Speeding Up Problem Solving by Abstraction: A Graph Oriented Approach

R.C. Holte, T. Mkadmi,
Computer Science Dept., University of Ottawa, Ottawa, Ontario, Canada K1N 6N5.
holte@csi.uottawa.ca

R.M. Zimmer,
Computer Science Dept., Brunel University, Uxbridge, England, UB8 3PH.
Robert.Zimmer@brunel.ac.uk

A.J. MacDonald
Electrical Eng. Dept., Brunel University, Uxbridge, England, UB8 3PH.
Alan.MacDonald@brunel.ac.uk

Abstract

This paper presents a new perspective on the traditional AI task of problem solving and the techniques of abstraction and refinement. The new perspective is based on the well known, but little exploited, relation between problem solving and the task of finding a path in a graph between two given nodes. The graph oriented view of abstraction suggests two new families of abstraction techniques, algebraic abstraction and STAR abstraction. The first is shown to be extremely sensitive to the exact manner in which problems are represented. STAR abstraction, by contrast, is very widely applicable and leads to significant speedup in all our experiments. The reformulation of traditional refinement techniques as graph algorithms suggests several enhancements, including an optimal refinement algorithm, and one radically new technique: *alternating search direction*. Experiments comparing these techniques on a variety of problems show that *alternating opportunism* (AltO) a variant of the new technique, is uniformly superior to all the others.

1. Introduction

Path-finding, the task of finding the shortest path between two given nodes in a graph, has been studied in computer science (CS) for almost forty years. Theoretical advances are still being made [Cherkassky et al., 1993] and real-world applications abound. For example, there are commercial products that find routes optimizing the cost or time to drive between two locations in a city or country.

Path-finding arises as a subtask in many areas of artificial intelligence (AI). For example, in natural language understanding, lexical disambiguation has been partially solved by finding paths between word senses in a knowledge-base [Hirst, 1988]. Another example is PROTOS [Porter et al., 1990], a case-based knowledge acquisition and reasoning system, that "explains" the differences between two cases' features by finding paths connecting them in a conceptual network. Path-finding is also an integral part of robot motion planning. Although a robot's state-space and actions are continuous, modern techniques [Donald et al., 1993; Kavraki and Latombe 1993, 1994] for motion planning discretize the space, use standard CS techniques to find a path in the graph created by discretization, and then transform the path into a continuous motion.

Path-finding is intimately related to the well-studied AI tasks of problem solving, planning, and heuristic search. The techniques developed for these tasks in AI differ from CS path-finding techniques in three main ways. Firstly, AI techniques extend and speed up the CS techniques by incorporating search control knowledge, such as problem-specific heuristics. Secondly, unlike the CS techniques, the techniques developed in AI are not always guaranteed to find optimal paths. The aim in AI is to develop techniques that very quickly find *near-optimal* paths.

The third difference between AI and CS techniques concerns how graphs are represented. In CS graphs are typically represented explicitly, for example with an adjacency matrix or with an explicit list of successors for each node. By contrast, graphs are represented implicitly in almost all AI research (for exceptions see [Banerji, 1980] and work cited therein). In an implicit representation each node is represented by a description in some language and its successors are computed by applying a set of successor functions (*operators*) to the node's description. In AI the language used to represent nodes and operators is usually a STRIPS-like notation [Fikes and Nilsson, 1971]. A node is described by a set of logical predicates (*literals*) and an operator is described by a precondition. A precondition is a set of literals a node must satisfy in order for the operator to be applicable to it, together with lists specifying which literals the operator adds to and/or deletes from the node's description when applied.

The speedup afforded by AI techniques would be welcome in any application currently using CS techniques. For example, in [Donald et al, 1988] the authors propose to replace their current search technique, breadth first search, by A^* [Hart et al., 1968].

There are two obstacles preventing AI techniques from being used to speed up applications currently using CS techniques. The first is that in these applications the graphs are represented explicitly, whereas the AI techniques require the graphs to be represented in the STRIPS notation. In fact, there is a generic way of encoding any explicit graph in the

STRIPS notation with just one operator and two predicates (see figure 1).

Operator	*Preconditions*	*Deletes*	*Adds*
$move(node_1, node_2)$	$\left\{ \begin{array}{c} at(node_1), \\ edge\_exists(node_1, node_2) \end{array} \right\}$	$\{at(node_1)\}$	$\{at(node_2)\}$

Fig. 1. Generic way of encoding an explicit graph in STRIPS notation.

The only other obstacle to speeding up path-finding applications by using AI techniques is that good search control knowledge is needed to do so. Without good heuristics, for example, A^* is no faster than standard CS techniques.

Unfortunately, good search control knowledge is often not readily available [Prieditis, 1993]. Recognizing this, some AI research has investigated techniques to automatically generate search control knowledge, either by learning from experience (e.g. [Minton, 1990]) or by analyzing the description – i.e. the implicit representation – of the graph (e.g. [Korf, 1985a], [Etzioni, 1993]).

Abstraction is the most widely studied analytical technique for creating search control knowledge. The general idea is to create from the given graph G a "simpler" graph $\widehat{G}$. To find a path between two nodes in G, one first finds a path $\widehat{p}$ between the corresponding nodes in $\widehat{G}$ and then uses $\widehat{p}$ to guide the search for the path in G. Various methods for using $\widehat{p}$ to guide search have been studied; the most common one is called *refinement*.

Having dealt with the only obstacles, it would seem that applications currently using CS path-finding techniques could be speeded up simply by using existing AI abstraction techniques. However, there is one additional obstacle that was not initially apparent. Existing abstraction techniques depend for their success on being given a "good" implicit representation. Consider, for example, ALPINE, a state-of-the-art abstraction system [Knoblock, 1994]. ALPINE can construct useful abstractions given certain implicit representations for the Towers of Hanoi graph (defined in Appendix A) but fails completely if given other implicit representations (e.g. the single-operator representation on p. 295 of [Knoblock, 1994]). As it happens, the generic way to encode an explicit graph in the STRIPS notation is a "bad" representation for ALPINE.

In this paper we investigate abstraction and refinement as techniques for searching explicitly represented graphs. This "graph-oriented" approach to these topics is very different from the "description-oriented" approach traditionally taken in AI. The main practical motivation for taking a new approach has been outlined in the preceding paragraphs: there exist important applications, even within AI, with explicitly represented graphs for which there is no known implicit representation that is "good" for existing abstraction techniques.

At the technical level, the main contributions of this paper are new high-performance abstraction and refinement techniques. The new abstraction technique, STAR abstraction, is specifically designed for explicitly represented graphs. Unlike any previous technique STAR abstraction is guaranteed to speed up search in a wide range of commonly occurring circumstances. Several new refinement techniques are presented, including one (AltO) that is superior in terms of robustness and performance. Although developed by analyzing existing refinement techniques from the graph-oriented perspective, all the refinement techniques can be applied equally well to explicitly or implicitly represented graphs. The combination of STAR abstraction and AltO refinement often produces impressive speedup. For example, when applied to a road map of Pittsburgh, the "search effort" (defined below) required to find a route between two locations is almost 30 times less using STAR/AltO than using breadth-first search. Moverover, the routes found are within 33% of the optimal length.

Because of their robustness, our new techniques also provide a fail-safe backup to existing techniques in the case of graphs that have a succinct implicit representation and are also sufficiently small that it is feasible to represent them explicitly. To illustrate this point, consider the Blocks World (defined in Appendix A). In [Bacchus and Kabanza, 1994] it is shown that, in the absence of domain-specific search control knowledge, both the paper's own planner and SNLP [McAllester and Rosenblitt, 1991] suffer from a combinatorial explosion with as few as 6 blocks. As there is a standard, succinct, implicit representation for the Blocks World, it is natural to apply ALPINE to this space. Unfortunately the standard representation is particularly bad for ALPINE; it produces no abstraction (p. 296, [Knoblock, 1994]). But with 6 blocks the graph contains only 7057 states and can readily be abstracted and searched using our new techniques.

In addition to these technical contributions, we feel that, at a more general level, the graph-oriented view of abstraction is itself a research contribution. It gives a fresh slant on an old subject which we have found particularly fertile. In the graph-oriented view an abstraction is defined by partitioning a graph's nodes, and *any* partitioning is a legitimate abstraction. This view immediately suggests a host different abstraction (partitioning) techniques. Two are discussed in detail in this paper, but we have experimented with several others, and our software is written to make it easy to define new ways of partitioning (abstracting) and to combine multiple partitioning methods in an arbitrary manner. Likewise, the graph-oriented view of refinement is simple and thought-provoking. Refinement is any technique for "caching" the results of search in the abstract graph and using them to guide (order or prune) search in the original graph. Described this way, possible refinement methods abound. In this paper alone we introduce edge-path refinement, node-path refinement, opportunism, first-successor versus all-successors, alternating direction, and path-marking. It is worth mentioning, although it will not be further discussed in this paper, that A^* [Hart et al., 1968], HPA [Pohl, 1970], and Graph Traverser [Doran and Michie, 1966] are also members of this broad family of refinement techniques [Holte et al., 1994, 1995]. This abundance of techniques for abstraction and refinement is unique in the literature, and is a direct result of the fertility of the graph-oriented approach. It naturally leads to many ideas and, in addition, makes them simple to implement.

Section 2 gives definitions for problem-solving, abstraction, and refinement as they typically appear in the literature. The techniques described in this section are referred to as *classical*, to distinguish them from the novel techniques developed in sections 4 and 5. Section 3 gives the graph-oriented counterparts of these definitions. Section 4 introduces two variations on classical refinement, a technique for finding the optimal refinement of an abstract solution, and a radically new technique, Alternating Search Direction. Section 5 presents two new families of abstraction techniques, one of which is the "Algebraic" abstractions. Algebraic abstractions have certain desirable properties, but are shown in an experimental study to be highly sensitive to the way in which a state space is described: similar descriptions of the same space can lead to very different performance of the algebraic abstraction techniques. The development of a new, robust abstraction technique begins with an simple analysis of the work involved in solving a problem using an abstraction hierarchy. This analysis produces specific recommendations which are the basis for a new abstraction algorithm. The new abstraction algorithm and refinement techniques are evaluated experimentally in section 6.

1.1 Methodological Comments

In addition to the major experiment in section 6, various experimental results are reported throughout sections 4 and 5. With the exception of the experiment in section 5.1, the purpose of these experiments is to compare the performance of two algorithms or to examine the effect on performance of some particular parameter. Generally speaking these follow the usual pattern for such experiments. But there are some points about their experimental design that deserve comment.

We feel it is important methodologically to use a diverse set of "domains" in any experiment in order to gain some appreciation of the generality of the results. The graphs used in our experiments are described in Appendix A. Three of these (Bitnet, KL-2000, and Permute-7) are drawn from real applications. One (Words) is a real graph that is not associated with any particular application, and three (5-puzzle, Blocks-6, and TOH-7) are based on puzzles. The puzzles were included because they are easily generated, can be scaled to a convenient size, and are widely studied in AI. An interesting outcome of our experiments is that these "toy" domains are not any easier for our systems than the "real" ones; if anything, performance on the "toy" domains is poorer than average.

In these experiments two performance measurements are of interest: solution length and "work". Solution length is of interest because abstraction, in combination with refinement, is not guaranteed to produce optimal solutions. Work is what abstraction aims to reduce: the purpose of abstraction is to speed up problem-solving. We originally measured work in terms of CPU time, but this was abandoned for three reasons: first, because it would have prevented us from executing the experiments on different CPUs; secondly, because it proved extremely sensitive to low-level programming details of no significance to the algorithms themselves; and finally, because our implementation is designed for flexibility (it is an experimental workbench). It is far from being optimally coded, or even equally-optimally coded for the various search techniques we compared. Instead of CPU time we measured *edges traversed*, which is closely related to the traditional *nodes expanded*. When a node is expanded, all of its neighbours are computed. Traditionally this counts as 1 unit of work; our measure counts the number of neighbours generated.

In addition to systems that we particularly wished to evaluate, we have included in our experiments systems whose performances provide useful points of reference. The *classical refinement* system, for example, is representative of existing refinement algorithms. The *optimal refinement* algorithm provides a bound on the solution lengths that could possibly be produced by systems in the same family as classical refinement. Breadth-first search provides the optimal solution length and also is representative of standard CS techniques.

Section 5 is the most interesting from a methodological point of view. The experiment in 5.1 was devised in response to casual observations made during the use of our first system, which was based on algebraic abstraction. Certain failings of the system were at first dismissed as unlucky accidents, but when they persisted we decided a systematic examination was needed. We identified particular aspects of the system we suspected as the source of difficulty. These are presented in section 5 as the defining characteristics of algebraic abstractions but they originally had no such special status. A system was built whose the express purpose was testing if these characteristics alone would cause the failings we had observed. The experiment confirmed our suspicions and forced us to investigate methods of abstraction that did not have these problematic characteristics.

The general research methodology underlying section 5 is one we feel would be effective in many areas in AI. For example, it has produced novel and significant contributions in the field of machine learning [Holte et al, 1989; Holte, 1993]. The key idea is to actively investigate the weaknesses in a system's behaviour with the aim of identifying the algorithmic sources of those weaknesses.

2. Problem Solving, Abstraction and Refinement: Standard Definitions

A *problem space* is a set of states and a set of *operators*, where an operator is a partial function[1] mapping states to states. A *problem* is a pair of states, $\langle start, goal \rangle$[2], and a solution to problem $\langle start, goal \rangle$ is a sequence of operators that maps $start$ to the $goal$. Problem solving is the task of finding a solution to a given problem. Except for special classes of problems, problem solving is a search intensive process.

The majority of research on problem solving has used a STRIPS-like notation [Fikes and Nilsson, 1971] to represent problem spaces, and defined problem space in terms of this notation as follows: A *problem space* is defined as a set of states, a set of operators, and a formal language (containing constants, variables, function symbols, predicate symbols, etc.). A *state* is a set of sentences in the formal language. An operator maps one state to another by adding to or deleting from the set of sentences (i.e. the state) to which it is applied. The preconditions of an operator, which specify the states to which it may be applied, are stated in the formal language.

An abstraction of a problem space P is a mapping from P to some other problem space, $\widehat{P}$. It is worthwhile to distinguish three types of

---

[1][Knoblock, 1994] assumes the application of an operator to a state produces a unique next state (p.249). [Bacchus and Yang, 1994] defines an operator to be a partial function but points out that most systems actually use operator "templates" not operators (p.68). As we shall see in section 5.1, requiring operators to be functions has far reaching consequences: it is expensive to enforce and it makes an abstraction system extremely sensitive to the exact manner in which a problem space is defined.

[2]Everything in the paper extends readily to the more common definition in which there may be multiple goal states (and multiple start states).

mappings between problem spaces: embeddings, restrictions, and homomorphisms[3].

If an embedding is applied to P, the result is a problem space that includes all of P and more besides. The best known embedding techniques are dropping operator preconditions and extending the set of operators with macro operators. A restriction does the opposite; if applied to P the resulting problem space is embedded in P. This type of mapping is not widely studied, but examples may be found in [Fink and Yang, 1993] and [Prieditis and Janakiraman, 1992]. A homomorphism is a many-to-one mapping from P to $\hat{P}$ that "preserves behaviour" (an exact definition is given in the next section). [Korf, 1980], [Knoblock, 1994], and [Yang and Tenenberg, 1990] use homomorphic mappings of problem spaces. Hybrid systems are possible but uncommon. One example, described in [Knoblock et al., 1991a], is PRODIGY with both EBL (which creates macros and therefore results in an embedding) and ALPINE (which is homomorphic). In the literature, the term *abstraction* is applied equally to all three types of mapping. In this paper we shall restrict its use to homomorphic mappings. This usage encompasses all the widely studied modern "abstraction" techniques except for dropping operator preconditions[4].

Within the STRIPS framework, abstraction is achieved by removing symbols from the formal language and from the definitions of the operators and states. This has several effects, all of which are intended to make problem solving much faster in the abstract space than in the original space. There are usually many fewer states: two states that differ only in symbols removed from the language are indistinguishable in the abstract space. This reduces the size of the problem space. Some operators become identities, because all the predicates they add and delete have been removed from the language: this can reduce the branching factor in the space. The abstract space will sometimes also have a higher density of goal states than the original space. Consider, for example, an abstract space which contains only two states: assuming that there is at least one goal state in the original space, then the abstract space has a solution density of at least 50%.

---

[3]There are problem space mappings of other types (such as isomorphisms [Korf, 1980]), but the term "abstraction" is not normally applied to these.

[4]All three types of mappings can be treated accurately in a graph-oriented framework. The reason to distinguish them is quite simply because they have very different formal properties. For example, embeddings introduce redundancy which causes the so-called "utility problem" (for macro-operators, see [Minton, 1985; Greiner and Jurisica, 1992]; for dropping preconditions, see [Valtorta, 1984; Hansson et al., 1992] where it is proven that this type of embedding cannot speedup A^* search). By contrast, restrictions and homomorphisms do not introduce redundancy: they have difficulties of their own, but not a utility problem caused by redundancy.

The abstract solution will not usually be a valid solution in the original space. When the symbols removed to create the abstraction are taken into account, some of the operators in the abstract solution may have unfulfilled preconditions and some of the predicates in the final goal state may not be established. Nevertheless, there are several ways that an abstract solution can be used to guide search for a solution in the original space. For example, its length can be used as a heuristic estimate of the distance to the goal [Pearl, 1984; Prieditis and Janakiraman, 1993]. Alternatively, its individual steps can be used as a sequence of subgoals whose solutions link together to form the final solution [Minsky, 1963; Sacerdoti, 1974; Chakrabarti et al., 1986; Yang and Tenenberg, 1990; Knoblock, 1994]. In the latter case, the abstract solution serves as a skeleton for the final solution. The process of "fleshing this skeleton out", called *refinement*, involves inserting operator sequences between the operators in the abstract solution. Refinement is the technique for using abstract solutions that will be discussed in this paper; elsewhere we have studied the "heuristic" use of abstract solutions and the relation between the two techniques [Holte et al., 1994, 1995].

The following description of the refinement process is based on the account in [Knoblock et al., 1991b]. In this section, and in the remainder of the paper, we shall generally denote the abstraction of an object x by $\widehat{x}$. Let ϕ be the mapping that maps an operator f in the original space to the abstract operator $\widehat{f}$. Thus $\phi^{-1}\left(\widehat{f}\right)$ is the set of operators that *refine* $\widehat{f}$. Now consider refining the abstract solution $\widehat{f}_1 \cdots \widehat{f}_n$ for the problem $\langle start, goal \rangle$. In outline, the refinement algorithm builds a final solution by constructing sequences f_i^* that are spliced in between f_{i-1} and f_i. Finally, if necessary, a sequence f_{n+1}^* is added after f_n.

In more detail, the procedure is as follows: First set $i = 1$, and $s_i = start$. Find any sequence of operators f_1^* mapping s_1 to a state s_1' that satisfies the preconditions of some $f_1 \in \phi^{-1}\left(\widehat{f}_1\right)$. If the preconditions of f_1 are satisfied by s_1 itself then f_1^* is the identity (empty sequence) and $s_1' = s_1$. Next, apply f_1 to s_1' to get s_2, increment i, and repeat this process until the operator sequence $f_1^* f_1 f_2^* f_2 \cdots f_n^* f_n$ has been constructed. This sequence maps s_1 ($start$) to some state s_{n+1}. The state s_{n+1} is guaranteed to be equivalent to the goal state under the abstraction mapping ϕ, but it may not be equal to the goal state. If it is not, construct a sequence f^*_{n+1} mapping s_{n+1} to the goal state. The final solution mapping start to goal, then, is $f^*_1 f_1 f^*_2 f_2 \cdots f^*_n f_n f^*_{n+1}$.

The two disk Towers of Hanoi

We shall illustrate the use of abstraction and refinement with the two disk Towers of Hanoi problem (defined in Appendix A).

Operator	*Preconditions*	*Deletes*	*Adds*
$move\_small(p_1, p_2)$	$\begin{Bmatrix} small\_on(p_1), \\ p_1 \neq p_2 \end{Bmatrix}$	$\{small\_on(p_1)\}$	$\{small\_on(p_2)\}$
$move\_large(p_1, p_2)$	$\begin{Bmatrix} small\_on(p_3), \\ large\_on(p_1), \\ p_1 \neq p_2, \\ p_1 \neq p_3, \\ p_2 \neq p_3 \end{Bmatrix}$	$\{large\_on(p_1)\}$	$\{large\_on(p_2)\}$

Fig. 2. The two disk towers of Hanoi problem.

In this problem the p_i are variables which stand for pegs, and specific pegs have the names 1, 2, and 3.

Example 1

If this problem space is abstracted by deleting predicate $small\_on$, the operators are as follows:

Operator	*Preconditions*	*Deletes*	*Adds*
$\overline{move\_small}(p_1, p_2)$	$\{p_1 \neq p_2\}$	$\{\ \}$	$\{\ \}$
$\overline{move\_large}(p_1, p_2)$	$\left\{\begin{matrix} large\_on(p_1), \\ p_1 \neq p_2, \\ p_1 \neq p_3, \\ p_2 \neq p_3 \end{matrix}\right\}$	$\{large\_on(p_1)\}$	$\{large\_on(p_2)\}$

Fig. 3. First abstraction of the two disk towers of Hanoi.

The abstract operator $\overline{move\_small}$ is the identity, and $\overline{move\_large}$ moves the large disk onto any peg.

Suppose we wish to use this abstraction to solve the following problem:

start state: $\{small\_on(1), large\_on(1)\}$
goal state: $\{small\_on(3), large\_on(3)\}$.

We first abstract the problem and then we solve it. The abstracted problem is:

start state: $\{large\_on(1)\}$
goal state: $\{large\_on(3)\}$.

Suppose we find the shortest abstract solution. This is $\overline{move\_large}(1, 3)$. This abstract solution is refined as follows to produce our final solution. Since $f_1 = move\_large(1, 3)$ is not directly applicable to the start state, we need to find a sequence of operators that maps the start state to a state in which f_1 can be applied. We obtain:

$f_1^* = move\_small(1,2)$,
$s_1 = \{small\_on(2), large\_on(1)\}$.

Now we apply f_1, to get

$s_2 = \{small\_on(2), large\_on(3)\}$.

There are no further operators in the abstract solution, but s_2 is not the goal state, so we need to find a sequence of operators that maps s_2 to the final goal state.

$$f_2^* = move\_small(2,3)$$

The complete solution, then, is $f_1^* f_1 f_2^*$, i.e.,

$$move\_small(1,2),\ move\_large(1,3),\ move\_small(2,3).$$

Example 2

Equally, we could have abstracted the space in example 1 by deleting the predicate $large\_on$. This leaves $move\_small$ unchanged while $move\_large$ is abstracted to the identity operator:

Operator	*Preconditions*	*Deletes*	*Adds*
$\overline{move\_small}(p_1, p_2)$	$\left\{\begin{matrix} small\_on(p_1), \\ p_1 \neq p_2 \end{matrix}\right\}$	$\{small\_on(p_1)\}$	$\{small\_on(p_2)\}$
$\overline{move\_large}(p_1, p_2)$	$\left\{\begin{matrix} small\_on(p_3), \\ p_1 \neq p_2, \\ p_1 \neq p_3, \\ p_2 \neq p_3 \end{matrix}\right\}$	$\{\ \}$	$\{\ \}$

Fig. 4. Second abstraction of the two disk towers of Hanoi.

The shortest abstract solution to our problem in this abstract space is:

$$\overline{move\_small}(1,3)$$

The refinement of this solution proceeds as follows. The first operator in the abstract solution is directly applicable to the start state, so f_1^* is the identity and $s_1' = s_1$ is the start state. Applying f_1 to s_1' produces the state

$$s_2 = \{small\_on(3), large\_on(1)\}.$$

The state s_2 is equivalent to the goal state but not equal to it. We now search for a sequence of operators, f_2^*, leading to the goal state from this state. One such sequence is

$$f_2^* = move\_small(3,2), move\_large(1,3), move\_small(2,3).$$

The final solution, then, is

$move\_small(1,3), move\_small(3,2),$

$move\_large(1,3), move\_small(2,3)$

The first thing to notice about this solution is that it is not optimal. This is often the case with solutions constructed by refinement. Abstracting a path in the original space always makes it shorter (or keeps it the same length) but not all paths are shortened the same amount. Consequently the abstraction of the shortest solution in the original space may not be the shortest abstract solution.

A more important thing to notice about this solution is that the states that are passed through in the course of executing f_2^* are not all equivalent to the goal state. For example, after applying the first operator of the sequence f_2^* we are at state $\{small\_on(peg_2), large\_on(peg_1)\}$. This is not equivalent, under this abstraction mapping, to the goal. This is not permitted in the particular definition of refinement we shall be using in this paper, called *monotonic refinement* in [Bacchus and Yang, 1994]. In monotonic refinement the operators added in refining the abstract solution must not change the abstract solution. According to this strict definition, the operator sequence f^* is a refinement of $\widehat{f}^*$ if and only if the abstraction of f^* is exactly the same sequence of operators as $\widehat{f}^*$. In other words, the only operators that can be added during refinement are ones that are identities at the abstract level. In our example this means that in f_1^*, f_2^*, etc., only $move\_large$ can be used. Restricted in this way, *there is no refinement of the abstract solution*: it is said to be *unrefinable*.

There are several possible approaches to the problem of unrefinable abstract solutions. Upon encountering an unrefinable step in an abstract solution one could temporarily relax the definition of refinement (as in "strategy first search" [Georgeff, 1981]). Or one could abandon the abstract solution and return to the abstract space to search for another abstract solution. Another alternative, which is shown in [Bacchus and Yang, 1994] to be particularly effective, is to construct abstractions in such a way that refinability is guaranteed (such abstractions are said to have the *downward refinement property*). This is the approach we have taken. The abstract space in Example 1 has the downward refinement property.

The result of abstraction is another problem space, to which the abstraction process may be applied recursively to create a tower of successively more abstract problem spaces. This tower is called an abstraction hierarchy. Although many systems truncate this tower, for the purposes of analysis and exposition, its top is assumed to be the trivial space consisting of one state and one operator (the identity).

3. A Graph Oriented View of Problem Solving

In this paper, a *graph* is a triple $\langle N, L, E \rangle$, where N is a set of *nodes*, L is a set of *edge labels*, and E is a set of *edges*. Nodes are unlabelled – in fact, we show later in this paper that it is useful to ignore both edge and node labels. An edge is a triple $\langle n_1, n_2, l \rangle$, where l is a label, and n_1 and n_2 are nodes. The direction of the edge is given by the order of n_1 and n_2: edge $\langle n_1, n_2, l \rangle$ leads from n_1 (the *source*) to n_2 (the *destination*). If there is an edge $\langle n_1, n_2, l \rangle$ then n_2 is called a *successor* of n_1, and n_1 a *predecessor* of n_2. A node is never counted among its own successors or predecessors even if there is an edge from the node to itself. An *invertible* graph is a graph in which for every edge $\langle n_1, n_2, l \rangle$ there exists an edge $\langle n_2, n_1, l' \rangle$. A graph is *non-deterministic* if there exist two edges with the same label and source node and different destination nodes, i.e., $\langle n, n_1, l \rangle$ and $\langle n, n_2, l \rangle$; otherwise it is *deterministic*.

An *edge-path* in a graph is a sequence of edges, $e_1 e_2 \cdots e_k$ such that, for all i such that $1 \le i \le k-1$, the destination node of e_i is the source node of e_{i+1}. The source node of the edge-path $e_1 e_2 \cdots e_k$ is the source node of e_1 and the destination node is the destination node of e_k. A *node-path* in graph G is a sequence of nodes $n_1 n_2 \cdots n_k$ such that there exists in G an edge $\langle n_i, n_{i+1}, l_i \rangle$ for all i, $1 \le i \le k-1$. Node-path $n_1 n_2 \cdots n_k$ has length k, source node n_1, and destination node n_k. For every edge-path there exists a unique node-path, and for every node-path there exists one or more edge-paths. *The term path, in this paper, means node-path, not edge-path.*

A problem space can be equated with a graph as follows. Each state in the problem space corresponds to a node in the graph. The labels in the graph are the names of the operators in the problem space. Each operator corresponds to a set of edges, one edge for every state satisfying the operator's precondition: there is an edge $\langle s_1, s_2, l \rangle$ if and only if the operator with name l maps s_1 to s_2. Because operators are functions, state space graphs are always deterministic. Because a solution to a problem $\langle start, goal \rangle$ is a sequence of operators it corresponds to an edge-path with source node $start$ and destination node $goal$. The process of solving problem $\langle start, goal \rangle$, then, is the process of finding an edge-path from $start$ to $goal$ in the state space graph.

3.1 A Graph Oriented View of Abstraction

An abstraction is a mapping from one problem space to another. Having established a correspondence between problem spaces and graphs, it follows that an abstraction, from a graph oriented perspective, is some sort of mapping from graphs to graphs. In particular, it is a *graph homomorphism*, defined as follows.

A graph *homomorphism,* ϕ, from one graph, G_1, to another, G_2, is a pair of functions (ϕ_n, ϕ_e), where ϕ_n maps nodes in G_1 to nodes in G_2, and ϕ_e maps edges in G_1 to edges in G_2, such that for every edge $e = \langle n_1, n_2, l \rangle$ in G_1, $\phi_e(e) = \langle \phi_n(n_1), \phi_n(n_2), l' \rangle$. That is, the image of e has as source and destination the images of the source and destination of e There is nothing assumed about the relationship between l and l'.

The definition of homomorphism imposes a strong constraint on the mappings of nodes and edges. For example, if G_1 is connected, then ϕ_e completely determines ϕ_n. Moreover, a homomorphism is specified up to label choices just by defining the node mapping, ϕ_n: the mapping of an edge $\langle n_1, n_2, l \rangle$ is forced to be an edge in G_2 from $\phi(n_1)$ to $\phi(n_2)$. Indeed, this is how most of our abstraction creating algorithms proceed. They partition the nodes into *classes* and map all the nodes in the same class to the same abstract node. The partition therefore determines the node mapping which, in turn, determines the edge mapping.

The constraint on mappings implies that if (ϕ_n, ϕ_e) is a homomorphism from G_1 to G_2, then ϕ_e can be extended in exactly one way to a mapping on paths. The extension works by simple juxtaposition; that is $\phi_e(e_1 e_2 \cdots e_n) = \phi_e(e_1)\phi_e(e_2)\cdots\phi_e(e_n)$. The definition of homomorphism ensures that this construction works. A consequence of this construction is that the pattern of connectivity in G_1 is, in some sense, preserved, or mirrored, in G_2. The opposite is not true: there can be paths in G_2 that are not images of G_1. For example, suppose there is an edge $e = \langle n_1, n_2, l \rangle$ in G_1 for which there is no *inverse*, i.e. no edge from n_2 to n_1. If n_1 and n_2 are mapped to the same node, n, in G_2, then e must be mapped to $\langle n, n, l \rangle$, which is a path from $\phi_n(n_2)$ to $\phi_n(n_1)$.

The abstractions we create contain only paths which are images of paths in the original graph. More precisely, the abstractions we construct are all *quotients* of the original graph. The quotient of a graph G_1 is defined as follows. Every homomorphism, ϕ, from G_1 to G_2 determines equivalence relations on both the edge set and node set of G_1 as follows: let x and x' be two nodes (or two edges), then we say that x and x' are *ϕ-equivalent* if $\phi(x)=\phi(x')$. For any node (edge) y in G_2, the *ϕ-equivalence* class of y, $\phi^{-1}(y)$, that is the set $\{x_1,x_2,\cdots\}$ of all nodes (edges) in G_1 such that $\phi(x_i)=y$, is called the *pre-image* of y. The pair of equivalence relations determines a new graph whose nodes are the pre-images of nodes of G_2, and whose edges are the pre-images of edges of G_2. This new graph is said to be a *quotient* of G_1. Our abstraction algorithms work by constructing such quotients. The definition of homomorphism is exactly what is needed to make the notion of quotient well defined.

The notion of homomorphism exactly captures the intuitive essence of "abstraction" as a mapping that ignores some features in the original space (e.g. the fact that n_1 and n_2 are distinct can be ignored by mapping them to the same abstract node) while preserving other features (e.g. connectivity). In the remainder of the paper, when we speak of a graph G_2 which is a homomorphic image of another, G_1, we are assuming that G_2 is a quotient of G_1. There are a very large number of homomorphisms of any given graph: every different way of partitioning the nodes is a different homomorphism. Any of these can be used as an abstraction. Not all will speed up search; indeed, identifying which homomorphisms are "good" abstractions (in the sense of speeding up search) is a central research issue, and is the subject of sections 5 and 6.

3.2 A Graph Oriented View of Refinement

Consider the refinement of the abstract solution $\hat{f}_1\hat{f}_2\cdots\hat{f}_n$ for the problem $\langle start, goal\rangle$. Following the standard definition, a solution is taken to be an operator sequence: in graph oriented terms, a solution is an edge-path. Each $\hat{f}_i$ is therefore an edge in the abstract graph; $\phi_e^{-1}(\hat{f}_i)$ is the set of edges that are mapped to $\hat{f}_i$ by the abstraction mapping ϕ_e. Refinement proceeds from the start state, s_1, and searches for a sequence of edges, f_1^*, leading from s_1 to any state, s_1', that is the source node of an edge, f_1, in $\phi_e^{-1}(\hat{f}_1)$. If s_1 is itself the source of an edge in $\phi_e^{-1}(\hat{f}_1)$ then f_1^* is the empty sequence and $s_1'=s_1$. The node s_2 is the destination node of f_1. This process repeats until a sequence $f_1^*f_1f_2^*f_2\cdots f_n^*f_n$ has been constructed. This sequence of edges leads from s_1 to some state

s_{n+1} that is guaranteed to be equivalent to the goal state under the abstraction mapping. If s_{n+1} is not equal to the goal state, the final refinement step is to find a sequence of edges f_{n+1}^* leading from s_{n+1} to the goal state. The final solution, an edge-path from start to goal, is $f_1^* f_1 f_2^* f_2 \cdots f_n^* f_n f_{n+1}^*$.

Recall that in monotonic refinement the operators added in refining the abstract solution must not change the abstract solution. To force refinement to be monotonic, one simply restricts the search that constructs f_i^* to expand only those nodes that are equivalent to s_i, i.e. to those states that are in $\phi_n^{-1}(\hat{s}_i)$, the pre-image of $\hat{s}_i$.

4. New Refinement Methods

4.1 Minor Variations

Figure 5 shows a typical intermediate situation during refinement in which the search constructing f_i^* has reached a state, s, in class $\hat{s}_i$. In the figure (and in all subsequent figures), the abstract classes are shaded. The bold arrows indicate the abstract solution.

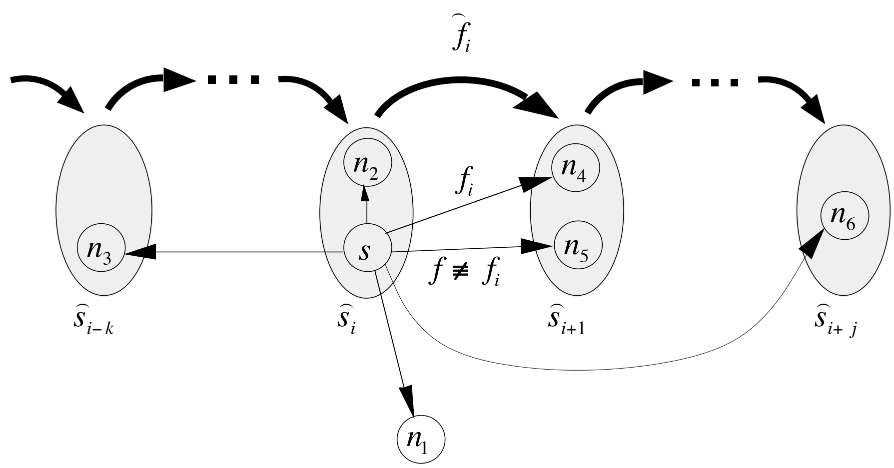

Fig. 5. Different possibilities for the successors of a state.

The six successors of s illustrate the different possibilities for the successors of a state.

n_1: is not in a class in the abstract solution.

n_2: is in the same class as s.

n_3: is in a class preceding $\hat{s}_i$ in the abstract solution.

n_4: is in the abstract class that immediately succeeds $\hat{s}_i$ in the abstract solution and is the destination of an edge labelled f_i.

n_5: is in the abstract class that immediately succeeds $\hat{s}_i$ in the abstract solution and is the destination of an edge whose label is not f_i.

n_6: is in an abstract class that is after $\hat{s}_i$ in the abstract solution but is not its immediate successor.

Ignoring nodes like n_1 is the essence of refinement. This precludes finding short paths from start to goal that involve even a single node that is not in a class in the abstract solution. Consequently, only rarely will refinement find the shortest solution to a problem. However, because search is focused on a small subset of the state space, refinement will be much more efficient than unconstrained search unless the solution it finds is much longer than the optimal solution.

A list of *open* nodes is maintained in the search process. These are nodes which have been reached, but whose successors have not yet been computed. Search proceeds by removing one node from this list and computing its successors. The order in which nodes are removed from this list defines the search strategy. Our current implementation uses breadth first search, which orders the open list in a first-in-first-out manner.

A node like n_2 is added to the open list to record the fact that it has been reached but its successors have not yet been computed.

The refinement algorithm defined in the preceding section processes the remaining nodes as follows. Nodes like n_3 are ignored: search never returns to an earlier class. This (together with the fact that n_1 is ignored) is what makes the search monotonic. In non-monotonic refinement, nodes like n_3 are processed like nodes of type n_2. A node like n_4 signals the completion of the construction of f_i^*: it is called the *terminator* of the search in $\hat{s}_i$. It also serves as s_{i+1}, the starting node of the search to construct f_{i+1}^*. This search begins afresh, its open list is initialized with

n_4. Nodes of type n_5 are ignored. Although n_5 is in the correct abstract class ($\widehat{s}_{i+1}$), the edge leading to it from s is not in the pre-image of $\widehat{f}_i$ and therefore is not acceptable in a refinement of the abstract edge-path. A node of type n_6 is ignored: the refinement algorithm proceeds from one abstract class to the next, never jumping ahead in the sequence.

Other ways of processing nodes n_3-n_6 are possible and give rise to variations on the refinement algorithm described above.

In *node-path* refinement, a solution is defined to be a node-path, not an edge-path. This means that the edge labels are no longer significant and hence nodes like n_5 are not distinguished from those like n_4. Consider figure 6 below, illustrating the edge-path refinement of the abstract path $\widehat{f}_1\widehat{f}_2\widehat{f}_3$. Nodes s_1, s_2, and s_3 are terminators of the searches in the abstract classes $\phi_n(s)$, $\phi_n(s_1)$, and $\phi_n(s_2)$ respectively. The dashed lines in the figure indicate edges at the frontier of the search: The search does not actually traverse these edges, but it does look at them to determine if their labels are in the pre-images of the corresponding abstract edges.

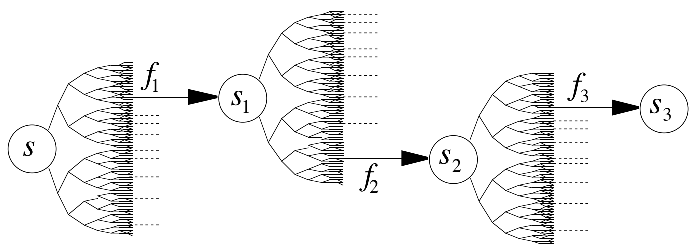

Fig. 6. Edge path refinement.

In the worst case, node-path refinement will behave identically to edge-path refinement. This occurs if refinement always happens to find nodes of type n_4 before finding nodes of type n_5. This is illustrated in figure 7.

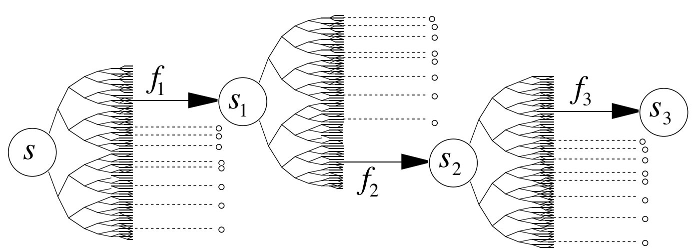

Fig. 7. Node-path refinement is never worse than edge-path refinement.

Edge-path refinement and worst case node-path refinement require essentially the same search: the only difference is that, in node-path refinement, the edges at the frontier of the search are actually traversed whereas, in edge-path refinement they are only examined to determine their labels.

In some cases, node-path refinement can do substantially less search than edge-path refinement and, at the same time, produce shorter solutions. This will happen if, in node-path refinement, nodes of type n_5 are encountered before nodes of type n_4. A situation of this sort is depicted in figure 8. Edge-path refinement would search the entire tree shown to find edge f_1. It would ignore the edge labelled f since this is not in the pre-image of $\widehat{f}_1$. Node-path refinement, on the other hand, will find nodes like s_1' irrespective of the labelling of the edges leading to them.

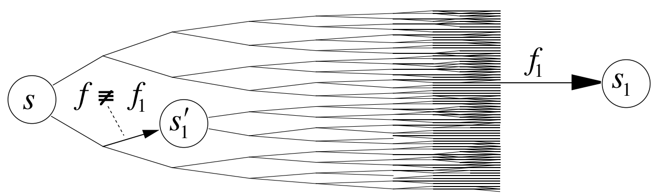

Fig. 8. Node-path refinement can be better than edge-path refinement.

In opportunistic refinement nodes like n_6 are treated like those of type n_4, thereby permitting parts of the abstract solution to be skipped. "Opportunities" to skip ahead certainly do arise, although they are much more common with some variations of refinement than with others. As figure 5 suggests, opportunistic refinement can do less search and produce shorter solutions than non-opportunistic refinement.

In the original refinement algorithm – monotonic, non-opportunistic, edge-path refinement – a node s could have at most one successor that was a terminator. However, in all the variations of refinement we have defined, s might have several successors that are terminators. Various policies for handling this situation are possible. With the *first successor* policy, construction of f_i^* terminates as soon as the first terminator is generated, and that node is used as the starting node of the search to construct f_{i+1}^*. An alternative is the *all successors* policy, in which the search to construct f_{i+1}^* uses as starting nodes all the successors of s that are terminators (in opportunistic refinement, only the terminators in the farthest abstract class are used).

Monotonic node-path refinement with the first successor policy will be referred to as *classical refinement*. It will be used in all our experiments to represent the standard refinement techniques found in the literature. It should be kept in mind that it is an improvement over all previously reported refinement algorithms in that it does node-path refinement, not edge-path refinement[5].

The performance of three of these variations is summarized in Table 1. Compared to classical refinement (CR) the all successors policy (CRall) reduces solution length by about 5% and increases work (edges traversed) by about 5%. Opportunism (CRopp) gives little advantage, although the fact that it differs from CR shows that some opportunities are arising that CR is missing. In the search spaces used in this experiment in can be shown that no opportunities can possibly arise when the all successors policy is used, so the table does not include an entry for this policy in combination with opportunism.

Table 1
Variations of Classical Refinement[6].

Search	Work			Solution Length		
Space	CR	CRall	CRopp	CR	CRall	CRopp
5-puzzle	139	151	139	30.2	29.5	30.0
Bitnet	305	305	305	7.5	7.1	7.5
Blocks-6	302	318	302	14.9	14.3	14.9
KL-2000	1642	1655	1644	8.9	8.3	8.8
Permute-7	242	267	242	11.5	11.3	11.5
TOH-7	502	525	484	93.5	86.2	90.0
Words	530	527	519	13.7	12.6	13.5

4.2 Optimal Refinement

Monotonic refinement is not guaranteed to find the shortest refinement of a given abstract solution. For example, in figure 9 the shortest refinement is the path $start\, x_1 x_2 x_3 x_4\, goal$. To find this refinement it is necessary, in order to reach x_1, to continue searching in $\hat{s}_1$ after having reached a node, y, in $\hat{s}_2$. It is also necessary, in order to

[5]ALPINE is described as doing node-path refinement in [Knoblock, 1991], but is also included as an example of the general algorithm in [Knoblock et al., 1991b] which uses edge-path refinement.

[6]Results are averages over 100 problems. Work is measured in "edges traversed". Abstractions were created by the STAR algorithm using Max-Degree with radius 2 and No-Singletons (see section 5.3).

construct the segment $x_2 x_3$, to return to $\hat{s}_1$ from $\hat{s}_2$. Monotonic refinement does neither: it discontinues search in $\hat{s}_1$ as soon as it reaches a node in $\hat{s}_2$, and it will not return to $\hat{s}_1$. The path it would find is $start\ y \cdots x_4\ goal$, which could be arbitrarily longer than the optimal path.

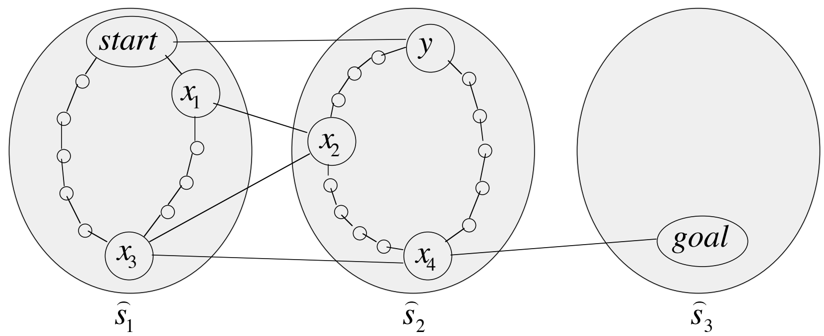

Fig. 9. Monotonic refinement can produce arbitrarily long solution paths.

An experiment was run to determine how poorly monotonic refinement performs in practice. In Table 2, the length of the solutions produced by classical refinement (CR) are reported beside the optimal solution length, computed by solving each problem using breadth first search in the original space. This is the first time that the solution lengths produced by classical refinement techniques have been compared to optimal solutions: previously in the literature the solutions produced by refinement have been compared with solutions produced by heuristic search methods (often the solutions produced by refinement are shorter than those produced by heuristic search). As can be seen, the solutions produced by classical refinement are often very poor. On average, they are 40% longer than optimal.

Table 2
New refinement methods[7].

Search Space	Work			Solution Length			
	CR	OptR	AltO	Optimal	CR	OptR	AltO
5-puzzle	139	186	136	21.2	30.2	26.1	25.5
Bitnet	305	687	305	6.5	7.5	6.5	7.1
Blocks-6	302	593	293	10.2	14.9	13.3	11.8
KL-2000	1642	1999	1447	6.6	8.9	8.0	8.1
Permute-7	242	553	265	6.1	11.5	10.4	7.8
TOH-7	502	585	504	64.4	93.5	76.2	80.7
Words	530	848	524	9.1	13.7	11.9	11.2

The optimal refinement of a given abstract solution can be found by using a standard shortest path algorithm (we use breadth first search) but having it ignore nodes of type n_1. In our experiment, optimal refinement (OptR) produced solutions that are about 13% shorter than CR's. The price paid for these shorter paths is increased work: optimal refinement does 60% more work than classical refinement. Because no technique for refining an abstract path can produce shorter solutions than optimal refinement, the 27% difference between the length of the optimal refinement and the optimal solution is a penalty incurred by all path refinement algorithms. In order to produce shorter solutions, we have developed an extension of path refinement which uses the abstract search tree, not just the solution path.

4.3 Alternating Search Direction

If search at the abstract level begins at the abstract start state, it creates a search tree rooted at the abstract start state; the abstract solution is the unique path in this tree that ends at the abstract goal state. For the classes in the abstract solution, the distance to the goal is known; this information is not known for any other class in the abstract search tree. It is precisely this information that is needed by any refinement algorithm. In order to decide how to process a node its type must be known, and to determine its type one must know the distance from its abstract class to the abstract goal. Because this information is known only for the classes in the abstract solution, refinement must confine its search to these

[7]Results are averages over 100 problems. Work is measured in "edges traversed". Abstractions were created by the STAR algorithm using max-degree with radius 2 and no-singletons (see section 5.3).

classes. Nodes in the other classes in the abstract search tree are considered to be of type n_1 and are ignored.

However, if search at the abstract level is conducted in the opposite direction, the search tree it creates will be rooted at the abstract goal state and the information needed by the refinement algorithms will be available for all the classes in abstract search tree. In figure 10 the solid edges show the abstract search tree rooted at the goal node, the bold edges indicate the solution, and the numbers in the classes indicate the distance to the goal.

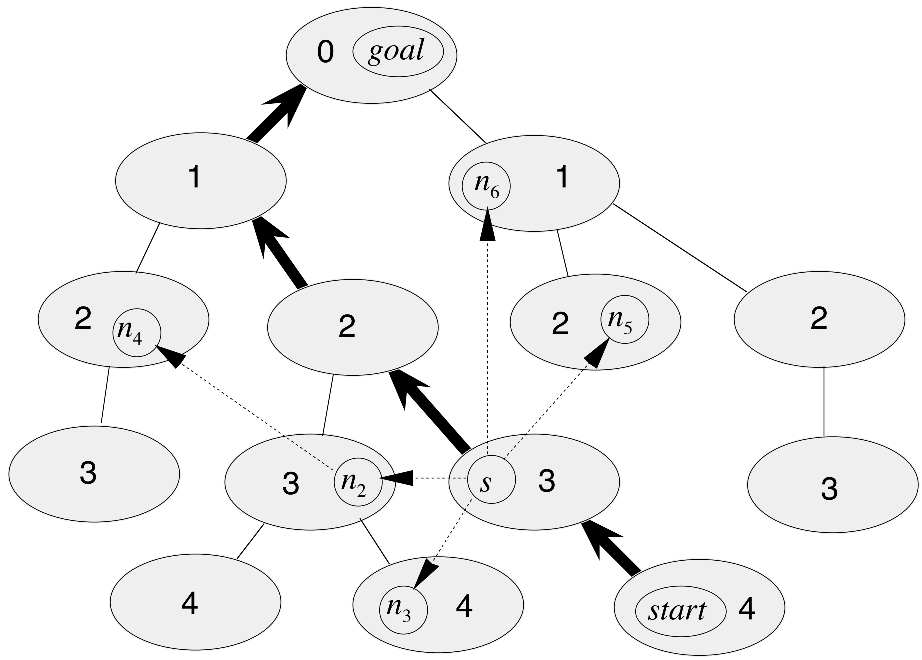

Fig. 10. The 6 nodes types in alternating search direction.

Refinement proceeds as usual, forward from the start state. The definitions of the six types of successor nodes are generalized to include nodes in every class in the abstract search tree. A node is type n_1 only if its abstract class is not in the search tree. A node is type n_2 if the distance from its class to the goal is the same as the current node (s in figure 10). A node is type n_3 if the distance from its class to the goal is greater than that of the current node. Types n_4, n_5, and n_6 are defined similarly. Using these definitions there are at least as many nodes of each of types n_2-n_6 as with the original definitions. For each type, figure 10 shows a typical node that is covered by the new definitions but not by the original definition.

The number of additional nodes of each type covered by the generalized definitions will, of course, depend on the particular graph and abstraction. It can happen that there are no additional nodes of any type: in this case alternating search direction reduces to ordinary refinement. Additional nodes of type n_3 have no effect on monotonic refinement, since they are treated the same as n_1 nodes. Additional nodes of type n_2 broaden search. This increases the search effort, but these additional nodes also influence subsequent search and the net result might be that shorter solutions are found and/or that less search is done in total. Additional nodes of types n_4, n_5, and n_6 will usually reduce the solution length and the amount of search required for monotonic refinement.

The net effect of alternating search direction on solution length and search effort is therefore not certain *a priori*. It can potentially produce shorter paths than optimal refinement and do less work than classical refinement. But improvement of either kind is not guaranteed. Because it is monotonic, alternating search direction, like classical refinement, is not guaranteed to find the optimal refinement. Consequently, the solutions it finds might be longer than those found by optimal refinement. Its broader search, although capable of producing shorter solutions more quickly than classical refinement is equally capable of leading search astray, increasing both the amount of search and the solution length.

Table 2 includes the results of opportunistic alternating search direction (AltO). OptR's solutions are shorter than AltO's in three of the spaces, and AltO's are shorter in the other four. On average, OptR's solutions are about 5% longer than AltO's. AltO is clearly the best of these refinement algorithms. It produces the shortest solutions (only 20% longer than the optimal solutions, on average) and does the least work (the same as is done by classical refinement). An additional advantage of AltO, which is evident in the experimental results in section 6, is that it is less sensitive than the other refinement algorithms to the abstraction hierarchy to which it is applied. This is because the performance of the other algorithms depends on exactly which abstract solution is found: This can change radically with even small changes in the abstraction hierarchy or search method, whereas AltO's performance depends on the abstract search tree, which is relatively insensitive to small changes in the abstraction hierarchy and search technique.

The alternating search direction technique has been described so far in terms of just two levels in the abstraction hierarchy: the original level and the abstract level immediately above it. In an abstraction hierarchy with several levels, search direction should alternate from one level to the next. Note that alternating search direction between levels is entirely different from bi-directional search (e.g. [Pohl, 1971; Dillenburg and

Nelson, 1994]). In bi-directional search the search direction within a single level changes from time to time, but in alternating search direction between levels, the search direction in any particular level is fixed.

4.4 Summary

Figure 5 succinctly summarizes the set of decisions one faces in designing a refinement algorithm: *how shall each type of node be processed?* These choices have not been previously discussed and at least one of the new variations, node-path refinement, is guaranteed to outperform the variation (edge-path refinement) used in existing systems.

Optimal refinement is useful for two purposes. The first is scientific: it provides a lower bound on the solution lengths that can be produced by any path refinement technique. When compared to an existing technique, such as classical refinement, this indicates how close the technique is to producing the best possible solutions. When compared to the optimal solution length, it summarizes the potential of the entire family of path-refinement techniques.

Optimal refinement is also a practical refinement technique, offering a balance between speed and solution length that is different from classical refinement. Classical refinement is faster, but produces longer solutions than optimal refinement. Intermediate positions are certainly possible: the all successors policy being just one example.

By alternating search direction between levels of abstraction, search is broadened to encompass all classes in the abstract search tree, not just those on the solution path. While this broadening introduces the risk of increasing search effort it also introduces opportunities for finding shorter solutions than optimal refinement and the concomitant reduction in search effort. Experimentally, the benefits have been found to outweigh the costs. Alternating search direction, in conjunction with opportunism, produces solutions slightly shorter than optimal refinement while doing about the same amount of search as classical refinement.

The graph oriented approach has been invaluable in developing these techniques. However, all of the new refinement techniques work equally well on explicitly or implicitly represented graphs. For example, node-path refinement simply requires that the solutions be represented as a list of nodes instead of as a list of operators and bindings. Alternating search direction requires the search tree to be recorded. If the entire tree is too large to fit in memory, the algorithm will work with whatever fragment of the tree is recorded. Optimal refinement can be based on any shortest path algorithm, including any of the memory efficient versions of A^* (e.g. IDA^* [Korf, 1985b]).

5. New Abstraction Techniques

5.1 Algebraic Abstraction

When describing a problem space, one normally chooses operator names that are meaningful and which specify enough detail to completely define the effect of each operation on any particular state. For example, in defining the Towers of Hanoi puzzle one might name one of the operations as "move the top disk on peg1 onto peg2". Although this name does not explicitly state which disk to move, it does uniquely determine the disk to move in any given state, and, in fact, it completely specifies the effect of this operation.

As defined in section 3.1, an abstraction (graph homomorphism) is not required to preserve either the determinism or the operator names (edge labels) of the initial graph. A graph homomorphism is free to label the abstract edges in any manner whatsoever. However, it does seem desirable to preserve determinism and edge labels, and there is no immediately obvious reason why these useful properties should not be preserved. Formally, a graph homomorphism, ϕ, of a deterministic graph G preserves edge labels and determinism if:

- ϕ_e maps edge $\langle s_1, s_2, l \rangle$ to $\langle \phi_n(s_1), \phi_n(s_2), l \rangle$, and
- $\phi(G)$ is deterministic.

Such a homomorphism is called an *algebraic abstraction*. Innocuous as the above two properties may seem, they interact with each other, and with the definition of homomorphism, so as to have far reaching implications. To see this, consider the situation depicted in figure 11.

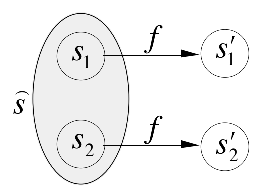

Fig. 11. Consequences of asserting that two nodes are in the same abstract class.

The states s_1 and s_2 are two states in the original, deterministic graph, and operator f is applicable to both, mapping s_1 to s_1' and s_2 to s_2'.

The shaded ellipse enclosing s_1 and s_2 indicates that, under ϕ_n, these two states are in the pre-image of the same abstract state $\hat{s}$. If ϕ_e

preserves edge labels then the two edges labelled f in the original space will also be labelled f in the abstract space. If, in addition, ϕ preserves determinism, there can only be one edge labelled f emanating from $\hat{s}$, and therefore s_1 and s_2 must be mapped by ϕ_n to the same abstract class. Formally, if ϕ is an algebraic abstraction, then from $s_1 \equiv s_2$ it necessarily follows that $f(s_1) \equiv f(s_2)$ for every operator f that is applicable to both s_1 and s_2. Because of this, a single assertion, $s_1 \equiv s_2$, may have immediate consequences which, in turn may have further consequences, and so on.

The set of consequences of a single assertion, $s_1 \equiv s_2$, can be small (even empty) or extremely large. This depends largely on the set of operators used to represent the space. For example, imagine a space having a prime number, p, of states arranged in a circle, with each state connected to its immediate neighbour on either side. The space could be represented with the p operators "Go To D" (where D identifies the destination state), or it could be represented with just two operators, "clockwise" and "anticlockwise". With the first representation, the assertion $s_1 \equiv s_2$ has no nontrivial consequences, but with the second representation, any assertion $s_1 \equiv s_2$ has as consequences all possible assertions: $\{s \equiv t, \text{ for all states } s \text{ and } t\}$[8].

The situation in which all possible assertions are forced is called *total collapse* because it means that in the abstract space there is only one state. In practical terms, total or "near" total collapse produces an abstract space that is useless, i.e. unable to speed up search.

Our initial abstraction algorithms created algebraic abstractions, using a variety of heuristics to choose the assertions $s_1 \equiv s_2$ on which to base the abstraction. In working with these algorithms, we observed that they were all extremely "representation dependent", in the sense that the quality of the abstractions they created depended critically upon the exact details of the representation they were given for a state space.

The practical significance of being highly representation dependent hinges on the prevalence of "natural" representations that cause near total collapse. If only highly contrived representations have this effect, representation dependency is an irrelevant concern. On the other hand, if a large fraction of "natural" representations cause near total collapse, then we will be forced to abandon algebraic abstraction in favour of some other, less representation dependent method of abstraction. To

[8] In a cyclic space the number of consequences of $s_1 \equiv s_2$ depends on the greatest common divisor (gcd) of the number of states (P) and the distance between s_1 and s_2 ; all possible assertions are consequences when the gcd is 1. The gcd is 1 in this example because P is prime.

determine if "natural" representations could cause near total collapse we ran the following experiment.

Experiment Design

A system was implemented that computes the minimum set of consequences of a single assertion $s_1 \equiv s_2$. Being the minimum set, the results produced by this system give a lower bound on the representation dependency of any algebraic abstraction technique. Determining the minimum set of consequences of a set of assertions is a straightforward transitive closure operation. One forms an initial set of consequences: the assertions themselves. Then an assertion is removed from this set and its immediate consequences are determined and added to the set. This is repeated until the set of consequences is empty.

Four different "natural" representations were devised for two different puzzles, the 5-disk Towers of Hanoi puzzle and the 5-puzzle. The system was applied to each representation for many different choices of the assertion $s_1 \equiv s_2$ (every state in the space was guaranteed to appear in at least one of the assertions).

Each trial consisted of one run of the system with one representation and one assertion of the form $s_1 \equiv s_2$. On each trial we measured how many states were involved in the consequences of $s_1 \equiv s_2$ (such a state is said to be *affected* by $s_1 \equiv s_2$), and how many abstract states were created. Total collapse corresponds to all the states being affected and one abstract state being created; "near" total collapse corresponds to "most" states being involved and "few" abstract classes being created.

The different representations for each of the puzzles are described below. We emphasize that the graphs defined by the different representations for a puzzle are identical except for labels on the edges, which are based on the operators' names and the values to which its parameters are instantiated.

Representations of the 5-disk Towers of Hanoi

TOH-1: $move(x,d)$ moves disk x in direction d. For example, $move(1,clockwise)$ moves the smallest disk clockwise.

TOH-2: $move(p,d)$ moves the top disk on peg p in direction d. For example, $move(1,clockwise)$ moves the top disk on peg 1 onto peg 2.

TOH-3: $move(x,y)$ moves a disk between the two pegs x and y, in whichever direction is permissible.

TOH-4: There are three operators: (a) move the smallest disk clockwise; (b) move the smallest disk anticlockwise; and (c) move a disk other than the smallest (there is at most one such move permissible in any state).

Representations of the 5-puzzle

5PUZ-1: There is a single operator, $move(d)$. This moves the blank in direction d. The possible directions are $left$, $right$, up, and $down$.

5PUZ-2: This is the same as 5PUZ-1 except that the possible directions are $clockwise$, $anticlockwise$, and $vertical$ ($vertical$ moves the blank down if it is in the upper row and moves it up if it is in the lower row).

5PUZ-3: There are three operators: $small$, $medium$, $large$. The operator $small$ exchanges the blank with its smallest neighbour, i.e. the tile with the smallest value of all the tiles adjacent to the blank. The operator $large$ exchanges the blank with its largest neighbour. If there are three tiles adjacent to the blank, $medium$ exchanges the blank with the one that is neither smallest nor largest.

5PUZ-4: $go\_to(row,col)$ moves the blank to the position specified. e.g. $go\_to(1,1)$, if permitted, would move the blank into the top left position.

Observations

As can be seen from table 3, TOH-1 and TOH-2 produce very similar results. On only about 10% of the trials are more than 10% of the states affected, and in virtually all trials there are just two states per abstract state. Thus, these representations are ideal for algebraic abstraction.

The results for TOH-3 and TOH-4 are similar to one another, and quite the opposite of the results for TOH-1 and TOH-2. In all but two trials, all states were affected, and, although total collapse never occurred, in many trials very few abstract states were created. Consequently, algebraic abstraction would fare poorly if given one of these representations for the Towers of Hanoi puzzle.

On one sixth of the trials with 5PUZ-1 every state was affected, but many abstract states were created with very few states in each. On the other five sixths of the trials very few states were affected and there were always two per abstract state. Algebraic abstraction would work well with this representation.

By contrast, algebraic abstraction would frequently fail if given the 5PUZ-2 representation of the 5-puzzle. With 5PUZ-2 all states are affected on every trial. Total collapse occurs on one third of the trials, and on another third of the trials all the states are mapped to two abstract classes.

Table 3
Effect of different representations[9].

Representation	Number of abstract states created	Number of states affected	Number of states per abstract state
TOH-1	8.04	17.08	2.04
TOH-2	5.78	11.58	2.00
TOH-3	7.43	242.33	80.0
TOH-4	13.56	242.33	39.47
5PUZ-1	22.32	65.8	2.26
5PUZ-2	33.79	360.0	182.18
5PUZ-3	3.0	360.0	267.76
5PUZ-4	41.82	179.5	10.1

The 5 disk Towers of Hanoi graph has 243 states.
The version of the 5-puzzle graph used in this experiment has 360 states.

5PUZ-3 is even worse: algebraic abstraction would always fail on this representation. All states are affected on every trial. There are three trials in which each state is paired with one other state, but on all the others the result is total collapse or near total collapse (2 abstract states).

5PUZ-4 gives mixed results. In half the trials, $s_1 \equiv s_2$ has no consequences whatsoever. In the rest, all states are affected but the number of abstract states created varies from 5, which is almost total collapse, to 180, which means each state is paired up with only one other state.

Discussion

Two important points are established by this small experiment. The first is that "natural" representations can render algebraic abstraction

[9]The rightmost column is not the ratio of the two to its left. This column is computed by taking the ratio of states to abstract states on each trial and averaging these over all the trials. This is the "average ratio", i.e. the average number of states per abstract state on each trial. Taking the ratio of column 2 to column 1 gives a different statistic, the ratio-of-averages, which can be quite different from the average ratio.

entirely useless. The clearest case of this is TOH-4, a representation drawn from the literature [Hayes, 1977]. The second point is that algebraic abstraction is very sensitive to the exact details of the representation, in the sense that it can perform radically differently on two representations that are, conceptually, very similar. This is exemplified by 5PUZ-1 and 5PUZ-2.

We conclude that, however desirable it might be to preserve determinism and edge labels, such a requirement renders an abstraction system too representation dependent to be useful. The abstraction techniques considered in the following sections will, in fact, ignore labels altogether. Consequently, they are perfectly free to consider any graph homomorphism whatsoever: all restrictions are removed. This freedom permits us to explore the question, what sort of homomorphisms are guaranteed to produce speedup in any graph, without any a priori constraints on the nature of the homomorphism.

Algebraic abstraction is not the only form of abstraction that suffers from representation dependency. Although it is a central concern for all abstraction systems, this issue is rarely mentioned in the abstraction literature. [Knoblock, 1994]'s "Limitations and future work" section (pp. 294-296) is the first clear statement of the issue. The main example there involves three different representations of the Towers of Hanoi puzzle. Two of these give rise to good abstraction hierarchies (having as many levels as there are disks), but using the third representation, the abstraction system, ALPINE, is unable to create a non-trivial abstraction. As a second example, ALPINE is also unable to generate a non trivial abstraction for the standard representation of the Blocks World (e.g. the one in [Nilsson, 1980]).

5.2 Derivation of a New Abstraction Algorithm

In order to determine what sort of homomorphism will speed up search, it is useful to undertake an analysis of the total "work" done in constructing a solution by refining, through successive levels of abstraction, an initial solution at the highest level of abstraction (which, being a space with only one node, always has the trivial solution). We shall number the levels of abstraction from 0 to α, with 0 being the highest level of abstraction and level α being the original space.

Given a solution (node-path) of length λ_i at level i, refinement replaces each individual node in this solution by a sequence of nodes in level $i+1$. If the "work" required to do one such replacement is ω, then the total work required to refine the solution from level i to level $i+1$ is $\omega\lambda_i$. We define χ, the *expansion factor*, to be λ_{i+1}/λ_i. Thus $\lambda_{i+1} = \chi\lambda_i = \chi^{i+1}$.

The exact values of ω and χ depend on the problem being solved, the level of abstraction, and the abstract state that is being refined. In the following ω and χ stand for the worst cases of these values over all problems, levels, and states. Assuming that refinement at each level need only be done once (i.e. that there is no backtracking across levels) the total work done to refine an initial trivial solution (whose length is 1) to a solution in the original space can be bounded above:

$$Total\ \ Work \le \omega \sum_{i=0}^{\alpha-1} \chi^i .$$

It is useful to replace the variables χ and α in this formula with variables that can be directly measured or controlled at the time the abstraction hierarchy is being created. There is no exact replacement for χ, but it is bounded above by d, the maximum diameter of any abstract state. The *diameter* of an abstract state is the maximum distance (length of the shortest path) between any two states in that abstract state. This substitution produces:

$$Total\ \ Work \le \omega \sum_{i=0}^{\alpha-1} d^i .$$

The sum can be replaced by the closed form formula $\left(d^{\alpha}-1\right)/\left(d-1\right)$ which, because $d \ge 2$, can be replaced by a simple upper bound d^{α}. Variable α can be replaced by $\log_c n$, which is equal to $(\ln n)/(\ln c)$, where ln is natural logarithm, n is the number of states in the original space and c is the number of states mapped to the same abstract state. We assume $c \ge 2$ and that c is the same for all abstract states and all levels. These substitutions produce:

$$Total\ \ Work \le \omega d^{(\ln n)/(\ln c)}.$$

Since $x^{\ln y}$ is a symmetric function, the value of the right hand side does not change if d and n are exchanged, producing the final form of the total work formula:

$$Total\ \ Work \le \omega n^{(\ln d)/(\ln c)}.$$

Ignoring the ω term for a moment, total work is minimized by minimizing d and maximizing c. Unfortunately these two variables are not independent and reducing one tends to reduce the other, the opposite of the effect needed to minimize total work. Nevertheless, a heuristic for building good abstractions that follows from this analysis is to fix d at a small value and, for this fixed value of d, try to maximize c.

An important feature of this formula is the fact that, if $d < c$, then the term involving n will be sublinear (n raised to a power less than 1). This is important because for a wide range of common circumstances and definitions of "work" the total work required to solve a problem using a blind search technique, such as breadth first search, without abstraction is at least linear in n. Therefore, if an abstraction can be created such that $d < c$ and ω is independent of n, problem solving with the abstraction will be guaranteed, in such circumstances, to be faster than blind search without abstraction.

Guaranteeing that $d \leq c$ is easy: any partitioning of a graph into connected components has this property. This sort of partition has the additional advantage that it is guaranteed to be monotonically refinable. This is because two nodes that are mapped to the same abstract node are in the same class of the partition and, by definition of "connected component", there must exist a path wholly within that class of the partition that connects the two nodes.

The diameter, d, will be equal to c in a connected component if and only if the component is a single state or a linear chain of states. The partitioning algorithm described in the next section has a provision for avoiding single states but not linear chains, so it is not guaranteed to produce abstractions in which $d < c$. But in many of the components it produces $d < c$, and in the others $d = c$; so searching using the abstraction hierarchy is at worst linear in n.

Finally, consider the term, ω, representing the work required to refine a single node in an abstract solution. The analysis so far holds for any definition of "work" and any refinement algorithm. If monotonic refinability is guaranteed, as it is with the sort of partitions we are now considering, there are at least two ways to make ω independent of n. The first is to build, at the time the abstraction hierarchy is being constructed, a routing table storing the shortest paths between every pair of nodes in the same class of the partition. With such a table, refinement is simply table lookup, and ω is the work involved in table lookup which is, at worst, logarithmic in the size of the table. If c is a constant (i.e. independent of n), then the extra space required for these tables (c^2) is also constant per node in the abstraction hierarchy. Consequently the total space required to store the abstraction hierarchy is the same order with or without the lookup tables.

A second way to make ω independent of n, which is the one implemented in our system, also requires c to be a constant (or at least very much smaller than n). The nodes in the same class of the partition, and the edges associated with them, are stored as a graph. Refinement of a single abstract node involves blindly searching in just one of these graphs. If monotonic refinability is guaranteed and one is careful to separate the edges that lead to nodes outside this graph ("exit edges") from edges that lead to nodes within the graph ("internal edges") then ω will be proportional to the number of internal edges[10] and therefore bounded above by c^2.

Intuitively, speedup is obtained by decomposing a large search problem into a set of smaller sub problems that can be solved independently. The analysis has provided specific definitions of "smaller" ($d < c$ and c is independent of n) and "independently" (no backtracking across levels of abstraction and ω is independent of n) that guarantee "speedup" (total work is sublinear in n).

This analysis differs from the one in [Knoblock, 1990] in several respects. Knoblock's k parameter is identical our χ, and in both analyses this parameter plays the key role of determining the length of the solution at each abstract level. However, in Knoblock's analysis this length is expressed as a fraction of the optimal solution length. This forces Knoblock to make the unnecessary, and untenable, assumption that refinement constructs the optimal solution. Our analysis makes no assumption about the length of the solution constructed by refinement. A second difference is that we defer making assumptions about how work is measured until after the formal derivation has been finished, whereas a specific work formula pervades Knoblock's analysis.

Perhaps the most important difference is the purpose served by the two analyses. Knoblock's aim is theoretical: to derive a fairly exact "work" formula in order to prove that, under certain conditions, refinement is exponentially faster than blind search. The primary purpose of our analysis is practical. It is intended to guide the design of our abstraction algorithm by relating controllable properties of an abstraction to the total work involved in using the abstraction. To be useful, there must exist a broad class of graphs satisfying the main assumptions of our analysis. The "no backtracking across levels" assumption is satisfied by creating abstract classes that are connected components. The other key assumption is that c (and therefore ω) is independent of n. To satisfy this, it must be possible to partition the graph into connected components whose size does not depend on n. This property holds for a very large set of commonly occurring graphs, including all sparse undirected graphs.

[10] In fact, the current implementation does not make this distinction and is therefore considerably less efficient than it might be. This improvement will be made in a future implementation.

5.3 The STAR Abstraction Algorithm

The preceding analysis provides three specific recommendations about how to partition the nodes in a graph so that the resulting abstraction hierarchies will speed up search:

- The classes should be connected.
- The classes should have small diameter (d).
- The number of nodes in a class c should be larger than d but much smaller than n.

A simple algorithm based upon these principles is the *STAR* algorithm, whose pseudo code is given in the box below. The algorithm builds classes one at a time by picking a node to act as the "hub" of the class and then gathering together all nodes that can reached from the hub by a short path that does not pass through any other class. The maximum distance from the hub, called the radius of abstraction, is specified by the user. In graphs whose edges all have inverses, such as the ones in this study, this method of construction guarantees the classes will be connected and have small diameters.

A hierarchy of abstractions is built up by running the STAR algorithm on the abstracted space to produce a further abstracted space. This space is in turn abstracted and so-on, until the abstracted space contains only one node.

Given: G, a graph,
$r \geq 2$, an integer (the radius of abstraction),
noSingletonsAllowed, a boolean indicating if it is unacceptable to have classes that contain only one node.

Produce: P, a partition (i.e. a set of classes) of the nodes of G.

The STAR Algorithm

0. Initially, $P = \{ \}$, and *nodes* is the set of all nodes in G.
1. Repeat until *nodes* is empty:
   (a) Select a node, *hub*, in *nodes*,
   (b) $NewClass \leftarrow$ the set of nodes n such that $n \in nodes$ and $n = hub$ or is connected to *hub* by a path of length r or less wholly within *nodes*,
   (c) $nodes \leftarrow nodes - newClass$,
   (d) Add *newClass* to P.
2. If *noSingletonsAllowed* then:
   Repeat until P contains no classes containing just one node:
   (a) Select a class, *singleton*, in P that contains just one node n_1,
   (b) Remove *singleton* from P,
   (c) Choose a neighbour n_2 of n_1, and add n_1 to the class in P containing n_2.

With this algorithm, there is no guarantee concerning the number of nodes in each class. It is not unusual to create singleton classes (containing just one node). At the opposite extreme, on rare occasions a very large class is created, containing almost all the nodes in the graph.

In the present implementation the only direct control over class size is a post processing step (step 2) to eliminate singleton classes. As can be seen in table 4, the difference in performance between allowing and eliminating singletons is not large except in the Bitnet, Blocks-6, and Words graphs, where eliminating singletons greatly reduces the work required for problem solving.

Indirect control over class size is possible by the choice of radius and by altering the criterion used to select the hub node in step 1(a). We have explored two criteria for selecting hubs. The first is to choose the node having the most neighbours in the set *nodes* (i.e. not already assigned to a class). This is a greedy way of optimizing $(\ln d)/(\ln c)$ which, providing the ω term does not grow too large, will minimize the total work for problem solving. Abstraction using this criterion is called *max-degree* abstraction. One potential drawback of max-degree abstraction is that it can create classes of very different sizes, with one class containing a large percentage of the nodes. The second criterion is to select hubs at random. Abstraction with this criterion is faster than max-degree abstraction and less subject to the drawback just mentioned.

Table 4
Effect of allowing singleton classes[11]

Search Space	Refinement Technique	Work		Solution Length	
		No Singletons	Singletons	No Singletons	Singletons
5-puzzle	CRall	179	151	29.5	27.9
	AltO	136	160	25.5	24.4
Bitnet	CRall	305	804	7.1	7.1
	AltO	305	805	7.1	7.1
Blocks-6	CRall	318	445	14.3	14.0
	AltO	293	443	11.8	11.9
KL-2000	CRall	1655	1549	8.3	8.4
	AltO	1447	1305	8.1	7.4
Permute-7	CRall	267	305	11.3	10.8
	AltO	265	282	7.8	8.1
TOH-7	CRall	525	569	86.2	90.1
	AltO	504	517	80.7	79.3
Words	CRall	527	915	12.6	12.3
	AltO	524	916	11.2	10.7

Results are averages over 100 problems
Work is measured in "edges traversed"
CRall is classical refinement with the "all successors" policy
AltO is alternating search direction + opportunism.

The STAR algorithm exploits the fact that the graph is explicitly represented to achieve two important advantages over existing abstraction algorithms, which are all based on based on implicit graph

[11]Abstractions were created by the STAR algorithm using max-degree with radius 2 and no-singletons.

representations. First, our algorithm constructs abstract classes that are strongly connected. This has several important consequences:

- All abstract solutions are monotonically refinable.

- Each step in an abstract solution is guaranteed to be refinable without backtracking across levels of abstraction and without backtracking to redo the refinements of earlier steps in the solution.

- In sparse undirected graphs the total work involved in refinement is virtually guaranteed to be sublinear ($d < c$ except in pathological cases) in the number of nodes in the original graph and therefore less than the work done by blind search.

Previous abstraction algorithms are not guaranteed to produce abstractions with these properties. For example, [Knoblock, 1994] illustrates the need for ALPINE to backtrack across levels of abstraction with the "extended STRIPS domain". The graph underlying this domain is, in fact, sparse and undirected so repeated applications of the STAR algorithm would, without backtracking, create an abstraction hierarchy in which all abstract solutions are monotonically refinable.

The second advantage of using an explicit representation is that it gives the user great flexibility in the construction of abstractions. The STAR algorithm gives the user direct control over the granularity (radius) of the abstraction and the selection of hubs. Our implementation also allows the user to specify the criteria used to decide which nodes to include in each abstract class. The STAR algorithm corresponds to the criterion we have found most successful to date, but several others have been examined (e.g. pair each node with one of its neighbours, create classes that are tree shaped or linear rather than star shaped) and many others are possible.

Existing abstraction techniques give the user no direct control over the granularity or any other aspect of the abstractions created. For example, [Knoblock, 1994] presents ALPINE in a series of successively more sophisticated versions. Each new version is motivated by observing that the previous version creates abstraction hierarchies that are too coarse grained (p. 258, p. 267, p. 270). Although the final version produces good empirical results, improving and controlling the granularity of its abstraction hierarchies is presented as the main direction for future research (pp. 294-296).

6. Experimental Evaluation

Section 4 introduced three main refinement techniques. Classical refinement (CR) is representative of refinement techniques reported previously in the literature. Optimal refinement (OptR) provides a lower bound on the length of solutions that will be produced by refinement techniques that restrict search to a single abstract solution. Alternating opportunistic refinement (AltO) relaxes that restriction: its search can include classes in the abstract search tree that are not in the abstract solution.

The experimental comparison of these techniques in section 4 was based on abstractions built by the STAR algorithm with an abstraction radius of 2 and using the max-degree criterion for selecting hubs. In this section we compare these techniques on abstractions built using a range of abstraction radii (2-7) and both criteria for selecting hubs (max-degree and random). Singleton classes are not permitted in any of the abstractions in this experiment.

An experiment of this kind simultaneously provides an evaluation of the refinement techniques and the abstraction parameters. On one hand it provides a comparison of the techniques and information about how each technique's performance is affected by the abstraction parameters. On the other hand, it provides a comparison of the different abstraction parameters. For example, it addresses the question of how much performance is affected by substituting the random-hub criterion for the more expensive max-degree criterion.

As before, the two performance measures of interest are the length of the solution found, and the amount of "work" required to find a solution. As a baseline for comparison, we include the performance of breadth first search in the original graph. "Work" is the sum of the number of edges traversed and the number of "overhead" operations required to the use the abstraction hierarchy during problem solving (for example, the work involved in transmitting a solution from one level of abstraction to the next). Roughly speaking, "overhead" accounts for about 30% of the work reported.

The cost of creating the abstraction hierarchy is not included in the "work" measure for two reasons. First, the cost of creating the abstraction is the same for all the refinement techniques and therefore does not affect the evaluation of the techniques. More importantly, because the abstractions are problem independent, the cost of creating them can be amortized over the whole set of problems that are solved. For a sufficiently large number of problems, the cost of creating an abstraction becomes negligible relative to the total problem solving cost.

Test problems were generated by choosing 500 pairs of nodes at random. Each pair of nodes, $\{s_1, s_2\}$, was used to define two problems of the form $\langle start, goal \rangle$, namely $\langle s_1, s_2 \rangle$ and $\langle s_2, s_1 \rangle$. The same 1000 problems were used for every different combination of search technique and abstraction parameter settings. All the results shown are averages over these 1000 problems.

Tables 5 and 6 show the solution length and work results using max-degree abstraction. These show that the conclusions in Section 4's comparison of refinement techniques hold for all small radii. AltO and CR do about the same of work, and OptR does more, sometimes much more. CR's solutions are the longest, about 10% longer than OptR's which, in turn, are the same length as AltO's on average (but about 10% longer than AltO's when the radius is 2).

Table 5
Solution length using max-degree abstraction.

		Radius of Abstraction					
		2	3	4	5	6	7
5-puzzle	CR	29.2	25.9	23.7	24.4	24.6	24.9
	OptR	25.1	23.5	22.6	23.9	23.9	24.2
(20.1)	AltO	24.0	23.9	22.7	22.8	22.9	23.0
Bitnet	CR	8.5	8.3	8.1	7.9	7.9	7.9
	OptR	8.0	7.9	7.9	7.9	7.9	7.9
(7.9)	AltO	8.1	8.1	7.9	7.9	7.9	7.9
Blocks-6	CR	22.3	17.2	17.2	16.2	16.3	15.3
	OptR	19.5	16.2	16.6	14.8	14.7	13.6
(13.2)	AltO	16.2	15.7	15.3	15.5	15.5	15.0
KL-2000	CR	15.2	14.2	13.1	12.7	12.4	11.9
	OptR	13.4	12.3	12.1	11.7	11.3	10.9
(10.8)	AltO	12.9	12.5	12.3	12.0	12.0	11.7
Permute-7	CR	12.7	11.0	11.0	9.8	8.7	7.5
	OptR	11.6	10.8	10.2	8.5	7.4	6.9
(6.6)	AltO	9.2	8.8	9.4	9.2	8.2	7.2
TOH-7	CR	97	89	83	83	83	82
	OptR	76	78	76	73	71	73
(67)	AltO	82	78	76	78	77	79
Words	CR	14.3	12.7	11.8	11.3	10.6	10.2
	OptR	12.4	11.1	10.8	10.3	9.8	9.4
(9.1)	AltO	11.1	10.8	10.8	10.6	10.2	10.0

The optimal solution length is shown in brackets.

Table 6
Work using max-degree abstraction

		Radius of Abstraction					
		2	3	4	5	6	7
5-puzzle	CR	233	235	277	292	281	311
	OptR	286	293	307	330	336	382
(865)	AltO	203	230	281	282	272	300
Bitnet	CR	634	788	1813	3712	4677	4948
	OptR	1249	1770	2891	4414	4878	4969
(3209)	AltO	654	802	1830	3796	4729	4958
Blocks-6	CR	682	1898	1466	1571	2646	4600
	OptR	1038	2206	1715	2894	4515	7255
(7973)	AltO	644	1896	1455	1414	2447	4681
KL-2000	CR	1363	2536	5937	6996	8433	8580
	OptR	2467	4460	8865	11855	12769	11623
(13247)	AltO	1345	2447	5143	6358	8306	9026
Permute-7	CR	474	705	735	1440	3573	7635
	OptR	1072	936	1515	3324	6147	9102
(7559)	AltO	474	666	819	1806	4772	8664
TOH-7	CR	846	734	796	755	782	815
	OptR	886	860	940	925	926	962
(3131)	AltO	764	698	746	753	768	844
Words	CR	795	1509	2561	5251	7451	9518
	OptR	1445	2552	4936	8425	10645	12389
(12002)	AltO	828	1447	2456	5412	7755	9815

The work done by breadth first search is shown in brackets
Work is measured as the number of edges plus overhead.

The choice of radius affects solution length and work in a similar manner for all the refinement techniques. Solution length steadily decreases as radius increases. Work increases as radius increases. When the radius of abstraction is sufficiently large, abstraction becomes ineffective because all the states are mapped to a few abstract states. This means that an abstract solution will provide very little guidance, if any, and the search techniques will degenerate to breadth first search. This degeneration can be seen in Table 6, where the work of the refinement techniques is much smaller than the work done by breadth first search for the small radii but the same or greater for the large radii (Permute-7 is the most extreme example). The exact radius at which this degeneration occurs varies from graph to graph, and depends primarily on the graph's diameter (maximum distance between two nodes). All but two of the graphs have small diameters (an approximate indication of the diameter is the average optimal solution length in Table 5), and abstraction of the small diameter graphs leads to little or no speedup for the larger radii.

A radius of 2 maximizes the speedup over breadth first search of every refinement technique on every graph (except Towers of Hanoi, where a radius of 3 maximizes speedup). However, for this radius CR, and, to a lesser extent, OptR, produce very long solutions. CR's solution are 50% longer than optimal, on average, and OptR's are 32% longer. Since OptR produces the optimal refinement of an abstract solution, one may conclude that its relatively poor performance when the radius is 2 is an inherent property of the general strategy of using a single abstract solution to guide search. By contrast, AltO's solutions are within 20% of optimal for all graphs and radii (including a radius of 2) except Permute-7, where its solutions, although much shorter than CR's or OptR's, are 40% longer than optimal when the radius is 2.

When the radius is 2 AltO's speedup over breadth first search is impressive, ranging from 4 (5-puzzle, Bitnet, TOH-7) to 16 (Permute-7). The combination of speedup and solution length achieved by AltO for a radius of 2 is not equalled by either of the other refinement techniques. For them to achieve the same solution length, they must do substantially more work.

Similar patterns occur when abstraction is done using random hubs (Tables 7 and 8). The solutions found by all techniques have increased in length but CR's have increased more than OptR's which, in turn, have increased more than AltO's. Thus random hub abstraction increases the difference in solution lengths produced by these three algorithms (except, perhaps, for the TOH-7 graph). OptR's solutions are now 5% longer than AltO's, and it is now the case for all graphs and all but the largest radii that the shortest of CR's solutions is longer than AltO's longest solution. AltO's solutions are still within 40% of optimal on all graphs when the radius is 2.

Table 7
Solution length using random-hub abstraction.

		Radius of Abstraction					
		2	3	4	5	6	7
5-puzzle	CR	31.2	30.2	26.0	25.9	25.8	25.1
	OptR	27.7	25.3	24.1	24.6	25.2	24.7
(20.1)	AltO	24.4	24.3	23.7	23.2	23.4	23.1
Bitnet	CR	8.6	8.5	8.3	8.1	7.9	7.9
	OptR	8.1	8.2	7.9	7.9	7.9	7.9
(7.9)	AltO	8.0	8.0	8.0	7.9	7.9	7.9
Blocks-6	CR	32.7	24.5	21.4	20.2	19.3	19.6
	OptR	25.7	22.4	18.5	18.1	18.3	18.3
(13.2)	AltO	17.6	18.0	16.8	16.5	16.7	16.8
KL-2000	CR	18.4	14.9	13.7	13.3	13.1	12.4
	OptR	14.7	13.2	12.2	12.2	12.0	11.4
(10.8)	AltO	12.8	13.0	12.6	12.6	12.3	12.0
Permute-7	CR	16.3	10.9	11.2	10.0	9.1	7.5
	OptR	12.9	10.6	10.5	8.8	7.8	6.9
(6.6)	AltO	9.2	8.9	9.3	9.2	8.6	7.2
TOH-7	CR	94.5	96.7	82.4	87.1	86.8	80.5
	OptR	82.5	77.1	69.0	77.5	75.5	73.9
(67)	AltO	76.5	80.1	77.3	81.2	80.9	78.0
Words	CR	19.5	15.4	12.1	12.0	11.6	11.1
	OptR	15.2	13.2	11.2	10.9	10.6	10.2
(9.1)	AltO	11.9	11.4	11.0	10.9	10.8	10.5

The optimal solution length is shown in brackets.

Table 8
Work using random-hub abstraction

		Radius of Abstraction					
		2	3	4	5	6	7
5-puzzle	CR	267	242	241	251	292	284
	OptR	343	282	304	332	354	366
(865)	AltO	225	213	234	242	281	278
Bitnet	CR	476	502	667	1777	3673	4664
	OptR	938	920	1507	2839	4384	4872
(3209)	AltO	589	550	687	1795	3767	4717
Blocks-6	CR	626	623	1098	1076	1148	1180
	OptR	823	970	1587	1514	1549	1762
(7973)	AltO	506	587	1115	1064	1088	1125
KL-2000	CR	1195	1908	3489	4289	5773	5032
	OptR	1993	3599	6470	6976	8867	8624
(13247)	AltO	973	1779	3595	3985	5839	5332
Permute-7	CR	654	822	795	1434	3106	7492
	OptR	937	1018	1540	3154	5529	8945
(7559)	AltO	557	793	879	1749	3947	8286
TOH-7	CR	855	797	740	784	833	846
	OptR	1017	905	848	942	972	1036
(3131)	AltO	746	713	745	788	829	889
Words	CR	760	1135	2246	3809	6329	8476
	OptR	1109	1900	3995	7044	9407	10863
(12002)	AltO	672	1074	2178	3670	6306	8553

The work done by breadth first search is shown in brackets
Work is measured as number of edges plus overhead.

In terms of "work", the change from max-degree to random hubs has had the same effect on all the refinement techniques. Work has slightly increased on three of the graphs (5-puzzle, Permute-7, and TOH-7) and substantially decreased on the others. For two graphs (Blocks-6, KL-2000), radii that previously led to degeneration are no longer problematic. Consequently, AltO's speedup over breadth first search is now even greater than with max-degree abstraction. On four of the graphs, AltO does at least 13 times less work than breadth first search.

AltO's solution lengths are remarkably insensitive to the manner in which hub states are chosen and to the choice of radius (as long as the radius is not so large as to cause degeneration). By contrast, CR and OptR are sensitive to both these choices: small and large radii must be avoided, as must random-hub selection. The robustness of a refinement technique is important because it makes problem solving performance independent of the details of the algorithm that constructs the abstraction hierarchy. This is useful because it permits the abstraction algorithm to be chosen to optimize other factors, such as the speed with which the abstraction is constructed (small radii, with randomly chosen hubs, will tend to optimize this).

7. Conclusions

In this paper we have presented a new perspective on the traditional task of problem solving and the techniques of abstraction and refinement. Our graph oriented approach produced several useful new insights and spawned improved techniques.

The graph oriented perspective led to two new families of abstraction technique. One of these, algebraic abstraction, has the desirable property of preserving the determinism and edge labels of the original graph. However, it was shown that this property has far reaching consequences: algebraic abstraction is extremely sensitive to the exact manner in which a problem is represented. The second new family of abstraction techniques was derived from an analysis of the "work" involved in successively refining a solution through multiple levels of abstraction. This analysis produced several specific principles for abstraction that guarantee speedup over blind search. These principles are easily expressed in graph oriented terms and are simple to implement in a graph based system. The STAR abstraction algorithm is one such implementation. In all our experiments the amount of work required for problem-solving was significantly less using STAR abstraction than using breadth-first search.

The graph oriented approach produced several insights into the refinement process. First, it raised the question: how much longer than the optimal refinements are the solutions produced by classical refinement, a greedy technique? An optimal refinement algorithm was implemented; its solutions were found to be only 10-15% shorter than classical refinement's. A second contribution was the technique of Alternating Search Direction, which uses the whole search tree at the abstract level for "refinement", not just the abstract solution. The AltO refinement algorithm combines Alternating Search Direction with Opportunism; this combination was uniformly superior to all other refinement techniques in our study. The final insight produced by the graph oriented approach, reported elsewhere [Holte et al., 1994], is that classical refinement and the "heuristic" use of abstract solutions are intimately related, not exclusive choices; in fact, it is possible to continuously vary a refinement strategy between these two extremes.

All the new insights and techniques in this paper, with the sole exception of the STAR algorithm, are directly applicable to problem solving with implicitly represented graphs. We conclude, therefore, that the graph oriented approach to problem solving, abstraction, and refinement is a productive one, and a useful complement to the traditional approach, which emphasizes the issues pertaining to implicit representation of graphs.

Acknowledgements

This research was supported in part by an operating grant from the Natural Sciences and Engineering Research Council of Canada and in part by the EEC's ESPRIT initiative (project "PATRICIA"). The software was written by Denys Duchier and Chris Drummond. M. B. Perez assisted in running the experiments.

References

[1] Amarel, S. (1968), On Representations of Problems of Reasoning About Actions, in *Machine Intelligence 3*, D. Michie (ed.), Edinburgh University Press, pp. 131-171.

[2] B. Donald, B., Xavier, P., Canny, J., and Reif, J., (1993), Kinodynamic Motion Planning, J*ournal of the ACM*, vol. 40(5), pp.1048-1066.

[3] Bacchus F. and Kabanza F. (1994), *Using Temporal Logic to Control Search in a Forward Chaining Planner*, unpublished.

[4] Bacchus, F. and Q. Yang (1994), Downward refinement and the efficiency of Hierarchical Problem Solving, *Artificial Intelligence*, vol. 71(1), pp. 43-101.

[5] Banerji, R.B. (1980), *Artificial Intelligence: A Theoretical Approach*, North-Holland Elsevier Science Publishing.

[6] Chakrabarti, P.P (1994), Algorithms for Searching Explicit AND/OR Graphs and their Applications to Problem Reduction Search, *Artificial Intelligence*, vol. 65(2), pp. 329-346.

[7] Chakrabarti, P.P, Ghose, S., and Desarkar, S. (1986), Heuristic Search Through Islands, *Artificial Intelligence*, vol. 29(3), pp. 339-347.

[8] Cherkassky, B. V., Goldberg, A.V. and Radzik T. (1993), *Shortest Path Algorithms: Theory and Experimental Evaluation*, technical report STAN-CS-93-1480, Computer Science Department, Stanford University.

[9] D. McAllester, D., and Rosenblitt, D., (1991). Systematic nonlinear planning, *Proc. AAAI'91*, pp. 634-639.

[10] Dillenburg, J.F. and Nelson, P.C. (1994), Perimeter Search, *Artificial Intelligence*, vol. 65(1), pp. 165-178.

[11] Doran, J.E. (1968), New Developments of the Graph Traverser, in *Machine Intelligence 2*, E. Dale and D. Michie (eds.), Oliver and Boyd, Edinburgh.

[11] Doran, J.E. and Michie D. (1966), Experiments with the Graph Traverser Program, *Proceedings of the Royal Society of London,* Series A, vol. 294, pp. 235-259.
[12] Etzioni, O., (1993), A Structureal Theory of Explanation-based Learning, *Artificial Intelligence*, vol. 60(1), pp. 93-140.
[13] Fikes, R. and Nilsson N.J. (1971), STRIPS: A New Approach to the Application of Theorem Proving to Problem Solving, *Artificial Intelligence*, vol. 2, pp. 189-208.
[14] Fink, E. and Yang Q. (1993), Characterizing and Automatically Finding Primary Effects in Planning, *Proceedings of IJCAI'93*, pp. 1374-1379.
[15] Georgeff, M. (1981), Search Methods Using Heuristic Strategies, *Proceedings of IJCAI '81*, pp. 563-568.
[16] Greiner, R. and Jurisica I. (1992), A Statistical Approach to Solving the EBL Utility Problem, Proc. *AAAI '92*, pp. 241-248.
[17] Hansson, O., Mayer, A. and Valtorta M. (1992), A new result on the complexity of heuristic estimates for the A* algorithm, *Artificial Intelligence*, vol. 55, pp. 129-143.
[18] Hart, P.E., Nilsson N.J., and Raphael B. (1968), A Formal Basis for the Heuristic Determination of Minimum Cost Paths, *IEEE Transactions on Systems Science and Cybernetics*, vol. 4(2), pp. 100-107.
[19] Hayes, P.J. (1977), A Note on the Towers of Hanoi Problem, *Computer Journal*, vol. 20 (3), pp. 282-285.
[20] Hirst, G., (1988), Semantic Interpretation and Ambiguity, *Artificial Intelligence*, vol. 34(2), pp. 131-178.
[21] Hitz, M. and Mueck T. (1994), Routing Heuristics for Cayley Graph Topologies, *Proceedings of the 10th Conference on AI and Applications (CAIA'94)*, pp. 474-476.
[22] Holte (1993), R. C., Very Simple Classification Rules Perform Well on Most Commonly-Used Datasets, *Machine Learning*, vol. 11, pp. 63-91.
[23] Holte, R. C., Acker, L, and Porter, B. W., (1989), Concept Learning and the Problem of Small Disjuncts, *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*, Morgan-Kaufmann, pp. 813-818.
[24] Holte, R. C., Perez, M.B. Zimmer R.M., and MacDonald A.J. (1995), Hierarchical A*, *Symposium on Abstraction, Reformulation, and Approximation*.
[25] Holte, R.C., Drummond C., Perez, M.B., Zimmer, R.M., and MacDonald A.J. (1994), Searching with Abstractions: A Unifying Framework and New High-Performance Algorithm, *Proc. of the 10th Canadian Conference on Artificial Intelligence (AI '94)*, Morgan Kaufman Publishers, pp. 263-270.
[26] Kavraki, L. and Latombe J.-C. (1993), *Randomized Preprocessing of Configuration Space for Fast Path Planning*, tech. report STAN-CS-93-1490, Computer Science Dept., Stanford University.
[27] Kavraki, L. and Latombe J.-C. (1994), Randomized Preprocessing of Configuration Space for Fast Path Planning, *Proceedings of the IEEE International Conference on Robotics and Automation*.
[28] Knoblock, C.A. (1990), A Theory of Abstraction for Hierarchical Planning, in *Change of Representation and Inductive Bias*, D. Paul Benjamin (ed.), Kluwer, pp. 81-104.

[29] Knoblock, C.A. (1991), *Automatically Generating Abstractions for Problem Solving*, technical report CMU-CS-91-120, Computer Science Department, Carnegie-Mellon University.
[30] Knoblock, C.A. (1994), Automatically Generating Abstractions for Planning, *Artificial Intelligence*, vol. 68(2), pp. 243-302.
[31] Knoblock, C.A., Minton S., and Etzioni O. (1991a), Integrating Abstraction and Explanation-Based Learning in PRODIGY, *Proc. AAAI '91*, pp. 541-546.
[32] Knoblock, C.A., Tenenberg, J.D. and Yang Q. (1991b), Characterizing Abstraction Hierarchies for Planning, *Proc. AAAI '91*, pp. 692-697.
[33] Korf, R.E., (1985a), Macro-operators: a weak method for learning, *Artificial Intelligence*, vol. 26, pp. 35-77.
[34] Korf, R.E., (1985b), Depth-1st Iterative-Deepening - An Optimal Admissible Tree-Search, *Artificial Intelligence*, vol. 27(1), pp. 97-109.
[35] Korf, R.E. (1980), Towards a Model of Representation Changes, *Artificial Intelligence*, vol. 14(1), pp. 41-78.
[36] Minsky, M. (1963), Steps Toward Artificial Intelligence, in *Computers and Thought*, E. Feigenbaum and J. Feldman (eds.), McGraw-Hill, pp. 406-452.
[37] Minton, S. (1985), Selectively generalizing plans for problem solving, *Proceedings of IJCAI '85*, pp. 596-602.
[38] Minton, S. (1990), Quantitative Results Concerning the Utility of Explanation-based Learning, *Artificial Intelligence*, vol.42(2-3), pp.363-392.
[39] Newell, A. and Simon H. (1963), GPS, a Program that Simulates Human Thought, in *Computers and Thought*, E. Feigenbaum and J. Feldman (eds.), McGraw-Hill, pp. 269-278.
[40] Nilsson, N. (1980), *Principles of Artificial Intelligence*, Tioga Press.
[41] Pearl, J. (1984), *Heuristics*, Addison-Wesley.
[42] Pohl, I. (1970), Heuristic Search Viewed as Path Finding in a Graph, *Artificial Intelligence*, vol. 1(3), pp. 193-204.
[43] Pohl, I. (1971), Bi-directional Search, in *Machine Intelligence 6*, B. Meltzer and D. Michie (eds.), Edinburgh University Press, pp. 127-140.
[44] Porter, B. W., Bareiss, E. R., and Holte, R. C., (1990), Concept Learning and Heuristic Classification in Weak-Theory Domains, *Artificial Intelligence*, 45(1-2), pp. 229-263.
[45] Prieditis, A., and Janakiraman B. (1992), Becoming Reactive by Concretization, *Proceedings of the Workshop on Change of Representation and Problem Reformulation*, NASA-Ames technical report FIA-92-06, Mike Lowry (ed.), pp. 153-159.
[46] Prieditis, A., and Janakiraman B. (1993), Generating Effective Admissible Heuristics by Abstraction and Reconstitution, *Proc. AAAI '93*, pp. 743-748.
[47] Reinefeld, A. and Marsland T.A. (1994), Enhanced Iterative-Deepening Search, *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 16(7), pp. 701-710.
[48] Sacerdoti, E. (1974), Planning in a hierarchy of abstraction spaces, *Artificial Intelligence*, vol. 5(2), pp. 115-135.
[49] Sen, A.K. and Bagchi A. (1989), Fast Recursive Formulations for Best-First Search that Allow Controlled Use of Memory, *Proc. IJCAI '89*, pp. 297-302.
[50] Valtorta, M. (1984), A result on the computational complexity of heuristic estimates for the A* algorithm, *Information Sciences*, vol. 34, pp. 48-59.

[51] Yang, Q. and Tenenberg J.D. (1990), Abtweak: Abstracting a nonlinear, least commitment planner, *Proc. AAAI '90*, pp. 204-209.

APPENDIX A. Graphs Used in the Experiments

5-puzzle.

This is a 2×3 version of the 8-puzzle. There are 6 positions, arranged in 2 rows and 3 columns, and 5 distinct tiles, each occupying one position. The unoccupied position is regarded as containing a blank. Tiles adjacent to the unoccupied position can be moved into it, thereby moving the blank into the position just vacated.

The state space comprises two unconnected regions each containing 360 states. One of these regions is used in the experiment in section 5.1. In all other experiments, we have connected the space by adding a single edge between one randomly chosen state in each of the two regions. Two-thirds of the states have only 2 successors, the other states have 3 successors.

Bitnet

This is a map (not up-to-date) of 3483 sites and their interconnections in the Bitnet computer network. Branching factors vary considerably – one of the nodes has a branching factor of 82, but many nodes are connected to only one other node. The average branching factor is 2.25.

Blocks-6

There are n distinct blocks each of which is either on the "table" or on top of another block. There is a "robot" that can hold one block at a time and execute one of four operations: put the block being held onto the table, put it down on top of a specific stack of blocks, pick up a block from the table, and pick up the block on top of a specific stack.

We used the 6-block version of this puzzle, which has 7057 states. The branching factor varies considerably from one to six, depending on the number of stacks in the state. The maximum distance between two states is 11.

KL-2000

This is the graph "connect2000_1000.res" provided to us by Lydia Kavraki and J-C. Latombe, of Stanford University. It is produced by their algorithm for discretizing the continuous space of states/motions of robots with many degrees of freedom [Kavraki and Latombe, 1993, 1994]. This graph has 2736 nodes and an average branching factor of 10.5.

Permute-7.

A state is a permutation of the integers from 1 to n.. There are $n-1$ operators numbered 2 to n. Operator k reverses the order of the first k integers in the current state. For example, applied to the state $[3,2,5,6,1,7,4,\cdots]$ operator 4 produces $[6,5,2,3,1,7,4,\cdots]$. Operator n reverses the whole permutation. This is a particular instance of a Cayley graph, a family of graphs being studied [Hitz and Mueck, 1994] as a promising interconnection topology for multiprocessor networks.

We used $n = 7$, which gives rise to 7! = 5040 states. All operators are applicable in every state, so each state has 6 successors. The maximum distance between two states is 14.

TOH-7

In the Towers of Hanoi puzzle there are three pegs and n different size disks sitting on the pegs with the smaller disks above the larger disks on the same peg. The top disk on a peg may be moved onto an empty peg or onto the top of any peg whose top disk is smaller than the one being moved.

We used the 7-disk version of this puzzle, which has 2187 states. Each state (except for the 3 states in which all disks are on the same peg) has 3 successors. The maximum distance between two states is 128.

Words

This graph was obtained from The Stanford GraphBase which was compiled by Donald Knuth and is available in directory *pub/sgb* at the ftp site *labrea.stanford.edu*. The nodes in the graph are the 5 letter words in English. Two words are connected by an edge if they differ in exactly one letter. We use the largest connected component of this graph, which has 4493 nodes and an average branching factor of 6.