UNIVERSITY OF ALBERTA

New Approaches To Moving Target Search

BY

Stanley Melax ©

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

DEPARTMENT OF COMPUTING SCIENCE

Edmonton, Alberta
Spring 1994

ISBN 0-612-11296-9

UNIVERSITY OF ALBERTA

RELEASE FORM

NAME OF AUTHOR: Stanley Melax

TITLE OF THESIS: New Approaches To Moving Target Search

DEGREE: Master of Science

YEAR THIS DEGREE GRANTED: 1994

(Signed) Stan Melax

Stanley Melax
11719-38A Avenue
Edmonton, Alberta,
Canada. T6J 0L8

Date: December 22 1993

UNIVERSITY OF ALBERTA

FACULTY OF GRADUATE STUDIES AND RESEARCH

The undersigned certify that they have read, and recommend to the Faculty of Graduate Studies and Research for acceptance, a thesis entitled **New Approaches To Moving Target Search** submitted by Stan Melax in partial fulfillment of the requirements for the degree of Master of Science.

[illegible]
Dr. J. Schaeffer (Supervisor)

J.E. Lewis
Dr. J. E. Lewis (External)

[illegible]
Dr. P. vanBeek (Examiner)

Dr. K. Smillie (Chair)

Date: December 21, 1993

Abstract

This thesis explores a variation of the state space search problem, moving target search. The objective for the problem [illegible] placed in a search space and subjected to severe time constraints, is [illegible] target. This thesis presents new methods for doing moving target [illegible] algorithm, Forgetful Depth-First Search, adapts the well-known De[illegible] to this problem domain. Also, a search technique called Marking [illegible] general knowledge about the search space. These methods are discussed and compared with other known methods. Experimental results show that Forgetful Depth-First Search and Marking give good performance and are better than previous methods.

Acknowledgements

I would like to thank my supervisor Dr. Jonathan Schaeffer and my colleague Sean Halliday for their helpful discussions during this research.

Contents

List of Figures

Chapter 1

The Search Problem

The problem researched in this thesis is a variation of common search problems. The general graph search consists of a graph with a start node and a goal node. The entity doing the search is referred to as the problem solver ([illegible]). In the general case there may be multiple goal nodes, however this problem is restricted to a single goal. The objective is for the solver to travel from the start to the goal in as few moves as possible. The difference between the problem studied here and most other search settings is that the goal node, which we will refer to as the target ([illegible]), also moves.

1.1 Legal Moves

One constraint on the search problem is the movements that the entities are allowed to make. Some search settings allow the solver to move from any node to any previously generated node. Such searches are called off-line or internal searches. In the moving target search setting, the target and the solver are only allowed to move to nodes adjacent to their respective current nodes. The problem can be viewed as a search where the solver and the target are real-world objects, physically moving in a space. The solver may conduct its own internal off-line searches during its allowed computation time to determine its next moves.

1.2 Real-Time Constraint

If the solver is allowed to use whatever computing resources it desires to compute its moves, then little challenge is added by having a target that moves around. In this search setting the solver must make its decisions in a small constant amount of time. This is what makes the problem interesting.

1.3 Initial Knowledge

The moving target search problem is further defined by the specifications about what information is available to the problem solver and the target. At all times during the search, each entity knows the current position (node) of the other. Because the solver computes its moves in constant time, deciding whether or not to allow the solver to initially know the structure of the graph (nodes and adjacencies) is not a big issue. When the solver computes moves it will only have time to look at its current neighbors anyway. On the other hand, the initial knowledge of the target has no specified limit. In the experimentation in this thesis, problem solvers are tested against a number of targets with varying strategies, speeds and degrees of intelligence.

1.4 Search Setting

Although the search problem and the algorithms that are discussed are applicable to any graph, the research presented in this paper is restricted to grid type graphs with some percentage of the nodes blocked-out. Note that this differs slightly from another common technique in which walls are placed (edges removed) between adjacent nodes. In search spaces with a perimeter it is often very easy for the solver to corner the target. The graphs used here are toroidal which means that the nodes along the top row are connected to those along the bottom and similarly for the left and right

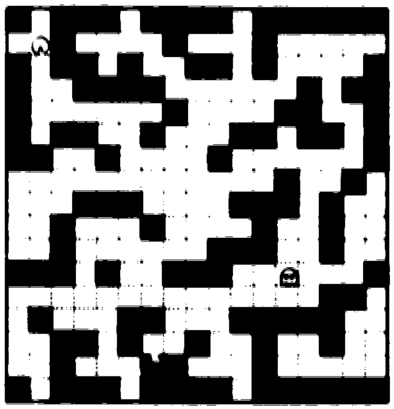


Figure 1: Search setting

columns. Figure 1 shows one such typical graph. Such graphs are commonly used for many search problems. They typically have high diameter which make the search challenging. Furthermore, this class of graphs allows for much variation.

Most of the graphs used in experimentation are randomly generated with some specified density which is the probability each node has of being blocked-out. At lower densities (below 25%) the search space is uninteresting. Moving through it is analogous to walking through a forest: although forced to make many side-steps, one strays little from the straight line from start to destination. However at higher densities (above 30%) the blocked-out nodes combine to form walls creating much more challenging obstacles and heuristic depressions. At even higher densities (beyond 35%) the search spaces tend to become disconnected. Interesting search spaces could also be produced using fewer nodes blocked-out if the blocked-out nodes are placed by hand.

1.5 Move Sequence

A minor detail concerning the search setting is the move sequence. The target and the problem solver alternate making moves. Another possibility would be to have the entities move concurrently. In other words, each would compute its next position,

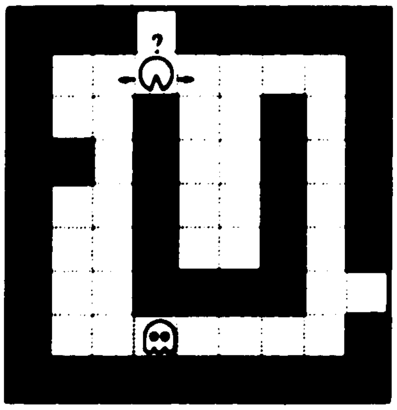


Figure 2: Critical decision

and then, simultaneously, each would move to its destination. However, it was found that using one method instead of the other makes almost no difference in the time required to complete the search.

1.6 Variance in Experimentation

One note concerning the experimentation is that there are large amounts of variance in the time needed to complete the search. One place where variance shows up is in the difficulty of search spaces. Two graphs with the same percentage of nodes randomly blocked-out may differ by an order of magnitude in the average time needed for the problem solver to reach the target. It all depends on which nodes get blocked-out during maze generation. Variance also occurs for different runs using the same search space. Sometimes when search parameters are varied slightly there is a large change in the search time. Such variance results from the high degree of sensitivity that can exist for individual moves. Figure 2 shows a typical example of a move decision with potentially high consequences. Furthermore, once two separate search runs begin to differ (one problem solver goes one way, the other chooses another path) the runs will not likely have any similarity from that point forward.

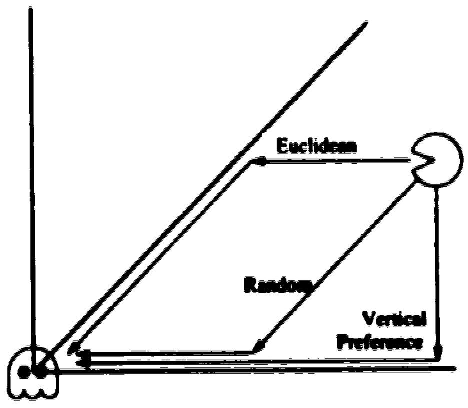


Figure 3: Various tie-breaking heuristics

1.7 Heuristics

There may or may not be heuristics to apply to the search. $h(a)$ is normally defined to be a predicted distance from the current node a to the target node. With a moving target, heuristic functions must be of the form $h(a, b)$ -the predicted distance between nodes a and b. The moving target search algorithms discussed in this thesis assume the existence of a reasonably good and admissible heuristic function. For the grid class of graphs previously mentioned, Manhattan distance is a common admissible function.

Often a secondary heuristic is used to decide which neighboring node to move to in the event of a tie. For instance, some of the algorithms discussed in this thesis use Euclidean distance as a secondary heuristic. Alternative options include making a random choice, giving precedence to either the vertical or horizontal move, or basing a decision on where the solver has been recently. To analyze effects that the different methods have, consider the space where the origin is the target's position (Figure 3). Using the Euclidean distance as the secondary heuristic, the problem solver will tend to travel to the 45 degree line of the quadrant it is in. Then it will travel along the 45 degree incline to the origin. This is good because it tends to maximize the number of "best" moves by allowing both horizontal and vertical movements in the direction

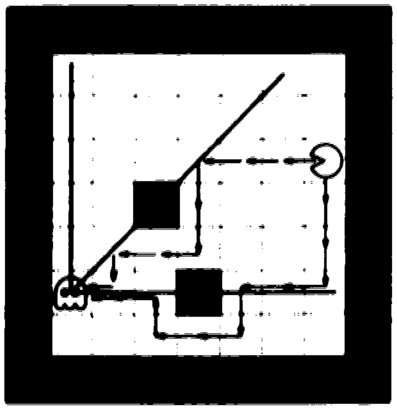


Figure 4: Disadvantage of approach along axis

of the target. Should the problem solver favor one direction over another given equal heuristic values, then the solver will travel directly to the axis and then travel along that axis to the target. Using random decision making, the solver will likely travel toward the respective axis along a 45 degree slope and then travel along that axis toward the problem solver. Figure 4 illustrates why it can easily be suboptimal to approach the target along one of its axes. To dodge a single node obstruction the solver moving along the axis of the target must make two extra moves. By approaching the target at an angle, the solver has an alternative move when it reaches the obstruction and therefore is still able to reach the target in the optimal number of moves. Simply stated, the Euclidean method maximizes the number of optimal paths to the target. Furthermore, "cornering" the target is easier if it has only 2 directions of movement that increase heuristic distances instead of 3 (Figure 5).

1.8 Target Catchable

Another premise concerning this search problem is that it is possible for the solver to catch the target. Certainly, this condition is satisfied with a slower moving target. In this case the "follow" technique discussed in Section 1.9 is generally the best approach. However the search problem is more interesting when the solver cannot depend on the

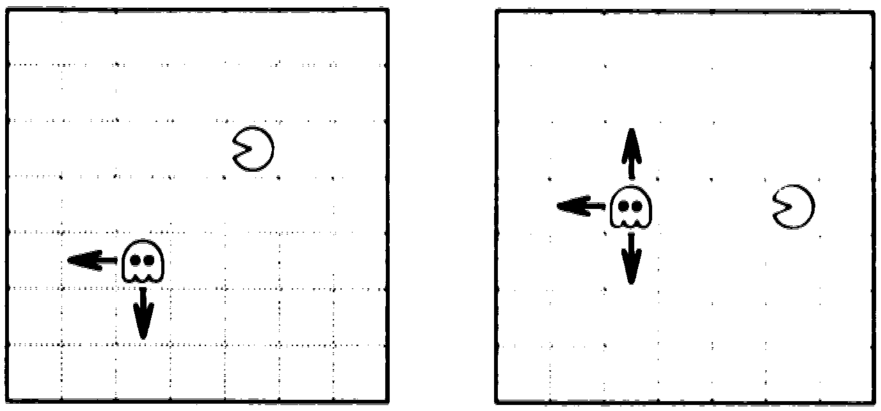


Figure 5: Target's escape routes

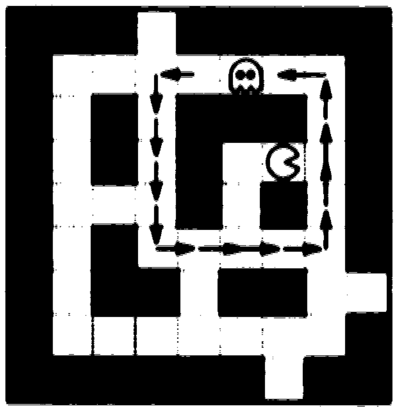


Figure 6: Cycling target

target's speed being slower than its own. Instead the target makes "mistakes", such as occasionally making moves "toward" the solver. The actual meaning of this is open to debate. One definition of a mistake is for the target to move one node closer to the solver according to the true distance of the search space. Another definition is that the target moves one node closer to the solver according to the solver's knowledge of the search space.

Figure 6 shows a situation where the target travels in an endless cycle. The solver may end up immediately behind the target, and at every step it will move toward the target along the counter-clockwise cycle. However, every step the target makes will be directly away from the solver and therefore it will never be caught. It is easy to see that the solver merely has to stand in the target's path to catch it. However, this requires the solver to anticipate where the target will be. In the real-time setting this demand is too great for the solver. Even though it can be argued that it is "possible" for the target to be caught, these situations are beyond the scope of the algorithms discussed in this thesis.

1.9 Follow Target's Trail

One simple approach for catching slow moving targets is based on keeping track of the target's movements. The problem solver first does a simple graph search to get on the target's trail and then follows this trail until it catches up to the target. The memory requirement for this method depends on how quickly the problem solver can pick up the target's trail, which is in $O(n)$ where n is the number of nodes in the search space. If the problem solver follows the trail exactly then almost no time is required to compute moves. Only a small constant amount of computing time is required to maintain the target's trail. Therefore this search algorithm satisfies the real-time constraint. Common search techniques can traverse a spanning tree of an n node graph in $2n$ moves. Using such an algorithm initially, the solver will take $2n$ moves, in the worst case, to find the target's trail. When the trail is found, the target will not have made more than $2n$ moves. Consequently, the length of the target's trail which the solver must follow is less than or equal to $2n$. The rate at which the solver catches up to the target depends on the speed difference between the entities. The solver can reach the target within $2nT_{miss}$ moves (where the target skips every T_{miss}'th move). From adding the time for both components of the search, the worst case time required to catch the target is $2n(1 + T_{miss})$ which is in $O(nT_{miss})$. It is important to note that this technique is designed to catch the "slower" moving target, not the target that occasionally makes "mistakes".

```
ReverseFloyd()
    for i = 1 to n
        for j = 1 to n
            H[i,j]=Heuristic(i,j)
    for k = 2 to n
        for i = 1 to n
            for j = 1 to n
                if H[i,j]==k
                    H[i,j]=min{H[i',j]+1}
                       over all neighbors i' of i
end
```

Figure 7: Reverse Floyd learning algorithm

1.10 Floyd's Algorithm

Another approach to doing moving target search is Floyd's algorithm [1]. This is an $O(n^3)$ algorithm for computing $h^*()$ which is the all pairs shortest path matrix of a graph. If the graph is known at the beginning of the search then Floyd's algorithm can be executed prior to moving the problem solver. If the structure of the graph is not known then the problem solver must first map the search space before executing Floyd's algorithm. After $h^*()$ has been computed, the problem solver just follows the direct path to the target. This method is guaranteed to find the target provided the target either moves slower or does not always move directly away from the problem solver. In situations where computation time is irrelevant this is an excellent approach to take. However, in real-time search this is too costly.

To take advantage of intermediate computations, Reverse Floyd (Figure 7), an alternative algorithm to Floyd's, can be used. Reverse Floyd builds up the heuristic function matrix (in our case starting from the Manhattan distance matrix). Its entries are increased until the matrix equals the all-pairs shortest path distances $h^*()$. Because the heuristic matrix monotonically increases toward the correct value, the intermediate computations are admissible (less than $h^*()$) and therefore useful to the problem solver. The Reverse Floyd learning technique can augment any search strategy. Furthermore, since the algorithm computes $h^*()$, it guarantees the problem solver can complete the search in n^3 time.

1.11 Algorithms

The two algorithms discussed, Floyd's and the "follow" technique, motivate issues in moving target search. Floyd's algorithm learns the maze so that later it can move optimally toward its destination. In contrast, the "follow" algorithm concentrates entirely on the target's moves. Traditional search algorithms, such as Depth-First Search [1], are based on where the solver has been. At every point in the search, the problem solver must decide which node to move to. To do this, each moving target search algorithm uses some combination of its knowledge about the search space, the target's current and previous positions, and its own current and previous positions.

A moving target search algorithm that learns the problem space must have a method of representing its knowledge. The all pairs shortest path matrix (produced by Floyd's or a similar algorithm) is perhaps the most complete and useful description of a graph that a searcher can have. However, in order to overcome the $O(n^3)$ bound and be useful in the real-time search setting, a solver that uses this knowledge representation must be able to compute and use only the portion of information that it needs. The method introduced by Korf and Ishida in the next section attempts to do this. Alternatively, other methods of learning the search space in an $O(n)$ representation are possible, as shown in Chapter 4.

1.12 Summary Of Thesis

In Chapter 2, this thesis reviews previous work on moving target search. This includes a paper by Korf and Ishida [6] which presents the problem and an algorithm for catching the target. A following paper by Ishida [7] makes improvements to this algorithm. Chapter 3 introduces a new technique called Forgetful Depth-First Search. This algorithm adapts the common Depth-First algorithm [1] to this problem domain. Marking, a new and efficient way of learning about a search space, is introduced in Chapter 4. Chapter 5 presents results of experiments measuring the performance and learning of the various algorithms. A summary and ideas for future work conclude this thesis in Chapter 6.

Chapter 2

Previous Work

2.1 Korf and Ishida '91

In the paper "Moving Target Search" [6], Korf and Ishida introduce the search problem examined in this thesis. In addition, they present an algorithm by which a problem solver can find the target.

2.1.1 Original Moving Target Search Algorithm

Their basic moving target search algorithm (BMTS) is an extension of the trivial "greedy" or "magnet" algorithm in which the problem solver always moves to an adjacent node with lowest heuristic value. The reason it does not get stuck forever in a "local minimum" (Figure 8) is that it modifies its heuristic information using simple deduction. The initial heuristic information must be admissible. When the solver is at a non-goal node s_1 ($h(s_1, t) > 0$) its true heuristic value (with respect to the target's node t) must be at least one greater than the least of its neighbors s_2. Therefore, when the actual heuristic does not reflect this, it is updated ($h(s_1, t) = h(s_2, t) + 1$). The solver does this each time it moves. Furthermore, the target's moves (from t_1 to t_2) are monitored. If the heuristic value changes by more than 1 ($h(s, t_1) - h(s, t_2) > 1$) a

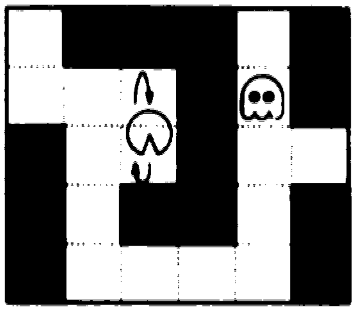


Figure 8: Local minimum (heuristic depression)

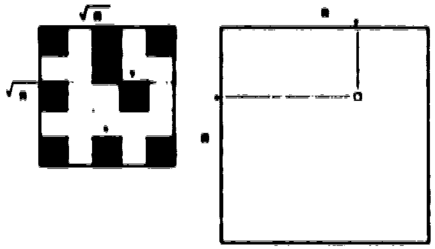


Figure 9: All pairs shortest path matrix

similar heuristic update is made. The problem solver builds an n by n matrix whose entries $h(x, y)$ represent a lower bound on the shortest path length between nodes x and y (Figure 9). This representation of knowledge is the same as that used by Floyd.

2.1.2 Heuristic Depressions

The BMTS problem solver has great difficulty with heuristic depressions. Figure 10 shows how the problem solver typically overcomes a misinformed heuristic situation such as a local minimum. The target is sitting still just on the other side of the wall at

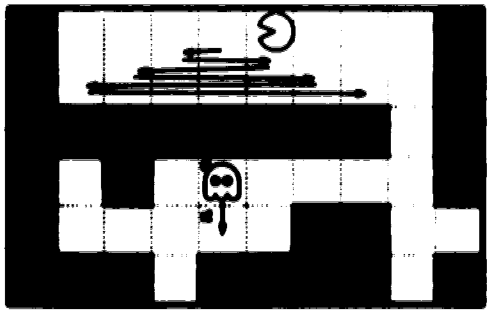


Figure 10: Adapting the heuristic function

node t_1. The problem solver will move back and forth along the other side of the wall increasing the heuristic function $h(X, t_1)$ for nodes X on the problem solver's side of the wall. The problem solver's amplitude increasing oscillations will eventually bring it to a node from where it can go around the wall. $O(k^2)$ moves are required to get out of a heuristic depression of k nodes. This inefficient "thrashing" behavior occurs because the solver increments heuristic information (usually by 1 or 2) one node at a time. Figure 10 also illustrates another problem. If just before the solver moves around the barrier the target moves from node t_1 to t_2, then the solver will have to repeat the learning that previously took place. This second problem is characterized as "loss of information" due to target's movement. In a follow up paper, discussed later, Ishida addresses these problems [7].

2.1.3 Tie Breaking

One issue not given much attention in the paper is how to break ties. As discussed in a previous section, a second heuristic could be used. For the particular search graphs used, the Euclidean distance is one suggestion. However, Korf and Ishida use a random choice in this event.

Experiments were done, for this thesis, to measure Euclidean and random secondary heuristics. Comparisons of the tables in Figure 11 show the difference in search time performance of the two tie breakers. These experiments used 100 by 100 search spaces where each node has a 35 percent chance of being blocked-out. 500 trials were done for each set of parameters. Two types of targets were used for testing. One of them moves completely randomly. The other tries to avoid the solver by giving higher probability to nodes farther away from solver as it makes its random moves. The basic moving target search algorithm of Korf and Ishida (BMTS) was tested as well as the same algorithm using commitment (CMTS), as discussed in Section 2.2.1 using a degree of commitment of 10. The experiments measured

RANDOM	Target	mean	median	min	max	num over
BMTS	random	10245	8965	248	>20000	128
	avoid	9435	7476	187	>20000	100
CMTS	random	2098	907	123	>20000	6
	avoid	2153	1184	145	>20000	1
EUCLID	**Target**	**mean**	**median**	**min**	**max**	**num over**
BMTS	random	13297	15281	350	>20000	199
	avoid	11943	12521	305	>20000	157
CMTS	random	3615	1321	133	>20000	17
	avoid	3124	1511	143	>20000	21

Figure 11: Using Euclidean as secondary heuristic

the number of moves the solver needed to catch the target. For each configuration, the mean, minimum, and maximum of the 500 trials is shown in Figure 11. The experiments do confirm that using random as a tie-breaker is a better choice than the secondary heuristic. Searches that required more than 20000 moves were halted and recorded as 20000. Consequently, the mean averages for configurations that had some long searches are deflated. This applies more to the searches using the Euclidean tie breaker. Therefore the true disparity between the two tie-breaking techniques is actually more pronounced.

The problem with using the Euclidean distance as a second heuristic function is that it keeps the solver in heuristic depressions much longer. Figure 12 is a snapshot of a search showing the solver choosing to move back down to the bottom of a heuristic depression for the second time when in fact the alternative move (with the same heuristic value) would have been better. With a random choice the solver may have made the correct move toward the exit of the depression.

A potential improvement to the random tie breaking currently used would be to give priority to the node that the solver has not been at most recently. This might discourage "local" thrashing.

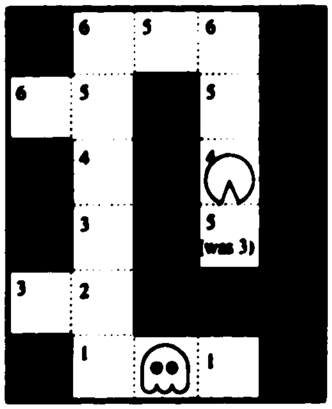


Figure 12: Using Euclidean as secondary heuristic

2.1.4 Analysis

BMTS is complete (the target will eventually be caught) under the assumption that the target makes "mistakes". The solver keeps increasing its heuristic function (matrix) which, of course, cannot increase past the all-pairs shortest path matrix. The sum of all n^2 entries in this matrix cannot exceed n^3 because each entry is bound above by the true minimal distance between its nodes which must be less than n. Learning also depends on the rate in which the target makes "mistakes". The worst case is $O(n^3 T_{err})$ where T_{err} is the period of error (number of moves per mistake that the target makes).

BMTS differs from Floyd's [1] because it concentrates its learning on correcting the heuristic values that pertain to the current positions of the target and the problem solver. In practice most of the heuristic values are never used. Experimentation shows that actual search times are significantly better than the worst case.

2.1.5 Completeness Questioned

The proof of completeness for BMTS relies on the assumption that occasionally the target moves in such a way that the heuristic distance between the problem solver and the target does not increase. Given this, it seems reasonable for one to conclude that this criterion would be satisfied if they use a target that, instead of occasionally skipping moves, would move semi-randomly or occasionally make moves that common sense dictates are "toward" the problem solver. However this is not so. In other words, the target can move in such a way that reduces both the Manhattan distance (initial heuristic) and the shortest path distance (optimal heuristic h*()) but, according to the problem solver's current heuristic information, the heuristic disparity increases. Although this sounds possible, it seems likely that a target's random move should have a good if not equal chance of decreasing heuristic disparity for any given heuristic information. Unfortunately this is not the case. Experimentation discovered a situation in which BMTS builds up its heuristic information such that any legal move to an adjacent node that the target makes will increase the heuristic disparity. Furthermore this is not an unreasonable or isolated setup.

Recall that in our (and their) experimentation a grid of nodes was used which is a 2-colorable graph (checkerboard). Without loss of generality, assume that when the target is on a red square the problem solver makes a move from red to blue, and likewise for blue moves to red. Consequently, the target will never be on a blue square when the problem solver is making a move from red to blue. Therefore the problem solver is never given the opportunity to increase the heuristic distance between a red and a blue as an outcome of one of its movements. These updates can be made as a result of a target's movement, but that is not sufficient.

Figure 13 shows a situation where the solver is never able to work its way out of a heuristic depression. The target is programmed to move randomly but only stay within the 3 by 2 area. Even after a long time the problem solver never moves out of the heuristic depression marked by the rectangle. The solver spends most of its time toward the left side of the depression. In general, the size of a heuristic depression that a solver can tackle does depend on the size of the area that the target moves about.

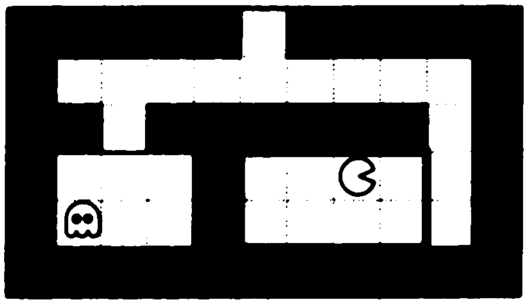


Figure 13: Stuck

Note that this problem only applies to bipartite search spaces (no odd cycles). Possible remedies include using a slower target (in which case just following the target is a better algorithm) or not using this class of search spaces (use a hexagonal or triangular tessellation instead). Fortunately, the commitment enhancement introduced by Ishida (next section) will restore completeness to searches with random targets that do not skip moves.

2.2 Ishida '92

In "Moving Target Search With Intelligence" [7], Ishida presents improvements to his and Korf's original algorithm BMTS. Ishida introduces commitment and deliberation to reduce the "information loss" and "thrashing" problems discussed in the previous section.

2.2.1 Commitment

In Korf and Ishida's original algorithm the solver learns with respect to the target's current position. If the target didn't move then the solver could concentrate its learning and get out of heuristic depressions sooner. Ishida achieves this effect using commitment. The solver ignores some of the target's moves, and instead concentrates on a "goal" node which is occasionally updated to the target's current position.

When to Update Goal

The main issue that arises by adopting this strategy is deciding when the goal node should be updated. Obviously, when the solver reaches the goal node it should be updated to the target's current position. Ishida points out that by itself, this rule for updating the goal requires too much commitment. If the solver makes a sufficient number of consecutive moves toward the goal node then it should be updated. This parameter (the number of moves toward the goal) is called the degree of commitment.

How Much Commitment

Now the question becomes how large the degree of commitment should be. The ideal value for the degree of commitment will depend on the search space. By experimentation on 100 by 100 search spaces with varying densities, Ishida finds 10 to be a good value.

Performance

In easy search spaces, where the solver is not likely to get caught in heuristic depressions, the solver is better off chasing the target directly instead of using commitment. For the more difficult search spaces (those with 35 % of the edges removed), the performance improves by 5 to 10 times over the original algorithm. This is consistent with experimentation done for this thesis shown later in Section 5.3.

Commitment is an effective method of concentrating the solver's learning to get the solver to the target sooner.

2.2.2 Deliberation

Commitment eases the problem of lost learning due to the target's movements. However, the problem solver still "thrashes" inside depressions as it incrementally updates its heuristic information. Ishida introduces a mechanism called deliberation in which the solver switches to off-line search to find its way out of heuristic depressions sooner.

In off-line search the solver doesn't actually move but instead expands nodes in its internal memory. One advantage is that this search is not restricted to moving only to adjacent nodes. To maintain the real-time constraint, the solver can only expand one node in off-line search per regular move. The off-line search is similar to an A^* search [1].

When To Deliberate

The solver switches from on-line search to off-line search as soon as it detects that it is in a heuristic depression, i.e. all neighboring nodes have higher heuristic value. For the off-line search, the solver maintains two lists, a closed list and an open list. Initially the closed list is empty and the open list contains the starting node of the search. For each move the best node in the open list is expanded and moved to the closed list. When a node is expanded all of its neighbors that have not previously been added to the open list are added to the open list. The solver expands its search outward from its beginning node until it finds its way out of the depression or it has exceeded its allowable time to do the off-line search. The solver knows it has found its way out of a depression when it expands a node on the open list x and it finds a previously unseen neighboring node x' with lower heuristic value, $h(x') < h(x)$. At this point, the solver stops deliberating, moves out of the depression, and resumes regular search. The limit placed on how much off-line search the solver is allowed to do (called the degree of deliberation) is the maximum number of nodes that the solver can expand during an off-line search.

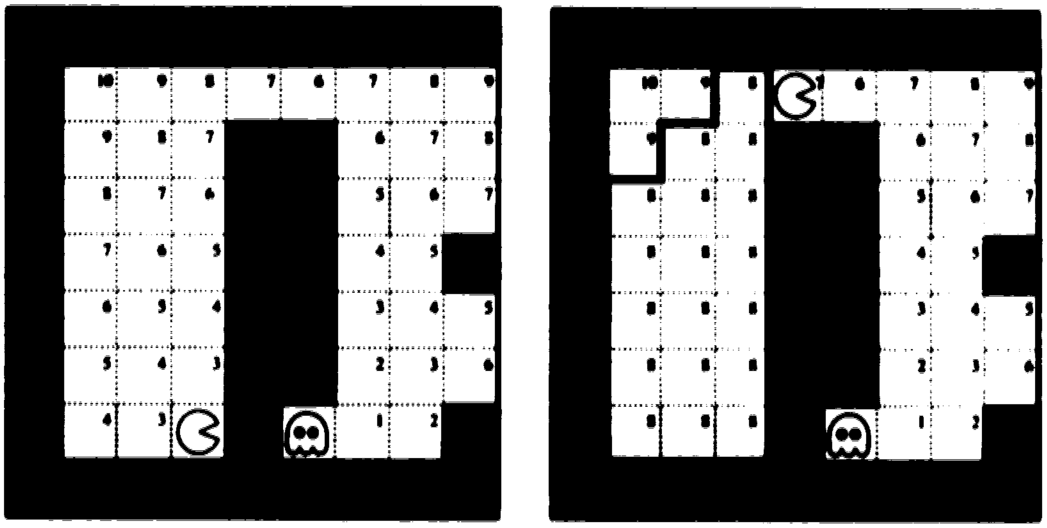


Figure 14: Deliberation

Better Learning

After doing the off-line search the solver updates the heuristic value of all of the nodes on the closed list x_{closed} to be one greater than the last node x expanded on open: $h(x_{closed}, y) = h(x, y) + 1$. The learning here is faster because some heuristic values are being increased by more than just 1 or 2.

Figure 14 illustrates the problem solver's use of deliberation. The solver works its way out of a heuristic depression of 18 nodes. The solver makes almost as many off-line moves to find its way out of the depression. At the end of the off-line search the solver is positioned at the same place it was when it started the off-line search, however it now has a destination. Some interpretations of the moving target search problem may allow the solver to jump from the bottom of the depression to the exit since it is still moving less than one node per turn on average. If the problem solver is bound by the rule that is can only physically move one node per turn then it will need another 7 moves to move to the edge of the depression. This direct path from the bottom of the heuristic depression to the exit does not have to be computed (which

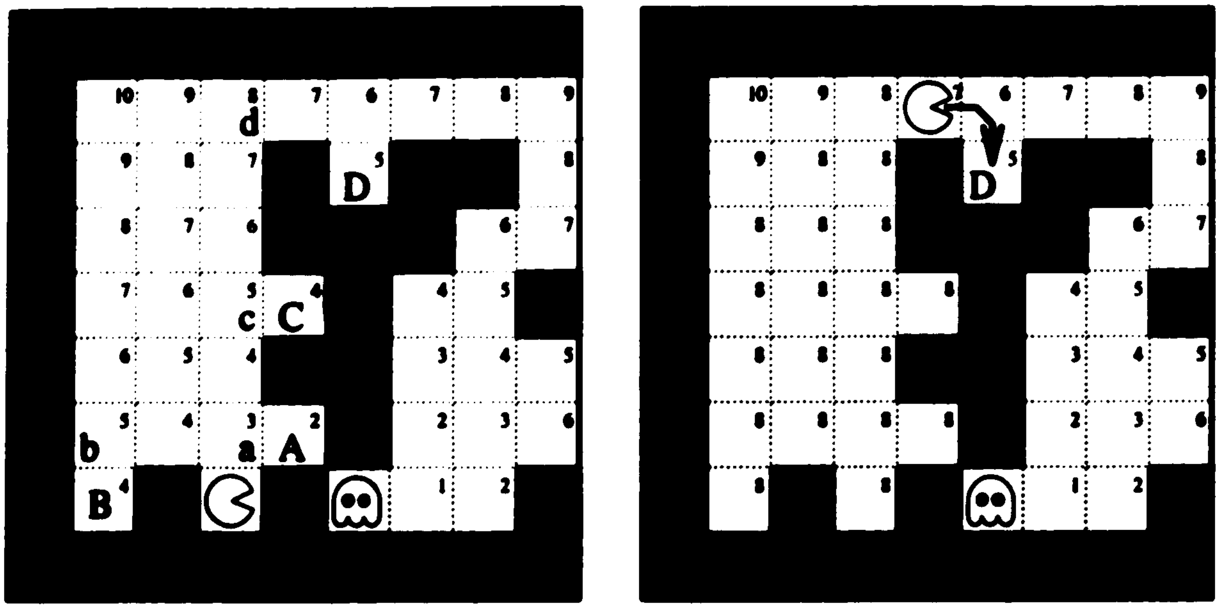


Figure 15: Deliberation in a non-ideal setting

would violate the real-time constraint). If, while doing the off-line search, the solver maintains information about the "parent" of each node whenever it adds a node to its open list, then the ancestral path from the exit node up to the bottom-of-the-depression node in the solver's off-line search tree gives what will be (in most cases) a direct path (in reverse order) out of the depression. Using deliberation the solver gets out of the depression in 25 moves. Note that there is a total increase of 46 in the heuristic information (sum of all new heuristic values minus sum of all old heuristic values).

Figure 15 illustrates the problem solver's use of deliberation in a more "random" search setting. The heuristic depression is not a nice square room. In this Figure the solver starts its off-line search from the bottom of the heuristic depression. Later when it finds node *a* it will stop its off-line search since *a* has a neighbor, namely node *A*, that has a lower heuristic value and is not on the solver's closed list. However when it later moves to node *A* it will have reached another depression and will again perform off-line search. Similarly the solver will stop off-line searching each time when it reaches nodes *b*, *c*, and *d* only to restart (and redo) the off-line search immediately after when it reaches nodes B, C, and D. The second part of Figure 15 shows the effect of ending off-line search too soon. The problem solver has finished all of its off-line searches up to node d. Node d, of heuristic value 8, has a new neighbor with lower heuristic so the solver stops its off-line search. When the solver reaches node D it will start a new off-line search. For the solver to find its way out of this big heuristic depression (around the upper right hand corner), it will have to do an off-line search of almost every node in the depression even though it has already "done" most of these nodes many times. All nodes with heuristic value of 8 or less will be searched before the true exit node (of heuristic value 9) is searched. Another cause of this same problem occurs when the degree of deliberation is less than the size of a heuristic depression. Again the solver stops the off-line search prematurely throwing away useful information. Certainly deliberation performs well on "small" and "regular" depressions. However, as the search space increases in difficulty, deliberation loses its advantage.

How Much Deliberation

How much deliberation should the solver do? The ideal value for the degree of deliberation will depend on the search space. For easy search spaces, more deliberation is inappropriate since it allows the target to get away as the solver remains stationary. In Ishida's experiments, using difficult search spaces, the problem solver performed better when it was allowed to expand up to 25 nodes off-line instead of just 5 nodes.

Performance

Deliberation in combination with commitment was added to the moving target search algorithm. Ishida shows that in easy search spaces using deliberation can be inappropriate. A problem solver chasing an avoiding target takes longer by using deliberation. In difficult search spaces the performance doubled in comparison to using commit-

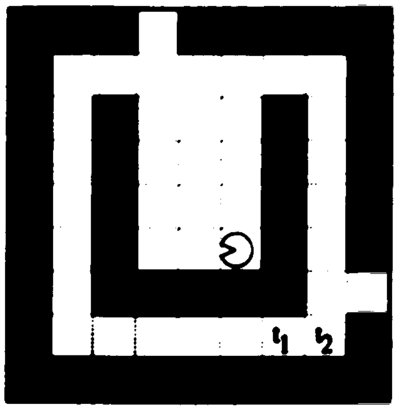


Figure 16: Learning lost

ment alone. With respect to the original algorithm the improvement is between 10 and 20 times.

Deliberation effectively reduces the thrashing behavior of the solver and results in more efficient searches.

2.2.3 Overall Learning

Commitment and deliberation significantly improve the moving target search algorithm. However even with these revisions, after the problem solver's immediate goal is satisfied, the learning that has taken place is not likely to be beneficial again. For example, consider a long grueling search where the problem solver learns its way out of a heuristic depression but later finds itself in the same place (see Figure 16). If at this time the target is even one node away (node t_2) from where it was during the first time that this happened (node t_1), the problem solver will have to learn its way out of this heuristic depression once again. In short, learning seems to get lost in the huge n by n heuristic matrix representing the problem solver's knowledge. The essential reasoning process of Ishida's revised problem solver remains the same as its predecessor. Knowledge is still represented by pairs of nodes. Ishida's improvements are in the efficiency of the algorithm. Learning is concentrated and localized, not generalized.

2.3 Other Related Work

Ideas for the works of Korf and Ishida [6] and Ishida [7] came from a previous paper by Korf [5] in which Korf solved other problems in real-time by adapting the heuristic function.

There are other similar works that deal with problem solving in time restricted and uncertain environments. Pollack and Ringuette [9] introduce an environment called Tileworld for testing problem solvers. Tileworld evaluates how efficiently a problem solver balances its time between "thinking" and "working". The Tileworld problem is different than the moving target search problem, and unfortunately algorithms for one are incompatible with the other.

Ishida's ideas of commitment and deliberation have been used in other problem domains. Kinney and Georgeff [4] experiment with commitment in the Tileworld [9] setting. The notion of commitment and amount of commitment is discussed extensively in Cohen and Levesque [2].

Chapter 3

Forgetful Depth-First Search

This thesis presents an algorithm with a radically different paradigm for searching for the moving target. The objective is to capture the way the Depth-First Search algorithm [1] (Figure 17) efficiently searches areas and quickly gets out of heuristic depressions. Figure 18 shows the path the Depth-First Search algorithm produces. At every step, the solver moves onto a node that it hasn't previously visited if there is one. If there is a choice of more than one new node, then normally a heuristic function is used to break the tie. When the solver is surrounded by old nodes, it backtracks.

```
DepthFirstSearch(Node n)
        ForAll new neighbors m of n
                move from n to m
                DepthFirstSearch(m)
                move from m to n
end
```

Figure 17: Depth-First Search algorithm

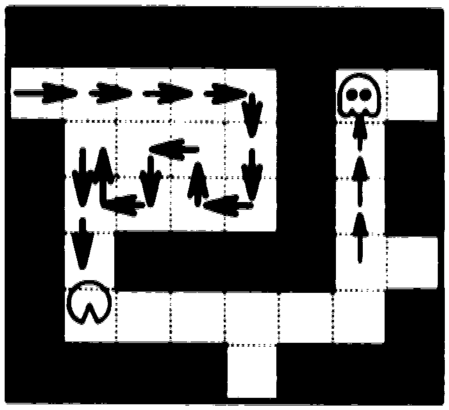


Figure 18: Avoid previously visited nodes

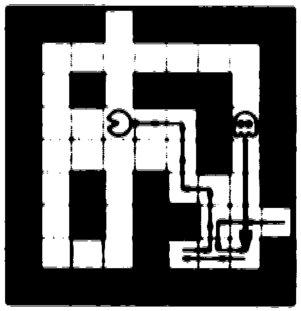


Figure 19: Target wanders onto searched nodes

3.1 Problems with Traditional DFS

Unfortunately, using normal Depth-First Search to hunt a moving target would not be an effective strategy. One problem is that old information becomes invalid. Nodes that have already been searched (those on the solver's closed list) are supposed to be places where the target is not. However this is not true; the target may move onto these nodes after they are searched (Figure 19).

Another problem is that the closed list constructed by the problem solver inhibits movement. This could block efficient paths to the target (Figure 20).

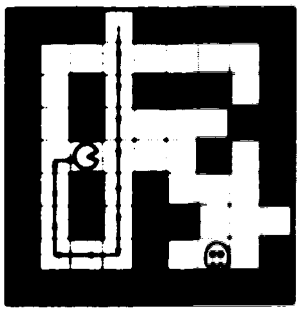


Figure 20: Closed list inhibits movement

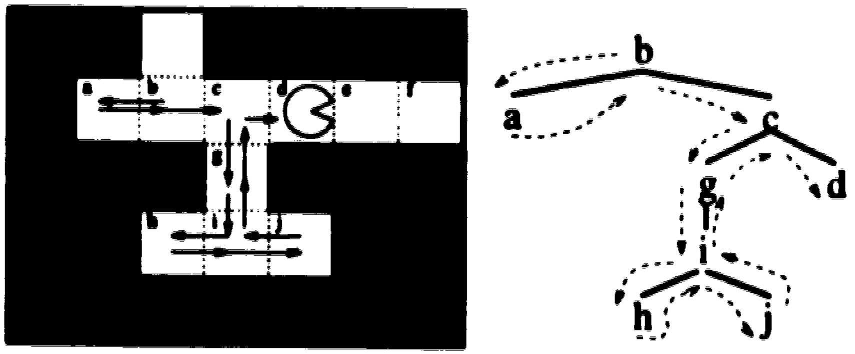
3.2 Forgetful Algorithm

These setbacks are overcome using an algorithm called Forgetful Depth-First Search (FDFS). The solver has limited memory and cannot support a continually growing closed list. When the solver moves onto a node it will add it to its data structure. This means that some previous piece of information must go. The victim is the oldest piece of information, which is the root of the search tree. The new root of the search tree is the first child of the previous root. This pruning occurs once for each move the solver makes whether advancing onto a fresh node or backtracking over old ones. Figures 21 through 25 show the problem solver's path and the changes in the DFS search tree as the search progresses.

Variations to DFS have been done before. Examples include iterative deepening [10] and iterative broadening [3]. These techniques are used to alter node ordering for searching large trees. Like these examples, FDFS attempts to adapt DFS to a different problem domain -moving target search.

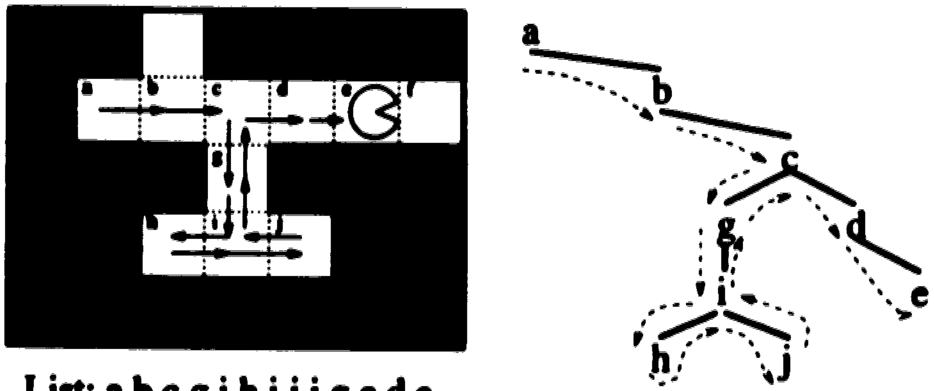
3.3 Implementation

For FDFS to be an acceptable candidate for doing moving target search, the transformation (root pruning) of the tree data structure must be achievable in real-time for arbitrary tree size. Fortunately there is a simple implementation. The solver keeps



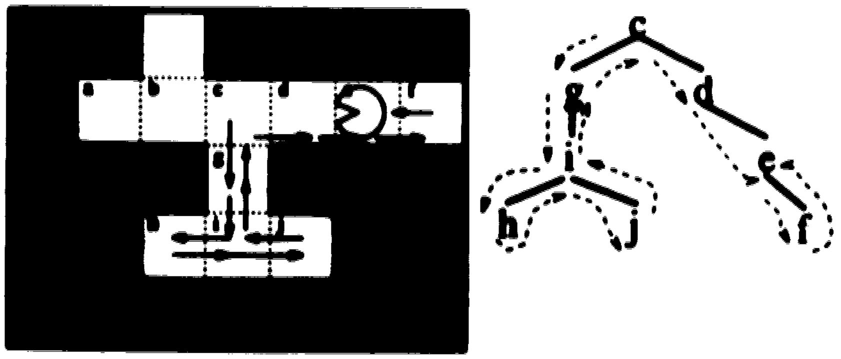
List: b a b c g i h i j i g c d

Figure 21: Problem solver's path and implicit DFS tree



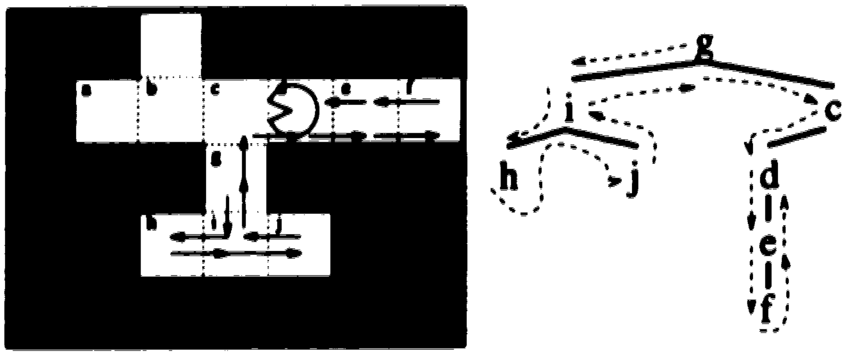
List: a b c g i h i j i g c d e

Figure 22: Same search 1 move later



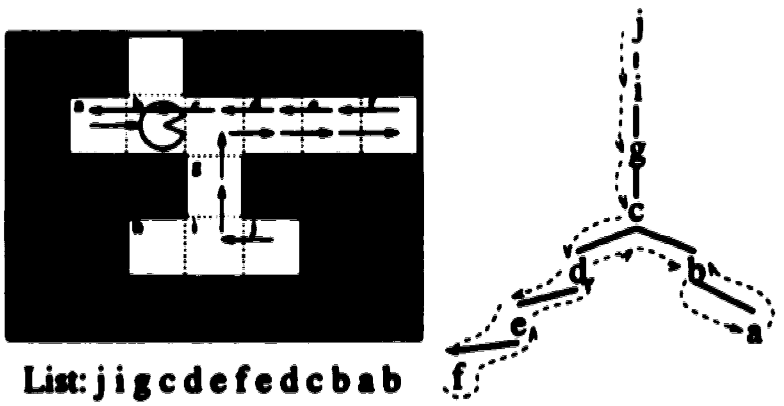
List: c g i h i j i g c d e f e

Figure 23: Same search at a total of 3 moves later



List: g i h i j i g c d e f e d

Figure 24: Same search at a total of 4 moves later



List: j i g c d e f e d c b a b

Figure 25: Same search at a total of 8 moves later

track of a list of visited nodes that allows duplicates. Whenever the solver moves it adds the new node onto the front of the list and removes the oldest node at the back of the list. When the solver is looking at its neighboring nodes to determine which to move to, it selects one that is not currently on its list if possible. If the problem solver cannot move to a node not on its list, it moves to the predecessor of the oldest occurrence on the list of the node it is currently at. This mechanism for deciding which node to move to, along with maintaining the list of the last few nodes, achieves the desired Forgetful Depth-First Search strategy. Figures 21 through 25 also show the list and how it represents the implicit DFS tree.

To ensure that the real-time constraint is not violated, the problem solver has to be able to quickly determine if a given node is in its history list. This can be achieved by having a flag for every node in the search space. The flag is set if the node is in the recent history of the solver. Information also stored for each node includes references to each occurrence of the node in the solver's history list. The number of occurrences of each node is limited by the branching factor of the search space. This extra information is needed so that the problem solver knows if it should reset a node's flag when an occurrence of the node is removed from the end of the history list. Furthermore, during backtracking the solver has to move to the predecessor of the oldest occurrence of the current node. By maintaining a flag and the references for each node, the problem solver does not have to search its history list to see if it has recently visited a node or to determine where to backtrack. Therefore the problem solver is able to maintain its data structures and make its decisions in a constant amount of time with respect to the size of the search space as well as the amount of history it wishes to maintain.

Alternative implementations of FDFS are possible. One variation is to not have duplicates in the history list. It is still possible to have the solver give priority to new nodes and backtrack the same way as previously discussed. This implementation

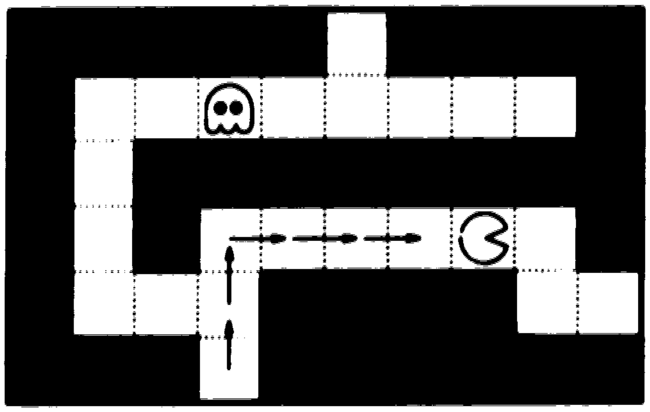


Figure 26: Unrecoverable bad decision

method modifies the algorithm slightly. Since pruning does not occur during backtracking the solver has a memory advantage in that using the same length history it will remember more nodes than the previously discussed implementation. However, this provides little advantage since the length of the history can be arbitrarily increased without affecting the time required by the problem solver to execute its moves.

3.4 Maneuverability and Recoverability

One concern with this method is that it might take the problem solver a long time to recover from an "incorrect decision" because it is biased against moving to recently visited nodes. Figure 26 illustrates this idea. The problem solver has little room to maneuver in search spaces with long narrow hallways. This problem is not as apparent in more open search spaces.

3.5 Target Trespasses on Solver's History

One special case to consider is when the target wanders onto any of the nodes that appear in the solver's list of nodes it has visited recently. Once again a situation arises that is similar to the one shown back in Figure 19. The problem solver must

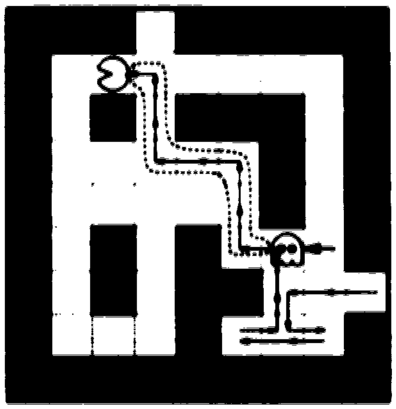


Figure 27: Target wanders onto solvers history

correct its current information.

3.5.1 Remove Minimal History

One method of dealing with the target wandering onto the recent history of the solver, is to invalidate the node the target is on. In other words the most recent occurrence of this node plus all prior nodes are removed from the solver's list. As much information as possible (all nodes more recent than the target's position) is salvaged in the solver's search tree. Now if the problem solver is chasing the target by following its trail, the problem solver will not be bumped off the trail if the target loops around onto nodes recently visited by the problem solver. Another comforting consequence is that the length of the problem solver's list tends to be proportional to the distance between it and the target.

3.5.2 Remove All History

In the previous method for dealing with the target moving onto the solver's recent past, the solver preserves the part of its list more current than the target's node. However, these nodes now represent a path (not necessarily the best path) from the solver to the target (Figure 27). Recall that the solver avoids nodes in this list. Consequently, the problem solver may have problems trying to get to the target.

minimal	full	neither
0.6	2.1	97.3

Figure 28: Comparison of history removal mechanisms

This is worse for dense search spaces (those with narrow hallways and little room to maneuver). Ironically, one possible counter-strategy for the target would be to closely follow the problem solver. An alternative way to deal with the situation where the target wonders onto nodes in the solver's list is to empty the list completely. This gives the solver the freedom to move over those nodes between it and the target.

3.5.3 Comparison

Removing all the history prevents the target from sneaking around behind the solver. Compared to removing the minimum amount of history, the difference is certainly noticeable with a user controlling the target's movements. However, the evasive strategy of computer target algorithms is far from intelligent. During experimentation, switching between history correction mechanisms made almost no difference. The search parameters included using 100 by 100 search spaces with 35 percent of the nodes blocked-out. The target was programmed to move randomly giving much higher probability to nodes farther away from the solver. The length of the solver's list of visited nodes ranged from 25 to 300. Using long list length maximums would show more differences between the two pruning mechanisms, however the range chosen results in better search times (see next section). A total of 330 searches were simulated. In general, the number of moves needed to complete these searches is in the order of a few hundred. The results are shown in Figure 28. For 2.1 percent of the scenarios the solver reached the target sooner by purging the entire list. Removing the minimum history proved to be better 0.6 percent of the time. Overall, for more than 97 percent of the runs it didn't matter which type of pruning was done. Varying other parameters, such as search space size and density, yielded similar results.

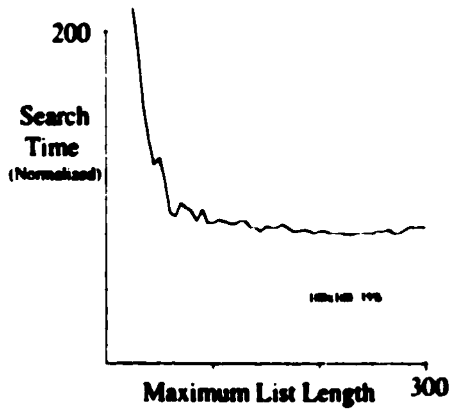


Figure 29: Search times for varying list maximums 100x100

3.6 Amount of Memory

An issue that arises with respect to using FDFS is how much memory should the solver have. That is, how long should the list be? The ideal list length will vary with the target's behavior and with the search space. At one extreme, with a zero length list, the solver easily traps itself. At the other extreme the solver only prunes nodes off its list when the target causes it. Thus, with a stationary target the solver's strategy is the same as normal Depth-First Search, which is the suggested approach for these circumstances. Perhaps pruning the list only when the target causes it may in fact be a good approach.

Experimentation was done to determine a good maximum length for the solver's list. Values from 5 to 500 (increasing by 5) were tested. For each length the solver was tested on 100 randomly generated search spaces. Due to the variance in difficulty of the search spaces, the results for each search space were normalized so that the average number of moves over all searches (5 to 500) is 100. Figures 29 shows the average of the normalized curves. The problem solver is hunting a randomly moving target in a 100 by 100 search space where each node has a 35 percent probability of being blocked. The data shows that the search time levels out when the maximum list length equals the dimension of the search space (100). Therefore this was chosen as the default length for FDFS.

Chapter 4

Marking

Any real-time learning technique strictly based on using and updating the heuristic function falls victim to the huge n^2 representation. Algorithms such as Forgetful Depth-First Search may have an excellent strategy for hunting down the target, however no learning occurs. Consequently, the performance of the solver does not improve in the long term. For example, if during a search the FDFS problem solver handles a situation poorly and later finds itself in a similar situation then it will likely make the same mistakes. The Marking mechanism introduced in this section attempts to provide the problem solver with the ability to learn efficiently and to maintain that knowledge in a higher level representation than the traditional all-pairs shortest path matrix.

4.1 Flat Marking

The objective of Marking is to factor the search space into sections so that during the decision process the solver can deal with collections of nodes instead of individual nodes. In particular, the problem solver can assign marking i to a node if it is useless for the problem solver to move to that node unless the target is at a node with the same marking i.

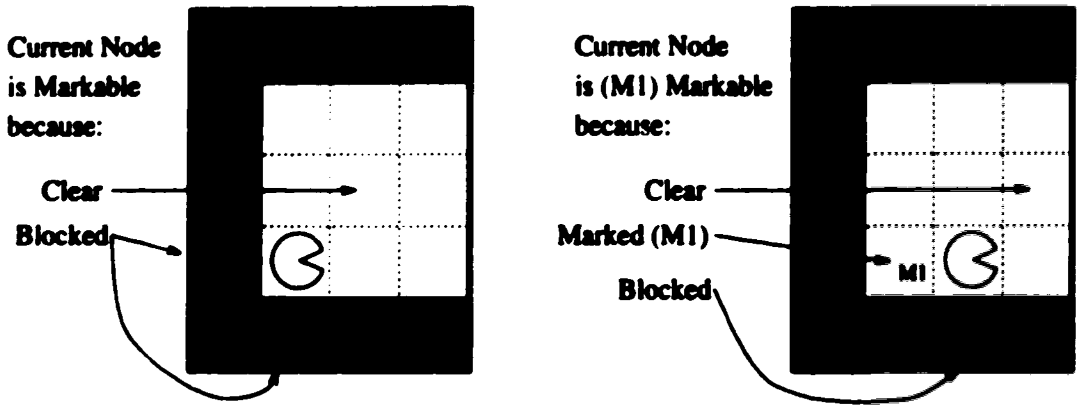

Current Node is Markable because:
Clear
Blocked
Current Node is (M1) Markable because:
Clear
Marked (M1)
Blocked
M1

Figure 30: Marking nodes

As stated, the prerequisite to marking a node is that the solver's ability to track the target is not inhibited by doing so. Formally, a node can be marked if for every pair of unmarked neighbors there is another unmarked node in the neighborhood of both of the neighbors. For grid-type graphs, Figure 30 shows examples when nodes can be marked. A unique marking number is assigned to a markable node when none of its neighbors are marked. The corner node is marked because one can travel from one of its neighbors to the other in two moves without going through the corner node. When marking a node with a marked neighbor normally the same marking value is used. The next node (to the right of the corner node) can now be assigned (with the same marking as the corner node) because its unmarked neighbors have an alternate shortest path between them. Figure 31 shows how a search space would typically be marked after a problem solver explored it.

4.2 Hierarchical Marking

Unfortunately there are some intuitive cases which this flat marking system cannot handle. Figure 32 shows an example where a long "dead-end" hallway cannot be conveniently designated as a region to avoid if the target is not within. The problem is that the two nodes adjacent to the intersection have different markings. The

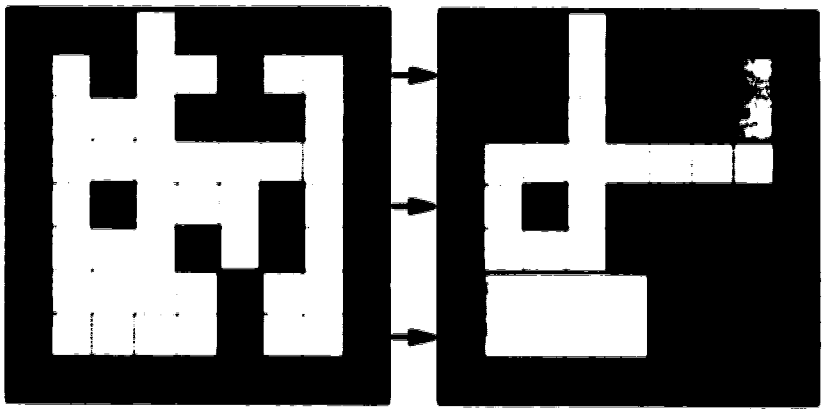


Figure 31: Search space marked into regions

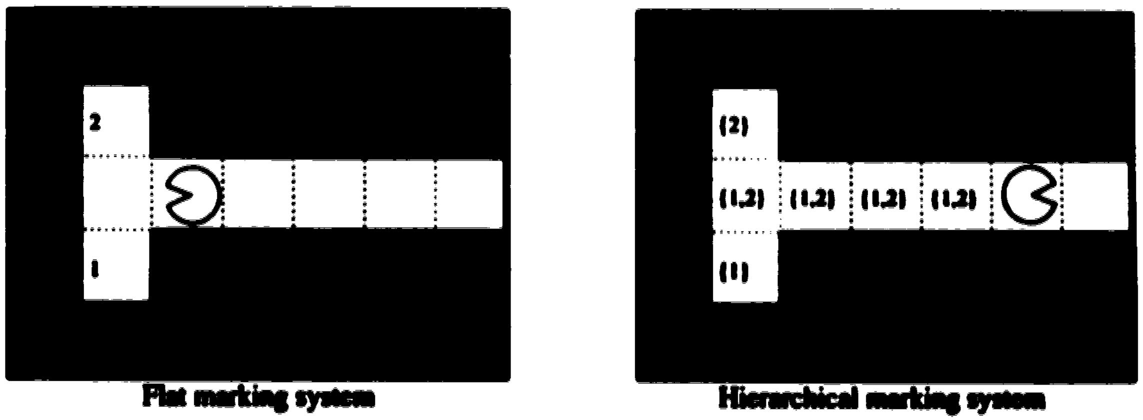


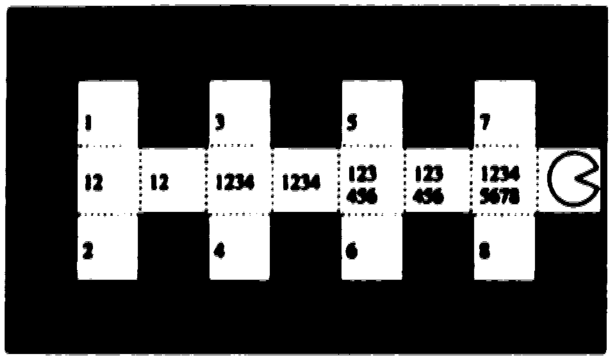
Figure 32: Multiple markings

solution, shown in the second part of Figure 32, is to allow a node to be marked with combinations of its neighbors' markings. In other words, each node has a set of markings, therefore allowing more information to be represented. Unmarked nodes are best thought of as being marked with all markings.

Initially each node n's marking set $M(n)$ is the set of all things (complement of the empty set).

$$M(n) = \bar{\emptyset}$$

$N(n)$ is the neighborhood of (set of adjacent nodes to) node n. A node n can define a new region when it is not on the only shortest path between any other two nodes. The marking set becomes a singleton set containing a unique number (for node n use



Node Multi-Markings

Figure 33: Too many markings

number n). If:

$$(\forall a) a \in N(n)(\forall b) b \in N(n)(\exists c) c \neq n \wedge c \in N(a) \wedge c \in N(b) \wedge (M(a) \cup M(b)) \subseteq M(c)$$

Then:

$$M(n) = \{n\}$$

Otherwise a node's marking set may be reduced based on the marking sets of its neighbors.

$$M(n) = \{m \mid \exists a \in N(n) \exists b \in N(n) \wedge a \neq b \wedge m \in (M(a) \cap M(b)) \wedge (\forall c) c \neq n \wedge m \notin M(c)\}$$

Note that $M(n)$ cannot be empty after applying this rule since that could only happen if the antecedent of the first rule was true in which case $M(n)$ is assigned to be $\{n\}$.

There is a potential danger of violating the real-time constraint of the search since a single node can have many markings. Figure 33 shows an exceptional case where nodes can have a large number of markings. Comparisons of large sets is not a constant operation. Fortunately the size of such multiple markings are typically small even for large spaces. In thousands of searches on graphs with 10000 nodes there have been no occurrences of individual nodes having more than 64 markings. Even so, the real-time constraint can be guaranteed if the number of markings per node is limited by a constant.

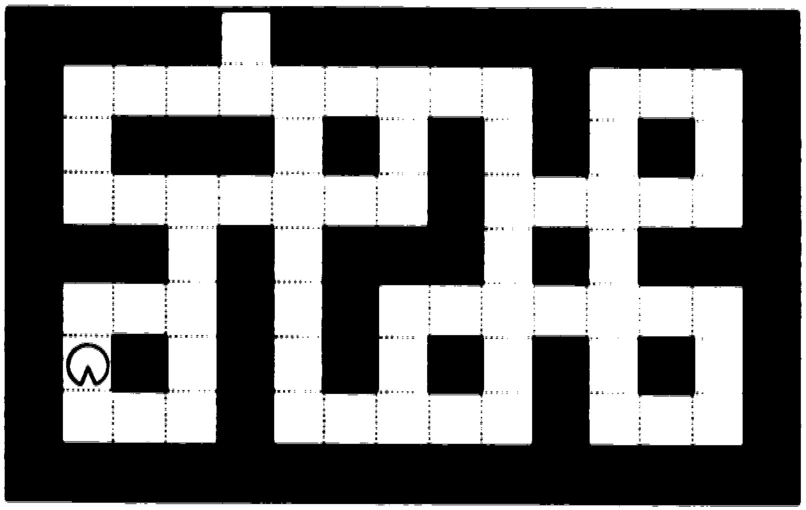


Figure 34: Unmarkable search space

4.3 Marking Limitations

Marking unfortunately has its limitations. Figure 34 shows a search space for which this strategy is helpless. The graph is two-connected and every 2 by 2 collection of nodes has at least one of them blocked-out. Consequently the problem solver cannot mark anything. On the other hand, simple human intuition would easily divide the space into appropriate sections. A more advanced mechanism that can look at more than just one node's neighbors is needed to do marking of this nature.

4.4 Using Markings

The problem solver uses information produced by the Marking technique as it decides where to move. It is useless for the solver to move onto a neighboring node that is marked unless the target's node is marked similarly under the flat Marking system. When the hierarchical Marking system is used, the solver should not move onto a node unless its markings are a superset of the target's node's markings.

This Marking technique may augment any moving target search strategy as easily as adding a heuristic function. Marking does not replace existing heuristic functions or learning techniques. By itself, Marking would not be an effective strategy. Instead Marking often reduces the number of neighboring nodes that the solver must choose from. Because Marking is based only on deductive inferences, it is unlikely that adding it will ever make a problem solver less effective.

Even this Marking strategy is not immune from the fact that learning takes time. This means that the solver using Marking will still enter "markable" heuristic depressions. However, it will enter them at most once. Consequently, Marking will make more of a difference in search time when the search is long and difficult. In this setting the solver may visit areas of the search space many times enabling it to take advantage of previous learning.

Chapter 5

Experiments

Experimentation of the various algorithms was done to help understand their behavior, measure, and compare them.

5.1 Implementation

Algorithms implemented include Korf and Ishida's original moving target search algorithm (BMTS), the same algorithm with the addition of commitment (CMTS), the same algorithm with both the commitment and deliberation enhancements (IMTS), Forgetful Depth-First Search (FDFS), and the hierarchical Marking technique combined with FDFS. The implementation was done in C++, an object oriented language. This facilitated quick prototyping, the easy introduction of new algorithms, and code reuse. Over 8000 lines of code were written for the tools and the experiments in this thesis.

Korf and Ishida's algorithms modify the heuristic matrix and therefore need space to store this information. Due to the enormous size of this matrix, sparse matrix methods had to be implemented.

The experiments were executed on Sparc Workstations. A single search on a 100 by 100 search space that lasts for 1000 moves will typically take a few seconds. For the experiments in Section 5.4, involving millions of searches, it took a dedicated network of 20 workstations 30 hours to process.

A graphics tool to animate searches was developed. The tool was implemented in C++ and uses X-windows. The search space, a problem solver marker ☺, and a target marker ☺, are drawn on the screen. The solver's and target's markers are continually updated as the entities make moves. The program has many useful features. The user can control the speed of the search. To help test a solver's ability in specific situations, the user has the ability to control the target's movements. Methods for tracing the internal state of the solver were implemented. When observing BMTS, current heuristic information (relative to either the solver's or the target's current position) can be drawn in each node. When commitment is used an extra marker indicating the solver's current goal node can be drawn. As the IMTS solver deliberates the nodes of its internal search are highlighted. For FDFS, the search tree is shown by drawing the relevant path history of the solver. When Marking is used, markings are shown within the nodes. An accompanying program lets the user build and edit search spaces, and set the starting positions of the solver and the target. These graphics tools were helpful for verifying the correctness of the implementations, studying the behavior of the different algorithms, and testing out new ideas.

5.2 Small Space and Stationary Target

The first collection of experiments presented here is based on the graph in Figure 35. The problem solver must work its way out of a heuristic depression to find a non-moving target. The main purpose of this experiment is to become familiar with the nature of each of the algorithms. Algorithms tested include BMTS, IMTS, FDFS, and FDFS combined with the hierarchical Marking technique. The results are shown in Figure 36.

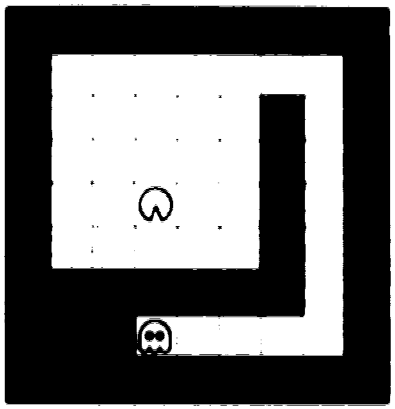


Figure 35: Stationary target search scenario

Algorithm	First Round	Round 10
BMTS	105	17
IMTS	38	37
FDFS	37	37
Marking	37	23

Figure 36: Search times

The second column of Figure 36 shows the time required for each of the algorithms to find the target. Clearly BMTS does not do very well. This problem solver keeps moving back and forth until it finally builds up its heuristic information. The IMTS problem solver moves down one square from the starting position and then begins deliberation. After searching all the nodes in the depression it finds the node in the upper right corner to be an exit. The solver then jumps to the exit node and continues toward the target. (The reason for letting the IMTS solver unfairly jump from the starting node to the exit node, is explained in the next section.) IMTS finishes much sooner than BMTS. The FDFS solver searches the area before it finds its way out of the heuristic depression. Consequently, it performs the same as IMTS. During this first round of search, adding Marking to FDFS does not improve performance.

The third column of Figure 36 shows the number of moves required a few rounds later (acquired knowledge is maintained between rounds), which helps illustrate the learning that occurs during these searches. For every round after the first 10, BMTS

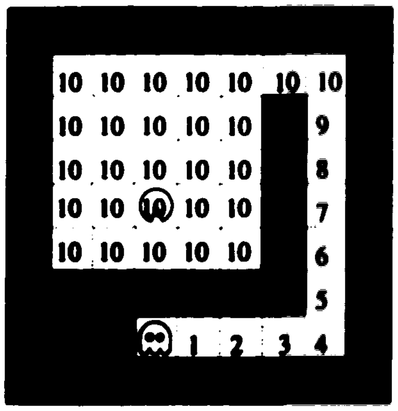


Figure 37: Heuristic information learned by IMTS solver

performs optimally heading directly from the start node to the goal node. At this time the BMTS solver's knowledge is equal to $h^*()$. The IMTS solver does not improve as well as BMTS, even though the same knowledge representation and similar learning is used. Instead of being incremented individually on a per node basis, heuristic information is set to the same value for a group of nodes after deliberation. Figure 37 shows the knowledge of the IMTS solver in the later rounds. Starting the search over again from the middle of the depression, the IMTS solver has to search the entire area again in order to find its way out of the depression. During later rounds FDFS (which does not adapt) requires the same number of moves that it did in the first round. This is where Marking pays off: after a few rounds the number of moves required drops but not quite as much as it did for BMTS.

The reason that Marking causes improvement in this scenario is due to the way the search space gets marked. The path of the solver is shown in Figure 38. The problem solver starts marking, with identifier $M1$, when it goes into one of the corners along the bottom edge of the depression. From there it expands the marked region outward. After the solver moves to (and marks) the corner node and the node above, it moves left onto a node that cannot yet be marked because the nodes above and below it are not marked at this time. Consequently marking is suspended until the solver reaches the other corner. Here it starts marking with a new unique marking

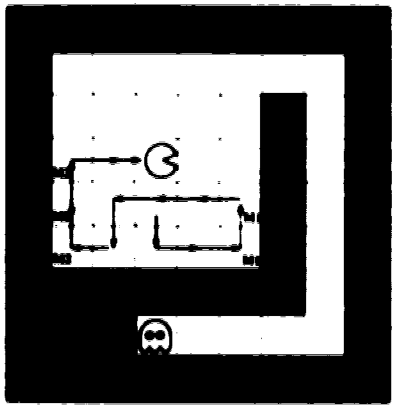


Figure 38: Path of searcher

number $M2$ and therefore creates a region distinct from the region beginning at the other corner. In later rounds the solver stays out of the two "sides" of the depression and heads almost directly on an optimal path to the target.

If the solver started at one of the corners and traveled first along the bottom to the other corner, then the entire bottom edge, and eventually the entire depression, would have had the same marking. This would have provided no information to the solver at the beginning of the following round when it gets placed back in the middle of the marked region. This suggests that the Marking technique may not be effective in some basic circumstances. However, Marking is designed to keep the solver out of regions, not to get the solver out of regions. The only reason that Marking would be ineffective in this case is that the solver is placed in the middle of a marked region. The flaw is not with Marking; it is with what is done to the solver. Furthermore, randomly generated search spaces do not normally have regular structures such as the rectangular depression created here. Instead they have many divots and sub-depressions that force the use of many marking identifiers when marking an area.

5.3 Performance Evaluation

The previous experiment illustrates the behavior of the various problem solvers. The experimentation in this section attempts to measure the ability of these algorithms.

100 by 100 toroidal (wrap-around) search spaces with a density of 35 percent were used. (Each node has a 35 percent probability of being blocked-out.) Although not representative of all possible graphs, these search spaces are challenging and appropriate for this moving target search problem. At the beginning of each search the solver and the target are placed a maximum distance (50 rows and 50 columns) apart. Note that the technique of generating search spaces differs from that of Korf and Ishida [6] [7]. They just removed edges instead of entire nodes. The difference between the two generation techniques and the graphs they produce is not significant.

Ten thousand randomly generated graphs (using the above parameters) were used for testing each of the problem solvers. The mean, median, and range are recorded. Searches lasting longer than 20000 moves were halted and 20000 was recorded for the search. The number of searches exceeding this limit is also recorded.

Two types of targets were tested against. One of them (RAND) behaves completely randomly giving equal probability to each neighboring node when deciding its next move. The other target (AVOID) gives higher probability to nodes farther away from the problem solver thus tending to avoid the solver.

Experimentation was done for BMTS (Ishida's and Korf's original algorithm), the same with Ishida's commitment technique but no deliberation (CMTS), Ishida's algorithm with commitment and deliberation (IMTS), Forgetful Depth-First Search (FDFS), and FDFS with the Marking technique. The results are shown in Figure 39. Note that the mean search times for BMTS are deflated because of the large number (almost 30 per cent) of truncated searches.

Figure 40 shows the probability that a search will require a certain number of

RAND	mean	med	min	max	over
BMTS	11174	10485	101	>20000	2964
CMTS	2110	915	109	>20000	91
IMTS	1700	512	109	>20000	142
FDFS	876	388	93	>20000	19
Marking	604	383	93	>20000	1

AVOID	mean	med	min	max	over
BMTS	9176	7300	135	>20000	1844
CMTS	2037	1162	123	>20000	20
IMTS	1476	512	112	>20000	77
FDFS	808	474	105	>20000	40
Marking	619	461	105	>20000	7

Figure 39: Algorithm performance

moves. To reduce clutter, only the densities for Marking, IMTS, and BMTS are shown. Figure 40 is based on the data for the avoiding moving target; the densities for the random moving target are similar. It should be noted that these curves were generated by smoothing the data from the search runs. The same smoothing algorithm was applied to each of the three sets of data. The resulting graphs may be inaccurate representations of the true populations, however they do correctly show the relationships between the three different algorithms. Having a larger portion of the area under a curve toward the left indicates a better algorithm.

For CMTS and IMTS the degree of commitment used was 10 which is the same value that Ishida suggests in his paper. For the search spaces used in this experiment, adding deliberation with a maximum off-line search of 25 nodes (as suggested by Ishida) yielded little improvement. Allowing for more deliberation, up to 250 nodes, resulted in better performance for IMTS. The results for IMTS shown in Figure 39 are based on this. Using more deliberation was beneficial because the search spaces were dense and had huge heuristic depressions. Recall that the search spaces were generated by blocking out nodes instead of just removing edges which is the method Ishida used.

For the experiments in this section, after completing the deliberation step the

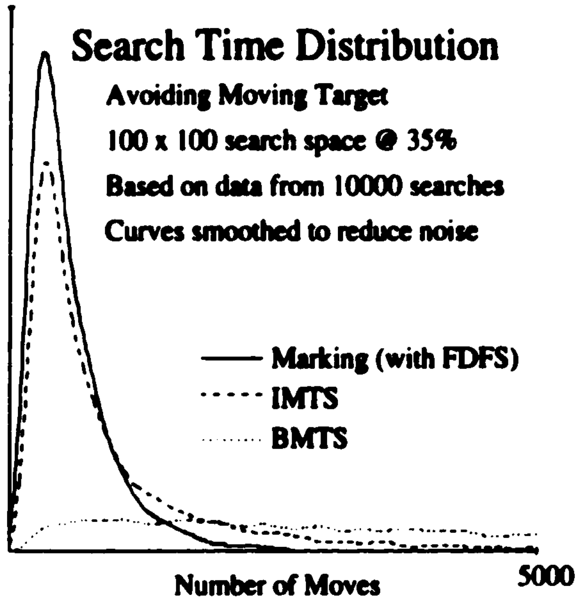


Figure 40: Algorithm performance

IMTS problem solver went directly (in one move) from where it started its off-line search to the exit of the depression. It can be argued that this violates the real-time constraint (especially with such a large degree of deliberation of 250) giving IMTS an unfair advantage. However if the solver had to take single moves to physically get out of a depression after off-line search then there would be doubt as to whether the results reflected the true potential of IMTS. One concern is that the value chosen for the degree of deliberation may be suboptimal. As well, the IMTS algorithm could be enhanced so that during off-line search while it expands one node per move it also makes a physical move "toward" the node being expanded. This may cut down on the distance the solver must travel after deliberation. By using a high degree of deliberation and by jumping the solver out of depressions the results in Figure 39 represent a safe lower bound of the performance limits for the IMTS solver. This strengthens the conclusion that the FDFS algorithm is a better approach than IMTS for the class of search spaces used in these experiments.

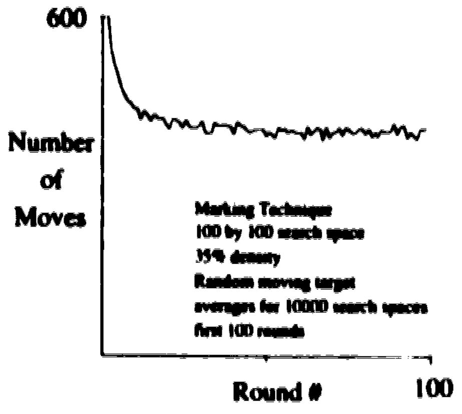


Figure 41: Long term learning using Marking

5.4 Long Term Learning

Experiments were done over a number of rounds to observe general learning on the part of the solver for algorithms that acquire knowledge about the search space. After a search the solver and target were placed back in their initial positions, the solver was allowed to retain any knowledge it had acquired from the previous search, and the search was carried out again.

The hierarchical Marking technique combined with Forgetful Depth-First Search and Ishida's moving target search algorithm with commitment (no deliberation) were tested for improvement over 100 rounds. The experiment setup is the same as the previous section, ten thousand times 100 by 100 spaces at density of 35 percent. The random moving target was used.

Figures 41 and 42 show the performance of Marking and Ishida's algorithm (CMTS) respectively. The data on the CMTS graph is fitted (using least squares) with a straight line. The progression along the x axis is the round number, which can be misleading because the amount of time spent learning during a given round is proportional to the number of moves taken in that round. Both algorithms show improvement over time. Note that the scales on the y-axis differ. CMTS has much more room for improvement than Marking whose performance is good from the beginning.

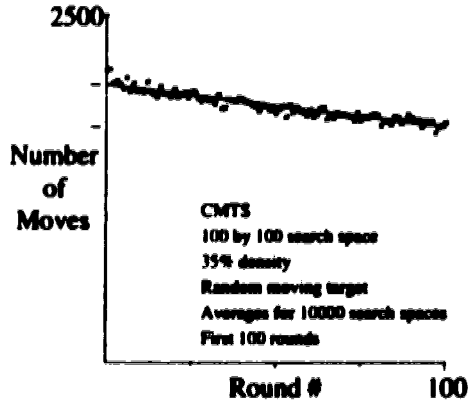


Figure 42: Long term learning in CMTS algorithm

Also, the data shows that the two algorithms learn at different rates. For Marking, the search time drops quickly in the first few rounds and then shows no improvement in later rounds. On the other hand, CMTS will likely maintain its slower rate of improvement until it approaches optimal performance.

Recall that sparse matrix methods had to be implemented for the CMTS solver. Consequently its memory requirement grows as it learns. Allowing this learning to continue for many rounds resulted in paging, thrashing, and slowdown on the machines executing this CMTS experiment. This shows that in practical situations the memory overhead is a concern.

5.5 Evaluation

Clearly the original moving target search algorithm does not perform well compared to the other algorithms discussed in this paper. The sensitivity to the target's movements and the slow incremental learning hamper this algorithm. However the research by Korf and Ishida is still significant because it laid the foundation for this field of study and also led to Ishida's improved algorithm.

Adding commitment makes a substantial improvement. The learning is focused for maximum benefit. The solver gets out of heuristic depressions much quicker.

Unfortunately, learning is "local" and helps little in the long term. In Section 5.4, which shows the benefit of learning over time, Ishida's problem solver improvements come slowly. Eventually (many rounds later) it will behave optimally.

Deliberation effectively reduces the thrashing problem and therefore helps the solver get out of local minimums. Ishida's techniques also require parameters, degree of commitment and deliberation, that must be fine tuned for the current search setting. Overall, deliberation and commitment are good techniques. Unfortunately they are specifically designed to work with the original moving target search algorithm. They cannot readily be added to other algorithms (Forgetful Depth-First Search, for example) with the guarantee of performance improvements.

Forgetful Depth-First Search is simple, requires little overhead and memory, and seems to be a reasonable approach to doing moving target search. The performance results from the experiments done in this paper are promising. For the search settings used here, FDFS is clearly superior to CMTS and IMTS. A criticism with FDFS is that it is not "$O(n^3)$ complete". (Although this is curable for any algorithm by augmenting it with Floyd's algorithm.) On a theoretical note, the Forgetful Depth-First Search algorithm (without any Marking technique) would be usable in a dynamic search setting where edges are added and deleted from the graph as the search progresses. Korf and Ishida's algorithms cannot handle edges being added

Marking proves to be a beneficial addition to the Forgetful Depth-First Search algorithm. Although its learning may not be as complete as knowing the true distance between all pairs of nodes, it comes much sooner and is general. The Marking technique can be easily added to any algorithm, it requires no parameters to be tuned, and it is unlikely that adding Marking will have a negative effect on an algorithm's performance.

Chapter 6

Conclusion

6.1 Summary

This thesis investigated the moving target search problem. The problem is a variation of the standard graph search in that the solver is constrained to computing its moves in a small constant amount of time and the target also moves. Other work in this research area was reviewed. Korf and Ishida [6] introduce the problem and an algorithm to solve it. The algorithm works by incrementing heuristic information. The analysis and experimentation in this thesis found this algorithm to be impractical. Ishida [7] recognizes the shortcomings with the algorithm and presents improvements, namely commitment and deliberation. This thesis verifies that these improvements do make the algorithm much more efficient. One criticism of these algorithms is that their representation of knowledge is large (the size of the search space squared). This thesis introduced an alternative approach to doing moving target search called Forgetful Depth-First Search. The algorithm outperformed the other methods in the experiments done in this thesis. The Forgetful Depth-First Search solver moves effectively about the search space but does not learn or adapt. A technique called Marking attempts to quickly learn about the search space at a higher level than the other method of improving heuristic distances. Experiments confirm that Marking

Algorithm	Knowledge Repn	Space Requirement
Floyd	all-pairs matrix	$O(n^2)$
Follow	target's trail	$O(n)$
BMTS	all-pairs matrix	$O(n^2)$
Commitment	all-pairs matrix	$O(n^2)$
Deliberation	all-pairs matrix	$O(n^2)$
FDFS	own history	$O(n)$
Marking	Regions	$O(n)$

Figure 43: Primary knowledge base of algorithms

Addable Technique	can be combined with	parameters requiring tuning
Floyd	Any	none
Commitment	BMTS	degree
Deliberation	BMTS	degree
Marking	Any	none

Figure 44: Comparison of algorithm enhancements

quickly acquires knowledge about the search space but that this knowledge has its limits. Marking was combined with Forgetful Depth-First Search and the resulting problem solver performed better than plain Forgetful Depth-First Search.

Figures 43 and 44 summarize various points on the algorithms discussed.

6.2 Future Work

The developments introduced in this thesis do not put an end to moving target research. Instead, the new ideas presented here probably can be improved or extended. There may be radically different approaches that are even better.

6.2.1 Mathematical Analysis

Many of the arguments concerning the worthiness of algorithms Forgetful Depth-First Search and Marking are based on intuitive notions rather than mathematical proof.

More work is needed to analyze these algorithms in this way. To further understand their capabilities and shortcomings, more testing should be done.

6.2.2 Solving Real World Problems

The constraints of the moving target search problem are severe. Solver algorithms are extremely limited. Important real searching problems having time constraints with a changing goal will likely have different specifications than the ones used in this thesis. When important real problems are identified, further research must be done to apply the moving target search algorithms to the problem domain. Problems that arise in practice will probably allow for more computation time per move. This leads to questions of how to use extra time most effectively. For example, should the solver use look-ahead or do learning during spare cycles. As the constraints of the search problem are lessened the potential ability (and complexity) of the solver algorithm can be increased in many directions.

6.2.3 Variations

There are many variations of the moving target search problem worth studying. One interesting extension of the problem is to allow for multiple solver agents. Algorithms would have to be modified to utilize this "parallelism" effectively. Algorithms based on learning would have the different solver entities trade knowledge. When Forgetful Depth-First Search solvers are working together they should avoid moving onto nodes that are in the history of any of the other searchers in order to keep apart from each other. Another variation of the search problem involves using a dynamic search space where edges between nodes are formed and broken during the search. Motivation for researching such various problems will occur as they arise in real situations.

A complementary problem to moving target search is moving solver evasion. The objective is to develop a target that is difficult to catch. In some ways this is a harder problem. If a solver knows the true shortest distance between every pair of nodes then it can easily determine the optimal move. This is not true for a target with the same knowledge. A target that always tries to increase the distance between itself and the solver will likely just trap itself in a local maximum with no way to escape as the solver closes in. Given smarter targets, a useful study would be to face them against the various moving target search algorithms discussed in this thesis. It would be interesting to see not just the performance of each of the solver algorithms, but the performance relative to the other solver algorithms as the target's evasive ability is increased.

6.2.4 Higher Learning

The reasoning process of Korf and Ishida's algorithms updates heuristic information. Marking works at a higher level by creating "regions". Such techniques may "perform well", however few would consider such approaches as "intelligent". When a human user controls the target, he/she can quickly become familiar with the solver's behavior and then exploit it's weaknesses. The challenge in most interactive computer games is coordination and speed instead of intelligence. One research direction would be to create a solver with higher reasoning ability. Such an entity might make a more "interesting" opponent in a simulation or video game.

Bibliography

[1] S. Baase: Computer Algorithms, Introduction to Design and Analysis, 2nd edition, Addison Wesley, 1988.

[2] P. R. Cohen and H. J. Levesque: "Intention is Choice with Commitment," Artificial Intelligence, volume 42, number 3, pp. 213-259, 1990.

[3] M. Ginsberg and W. Harvey: "Iterative Broadening," AAAI-90, pp. 216-220, 1990.

[4] D. Kinny and M. Georgeff: "Commitment and Effectiveness of Situated Agents," IJCAI-91, pp. 82-88, 1991.

[5] R. E. Korf: "Real-Time Heuristic Search," Artificial Intelligence, volume 42, number 2-3, pp. 189-211, 1990.

[6] T. Ishida and R. E. Korf: "Moving Target Search," IJCAI-91, pp. 204-210, 1991.

[7] T. Ishida: "Moving Target Search With Intelligence," AAAI-92, pp. 525-532, 1992.

[8] S. Melax: "New Approaches To Moving Target Search," AAAI-93 Fall Symposium On Games, Planning and Learning, proceedings pp. 30-38, 1993.

[9] M. Pollack and M. Ringuette: "Introducing the Tileworld: Experimentally Evaluating Agent Architectures," AAAI-90, pp. 183-189, 1990.

[10] D. J. Slate and L. R. Atkin: "Chess 4.5 The Northwestern University Chess Program," Chess Skill In Man And Machine, Chapter 4, pp. 82-118, editor P. Frey, 1977.