Performance in Practical Problem Solving

Leo B. Hartman Josh D. Tenenberg

Computer Science Department
University of Rochester
Rochester NY 14627

Abstract

The quantity of resources that an agent expends in solving problems in a given domain is determined by the representations and search control strategies that it employs. The value of individual representations or strategies to the agent is determined by their contribution to the resource expenditure. We argue here that in order to choose the component representations and strategies appropriate for a particular problem domain it is necessary to measure their contribution to the resource expenditure on the actual problems the agent faces. This is as true for a system designer making such choices as it is for an autonomous mechanical agent. We present one way to measure this contribution and give an example in which the measure is used to improve problem solving performance.

1 Introduction

A primary goal of Artificial Intelligence research is to enable automated agents to have general reasoning abilities over a wide range of domains. Due to the inherent computational complexity of this task, system designers make *performance choices* that trade generality for improved resource use. Such performance choices are necessary for any agent with limited amounts of space and time to be effective within a dynamic world. Most running AI systems have embedded within them the choices that their designers felt would maximize their usefulness. Unfortunately, such choices rely on hidden assumptions, such as what variety and frequency of problems will be encountered, or now correct or close to optimal the eventual solution should be. Often, the designers themselves may not know what these assumptions are, since they are further obscured, for example, by choices implicit in the implementation of the representation language, or by insufficient prior information on the range of problems that the system may encounter. Unfortunately, this makes it difficult for other researchers to determine if the tradeoffs are appropriate for their problem domain.

This work was supported in part by the NIH Public Health Service under grant 1 R01 NS 22407-01, by the National Science Foundation under grant DCR-8602958, and by the Air Force Systems Command, Rome Air Development Center, Griffiss Air Force Base, New York 13441-5700, and the Air Force Office of Scientific Research, Boiling AFB, DC 20332 under Contract Number F30602-85-C 0008 which supports the Northeast Artificial Intelligence Consortium (NAIC).

The authors would like to thank the Xerox Corporation University Grants Program for providing equipment used in the preparation of this paper.

In this paper, we argue that performance choices should be made *explicit* in order to establish measures with which different choices can be compared. Further, we provide an example of how this might be done by developing a cost metric defined over search control strategies. The short-term benefit of this approach is that system designers will have a formalism which can be applied in determining preferred performance choices for the problems they are solving. The long-term benefit is that an automated agent will have the ability to evaluate its success in satisfying its goals in comparison with other candidate control strategies.

2 Intractability

Intelligent agents perform computations upon an internally stored *representation* of the world. Examples in [Levesque and Brachman, 1985] show how the choice of representation language bears on the computational complexity of solving problems using that language. As they point out, reasoning systems "will either be limited in what knowledge tney can represent, or unlimited in the reasoning effort they might require." For example, if we use a first-order predicate representation language, then there exists no algorithm that will decide for any theory T and sentence S encoded in this language whether S is a theorem of T. However, it might be the case that a particular theory T *in* which we are interested can be encoded within a weaker representation, such as finite automata. Then there exist algorithms of bounded computational complexity that will answer most questions with regard to this theory, such as if a string is accepted by the automaton. Unfortunately, no algorithm exists that decides whether some first-order theory can also be expressed in a weaker representation language.

We will say that a problem P is intractable if there is no expression language L in which to state problem instances of P sucn that there is a deterministic Turing machine that can compute solutions to such instances within polynomial time and space. The following, a version of the travelling salesman problem, is an example of a problem believed to be intractable. Imagine that an agent finds itself in a city and is given a set of n buildings to visit, along with information as to the distance between each pair of buidings. The question to determine is whether the agent can visit each building exactly once on a path no more than k units long. It is possible that the agent can answer this question quickly for some given set of parameters, but

it is believed that there is no deterministic algorithm (serial or parallel) that will solve every possible instance of this problem within an amount of time polynomial in the number of buildings in the problem. Note that this is not a function of the representation, but of the problem, since it is believed that there exists no representation for which this problem is tractable. Many problems that agents are called upon to solve are intractable.

The fact that an agent will encounter difficult problems does not mean that the agent should make no attempt to solve them. But in order to perform effectively, an agent must reason not about whether it can solve all possible problems optimally but, rather, how it can *maximize resource use over the entire sample of problems that it will encounter.* In order to do this, we believe that defining notions *of problem class* and *resource cost* is essential.

3 Problem Classes

Standard complexity theory typically defines a problem class in terms of the worst-case time or space behavior of a particular computational model, as a function of the size of the input problem. For example, the travelling salesman problem mentioned above falls in the class *NP-TIME,* which means that there is a non-deterministic Turing machine that solves every instance of a travelling salesman problem within time polynomial in the size of the input. The fastest known deterministic Turing machines for these problems take at least time exponential in the size of the input. Unfortunately, it is not known whether there are faster deterministic algorithms. It is in fact quite difficult to determine what is the fastest algorithm one can construct for a given problem.

There are several aspects of this definition of class which are inadequate for the task of optimizing an agent's use of its resources. First, each input problem instance to an algorithm is assumed to be in the correct form so that all operations of the algorithm are defined. For example, a typical sorting algorithm requires a list of elements from a set as input. The interpretation of this data structure is implicitly defined by the algorithm. Intelligent agents, however, are required to determine from general representations of its input which specific type of problem its input is an instance of. For example, if an agent has the goal of ordering a set of objects, it must recognize that this is a *sorting problem* and that there are algorithms for solving such problems. The difficulty of this recognition task will be a property of the language in which sorting problems are expressed.

Second, worst-case behavior is often too pessimistic. In practice the worst case may occur infrequently. Third, there might be restrictions one can place on the input that allow a subset of the problems to be solved quickly. For example, in the travelling salesman problem, if the cities are known to be collinear, then the solution can be trivially generated. Typically, general algorithms do not attempt to determine if the given input falls within one or the easier restricted subsets.

We will consider a much weaker notion of problem class. In general, an algorithm will define a problem class, in tnat a problem instance is a member of the class if the algorithm solves that instance.We can assume that any complex agent will be called upon to solve a variety of tasks, some as easy as finding the largest element from a set, and some at least as difficult as the travelling salesman problem. If a domain is encoded using a general representation for which there exists an algorithm capable of solving each problem expressible in this representation, then we can consider an agent using this algorithm as solving problems from a single class. That is, every problem that the agent encounters falls within the large problem class defined by the algorithm.

Many algorithms fail to distinguish between tasks of various difficulties, as was noted earlier with the travelling salesman problem. Due to the general nature of the algorithms, a high computational cost is incurred when solving both the easy and the hard tasks. Taking a more general example, suppose an agent's domain is represented in the first order predicate calculus, and the agent is able to solve each problem it is presented, i.e., each theorem to be proven, using a complete proof procedure. Unfortunately, these proof procedures take greater than exponential time in the length of the input to generate a proof in the general case, if a proof exists. Assuming that some of the input problems involve finding the largest element from a set, the agent is expending considerably more resource than necessary to solve these problems using the general proof procedure. That is, it the agent were able to distinguish these problems at low cost, then the agent could use one of the known polynomial time algorithms for their solution. This fact motivates the use of, for example, *procedural attachment* in theorem proving [Nilsson, 1980].

An agent, then, can be provided with a repertoire of algorithms, each one optimized to a particular set of problems. The agent can be considered to encounter problems from a variety of classes - one problem class for each algorithm. Unlike the standard complexity-theoretic model where the algorithms are never required to distinguish between the various restricted subcases, we assume that complex agents will be obliged to do so. There will therefore be a cost associated with determining into which class a given input problem falls, i.e, which algorithm should be used to solve the problem. There might thus be no computational advantage to an agent in having a large repertoire of algorithms. The usefulness of a member algorithm will be a function both of the *frequency* with which problems falling within this algorithm's class are encountered by the agent, and of the comparative costs of solving problems using this algorithm versus other candidate algorithms. By considering the entire set of encountered problems as falling into a variety of problem classes, the statistical properties of the various classes can be exploited to improve performance. A faster running time of an algorithm whose class occurs with high relative frequency will improve the overall performance of the agent. The longer a period of time over which an agent solves problems, the greater will be the gains from the algorithm optimization, assuming the relative frequencies remain invariant. Hence such sustained problem solving activity justifies the expenditure of resource to perform the optimization.

4 Costs

There are few precedents in AI that involve measuring costs. One example is the A* algorithm [Nilsson, 1980]. A* is an algorithm for searching a *state space,* described by an initial state, a goal state, and the legal transitions between states, which computes a path between the initial and goal states. This algorithm has the characteristic that it is *admissable,* meaning that the first path it finds that solves the problem will be of minimum length, if any solution patn exists. The algorithm works by searching through the current least-cost transition, where cost is a function of the number of transitions already taken on that path, and an estimate of the number of transitions required to reach the goal along that path. This cost, therefore, only measures the length of the solution. A* makes no attempt to model *the resources required to arrive at a state,* which includes the resources expended on the other incomplete or failed attempts up to that point in the computation. It is just such a measure of total computational costs that we are attempting to model. This replaces search for minimal *length* solutions with search for minimal *resource use* solutions.

A* places a premium on the optimality and correctness of every solution it reaches, which makes it generally un suited for the task of searching through complex domains. It will typically be cost-effective for an agent to trade some degree of minimality, completeness and correctness of the solution for a speed-up in the time required to compute it. For example, several polynomial time algorithms have been constructed that find sub-optimal solutions to NP-complete problems [Garey and Johnson, 1979]. Many of these algorithms are such that the larger the upper bound on the acceptable distance between an approximate solution and the optimal solution, the taster the algorithm returns one of these approximate solutions.

Augmenting the symbolic approach of AI by decision theoretic techniques is suggested in [Feldman and Sproull, 1977]. It is argued there that for many problems in AI there are natural numeric measures of cost and that it is only on the basis of such measures that good decisions can be made. That is, a decision theoretic form is an appropriate representation for certain problems. In the search for a solution to a problem instance, the paper discusses how to select between alternate plans, how to deal with the uncertainty of the outcome of plan steps and how to introduce knowledge producing actions into plans. The basic mechanism is to compute the expected utility of a problem instance's candidate solutions. Our goal here is to investigate the expected utility, or more specifically, the expected computational expenditure, of a problem solving strategy over an agent s entire sample of problem instances. Similar to choosing, from a set of candiates, a solution that maximizes expected utility, we intend to choose from a set of candidate problem solving strategies one that maximizes expected computational cost.

Also discussed in [Feldman and Sproull 1977] is the utility of additional planning. If an agent possesses a solution to a problem then in order for further planning effort to be profitable, the improvements in the solution must offset the additional cost. In a similar way the measurements and computation required to determine the expected computational cost of a strategy must be offset by the performance improvement that results from this improvement.

As mentioned earlier, finding lower bounds proofs is difficult for people, and will certainly be so for automated agents. This means that most agents will rarely be able to prove that they are doing the best possible. Therefore, it will usually be in the interests of an agent (and its designer) to express its performance choices in such a way that measuring the efficacy of these choices is possible. This requires considering all resource expenditure as cost, and expected improvements of one strategy over another as benefits. We claim that for agents who are called upon to make decisions in domains containing intractable problems, performance choices can only be made with respect to the sample of problem instances that an agent encounters. Whereas a particular performance choice may work quite well for a given problem sample, it may give equally poor performance for another sample. That is, agents cannot solve all problem instances optimally within such domains. By making the performance choices explicit, it is possible to evaluate whether a given choice is a good one for a particular sample. Thus, one is able to exploit any information about the sample of problems that is believed will be encountered in the future. Although the properties of future problems are unknown, predictions can be based upon the sample of problems that have already been seen. One of the long-term objectives of the approach advocated here is to state in a domain independent fashion those invariant properties of a particular domain or sample of problems that justify the belief that one has a good algorithm for the domain. Such domain independent properties also provide justification for applying the algorithm to other domains that are similar to the original domain with respect to these properties. Additionally, future performance improvements may also transfer across these similar domains.

5 Definitions

This section defines a cost metric for problem solving that illustrates some of the points raised above. This metric corresponds to the expected cost per problem that an agent expends. The metric is based on problem classes and the relative frequency that a given problem is a member of a particular class. The *solution strategy* used by an agent partitions problems into equivalence classes. One of these equivalence classes is intended to correspond to a set of problems of comparable difficulty or computational complexity. The cost metric makes clear how classes of difficult problems increase the expected cost and how an agent can exploit the presence of a class of easy problems.

We take the agent to consist of a search strategy. The agent exists in an environment in which problems are presented to it one at a time. The agent's response to the current problem is a solution to that problem obtained by the search strategy. We assume that the statistical properties of this seauence of problems presented to the agent are fixed. Specifically, we assume that the probability of members of a problem class being presented to the agent are constant and

that this probability is asymptotically approximated by the relative frequency of the problem class.

Let X be a countable set of problems. The problem instances that are presented to an agent are members of X, Let P be the set of solutions that correspond to the problems in X. We intend that P have a flexible interpretation. The members of P may be, for example, *literal* solutions as a sorted list is the solution to a sort problem. Alternatively the members of P may be algorithms that generate such literal solutions. However P is interpreted, the agent's problem-solving strategy selects members from P. Without loss of generality, every problem in X is assumed to have a solution in P. For the present purposes we also assume that whether a member of P solves a member of X is decidable.

Let s be the search function that the agent uses to generate solutions to problems. That is,

$$s : X \rightarrow P$$

Given a problem $x \in X$, $s(x)$ is a member of P that is a solution to x. Note that there may be several members of P that solve x but each search function selects only one such solution. Corresponding to each search function s is a function c_s such that the value $c_s(x)$ is the cost that s expends to solve problem x.

$$c_s : X \rightarrow \mathbf{R}$$

The function c_s is real-valued but may be interpreted as the number of units of any arbitrary resource, e.g., time, space or food.

The strategy s partitions the problems into classes on the basis of the solutions chosen by s. We will consider two problems equivalent with respect to s if s returns the same solution for both. That is,

$$x \sim_s y \quad \text{iff} \quad s(x) = s(y)$$

We use $[x]_s$ to denote the equivalence class of x under $\sim$

Consider, for example, the situation in which X is a set of motion planning problems, the set of solutions P is a set of specialized motion planning algorithms each of which generates a sequence of robot actions. When the agent is presented with problem x, namely, a description of the environment and the robot's initial and final state, s selects some $p \in P$, an algorithm that yields a motion plan. The equivalence class of problem x, $[x]_s$, is just the set of problems for which s selects algorithm $p = s(x)$ to generate the motion plan. Without placing further conditions on P, the set of problems for which p generates correct motion plans will in general properly contain $[x]_s$, the set of problems for which s selects p as a solution. In general the actual cost of executing p is not included in $c_s(x)$ but is part of the cost of obtaining a literal solution for x. This point is discussed further in §6.

A problem sample is the denumerable sequence of problem instances presented to the agent. We represent a problem sample as a set of pairs $<i\,x>$ consisting of an index i and a problem instance x. If $<i\,x> \in H$, problem instance x is the ith element of the sequence. Thus, a problem sample H is a subset of $\mathbf{N} \times X$ with the following properties:

if $<i\,x>, <i\,y> \in H$ then $x = y$

if $i \in \mathbf{N}$ then there is some x such that $x \in X$ and $<i\,x> \in H$.

A *prefix* H^n of a problem sample H is just the set of problem instances up to some point n in the sequence:

$$H^n = \{<i\,x> \mid <i\,x> \in H \;\&\; i \leq n\}$$

We can extend the notion of problem equivalence classes to sets of problem instances and thus to problem samples and sample prefixes. For problem x and set of problem instances H

$$[x\,H]_s = \{<j\,y> \mid <j\,y> \in H \;\&\; y \in [x]_s\}$$

If H is a problem sample then $[x\,H]_s$ is the subset of the sequence whose members are in the equivalence class of x as determined by strategy s. That is, $[x\,H]_s$ is the set of problem instances in H for which s returns the same solution as it returns for x. If a particular problem x appears more than once in H, each pair that includes x is in $[x\,H]_s$. Similarly $[x\,H^n]_s$ is the set of instances in H^n of problems equivalent to x. We use $[H]_s$ to denote the set of all equivalence classes of problem instances of H with respect to s:

$$[H]_s = \{[x\,H]_s \mid \exists i \; <i\,x> \in H\}$$

Similarly $[H^n]_s$ is the set of all equivalence classes of problem instances of H^n with respect to s.

Our cost evaluation of a strategy s for generating solutions is based on the above definition of equivalence classes of problem instances. The cost function for s can be extended in a simple way to members of a problem sample:

$$c_s(<i\,x>) = c_s(x).$$

Note that for problems x, y in the same equivalence class, we will not in general have $c_s(x) = c_s(y)$. Even though s obtains the same solution for x and y, s may expend more effort in one case than in another. Thus, for a prefix H^n the average cost of solving a problem instance in $[x\,H^n]_s$, denoted by $c_s[x\,H^n]_s$, is the sum of the cost of each individual problem divided by the size of $[x\,H^n]_s$. That is,

$$c_s[x\,H^n]_s = \frac{1}{|[x\,H^n]_s|} \sum_{<i\,y> \in [x\,H^n]_s} c_s(<i\,y>)$$

where $|A|$ denotes the cardinality of set A.

Let $\%(A\ B)$ denote the relative frequency of members of set A in set B. For finite sets A and B, where A is a subset of B,

$$\%(A\ B) = \frac{|A|}{|B|}$$

Then the relative frequency with which problems in $[x\,H^n]_s$ appear in H^n is

$$\%([x\,H^n]_s\ H^n) = \frac{|[x\,H^n]_s|}{|H^n|} = \frac{|[x\,H^n]_s|}{n}$$

Consider a single problem class $q \in [H^n]_s$. That is, let q be a set of equivalent problem instances in H^n with respect to search strategy s. The part of the average cost incurred by s on H^n that is attributable to q is just the product of the average cost that s expends on problems in q and the relative frequency of q in H^n:

$c_s(q)\,\%(q\,H^n)$

Now summing over all such equivalence classes of problem instances, the average cost incurred by s to solve a problem in H^n is

$$C_s(H^n) = \sum_{q \in |H^n|_s} c_s(q)\,\%(q\,H^n)$$

If $c_s(q)$ is bounded for all problem classes q in H then from the assumption that the statistical properties of H are fixed we have that

$$C_s(H) = \lim_{n \to \infty} C_s(H^n)$$

is well defined and finite. This limit is the expected cost that s expends to solve a problem in H. For a particular s by recording the average cost and relative frequency of problem equivalence classes we can evaluate $C_s(H^n)$ and approximate $C_s(H)$. Thus we have some information about what performance we can expect from s on problem sample H. It is here that the invariance of problem class frequencies play a role. As shown in the next section this performance measure can be the basis for comparing and evaluating two different problem solution strategies.

6 Example: A hybrid algorithm

Having defined a measure of computational cost, the question arises as to how it may be used. The following example illustrates the use of this measure to improve performance. A problem solution strategy is defined such that, given a problem, it selects an algorithm whose output is the solution to the problem. From a family of such strategies we show how to choose the optimal one according to the performance measure.

Consider a sequence of algorithms A1 *A2* ... A_n, each member of which terminates on all input, takes as input an encoding of a problem and yields as ouput a solution to the problem or an indication of failure. Let the sequence have the additional property that any member of the sequence solves at least all of the problems of its predecessor. We refer to this property of a sequence of algorithms as the *subsumption* property. A specific instance of such a sequence of algorithms can be constructed to address the motion planning problem in robotics. The motion planning problem, e.g., [Lozano-Peres, Wesley, 1979], is to find a collision free path for an object from an initial position to a desired final position. For an object B, if we have a sequence of object approximations *B1 B2* .. Bn *-B* and a motion planning algorithm *M*, we can specialize *M* by each of the Bi's to yield *M1 M2*... M_n so that M_i computes paths only for Bj. That is, M_i computes paths for the ith approximation of object B. if B; strictly contains its successor in the sequence and if M is suitably well-behaved then the Mi's have the subsumption property. That is, since *Mi* uses an approximation to the object that is bigger than that used by *Mi* + /, Mi; can only find a path if *Mi +1* does (figure 6.1).

We are appealing to the intuition that giving up detail and *completeness,* i.e., the property that a solution is found if one exists, enables us to obtain more quickly the solutions we do find. In ascending order the Bi's are increasingly better approximations to the original object. Similarly, the *Mi's* are increasingly better

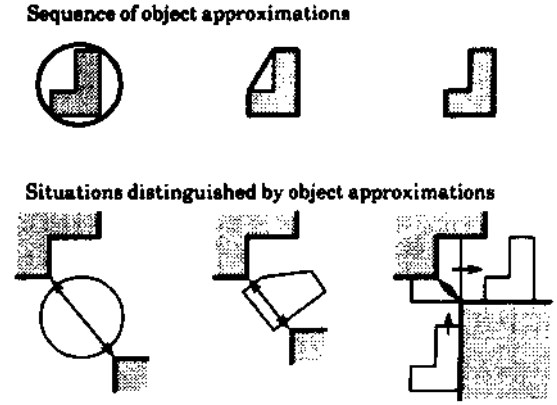

Figure 6.1 Object approximations generate motion planning algorithms

An example of a sequence of object approximations that can be used to define a sequence of motion planning algorithms with the subsumption property.

approximations to a complete algorithm to find motion plans for the original object. In this instance we may profit from giving up the precision of a complete algorithm by being able to use representations of lower combinatorial complexity.

Now let the search strategy *s* be such that, given a problem, *s* applies the A;'s in turn until one of the Aj's returns a literal solution to the problem. Such a search strategy, which we will refer to as a *hybrid algorithm,* is unambiguously specified by its component algorithms. Intuitively *s* has generality and also exploits the presence of simple problems. The strategy *s* can be as general as we want by appending to the sequence the most general known algorithm. In particular, *s* can be made complete if a complete algorithm is known. Because the simpler and, presumably, faster approximation algorithms are applied first, *s* solves easy problems quickly. To some degree a hybrid algorithm invests an appropriate amount of effort in solving a problem instance.

In the interests of applying the definitions of §5 we interpret s as selecting one of the A_i's. That is, the solution that s returns is a member of $\{A_1\ A_2\ ..\ A_n\}$. The A_i selected by s is one that can be executed to obtain a literal solution to the problem but trivially, by the time s selects A_i, we already have the literal solution that A_i generates. Now according to the previous definitions there is an equivalence class of problems associated with each of the A_i's. Specifically, any problem x is in the equivalence class associated with A_i if A_i solves x (i.e., produces a literal solution for x), but no A_j, $j<i$, solves x. We refer to this equivalence class by $[A_i\,H]_s$ and note that $[A_i\,H]_s = [x\,H]_s$ for any problem x that s solves by selecting A_i.

As before $\%([A_i\,H^n]_s\,H^n)$ is the relative frequency of members of $[A_i\,H^n]_s$ in H^n, the first n problems of sample H. If $c(A_i\,H^n)$ is the average cost of executing A_i on the problems of H^n then

$$c([A_i\,H^n]_s) = \sum_{j \le i} c(A_i\,H^n)$$

This is the average cost that s incurs when s is applied

to H^n and selects A_i. The total cost $C_s(H^n)$ that s incurs on H^n is

$$C_s(H^n) = \sum_{a \in \{A_1 \ldots A_n\}} c_s([a\ H^n|_s])\,\%([a\ H^n|_s\ H^n])$$

Given this measure, it can be determined if each component algorithm is providing more benefit than the cost it is incurring. That is, although the expected cost of solving a given problem with any A_k, $k<i$, is less than with A_i, the cost of executing each such A_k is incurred in solving every problem from the class $[A_i\ H^n]_s$. Classes that are too expensive or of low relative frequency can be dropped from s to yield a new strategy that has better overall performance. Given measurements of the relative frequencies and costs for the problem classes a dynamic programming method can be used to select the subset of component algorithms with lowest expected cost for the current problem sample [Hartman, 1987]. Note that if we take a hybrid algorithm that is optimized in this way for a particular problem sample and apply it to a sample with different statistical properties, not surprisingly, we are likely to obtain different performance.

In addition to obtaining a globally optimal set of component algorithms for a particular problem sample, we can also determine under what conditions an incremental change to s, e.g., the insertion of a new component algorithm into the sequence, yields an improvement on the expected cost per problem. That is, we know where candidates for future improvements may be found.

We do not claim that this is the best way to solve any particular kind of problem. The optimization is only within the narrowly defined class of hybrid algorithms. However, the example does show how the cost measure defined in the previous section can be used to optimize performance, In particular, the Derformance improvement is independent of the actual representations and component algorithms used. It does not depend on increasing knowledge about how to solve the particular problem such as, in this case, motion planning. The improvement is based solely on the invariant statistical properties of the problem sample and measurement or performance.

7 Conclusion

Agents in real world domains will be called upon to solve difficult problems. Although an agent might choose to expend arbitrarily large amounts of a resource in solving a particular problem, the opportunity costs of doing so typically make such a choice prohibitively expensive. This resource would be better spent in either solving approximations of this problem, ignoring this problem completely and solving other, easier problems, or improving the agent's general ability to solve these difficult problems. This is as true for agents who introspect upon their own problem solving ability as it is for designers who attempt to ascertain the utility of AI systems that they develop. We argue that these *performance choices* should be made explicit, and that they should be based upon the resource cost incurred in solving actual problem samples.

We have provided one example of a resource cost measure. Although by no means definitive, it measures the search costs associated with finding the solution to a given sample of problems using a given search strategy. As opposed to previous formal analyses of search strategies within artificial intelligence, the emphasis here is not upon the efficiency of the solution, as it is in A*, but upon the amount of resources expended in finding solutions. Central to this endeavor is the concept of dividing a problem space into a set of problem classes. We can measure both the frequency with which a search strategy places problems from the sample within a class, ana the average cost of solving each problem placed into a class. This allows one to demonstrate improved performance by changing the search strategy to place more of the problems into tnose classes for which solutions can quickly be generated, or to ascertain more quickly into which class a problem belongs. Cost measures such as the one described here, will be necessary, we believe, in order to maximize the resource use of an agent which must solve many problems drawn from an intractable class.

Acknowledgements

We would like to thank our advisor, Dana Ballard, for his patience and insight, and for the opportunity to do this research.

References

Feldman, J.A., Sproull, R.F., Decision Theory and Artificial Intelligence II: The hungry monkey, *Cognitive Science* 2, pp.158-192 (1977).

Garey, M.R., Johnson, D.S., *Computers and Intractibility,* W.H. Freeman, New York (1979).

Hartman, L., The practical solution of geometric problems, Technical Report 199, Computer Science, University of Rochester, in preparation (1987).

Levesque, H.J., Brachman, R.J., A fundamental tradeoff in knowledge representation and reasoning, in Levesque, H.J., Brachman, R.J. (eds.), *Readings in Knowledge Representation,* Morgan Kaufmann, Palo Alto CA (1985).

Lozano-Peres, T. Wesley, M.A., An algorithm for planning collision-free paths among polyhedral obstacles, C*ACM 22,*10 (1979).

Nilsson, N.J., *Principles of Artificial Intelligence,* Tioga Publishing, Palo Alto CA (1980).