published in the Proceedings of the 10th Canadian Conference on Artificial Intelligence (AI'94), pp. 263-270, Morgan-Kaufman, 1994.

Searching With Abstractions: A Unifying Framework and New High-Performance Algorithm[1]

R.C. Holte, C. Drummond, M.B. Perez
Computer Science Department
University of Ottawa
Ottawa, Ontario, CANADA K1N 6N5
{holte , cdrummon , mbperez}@csi.uottawa.ca

R.M. Zimmer, A.J. MacDonald
Electrical Engineering Department
Brunel University
Uxbridge, Middlesex, ENGLAND UB8 3PH
{Robert.Zimmer , Alan.MacDonald}@brunel.ac.uk

Abstract

This paper presents a common algorithmic framework encompassing the two main methods for using an abstract solution to guide search. It identifies certain key issues in the design of techniques for using abstraction to guide search. New approaches to these issues give rise to new search techniques. Two of these are described in detail and compared experimentally with a standard search technique, classical refinement. The "alternating opportunism" technique produces shorter solutions than classical refinement with the same amount of search, and is a more robust technique in the sense that its solution lengths are very similar across a range of different abstractions of any given space.

1 Introduction

Heuristic search is ubiquitous in AI. A particular form of heuristic search, state-space search, is the cornerstone of many AI systems, including most planning and problem-solving systems. Consequently, techniques for speeding up heuristic search, or for automatically generating or improving heuristics, are of central importance to AI.

Abstraction is a widely studied means of speeding up state-space search. Instead of directly solving a problem in the original search space, the problem is mapped into and solved in an "abstract" search space. The abstract solution is then used to guide the search for a solution in the original space.

Abstraction very often produces impressive performance improvements (e.g. (Knoblock,1991)), but it is not guaranteed to speedup search. The guidance provided by an abstract solution is not even guaranteed to reduce the amount of search in the original space; if the guidance is positively misleading, it will increase the amount of search in the original space (e.g. (Holte et al., 1992)). Even if abstraction does speedup search in the original space, there are overhead costs associated with abstraction: the cost of creating an abstract space, and the costs associated with finding one or more abstract solutions and using them to guide search in the original space. The former cost can be amortized if there are many searches of the same space and the same abstract space is used each time. The speedup in the original space that results from using abstraction must more than compensate for these costs in order for a net speedup to be achieved.

Research on abstraction aims to find methods for creating and using abstract spaces that reliably speedup search without undue degradation in solution quality (i.e. the length of the solution). Most research on abstraction has investigated different methods for creating an abstract space; in this paper we investigate different methods for using an abstract solution to guide search.

There are two principal methods for using an abstract solution to guide search. In one method, the length of the abstract solution is used as a heuristic estimate of the distance to the goal (Pearl,1984; Prieditis and Janakiraman, 1993). In the other method, the individual steps of the abstract solution are used as a sequence of subgoals whose solutions link together to form the final solution (Minsky,1963; Sacerdoti,1974; Knoblock,1991; Yang and Tenenberg,1990). In the latter method, the abstract solution serves as a skeleton for the final solution; the process of "fleshing out" the abstract solution is called "refinement".

[1] 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").

Until now, these two methods have been seen as mutually exclusive. This paper presents a computational framework in which the two methods are seen to be very similar. As algorithms, their differences are "minor", in the sense of being small in number and highly localized (i.e. independent of one another and of other aspects of the algorithms). Consequently, hybrids of the two methods can easily be constructed. But there is a much more important consequence. The algorithmic differences correspond to specific issues that arise in designing techniques for using abstract solutions. Having clearly identified these issues for the first time, it becomes immediately apparent that there are numerous promising alternatives to the existing methods. Two of these alternatives, called "path-marking" and "alternating opportunism", are examined in this paper. These techniques are experimentally compared with the classical refinement technique on a range of problems and abstraction methods. "Alternating opportunism" emerges as the best of the three techniques compared. It is between 3 and 12 times faster than breadth-first search, and very reliably produces near-optimal solutions.

Section 2 describes the method used to create abstractions, and the parameters of this method varied in the experiment. Section 3 describes the general computational framework encompassing both standard methods for using an abstract solution, and presents the three specific techniques studied in the experiment. Section 4 describes the experimental setup and results.

2 The "Star" Method of Abstraction

In most AI search systems, a search space is defined implicitly, typically in the STRIPS notation (Fikes and Nilsson,1971)). A state is defined to be a set of sentences in a formal language (containing constants, variables, predicate symbols, etc.). The successor relation between states is represented by operators that map one state to another by adding to or deleting from the set of sentences (i.e. the state) to which it is applied. Each operator has preconditions, stated in the formal language, specifying to which states the operator may be applied. An abstract search space is created by removing symbols from the formal language and/or the definitions of the operators and states (e.g. see (Knoblock et al., 1991)).

By contrast, in our system a search space is represented by an explicit graph. A state is a node in the graph; the successor relation between states is represented by edges connecting each node to its successors. In this way of representing search spaces, search space SSA is an abstraction of search space SSB iff there is a graph homomorphism from SSB to SSA.

In a graph homomorphism, each state in the abstract space, SSA, corresponds to one or more states in SSB. We view the abstract state as a class "containing" the corresponding states from SSB. Henceforth the terms "class" and "abstract state" will be used interchangeably, and the term "state" will mean a state in the original space. In the figures a class will be indicated by drawing a circle around the states it contains.

Each edge in the abstract space connects one class to another. There must be an edge in the abstract space corresponding to each edge in the original space. Thus, if there is an edge in the original space from state S1 to state S2, then there must be an edge in the abstract space from the class containing S1 to the class containing S2. If S1 and S2 are in the same class, an edge from S1 to S2 corresponds to an identity edge in the abstract space. Identity edges are not drawn in the figures.

The use of an explicit graph has the obvious drawback that it is feasible only for relatively small search spaces (the largest we have studied to date had 50,000 states). But it has the very great advantage, for research purposes, of flexibility and generality. It permits a very wide range of different abstraction-creating and abstraction-using techniques to be easily implemented and investigated. The ultimate aim of our research is to develop techniques that operate on implicit graphs; and indeed, the principles underlying new search techniques described in this paper can be used in the ordinary STRIPS representation as readily as in the explicit graph representation.

The standard STRIPS-based definition of abstraction is a highly restricted type of graph homomorphism. Some of the limitations and weaknesses of this type of graph homomorphism are discussed in (Holte et al.,1993). To overcome them, we have been developing alternative methods of abstraction.

The "star" method of abstraction was first investigated in (Mkadmi,1993). Each class consists of a "hub" state and all the neighbours of the hub within a given distance, R, called the "radius" of abstraction. The classes are built one at a time; once a state is included in a class it is ineligible to be included in any other. The process is repeated until all states have been assigned to a class (it may happen that a class contains just one state). Then edges are added between classes, as described above, to complete the construction of the abstract space.

As with any abstraction method, the star method can be applied recursively to the abstract space it creates in order to construct a "hierarchy" of abstract spaces. In our current implementation, successively more abstract search spaces are added to the hierarchy until the trivial search space is produced (the trivial search space consists of just one class). For simplicity, the discussion will speak as if there were only two levels in the hierarchy, the original search space (which is always at the bottom of the hierarchy) and one abstract search space. But all of the discussion applies equally to any two adjacent levels in a larger abstraction hierarchy.

The two main decisions in using the star method are: how to choose the hub states, and what value of R to use. These are parameters that will be varied in the experiments below. We will consider two ways of choosing hub states: choose a random state, and choose the state having the greatest number of immediate neighbours. The radius will be varied from 2 (which means a class contains only the hub and its immediate neighbours) to 9.

The value of R has great impact on the performance of search systems that use abstraction. For example, the total amount of search done at all levels of abstraction is bounded above by $W(R)\times(2R)^A$, where A is the number of levels in

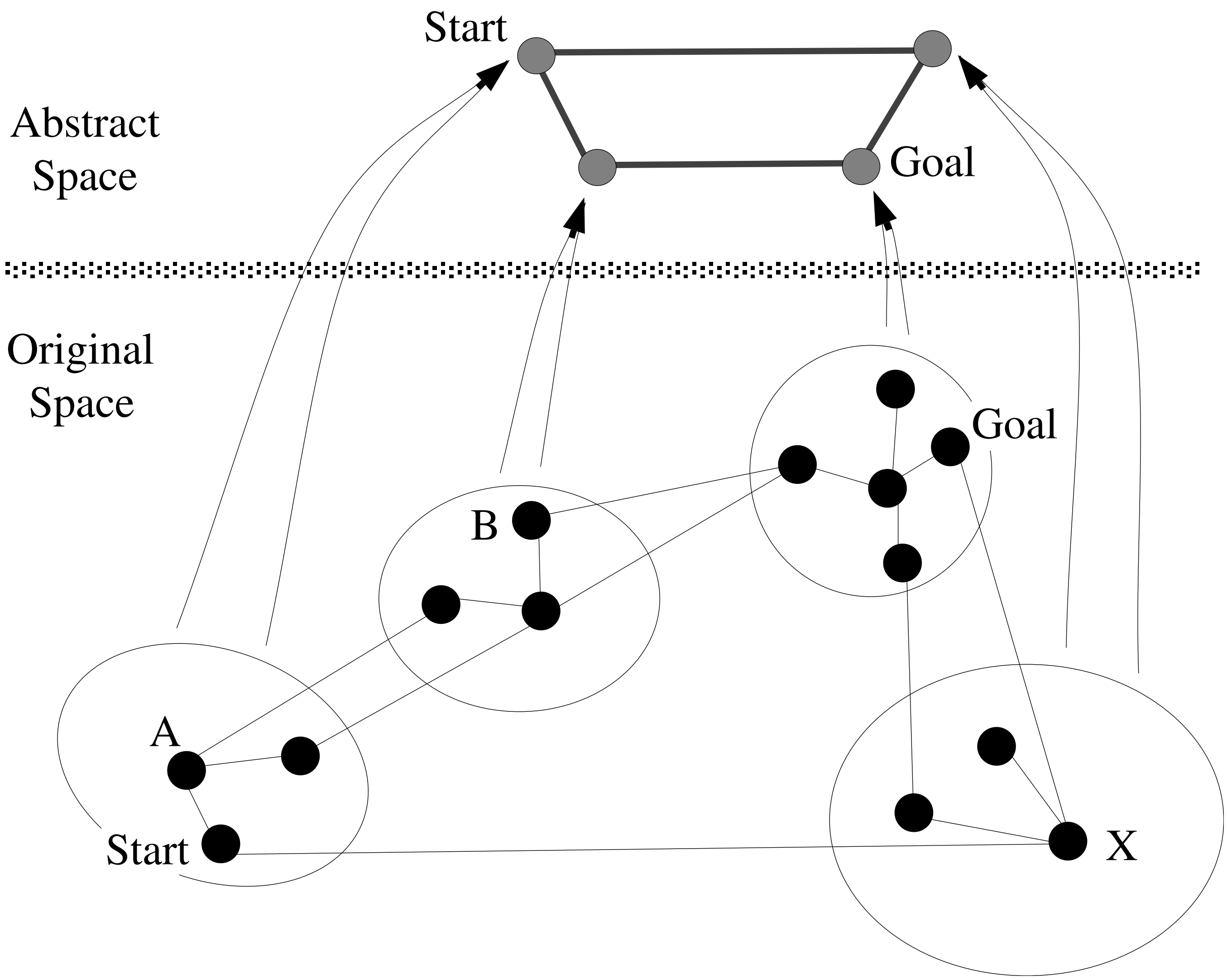

Figure 1.

the abstraction hierarchy and W(R) is the amount of search done to solve a problem when the start and goal are both in the same class of radius R (Mkadmi,1993). Solution length is also affected by R, typically decreasing as R increases. Certain overhead costs increase as R increases, but others decrease. Generally speaking, the choice of R affects – in antagonistic ways – the quantity and quality of the guidance that an abstract solution provides for search in the original space. When R is small, there are few states in each class. Consequently, knowing which classes are in the abstract solution provides information about very few states, but the information is specific to those states and therefore is highly reliable. When R is large there are many states in each class: the abstract solution provides information about many more states, but the information is not very specific and so is indiscriminate, possibly even misleading.

3 Search Methods That Use Abstractions

There are two main ways that abstract solutions can be used to guide search. "Heuristic" methods are based on the observation that distance (number of edges in the shortest path) between two states is greater than or equal to the distance, in the abstract space, between the corresponding classes. For example, in Figure 1, the distance between Start and Goal in the original space is 3 (the shortest path passes through state X); the distance between the corresponding classes is 2. Distance in the abstract space is therefore an admissible heuristic and can be used in the A* algorithm (Hart et al., 1968) to find optimal solutions.

When the A* algorithm visits state S it computes h(S), an estimate of the distance from S to a goal state. If h(S) is defined to be the abstract distance from the class containing S to a goal class, computing h(S) involves searching in the abstract space. The result of this search is a shortest path, i.e. a sequence of classes, connecting S's class to a goal class. In Figure 1, when A* visits state A, it would compute h(A) by finding a shortest path between the class containing A and the goal class. In this example there are two shortest paths. h(A) would be 2, the length of whichever of the two was actually found.

We note, however, that the abstract path found in the course of computing h(A) provides the information needed to compute h(-) for many other states. Suppose, in the

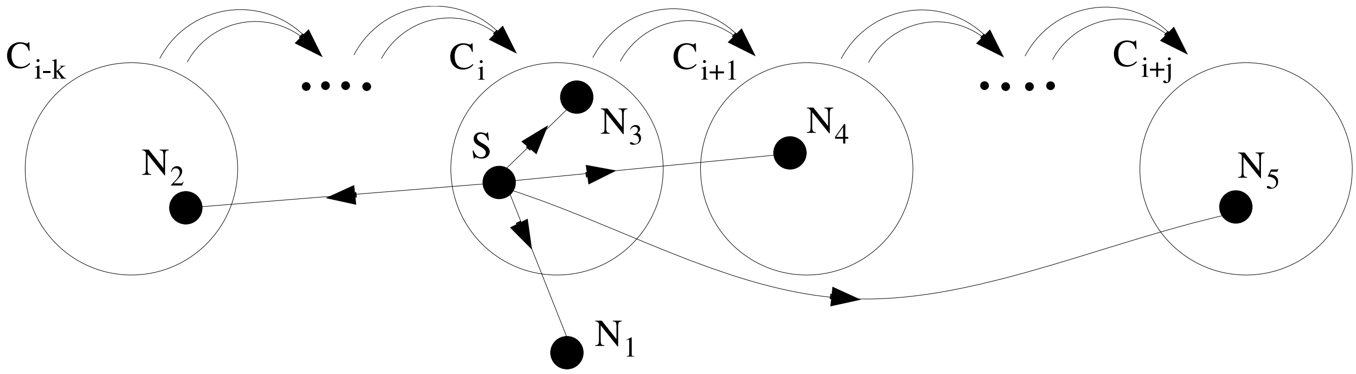

Figure 2.

example, that the path found is the one including B's class. Firstly, for every state S in the same class as A, h(S)=h(A). Secondly, this path also allows us to compute h(B), because it includes a shortest path from B's class to the goal class. And, of course, it allows us to compute h(S) for every S in the goal class. In general, one abstract shortest path enables the computation of h(S) for every state S contained in every class on the path.

This h(-) information is easily cached with each state. Now when A* visits state S it only needs to search in the abstract space if the class containing S has not occurred on any of the abstract paths previously computed. Although this may seem like a minor implementation detail, it provides a conceptual link to the other method of using abstract paths to guide search.

"Refinement" methods find one abstract path from Start to Goal and use each class in this path as a subgoal when searching in the original space. The Start state, S1, is, by definition, in the first class, C1, on this path. A path (in the original space) is sought, typically using breadth-first search, from S1 to any state, S2, in C2, the second class on the abstract path. In searching for this path, only states in C1 are considered: all other states are disregarded (this is the graph equivalent of "goal protection"). In our example (Figure 1), a path in the original space is sought from Start to a state in the same class as B. In searching for this path, state X, and other states in X's class are ignored, because that class is not on the abstract path.

Having reached S2 in class C2, a path wholly within C2 is sought from S2 to any state, S3, in C3, the third class in the abstract path. This process, called refinement, is repeated for each successive class on the abstract path until a path in the original space has been constructed from Start to a state, Sg, in Cg, the class containing the goal state. Then, as the final step, a path wholly within Cg is sought from Sg to the actual goal state.

Figure 2 shows a portion of the abstract path $C_1...C_n$ being refined and a typical intermediate situation during refinement, in which we are searching forward from state S, in class C_i, looking for any state in class C_{i+1}. The five successors of S illustrate the different possibilities for a state's successors. N_1 is not in a class on the abstract path; it will be ignored. N_2 is in a previous class; it too will be ignored. N_3 and N_4 are in the current and next class, respectively. N_4 would be preferred to N_3, and would result in C_{i+1} becoming the current class. But if S happened not to have any successors in the next class, states in the same class (N_3) would be pursued during refinement. State N_5 is in a class more than 1 ahead of the current class. If the abstract path is a shortest path, such states cannot exist. However, refinement techniques (unlike the heuristic methods described above) are not guaranteed to find shortest paths, so this situation certainly can arise. In classical refinement, this opportunity to jump ahead is ignored. Techniques that exploit such opportunities we call "opportunistic".

Thus, when a state is visited during refinement, it is necessary to know whether or not it is in a class on the abstract path, and, if so, the position of the class in the abstract path. This is precisely the information that would be cached when h(Start) is computed by the implementation of the heuristic method described above. It is clear therefore that the two methods for using abstract paths are very similar, differing only in two respects:

(1) the refinement method restricts itself to states that are part of the first abstract path computed, whereas the heuristic method considers all states (and may have to find additional abstract paths to do so);

(2) the refinement method does not take into account the distance of a state from the start state, whereas the heuristic method adds this to the abstract distance to the goal in order to compute a state's overall "score".

These two classical methods are not the only ways in which the information derived from an abstract path can be used to guide search. The following are two novel variants of the classical refinement technique.

Path-Marking

Classical refinement derives its efficiency from two sources: from ignoring states that are not part of the abstract path, and from associating a position in the abstract path with each state. The path-marking technique uses only the first of the sources: it does ordinary breadth-first search (in the original space) but ignores all states that are not in classes on the abstract path. This is guaranteed to find the shortest possible refinement of the abstract path.

Alternating Search Direction

While searching in the abstract space, many classes may be visited that, in the end, are not on the abstract path. For these classes the distance to a goal class is not known; that distance is known only for classes on the abstract path. However, the distance to the abstract start class is known for all classes visited during the search, because the start class is where the search began. This information has no utility if search in the original space is in the same direction as search in the abstract space (e.g. from start to goal). But, if search in the original space proceeds in the opposite direction, from goal to start (using the inverse of the successor relation), then distance from the start is precisely what is need to guide search. If there are several levels of abstraction, search direction alternates from one level to the next.

Using this technique, search is not confined to states that are on the abstract path: heuristic distance information is available for every state in every class visited during the abstract search. This means that the solutions found by alternating search direction could be shorter than those found using the path-marking technique. Such solutions would not, of course, be refinements of the abstract path in the normal sense.

Our implementation of this alternating-direction technique is "opportunistic", as defined above.[2] Like classical refinement, alternating opportunism never moves "away" from the destination: once it finds a state whose heuristic distance from the destination is D, it ignores all states whose heuristic distance is greater than D. For this reason, the solutions found by alternating opportunism could be longer than those found by the path-marking technique.

4 Experimental Comparison

This experiment compares three search techniques — classical refinement (abbreviated CR), path-marking (PM), and alternating opportunism (AO). The two performance measures of interest are the length of the solution found, and the amount of "work" required to find a solution.

We originally measured work in terms of CPU time, but this proved extremely sensitive to low-level programming details of no significance to the algorithms themselves. In this paper, work is measured by counting the number of edges traversed during search (at all levels of abstraction) and the number of "overhead" operations performed during search (for example, to pass the heuristic distance information from one level of abstraction to the next). As these two counts have roughly comparable units, they can sensibly be added to give a composite "total work" figure. Overhead costs associated with creating the abstraction are not included in the "work" measure because our aim is to compare different techniques for using abstractions; the cost of creating the abstractions is the same for them all.

Also of interest is the robustness of a search technique: to what extent does good performance depend upon the abstract space that is used. To investigate this, abstractions were created using several different radii and two different methods for choosing the hub states (see section 2).

Four different search spaces, derived from well-known puzzles, were used in the experiment. These are described below. For each space, 100 pairs of states were chosen randomly. Each pair <S1,S2> gives rise to two problems: <start=S1,goal=S2> and <start=S2,goal=S1>. The same 200 problems were used for every different combination of system and abstraction-parameter settings. All the results shown are averages over these 200 problems.

A simple breadth-first search system was also run on the 200 test problems for each search space. Its performance figures allow us to compare the solutions found using abstraction to the optimal solutions, and to measure how much work is being saved by using abstraction to guide search.

4.1 Search Spaces

Towers of Hanoi. The 7-disk version has $3^7 = 2187$ states. Each state (except for the 3 extreme "corners") has 3 successors, but the effective branching factor is considerably less than 3 because of the structure of the space. The maximum distance between two states is $2^7 = 128$.

5-puzzle. This is a 2×3 version of the 8-puzzle. The state space comprises two unconnected regions each containing 360 states. 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, which means the branching factor at these states is effectively 1 (because every edge has an inverse, so one of the 2 successors will be the state from which the current one was reached). The other states have 3 successors.

Blocks World. There are 6 distinct blocks, numbered 1 to 6, each of which is either on the "table" or on top of another block. There is a "hand" 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. With 6 blocks there are 7057 states. Unlike the other search spaces, in the blocks world the branching factor varies considerably from one state to another, depending as it does on the number of stacks in the state. The maximum distance between two states is 11.

Permutation. A state is a permutation of the integers 1−7; there are 7! = 5040 states. There are 6 operators numbered 2 to 7. Operator N reverses the order of the first N integers in the current state. For example, applied to the state [3,2,5,6,1,7,4] operator 4 produces [6,5,2,3,1,7,4]. Operator 7 reverses the whole permutation. All operators are

[2] we have also implemented a variation of classical refinement that is opportunistic. This was not used in the experiments because with star abstraction and the types of graphs used in the experiments, it can be proved that the heuristic distance must decrease by 1 if search is confined to the abstract path.

TABLE 1. Solution Length.
States with the most neighbours are used as the hubs of the abstract classes.
The optimal solution length is shown in brackets.

	Towers of Hanoi (66)			5-puzzle (21)			Blocks World (9.5)			Permutation (6.2)		
radius	CR	PM	AO	CR	PM	AO	CR	PM	AO	CR	PM	AO
2	98	88	80	29	27	25	14.7	13.2	11.2	11.6	10.6	8.3
3	101	77	76	27	24	24	11.5	11.0	10.8	9.5	9.3	8.0
4	80	72	75	24	23	23	12.1	11.6	11.2	9.7	8.9	8.1
5	82	72	76	29	25	25	11.9	11.2	11.3	8.1	7.3	7.4
6	82	72	77	27	26	25	11.6	10.5	10.8	7.1	6.4	6.8
7	79	69	75	26	25	24	10.3	9.7	10.2	7.3	6.4	6.8
8	73	69	71	25	25	24	9.5	9.5	9.5	6.2	6.2	6.2
9	78	68	74	26	25	24	9.5	9.5	9.5	6.2	6.2	6.2

TABLE 2. Total Work (#edges + overhead).
States with the most neighbours are used as the hubs of the abstract classes.
The work done by ordinary breadth-first search is shown in brackets.

	Towers of Hanoi (3058)			5-puzzle (802)			Blocks World (3936)			Permutation (5570)		
radius	CR	PM	AO	CR	PM	AO	CR	PM	AO	CR	PM	AO
2	894	1048	767	299	362	255	724	1073	762	512	868	460
3	867	903	711	251	308	238	1200	1339	1227	616	762	609
4	776	903	751	294	333	301	902	1127	889	750	1191	790
5	752	927	740	314	371	300	1268	1767	1395	1407	2443	2527
6	777	944	765	335	392	321	2124	2906	2146	3055	3926	5454
7	802	950	828	333	392	325	3424	4258	4149	5739	7053	6416
8	932	1079	938	348	413	346	5915	5915	5973	7982	7982	7982
9	916	1082	946	386	469	389	5973	5973	5973	7982	7982	7982

applicable in every state, so each state has 6 successors. The maximum distance between two states is 14.

4.2 Results

Tables 1 and 2 show the solution length and total work results when states having the most immediate neighbours are used as the hubs of abstract classes. The solutions found using abstraction are relatively short, within 30% of the optimal length in most cases and never more than double the optimal length. Total work is impressively small for small radii, in most cases, but for large radii search using abstraction is not cost-effective. This is particularly evident in the Blocks World and Permutation spaces, whose maximum distance between states is small: a radius of 7 or greater causes almost all states to be put into the same class, making abstraction of little value. Only in the Towers of Hanoi space, where the maximum distance between two states is large, does abstraction with large radii pay off. In the comparisons that follow, data for radius ≥ 7 for the Blocks World and Permutation spaces will be ignored.

The experiment shows that PM's solutions are roughly 10% shorter than CR's. Regarding total work, it is possible for PM to do less than CR, if their solutions are different lengths. In the experiment, this never happened: PM always did more work, sometimes much more. In most circumstances, the 10% improvement in solution length PM provides is probably not sufficient to justify the additional work.

CR produces its poorest solutions when radius=2. It was hoped that PM would find much shorter solutions in this case, but the experiment reveals that its solutions are just 10% shorter, as usual. Since PM produces the optimal refinement of an abstract solution, one may conclude that the relatively poor performance when radius=2 is an inherent property of the the general strategy of using a single abstract solution to guide search.

AO does not follow the same general strategy, and its solutions are 10-15% shorter than PM's when radius=2. For larger radii, AO and PM give solutions of similar lengths – PM's are slightly shorter in the Towers of Hanoi space, AO's in the Permutation space. However, AO always does much less work than PM. AO does the same amount of work as CR and produces shorter solutions, sometimes very much shorter. The single exception is the Permutation space, where AO begins to degenerate to breadth-first search slightly sooner (radius=5) than CR.

The same patterns arise when random states are used as the hubs when constructing abstract classes (see Tables 3 and 4). The solutions found by all techniques have increased in length but CR's have increased more than PM's which, in turn, have increased more than AO's. Consequently, the difference in solution lengths has become

TABLE 3. Solution Length.
Randomly chosen states are used as the hubs of the abstract classes.
The optimal solution length is shown in brackets.

	Towers of Hanoi (66)			5-puzzle (21)			Blocks World (9.5)			Permutation (6.2)		
radius	CR	PM	AO	CR	PM	AO	CR	PM	AO	CR	PM	AO
2	91	80	75	32	29	26	27.2	21.3	12.7	15.6	12.0	8.4
3	95	81	83	30	29	26	22.0	19.1	13.1	11.4	10.0	8.1
4	82	72	77	27	25	25	19.7	16.5	13.6	10.1	9.6	8.5
5	86	74	78	28	25	24	14.3	13.6	12.1	9.5	8.5	8.6
6	95	86	79	26	26	24	15.6	14.8	13.0	9.0	7.7	8.0
7	88	81	79	25	25	25	17.0	16.2	13.8	6.9	6.4	6.6

TABLE 4. Total Work (#edges + overhead).
Randomly chosen states are used as the hubs of the abstract classes.
The work done by ordinary breadth-first search is shown in brackets.

	Towers of Hanoi (3058)			5-puzzle (802)			Blocks World (3936)			Permutation (5570)		
radius	CR	PM	AO	CR	PM	AO	CR	PM	AO	CR	PM	AO
2	847	1021	765	316	411	285	1310	1526	1070	806	1015	817
3	794	957	763	282	338	264	571	783	546	800	978	806
4	734	870	738	269	337	264	699	1027	738	811	1250	1038
5	776	924	763	280	360	265	902	1093	896	1258	2491	1456
6	931	1111	830	306	363	303	808	1048	780	3142	5064	3674
7	942	1106	916	292	354	302	964	1297	964	6295	7008	6566

greater (except, perhaps, for the Towers of Hanoi). It is now the case that the shortest of CR's solutions for any radius is comparable in length to AO's longest solution.

For radii other than 2, total work has decreased in most cases; dramatically so in the Blocks World. But all the search techniques have benefited roughly equally: CR and AO still do comparable amounts of work, and PM does significantly more.

AO'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 AO to degenerate to breadth-first search). This robustness of the search technique is important because it relieves the abstraction-constructing system of the responsibility of ensuring good solutions. The abstraction-construction system can therefore focus on other issues; for example, it could attempt to construct abstract spaces that were "meaningful" to a human in the sense that each class has a succinct description.

6 Conclusions

This paper has provided a common algorithmic framework encompassing the two main methods of using an abstract solution to guide search. In doing so, it has identified certain key issues in the design of techniques for using abstraction to guide search. The clear identification of these issues is important for research in abstraction, because doing so sharply focuses research. In response to these issues, two new new search techniques have been developed — path-marking and alternating opportunism. These have been compared experimentally with a standard search technique, classical refinement. Path-marking is guaranteed to find the optimal refinement of a given abstract path; the experiments show that classical refinement, in general, produces refinements whose length are within 10% of the optimal refinement. However, the optimal refinement of a somewhat arbitrarily chosen abstract path is not necessarily close to being an optimal solution. The alternating opportunism technique is based on a generalization of the notion of "refinement of an abstract path". It produces shorter solutions than classical refinement with the same amount of search. It is also a more robust technique, in the sense that its solution lengths are very similar across a range of different abstractions of any given space.

This study has been carried out with a system in which search spaces are represented as explicit graphs. However, this is incidental to the general framework and specific techniques developed and compared in the paper. Path-marking, alternating search direction, and opportunism could all be implemented as search techniques in a traditional STRIPS-style search system.

References

[Fikes and Nilsson, 1971] Fikes, R. and N.J. Nilsson (1971), "STRIPS: A New Approach to the Application of Theorem Proving to Problem Solving", Artificial Intelligence, vol.2, pp.189-208.

[Hart et al., 1968] Hart, P.E., N.J. Nilsson, and B. Raphael (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.

[Holte et al., 1993] Holte, R.C., R. Zimmer, and A.J. Macdonald (1993), "A Study of the Representation-Dependency of Abstraction Techniques", ML'93 workshop on Knowledge Compilation and Speedup Learning, June 1993. (unpublished)

[Holte et al., 1992] Holte, R.C., R. Zimmer, and A. MacDonald (1992), "When does Changing Representation Improve Problem-Solving Performance ?", in M. Lowry (ed.), Proceedings of the Workshop on Change of Representation and Problem Reformulation, NASA Ames technical report FIA-92-06. May 1992.

[Knoblock, 1991] Knoblock, C.A. (1991). Automatically Generating Abstractions for Problem Solving. tech. report CMU-CS-91-120, Computer Science Dept., Carnegie-Mellon University.

[Knoblock et al., 1991] Knoblock, C.A., J.D. Tenenberg, and Q. Yang (1991), "Characterizing Abstraction Hierarchies for Planning", Proc. AAAI, pp.692-697.

[Minsky, 1963] Minsky, M. (1963), "Steps Toward Artificial Intelligence", in Computers and Thought, E. Feigenbaum and J. Feldman (eds.), McGraw-Hill, pp.406-452.

[Mkadmi, 1993] Mkadmi, T. (1993). Speeding Up State-Space Search by Automatic Abstraction. Master's Thesis. Computer Science Dept., University of Ottawa.

[Pearl, 1984] Pearl, J. (1984), Heuristics, Addison-Wesley.

[Prieditis and Janakiraman, 1993] Prieditis, A., and B. Janakiraman (1993), "Generating Effective Admissible Heuristics by Abstraction and Reconstitution", Proc. AAAI, pp.743-748.

[Sacerdoti, 1974] Sacerdoti, E. (1974). Planning in a hierarchy of abstraction spaces. *Artificial Intelligence*, vol. 5(2), pp. 115-135.

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