Computer Science Department Technical Report
University of California
Los Angeles, CA 90024-1596

MAKING LOCALLY OPTIMAL DECISIONS ON GRAPHS WITH CYCLES

J. Pemberton
R. E. Korf

March 1992
CSD-920004

Making Locally Optimal Decisions on Graphs with Cycles

Joseph Pemberton and Richard E. Korf*
Computer Science Department
University of California, Los Angeles
Los Angeles, CA. 90024

March 20, 1992

Abstract

We present two new algorithms, LCM and IBFS, that make locally optimal incremental planning decisions for the task of finding a path to a goal state in a problem space that contains cycles. Earlier work (RTA* [10]) only solves this locally optimal decision problem when the problem space is tree structured. We precisely characterize the time and space complexity of both new algorithms, and show that they are asymptotically optimal. In addition, we present empirical evidence for a variety of maze problem spaces which shows that LCM and IBFS consistently produce shorter solutions than RTA*, and that the average computation cost of IBFS and LCM grows only slowly with the size of the problem. Finally, we show that as the lookahead depth increases, both algorithms require less computation per move decision than RTA*. An earlier version of this paper appeared in [15].

*This research was supported by an NSF Presidential Young Investigator Award, No. IRI-8552925, and a grant from Rockwell International.

1 Introduction

Consider the following planning problem. You are in a maze, and your objective is to find your way out as efficiently as possible. Your sensory information is limited in that you can only see those locations that are adjacent to your current position. You can keep track of the portion of the maze you have seen so far, however, and recognize when you have returned to a previously visited location. The sensory limitation precludes planning an optimal sequence of moves in advance, since you can only discover the maze by moving through it. The problem is how to minimize the total distance you must travel to get out, including all exploration and backtracking. In other words, what method should you use to decide which maze cell to visit next in order to minimize the distance you must cover.

When the maze is tree structured (*i.e.,* it does not contain cycles), then Real-Time A* (RTA*) [10] solves this problem in constant time per move decision. RTA*'s decisions are not locally optimal, however, when the maze contains cycles. Given this deficiency, there are three ways to proceed. The first is to apply RTA* directly to problem spaces with cycles, and accept locally suboptimal decisions. The second option is to generate an algorithm that makes locally optimal decisions on a graph, although not in constant time per decision. Two algorithms which implement this option are presented in section 3. The last option is to generate an extension of RTA* that makes locally optimal planning decisions in constant time per decision for graphs with cycles. Theoretical results presented in section 4 show that this is not possible. Empirical results in section 5 show that our new algorithms outperform RTA* when the problem space contains even relatively few cycles.

2 Background

Traditional search algorithms, such as A*, can be used to preplan an optimal sequence of move decisions only when the entire problem space is available to the problem solver, albeit at some computational cost. This allows the problem solver to delay any action until a complete, optimal solution path is found. Although the algorithms presented here can be used instead of A* to generate globally suboptimal solution paths more efficiently than A*, their main objective is to provide incremental move decisions in situations where the sensory information is limited.

RTA* operates by storing in memory all states that have been previously visited by the problem solver, along with a heuristic estimate of each state's distance to a goal. For example, consider the graph in figure 1a. Here we assume that all edges have unit cost, that the initial static heuristic values are as shown, and that the problem solver can only "see" new nodes that are adjacent to its current state (thus the dashed edge is not initially visible). We use a lowercase h to denote initial or static heuristic values, and an uppercase H to denote stored heuristic values. For each move, RTA* generates each neighbor n' of the current state n, and computes its $f_n(n')$ value as the sum of the edge cost to the node, $k(n, n')$, plus that node's heuristic value, $h(n')$ or $H(n')$. If the node was visited before, then it uses $H(n')$, the

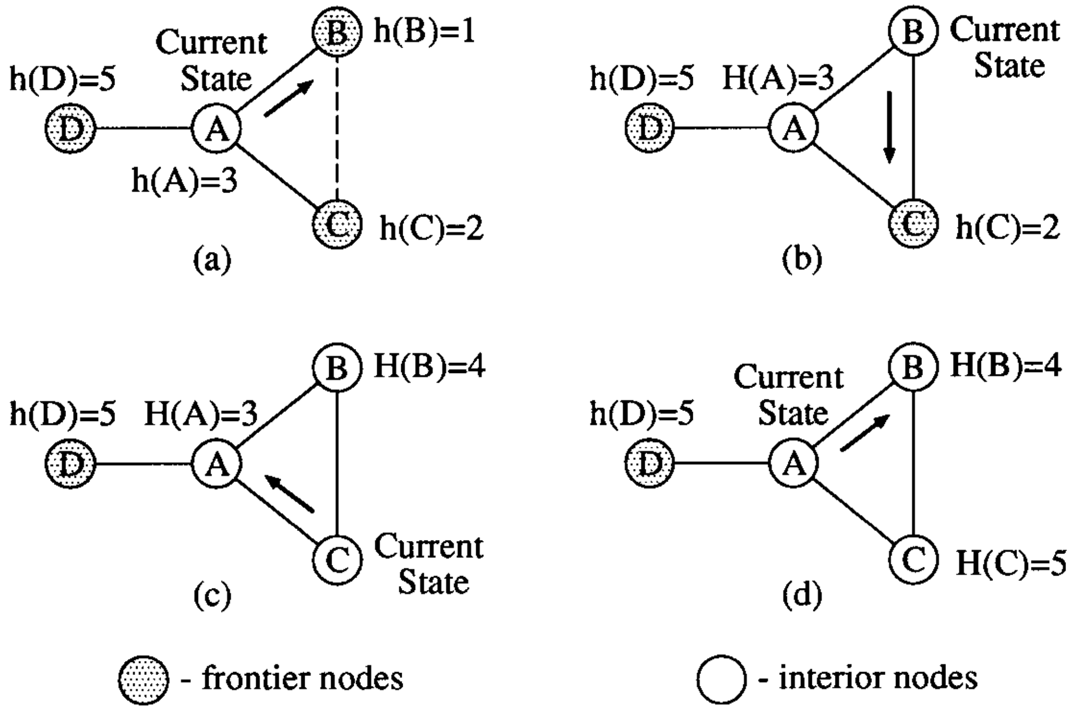


Figure 1: Example of RTA* on a Cycle.

value that was stored with that node when it was last visited. Otherwise, its heuristic value is determined by the static evaluation function $h(n')$. Initially, A is the start node, so RTA* computes $f_A(D) = k(A, D) + h(D) = 1 + 5 = 6$, $f_A(B) = k(A, B) + h(B) = 1 + 1 = 2$, and $f_A(C) = k(A, C) + h(C) = 1 + 2 = 3$. The problem solver then moves to a neighbor n' with the lowest $f_A(n')$ value (node B), and stores the $f_A(n'')$ value of the second-best neighbor n'' (node C) as the new H-value for the previous state. Thus $H(A) = f_A(C) = 3$ is stored with node A (see figure 1b). The second-best value is stored because it represents the estimated cost of backtracking to a goal through that node. RTA* is guaranteed to eventually find a solution path in a finite graph, and makes locally optimal planning decisions when the problem space is a tree [10][1].

When the problem space contains cycles, however, RTA* may make locally suboptimal move decisions as can be seen by continuing the example. After moving from node A to node B (see figure 1b), RTA* will discover the edge from node B to C, compute $f_B(A) = 1+3 = 4$, $f_B(C) = 1 + 2 = 3$, move to node C, and store $H(B) = f_B(A) = 4$ with node B (see figure 1c). Next, it will compute $f_C(A) = 1 + 3 = 4$, $f_C(B) = 1 + 4 = 5$, move to node A, and store $H(C) = f_C(B) = 5$ with node C, resulting in the situation in figure 1d. In the next step, RTA* computes $f_A(D) = 1 + 5 = 6$, $f_A(B) = 1 + 4 = 5$, $f_A(C) = 1 + 5 = 6$, and moves to node B, storing $H(A) = f_A(C) = f_A(D) = 6$, with node A. Unfortunately, this is not a locally optimal decision, since RTA* has seen enough information at this point to realize that B and C form a dead-end loop, and that moving to node D is the correct decision.

[1]A similar algorithm, called Learning RTA* (LRTA*) [10], stores the *best* value with each visited node, but its move decisions are not guaranteed to be locally optimal, even on a tree.

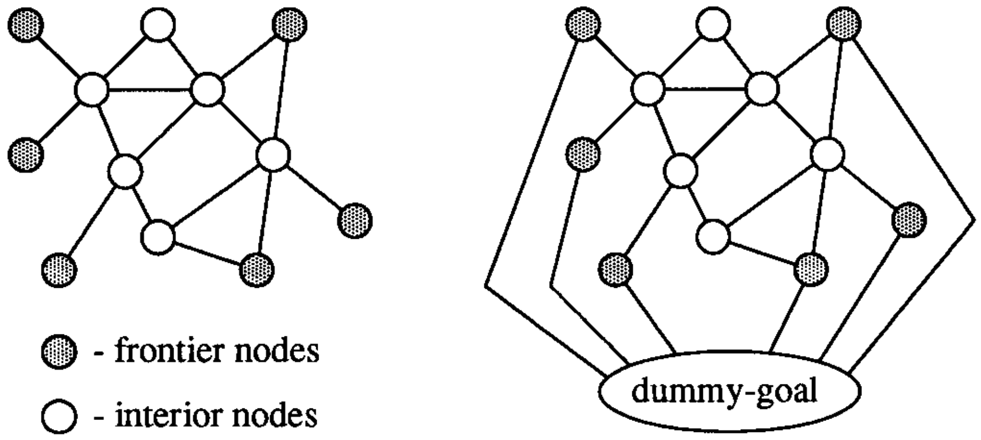


Figure 2: Sample Local Graph and Corresponding Augmented Local Graph.

RTA* will eventually move to node D, but by making the initial value of $h(D)$ sufficiently large, we can force RTA* to go around this loop an arbitrary number of times before it finally exits[2]. Correcting this deficiency of RTA* was the original motivation behind this work.

Brute-force search algorithms will also make locally suboptimal move decisions. For example, consider a modified depth-first search that breaks ties in favor of the smallest heuristic value. For the graph in figure 1, this modified depth-first search will move to node B and then to node C. Since both neighbors of node C have already been visited, the algorithm backtracks to node A via node B rather than moving directly to node A. Although this algorithm will not loop, its backtracking behavior is locally suboptimal, which can significantly increase the length of the solution.

Before continuing, we define the following terms which are used in the remainder of the paper (see figure 2). The *local graph* is the subset of the problem space nodes and edges that the problem solver has "seen" so far. The nodes in the local graph are divided into *interior nodes*, which have been expanded by the problem solver, and *frontier nodes*, which have been generated but not yet expanded. The *augmented local graph* consists of the current local graph plus a *dummy-goal node* and one edge from each frontier node to the *dummy-goal node*, with edge costs that correspond to the static heuristic values of the frontier nodes. This augmentation is based on the face-value principle, namely that the heuristic values of frontier nodes are treated as accurate estimates of the distance to a goal node. This is a reasonable assumption when additional information about the problem space and heuristic function is not available. We define a *locally optimal* move as the first step along an optimal path, assuming the static heuristic values of the frontier nodes are exact, or in other words, a move along an optimal path towards the dummy-goal node in the augmented local graph.

[2]LRTA* performs even worse than RTA* when the problem space contains cycles.

LCM Algorithm:

```
1. current state ← start node;
   {frontier} = {current state};
2. If the current state is a goal state, then success;
   If the frontier set is empty, then no goal exists;
3. Expand the current state and update the *frontier* and *interior* sets if necessary;
4. Add the current state to the *update* queue;
   While the *update* queue is not empty, execute the following:
         a. Remove a state from the *update* queue;
         b. Make its stored heuristic value consistent with its neighbors;
         c. If the node's stored heuristic value has changed,
            then add its neighbors to the *update* queue;
         d. If the number of nodes updated exceeds the cost of finding
            the shortest path tree from the dummy-goal node,
            then empty the update queue and calculate the shortest path tree;
5. Move the current state to a neighbor which has a minimum $f$-value;
6. If the current state has been previously expanded, goto 5,
   Else goto 2.
```

Figure 3: Local Consistency Maintenance Algorithm

3 Two Locally Optimal Graph Algorithms

In this section, we present two new algorithms that make locally optimal decisions on graphs that may contain cycles.

3.1 Local Consistency Maintenance

Local consistency maintenance (LCM) is based on the idea that if we maintain for each interior node a heuristic value that represents the best estimate of the cost of reaching a goal from that node, then locally optimal decisions can be based on the values of the current state's neighbors. The heuristic value of a node n is *locally consistent* if and only if it is equal to the minimum $f_n(n')$ value of all its neighbors n':

$$H(n) = \min_{n'}[f_n(n') \mid f_n(n') = k(n,n') + H(n')] \text{ for all adjacent nodes } n',$$

where $H(n') = h(n')$ when n' is a frontier node. A local graph is *locally consistent* if each interior node is locally consistent, in which case the heuristic value of each node represents the best estimate of the distance from that node to a goal.

The LCM algorithm (shown in figure 3) operates by expanding the current state, possibly generating new frontier nodes. It then updates the stored heuristic value of the current state and recursively updates the stored heuristic values of neighbors of updated nodes as needed until all interior nodes are locally consistent. Finally, LCM moves to a neighbor with minimum f-value, where ties are broken randomly. The procedure is repeated until a goal node is visited, or all frontier nodes are expanded. For example, on the graph in figure 1, LCM initially makes the same decisions as RTA*, moving to node B and then to node C (see figure 1c). Once C is expanded, node D is the only remaining frontier node, and the updating step terminates with $H(A) = 6$, and $H(B) = H(C) = 7$. At this point, the problem solver moves to node A. Since node A was previously expanded, no updating is necessary, and the problem solver moves to node D, which is the locally optimal move.

There are a number of ways to update the heuristic values of interior nodes, all of which are variations of the *value iteration* method of dynamic programming [7]. The simplest consists of a processor at each interior node that continually polls its immediate neighbors in order to maintain a locally consistent value. A serial implementation of this mechanism maintains a queue of nodes to be updated. After a move is performed, all newly-expanded nodes are added to the queue. Nodes are then removed from the queue one at a time, and a new consistent heuristic value is calculated based on the current values of its neighbors. If a node's value has changed, then its immediate neighbors are added to the update queue. This is continued until the update queue is empty. This scheme can be inefficient, for example, since the updating may merely simulate the movement performed by RTA* in figure 1, thus making the complexity of the updating process dependent on the heuristic value of the frontier node D.[3]

The way we avoid this problem comes from the observation that the locally consistent heuristic values of interior nodes are equal to their shortest path distance to the dummy-goal node in the augmented local graph, and thus can be calculated using a shortest path algorithm. Although the worst-case complexity of this shortest path approach is independent of the heuristic values of frontier nodes, the average case complexity is at least linear in the size of the local graph since every interior node is processed after each move, even if its value doesn't change.

Our actual implementation combines both approaches by using the queue method while keeping track of the number of updates that are performed. If the number of updates exceeds the worst case number of updates required by the shortest path approach ($O(|V|^2)$ where V is the set of local graph nodes), then we interrupt the queue approach and switch to the shortest-path approach. Thus, the worst-case complexity of this hybrid algorithm is bounded by the shortest-path algorithm, while the average case complexity is typically less, depending on the problem space.

By maintaining a locally consistent local graph, LCM can be viewed as planning a locally optimal move decision for each interior node. The end result is that each interior node is labeled with its shortest-path distance in the augmented local graph. Although this makes it possible to base move decisions only on locally stored information, it also requires the heuristic values of all inconsistent nodes to be updated after each move, whether or not they

[3]This observation is due to Moises Goldszmidt.

```
IBFS Algorithm:
1. current state ← start node;
   {frontier} = {current state};
2. If the current state is a goal state, then success;
   If the frontier set is empty, then no goal exists;
3. Expand the current state and update the frontier and interior sets, if necessary;
4. Perform a best-first search on the augmented local graph,
       using the A* cost function ($f = g + h$),
       stopping when a frontier node is chosen for expansion;
5. Move the current state to the neighbor on the path to the chosen frontier node;
6. If the current state has been previously expanded, goto 5,
   Else goto 2;
```

Figure 4: Incremental Best-First Search Algorithm

affect the current move decision, which can be inefficient.

3.2 Incremental Best First Search

An alternative to LCM is Incremental Best-First Search (IBFS), which is designed to spend the least amount of effort necessary to make a single locally optimal move. The IBFS algorithm (shown in figure 4) operates by performing a best-first search on the local graph starting from the current state. Interior and frontier nodes are evaluated using the A* cost function ($f = g + h$), where g is the cost of the best path from the current state and h is the static heuristic estimate of the distance to the goal. Ties between nodes with the same cost relative to the current state are broken at random, and the search process stops when a frontier node is chosen as the lowest cost node. IBFS then commits to moves along the locally optimal path toward the chosen frontier node, until the local graph is changed by the expansion of a frontier node. At this point, all intermediate results from the previous search are discarded, and the search process is repeated until a goal state is visited (success) or until all frontier nodes have been expanded (no goal exists).

For example, on the graph in figure 1, IBFS will initially make the same decisions as RTA* and LCM, moving from node A to node B and then to node C (see figure 1c). After expanding node C, IBFS performs a best-first search on the augmented local graph by generating the neighbors of the current state (node C), marking node C as examined[4], and calculating the heuristic value of all neighbors of the current state ($f_C(A) = k(C, A) + h(A) = 1 + 3 = 4, f_C(B) = 1 + 1 = 2$). IBFS then continues the best-first search, examining all generated nodes in order of increasing f-value, until a frontier node is chosen for examination. Since

[4]We use the term "examination" when the problem solver expands a node in the local graph, and "expansion" when the problem solver moves to a frontier node and expands it, increasing the information in the local graph.

the sensory limitation precludes generating the neighbors of a frontier node, IBFS stops searching and moves toward the chosen frontier node. In this example, node B is examined first followed by node A. Finally the frontier node D is examined, and the problem solver moves to node A which is the first step on the shortest path to D. Since node A was already expanded before, the problem solver continues moving to node D which is the next locally optimal move.

IBFS differs from A* in that it commits to partial solution paths in order to be able to further explore the problem space, whereas A* requires access to the complete problem space so it is only able to preplan the complete solution when the complete problem space is accessible. IBFS only stores the current local graph and its static heuristic values, and thus must plan the next series of moves from scratch every time a frontier node is expanded. Because it does not store any of the results from previous decisions, IBFS can also be inefficient in some cases.

4 Theoretical Properties

For the theorems in this section, we assume that all static heuristic values are non-negative, that all edge costs are positive, and that a goal node is reachable from every node. Although many planning domains deal with the possibility of reaching a dead-end, we have assumed that this doesn't happen to simplify our analysis. Lastly, all goal nodes have heuristic values equal to zero, and they remain frontier nodes once they are generated. The first property we consider is completeness; namely, under what conditions are the move decisions by LCM and IBFS guaranteed to lead to a goal state?

Theorem 1 *In a finite problem space with positive edge costs, where a goal is reachable from every state, LCM and IBFS will eventually visit a goal.*

Proof: IBFS always moves along a finite locally optimal path, which does not contain cycles because the best-first search does not reconsider nodes already in the best-first search tree, until it expands a frontier node or visits a goal node. Since there are only a finite number of frontier nodes, IBFS will eventually visit a goal node if one exists.

LCM always moves to a neighbor with minimum f-value. Since the edge costs are assumed to be positive, the h-value of the new current state will be less than the h-value of the previous state. Thus LCM can only move a finite number of steps before expanding a frontier node. Since the number of frontier nodes is finite, LCM must eventually visit a goal node if one exists. □

The second theorem addresses the issue of correctness, namely that the move decisions made by LCM and IBFS are locally optimal.

Theorem 2 *Each move made by LCM and IBFS is along a path whose estimated cost of reaching a goal is minimum, based on the local graph.*

Proof: The proof of this theorem follows directly from the descriptions of both algorithms. LCM moves to a neighbor with minimum f-value in the local graph based on locally consistent heuristic values. The definition of local consistency guarantees that this is a locally optimal decision. Likewise, IBFS bases its move decision on the results of a best-first search of the local graph. Its decision is locally optimal because a minimum cost path in the augmented local graph corresponds to a path whose estimated cost of reaching a goal is minimum.$\square$

It is interesting to note that the update rule used by LCM is identical to the update rule employed by LRTA* [10], and thus LCM can be viewed as a distributed version of LRTA*. If the stored heuristic values for interior nodes are reused in subsequent trials on the same problem space, then LCM can be used to learn actual distances from each node to the goal. This will occur when the local graph contains the entire problem space, and all non-goal frontier nodes have been expanded. When the update stage is complete, all heuristic values for the interior nodes will correspond to their actual minimum distance to a goal node. At this point, since the heuristic values are completely accurate, locally optimal move decisions will follow a globally optimal path to the goal from any node in the problem space.

The next two properties describe the worst-case behavior of both algorithms. Since LCM and IBFS both store the local graph ($G(V,E)$) plus a constant amount of additional information per node, their space complexity is $O(|E|+|V|)$. Time complexity is addressed by the following theorem.

Theorem 3 *Given a local graph, $G(V,E)$, the worst-case time complexity of a single planning step is $O(|V|^2)$ for both IBFS and LCM .*

Proof: The worst-case time complexity for LCM occurs when the heuristic values of all nodes must be updated, in which case LCM reduces to calculating the shortest path tree in the augmented local graph from the dummy-goal node. For IBFS, the worst-case time complexity occurs when the best path from the current state of the problem solver to the dummy-goal node has a cost that is large enough to force the examination of all interior nodes, which is equivalent to calculating the shortest path tree in the augmented local graph from the current state of the problem solver. The worst case for finding the shortest path tree in a graph occurs with the graph is complete, in which case $|E| = |V|^2$. If we use Dijkstra's algorithm [8] to determine the shortest path tree in the local graph, then the worst-case time complexity of both LCM and IBFS for a single decision is $O(|V|^2)$. [5] $\square$

Next we discuss a lower bound on the complexity of making locally optimal move decisions when the problem space is a general graph with cycles. For the following theorem, we assume

[5]When the graph is bounded-degree, then the worst-case time complexity will depend on the algorithm used to find the shortest path, since Dijkstra's algorithm is not optimal for sparse graphs.

that the problem space can be represented as an undirected weighted graph, that the problem solver has no prior knowledge of the problem space, and that the initial heuristic information is monotone (*i.e.,* $h(n) \leq h(n') + k(n,n')$ [14]).

Theorem 4 *Given a local graph, G(V,E), a series of $|V|$ locally optimal move decisions requires at least $O(|V| * |E|)$ computations in the worst case.*

Proof: (See appendix A.)

As a result of theorem 4, there is no general way to amortize the cost of one worst-case move decision over a series of moves, thus, in general, it is not possible to make locally optimal planning decisions in real time on a graph with cycles. This result and the fact that in the worst case $|E| = O(|V|^2)$ lead to the following corollary.

Corollary 1 *IBFS and LCM are asymptotically optimal algorithms for making a series of locally optimal move decisions.*

Proof: The worst-case time complexity of LCM and IBFS is bounded by the algorithm used to find the shortest path tree. In the worst case of a complete local graph ($|E| = |V|^2$), IBFS and LCM use Dijkstra's algorithm to find the $|V|$ shortest path trees in $O(|V|^3)$ computations. From theorem 4 we know that a series of $|V|$ locally optimal move decisions on a complete local graph requires at least $O(|V| * |V|^2) = O(|V|^3)$ computations. IBFS and LCM are asymptotically optimal in terms of space complexity because their storage is linear in the size of the local graph, and an additional result of theorem 4 is that the local graph must be stored in order to make locally optimal decisions. Thus LCM and IBFS are asymptotically optimal. □

5 Experimental Results

5.1 Experiment description

RTA*, LCM, and IBFS were tested in a number of experiments to determine how much better the locally optimal planning decisions made by LCM and IBFS are compared to RTA* when the problem space contains cycles, and the cost of this improved decision quality in terms of average computation (number of node examinations) per planning decision.

The algorithms were tested primarily on mazes created by performing a depth-first search on a graph which corresponds to a complete grid, where the children are ordered randomly. As the depth-first search proceeds, graph edges are removed if they lead to a previously visited node (figure 5a shows the first edge removed). The end result is a depth-first search tree (see figure 5b) which corresponds to a maze without cycles. Cycles are then created by

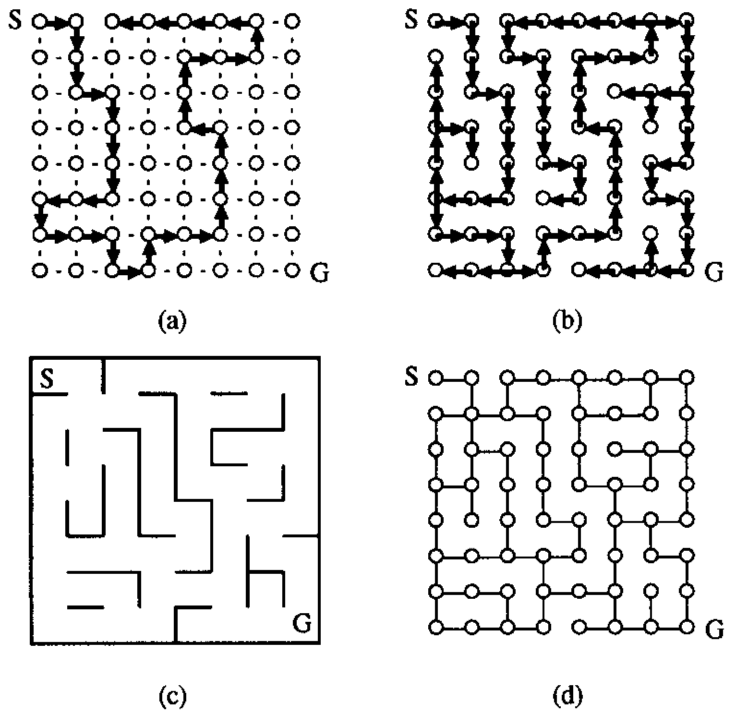
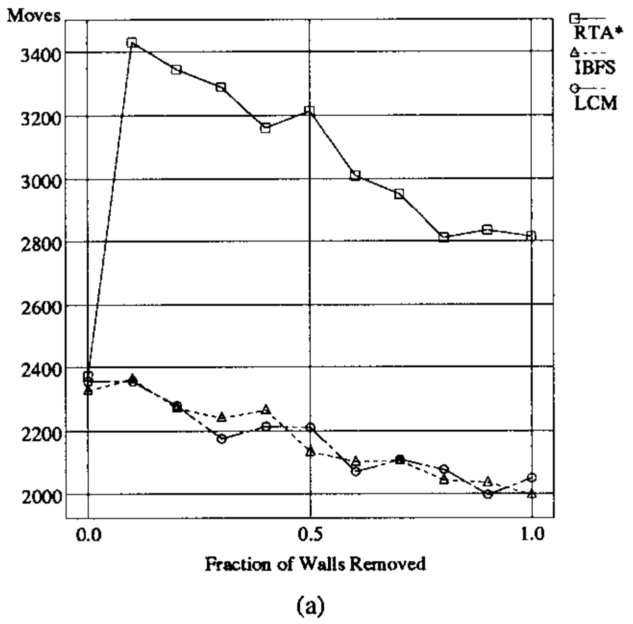


Figure 5: Sample problem space: (a) Depth-First Search on a Grid, (b) Depth-First Search Tree, (c) Maze with 20% of the Walls Removed, (d) Corresponding Graph (G is the Goal and S is the Start Node).

randomly adding edges back into the tree or similarly removing walls from the maze (see figure 5c and d). The problem spaces considered range from a tree (no walls removed) to a complete grid (all internal walls removed), and the start and goal states are placed at opposite corners of the problem space. Note that the fraction of walls removed is correlated with the number of cycles in the problem space.

For the following set of experiments, the problem solver was only able to "see" new nodes that are adjacent to its current location. For all three algorithms, we measured the total distance traveled to the goal and the average number of node examinations per move for a variety of maze types. Two different sets of initial heuristic information were used: $h(i) = 0$ everywhere, corresponding to the case where the problem solver does not know the location of the exit, and $h(i) = \mathit{manhattan\ distance}(i, goal)$, which is the length of a shortest path if all walls were removed, corresponding to the case where the problem solver knows the relative coordinates of the exit. The maze types ranged from no walls removed to 100% of the internal walls removed in increments of 10%. The mazes contain 2500 nodes (a 50 by 50 grid), and the results are averaged over 100 different random mazes of the same type (*e.g.,* 100 mazes with 20% of the walls removed) with 10 independent trials per maze. After each trial, all information about the problem space was discarded before the next trial was started. Different mazes of the same type were used to average out the effect of a particular random choice in the maze generation, whereas multiple independent trials were used to average out the effect of a particular random tie-breaking choice by the algorithms.



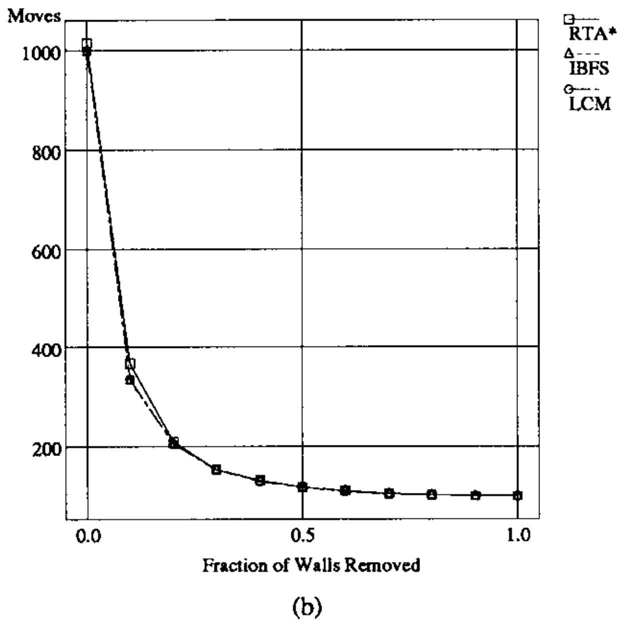
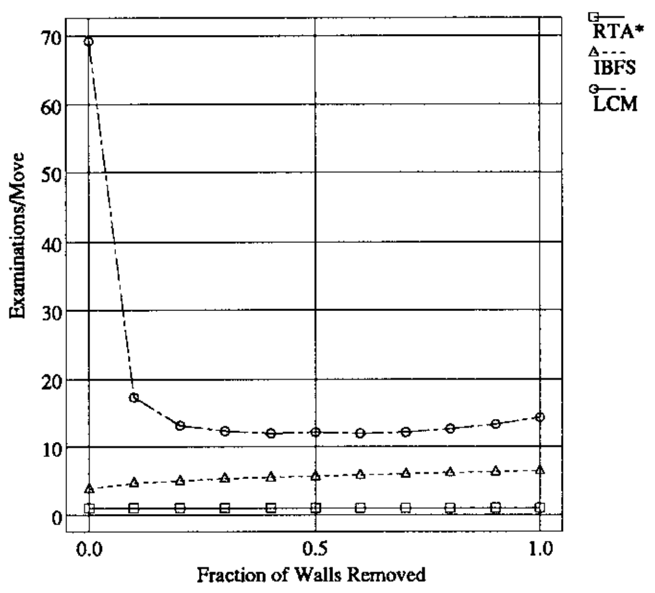


Figure 6: Moves vs. Fraction of Walls Removed, (a) $h(i) = 0, \forall i$; (b) $h(i) = manhattan\ distance(i, goal)$.

5.2 Results

Figure 6a shows the average moves as a function of the fraction of walls removed when the initial heuristic information is zero everywhere. As expected, the number of moves required by both IBFS and LCM is fewer on the average than RTA* (*e.g.,* one-third fewer moves for mazes with 20% of the walls removed). When the initial heuristic information is the manhattan distance (figure 6b), the number of moves required by LCM and IBFS is only slightly less than RTA* with the maximum occurring with 10% of the walls removed (roughly 8% fewer moves). This is due to the fact that, when the fraction of walls removed is greater than .30, the manhattan distance information, plus the fact that the number of obstacles is reduced, make it possible for all three algorithms to follow a path to the goal which is nearly optimal (98 moves). When the problem space is a tree (no walls removed), all three algorithms make locally optimal decisions, and thus require roughly the same number of moves to find the goal.

Figure 7 shows the average number of node examinations per move as a function of the fraction of walls removed. In both cases, RTA* only requires one examination per move, whereas IBFS and LCM both require more. LCM requires many more examinations per move when the problem space is a tree because it often updates the stored values of each node along a branch each time the branch is extended. In general, the relative efficiency of LCM and IBFS depends on both the fraction of walls removed and the initial heuristic information. For example, when $h(i) = 0$ everywhere (figure 7a) LCM requires more examinations per move than IBFS, whereas when $h(i) = manhattan\ distance$ (figure 7b), LCM requires less examinations per move than IBFS when the fraction of walls removed is greater than



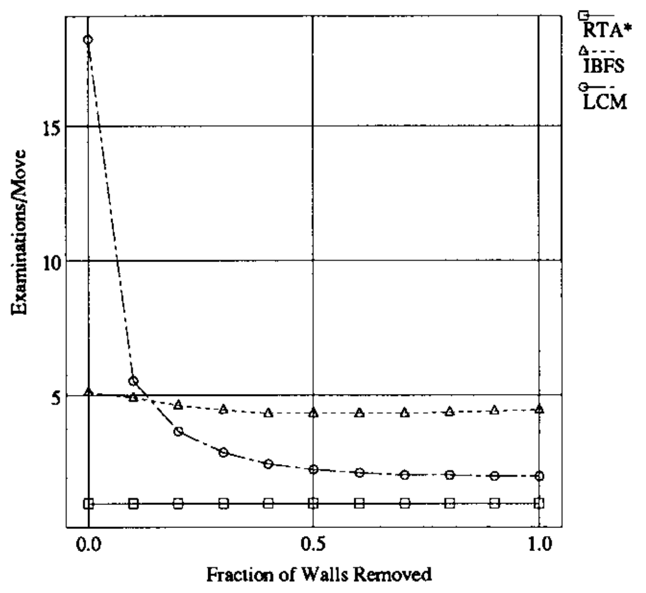


Figure 7: Average Examinations/Move vs. Fraction of Walls Removed, (a) $h(i) = 0,\ \forall i$; (b) $h(i) = manhattan\ distance(i, goal)$.

20%. This is because LCM maintains estimates of the distance to the goal for each interior node. Thus when the heuristic value is zero everywhere, some interior nodes will have to be updated after each move, whereas when the initial heuristic value is the manhattan distance, the heuristic values of the interior nodes need not to be updated as frequently.

Figure 8 shows the examinations per move versus problem space size (*i.e.*, number of nodes). For both IBFS and LCM, when the initial heuristic information is zero everywhere, the examinations per move grow slowly with the size of the problem space, whereas when the initial heuristic information is manhattan distance, the examination per move remain fairly constant. This behavior is typical for the problem spaces considered.

The general (and not unexpected) observation is that IBFS and LCM produce shorter solutions than RTA*, although at a greater computational expense. Thus when solution length is the only performance criterion, IBFS and LCM are the preferred approach. The obvious question is what if the solution quality is measured as a combination of the computation cost and solution length. As an example, figure 9 shows the total cost as a function of the ratio, r, of computation cost to solution length (*i.e.*, r is the rate of exchange between one computation and one unit of the solution length[6]) for the case where the heuristic information is zero everywhere on mazes with 20% of the walls removed. In this case, IBFS is the preferred algorithm when the cost ratio is less than 0.13.

[6]This follows the presentation in [17]