nouns are classes and proper nouns instances of classes and verbs methods) so there may be tens of thousands of modules in the library. As we will discuss in Section 3, when we come to understand new problems, we do engage in search over programs, but it is a search so heavily biased and constrained by existing structure as to occur in feasible time.

Where in a human is this compact Occam program encoded? A number of arguments indicate that a critical kernel of it is encoded into the genome.

The genome is extraordinarily compact. Its functioning core (after the so-called "junk" is stripped away) is believed to be smaller than the source code for Microsoft Office. The brain, by contrast, is 100 million times bigger. Moreover the genome encodes, in a sense, the results of an extraordinary amount of data and computation: Warren Smith has recently improved on my estimate, and the best estimate now appears to be that some 10^{44} creatures have lived and died, each contributing in a small measure to evolution (footnote 95 in [5]).

The proposal of the genome as the Occam program is controversial, because many psychologists, neural network practitioners, and philosophers have argued that infants are born in a state of tabula rasa, a blank slate from which we acquire knowledge by learning. Learning theory, however, requires one have an inductive bias to learn. In my picture the genome encodes this inductive bias, programs that execute, interacting with sensory data, to learn. The learning so innately programmed appears quite automatic, reliably resulting in creatures with similar abilities provided that they are allowed interaction with the world during development.

Complexity theory suggests that learning is a hard problem, requiring vast computation to extract structure. Yet we learn so fast that we do not have time to do the

requisite computation. This is possible only because creatures are preprogrammed to extract specific kinds of meaning. The bulk of the requisite computation, and thus the guts of the process from the point of view of complexity theory, went into the evolution of the genome.

Empirical evidence shows that creatures are in fact programmed with specific inductive biases. If a rat is shocked once at a specific point in its maze, it will avoid that corner. If a rat is sickened only once after eating a particular food, it will never eat that type of food again. However it is difficult to train a rat to avoid a location by making it sick or to avoid a type of food by shocking it. The rat is innately programmed to learn particular behaviors from particular stimuli. Similar results hold true for people. A wealth of evidence supports the view that children learn grammar rapidly and almost automatically, because of strong inductive bias built into their brain by their genome. Moreover, while humans can learn ``meaningful" facts from a single presentation, they would find it almost impossible to learn things they are not programmed to recognize as meaningful. This meaning is, in my picture, defined by underlying programs, largely coded into the genome, that exploit particular underlying structure in the world. Such programs, in fact, essentially define meaning.

Consider the development of visual cortex. The brain uses parallax to calculate the depth of objects. This calculation must be tuned to the width between the eyes. The DNA program is the same in every cell, but its execution differs from cell to cell as the chemical environment differs. As the brain develops, and the skull grows, the DNA program automatically adjusts brain wiring to compute depth. How did this evolve? DNA programs that better develop function in the chemical environment were preferentially selected. But the chemical environment in cortex includes neural firings stimulated by the sensory flow. Thus a program evolved that, in effect, learns the distance between the eyes. Similarly, the same DNA programs cells to develop into visual cortex or auditory cortex depending on the ambient environment of the developing cell. Similar mechanisms would explain development of more abstract thought, for example, Seay and Harlow's famous discovery that monkeys can only acquire normal social behavior during a critical period in development. Critical periods are a flag indicating genomic programming, as many aspects of development are carefully timed. The DNA program exploits the sensory stream (reflected in the chemical environment of developing cells) to grow appropriate neural structures, which may implement powerful procedures for reasoning about the world. Learning and development are two sides of the same coin, and so we should expect evolution of tailored learning programs.

If the genomic program encodes development of modules for a variety of meaningful concepts, everything from valuable resource management and understanding 2-topology to

causal reasoning and grammar learning, this should be manifested through local variations of gene expression in developing brains. Gray et al. recently published an image in Science magazine that seemed to show quite detailed structure in gene expression [6]. As the technology improves, gene expression data should show programming of the kind required by this theory, or refute it.

The working hypothesis proposed here is that the genome encodes a compact ``Occam'' program that (metaphorically) ``compiles'', interacting with sense data, into executable modules we use to perceive and reason. The modules that are built may be relatively large and complex, yet still generalize because they are constrained by the compact genome. Lakoff and Johnson have cataloged so many diverse types of causal and temporal reasoning that they suggest there can exist no ``objective metaphysics'' of time or causality [7]. I believe this is mistaken. Our universe does indeed display amazingly rich phenomena, but these all arise from an amazingly concise underlying physics. Our mental modules are varied, but constrained by a concise program to exploit this concise structure. Thus all the different types of causal reasoning are related and constrained.

The human mind evidently employs algorithms not explicitly coded in the genome. For example, the reader has procedures that allow him or her to read this text. Such knowledge is built as meaningful modules that invoke more basic modules. Experience suggests that complex programs must be built in such a modular fashion, sometimes called an abstraction hierarchy. The Occam's razor hypothesis suggests that the modules coded in the genome are meaningful precisely in the sense that powerful programs can be built on top of them. That is: these compact modules are such that in past experience, powerful programs have emerged to solve problems such as navigating in the jungle, therefore these modules should be such that powerful superstructures will emerge in new domains such as reasoning about higher mathematics.

Finding new meaningful modules is a hard computational problem, requiring substantial search. This suggests a mechanism by which human mental abilities differ so broadly from chimpanzees, who are genetically almost identical. Chimpanzees can discover new meaningful modules only over one lifetime of study. But humans, because of our ability to transmit programs through language, have cumulatively built and refined a vast library of powerful computational modules on top of what is coded in the genome.

This additional programming does not just include what we think of a technology, but rather includes qualities regarded as core mental abilities. For example, only humans are said to have a theory of mind in that we understand that other individuals may have different beliefs than our own, and reason accordingly. However, apes display behaviors indicating aspects of theory of mind; and plovers feign injury, limping off to distract a predator from their nest, only when the predator seems to notice the nest. Thus plovers

attend to the perception of the predator and modify their behavior accordingly. This ability requires subroutines on which any theory of mind must rely. This suggests that the human theory of mind is a complex modular program built on top of meaningful modules coded more explicitly in the genome, and that humans have been able to discover this more powerful program over generations, because we pass partial progress on. Bedtime stories and fiction are means of passing such programs on to children. Psychophysics experiments are consistent with this view, showing that children acquire theory of mind over a period of years.

Thus this computational theory of mind, while it suggests that much of thought is coded in the genome, simultaneously suggests that most of the difference between human and ape cognition can be regarded as the product of better nurture. The key is that the program of mind is modular and hierarchic, and that the discovery of new modules is a hard computational problem. By virtue of our ability to communicate partial results, humans have made sustained progress in developing an ever more powerful superstructure on top of the innately coded programming.

In summary, computational learning theory has made a compelling case that generalization comes from Occam's razor. Understanding can be explained by a simple extrapolation of this as arising from evolution of a compact genomic program that builds a set of procedures that exploit the underlying structure. This and many other aspects of thought (for example, why the various aspects of consciousness, both subjective and objective, arise in this fashion), are explained in much more detail (and with many more citations) in [2], where some alternatives to the above working hypothesis are also briefly discussed.

This picture suggests that a truly ``general'' intelligence is unattainable. Humans solve new problems very fast when we already have modules for dealing with them. Over hours or years, humans can solve somewhat more general new problems, those requiring construction of new code or new mental modules, only when we have modules that can be put together with only a reasonable amount of search (possibly guided or orchestrated by modules designed for putting other modules together into the right code). But according to the Occam thesis, all this machinery only works for problems in some sense drawn from a natural distribution of problems-- relating to the a priori structure of mathematics and the underlying compact structure of our universe. General problems (a likely example is given by arbitrary problems in the class NP) may simply not be rapidly solvable. Universal methods may be given that are argued in one sense or another to address such problems as well as possible [3, 4], but such solutions may take arbitrarily long and in my opinion don't relate directly to human intelligence; and may distract from what we want to study in AGI. To achieve AGI, I suggest, we will need to develop the kinds of modules on which human

thought is built, that exploit the kind of underlying structure in our universe.

This picture explains why many AI programs do not seem to display ``understanding" [8]. These programs were not compact or constrained. Mostly, they were simply built by human programmers without Occam's razor in mind. Arguably, AI programs that have displayed some understanding have been built on very compact structures. For example, the core of chess programs such as Deep Blue, which do seem to understand chess quite well, is tiny: a program consisting of alpha-beta (a few lines of code) plus simple material as an evaluation function (also a tiny program) plus a short quiescence search routine would already play phenomenal chess if applied in an 11-ply search[1]. If we want to build an

---

[1] Some readers may be suspicious of the assertion that Deep Blue in some sense understands chess since it does a huge search. A referee notes, for example, that a completely brute force exhaustive search program, a very few lines of code, would solve any finite problem eventually. In an attempt to capture normal notions of understanding, and yet discuss the subject in a concrete way, let us consider that an entity understands an item (or a domain) if it almost always does the right thing on the item or domain, and on almost all naturally occurring variations of it, and behaves in a reasonable way on other variations of it. By this definition, Deep Blue clearly understands most chess positions: pick almost any chess position you like, and it will make the right move. Moreover, Kasparov attempted to play toward types of positions not normally occurring in human play in a deliberate attempt to stump it, and largely failed. Moreover, you could vary the rules of chess within some reasonable range, and alpha-beta + material + quiescence would still play well-- if you vary far enough, you will have to change the evaluation function, but if you allow tweaks like that, the same methods extend strongly all the way in variation space to, for example, Othello. Thus this analysis technique provides a useful basis for a scaffold, that may be applied to understand various games, cf section 3. That is, a scaffold that learns an evaluation function and fits it into the alpha-beta + evaluation function + quiescence framework has so much bias, it could readily learn to process a wide variety of games effectively. In my view, this is not so dissimilar from how I might pick up a new game (although I would adjust and apply other scaffolds as well.) Similarly, the exhaustive search algorithm would indeed be said to understand finite problems, where one has the resources to perform the computation. (Its worth noting in passing that Deep Blue is very far from brute force, since it arrives at its determinations while examining an infinitesimal fraction of the search tree, indeed it examines a finite search tree yet would presumably be effective for many variations of chess with an infinite search tree.)

It's not hard to give positions which Deep Blue doesn't understand (cf (Baum 2004)for discussion). The simplest way is to consider variations where humans analyze the problem from a different angle that is impervious to scaling, and you consider the problem on an n by n board as n gets large. The same applies to the exhaustive search algorithm. So I am not claiming that Deep Blue's understanding is the same as ours -- we have many more modules and scaffolds in our minds that allow us better understanding.

It might be objected that time complexity should figure in understanding, and indeed it does. I am implicitly positing that the colloquial notion of understanding is relative to the amount of time alloted. Thus to be said to understand, the entity must do the right thing fast enough. For example, I might be said to understand some branch of mathematics if I could prove deep theorems in it, even though this might take me years, but you wouldn't say I understand it just because I might prove such a theorem if you gave me a millennium. Deep Blue, which as noted above is not brute force, is fast enough to work in tournament time limits, and in fact it's very far from apparent that Deep Blue does more computation than I do in understanding a chess position.

The main point I am making, however, is that generalization and thus understanding follows from an extrapolated Occam's razor. When you find a very constrained program that's effective on a large collection of

AGI, we will have to build something that reflects the Occam property.

It is possible that we will never have the computational resources to accomplish this. Finding Occam programs seems to generically be NP-hard at least, and thus may require extensive computational effort. A human may be no more able to sit down and write Occam programs exploiting the structure of the world in the way the mind does, than a human is able by hand to solve other NP-hard problems. Evolution had more computational resources for solving this problem than we ever will. On the other hand, we start with a big advantage-- our brains. The most promising approach, as I see it, is thus to build the AGI by a collaboration of human programmer and computer. The humans must encode as much inductive bias as possible, and extensive computation must build on this to construct Occam programs.

One problem with this Occam picture is that it doesn't necessarily tell us what the Occam code looks like, beyond being very compressed and produced by evolution. In fact, it suggests that there is no much better understanding of the Occam code than the code itself, since understanding comes from a compressed description, and the Occam code is already so highly compressed that further compression, if it even exists, should be very difficult to find. Moreover, according to the Occam intuition (and some of the formal results on which it is based), any very highly compressed program effective on the data seen, rather than only the most compressed possible program, is sufficient for generalization. NP-hard problems such as that of finding such compressed descriptions often have large numbers of local optima, which may look unlike one another in detail. For example, for any huge planar Traveling Salesman Problem, almost any planar tour will be quite short but such tours may share very few links[2]. This explains why the inner workings

---

examples, so constrained that it should not exist unless there is some very special structure underlying the examples, then it will generalize. If you find a program that is not so constrained yet works on the same set of examples, it can not be expected to generalize. The constraints here are a restriction on the representation space-- the program space must be constrained in the sense that it effectively can not represent all functions, so that it is unlikely that a solution to a random problem falls in the space. Thus lookup tables of size comparable to the number of examples, although they are blindingly fast, are not constrained and will not generalize, and many AI programs are akin to this. However, I do not restrict to code length as the only possible constraint. It is entirely possible that one might achieve generalization with constraints that combine a time restriction and coding restrictions, and indeed I implicitly assume throughout that the Occam program solves the training examples fast enough. As discussed in chapter 6 of [2], evolution actually seems to have employed as constraints a combination of early stopping, evolvability (which strongly affects what representations are achievable), code length, and time complexity.

[2] Boese [12] found empirically that most short tours found by local search algorithms are related. However, there are an exponential number of locally optimal (for example, planar) tours and one could easily construct examples at various edit distances. Even in relaxation systems, such as metals or crystals, where it is trivial to compute the global optimum, physical relaxation processes produce domain structures, and two different configurations of domains will have substantial hamming distance.

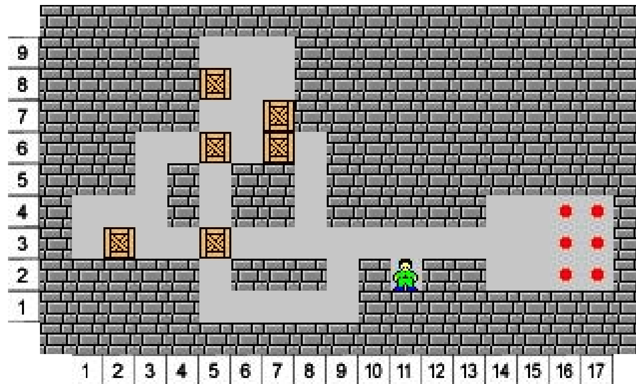
of trained neural nets are sometimes inscrutable. On the other hand, in my working hypothesis, the Occam core is in the genome, and the program in the brain is rather larger, thus admitting of a shorter description, so we might expect to be able to say something about the code in the brain. In the next section, I discuss a simple but highly non-trivial example of thought, attempting to further illuminate the nature of the code and how it is produced.

Introspective Sokoban

Sokoban is a single player game consisting of a collection of two dimensional planning puzzles called levels. The player controls a robot that can push barrels around a maze (see figure). Only one barrel can be pushed at a time, and the object is to push all the barrels into marked goals. The game is best understood by playing a few levels-- numerous levels can be found on the web cf: [9,10]. Sokoban is p-space complete [11] so there is no hope of producing a (fast) general solver[3].

---

[3]Roughly speaking, the standard proof that a particular problem class C is hard is given by showing that you can map another hard problem class H into it. Since there is (believed to be) insufficient underlying structure to solve H rapidly, this then exhibits a subclass of C that is not (believed to be) rapidly solvable. In the case of Sokoban, the proof proceeds by constructing, for any given Turing machine with given finite tape length, a particular Sokoban instance that is solvable if and only if the Turing machine halts in an accepting state. This leaves (at least) two outs for solving problems in C rapidly. First, since the mapping is into, with range only a small subset of instances of C, almost all instances of C may be rapidly solvable. For example, in the case of Sokoban, it intuitively seems that random instances can only be hard if they have a high enough density of barrels and goals, because otherwise solving the different goals decouples. Furthermore, it is plausible that random solvable instances with too high a density of barrels and goals would be over-constrained and hence readily solvable by algorithms that exploit over-constrainedness, which I believe humans often do, including in Sokoban. It would be an interesting experiment to write a random instance generator and see in what domains humans can easily solve large random Sokoban instances. Second, such mappings generically map a problem instance in H of size S into a problem instance in C of size Sk, for k some constant of rough order 100. Thus an option for solving instances in C that derive from instances in H is to perform the inverse mapping, and apply an exponential time algorithm for solving instances in H. This will be practicable for substantial (although of course, not asymptotic) instances of C. Human strategies for Sokoban, roughly speaking, exploit both these avenues. They attempt local solutions, which will work if the density of constraints is low and the problem factors. They also propagate constraints arising from barrel interactions causally, which seems likely to solve over-constrained random problems, and also ferrets out mapping structure.



Humans have however crafted a large collection of ingenious levels that humans find interesting to solve. One reason the domain is interesting for students of AGI, in fact, is that these levels appear to have been crafted so as to exercise a range of human cognitive abilities.

From the point of view of understanding general intelligence, we are interested in two problems. After having practiced the game for a while, I have a toolbox of techniques-- a mental program involving several distinct modules-- that I apply to solve new levels. The first problem is, what does this mature program look like? The second problem is, how do I build this program when I first begin playing Sokoban? The key problem in both cases is time complexity. Searching over all possible solutions, and all possible short programs for solving Sokoban, is way too slow. So we must have vast knowledge that we bring and use. Since all our thoughts correspond to computations, my working hypothesis is that this knowledge is in the form of some procedural program like the kind described in the first section that allows us rapidly to construct an effective solving program. We come to the game with numerous modules useful for problems such as planning in 2 dimensions, for example modules that calculate aspects of 2-topology like the inside of an enclosure, and modules that perform goal-directed planning. Using this head-start, we rapidly construct (within minutes or weeks) programs for analyzing Sokoban and other problems we encounter.

Introspection into my mature thought processes in Sokoban-solving motivated the design of a general planning approach that I call Relevance Based Planning (RBP). RBP may be viewed as a domain independent core algorithm that organizes computation using a number of domain dependent objects. RBP employs modules that specify various domain dependent knowledge, including all or some of the following: identifying which classes of objects in the domain are potentially affectable by operators, specifying methods for dealing with classes of obstacles, specifying contra-factual knowledge about what states

would be reached by applying not-currently-enabled operators in a given state if affectable obstacles to applying them could be dealt with, identifying local configurations of objects that prevent solution of goals (comprise a deadlock), and, where multiple potentially interacting goals are to be achieved, analyzing constraints on the order in which goals can be achieved.

RBP first applies a method to find candidate plans that could potentially succeed in solving the problem if various affectable obstacles could be dealt with. An example of such a candidate plan might be a plan to get from point A to point B, which has to get through two locked doors, but otherwise does not need to do anything impossible, such as teleport through a wall. A powerful way to generate such candidate plans is by dynamic programming, invoking the contra-factual knowledge (eg to say, if only I could unlock this door, I could get to the other side, and doors are known to be potentially unlockable, so I will proceed to see if there is a path from the other side to my goal). RBP then refines such candidate plans to see whether the obstacles can be dealt with. By proceeding in time order, it deals in this refining only with states it knows it can reach (thus avoiding much irrelevant work often engaged in by planners). If it hits a deadlock or other obstruction, it employs domain-specific knowledge for dealing with it (e.g. applies a module for dealing with that particular obstacle class, such as a module that searches for a key and uses it to unlock a door, or, because it knows what elements of the domain are comprising the deadlock, tries to move them out of the way in ways that would prevent the deadlock from arising). Also, as it takes actions on the domain simulation, it marks affected objects. If an affected object later obstructs a desired action, it considers the alternative plan of doing the other action first, to see if it can escape the obstruction. It contains a procedure for ordering the search over possible candidate plans in an optimized fashion and a hash table over states to avoid duplicating work. The algorithm that results is called *Relevance* Based Planning because it searches only possible actions that are directly relevant to achieving the goal, that is: are refining a candidate plan already known capable of achieving the goal if certain obstacles known to be potentially affectable are dealt with, and furthermore are actions that actually affect a relevant obstacle or are relevant to a plan to affect a relevant obstacle.

Tom Schaul and I first tested RBP on the sub-problem within Sokoban of getting the pusher from point A to point B without creating a deadlock. The domain dependent knowledge necessary for solving this task is relatively straightforward. Barrels can be pushed, walls can not, if the pusher attempts to move somewhere and a barrel is in the way, then the pusher may arrive in the position on the other side in the event that the barrel can be dealt with, and so on. RBP solves this problem in a rapid, efficient and very natural way. Introspection does not suggest any important differences between RBP and my mental

planning process on this problem[4].

Schaul and I have since been extending this approach to tackle the general problem of solving Sokoban instances. For this purpose a number of modules have been created. These modules implement concepts that I believe I use in solving Sokoban, and thus illustrate the procedural nature of human reasoning.

One is a deadlock detector capable of recognizing deadlocks arising from the inability to get behind any barrier formed by barrels. If a deadlock is found, the detector returns a list of the barrels comprising the deadlock (so that RBP can consider pushes of these barrels relevant to preventing the deadlock from arising.) The concepts of "deadlock", "behind", and "barrier" are fundamental to human thought on Sokoban, and seem naturally to require procedures to implement or recognize them.

Note that barriers are non-local (they can involve widely dispersed barrels that nonetheless form a barrier separating one region of the level from another). RBP straightforwardly follows complex causal chains of reasoning where it will attempt to move a barrel involved in a barrier on the far side of the level in the effort to prevent a deadlock arising when a barrel is pushed on the near side, while ignoring barrels that are not causally involved.

Another module analyzes goal placement and returns orders in which the goals can be solved. If the goals are solved in an invalid order, an earlier solved goal may prevent later solution. Some orders may require that regions be cleared before a goal is solved, or that barrels be stowed in particular locations before a goal is solved so that they will be in position to later be moved into goals. The goal area analyzer calculates all such constraints by finding a hypothetical barrel configuration from which the problem could be solved in the given order. Note that the goal area analyzer, since it doesn't pay attention to the actual locations of barrels in the domain, can return orders that are not in fact achievable from the initial position of the Sokoban level, if the reason they are not achievable is because of constraints to which it is not sensitive, such as if current positions of barrels block pusher

---

[4] All high level decisions are made by RBP in such a way that only a very restricted set of alternatives known to be relevant are considered. As finer possibilities are encountered, RBP may run out of knowledge that restricts what possible moves may be relevant. Then, if it wishes to be complete, it must search over all remaining possible moves. For some problems, one could imagine improving it by giving it other or better procedures that rule out some of these as irrelevant. In the problem of moving the pusher from point a to point b, if some barrel c obstructs a desired push, RBP might consider all pushes of c as relevant, whereas a human might say that some of them should be ruled out as not possibly useful. But the human has to do some computation to decide that this push is not useful. Some of such computations are done, both according to my introspection and in our program, by perceptual analysis, and others are done at the level of the planner. I wouldn't assert that the division is identical between my mind and our program, only that I can't point to an obvious difference and don't expect any differences on this problem are critical-- for example that they would lead to time-complexity scaling differently as more complex problems are considered.

access which is necessary for achieving a given order. We have thus factored the problem cognitively into the calculation of a valid goal order, and the calculation of how to push the barrels into that order, or if the order is not achievable, finding a different valid order. Introspectively, I use a related factorization.

Another module finds a bipartite match of all barrels to goals, where each barrel matched to a goal is capable of being pushed to the goal if one disregards other barrels that may be in the way. This is built on top of a "flowmap" that amounts to a perceptual analysis of the Sokoban level. The flowmap is a concise data structure that contains a graph of all possible pushes (under the assumption that all other barrels are deleted), and also indicates the strongly connected components of the level. The matching analyzer thus reflects several constraints on solution (such as the concept of 1-1 matching). I am not conscious of mentally working out an exact assignment before starting in on every Sokoban instance--but I am conscious of thinking about it whenever the instance is such that it may be problematic, indicating I am attending to it.

Another module finds and implements forced moves. In Sokoban, it frequently happens that once one is in a given position (very often the starting position has this nature, but it also arises during search) a sequence of pushes is forced (up to transpositions), because any other pushes will lead to deadlock. When such a circumstance is found, it makes sense to immediately make this sequence of pushes rather than analyzing the current position further. Again, introspection indicates that I interleave state space search in this way with higher level cognitive analysis such as that computed by the other modules and the overall planning algorithm.

The overall Sokoban program works by applying Relevance Based Planning to the problem of solving a given Sokoban level. It first finds candidate plans that could solve the level if affectable obstacles (namely, the other barrels) could be dealt with. Such a candidate plan combines a valid goal order and a valid matching. It then refines such plans. This involves a backtracking search where, when a given plan is ruled out, it backs up to a previous branch that is relevant to solve the particular obstacle preventing the previous plan from succeeding. It explores only backups and changes that are relevant to fixing a flaw in a previously considered plan.

Schaul has implemented and largely debugged a first cut at all of the above modules, but some components are not yet implemented efficiently or effectively enough, and several have not yet been integrated. At present, our program is solving 31 of the 90 standard levels(Meyers).

A few comments on the effort as it stands. First, it is clear that I do calculate the above modules. Some researchers are suspicious of introspection, but it is nonetheless objectively verifiable that I calculate these concepts, because I can answer questions such as: “what

barrels in this configuration form a deadlock". Unfortunately, the means I use to calculate them do not seem introspectively to be the same as how Tom has coded them, and this is causing difficulties-- his code sometimes has problems in calculating some one of these modules that I find trivial. Nonetheless, I do compute similar concepts and integrate them in related (but possibly more sophisticated) ways. My mature mental program has thus got to be quite complex. I must have some ability to build this complex program on a timescale of days. I would assert that any system aspiring to do human-level AI will need the capability to build procedures of this general nature and complexity.

Second, it seems at best borderline possible to implement my mental program for Sokoban by hand. Doing so has so far required a few man years of work, and is not yet finished. Even if it can be accomplished for Sokoban, one might imagine that other domains that are less well formalized would prove daunting. There are also a few computational modules that I introspectively use in Sokoban that have not yet been coded, and may prove beyond our abilities, or at least beyond our funding. It's not yet clear whether these are essential, or whether instances I solve using them can all be solved roughly as easily using other implemented techniques. Some of them certainly improve efficiency. For example, I mentally cache meaningful computations in a way our system does not yet capture. It's also plausible that if I encountered a new level that required a new kind of technique, I would build it mentally, so simply implementing a program that solves any given set of levels might not exhaust human capability.

But perhaps the biggest problem is, our ability to introspect about how modules work seems limited. The top level RBP algorithm seems much more accessible to introspection than the inner workings of the modules it calls. This is consistent with the picture of consciousness put forth in chapter 14 of [2], where awareness is identified with computations of a top level module that accesses lower level modules only through procedure calls and the like, and is thus not able to directly examine their mechanism. As a result, the algorithms implemented within the Sokoban modules are not necessarily similar to the algorithms implementing similar computations within human thought, and it has turned out as a result that they do not always perform adequately. It appears that such modules will have to be learned, rather than hand programmed.

Code Construction and Scaffolds

Two key question are thus how do I mentally build an algorithm to play Sokoban when I take up the game, and how can we produce a program that is capable of building such programs when presented new problem domains?

My working hypothesis is that building such a program as fast as I do is only possible if almost all pieces of the program are already constructed, and only a series of relatively small searches is necessary to put them together. So I suggest that I have mentally already something very much like an RBP module, that needs only to build the various domain dependent objects in order to apply to Sokoban. And moreover, I have various modules already present that are useful for building these modules. I am then able to build it by a series of module constructions, each involving sufficiently small search that it is feasible.

Experience indicates that evolutionary programming is quite capable of producing programs that solve interesting problem domains, if it is given sufficient guidance, for example in the form of an instruction set containing pertinent macro-instructions, and a series of increasingly hard problems that allow it to make each new programmatic discovery in a feasible length of time. For example, Igor Durdanovic and I performed experiments with an evolutionary program we called Hayek [13, 14, 2]. Hayek received at least three kinds of guidance. First, it was based on an evolutionary economic system designed to bias in the building of a modular program, and to rapidly prune candidate modules not working in consort with the rest of the system. Second, it was presented problems with considerable structure, and given training examples that increased in size and difficulty as it learned. Third, it was given some guidance in the form of its instruction set, which in some runs included useful macros. Given all this, it was able to rapidly produce programs that solved interesting problems. Given a useful macro, it rapidly generated a program that solved very large Blocks World Problems. Given an expression language that allowed useful pattern matching and turned out to admit a 5 line program solving arbitrary Blocks World instances, it generated such a program-- but only when given syntax that restricted the search space by building in some aspects of the topology. Not given such syntax, it failed for a simple and obvious reason-- it couldn't in a week of computation find a program that solved even the easiest problems presented, and thus it got no feedback whatsoever, hence was engaged in blind search. Control experiments using the same instruction sets, but dropping particular aspects of the economic system, showed that the economic system also provided helpful bias useful to the successful outcome.

Likewise, as I read Schmidhuber's description of his experiments building a Tower of Hanoi solver [4], what jumps out is that after putting in a few carefully chosen macro-instructions, and by training on increasingly larger instances, his system was able to find a powerful program after affordable search[5].

---

[5] Schmidhuber's paper emphasizes the bias-optimality of his program search technique. The extent to which ``bias-optimality" was helpful was not empirically tested by comparison to a hill-climbing alternative. Roughly speaking, Bias-optimality was defined to mean, a search over possibilities for the shortest solution, where the search never invests a much higher fraction of one's computation in a particular candidate than the likelihood that

We also tested such a hypothesis in preliminary experiments in Sokoban. Given instructions such as ``getBarrelTo(B,L)" (which computed and applied a sequence of pushes taking barrel B to location L) and "openupaZone" (which computed and applied a sequence of pushes getting behind a nearby barrier), Hayek was rapidly able to create a solver capable of solving an interesting Sokoban level [15].

What does introspection tell me about how I built my mental program to play Sokoban? To begin with, when I approached Sokoban, I already had in my mind at the least something much like RBP implementation of move(A,B) (the module that takes the pusher from point A to point B). I use this for navigation constantly in real life-- calculating how to get from my seat to go get a drink, for example, even if some chairs are in the way. This module also (with minor amendment) allows me to push a barrel from point A to point B[6]. Note also that animals need to navigate around the world. Although move(a,b) involves somewhat complex code, it seems likely evolution discovered something like it and built it in to the genome. After all, no-one doubts that evolution discovered and built in the program for the ribosome, which is rather more complex.

I started doing pushing barrels toward goals,, and shortly encountered the possibility that filling one goal would interfere with later goals. I then recognized that the problem separated into bringing barrels up and finding a non-obstructive goal order, and built a goal order analyzer. It's not entirely obvious how I recognized here that the problem factored-- my working hypothesis is that I essentially already knew it, because I had knowledge coded in that I could analyze locally in space, and locally in time; that I could figure how to bring an object into a space and then separately analyze how to use it there. I expect such knowledge is already coded into animal genomes. And as I continued to address Sokoban levels, I encountered several other conceptual problems, each of which I recognized as separately solvable; in each case, I suspect, recognizing this because I already essentially had a module coding for it in another context, or a programmatic structure that told me it

---

it is the best program according to one's bias. Given no particular bias, one thus searches uniformly breadth first. By contrast, hill-climbing roughly speaking adopts the position that good programs are clustered. If one wants to find the absolute shortest program, making variations in a good program might be a worse idea than searching in an unbiased way. But in a domain where good programs are clustered, the bias-optimal approach might fail where a hill climbing approach would find a program sufficiently compact to generalize. For example, Ellington and Szostack [16] suggested that RNA might find interesting programs because the space of RNA sequences contains a sufficient density of sequences that have a small ability to catalyze a particular reaction, that one such sequence can be found with feasible-size random search; and then hill-climbing on this sequence may rapidly generate a good catalyst. It won't be the best possible, but an unbiased search for the best would likely be completely infeasible.

[6] I also had previously developed the ability to apply this module in navigating around a map, from a top down view. I recall watching my young child develop this ability on a video game, and it took some days or weeks to construct.

could be analyzed separately[7].

My working hypothesis is thus that code may be feasibly constructed if enough guidance is provided, and we should ask how and in what forms guidance is provided within humans, and how we can provide guidance for artificial systems.

My model for how pre-coded structure can guide program construction is based on what I call a ``scaffold". A scaffold is a procedure, which may have arguments, together with annotations that give guidance in how to construct or select procedures to feed in to the arguments. Thus a scaffold may be written:

$P(a_1, a_2, ..., a_n)[c_1, c_2, ..., c_n]$

where P is a procedure or a function, the a_i are its arguments, and the c_i are annotations, c_j giving instructions in how to construct the j-th argument. Here $P(a_1, a_2, ..., a_n)$ may be a specific procedure or function in the ordinary sense of computer programming, and may be written in a standard programming language, such as C, Python, or Lisp. P may also take an indefinite number of arguments, as for example dotted tail notation supports in Lisp.

When a scaffold is used, it first must be trained. In this phase, programs are constructed or evolved that substitute for the arguments. The annotations guide this process. The trained scaffold is then a module that can be applied.

An example of a scaffold without annotations might be the following. In Schmidhuber's experiments, he supplied some key macro-instructions called defnp, calltp, and endnp, each a separate module about 15 primitive instructions long. These proved to be useful for constructing recursive functions, and after he added them to the instruction set, his searcher was able to discover a program solving Towers of Hanoi after some days of search. But in fact, the final program's use of these instructions turned out to be in the forms (defnp a calltp b endnp) and (defnp a calltp b calltp endnp) where a and b represent slots into which other code was substituted. So maybe what was really wanted was a framework (defnp ((a calltp) b .) endnp) where by the notation (x .) I mean an arbitrary number of copies of structure x (biased toward a small number). Given such a structure, we have to construct code for a number of modules (in this case, by use of the dot notation, an indeterminate number of modules but in many cases a fixed number), and substitute them in to give the final module. This would have restricted the search space for the overall program much more than the simple instructions that Schmidhuber in fact used, which for example did not contain ordering information. The idea of scaffolds is to support such constructions, and to furthermore greatly guide the search for appropriate programs by

[7] My ability to continue adding new techniques when faced with new problems requiring them, is thus conjectured to rest on my large existing library of modules not yet tapped.

allowing appropriate annotations.

There are at least five types of annotations that may be supported. Type 1 annotations suggest plugging in another scaffold in place of a particular argument. A list may be supplied, and tried in order (until one is found that works). These scaffolds are trained also (in depth first manner, discussed more below).

Type 2 annotations are text, giving instructions for example to a human programmer. (These same kinds of instructions could be read and processed by an appropriate module or program, provided it had sufficient knowledge already coded in to recognize and appropriately respond. At the present state of AGI development, human programmers are more readily available.) They often suggest supplying certain kinds of training examples to learn a program. For example, a type 2 annotation might suggest supplying examples of deadlocks to a module constructor to produce a deadlock detector, which would then be substituted in to the appropriate place in an RBP scaffold with a slot demanding a deadlock detector. My hypothesis is, whenever we can separate out a sub-problem in a useful way, and either supply code for it, or train it from independent examples, that greatly cuts the search in finding the overall program, so such sub-problems should be addressed first where feasible.

Type 3 annotations specify particular kinds of module constructors to be used in constructing a program for a slot. By a module constructor, I mean a program that is fed such things as training examples, an instruction set out of which to build programs, an objective function, etc., and which then builds an appropriate program, for example by performing search over programs or evolutionary programming. For example, I have developed specific types of evolutionary economic systems (EES) that evolve programs that efficiently search for solutions, and variants of these that work in adversarial environments such as games. Standard evolutionary module constructors evolve programs to solve a problem. Hayek, for example, discovers a collection of agents so that, presented with a problem, a single sequence of agents will be considered that (hopefully) solves it. These new types of EES evolve programs that apply a focused search when solving a new problem. They thus learn search control knowledge appropriate for responding to various outcomes. Certain classes of problems seem to require search, so they will be appropriate in certain slots of scaffolds. In other slots of scaffolds, a neural net training might be more appropriate.

Type 4 annotations specify or constrain which instruction sets or instructions are to be used in building the program. Program evolution or construction is much more efficient when it is restricted to building the module out of instructions likely to be appropriate for the task at hand, and as few distracting instructions as possible. Guiding the instruction set can make a vast difference to the efficiency of a module constructions.

Type 5 annotations specify improvement operators (e.g. kinds of mutations to be used in evolving a module) (which again may be critical). Other kinds of annotations might be imagined.

Scaffolds may be trained by a straightforward algorithm. We begin by walking down depth first filling in scaffolds suggested by annotations of type 1 for particular slots. When we find an internal problem with independent training data (a sub-problem that can be independently trained, for example with an annotation specifying what data is to be supplied) we train that first, recursively, and assuming we find code that solves that sub-problem, we fix that code. Alternatively if we fail on such a specified sub-problem, we backup to a previous choice point in the search. When doing a module construction (for example, at the top level, with candidate scaffolds within filled-in and all internal independent problems already solved) we use a method that preserves whatever parts of the structure have been fixed. For example, we may do evolutionary programming where the only crossovers and mutations that are allowed respect the fixed parts of the code; or we may do search where parts of the code are fixed.

Note that this algorithm involves a certain amount of search- but if the scaffolds are appropriately coded, vastly less than would be necessary without them.

Of course, especially if annotations indicate what is wanted, a programmer may supply some modules by hand. This may sound contrary to the spirit of AGI, but actually much of human thought works that way also. In my picture, for example, humans approach games with (among other things) a scaffold that knows about search and needs to be fed an evaluation function. But when you first learned to play chess, you did not develop your own evaluation function. A teacher told you, consider material, 9 points for a queen, 5 for each rook, and add it all up. Later, you improved this evaluation function with other terms.

Some examples of interesting scaffolds are the following. A scaffold for combining causes. It asks to be presented with training examples of cause 1, and then applies a specified EES module constructor to build a collection of agents dealing with cause 1. Then it repeats this process for as many different causes as are presented. Finally, it extracts all the separate agents, merges them into a single EES, and trains this on presented examples in order to create code for interaction effects between the causes. A particular application for such a scaffold might be to solving life or death problems in Go. Game search Evolutionary Economic Systems that have agents suggesting moves killing or saving groups are first trained on examples with a single theme-- for example problems where the group can run to safety (or be cut and so prevented from running to safety). A second such G-S EES is trained on examples where the group can make internal eyes, or be prevented from making internal eyes. And so on. The agents are then extracted from these

individual EES-es, and placed into a single EES, which is then trained on general examples (and thus learns to deal with interaction effects, for example feinting at a run in order to put down a stone useful for making internal eyes.) Such a scaffold includes fixed code: such as the EES code that holds auctions and determines what computation is done given a particular collection of agents reacting to a particular problem. It also includes annotations, specifying that a sequence of examples be presented of differing causes; specifying that specific module constructors are applied to learn from them; possibly specifying that specific instruction sets (e.g. pattern languages) are used to code the agents. It breaks up an evolutionary programming problem that would possibly be intractable into a series of more tractable ones.

A scaffold for graph based reasoning GBR(W,P,Q,R)[annotations] embodies knowledge of two dimensional graphs. It is presented with W, a world state supplied to the program consisting of a grid (representing a 2-dimensional problem) with a data structure assigned to each node of the grid (representing state at that location). The annotation specifies that an appropriate W be presented. P is a program that processes W and marks grid points according to whether they belong to local structures, which we call groups. The annotation specifies that either such a P should be supplied, or examples supplied from which it can be evolved. Q then is a program that evaluates groups, in the context of neighboring groups. Again, Q may be supplied or learned from examples. Finally R is a program that combines Q evaluations to an overall evaluation. R may be chosen from a list of useful scaffolds and functions (summing values will often be useful, in some adversarial games a complicated alternating sum will be appropriate) or learned from examples.

RBP can be formulated as a scaffold. It contains a fixed procedure that finds candidate plans, then organizes the refinement of candidate plans to find an effective plan. To find the candidate plans, it needs to call domain dependent operators that must be supplied or learned. The fixed procedure may code in dynamic programming, but it must plug in actions and a simulation that are domain dependent, and in order to do this it must have a module that recognizes which obstacles to applying operators may be affected, and specifies what state will be reached by contra-factual application of an operator (under the hypothesis that one or more affectable obstacles can be overcome). It also benefits from a supplied, domain dependent deadlock detector, which not only detects configurations of objects preventing success, but identifies the objects. But such modules as a deadlock detector, or a contra-factual state generator, can in principle be learned from supplied examples, and if appropriate examples can be supplied by a programmer (or generated automatically), this is a much smaller task than producing a program to solve the initial planning domain at one fell swoop.

Conclusion

My current working hypothesis is that understanding a domain requires having a library of scaffolds and modules exploiting the structure of the domain, either allowing problems to be immediately solved, or giving sufficient guidance that programs solving them can constructed with feasible amounts of computation. Such structure was evolved before humans came on the scene, and humans have built extensively on it, greatly increasing the power of human understanding over ape. Evolution threw vastly more computational power at this problem than we have available, and the human contribution involved the efforts of billions of brains, so constructing understanding systems may be difficult.

However, we have advantages that evolution didn't have-- namely our brains, which use all this previous structure. It is thus possible we can construct the requisite structures, and more likely still that we can get there by a collaboration of programming and computation. So the path I propose is that we set out to construct useful modules and scaffolds. When we come to a module embodying a concept that we can provide examples of, but not readily program, we attempt to produce it using a module constructor. If that is too difficult, we may supply useful sub-concepts, or recursively produce the useful sub-concepts from supplied examples and instructions. At any given step, both at top or at bottom, we have available automatic module construction as well as programming. That is, a module embodying a new concept can always call existing programs if their names are supplied in the instruction set to the module constructor; and if it can not readily be built like that, we may first try to produce (by training or programming) other useful sub-concepts. The only requirement is that enough guidance be provided, either by existing scaffolds, or by programmer intervention, or by supplying training examples, that each module construction be feasible.

Acknowledgements

Work funded in part by DARPA under grant #HR0011-05-1-0045 Thanks to Tom Schaul for writing the Sokoban program.

References

[1] Laird, J.E., A. Newell & P.S. Rosenbloom.(1987) SOAR: An Architecture for General Intelligence. Artificial Intelligence 33:1-64,.

[2] Baum, Eric. (2004) What is Thought? MIT Press, Cambridge, MA.

[3] Hutter, M. (2006) Universal Algorithmic Intelligence: A Mathematical Top->Down Approach,pp 228-291 in Artificial General Intelligence (Cognitive Technologies) (Hardcover) by Ben Goertzel and Cassio Pennachin (eds), Springer

[4] Schmidhuber, J.(2004) Optimal Ordered Problem Solver, Machine Learning 54, 211-254.

[5] Smith, W. D. (2006) Mathematical Definition of `Intelligence' (and Consequences), preprint, reference 93 on http://math.temple.edu/~wds/homepage/works.html

[6] Gray, P A. et al,(2004) Mouse Brain Organization Revealed Through Direct Genome-Scale TF Expression nalysis Science 24 December 2004: Vol. 306. no. 5705, pp. 2255 - 2257

[7] Lakoff, G., Johnson,(1999) M. Philosophy in the Flesh, the embodied mind and its challenge to western thought, New York, Basic Books.

[8] Dreyfus, H. L.,(1972) What Computers Can't Do, Cambridge MA MIT Press

[9] Myers, A. Xsokoban, http://www.cs.cornell.edu/andru/xsokoban.html

[10] Nugroho, R. P. ,(1999), Java Sokoban, http://www.billybear4kids.com/games/online/sokoban/Sokoban.htm

[11] Culbertson, J. C., (1997) Sokoban is PSPACE-complete. Technical Report TR 97-02, Dept. of Computing Science, University of Alberta.

[12] Boese, K. D. (1995) Cost Versus Distance in Traveling Salesman Problem, UCLA TR 950018, http://citeseer.ist.psu.edu/boese95cost.html

[13] Baum, E. B. & I. Durdanovic. (2000) Evolution of Cooperative Problem-Solving in an Artificial Economy. Neural Computation 12:2743-2775.

[14] Baum, E. B. & I. Durdanovic.(2002) An artificial economy of Post production systems. In Advances in Learning Classifier Systems, P.L. Lanzi, W. Stoltzmann & S.M. Wilson (eds.), Berlin: Springer-Verlag, 3-21.

[15] Schaul, T. (2005) Evolution of a compact Sokoban solver, Master Thesis,, École Polytechnique Fédérale de Lausanne, posted on http://whatisthought.com/eric.html

[16] Ellington, A. D., and J. W. Szostack, (1990), In vitro selection of RNA molecules that bind specific ligands. Nature 346:818-822.