Optimal and Efficient Path Planning for Unknown and Dynamic Environments

Anthony Stentz

CMU-RI-TR-93-20

The Robotics Institute
Carnegie Mellon University
Pittsburgh, Pennsylvania 15213

August 1993

This research was sponsored by ARPA, under contracts "Perception for Outdoor Navigation" (contract number DACA76-89-C-0014, monitored by the US Army TEC) and "Unmanned Ground Vehicle System" (contract number DAAE07-90-C-R059, monitored by TACOM). Views and conclusions contained in this document are those of the author and should not be interpreted as representing official policies, either expressed or implied, of ARPA or the United States Government.

Table of Contents

List of Figures

Abstract

The task of planning trajectories for a mobile robot has received considerable attention in the research literature. Algorithms exist for handling a variety of robot shapes, configurations, motion constraints, and environments. Most of the work assumes the robot has a complete and accurate model of its environment before it begins to move; less attention has been paid to the problem of unknown or partially-known environments. This situation occurs for an exploratory robot or one that must move to a goal location without the benefit of a floorplan (indoor) or terrain map (outdoor). Existing approaches plan an initial global path or route based on known information and then modify the plan locally as the robot discovers obstacles with its sensors.While this strategy works well in environments with small, sparse obstacles, it can lead to grossly suboptimal and incomplete results in cluttered spaces. An alternative approach is to replan the global path from scratch each time a new obstacle is discovered. While this approach is optimal, it is grossly inefficient and can require a high-performance computer for real-time operation. This paper introduces a new algorithm, D*, capable of planning paths in unknown, partially known, and changing environments in an efficient, optimal, and complete manner. D* models the environment as a graph, where each node represents a robot state (e.g., a location in a room), and each arc represents the cost (e.g., distance to travel) of moving between two states. Initially, a path is planned from the goal to the robot's location using known information. As the robot moves, its sensors discover obstacles in its path. These discoveries are handled by *modifying the arc costs*. D* propagates information minimally about these arc changes in the graph to compute a new optimal path. The process repeats until the robot reaches the goal or determines that it cannot. After a discussion of prior work, the paper introduces the algorithm, proves its soundness, optimality, and completeness, illustrates some path planning applications, compares it to an alternative algorithm, and summarizes the results.

1.0 Introduction

The research literature has addressed extensively the motion planning problem for one or more robots moving through a field of obstacles to a goal. Most of this work assumes that the environment is completely known before the robot begins its traverse (see Latombe [10] for a good survey). The optimal algorithms in this literature search a state space (e.g., visibility graph, grid cells) using the distance transform [6] or heuristics [14] to find the lowest cost path from the robot's start state to the goal state. Cost can be defined to be distance travelled, energy expended, time exposed to danger, etc.

Unfortunately, the robot may have partial or no information about the environment before it begins its traverse but is equipped with a sensor that is capable of measuring the environment as it moves. One approach to path planning in this scenario is to generate a "global" path using the known information and then attempt to "locally" circumvent obstacles on the main route detected by the sensors [5]. If the main route is completely obstructed, a new global path is planned. Lumelsky [13] initially assumes the environment to be devoid of obstacles and moves the robot directly toward the goal. If an obstacle obstructs the path, the robot moves around the perimeter until the point on the obstacle nearest the goal is found. The robot then proceeds to move directly toward the goal again. Pirzadeh [16] adopts a strategy whereby the robot wanders about the environment until it discovers the goal. The robot repeatedly moves to the adjacent location with lowest cost, and increments the cost of a location each time it visits it to penalize later traverses of the same space. Korf [9] uses initial map information to estimate the cost to the goal for each state and efficiently updates it with backtracking costs as the robot moves through the environment.

While these approaches are complete, they are also suboptimal in the sense that they do not generate the lowest cost path given the sensor information as it is acquired and assuming all known, a priori information is correct. It is possible to generate optimal behavior by computing an optimal path from the known map information, moving the robot along the path until either it reaches the goal or its sensors detect a discrepancy between the map and the environment, updating the map, and then replanning a new optimal path from the robot's current location to the goal. Although this brute-force, replanning approach is optimal, it can be grossly inefficient, particularly in expansive environments where the goal is far away and little map information exists. Zelinsky [22] increases efficiency by using a quad-tree [19] to represent free and obstacle space, thus reducing the number of states to search in the planning space. For natural terrain, however, the map can encode robot traversability at each location ranging over a continuum, thus rendering quad-trees inappropriate or suboptimal.

This paper presents a new algorithm for generating optimal paths for a robot operating with a sensor and a map of the environment. The map can be complete, empty, or contain partial information about the environment. For regions of the environment that are unknown, the map may contain approximate information, stochastic models for occupancy, or even a heuristic estimates. The algorithm is functionally equivalent to the brute-force, optimal replanner, but it is far more efficient.

The algorithm is formulated in terms of an optimal find-path problem within a directed graph, where the arcs are labelled with cost values that can range over a continuum. The robot's sensor is able to measure arc costs in the vicinity of the robot, and the known and estimated arc values comprise the map. Thus, the algorithm can be used for any planning representation, including visibility graphs [11] and grid cell structures.

The paper continues with a description of the algorithm, followed by proofs of its soundness, optimality, and completeness. A number of path planning applications are illustrated. First, optimal paths are generated for a point-sized robot with no map information. More involved problems are then addressed, including planning with robot shape, dead-reckoning error, dynamic environments, occupancy maps, potential fields, natural terrain environments, multiple goals, and multiple robots. The paper concludes with an empirical comparison of the algorithm to the optimal replanner, and the results are summarized.

2.0 The D* Algorithm

The name of the algorithm, D*, was chosen because it resembles A* [14], except that it is *dynamic* in the sense that cost parameters can change during the problem-solving process. Provided that robot motion is properly coupled to the algorithm, D* generates optimal trajectories. This section begins with the definitions and notation used in the algorithm, then presents the D* algorithm, and closes with an illustration of its operation.

2.1 Definitions

The objective of a path planner is to move the robot from some location in the world to a goal location, such that it avoids all obstacles and minimizes a positive cost metric (e.g., length of the traverse). The problem space can be formulated as a set of *states* denoting robot locations connected by *directional arcs*, each of which has an associated cost. The robot starts at a particular state and moves across arcs (incurring the cost of traversal) to other states until it reaches the *goal* state, denoted by G. Every state X except G has a *backpointer* to a next state Y denoted by $b(X) = Y$. D* uses backpointers to represent paths to the goal. The cost of traversing an arc from state Y to state X is a positive number given by the *arc cost* function $c(X, Y)$. If Y does not have an arc to X, $c(X, Y)$ is undefined. The arc cost function can be either directional (i.e., $c(X, Y) \neq c(Y, X)$) or bidirectional (i.e., $c(X, Y) = c(Y, X)$). Two states X and Y are *neighbors* in the space if $c(X, Y)$ or $c(Y, X)$ is defined.

Similar to A*, D* maintains an $OPEN$ list of states. The $OPEN$ list is used to propagate information about changes to the arc cost function and to calculate path costs to states in the space. Every state X has an associated *tag* $t(X)$, such that $t(X) = NEW$ if X has never been on the $OPEN$ list, $t(X) = OPEN$ if X is currently on the $OPEN$ list, and $t(X) = CLOSED$ if X is no longer on the $OPEN$ list. For each state X, D* maintains an estimate of the sum of the arc costs from X to G given by the *path cost* function $h(G, X)$. For the goal state, $h(G, G) = 0$. Under the proper conditions, this estimate is equivalent to the optimal (minimal) cost from state X to G, given by the function $o(G, X)$. For each state X on the $OPEN$ list (i.e., $t(X) = OPEN$), the *previous cost* function, $p(G, X)$, is defined to be equal to $h(G, X)$ before insertion on the $OPEN$ list. Thus, the previous cost function classifies a state X on the $OPEN$ list into one of two types: a $RAISE$ state if $p(G, X) < h(G, X)$, and a $LOWER$ state if $p(G, X) \geq h(G, X)$. D* uses $RAISE$ states on the $OPEN$ list to propagate information about path cost increases (e.g., due to an increased arc cost) and $LOWER$ states to propagate information about path cost reductions (e.g., due to a reduced arc cost or new path to the goal). The propagation takes place through the repeated removal of states from the $OPEN$ list. Each time a state is removed from the list, it is *expanded* to pass cost changes to its neighbors. These neighbors are in turn placed on the $OPEN$ list to continue the process.

States on the $OPEN$ list are sorted by their *key* function value, $k(G, X)$, defined to be $min(h(G, X), p(G, X))$ if $t(X) = OPEN$ and undefined if $t(X) \neq OPEN$. The parameter k_{min} is defined to be $min(k(X))$ for all X such that $t(X) = OPEN$. The parameter k_{min} represents an important threshold in D*: path costs less than or equal to k_{min} are optimal, and those greater than k_{min} may not be optimal. The parameter k_{old} is defined to be equal to k_{min} prior to most recent removal of a state from the $OPEN$ list. If no states have been removed, k_{old} is undefined.

An ordering of states denoted by $\{X_1, X_N\}$ is defined to be a *sequence* if $b(X_{i+1}) = X_i$ for all i such that $1 \leq i < N$ and $X_i \neq X_j$ for all (i,j) such that $1 \leq i < j \leq N$. Thus, a sequence defines a path of backpointers from X_N to X_1. A sequence $\{X_1, X_N\}$ is defined to be *monotonic* if ($t(X_i) = CLOSED$ and $h(G, X_i) < h(G, X_{i+1})$) or ($t(X_i) = OPEN$ and $p(G, X_i) < h(G, X_{i+1})$) for all i such that $1 \leq i < N$. D* constructs and maintains a monotonic sequence $\{G, X\}$, representing decreasing current or previous path costs, for each state X that is or was on the $OPEN$ list. Given a sequence of states $\{X_1, X_N\}$, state X_i is an *ancestor* of state X_j if $1 \leq i < j \leq N$ and a *descendant* of X_j if $1 \leq j < i \leq N$.

For all two-state functions involving the goal state, the following shorthand notation is used: $f(X) \equiv f(G, X)$. Likewise, for sequences the notation $\{X\} \equiv \{G, X\}$ is used. The notation $f(°)$ is used to refer to a function independent of its domain.

2.2 Algorithm Description

The D* algorithm is presented in this section. The algorithm consists primarily of two functions: $PROCESS-STATE$ and $MODIFY-COST$. $PROCESS-STATE$ is used to compute optimal path costs to the goal, and $MODIFY-COST$ is used to change the arc cost function $c(°)$ and enter affected states on the $OPEN$ list. Initially, $t(°)$ is set to NEW for all states, $h(G)$ is set to zero, and G is placed on the $OPEN$ list. The first routine, $PROCESS-STATE$, is repeatedly called until the robot's state, X, is removed from the $OPEN$ list (i.e., $t(X) = CLOSED$) or a value of -1 is returned, at which point either the sequence $\{X\}$ has been constructed or does not exist respectively. The robot then proceeds to follow the backpointers in the sequence $\{X\}$ until it either reaches the goal or discovers an error in the arc cost function $c(°)$ (e.g., due to a detected obstacle). The second routine, $MODIFY-COST$, is immediately called to correct $c(°)$ and place affected states on the $OPEN$ list. Let Y be the robot's state at which it discovers an error in $c(°)$. By calling $PROCESS-STATE$ until it returns $k_{min} \geq h(Y)$, the cost changes are propagated to state Y such that $h(Y) = o(Y)$. At this point, a possibly new sequence $\{Y\}$ has been constructed, and the robot continues to follow the backpointers in the sequence toward the goal.

The algorithms for $PROCESS-STATE$ and $MODIFY-COST$ are presented below. The embedded routines are $MIN-STATE$, which returns the state on the $OPEN$ list with minimum $k(°)$ value ($NULL$ if the list is empty); $GET-KMIN$, which returns k_{min} for the $OPEN$ list (-1 if the list is empty); $DELETE(X)$, which deletes state X from the $OPEN$ list and sets $t(X) = CLOSED$; and $INSERT(X)$, which sets $t(X) = OPEN$, computes $k(X)$ from $h(X)$ and $p(X)$, and places or re-positions state X on the $OPEN$ list sorted by $k(°)$.

In function $PROCESS-STATE$ at lines L1 through L4, the state X with the lowest $k(°)$ value is removed from the $OPEN$ list. Before X increases or reduces the path cost of its neighbors, it first checks if any of its neighbors can reduce its own path cost at lines L5 through L11. Note that the check is limited to $CLOSED$ neighbors with optimal $h(°)$ values (i.e., less than or equal to the old k_{min}). At lines L12 through L57 each neighbor of X is examined again. All neighbors that receive a new path cost are placed on the $OPEN$ so that they will propagate the cost change to their neighbors. At lines L15 through L20, the path cost is computed from a NEW neighbor Y to the goal. The backpointer is set to X so that the monotonic sequence $\{Y\}$ is constructed. At lines L23 through L32, all neighbor states Y that have a backpointer to X receive a new path cost, regardless of whether the new cost is greater than or less than the old. Since these states are descendants of X, any change to the path cost of X affects their path costs as well. At lines L36 through L46, state X reduces the path cost of its neighbors (if possible) that are not immediate descendants of X and redirects their backpointers to point to X. Note that this reduction is permitted only if X is a $LOWER$ state. It is shown in the next section that this requirement is essential to avoid creating a closed loop in the backpointers. If X is a $RAISE$ state, it is placed back on the $OPEN$ list for future expansion. At lines L49 through L53, the neighbors Y of X that are able to reduce the path cost of X are placed on the $OPEN$ list. Since these neighbor states have path costs greater than the old k_{min}, their path costs are not guaranteed to be optimal. Thus, the updating is "postponed" until the neighbors are selected for expansion, at which time they will be optimal. Finally, at line L59 the current k_{min} is returned.

Function: cost:PROCESS-STATE ()

```
L1   $X = MIN-STATE(\ )$
L2   if $X = NULL$ then return $-1$
L3   $k_{old} = GET-KMIN(\ )$
L4   $DELETE(X)$
L5   # Reduce $h(X)$ by lowest-cost neighbor if possible
L6   for each neighbor $Y$ of $X$:
L7       if $t(Y) = CLOSED$ and $h(Y) \leq k_{old}$ and $h(X) > h(Y) + c(Y, X)$ then
L8           $b(X) = Y$
L9           $h(X) = h(Y) + c(Y, X)$
L10      endif
L11  endforeach
```

```
L12  # Process each neighbor of $X$
L13  for each neighbor $Y$ of $X$:
L14      # Propagate cost to $NEW$ state
L15      if $t(Y) = NEW$ then
L16          $b(Y) = X$
L17          $h(Y) = h(X) + c(X, Y)$
L18          $p(Y) = h(Y)$
L19          $INSERT(Y)$
L20      endif
L21      else
L22          # Propagate cost change along backpointer
L23          if $b(Y) = X$ and $h(Y) \neq h(X) + c(X, Y)$ then
L24              if $t(Y) = OPEN$ then
L25                  if $h(Y) < p(Y)$ then $p(Y) = h(Y)$
L26                  $h(Y) = h(X) + c(X, Y)$
L27              endif
L28              else
L29                  $h(Y) = h(X) + c(X, Y)$
L30                  $p(Y) = h(Y)$
L31              endelse
L32              $INSERT(Y)$
L33          endif
L34          else
L35              # Reduce cost of neighbor if possible
L36              if $b(Y) \neq X$ and $h(Y) > h(X) + c(X, Y)$ then
L37                  if $p(X) \geq h(X)$ then
L38                      $b(Y) = X$
L39                      $h(Y) = h(X) + c(X, Y)$
L40                      if $t(Y) = CLOSED$ then $p(Y) = h(Y)$
L41                      $INSERT(Y)$
L42                  endif
L43                  else
L44                      $p(X) = h(X)$
L45                      $INSERT(X)$
L46                  endelse
L47              else
L48                  # Set up cost reduction by neighbor if possible
L49                  if $b(Y) \neq X$ and $h(X) > h(Y) + c(Y, X)$ and
L50                      $t(Y) = CLOSED$ and $h(Y) > k_{old}$ then
L51                      $p(Y) = h(Y)$
L52                      $INSERT(Y)$
L53                  endif
L54              endelse
```

```
L55             endelse
L56         endelse
L57     endforeach
L58     # Return $k_{min}$
L59     return $GET-KMIN()$
L60     endfunction
```

In function $MODIFY-COST$ at line L2, the arc cost function is updated with the changed value. Since the path cost for state Y will change, X is placed on the $OPEN$ list. When X is expanded via $PROCESS-STATE$, it computes a new $h(Y) = h(X) + c(X, Y)$ and places Y on the $OPEN$ list. Additional state expansions propagate the cost to the descendants of Y.

Function: cost:MODIFY-COST (state: X, state: Y, cost: cval)

```
L1      # Change the arc cost value
L2      $c(X, Y) = cval$
L3      # Insert state $X$ on the $OPEN$ list if it is $CLOSED$
L4      if $t(X) = CLOSED$ then
L5          $p(X) = h(X)$
L6          $INSERT(X)$
L7      endif
L8      # Return $k_{min}$
L9      return $GET-KMIN()$
L10     endfunction
```

2.3 Illustration of Operation

The role of $RAISE$ and $LOWER$ states is central to the operation of the algorithm. The $RAISE$ states (i.e., $h(X) > p(X)$) propagate cost increases, and the $LOWER$ states (i.e., $h(X) \le p(X)$) propagate cost reductions. When the cost of traversing an arc is increased, an affected neighbor state is placed on the $OPEN$ list, and the cost increase is propagated via $RAISE$ states through all state sequences containing the arc. As the $RAISE$ states come in contact with neighboring states of lower cost, these $LOWER$ states are placed on the $OPEN$ list, and they subsequently decrease the cost of previously raised states where ever possible. If the cost of traversing an arc is decreased, the cost decrease is propagated via $LOWER$ states through all state sequences containing the arc, as well as neighboring states whose cost can also be lowered.

Figure 1 through Figure 7 illustrate the operation of the algorithm for a "potential well" path planning problem. The planning space consists of a 50 x 50 grid of *cells*. Each cell represents a state and is connected to its eight neighbors via bidirectional arcs. The arc cost values are small for the free cells and prohibitively large for the obstacle cells. The robot is point-sized and is equipped with a contact sensor. Figure 1 shows the results of an optimal path calculation from the goal to all states in the planning space. The two grey obstacles are stored in the map, but the black obstacle is not. The arrows depict the backpointer function; thus, an optimal path to the goal for any state can be obtained by tracing the arrows from the state to the goal. Note that the arrows deflect around the grey, known obstacles but pass through the black, unknown obstacle.

In Figure 2, the robot follows the backpointers from its starting location at the center of the left wall toward the goal. Its path is depicted by the horizontal black line. When it reaches the unknown black obstacle, it detects a discrepancy between the map and the world, updates the map, and enters the state on the $OPEN$ list via $MODIFY-COST$. The cell is changed to grey, indicating that it is now known to be an obstacle. $PROCESS-STATE$ is called to compute a new path to the robot. The state containing the detected part of the obstacle becomes a $RAISE$ state that passes the path cost increase to its descendants upon expansion. The *RAISE* states place the upper neighbor of the detected obstacle on the *OPEN* list as a *LOWER* state. This neighbor redirects the robot cell's backpointer up and to the right. When the robot attempts to move in this direction, its sensor discovers that this new cell is also an obstacle. Successive calls to *MODIFY – COST* and *PROCESS – STATE* lead the robot up along the unknown obstacle (see Figure 3).The dark grey cells in the center of the well are *RAISE* states, and the light grey cells are *LOWER* states. Note that the *LOWER* states have directed the backpointers in the upper half of the well to point toward the upper portion of the unknown obstacle.

When the robot reaches the top of the unknown obstacle, it discovers that the "gap" is sealed (Figure 4). *RAISE* states propagate this information through the upper half of the well, and then *LOWER* states expand from the remaining, lower portion of the unknown obstacle to redirect backpointers toward the lower half of the well. The robot then moves down along the obstacle to the lower portion of the well and discovers that the entire gap between the two original map obstacles is sealed (Figure 5). Thus, the path cost to all cells in the well is increased, and this information is propagated via *RAISE* states which expand out of the well to the left.

As the *RAISE* states move out of the well, they activate *LOWER* states around the lip which proceed to sweep into the well around the upper and lower obstacles (Figure 6) and redirect the backpointers out of the well. This process is complete when the *LOWER* states reach the robot's cell, at which point the robot moves around the lower obstacle to the goal (Figure 7). Note that after the traverse, the backpointers are only partially updated. Backpointers within the well point outward, but those in the left half of the planning space still point into the well. All states have a path to the goal, but optimal paths are computed to a limited number of states. This effect illustrates the efficiency of D*. The backpointer updates needed to guarantee an optimal path for the robot are limited to the vicinity of the obstacle.

Figure 1: Backpointers Based on Initial Propagation from Goal State

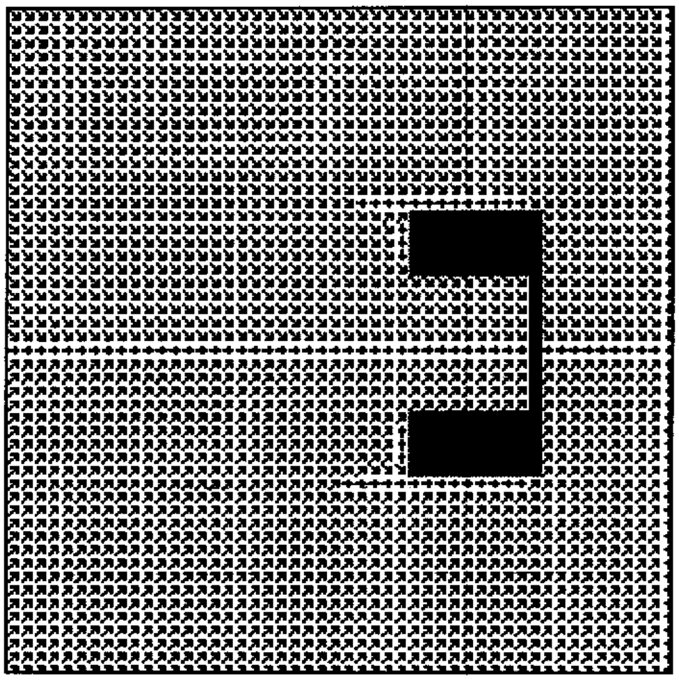


Figure 2: Robot Discovers First Unknown Obstacle Cell

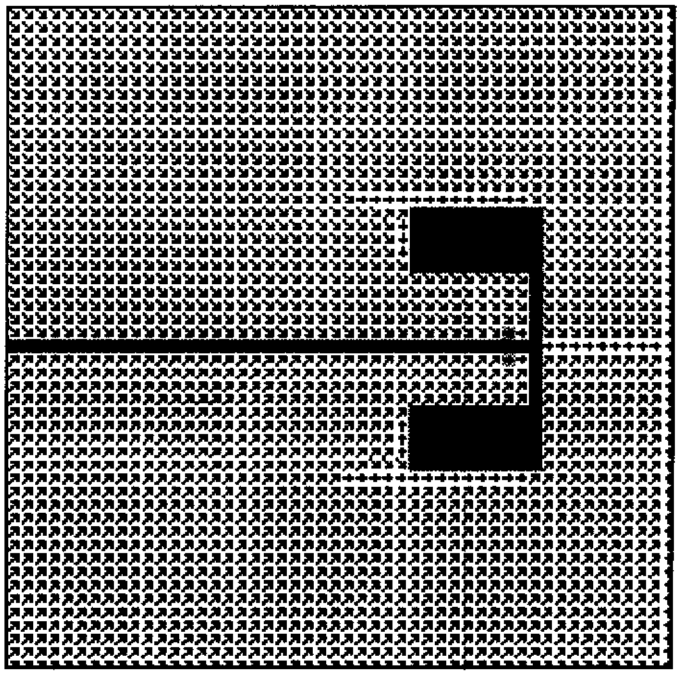


Figure 3: Robot Moves Up in Search of Path around Obstacle

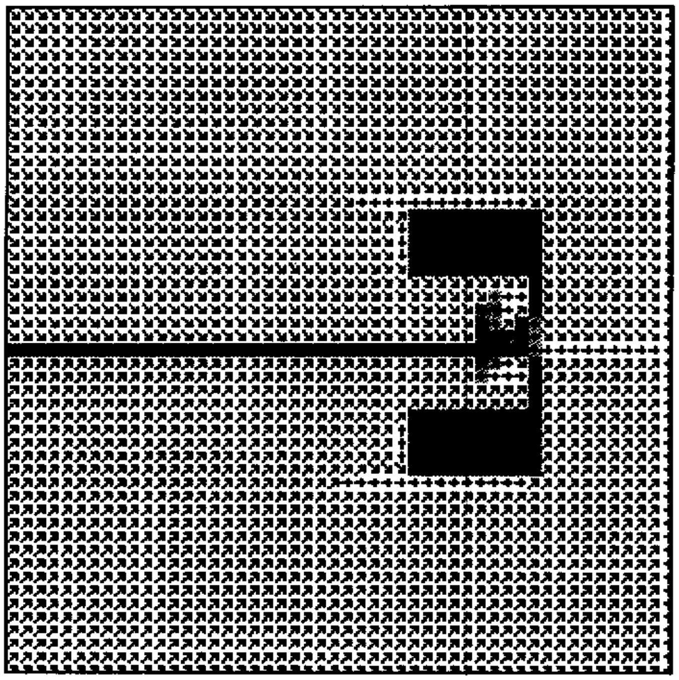


Figure 4: Robot Moves Down in Search of Path around Obstacle

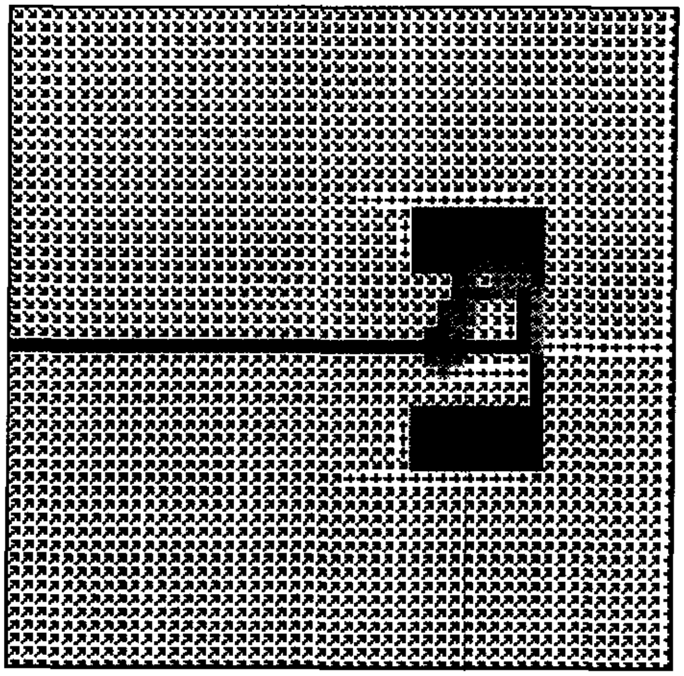


Figure 5: RAISE States Propagate out of Well

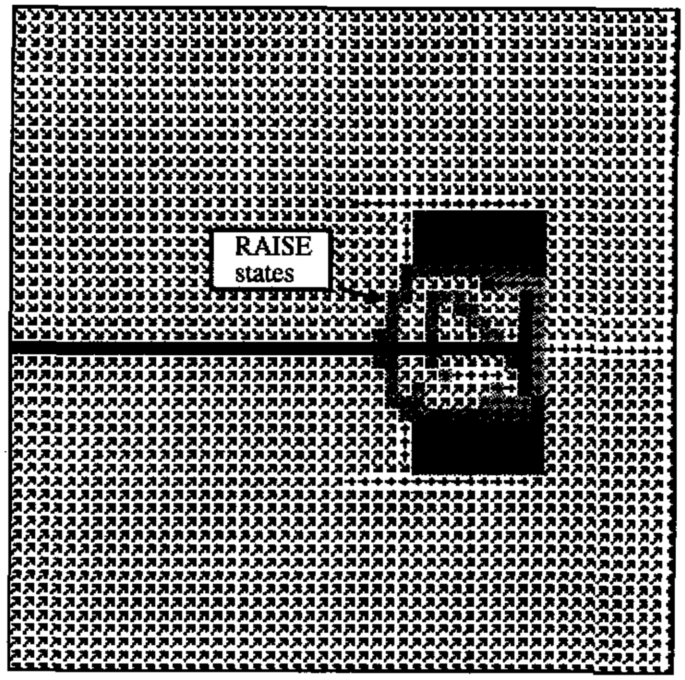


Figure 6: LOWER States Sweep into Well

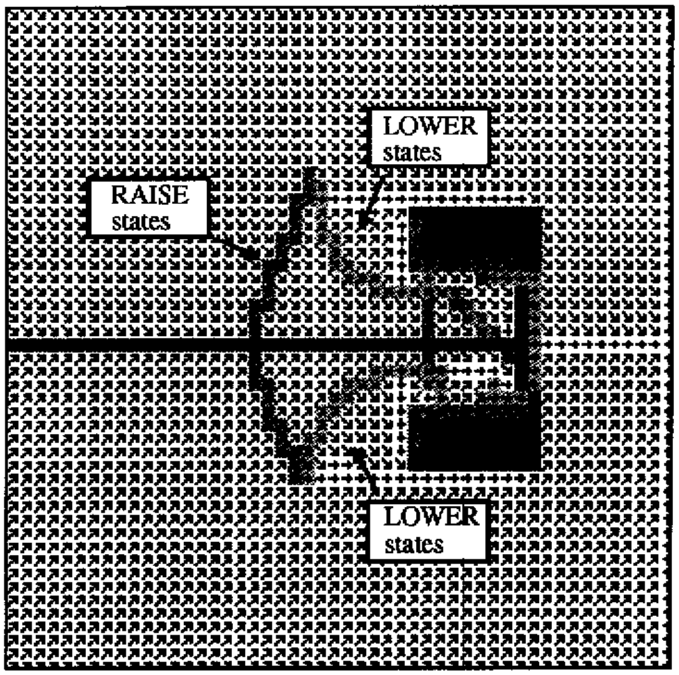
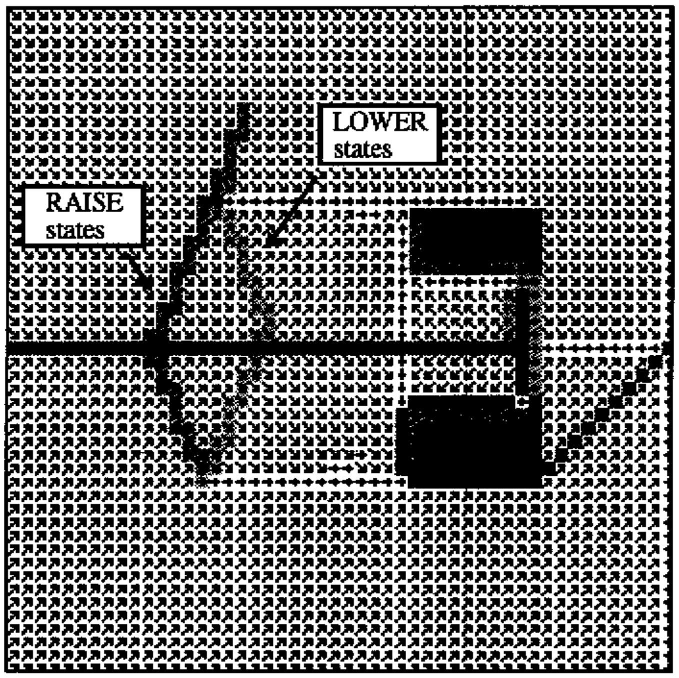


Figure 7: Final Backpointer Configuration after Robot Reaches Goal



3.0 Proofs of Soundness, Optimality, and Completeness

After all states X have been initialized to $t(X) = NEW$ and G has been entered onto the $OPEN$ list, the function $PROCESS-STATE$ is repeatedly invoked to construct state sequences. The function $MODIFY-COST$ is invoked to make changes to $c(°)$ and to seed these changes on the $OPEN$ list. In this section, D* is shown to have the following properties:

Property 1: If $t(X) \neq NEW$, then a sequence $\{X\}$ is constructed and is monotonic.

Property 2: When the value k_{min} returned by $PROCESS-STATE$ equals or exceeds $h(X)$, then $h(X) = o(X)$.

Property 3: If a path from X to G exists, and the search space contains a finite number of states, then $\{X\}$ will be constructed after a finite number of calls to $PROCESS-STATE$. If a path does not exist, then $PROCESS-STATE$ will return -1 with $t(X) = NEW$.

Property 1 is a soundness property: once a state has been visited, a finite sequence of backpointers to the goal has been constructed. Property 2 is an optimality property. It defines the conditions under which the chain of backpointers to the goal is optimal. Property 3 is a completeness property: if a path from X to G exists, it will be constructed. If no path exists, it will be reported in a finite amount of time. All three properties hold regardless of the pattern of invocation for functions $MODIFY-COST$ and $PROCESS-STATE$.

In order to prove that D* has the above properties, a number of theorems are needed to lay the groundwork. As D* propagates costs, it redirects backpointers in such a way as to create new and optimal state sequences. The first group of theorems defines the conditions under which backpointers can be modified and still preserve Property 1. Theorem 1 establishes the uniqueness of a state sequence.

Theorem 1: If each state has only one backpointer, only one sequence $\{X_1,X_N\}$ can exist between any two states X_1 and X_N.

Proof: Let $\{Y_1,Y_M\}$ be another sequence from X_1 to X_N. Clearly, Y_M must be X_N since the sequences must end at the same state. Y_{M-1} must be X_{N-1} unless Y_M or X_N has two backpointers. By induction, the sequences must be identical down to and including state X_1. If $\{X_1,X_N\}$ and $\{Y_1,Y_M\}$ are of different lengths, then either the longer sequence does not end at X_1 or it contains multiple copies of X_1, thus violating the definition of a sequence. QED.

Theorem 2 defines the condition under which a state can be cut off from the goal. If the backpointer of an ancestor of state X is redirected to point to X or one of its descendants, then a cycle in the backpointers is introduced in sequences that contain X.

Theorem 2: Assume that Property 1 holds for a given set of states. Given the sequence $\{G,X\}$, let y be the set of states, such that $Y \in y$ if the sequence $\{X,Y\}$ exists. If $b(X)$ is set to some state Y_s in y, then no sequence $\{G,Y\}$ exists for a state Y iff $Y \in y$.

Proof: If $Y_s \in y$, then any sequence $\{G,Y_s\}$ must contain the subsequence $\{G,X\}$ (by the definition of y and Theorem 1). But if $b(X)$ is redirected to point to a member of y, then $\{G,X\}$ must contain X again. This violates the definition of a sequence. Therefore, all states $Y \in y$ will not have valid sequences. Since only $b(X)$ is redirected, any states which are not members of y are unaffected, and Property 1 still holds for these states. QED.

One way to avoid a cycle when redirecting the backpointer of state Y to point to X is to determine whether or not Y is an ancestor of X. Theorem 3 establishes a necessary condition for this determination by capitalizing on the monotonicity of state sequences under Property 1.

Theorem 3: Assume that Property 1 holds for a given set of states. Given a monotonic sequence $\{X_1,X_N\}$ and a state Z with $h(Z) > h(X_N)$, Z cannot be a member of $\{X_1,X_N\}$ unless $\{X_1,X_N\}$ contains at least one *RAISE* state.

Proof: Assume Z is a member and $\{X_1,X_N\}$ contains no *RAISE* states. For any pair of adjacent states X_i and X_{i+1} in the sequence, if $t(X_i) = CLOSED$, then $h(X_i) < h(X_{i+1})$ by definition. If $t(X_i) = OPEN$, then $p(X_i) < h(X_{i+1})$. But since X_i must be a *LOWER* state, then $h(X_i) \le p(X_i)$, and therefore $h(X_i) < h(X_{i+1})$. Therefore, by induction and transitivity of the inequality relation, $h(X_i) < h(X_N)$ for all i such that $1 \le i < N$. So by contradiction, either Z is not a member or the sequence contains a *RAISE* state. QED.

The following theorem strengthens the condition on the *RAISE* state for determining state ancestry by establishing a relational test on its $k(°)$ value.

Theorem 4: Assume the sequence $\{X_1,X_N\}$ is monotonic. If any of the elements X_1 through X_{N-1} are *RAISE* states, then for at least one of these *RAISE* states X_S, $k(X_S) < h(X_N)$.

Proof: Assume that $\{X_1,X_{N-1}\}$ contains at least one *RAISE* state, and let X_S be the *RAISE* state with the largest index. Therefore, the subsequence $\{X_{S+1},X_N\}$ contains no *RAISE* states. By the definition of a monotonic sequence, $h(X_i) < h(X_{i+1})$ for all i such that $S < i < N$. Therefore, by induction and transitivity of the inequality relation, $h(X_{S+1}) \le h(X_N)$. (Equality of the $h(°)$ values occurs when $S+1 = N$.) Since X_S is a *RAISE* state, then $k(X_S) = p(X_S) < h(X_{S+1})$; thus, $k(X_S) < h(X_N)$. QED.

Therefore, by redirecting backpointers only to states with $h(°)$ less than or equal to the minimum $k(°)$ on the *OPEN* list, it is impossible to create a backpointer cycle. This is formalized in Theorem 5.

Theorem 5: Assume that Property 1 holds for a given set of states. Let X be a state such that $h(X) \le k_{min}$. Let X_i be a neighbor state of X such that $c(X, X_i)$ is defined and $h(X) < h(X_i)$. If $b(X_i)$ is set to X, then Property 1 is preserved.

Proof: Consider the two cases: 1) X_i is not a member of the sequence $\{G,X\}$; and 2) X_i is a member. Case 1: Since X is not a descendant of X_i, then sequences exist for all states with $t(°) \ne NEW$ after the backpointer is redirected (Theorem 2). Since the sequence $\{G,X\}$ is monotonic, $h(X) < h(X_i)$, and all sequences beginning with X_i are monotonic (Property 1), then all resultant sequences are monotonic. Case 2: Since $h(X)$ is less than or equal to the minimum $k(°)$ value on the *OPEN* list, then from Theorem 4, no members of the sequence $\{G,X\}$ can be *RAISE* states. But from Theorem 3, if no members are *RAISE* states, then X_i cannot be a member of $\{G,X\}$ unless $h(X) \ge h(X_i)$. But, $h(X) < h(X_i)$, so case 2 cannot exist. QED.

In addition to theorems governing the modification of $b(°)$, a theorem governing modifications to $h(°)$ is needed to preserve monotonicity of the sequences as required under Property 1.

Theorem 6: Assume that Property 1 holds for a given set of states. Consider two states, X and Y, such that $b(Y) = X$. The following modifications can be made to $h(Y)$ while still preserving Property 1. If $t(Y) = OPEN$, then $h(Y)$ can be modified to assume any value provided that $h(Y) > p(X)$ if $t(X) = OPEN$ and $h(Y) > h(X)$ if $t(X) = CLOSED$. If $t(Y) = CLOSED$, then $h(Y)$ can be adjusted if it satisfies the same lower bound constraints and is also not increased.

Proof: Let y be the set of states such that $Y_i \in y$ iff $b(Y_i) = Y$. Consider the case where $t(Y) = OPEN$. From the definition of a monotonic sequence, $p(Y)$ must be less than $h(Y_i)$ for all Y_i in y. Modifying $h(Y)$ does not affect $p(Y)$, so the condition still holds. From the definition of a monotonic sequence, the value of $h(Y)$ must be greater than $p(X)$ if X is *OPEN* and greater than $h(X)$ if X is *CLOSED*, but these conditions are stated in the theorem. Consider the case where

$t(Y) = CLOSED$. From the definition of a monotonic sequence, $h(Y) \le h(Y_i)$ for all Y_i in y. Decreasing $h(Y)$ preserves this condition, provided $h(Y)$ satisfies the lower bound constraints stated above for the same reasons. Thus, all sequences remain monotonic and Property 1 holds. QED.

A corollary can be derived for modifications to $p(°)$.

Corollary 6: Assume that Property 1 holds for a set of states. Given a state X such that $t(X) = OPEN$, if $p(X)$ is reduced then is Property 1 preserved.

Proof: See the proof to Theorem 6. Since $p(X)$ has an upper bound but no lower bound, it can be reduced and Property 1 still holds. QED.

A similar theorem governing modifications to $t(°)$ is needed to preserve sequence monotonicity under Property 1.

Theorem 7: Assume that Property 1 holds for a given set of states. If $t(X)$ is changed from $OPEN$ to $CLOSED$, Property 1 is preserved if $h(X) \le p(X)$. If $t(X)$ is changed from $CLOSED$ to $OPEN$, Property 1 is preserved if $p(X) \le h(X)$.

Proof: Consider the case where $t(X)$ is set to $CLOSED$. Let x be the set of states such that $X_i \in x$ iff $b(X_i) = X$. Before closure, $p(X)$ must be less than $h(X_i)$ for all X_i in x, and after closure, $h(X)$ must be less than $h(X_i)$ for all X_i (definition of a monotonic sequence). Thus, if $h(X) \le p(X)$, then $h(X) \le p(X) < h(X_i)$ for all X_i in x and the theorem holds. Consider the case where $t(X)$ is set to $OPEN$. If $p(X) \le h(X)$, then $p(X) \le h(X) < h(X_i)$ for all X_i in x and the theorem holds. QED.

The supporting theorems are now in place to prove that D* preserves Property 1 in all cases.

Theorem 8: D* preserves Property 1.

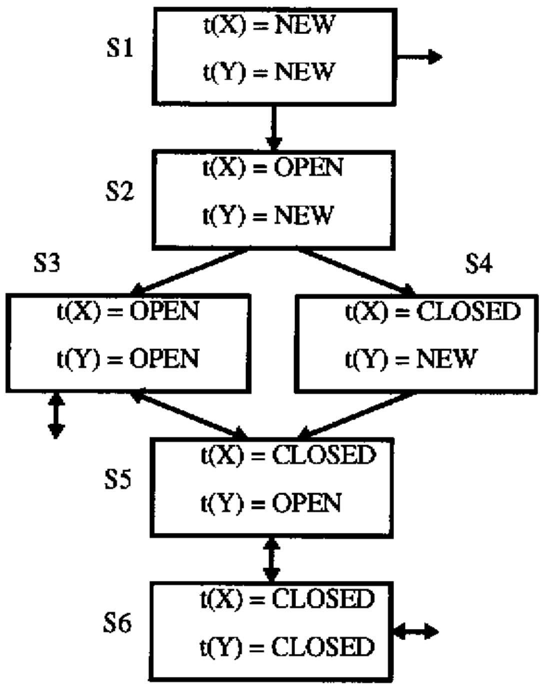
Proof: The portions of $PROCESS-STATE$ and $MODIFY-COST$ that modify the functions $b(°)$, $h(°)$, $p(°)$, and $t(°)$ need to be examined. In $PROCESS-STATE$ at line L4, X is removed from the $OPEN$ list. It is shown below that Property 1 is preserved by modifying the $h(°)$ values of states with backpointers to X. At line L8, $b(X)$ is redirected. Since $h(Y) \le k_{old} \le k_{min}$ and $h(Y) < h(X)$, then by Theorem 5 Property 1 is preserved. At line L9, $h(X)$ is modified. From the conditional, $h(X)$ is modified only if it can be decreased, and after modification, $h(X) > h(Y)$. Since both X and Y are $CLOSED$, from Theorem 6 Property 1 is preserved. At line L16, $b(Y)$ is assigned. Since $t(Y)$ is NEW, it was not previously part of a sequence, and after the assignment of $b(Y)$, Y is the last state in the sequence. At line L17, $h(Y)$ is assigned such that $h(Y) > h(X)$, thus preserving Property 1. At lines L18 and L19, Y is inserted onto the $OPEN$ list. Since no other states have backpointers to Y, then $p(Y)$ can assume any value, and Property 1 is preserved. At lines L26 and L29, $h(Y)$ is modified so that it is greater than $h(X)$ for all Y such that $b(Y) = X$, thus restoring monotonicity possibly lost from the removal of X from the $OPEN$ list at line L4. At line L25, $p(Y)$ is reduced. From Corollary 6, Property 1 is preserved. At line L32, Y is inserted or repositioned on the $OPEN$ list. If $t(Y)$ is changed to $OPEN$, then since $p(Y) = h(Y)$, from Theorem 7 Property 1 holds. At line L38, $b(Y)$ is redirected. Since $h(X) = k_{old} \le k_{min}$ and $h(X) < h(Y)$, then by Theorem 5 Property 1 is preserved. At line L39, since $h(Y)$ is reduced, $h(Y) > h(X)$, and X is either $CLOSED$ or $OPEN$ with $p(X) = h(X)$ (lines L44 and L45), then from Theorem 6 Property 1 is preserved. At line L41, Y is inserted or repositioned on the $OPEN$ list. For the same reason as line L32, Property 1 holds. At lines L44 and L45, X is placed back on the $OPEN$ list. Since $p(X) = h(X)$, then from Theorem 7 Property 1 is preserved. For the same reason, Property 1 is preserved when Y is placed back on the $OPEN$ list at lines L51 and L52.

The function $MODIFY-COST$ affects $t(°)$ but does not modify $b(°)$, $h(°)$, or $p(°)$. Note: $p(°)$ is assigned but not changed. At line L6, the $CLOSED$ state X is placed on the $OPEN$ list. Since $p(X)$ is set to $h(X)$, then from Theorem 7 Property 1 is preserved. QED.

Thus, Theorem 8 proves that D* will not "cut off" a state X from the goal once the sequence $\{G,X\}$ has been constructed. The sequence may be modified later, but at all times it is possible to trace backpointers from X to G. Thus, D* generates only sound sequences. This property is important for moving the robot toward the goal before information has been fully propagated through the set of states.

The optimality of D* is shown next, as stated in Property 2. The first theorem addresses the relationship between states on and off the *OPEN* list. Figure 8 illustrates the possible combinations and transitions for the $t(°)$ values of a pair of neighbor states X and Y.

Figure 8: Possible transitions for states X and Y



After the states X and Y are initialized, they are both labelled *NEW*, as shown in box S1. Since a state tag cannot be set to *NEW* again after initialization, box S1 cannot be re-entered (as shown by the arrows). The dangling arrow points to isomorphic copies (not shown) of boxes S2 through S6 with X and Y swapped. Box S2 represents the placement of X on the *OPEN* list, after which Y can be opened (S3) or X can be closed (S4). The two states can transition between S3, S5, and S6 as the states are individually inserted and deleted from the *OPEN* list. Again, the dangling pointers from S3 and S6 point to a box isomorphic to S5.

Using this state transition diagram, a theorem about the relationship between cost values for states in box S5 is proved below.

Theorem 9: Given any two states, X and Y, such that $c(X, Y)$ is defined, $t(X) = CLOSED$, and $t(Y) = OPEN$, then $k(Y) \le h(X) + c(X, Y)$.

Proof: Initially, X and Y are *NEW* states as shown in box S1. Since X and Y can be opened by different states, no assumptions are made about their $h(°)$ and $k(°)$ values in the transition through box S2 to S3. In the transition from S2 through S4 to S5, X is *CLOSED* and Y is placed on the *OPEN* list. This transition occurs at lines L15 through L19 in *PROCESS – STATE*. In this segment, $k(Y) = h(Y)$ is set to $h(X) + c(X, Y)$, thus the theorem holds. In the transition from S3 to S5, X is

CLOSED and Y remains *OPEN*. Consider the two cases: 1) $b(Y) = X$; and 2) $b(Y) \neq X$. Case 1: At lines L23 through L32, $h(Y)$ is set to $h(X) + c(X, Y)$ and $k(Y) \leq h(Y)$; thus, the theorem holds. Case 2: Consider two subcases: 2a) X was a *LOWER* state; and 2b) X was a *RAISE* state. Case 2a: At lines L37 through L41, if $h(Y) \leq h(X) + c(X, Y)$, then since $k(Y) \leq h(Y)$ and no action is taken the theorem holds. If $h(Y) > h(X) + c(X, Y)$, then $h(Y)$ is set equal to $h(X) + c(X, Y)$. Since $k(Y) \leq h(Y)$, the theorem holds. Case 2b: If $h(Y) > h(X) + c(X, Y)$, then X is placed back on the *OPEN* list at lines L44 and L45, and X and Y are transitioned back to box S3. In box S5, it is possible for a state other than X to modify $h(Y)$. Since reducing $h(Y)$ can only reduce $k(Y)$, only those modifications that increase $h(Y)$ are of concern (lines L25 and L26). In this segment, $p(Y)$ is reassigned in order to prevent $k(Y)$ from increasing, and the theorem holds.

In the transition from S5 to S6, consider three cases: 1) an immediate transition is made within *PROCESS – STATE* from S6 back to S5; 2) an immediate transition is made from S6 to the box isomorphic to S5; and 3) the states remain in S6. Case 1: At lines L44 and L45, Y is placed back on the *OPEN* list with $k(Y) = h(Y)$ after a possible reduction of $h(Y)$ at lines L7 through L9. Since $k(Y)$ can only be reduced, $k(Y) \leq h(X) + c(X, Y)$ and the theorem holds. Case 2: Y is closed and X is opened. At lines L29 through L32 and L36 through L41, $k(X)$ is set to $h(Y) + c(Y, X)$ and the theorem holds. At lines L49 through L52, $k(X)$ is set to $h(X)$. Since $h(Y) > h(X) + c(X, Y)$, then $k(X) = h(X) < h(Y) - c(X, Y) < h(Y) + c(Y, X)$ and the theorem holds. Case 3: The only segment of *PROCESS – STATE* that leaves both states *CLOSED* is lines L7 through L9. If $h(Y)$ exceeds $h(X)$ by more than $c(X, Y)$, then $h(Y)$ is set to $h(X) + c(X, Y)$; thus, $h(Y) \leq h(X) + c(X, Y)$. Note also that $h(X) \leq h(Y) + c(Y, X)$; otherwise, X or Y will be placed on the *OPEN* list at lines L23 through L52, and the states will transition out of S6. In the transition from S6 back to S5, Y is placed back on the *OPEN* list, then $k(Y) = h(Y) \leq h(X) + c(X, Y)$ and the theorem holds. If the transition is made from S6 to the isomorphic box to S5, then $k(X) = h(X) \leq h(Y) + c(Y, X)$ and the theorem holds.

This last case is important for analyzing the effects of function *MODIFY – COST*. This function is able to change a state from *CLOSED* to *OPEN*. Since box S4 is only a temporary transition within *PROCESS – STATE*, it cannot be affected by *MODIFY – COST*. *MODIFY – COST* can effect transitions from S5 to S3, but the theorem states nothing about these transitions. The only applicable transitions are S6 to S5 and the box isomorphic to S5. Since $h(Y) \leq h(X) + c(X, Y)$ for states in S6 (see above), if Y is placed on the *OPEN* list, then $k(Y) \leq h(X) + c(X, Y)$ and the theorem holds. This operation occurs at lines L5 and L6 of *MODIFY – COST*. The same reasoning applies to the transition from S6 to the isomorphic box to S5. QED.

Theorem 9 is very powerful, because it proves that *CLOSED* neighbors of an *OPEN* state cannot reduce the $k(°)$ value of the *OPEN* state. The following corollary derived from the proof for the previous theorem describes the relationship between $h(°)$ values for a neighboring pair of *CLOSED* states.

Corollary 9: Given two states X and Y such that $t(X) = t(Y) = CLOSED$, if $c(Y, X)$ is defined, then $h(X) \leq h(Y) + c(Y, X)$, and if $c(X, Y)$ is defined, then $h(Y) \leq h(X) + c(X, Y)$.

Proof: See the proof for Theorem 9. QED.

From Theorem 9, the monotonicity of the parameter k_{min} is proved in the theorem below.

Theorem 10: Between calls to *MODIFY – COST*, the parameter k_{min} increases or remains at the same value a finite number of times with each invocation of *PROCESS – STATE*.

Proof: Let X be the next state to be removed from the *OPEN* list; therefore, $k(X) = k_{min}$. It will be shown that X can only insert or reposition states on the *OPEN* list to have $k(°)$ values greater than $k(X)$. Since there are a finite number of states on the *OPEN* list with $k(°)$ values equal to $k(X)$, then k_{min} must increase or remain the same for a finite number of iterations. At lines L7 through L9 in *PROCESS – STATE*, $h(X)$ can be reduced. Since Y is *CLOSED*, the value of $h(X)$ cannot be reduced below $k(X)$ (Theorem 9). At lines L15 through L19, Y is inserted onto the $OPEN$ list. Since $k(Y) = h(Y) = h(X) + c(X, Y) > h(X) \geq k(X)$, the theorem holds. At lines L23 through L32, Y is inserted or repositioned on the $OPEN$ list. Consider the case where Y is already on the $OPEN$ list. Reassigning the value of $h(Y)$ can only reduce $k(Y)$ or leave it unmodified. If $k(Y)$ is reduced, then $k(Y) = h(Y) = h(X) + c(X, Y) > h(X) \geq k(X)$ and the theorem holds. If the value of $k(Y)$ is unmodified, then either it was already greater than $k(X)$ or was equal to it, and the theorem holds. Consider the case where Y is inserted onto the $OPEN$ list. For the same reasons as lines L15 through L19, the theorem holds.

At lines L37 through L41, Y is inserted, repositioned, or left in the same place on the $OPEN$ list. Since $p(X) \geq h(X)$, then $h(X) = k(X)$. Since after modification $h(Y)$ is greater than $h(X)$, if Y is inserted onto the $OPEN$ list, then $k(Y) = h(Y) > h(X) = k(X)$ and the theorem holds. If Y is already on the $OPEN$ list, then either $p(Y)$ is less than the modified $h(Y)$ and Y is not repositioned, or $p(Y)$ is greater than or equal to $h(Y)$, and $k(Y) = h(Y)$; thus, the theorem holds for the above reasons. At lines L44 and L45, X is re-inserted on the $OPEN$ list. Since $p(X) < h(X)$ before re-insertion and $p(X) = h(X)$ after re-insertion, then $k(X)$ must increase and the theorem holds. At lines L49 through L52, Y is inserted onto the $OPEN$ list. Since $k(Y) = h(Y) > k_{old} = k(X)$, the theorem holds. QED.

Now that the monotonicity of k_{min} has been established, the inductive step for optimality can be constructed. It is shown below that if states with $h(°)$ values less than or equal to k_{min} at the i-th invocation of $PROCESS-STATE$ are optimal, then states with $h(°)$ values less than or equal to the new k_{min} at the (i+1)-st invocation of $PROCESS-STATE$ are optimal.

Theorem 11: If $h(X) = o(X)$ for all states X such that $h(X) \leq k_{old}$, then $h(Y) = o(Y)$ for all states Y such that $k_{old} \leq h(Y) \leq k_{min}$.

Proof: A state Y_i, such that $k_{old} \leq h(Y_i) \leq k_{min}$ must be either 1) $CLOSED$ or 2) $OPEN$ with $h(Y_i) = k_{min}$. Order the states Y_i by $h(°)$ value such that $k_{old} \leq h(Y_1) \leq h(Y_2) \leq \ldots \leq h(Y_N) \leq k_{min}$. Consider Y_1 first. Case 1: From the premise, $h(Z) = o(Z)$ for all neighbor states Z (i.e., $c(Z, Y_1)$ is defined) such that $h(Z) < h(Y_1)$. From Corollary 9, none of these optimal neighbor states can reduce $h(Y_1)$. Nor can any neighbor states on the $OPEN$ list reduce $h(Y_1)$, since the $h(°)$ values for states on the $OPEN$ list are greater than or equal to k_{min}. These $OPEN$ states cannot reduce each other's $h(°)$ values below k_{min}, nor can their $CLOSED$ neighbors (Theorem 9). Thus, since $h(Y_1) \leq k_{min}$, then $h(Y_1) = o(Y_1)$. Case 2: Since $h(Y_1) = k(Y_1)$, then from Theorem 9, a neighbor state X with $h(X) = o(X) < h(Y_1)$ cannot reduce $h(Y_1)$. Thus, $h(Y_1) = o(Y_1)$. The above argument can be repeated for the remaining states in order, and by induction, $h(Y_i) = o(Y_i)$ for $i = 2$ through $i = N$. QED.

The monotonicity of k_{min} established in Theorem 10 combined with the inductive step established in Theorem 11 is used below to prove the optimality of D* as given in Property 2, regardless of the calling sequence of $PROCESS-STATE$ and $MODIFY-COST$.

Theorem 12: D* preserves Property 2.

Proof: Initially, the goal state G is placed on the $OPEN$ list with $h(G) = o(G) = 0$. There are no $CLOSED$ states since $t(°) = NEW$ for all states except G. When $PROCESS-STATE$ removes and expands a state from the $OPEN$ list, k_{old} is set to k_{min} and k_{min} is increased monotonically (Theorem 10). From Theorem 11, for the states X such that $k_{old} \leq h(X) \leq k_{min}$, $h(X) = o(X)$; therefore, $h(X) = o(X)$ if $h(X) \leq k_{min}$.

Assume that $c(X, Y)$ is modified, and the change is seeded via $MODIFY-COST$. If X is $OPEN$, then the change to $c(X, Y)$ cannot affect optimality of states with $h(°)$ less than or equal to k_{min}, since $h(X)$ can be reduced no lower than k_{min} (Theorem 9). Since $h(X) + c(X, Y)$ must be greater than k_{min}, then $h(°)$ values equal to or lower than k_{min} cannot be reduced; thus, the premise to Theorem 11 is still true and $PROCESS-STATE$ can be invoked to propagate the modification. If X is

CLOSED, then *MODIFY−COST* ensures that $k_{min} \le h(X)$ by placing X on the *OPEN* list. This operation preserves the truth of the premise to Theorem 11, since reducing k_{min} selects a subset of the optimal *CLOSED* states. Thus, Property 2 is preserved regardless of the pattern of cost modification and propagation. QED.

The theorem below shows that D* finds a monotonic sequence to any state X that is reachable from the goal.

Theorem 13: D* preserves Property 3.

Proof: Assume that a state X_N is reachable from the goal (i.e., $\{X_1,X_N\}$ exists, where $G = X_1$). Initially, G is placed on the *OPEN* list. When state X_i is expanded, it places or repositions X_{i+1} on the *OPEN* list and redirects its backpointer if necessary (lines L15 through L19, L23 through L32, and L37 through L41 in *PROCESS−STATE*). By induction, the sequence $\{X_1,X_N\}$ will be constructed unless some state X_S exists in the sequence $\{X_1,X_N\}$ such that $t(X_S) = OPEN$, and X_S is never selected for expansion. Once a state X_S has been placed on the *OPEN* list, its $k(°)$ value cannot be increased, since all modifications to $h(°)$ for *OPEN* states are reductions. Lines L24 through L26 are the exception, where $p(°)$ is reassigned to prevent $k(°)$ from increasing. Thus, due to the monotonicity of k_{min} (Theorem 10), X_S will be eventually removed from the *OPEN* list.

If a state X_N is unreachable from the goal, eventually all reachable states will be placed on the *OPEN* list (via lines L15 through L19 in *PROCESS−STATE*) and will be removed given the above reasoning. Since the number of search states is finite, the list will empty after a finite number of calls to *PROCESS−STATE*, and -1 will be returned with $t(X_N) = NEW$. QED.

The significance of the results in this section is that when *PROCESS−STATE* visits a state X (i.e., $t(X) \ne NEW$), it constructs a sequence $\{X\}$ to the goal. Thereafter, a sequence is maintained regardless of subsequent arc cost modifications and propagation. If *PROCESS−STATE* returns -1 before X is visited, then no path exists from X to the goal. If *PROCESS−STATE* is invoked repeatedly until k_{min} is greater than or equal to $h(X)$, then the optimal sequence $\{X\}$ to the goal has been found. The possible uses of D* for mobile robot path planning are discussed in the next section.

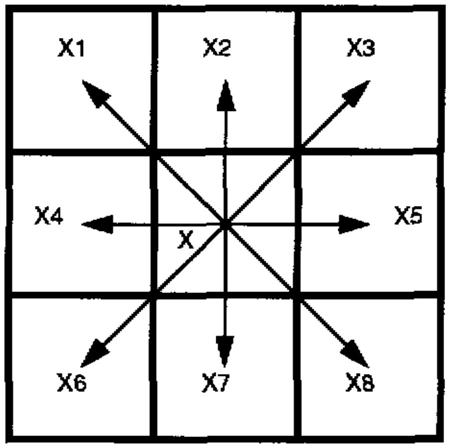
4.0 Motion Planning Applications

In this section, D* is used to solve a number of problems, beginning with a simple indoor mobile robot path planning problem and then extending to more difficult and relevant planning problems. All of the examples in this section were generated using an implementation of D*.

4.1 Simple Path Planning

For all of the path planning problems in this section, the environment model is an 8-connected cartesian lattice or grid of cells as shown in Figure 9. Note that D* is not limited to this representation; instead, it requires only that the planning space be a graph of states. A state is defined to be the center of a cell in the lattice. For a given state X, $c(X_i, X)$ is defined for $i = 1$ to 8. Thus, it is possible to move from state X to neighboring states whose cells share an edge or corner in the lattice. The arc cost of moving from X to X_i is defined to be equal to the cost of moving from X_i to X; thus, $c(X, X_i) = c(X_i, X)$. Let function $s(X)$ be the *presumed cost* of traversing the width of the cell containing state X, and let function $a(X)$ be the *actual cost* of traversing the cell. From the figure, $c(X_i, X) = s(X)/2 + s(X_i)/2$ if X and X_i share an edge, and $c(X_i, X) = (s(X) + s(X_i))\ (\sqrt{2}/2)$ if X and X_i share a corner.

Figure 9: Cell Lattice for Path Planning



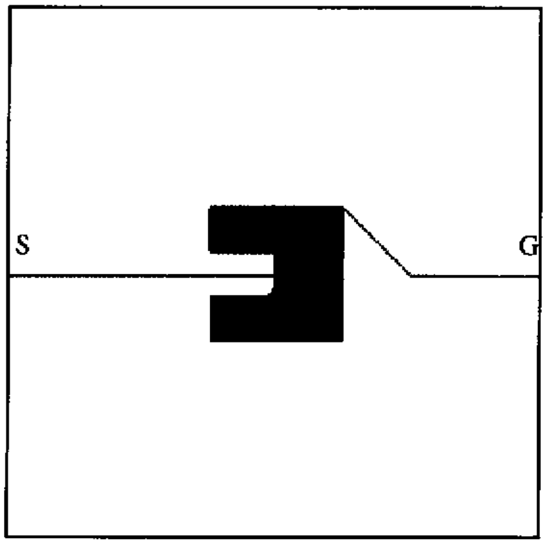
For the simple path planning problem, the following assumptions are made about the robot and its environment:

- **Omnidirectionality:** the robot is capable of moving from state Y to any state X for which $c(X, Y)$ is defined. For the above environment model, the robot can move in any of the eight directions.
- **Point Size:** the robot has zero physical extent; thus, its workspace equals its configuration space.
- **Minimal Field of View (FOV):** the robot is equipped with a sensor capable of measuring the $a(\circ)$ value of the cell in which it resides or that of a neighboring cell.
- **Deterministic Motion:** the robot accurately follows planned paths so that "safety buffers" around obstacles are not needed.
- **Static Environment:** the $a(\circ)$ values of the cells are static (i.e., they do not change in time).
- **Binary Obstacle Representation:** the $s(\circ)$ and $a(\circ)$ values for a given cell can take on one of two values: $EMPTY$, a small positive value representing the cost of traversing a cell free of obstacles; and $OBSTACLE$, a large positive value indicating the cell contains an obstacle and is untraversable. Note: these values must be chosen so that the path cost for any sequence of states through $EMPTY$ cells is less than $OBSTACLE$.
- **No Initial Map Information:** nothing is known about the environment initially, and it is presumed that $s(X) = EMPTY$ for all states X.
- **Single Goal State:** only one state has an $h(\circ)$ value of zero.
- **Single Robot:** only one robot moves through the environment.

Let S be the state in which the robot begins. To use D*, the goal state is placed on the *OPEN* list with $k(G) = h(G) = 0$, and *PROCESS – STATE* is called repeatedly until $h(S)$ is less than or equal to k_{min}. At this point an optimal path has been computed from S to G. (Since all cells are presumed *EMPTY*, this path is direct.) The robot proceeds to step toward the goal, following the backpointers in the state sequence, until either it reaches the goal or its sensor detects that $s(X) \neq a(X)$ for a state X. In the latter case, the robot has detected an obstacle, and $s(X)$ is set to $a(X)$, $c(X, °)$ and $c(°, X)$ are updated for all neighboring states, and the changes are entered onto the *OPEN* list via function *MODIFY – COST*. The routine *PROCESS – STATE* is repeatedly called until k_{min} equals or exceeds the $h(°)$ value of the state containing the robot. At this point, a new optimal path has been computed, and the above process repeats until the goal is reached or until k_{min} equals or exceeds *OBSTACLE*. In the latter case, the optimal path contains an obstacle; therefore, no obstacle-free paths exist to the goal.

Figure 10 illustrates simple motion planning with an unknown potential well. Unless otherwise stated, all environments in this section are 200 x 200 cells. Unlike the example in Figure 1 through Figure 7, the robot assumes the environment is devoid of obstacles when it begins its traverse. Initially, the robot moves straight toward the goal. When it encounters the obstacle's perimeter, the robot moves down in an attempt to find an opening through the barrier, edges up slightly along the lower wall as it detects it, and then doubles back up before finally moving along an interior wall and around the exterior to the goal. The *OBSTACLE* cells detected by the robot's sensor are shown in grey, and the unsensed cells are shown in black.

Figure 10: Simple Path Planning with an Unknown Potential Well

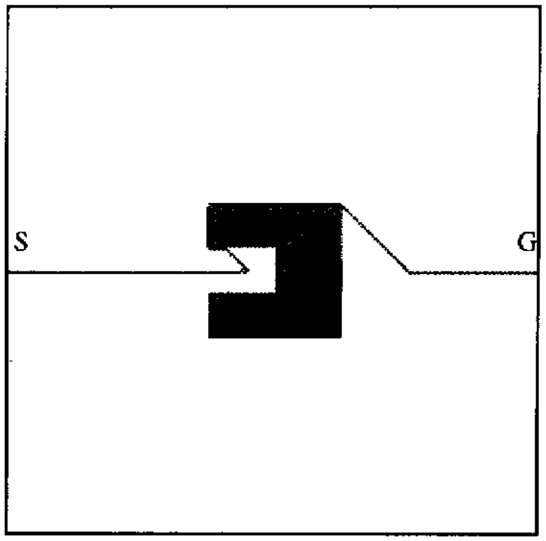


4.2 Field of View Considerations

Except for contact sensors, most robot sensors (e.g., sonars, laser rangefinders, video cameras) have a field of view (FOV) covering an area in the vicinity of the robot. D* can easily accommodate a sensor with a FOV of any size. When the robot is moved from state X to state Y and new sensor data is taken, $s(Z)$ is compared to $a(Z)$ for all states Z in the FOV. If $s(Z) \neq a(Z)$ for any state in the FOV, $s(Z)$ is set to $a(Z)$, $c(Z, °)$ and $c(°, Z)$ are revised via *MODIFY – COST*, and *PROCESS – STATE* is repeatedly called until $k_{min} \geq h(Y)$. Thus, the only revision to the simple path planning algorithm is that potentially more than one cell's cost is updated per sensor reading.

Figure 11 shows the same path planning problem as Figure 10, except that the robot is equipped with a circular-FOV sensor with a radius of 15 cells. For ease of implementation, the FOV was chosen to include all cells in the circle without occlusion. Since the robot can see the "bottom" of the well before reaching it, it changes course and follows the exterior perimeter. In general, the larger the robot's FOV, the shorter the path it will traverse, since it can see obstacles before it "bumps" them and will begin to avoid them sooner.

Figure 11: Path Planning with a Circular Field of View

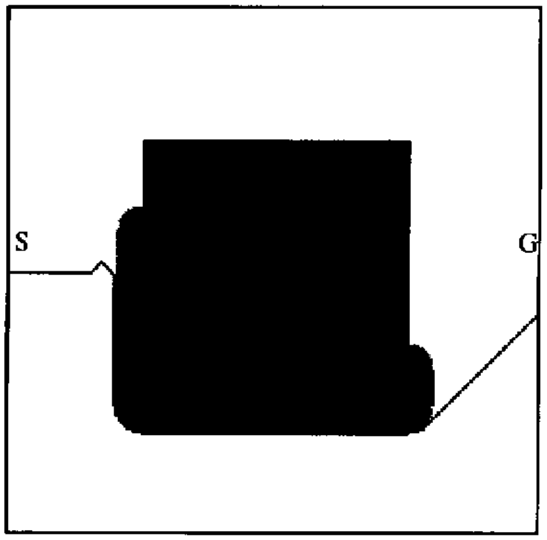


4.3 Shape Considerations

Real robots have non-zero size; thus, a path planning algorithm must ultimately be able to model robot shape. D* can handle robot shape by constructing the configuration space [11] as the robot's sensor discovers obstacles in the environment. Define three functions: $a_o(X, Y)$, the *actual C-space obstacle cost* of $EMPTY$ or $OBSTACLE$ at robot configuration Y due to an obstacle at location X; $a_c(Y)$, the *actual C-space total cost* for configuration Y equal to $OBSTACLE$ if any $a_o(X, Y) = OBSTACLE$ for applicable X; and $s_c(Y)$, the *presumed C-space total cost* for configuration Y. Whenever the sensor discovers a discrepancy between the map and the environment (i.e., $s(X) \neq a(X)$), $s(X)$ is set to $a(X)$, and $a_c(Y)$ is computed using $a_o(X, Y)$ for all applicable Y. For each configuration Y for which $s_c(Y) \neq a_c(Y)$, $s_c(Y)$ is set to $a_c(Y)$, $MODIFY-COST$ is called to update $c(°)$ using $s_c(°)$ rather than $s(°)$, and $PROCESS-STATE$ is called repeatedly until a new optimal path is found.

Figure 12 illustrates the results for a disc-shaped robot. In this example, the configuration space and the workspace are both two-dimensional, so the two spaces are shown in the same figure. The black area represents an unknown obstacle in the workspace (i.e., $a(°)$ values), and the grey represents the known C-space obstacle (i.e., $s_c(°)$ values) for cells exterior to the black square and the known workspace obstacle (i.e., $s(°)$ values) for cells interior to the square. The field of view of the robot is 20 cells, and the radius of its disc shape is 10 cells. Note that the configuration space is generated as the robot discovers the unknown obstacle, and it moves around the obstacle at a stand-off distance equal to its radius. Of course, the field of view of the robot must be at least as great as the robot's radius or collisions cannot be avoided. This approach can be extended to higher-dimensional configuration spaces (e.g., three dimensional for translation and rotation of a polygonal robot) by increasing the dimensionality of the planning space.

Figure 12: Path Planning with a Disc-Shaped Robot



4.4 Dynamic Environments and Dead-Reckoning Error

In many cases, it is inappropriate to assume the robot's world is static [2][3][4][7]. Obstacles may move around within the environment, new obstacles may enter, and old ones may leave. These changes may occur while the robot is enroute to the goal. D* is capable of handling dynamic environments. Changes in the environment are detected by the robot's sensor. These sensor readings modify the function $a(°)$. Whenever a discrepancy is seen between the actual world and the presumed world (i.e., $a(°) \neq s(°)$ for at least one state), the changes are entered on the *OPEN* list and the robot's trajectory is modified if needed to preserve optimality. These changes can be of two types: 1) an *EMPTY* cell becomes an *OBSTACLE*; and 2) an *OBSTACLE* cell becomes *EMPTY*.

Figure 13 shows an environment in which the obstacle has moved without the robot's knowledge. The grey rectangle shows the original, presumed position, and the black rectangle shows the current, unknown position. Figure 14 shows the trajectory of the robot through the environment. Believing the obstacle to be in the grey position, the robot initially deflects toward the bottom. When it reaches the center of the environment, it encounters the obstacle in its current location. It moves along the obstacle toward the lower boundary of the environment, entering the new location of the obstacle into the map and "coloring" it grey. After discovering the lower route is obstructed, it doubles back to the top whereupon it discovers there is no obstacle in the original location. The obstacle is deleted from the map as the robot moves around the top of the obstruction and to the goal. Note that part of the original obstacle remains in the map, since this portion was unseen by the robot's sensor.

In the presence of inaccurate map data, it is possible for the robot to mistakenly believe that no path exists to the goal. This condition is detected by the function *PROCESS – STATE* returning a value equal to or greater than *OBSTACLE*, thus indicating that the shortest path to the goal passes through at least one *OBSTACLE* state. This condition should not be confused with a returned value of '-1', indicating that no sequence of arcs of even *OBSTACLE* cost exists to the goal. In the event that a value of *OBSTACLE* or greater is returned, one strategy is to set $s(°)$ to *EMPTY* for all states in the map, enter the goal state on the *OPEN* list, invoke *PROCESS – STATE* repeatedly until k_{min} is greater than or equal to $h(X)$, where X is the current state of the robot, and then move the robot. This action has the effect of discounting all map information and forcing the robot to verify that the obstructions still exist or discover that they do not.

Figure 13: Before and After Positions of a Dynamic Obstacle

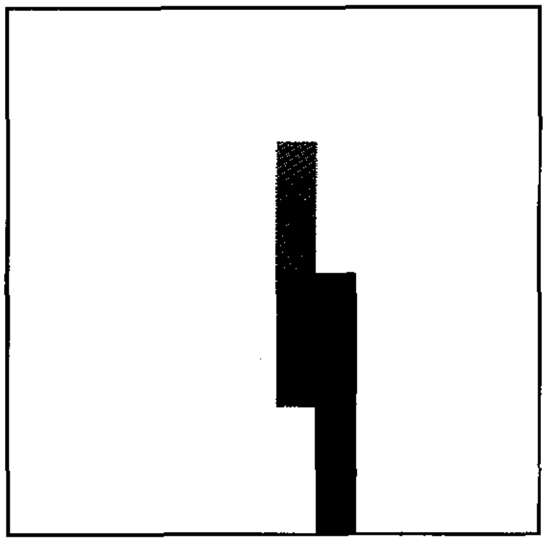
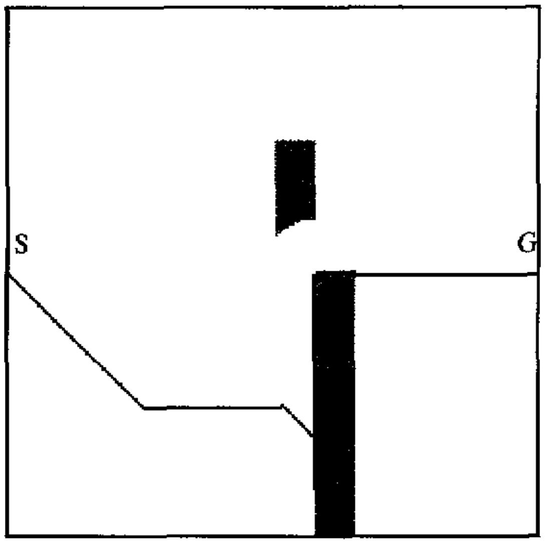


Figure 14: Discovering that the Obstacle has Moved



D* can also handle uncertainty in the robot's motion (e.g., dead-reckoning error). If the robot is located at state *X* but its position estimation system reports that it is at *Y*, its sensor will perceive (incorrectly) discrepancies between the map and the real world (e.g., shift in an obstacle's location). The discrepancies will be entered onto the *OPEN* list via *MODIFY – COST* and propagated via *PROCESS – STATE* to compute a new trajectory. This new trajectory will correctly avoid the obstacles in the current FOV at their "new" locations. Provided the motion error is small compared to the size of the free-space corridors in the environment, the robot will plan a correct local trajectory (i.e., for obstacle avoidance) without adversely affecting its global trajectory (i.e., route to the goal).

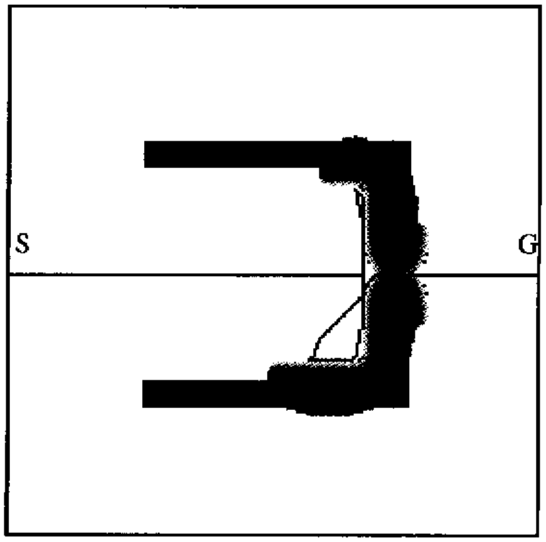
4.5 Occupancy Maps and Potential Fields

In some applications, a binary obstacle representation is inadequate. Due to sensor uncertainty, the robot cannot always determine whether a cell is occupied or empty but can assign a probability of occupancy[1] [20]. Other applications may call for a safety buffer around the obstacles to minimize the probability of collision. This technique can be used to account for robot motion error. A potential field [8] can be constructed by assigning high cost values to cells containing an obstacle or near an obstacle and lower cost values that decrease to zero to cells farther away.

D* is capable of handling continuous values for $a(°)$, $s(°)$, and consequently $c(°)$. Define three functions: $a_t(Y)$, the *actual total potential* at state Y; $s_t(Y)$, the *presumed total potential* at state Y; and $a_p(X, Y)$, the *actual obstacle potential* at state Y due to an obstacle at state X. In the example below, for each state X for which $s(X) \neq a(X)$, $s(X)$ is set to $a(X)$, $a_t(Y)$ is computed for each affected Y by summing $a_p(X, Y)$ over all applicable X. For each state Y for which $s_t(Y) \neq a_t(Y)$, $s_t(Y)$ is set to $a_t(Y)$, $MODIFY-COST$ is called to update $c(°)$ using $s_t(°)$ rather than $s(°)$, and $PROCESS-STATE$ is called repeatedly until a new optimal path is found.

Figure 15 illustrates path planning with potential fields. The robot has no knowledge of the obstacles before it begins its traverse; thus, it can construct the potential field only when an obstacle appears in its field of view. The sensor's field of view is 10 cells, and the potential field decreases proportionally to $min(MAXCOST, 1/r^2)$, where $MAXCOST$ is a maximum cost value and r is the distance from the obstacle. The two L-shaped unknown obstacles (black) are changed to light grey as the robot's sensor detects them. The grey "blur" along the obstacle edges is the potential field created from the detected portion of the obstacles. Initially, the robot presumes the environment is $EMPTY$ and heads directly toward the goal. It is repelled by the potential field in the narrow channel and moves along the boundary of the upper obstacle. Once it has moved sufficiently far from the goal, it doubles back along the boundary of the lower obstacle until the estimated cost of the longer path around the obstacle exceeds that of the more direct route through the potential field. Thus, the planner chooses a shorter path through a riskier area (i.e., close to obstacles) rather than a longer, safer one and heads for the goal.

Figure 15: Path Planning with Potential Fields



4.6 Map Information and Outdoor Navigation

So far in this section, the planning applications have assumed little or no a priori map information. Map information is useful because it can guide the robot around obstacles or clear of impasses long before they appear in its sensor's field of view. Even incomplete or approximate information is often more beneficial than no map information. In general, the better the presumed map information approximates the actual world (i.e., $s(°) \approx a(°)$), the lower the cost of the trajectory driven by the robot. A priori data, or map information, can assume a number of forms. Three possibilities are listed below:

- **Dense Resolution:** each cell is assigned an $s(°)$ value corresponding to a separate measurement from a dense set. An example of this type of map is a surveyed room or field. The resolution of the map data is commensurate with the survey data.

- **Coarse Resolution:** each cell is assigned the $s(°)$ value of a sparse measurement taken near the cell or an interpolated value between measurements. An example of this type of map is 100-meter elevation data recorded aerially that is interpolated to fill a 1-meter resolution map.
- **Feature Data:** cells corresponding to large or significant features in the world (e.g., hills, lakes, buildings, roads) are labelled appropriately, and the rest of the cells are presumed to be *EMPTY*.

Maps for outdoor navigation typically record cost information for difficulty of traverse. For example, steep hills have high $s(°)$ values since extra fuel must be expended to propel the robot. Likewise, pothole-ridden terrain has high $s(°)$ values since a bumpy ride is undesirable. Extremely steep terrain and stumps, boulders, and other large objects have prohibitively high $s(°)$ values, since these terrain features are essentially obstacles and cannot be traversed at any cost. Paved roads have low $s(°)$ values and are preferred. D* is capable of representing terrain costs since $s(°)$, and therefore $c(°)$, can represent a continuum of values.

Figure 16 shows path planning across fractally-generated natural terrain using a complete, dense map of the terrain. The environment is 450 x 450 cells, and the robot's field of view is 20 cells. The start state is the lower left corner, and the goal state is the upper right corner. Black regions are obstacles and cannot be traversed at any cost. The grey scales represent a continuum of cost values such that dark grey regions are five times more difficult to traverse than white regions. Since the map is complete, $a(°) \equiv s(°)$, and the complete and final path can be planned to the goal before the robot begins its traverse. The cost of the path is 40,426. This path is referred to as *omniscient optimal*, since it is the lowest-cost path given complete and accurate a priori map information.

Figure 16: Path Planning across Natural Terrain with a Complete Map

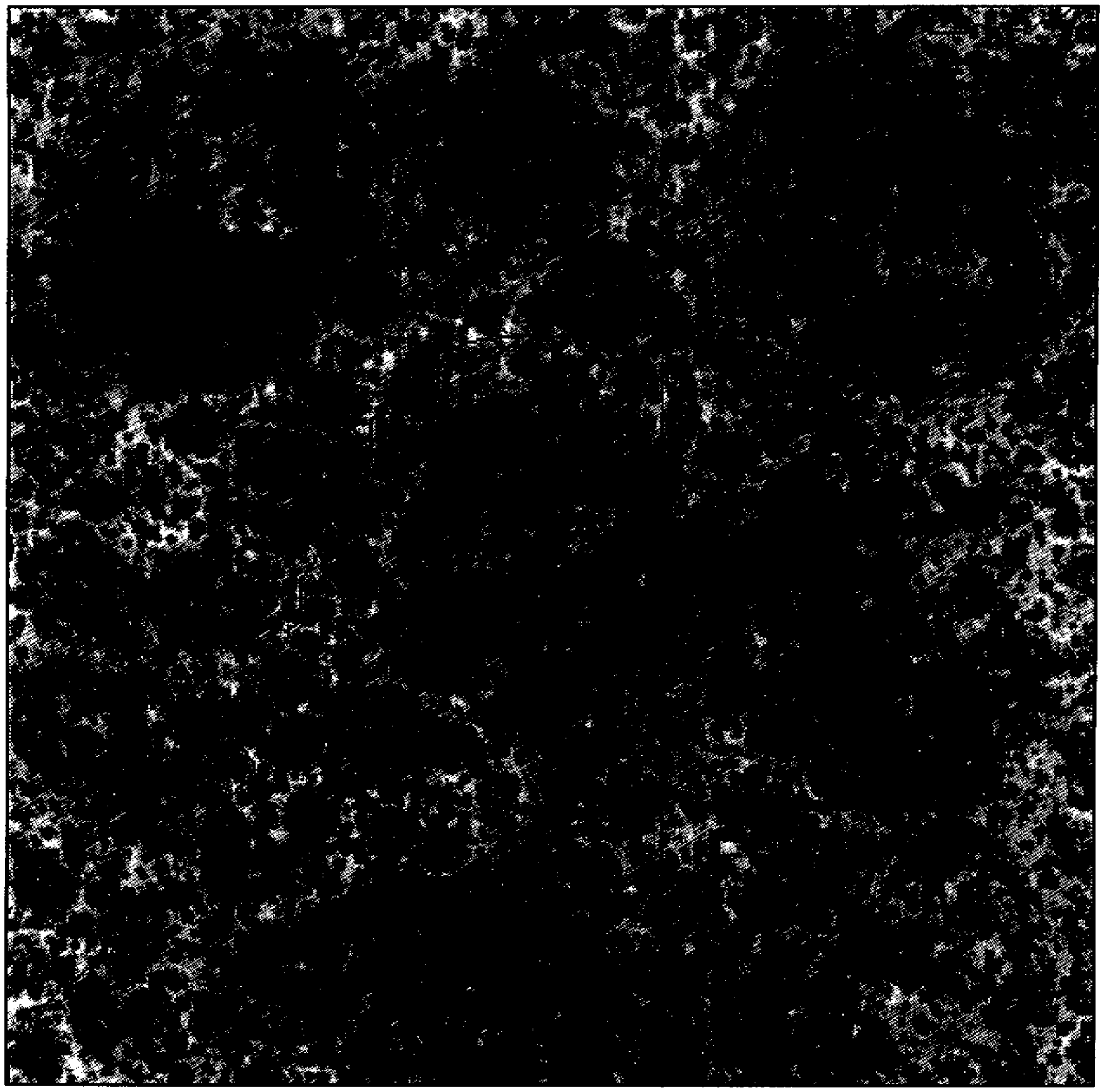
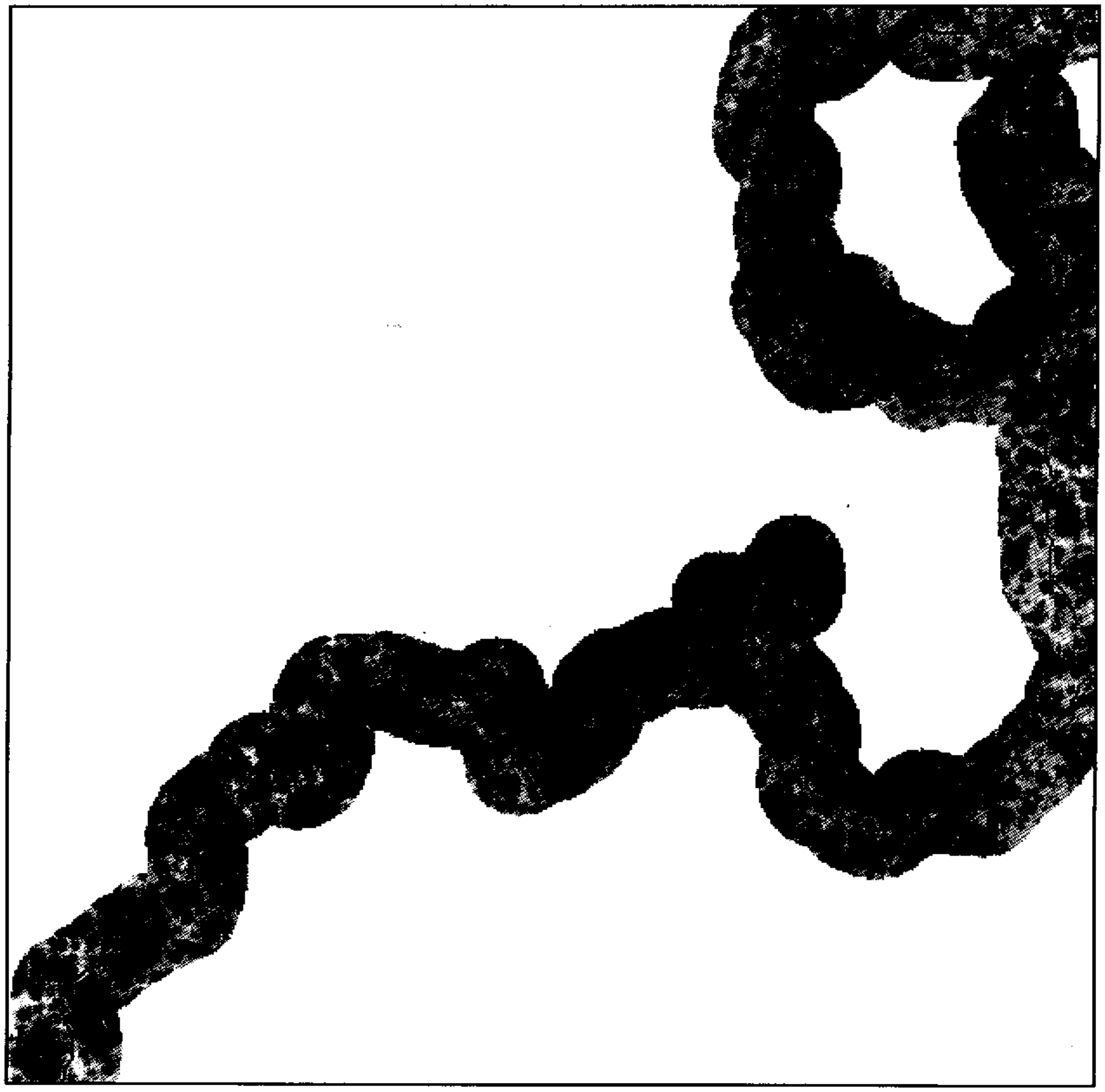


Figure 17 illustrates path planning in the same terrain with no a priori map information. In this case, the optimistic assumption is made that the environment consists only of the lowest-cost (white) cells. Initially, the robot heads directly for the goal and optimizes its path "locally" within its field of view. Unfortunately, given the lack of a priori information the robot chooses to go to the right of the large black obstacle region upon its first encounter and finds itself moving around the long side looking for an opening in the direction of the goal. After wandering into a dead end, the robot backtracks around the last obstacle and finds the goal. The cost of the traversed path is 107,608: over twice that of the omniscient optimal path. Even though the path is of higher cost than omniscient optimal, it is still optimal given the information the robot had when it acquired it.

Figure 17: Path Planning Across Natural Terrain without a Map



These two examples illustrate opposite ends of a continuous spectrum, and for most applications they are unrealistic. In general, some a priori map information is available. Figure 18 illustrates planning over the same terrain with coarse map information, perhaps measured from a satellite or aircraft. In this example, the coarse map was created by averaging the $a(°)$ values in each coarse cell (i.e., square region) and writing the average into $s(°)$ for each dense cell in the region. Each coarse cell is 56 x 56 dense cells. Therefore, $s(X) \neq a(X)$ for most X, but for the most part, $s(°)$ is an approximation to $a(°)$. The map information is accurate enough to properly guide the robot around the large obstructions, and the resultant path is "globally" similar to that in Figure 16 barring some "local" variations. The cost of the path in Figure 18 is 42,882. Thus, it overshoots the omniscient-optimal path by 6% in cost. Figure 19 is similar to Figure 18 except that a higher-resolution map is used (i.e., coarse cells = 7 x 7 dense cells). The cost of this path is 41,079, and it overshoots the omniscient-optimal path by 1.6%. From this set of examples, it can be concluded that the more accurate the prior map data, the lower the cost of the traverse.

Figure 18: Path Planning with a Coarse-Resolution Map

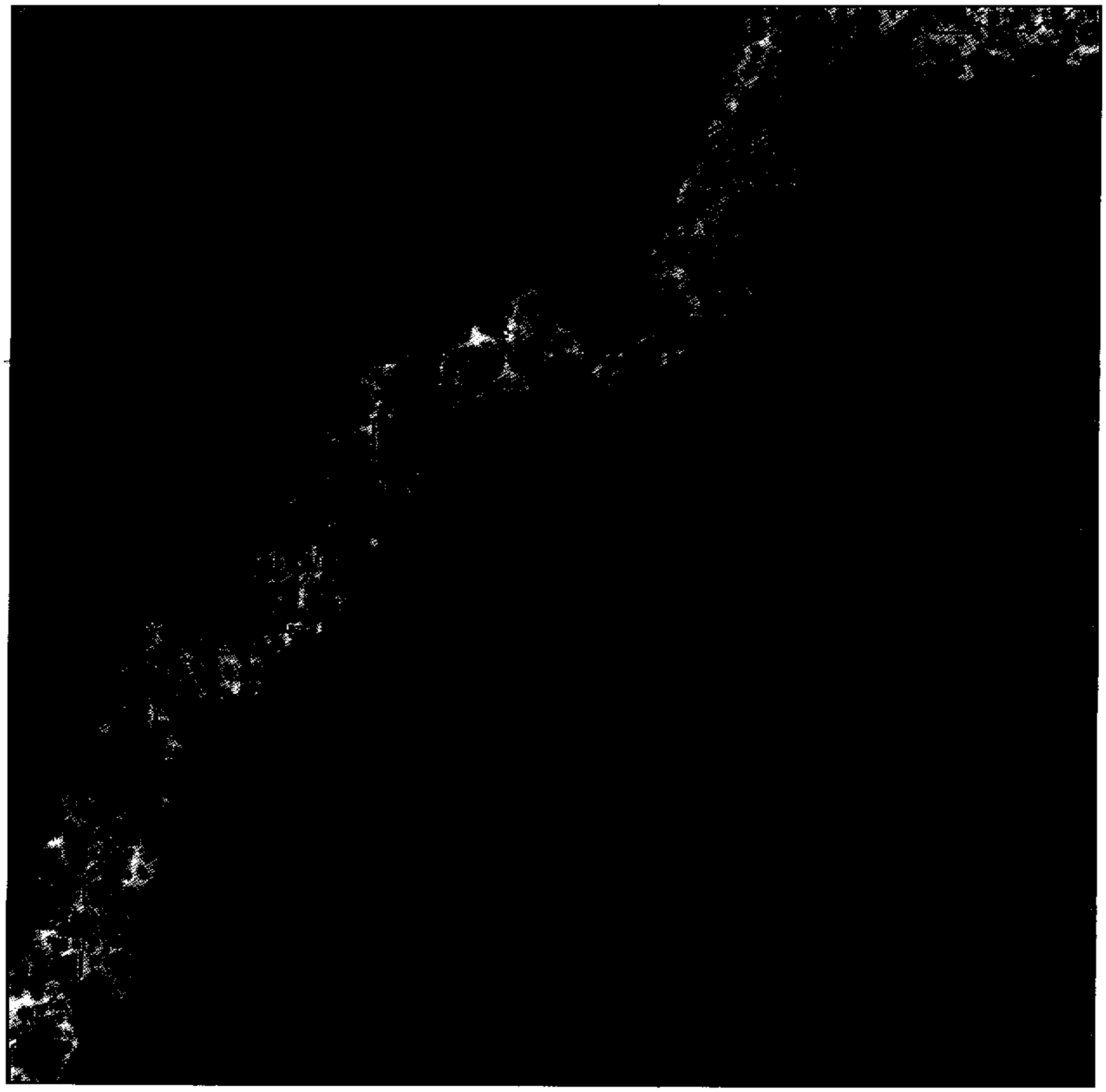
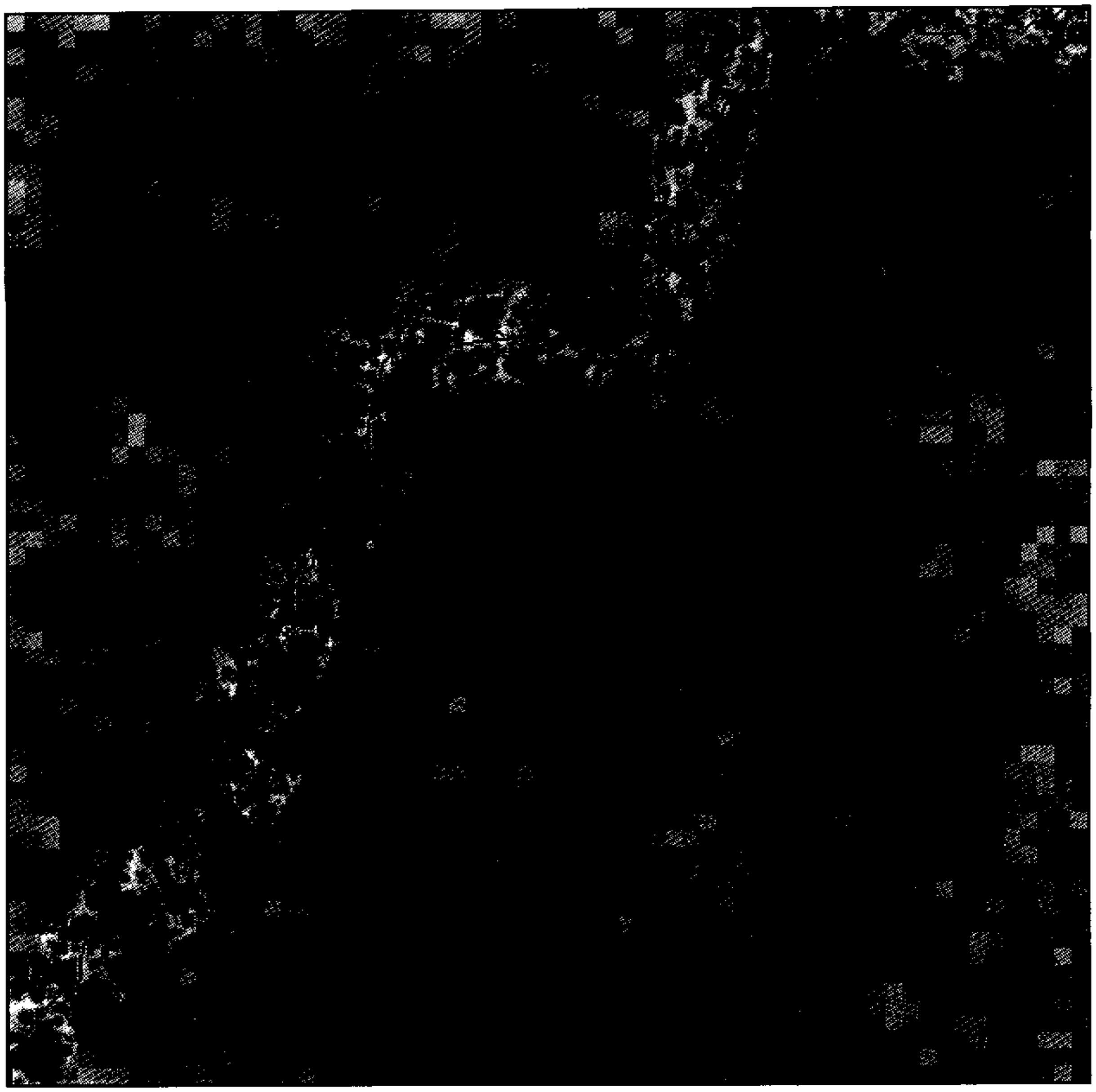


Figure 19: Path Planning with a Medium-Resolution Map



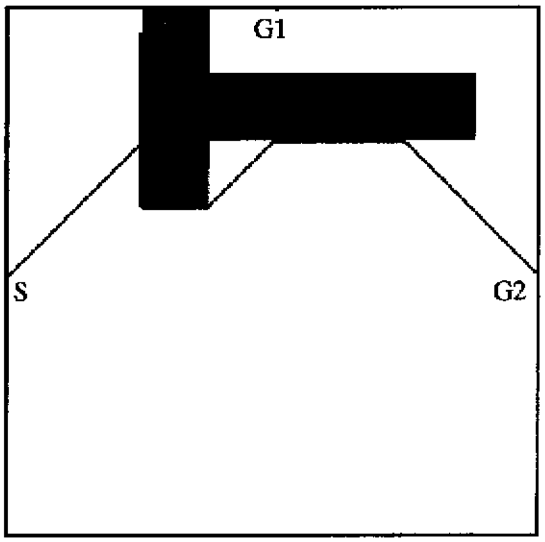
4.7 Multiple Goal States

In the description of D* in Section 2.0, it is assumed that only one state has an $h(°)$ value of zero (i.e., there is only one goal state). D* permits more than one goal state. This feature has several uses. For example, any part of a designated area on the floor may suffice as a goal; thus, all cells in the area are equivalent goals. Furthermore, two widely-separated doors leading out of a room may be considered equivalent goals.

It may even become important to introduce new goal states while the robot is moving through the environment. For example, the robot may be heading for the one, known door in a room when it detects a second door with its sensor. In this case, the new goal state X with $h(X) = 0$ is entered on the $OPEN$ list and $PROCESS-STATE$ is called repeatedly until k_{min} equals or exceeds $h(Y)$, where Y is the robot's current state.

Figure 20 illustrates the use of two goal states in path planning. The a priori map contains only *EMPTY* cells, and therefore the robot does not know about the T-shaped obstacle. Initially, both goal states, G_1 and G_2, are entered on the *OPEN* list with $h(G_1) = h(G_2) = 0$. The routine *PROCESS – STATE* is called repeatedly until the robot's start state has an optimal path. Note that initially the robot moves toward G_1 since G_1 is closer, and it presumes the straight-line path is unobstructed. When the robot detects the obstacle, it attempts to move around it to the top and then doubles back down and to the right. Eventually it works itself into a position where G_2 is closer than G_1 and proceeds to move to G_2 instead. The robot's field of view in this example is 10 cells.

Figure 20: Path Planning with Two Goal States

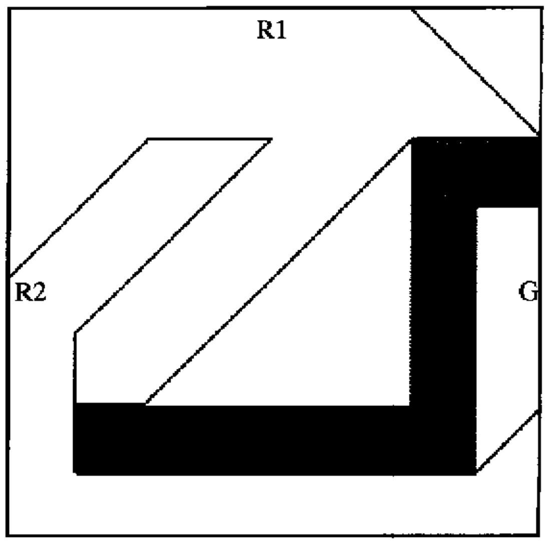


4.8 Multiple Robots

In some applications, two or more robots operate in an environment [15][21]. For example, consider two robot scouts returning to a home base across largely unknown terrain after performing a reconnaissance mission. Both robots are equipped with sensors to measure terrain properties. Assuming the robots cannot interfere with each other, a simple way of implementing this mission is to equip both robots with a priori maps and instruct them to move independently toward the goal. It is clear from previous sections how to implement this mission with D*. With this arrangement, however, neither robot benefits from the sensor readings of the other robot. If the two robots share a map, then obstacles detected by one robot are available to the other.

Consider the example in Figure 21. Robots R_1 and R_2 are equipped with a one-cell FOV. They move toward the goal state G using a map of the environment containing the grey obstacle. Because they are unaware of the black obstacle, both robots plan to move over the top of the grey obstacle and down to the goal. R_1 is ahead of R_2. As both robots move forward, R_1 discovers that the black obstacle obstructs its path. This information is effectively communicated to R_2 via the shared map, and R_2 chooses an alternate route and changes course long before it reaches the impasse. In order to use D* in a shared arrangement, whenever a discrepancy is discovered between the presumed world and the actual world (i.e., $s(X) \neq a(X)$ for some state X) by either robot's sensor, the changes are entered onto the *OPEN* list via *MODIFY – COST*, and *PROCESS – STATE* is called repeatedly until k_{min} exceeds the $h(°)$ values of both robots. Each robot is free to move (optimally) when its own $h(°)$ value equals or is exceeded by k_{min}. D* is most efficient when the two robots are located on cells with approximately the same $h(°)$ value.

Figure 21: Path Planning with Two Robots



5.0 Experimental Results

D* was compared to the optimal replanner to verify its optimality and to determine its performance improvement. The optimal replanner initially plans a single path from the goal to the start state. The robot proceeds to follow the path until its sensor detects an error in the map (i.e., $s(X) \neq a(X)$ for some X). The robot updates the map, plans a new path from the goal to its current location, and repeats until the goal is reached. An optimistic heuristic function $\hat{g}(X)$ is used to focus the search, such that $\hat{g}(X)$ equals the "straight-line" cost of the path from X to the robot's location assuming all cells in between are *EMPTY*. The replanner repeatedly expands states on the *OPEN* list with the minimum $\hat{g}(X) + h(X)$ value. Since $\hat{g}(X)$ is a lower bound on the actual cost from X to the robot for all X, the replanner is optimal [14].

The two algorithms were compared on planning problems of varying size. Each environment was square, consisting of a start state in the center of the left wall and a goal state in center of the right wall. Each environment consisted of a mix of map obstacles (i.e., available to robot before traverse) and unknown obstacles measurable by the robot's sensor. The sensor used was omnidirectional with a 10-cell field of view. Figure 22 shows an environment model with 100,000 states. The map obstacles are shown in grey and the unknown obstacles in black.

Figure 22: Typical Environment for Path Planning Comparison

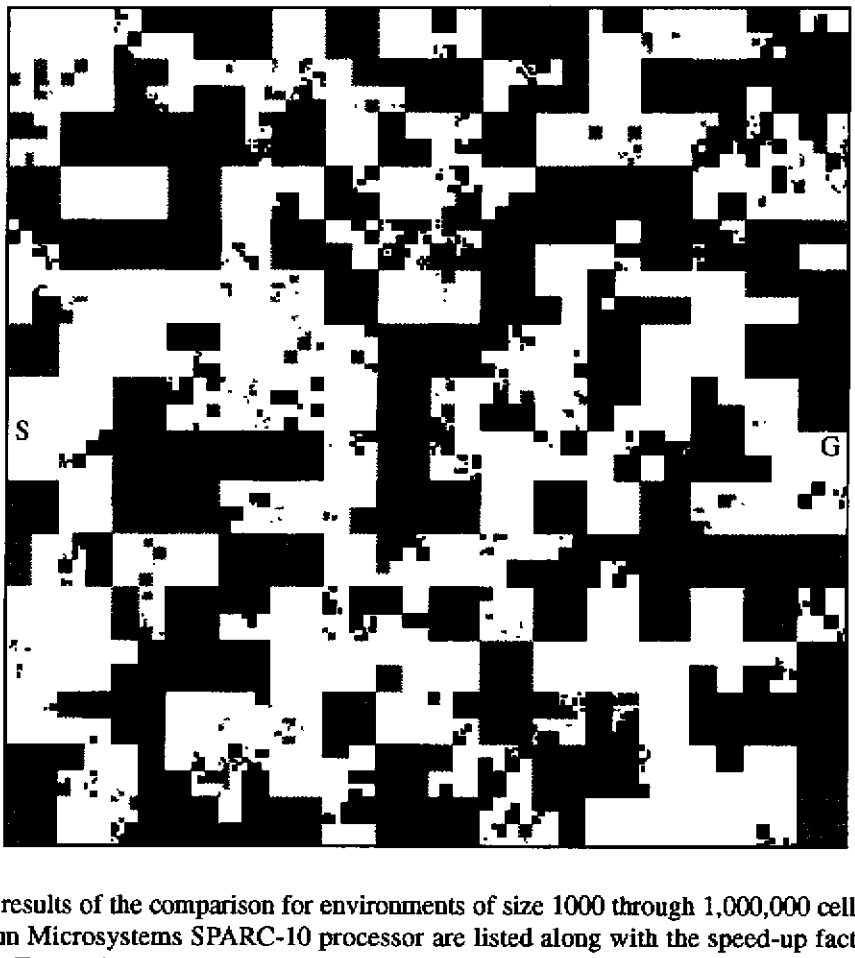


Table 1 shows the results of the comparison for environments of size 1000 through 1,000,000 cells. The runtimes in CPU time for a Sun Microsystems SPARC-10 processor are listed along with the speed-up factor of D* over the optimal replanner. For each environment size, the two algorithms were compared on five randomly-generated environments, and the runtimes were averaged. The speed-up factors for each environment size were computed by averaging the speed-up factors for the five trials.

The runtime for each algorithm is highly dependent on the complexity of the environment, including the number, size, and placement of the obstacles, and the ratio of map to unknown obstacles. The results indicate that as the environment increases in size, the performance of D* over the optimal replanner increases rapidly. The intuition for this result is that D* replans locally when it detects an unknown obstacle, but the optimal replanner generates a new global trajectory. As the environment increases in size, the local trajectories remain constant in complexity, but the global trajectories increase in complexity.

Note that even for large environments (e.g., 1,000,000 cells), D* is a real-time algorithm. If the environment consists of square-meter resolution cells, then a 1,000,000-cell environment is a square kilometer. If a robot drives the width of the square-shaped terrain using the optimal replanner, its average speed will be limited to 1.2 km/hr at best. If D* is used, the speed will be limited by the robot itself. This is important since any planner for unknown and dynamic environments must necessarily operate in lock-step with a moving and sensing robot.

Table 1: Comparison of D* to Optimal Replanner

Algorithm	1,000	10,000	100,000	1,000,000
Replanner	427 msec	14.45 sec	10.86 min	50.82 min
D*	261 msec	1.69 sec	10.93 sec	16.83 sec
Speed-Up	1.67	10.14	56.30	229.30

6.0 Conclusions

6.1 Summary

This paper presents D*, a provably optimal and efficient path planning algorithm for sensor-equipped robots. The algorithm can handle the full spectrum of a priori map information, ranging from complete and accurate map information to the absence of map information. A number of applications are illustrated, including planning with robot shape, field of view considerations, dead-reckoning error, changing environments, occupancy maps, potential fields, natural terrain, multiple goals, and multiple robots.

D* is a very general algorithm and can be applied to problems in artificial intelligence other than robot motion planning. In its most general form, D* can handle any path cost optimization problem where the cost parameters change during the search for the solution. D* is most efficient when these changes are detected near the current starting point in the search space, which is the case with a robot equipped with an on-board sensor.

6.2 Future Work

For unknown or partially known terrains, recent research literature has addressed the exploration and map building problems [12][16][17][18][22] in addition to the path finding problem. Using a strategy of raising costs for previously visited states, D* can be extended to support exploration or acquisition tasks.

Quad trees have limited use in environments with cost values ranging over a continuum, unless the environment includes large regions with constant traversability costs. Future work will incorporate the quad tree representation for these environments as well as those with binary cost values (e.g., *OBSTACLE* and *EMPTY*) in order to reduce memory requirements [22].

Although D* has been shown to be efficient, there is room for improvement. Presently, the effects of a cost change are propagated out from the modified arc in all directions. It may be possible to bias this propagation in the direction of the robot by using a heuristic function similar to that employed in A* [14], thus resulting in a new optimal path to the robot's state with fewer state expansions. A function $\hat{g}(\circ)$ must be selected that provides a lower bound on the true cost $g(\circ)$ from the modified arc to the robot's state to preserve optimality. There are two complications. First, the robot is in motion, so $\hat{g}(\circ)$ must be recomputed for each state on the *OPEN* list. Second, using $h(\circ) + \hat{g}(\circ)$ instead of $h(\circ)$ will require a new definition for $k(\circ)$ that preserves completeness and optimality.

Acknowledgments

The author thanks Alonzo Kelly and Paul Keller for graphics software and ideas for generating fractal terrain. The author also thanks Chuck Thorpe for comments on the document and the Unmanned Ground Vehicle and Autonomous Planetary Exploration groups at CMU for feedback on uses for the algorithm.

References

[1] Elfes, A., "Occupancy Grids: A Probabilistic Framework for Robot Perception and Navigation," Ph.D. thesis, Electrical and Computer Engineering Department / Robotics Institute, Carnegie Mellon University, May, 1989.

[2] Erdmann, M., Lozano-Perez, T., "On Multiple Moving Obstacles," Technical Report AIM-883, MIT Artificial Intelligence Laboratory, 1986.

[3] Fujimura, K., "On Motion Planning amidst Transient Obstacles", Proc. of the IEEE International Conference on Robotics and Automation, May, 1992.

[4] Fujimura, K., Samet, H., "Motion Planning in a Dynamic Domain," in Proc. IEEE International Conference on Robotics and Automation, May, 1990.

[5] Goto, Y., Stentz, A., "Mobile Robot Navigation: The CMU System," IEEE Expert, Vol. 2, No. 4, Winter, 1987.

[6] Jarvis, R. A., "Collision-Free Trajectory Planning Using the Distance Transforms," Mechanical Engineering Trans. of the Institution of Engineers, Australia, Vol. ME10, No. 3, September, 1985.

[7] Jarvis, R. A., "Collision-Free Path Planning in Time-Varying Environments," RSJ International Workshop on Intelligent Robots and Systems, September, 1989.

[8] Khatib, O., "Real-Time Obstacle Avoidance for Manipulators and Mobile Robots", the International Journal of Robotics Research, Vol. 5, No. 1, Spring, 1986.

[9] Korf, R. E., "Real-Time Heuristic Search: First Results," Proc. Sixth National Conference on Artificial Intelligence, July, 1987.

[10] Latombe, J.-C., "Robot Motion Planning", Kluwer Academic Publishers, 1991.

[11] Lozano-Perez, T., "Spatial Planning: A Configuration Space Approach", IEEE Transactions on Computers, Vol. C-32, No. 2, February, 1983.

[12] Lumelsky, V. J., Mukhopadhyay, S., Sun, K., "Dynamic Path Planning in Sensor-Based Terrain Acquisition", IEEE Transactions on Robotics and Automation, Vol. 6, No. 4, August, 1990.

[13] Lumelsky, V. J., Stepanov, A. A., "Dynamic Path Planning for a Mobile Automaton with Limited Information on the Environment", IEEE Transactions on Automatic Control, Vol. AC-31, No. 11, November, 1986.

[14] Nilsson, N. J., "Principles of Artificial Intelligence", Tioga Publishing Company, 1980.

[15] Parsons, D., Canny, J., "A Motion Planner for Multiple Mobile Robots," in Proc IEEE International Conference on Robotics and Automation, May, 1990.

[16] Pirzadeh, A., Snyder, W., "A Unified Solution to Coverage and Search in Explored and Unexplored Terrains Using Indirect Control", Proc. of the IEEE International Conference on Robotics and Automation, May, 1990.

[17] Rao, N. S. V., "An Algorithmic Framework for Navigation in Unknown Terrains", IEEE Computer, June, 1989.

[18] Rao, N.S.V., Stoltzfus, N., Iyengar, S. S., "A 'Retraction' Method for Learned Navigation in Unknown Terrains for a Circular Robot," IEEE Transactions on Robotics and Automation, Vol. 7, No. 5, October, 1991.

[19] Samet, H., "An Overview of Quadtrees, Octrees and Related Hierarchical Data Structures," in NATO ASI Series, Vol. F40, Theoretical Foundations of Computer Graphics, Berlin: Springer-Verlag, 1988.

[20] Sharma, R., "A Probabilistic Framework for Dynamic Motion Planning in Partially Known Environments", Proc. of the IEEE International Conference on Robotics and Automation, May, 1992.

[21] Warren, C. W., "Multiple Robot Path Coordination Using Artificial Potential Fields," in Proc. IEEE International Conference on Robotics and Automation, May, 1990.

[22] Zelinsky, A., "A Mobile Robot Exploration Algorithm", IEEE Transactions on Robotics and Automation, Vol. 8, No. 6, December, 1992.