Shortest Path Planning on Topographical Maps

Youssef Saab and Michael VanPutte

***Abstract*—This paper introduces a new algorithm for quickly answering repetitive least-cost queries between pairs of points on the earth's surface as represented by digital topographical maps. The algorithm uses a three-step process; preprocessing, geometrically modified Dijkstra search, and postprocessing. The preprocessing step computes and saves highly valuable global information that describes the underlying geometry of the terrain. The search step solves shortest path queries using a modified Dijkstra algorithm that takes advantage of the preprocessed information to "jump" quickly across flat terrain and decide whether a path should go over or through a high-cost region. The final step is a path improvement process that straightens and globally improves the path. Our algorithm partitions the search space into free regions and obstacle regions. However, unlike other algorithms using this approach, our algorithm keeps the option of passing through an obstacle region.**

I. Introduction

Finding the least-cost path is a challenging classical problem in computer science. For decades, research has been conducted to solve the least-cost path problem under a variety of situations. Today many real-time decision making applications do repetitive mode queries, solving many shortest-path problems while generating a solution to a larger problem. Decreasing the effort when finding a path can greatly improve the overall performance of these software systems.

Many methods are currently used to find the least-cost path from a single source to a single goal on a two-dimensional (2–D) representation of a surface. Current research in shortest path planning primarily consists of a node generating blind search or complete obstacle avoidance. Both techniques have benefits and hazards when applied to topographic path planning. This paper presents a new algorithm that quickly finds a near optimal path. Our algorithm combines the benefits of node generating searches with geometric techniques to find near optimal least-cost paths quickly.

II. Topographical Maps

The domain consists of a topographical map, a cellular-decomposition representation of the earth's surface [1]. This map has two degrees of freedom x and y, looking down perpendicular on the environment. The variables x and y map to the elevation of the terrain, the z coordinate.

The elevations found on the topographical map are used to construct the terrain matrix database or elevation array. Each $\{x, y\}$ entry in the database corresponds to the *average elevation* (z value) for the corresponding piece of terrain.

This paper is only concerned with the elevation above sea level. Many other features can be analyzed, such as surface type, vegetation, pollution, or population data. This abstraction of the terrain permits rapid computation and problem solving. More information on geocoding, the process of creating a digitized representation of the earth's surface, can be found in [2] and [3].

Manuscript received June 2, 1997; revised February 24, 1998.

The authors are with the Department of Computer Science and Engineering, University of Missouri, Columbia, MO 65211 USA (e-mail: saab@cedars.cecs.missouri.edu).

Publisher Item Identifier S 1083-4427(99)00896-6.

The elevation array can be transformed into a simple graph. The $\{x, y\}$ coordinates in the matrix correspond to vertices on a simple graph and are called nodes. The grid creates a regular lattice with diagonal arcs, such that each node is connected to its eight adjacent nodes [4]. From any location on the map (except those on the perimeter of the map) one can move to eight adjacent locations (North, Northeast, East . . .). The use of the eight squares rather than four squares better models how objects move on the earth's surface (Figs. 1–3).

III. Cost

Crossing terrain requires the expenditure of energy. Climbing a hill costs more than walking down a hill, which costs more than moving over flat terrain (assuming a driver applies brakes and steering to maintain control of a descending vehicle). This expenditure is the cost to move from one vertex to another. If the slope and resultant cost between two vertices is too high, then the terrain is too steep to travel over safely, and the edge between the two vertices is removed. The cost to traverse the terrain is not explicitly stated in the database, but is a function of the change in elevation of two adjacent vertices.

Elevation arrays commonly used in terrain analysis applications produce a "digital bias" [5]. Although some algorithms return a shortest path as an ordered set of adjacent nodes, they do not produce an Euclidean shortest path. Fig. 4(a) represents an elevation array of a flat plane. All direct paths from the start point S to the goal point G in the polygon bound by *SAGB* have the same length, 8.828. In Fig. 4(b), path 2 is a more direct path than path's 1 or 3, yet all have the same length. Fig. 4(c) shows the true shortest distance on the same plane. The line segment $S \rightarrow G$ is an example of the saying, "The shortest distance between two points is a straight line." A shortest path is then defined as the shortest ordered set of line segments from S to G.

On a flat plane, the true cost is the Euclidean length of the path. On rolling terrain, or terrain that changes elevation, the cost is a function of the change in elevation of each point (Table I). Constraints imposed by the system that traverses the terrain must also be taken into account when developing the shortest path.

IV. Dynamic Obstacles

In nontrivial applications additional obstacles may be created and destroyed at runtime. The ability to create and destroy dynamic obstacles permits users to analyze situations and perform "what-if" analysis. Examples of dynamic obstacles are polluted areas in cities, minefields encountered or planned in military mission planning, and air corridors closed due to bad weather.

The term dynamic obstacles as used in this paper should not be confused with dynamic obstacles used in robot motion planning. The field of motion planning refers to dynamic obstacles as objects that move about the search space while a search is being performed [6]. Finding a path across town for an automobile in the presence of pedestrians is an example of dynamic obstacles used in motion planning. This paper is concerned with finding an algorithm that will quickly find a path in the presence of terrain features and temporary impenetrable obstacles.

On a typical real-time application a user may add, move, or delete hundreds of these temporary dynamic obstacles while doing an analysis. These impenetrable obstacles are considered barriers that

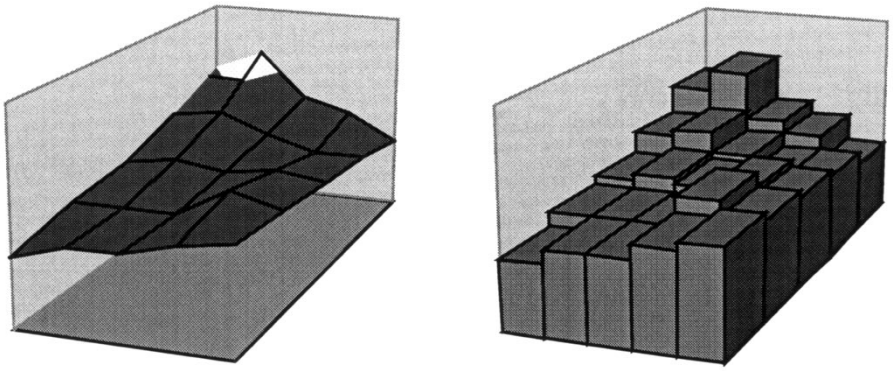

Fig. 1. Topographical map transformation.

TABLE I
EXAMPLE OF AN ELEVATION ARRAY

		X				
		1	2	3	4	5
Y	1	10	12	12	14	15
	2	12	12	12	16	14
	3	13	14	13	14	13
	4	16	17	14	12	12
	5	20	23	16	14	10

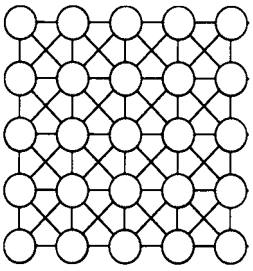

Fig. 2. Graph representation of an array.

must be bypassed. The constraints imposed by dynamic obstacles have a significant effect on the algorithm analysis for this problem.

V. TYPES OF SURFACE TERRAIN

The earth's surface has a variety of features and terrain types ranging from flat plains to very rough mountain ranges. The shortest path on flat plains represents a trivial problem, being a straight line. Mountain ranges represent the other extreme, where the shortest path is much more difficult to compute.

This paper will focus on valley floors of desert plains, as found in the Mojave Desert of Southern California. These valleys are surrounded by very steep, impenetrable mountain ranges. The valley floors are roughly planar, with hills and outcroppings interspersed. This abstraction of the environment's surface is sufficient for introductory research, yet remains computationally challenging.

VI. APPLICATIONS

Shortest path algorithms are used in geographic information systems, military mission planning systems, autonomous robot route planning systems, terrain following "nap-of-the earth" aircraft route planning and city regional planning systems. Huge elevation databases have been developed by the U.S. Defense Mapping Agency [2] for use in both civilian and government applications. These

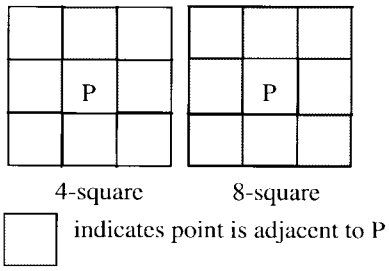

Fig. 3. Four-square versus 8-square adjacent nodes.

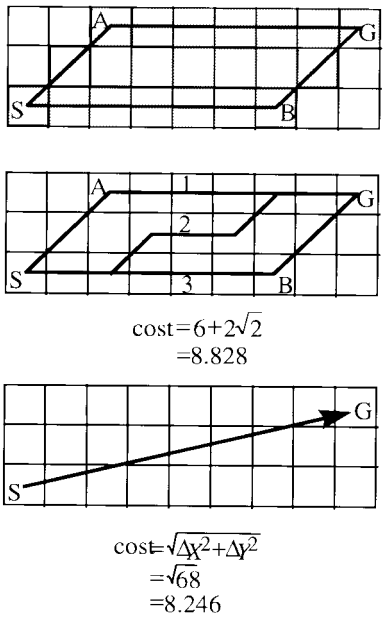

Fig. 4. Cost calculations on elevation arrays.

databases provide a rich area for theoretical and practical work on digital representations of real terrain.

Other areas of shortest path applications similar to this domain include:

Shipping/Distribution Problem—planning the movement of products from a factory to a warehouse [7].

Economics—building a network of nodes where each node represents a state, and edges represent costs to achieve future states. A decision at a node to take an edge transforms the problem to a

future state. The goal is to find the shortest path that maximizes a future profit [7].

Critical Path Analysis—performing operations research and planning analysis in management [7].

Circuit Board Layout and VLSI Design—hills represent electrical components on the circuit board and paths over hills represent jumper wires [8].

Automated Traveling Advisory System—costs represent distances and dynamic obstacles represent potentially hazardous weather [8].

Remote Sensing—determining surface properties from images [9].

VII. Background

A technique to improve performance on repetitive-mode query applications is to precompute some information on the problem, and use that information to prune the search. One extreme is to precompute and store the entire finite set of all possible queries, providing an immediate access time with a very large storage overhead. The other extreme is to eliminate the precompute step and compute the entire solution with a high response time and no precomputation cost or storage overhead. The optimal solution is somewhere in the middle of these two extremes; use some initial precomputation effort and storage overhead to obtain a long-term gain in search time [10].

Shortest path research generally falls into two categories:

Blind Search Over the State Space. These algorithms build a search tree superimposed over the state space (Fig. 5). They conduct a blind search through the state space one node at a time without using global knowledge of the entire problem. These algorithms return optimal solutions, but are computationally and memory expensive.

Perfect Obstacle Avoidance on a Plane. These algorithms partition the domain space into perfect obstacles and perfect free space. The search for the best path avoids obstacles at all cost without checking if the path should go over an obstacle rather than around. Precomputed information is either paths that avoid obstacles or obstacle boundaries that must be avoided. Obstacle avoiding algorithms return quick solutions but may overlook "short cuts" through obstacles that lead to globally optimal solutions.

Before proceeding, the following notation is defined. The size of the elevation array database is N. The value of N is equal to the number of columns in the array multiplied by the number of rows. This N is also equal to the number of nodes in the state space of a graph representation of the elevation array.

VIII. Blind Search on the State Space

Blind search algorithms, or "informed best-first search algorithms" [11], represent one extreme to the solution of the shortest path problem. These blind searches, whether Dijkstra's algorithm, best-first, A^*, or bidirectional A^* [12] are essentially a modified Dijkstra algorithm. These algorithms build a search tree superimposed on the state space. The leaf nodes are nodes of the tree discovered or that have no children generated, and are maintained on an *open list*. Nodes that have been expanded are called inner nodes and are maintained on a *closed list* [8]. These algorithms use no precomputed information, computing the entire solution to a query on demand.

Heuristic search algorithms are called informed searches in the artificial intelligence community [8]. This term is misleading since heuristic searches have no information except that found in the state definitions and the operators that change the states. Although heuristic searches have problem specific knowledge embedded in the operators [4] that causes them to search more efficiently than "uninformed" searches (breadth-first and depth-first), they do not use global knowledge inherent to the specific problem.

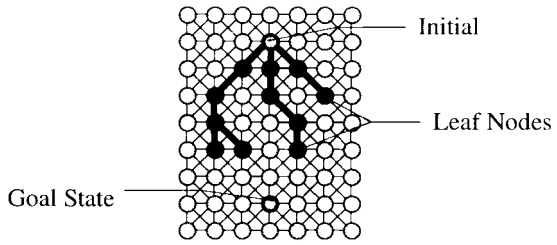

Fig. 5. Blind search procedure.

These algorithms use a priority scheduling algorithm to select the next node for expansion, determine which adjacent children of a node are to be generated, and place generated nodes onto the priority queue. These algorithms suffer from the same lower-bound best-first time $O(n \log n)$ where n is the number of nodes generated. This worst time is due to the management of the open-list [8]. Research has led to improving the heuristic used for placing a node onto the priority queue and bounding the search region [13].

Although all of the algorithms may return an optimal solution providing they have an admissible heuristic, they all suffer from serious inefficiencies. These flaws include:

Time—The number of nodes generated and expanded is large which implies long search time. Often such algorithms waste time exploring unnecessary nodes in terrain databases.

Space—These heuristic algorithms maintain all nodes generated and expanded in memory, which is a major constraint for any large map. Storing pointers and cost labels is a major drain on memory resources, causing the algorithm quickly to run out of memory on long searches.

Ties on the priority queue and the order that nodes are created cause an arbitrary implied ordering. This may result in search paths predisposed toward one direction [11].

Several ideas have been tried to improve heuristic searches. Techniques include generating the best child of a node (which does not guarantee to return optimal solution due to the *horizon effect*) and not searching the open or closed list that results in duplicate storage of identical nodes [8].

IX. All-Pair Shortest Path

The all-pair shortest path represents the other extreme to the shortest path problem. This algorithm precomputes the shortest path among all pairs of N points on a map, and saves the result to secondary storage. This algorithm has two extreme costs, the effort to precompute all of the shortest paths between N points and the storage of the solutions. The all-pair shortest path can be computed naively using a repeated Dijkstra's algorithm in $O(N^2 \log N)$ time. The storage for the N^2 paths is $O(N^3)$. By storing only an intermediate node m on the shortest path from $i \rightarrow j$ into an array $A[i, j] = m$, the shortest paths can be constructed by recursive queries to $A[\ ]$. The array would require a total storage of size $\Theta(N^2)$.

This trivial solution has two disadvantages that eliminate it as a practical solution. Dynamic obstacles may be inserted, deleted, or moved on the terrain at runtime, which will eliminate precomputed shortest path solutions. A possible alternative is to precompute and store some shortest paths. This would involve preprocessing critical shortest paths between nodes and storing the preprocessed data efficiently. This solution also relies on secondary storage (since searching will still need to be done for nodes not precomputed) and suffers from the dynamic obstacle problem.

X. Motion Planning

Extensive research has been conducted in motion planning. Much of the research involves robot manipulator planning or perfect polygon-obstacle avoidance [6]. Latombe even goes as far as to define motion planning as "planning a collision free path for rigid objects among static impenetrable obstacles" [14].

Motion planning is performed on a planar map. The configuration space is the transformation from the real space of the earth's surface to another space where the robot is considered a point. The "difficult" terrain is surrounded by a perimeter considered impenetrable, creating obstacle regions. These *perfect obstacle regions* may be created by bounding the obstacle space with circles [15], minimum enclosed rectangles [16] or convex regions [17]. The problem becomes the finding of the shortest path between two points in a fixed environment with a collection of disjoint, impenetrable, immobile obstacles [18].

XI. Skeletonization

Skeletonization is a class of algorithms that precompute some information and uses this precomputed information to decrease response time to queries. Two techniques applicable to topographic map research are visibility graphs and Voronoi diagrams. Both techniques require a free space/obstacle space representation of the search space. These techniques precompute a network (the skeleton) which represents a global overview of the terrain. The path from a start point S to a goal point G is found in three steps. First, a minimal path is found from S to the skeleton, the intersection being I_1. Next, a minimal path is found from G to the skeleton with the point of intersection being I_2. Finally, a path is found using graph theory from I_1 to I_2.

Visibility graphs take the planar map and perfect obstacle regions and create a graph whose edges connect all pair of boundary corners that are visible from each other [19]. Extensive research has been done on the use of visibility graphs.

Voronoi diagrams are networks consisting of a set of points equal distance from two or more object features [10]. Paths found on Voronoi diagrams appear to travel down the middle of valleys avoiding obstacles.

Both visibility graphs and Voronoi diagrams have the benefit of capturing the global topology, thus finding paths quickly, although the resulting paths may be far from optimal.

XII. Cellular Decomposition

The cellular decomposition algorithm [20] must also be given the free space and obstacle space representation as input. These algorithms divide the entire state space into cells, and build a connectivity graph of adjacent cells. The shortest path is found by a simple graph search through the connectivity graph. Schwartz's papers [19] on the collision-free path problem termed the "Piano-Movers problem" is considered a classic in motion planning. Quadtrees are another version of the cellular decomposition solution [6].

Although all of the preceding algorithms work very efficiently for perfect, impenetrable obstacles on a planar map, they do not work for this problem domain. The input for these algorithms requires the creation of impenetrable obstacle regions that cannot be traversed. One approach is to designate any region whose minimal elevation exceeds a threshold value. This approach may lead to a locally optimal, low quality solution, when a more optimal decision is to go over a small ridge than around it. A robot centered at the base of a very long ridge whose elevation narrowly exceeds the threshold would choose to go around the ridge when finding a point to cross over the ridge may be much shorter. A second limitation is that these algorithms do not permit the start point or end point to be inside an obstacle region. These constraints are too excessive for practical applications on terrain.

A more serious limitation of the skeletonization algorithms is the introduction of dynamic obstacles. If a dynamic obstacle is placed on a point on the map that intersects a segment of the network then this segment of the network is no longer usable. Two techniques can deal with this situation; find an alternate route or calculate a bypass. A significant portion of the visibility graph or connectivity graph (on the cellular decomposition) may need to be recomputed every time a dynamic obstacle is introduced or deleted. These computations every time a dynamic obstacle is created, moved, or destroyed are very expensive in time and space, and may result in excessive computations to maintain the networks.

XIII. Weighted Region

In weighted region algorithms the plane is divided into regions with weights representing terrain features. The weights are the costs to move per unit distance across the terrain. Dynamic obstacle regions would be given the weight infinity. This technique turns out to be no more than conducting a blind search using Dijkstra or other technique. The "good runner—poor swimmer" problem and the "maximum concealment" problems are both solved on terrain maps using the weighted region algorithm. The weighted region algorithm can be converted into a graph problem by transforming the regions with equal weights into polygons and performing a graph search [5].

The weighted region algorithm is not applicable to this paper because of its technique of modeling cost. Cost on weighted regions is the value of that piece of terrain in an array (a node in a graph). Cost on the elevation array is the change in elevation between two points (the edge that connects two vertices). The cost to get to a piece of terrain is dependent on the direction used to move to that piece of terrain. This observation eliminated weighted regions as a possible solution.

XIV. Potential Fields

Potential fields calculate a scalar quality for every point on the terrain surface. This quantity is zero at the goal, a very high value at obstacles, and decreases as a point moves away from obstacles toward the goal. The path from the start to the goal is synonymous to a marble rolling down a hill toward the goal [21]. The value for a point $\{x, y\}$ on the ground is:

$$\left(\sum_{i=1}^{\text{number of obstacles}} \frac{1}{\text{distance}(\{x,y\}, \text{obstacle}_i)}\right) - \frac{1}{\text{distance}(\{x,y\}, \text{goal})}$$

Potential field algorithms can deal with dynamic obstacles by setting the value of points in dynamic obstacles to ∞. A significant drawback to these algorithms is that the entire region $O(N)$ must be computed for every query and every time a dynamic obstacle is created or destroyed. This is a significant effort for a very large search space.

XV. Geometric Techniques

Geometric techniques can be used to find a solution on two or more dimensional surfaces quickly. These algorithms have the benefit of using line segments and returning paths as sequences of line segments rather than long lists of adjacent coordinates. Given the elevation array, a free space/convex polygon obstacle space representation of the surface is precomputed. Finding the shortest path from $S \rightarrow G$ is called the "taut string" approach. A path is found when a string

is threaded from the start point through the obstacles to the goal and pulled tight. The difficulty with this algorithm is deciding how to thread the string through the obstacles, since there are an infinite number of ways [5].

Solving the problem with simple polygons is similar, except that the number of points to store, and the computation effort is significantly more. It should be noted, that a line segment would not enter the concave portion of a simple polygon when finding the shortest path unless the start and goal are inside the concave portion [10]. This observation will be used later in this paper.

XVI. Shortest Path on a Surface

The elevation database is transformed into a three-dimensional (3-D) representation of the surface. Techniques include flat plane and cylinders [5] and polyhedral surfaces with faces. These algorithms are related to triangulated irregular networks (TIN) in geography [2].

Many algorithms have been found that make a good quick initial guess to the solution, followed by a path improvement algorithm. Kimmel's continuing research has had success finding the shortest path on a 3-D polyhedral surface. This approach was stated as useful in CAD and land-surveying applications.

XVII. Edge Detection—Computer Vision

Edge detection and region analysis in computer vision can be used to bound high-cost regions (Fig. 6). These techniques take key points on the terrain map and combine them into sets of closed line segments that represent simple polygons [9]. Although these techniques do not represent a shortest path problem-solving strategy, they do suggest techniques and data structures that can be used to precompute information about the global characteristics of the terrain. This precomputed information can be saved and used in a shortest path algorithm.

XVIII. Geometrically Guided Search Algorithm

The central idea to this new algorithm is to exploit as much global information available about the terrain to make a quick, high-quality decision. A value in the elevation array describes the characteristics of the piece of terrain that value represents, while implicitly describing the relationship the piece of terrain has to its neighbors. This algorithm begins by taking a step back, examining the entire terrain, looking for regions a path "might" not want to travel through, and stores these precomputed high-cost regions for future use. This precomputing is synonymous with a person performing a quick terrain analysis before trying to solve a geographic problem. A terrain analysis permits the system to exploit the underlying geometry of the terrain rather than concentrating on single points on the ground. In repetitive mode applications, this initial cost of preprocessing and storage provides a long-term gain in searching.

This algorithm solves the shortest path problem in three steps. First, highly valuable global information about the terrain is precomputed. In this step the high-cost regions are found and saved as closed convex polygons. The next phase computes and saves the cost for paths among all points on the perimeter of the polygons.

The second step of the algorithm involves solving search queries. The algorithm projects the polygons onto a plane and searches for the shortest path by projecting a line segment from the start point to the goal across the plane. If the line intersects a polygon, the algorithm determines if going over is shorter than around the region, using the precomputed global information. In the final step, the algorithm does a path-improvement, postprocessing step to globally improve the path found. The algorithm quickly returns an approximate shortest path.

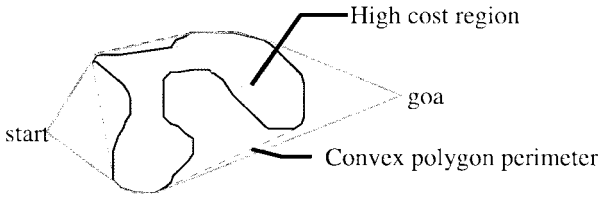

Fig. 6. High-cost convex polygon.

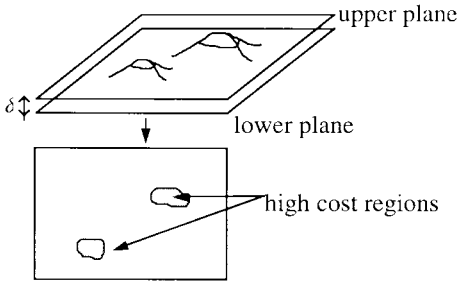

Fig. 7. The partitioned map.

XIX. Bounding High Cost Terrain—Preprocessing

Given an elevation array, the algorithm bounds high cost regions with convex polygons. These convex polygons are similar to contour intervals or iso-elevation curves found on geography maps [2]. This preprocessing step is performed once with the high-cost regions saved and used during path finding queries.

The decision to use convex polygons over simple polygons was due to the smaller space and reduced computational complexity of convex polygons. A shortest path between two points will not enter the concave portion of a simple polygon unless the start or end points are contained inside the concave region. The storage of convex polygons is significantly smaller (fewer numbers of points on the perimeter), and computations are significantly faster and less complex than similar computations using simple polygons.

High cost regions are found by slicing the terrain with two parallel horizontal planes. All points between the planes are considered low-cost terrain points. All points above the upper plane and below the lower plane are considered high cost terrain points (Fig. 7). High cost terrain points that are in close proximity are bound by a convex polygon. These polygons are saved as the obstacle space.

For this paper, the two planes are the base plane of the topographic map and a horizontal plane at an elevation δ. The δ value is a subjective elevation above which paths do not tend to enter because of the costs incurred. The intersection of the plane at δ and any high cost region is a contour, a simple polygon, which can be transformed into a convex polygon using Graham's Scan [10].

The next step to preprocessing is computing the shortest path cost among all pairs of perimeter points for each convex polygon. The shortest paths are stored in a ragged array [22] as in Fig. 8. If i and j are coordinates on the polygon perimeter, then the entry in row i, column j is equal to the entry in row j, column i. Both values do not need to be stored, so only $\frac{1}{2}n(n-1)$ entries are necessary per polygon where n is the number of points on a polygon perimeter.

XX. Search Procedure

A uniform cost horizontal plane is used representing the low-cost terrain. The convex polygons, representing the high cost regions, are projected onto this plane. Given a start point and a goal point, a line segment is projected between these two points. If the line segment

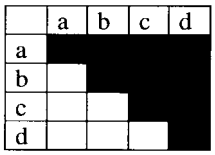

Fig. 8. Shortest path cost ragged array.

intersects a polygon, the algorithm determines if going around to the left, to the right, or over the region is best.

Labels on the map represent known costs from the start point to the node. All points on the terrain map begin unlabeled, and only explored nodes are labeled which eliminates initializing the entire map. The below algorithm applies to cases where the start and goal are outside polygons. Additional steps are necessary if the start or goal is inside polygons. The search algorithm requires three functions; add_or_relabel(), process_line(), and search_routine().

Given a point x on the map, a cost label c, and a priority-queue Q, add_or_relabel determines if a new path to x with cost c is a new or better path to x. If x has not been labeled before, it is labeled with cost c and pushed to the priority queue. If x has been labeled and c is less than the old label then x is relabeled and repositioned in Q based upon this new label. A point on the perimeter of a polygon is said to be visible from G, if ignoring all other polygons, a line segment can be drawn from G to the point without entering the polygon. Procedure add_or_relabel() is only called for the start point, goal point, and points on the perimeter of a polygon that are visible from the goal.

Procedure $\mathbf{add\_or\_relabel}(x, c, Q)$
Input: A point on the map x, a cost label c, and a priority queue Q

begin
  if x is not labeled **then**
    label $(x) = c$
    $Q \leftarrow x$
  else if $c \leq$ label(x) **then**
    label$(x) = c$
    reposition x in Q
end

The function process_line() takes a start point s, a goal G, and the priority queue Q. Process_line uses a stack to store lines(start,goal) that it is currently processing. It pops a line segment off the stack and checks if the line intersects a polygon. If the line does not intersect a polygon it sends the point and its newly computed label to add_or_relabel(). If the line intersects a polygon then the function computes the cost to get to three points on the polygon perimeter visible from G, and sends these three points to add_or_relabel(). These three points represent the paths to go through, around to the left, and around to the right of the polygon. See algorithm at the bottom of the page.

Process_line() uses a function $d(a, b)$ which calculates the Euclidean distance from a to b. It also uses the precomputed polygon path costs maintained in saved $[a, b]$, where a, b are points on the perimeter of polygons.

The function search_routine() is the main search routine. This function is sent the start and goal points for a path query, The function terminates when it finds its best path.

Procedure search_routine(S, G)
Input: Points on the map S, G

begin

Procedure $\mathbf{process\_line}(x, G, Q)$
Input: A point x that is either a startpoint or a point on a polygon visible from G, the main goal point G, and a priority queue Q
  function $d(a \rightarrow b)$ distance from a to b
  saved $[a, b]$ = precomputed cost of path for points a, b on perimeter of p

```
begin
  stack = Ø
  stack ← (x, G)
  while stack ≠ Ø do
    begin
      (s, g) ← stack
      if line (s, g) does not intersect any polygon then
        add_or_relabel (s, l(s) + d(s, g), Q)
      else                          /* refer to Fig. 9 */
        let p be the first polygon intersected on line (s, g)
        let F1 and F2 be the intersection of line (s, g) with the boundary of p.
          F1 is on side of S and F2 on side of G.
        add_or_relabel (F2, l(s) + d(s → F1)+ saved [F1, F2], Q)
        if line (s, t1) does not intersect any polygon then
          find nearest boundary point tp1 of p visible from G
          add_or_relabel(tp1, l(s) + d(s → t1)+ saved [t1, tp1], Q)
        else
          stack ← (s, t1)
        if line(s, t2) does not intersect any polygon then
          find nearest boundary point tp2 of p visible from G
          add_or_relabel(tp2, l(s) + d(s → t2)+ saved [t2, tp2], Q)
        else
          stack ← (s, t2)
    end
end
```

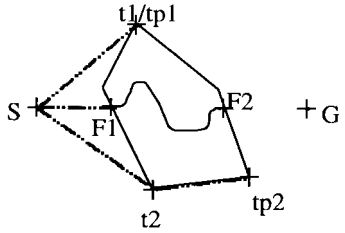

Fig. 9. Search through a single polygon.

$Q \leftarrow x$
while $Q \neq \emptyset$ **do**
begin
$x \leftarrow Q$
process_line(x, G, Q)
end
end

This algorithm resembles a Dijkstra's algorithm that begins at S. Unlike traditional Dijkstra's that examines its eight adjacent nodes at every step, this algorithm only examines the nodes visible from G on the perimeter of intersecting polygons. These descendant nodes are points visible from G and are labeled with a cost and a pointer to their ancestor. These descendant nodes are pushed onto a priority queue and the next node explored is the node on the queue with the lowest cost. When process_line() is called from search_routine() it is sent the main goal G, and searches from the node x will always seek G. If a node is discovered that was previously discovered and the new path from start to the node is shorter than the previous path, the cost label and the parent pointer are updated to reflect this better path. Using this method duplicate nodes for the same piece of terrain are not maintained.

The algorithm does not perform node generation on relatively flat terrain but "jumps" to another polygon or to the goal. The result is that only the start node and nodes on the perimeter of polygons visible from G can ever be expanded. This significantly reduces the search effort. When a line segment intersects a polygon, the precomputed costs are used to avoid searching inside the polygon.

If a start point is inside a polygon, Dijkstra's algorithm is used to find an efficient path out of the polygon to a point p that is visible from G. Once a path out of the polygon has been found the algorithm solved the problem from p to G and the solution path is $S \rightarrow p \rightarrow G$. If the goal is inside a polygon, the algorithm finds a path out of the polygon to a point p that is visible from S and the solution is $S \rightarrow p \rightarrow G$.

Performance of the search algorithm is very much dependent on the terrain data. If the input map contains no obstacles then the algorithm returns a straight line segment $S \rightarrow G$. If the terrain map in contained inside a single high-cost polygon then the algorithm performs a Dijkstra search requiring $O(N \log N)$ time and $O(N)$ space. In practice the algorithm performs much more efficiently.

The total space required for the search is the number of nodes on the open list and the path pointers maintained. Since searching for paths through polygons is delayed until last, the only nodes that have pointers are nodes on the perimeter of polygons. The space requirement therefore is $O(k)$ where k is the number of points on the boundary of polygons.

XXI. Path Improvement—Postprocessing

A postprocessing step is not necessary when solving a perfect-obstacle, shortest path problem. The resultant path segments always start and end on polygon perimeters, except possibly the start and goal. With these algorithms a solution path cannot be shortened without entering a polygon, which is not permitted in perfect-obstacle algorithms. When using the three-step algorithm it is possible to remove nodes from a path which result in less bends and a shorter path.

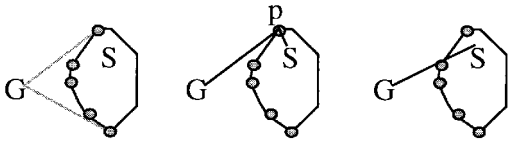

Fig. 10. Postprocessing.

The postprocessing algorithm examines every ordered subset of three consecutive nodes $\{n_1, n_2, n_3\}$ in a resultant path set Q. The center node n_2 in the path Q is discarded if the cost of the resultant path $\{n_1, n_3\}$ is less than the original path $\{n_1, n_2, n_3\}$.

Fig. 10 represents a flat plateau bounded by a convex polygon. The start point S is on the plateau and the goal G is to the left of the plateau on the plane. Fig. 10(a) shows all points visible from G (small circles). The path returned from step-2 of the search algorithm is a path off the plateau to a point visible from G, and concludes with a line segment from p to G. The path returned, Q, is $\{S, p, G\}$, shown in Fig. 10(b).

The path improvement algorithm examines every ordered set of three consecutive nodes beginning at the start coordinate. Path $\{S, p, G\}$ is examined. The path segment $S \rightarrow p \rightarrow G$ is longer than the path segment $S \rightarrow G$ [Fig. 10(c)]. Removing p shortens the global path, therefore p is removed from Q. The final path Q is $\{S, G\}$, Fig. 10(c).

This approach uses a heuristic to solve the problem of start or goal inside polygons. A point on the perimeter of a polygon is said to be visible from G, if ignoring all other polygons, a line segment can be drawn from G to the point without entering the polygon.

This algorithm performs a rapid approximation of the shortest path. The quality of this solution is bound by the value of δ. In one extreme case δ is set to the maximum elevation of the topographic map (or greater). In this case, the search space will be a flat plane with no obstacles and all searches $S \rightarrow G$ will return a straight line. In the other extreme case δ is set to one, and the algorithm will perform Dijkstra's algorithm across the entire state space. The quality and time/space cost is a function of the resolution of the contour interval.

Dynamic obstacles do not effect this algorithm. They are treated as polygons with cost to enter. Searches through the center of dynamic obstacles would not be performed since this would only return a path around the perimeter of the dynamic region polygon.

XXII. Experimental Results—Implementation

All algorithms were written in C and run on an Intel Pentium 166 machine. The terrain elevation databases consisted of 90 × 60 arrays of integers that permitted detailed analysis of the search algorithms.

Most maps were generated with a simple random terrain generator. The δ was chosen as the value of the base plane of the elevation array. Next, a random number of hills are assigned to the map, with each hill assigned a random elevation and radius. A cone is projected from the hilltop elevation to the base plane given the assigned radius. The elevation data is then transformed into an elevation array. By varying the number of hills, and the radius and elevation of the hills, a diversified group of maps were produced and tested. Some of these maps were manually adjusted to test worst case terrain. All elevation arrays are identified topo n where n is the identifier to the array. Fig. 11 is a 3-D representation of an elevation array.

Fig. 12 is a 2-D representation of the map, looking down onto the terrain. The shade of a region corresponds to the elevation of

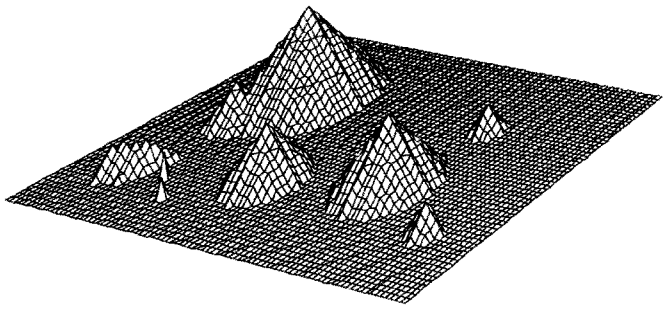

Fig. 11. Three-dimensional representation of an elevation array.

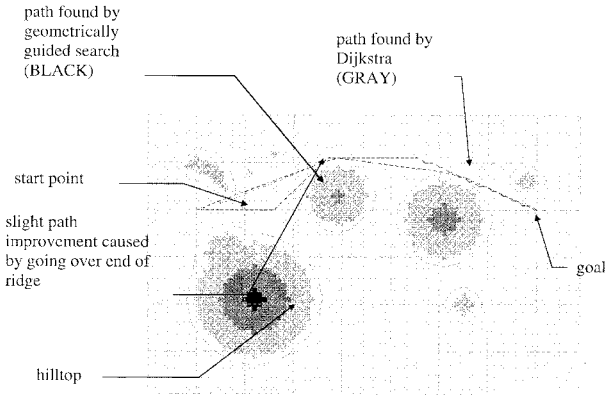

Fig. 12. Two-dimensional map representation.

that portion of the terrain. White regions are on the low-cost regions of the plane. Light gray, dark gray, and finally black regions have increasing values of elevation. High cost regions are bound by dotted lines representing the boundary of high-cost convex polygon regions found during preprocessing. Heavy dark lines are paths found using the geometrically guided search algorithm and thinner lines are paths found by a Dijkstra search. The x and y axis are the horizontal and vertical axis's on the map respectively. Points on the terrain are noted by x*-coordinate*, y*-coordinate*.

A variety of maps has been used to test the algorithms throughout this paper. Six maps were chosen to summarize the results. The terrain data topo3 and topo6 were added to test the worst case resulting from running Dijkstra algorithm on large regions. All maps are found in the appendix.

topo1.map}: The basic map used for initial research consisting of eight various, almost convex outcroppings.
topo2.map}: A map with many, very small, convex regions.
topo3.map}: Two major mountain regions shaped as simple polygons whose shape traps major regions inside their convex regions after preprocessing.
topo4.map}: Several long ridges as found in desert plain regions.
topo5.map}: A map with several large convex polygons.
topo6.map}: One large U-shaped ridge line that traps most of the planar map inside its convex polygon after preprocessing.

XXIII. Preprocessing

A preprocessing step represents a one-time fixed cost that must be taken into account when evaluating an algorithm. In many applications this preprocessing can be completed in advance and saved, thus save time during queries.

Simply adding the fixed preprocessing cost into the search time can be deceiving. If only one search is performed and the preprocessing time is added to the search time, the preprocessing will seem detrimental. As the number of searches increase the fixed cost is divided over the number of searches performed. The preprocessing time per search will approach zero as the number of searches approaches ∞.

XXIV. Search

Searches were performed and the results are summarized in Table II. An analysis of the running time between the two algorithms showed an average decrease by a factor of about four. Using the running time as a metric for performance can also be deceiving, since the running time is implementation and machine dependent. The measure of the number of nodes explored was used because it is not implementation specific and offers a more scientific measurement.

Table II lists a number of searches performed on each of the six maps. The specific searches were used to illustrate specific principles and issues. The first column of the table is the terrain map. The second and third columns are the start and goal points on the specific map and are illustrated in the Appendix. The path costs are the costs that were found for specific paths. The unimproved geometric search cost (UGMD) was included to show the performance improvement of the postprocessing step. The percent change is the increased cost of a path found by the geometrically modified Dijkstra (GMD) over the standard Dijkstra's algorithm (D).

Three extreme cases are very interesting. Topo5 search from $A \rightarrow B$ represents a path across the diagonal of the map that does not intersect any polygon. In this case Dijkstra explored 99% of the map to find the path, while geometrically guided algorithm only explored one node, the start point. The path cost is very close to the optimal path. Topo3 search $E \rightarrow F$ performs two searches to get out of two polygons, and combines these paths with a line segment. The cost represents the worst case, often higher than the Dijkstra path. Topo6 search $C \rightarrow D$ represents both the start and goal inside a single polygon, so the resultant path is equal to the standard Dijkstra algorithm.

The results in Table III are from 200 random searches on each input map. The results illustrated in Table III show a slight increase in cost. The increase in path cost was minimal on maps with smaller high-cost regions (1, 2, 4), with the geometrically guided algorithm returning near optimal paths. On these maps' paths found through the high-cost regions, when combined with line segments outside these regions, approach optimal paths.

Elevation arrays approximate the real terrain, and errors are introduced into the elevation array during the geocoding process. Node searching introduces digitization bias on the path, and rounding errors are introduced when working with real numbers on digital systems. Since the actual input and processing produce rough approximations to the actual terrain and paths, the percentage of change in search cost was not significant.

The search effort, as measured by the number of nodes explored, was very small on sparse maps. Long, relatively flat regions were "jumped over" quickly rather than searched one node at a time. The algorithm jumps from the start to nodes on the perimeter of obstacles in the direction of the goal. The algorithm does not explore any nodes except those on the perimeter or inside obstacles, resulting in a tremendous effort reduction on sparse maps.

In sparse maps the search quickly found line segments to paths outside polygons and could perform quick heuristic searches in the small convex polygons. The effort for these searches was very small

TABLE II
SEARCH RESULTS

Terrain Map	Search start x,y	Search goal x,y	Path Cost UGMD	Path Cost GMD	Path Cost D	Path Cost % change D to GMD	Nodes Examined GMD	Nodes Examined D	Nodes Examined Ratio D:GMD	Fig.
Topo1	A(5,5)	B(85,55)	103.46	103.5	100.71	2.66	15	1537	101.47	
	C(30,5)	D(50,40)	68.44	68.44	64.93	5.13	4	2489	621.25	5.3
	E(10,20)	F(80,20)	119.55	118.7	118.67	0	10	4850	484.00	5.2
	G(5,30)	H(85,30)	136.56	136.6	131.18	3.94	10	4905	489.50	
Topo2	A(5,5)	B(85,55)	155.19	155.2	151.07	2.65	9	5351	593.56	
	C(10,21)	D(80,21)	108.73	107.4	107.41	0	11	4477	406.00	
	E(20,35)	F(65,25)	95.06	92.06	85.2	7.45	30	3946	130.53	
	G(20,20)	H(40,55)	64.66	64.66	64.66	0	13	3286	251.77	
Topo3	A(5,5)	B(85,55)	172.88	170.4	170.4	0	7	5365	765.43	
	C(30,30)	D(60,30)	53.25	53.25	53.25	0	90	1848	19.53	
	E(35,20)	F(60,40)	83.33	82.8	75.06	9.35	636	3392	4.33	
	G(5,7)	B	178.35	178.4	156.57	12.21	5	5352	1069.40	
Topo4	A(5,25)	B(85,25)	146.65	139.3	135.19	2.92	13	5265	404.00	
	C(10,18)	D(85,55)	148.22	141	139.22	1.24	27	5352	197.22	
	E(45,2)	F(45,40)	70.97	67.97	67.09	1.29	7	2628	374.43	
	G(70,40)	H(5,20)	109.93	109.9	109.93	0	10	5010	500.00	
Topo5	A(5,5)	B(85,55)	158.1	155.5	151.07	2.82	1	5352	5351.00	
	C(25,40)	D(80,5)	139.05	139.1	130.26	6.32	76	5266	68.29	
	C	E(40,20)	101.12	80.49	76.97	4.37	111	2607	22.49	5.4
	F(5,40)	G(82,34)	138.9	137.1	136.26	0.64	13	4916	377.15	
Topo6	A(5,5)	B(85,55)	182.7	182.7	180.28	1.32	4	5369	1341.25	
	C(25,40)	D(55,40)	77.71	77.71	77.71	0	2515	2515	0.00	
	E(40,35)	F(55,55)	51.62	39.32	39.32	0	662	1166	0.76	
	E	G(5,55)	82.5	78.11	70.2	10.13	662	3264	3.93	

UGMD-- unimproved geometrically modified Dijkstra's algorithm
GMD -- postprocessed geometrically modified Dijkstra's algorithm
D -- Dijkstra's algorithm

TABLE III
COMPARISON OF 200 RUNS OF GMD VERSUS DIJKSTRA SEA

Map	Increase in path cost
topo1	5.48%
topo2	0.62%
topo3	7.50%
topo4	2.32%
topo5	7.49%
topo6	11.91%
Average	6.05%

because sparse areas are quickly *jumped over* rather than searched one node at a time.

Fig. 13 is a typical search where neither start nor end points are inside convex polygons. The cost between Dijkstra and geometric search are close, but the path returned from the geometrically guided search (dark line) is a smoother, direct route.

When a line segment projected from the start to the goal intersected a polygon the algorithm found three paths; left, right, and through the polygon. The right path (from C to D) was found to have a lower cost. The path is a line segment from the start point to the right tangent point, to the goal.

Fig. 14 is an example of a path with both the start and goal inside polygons. Dijkstra's algorithm returns an optimal path. The geometrically guided algorithm quickly found a path out of the polygons to reduce the search effort, and completes the route to the goal. The postprocessing determined that it could prune off some of the route making the global path shorter and more direct to the goal.

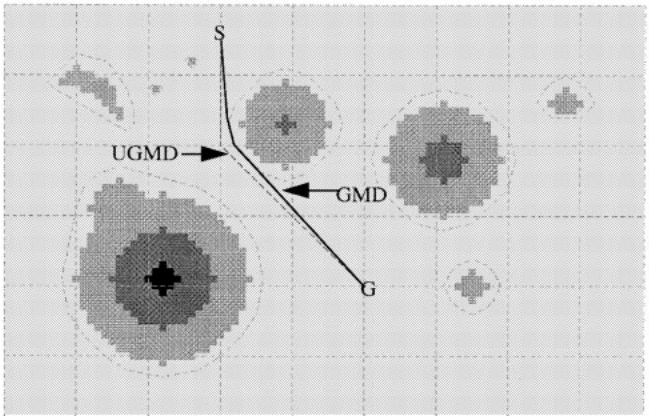

Fig. 13. Path avoiding polygons.

The worst cost paths were those with large convex polygon regions. This performance is attributed to the large node-generating searches that must be performs, in addition to the overhead of the geoguided search. On these regions, the preprocessing bounds large areas of low-cost regions inside perimeters when it converts polygons from simple to convex.

If both the start and goal are in the same polygon, then the search is simply a Dijkstra search (Fig. 15). If the start and goal are outside these regions, the algorithm quickly finds a path around without searching the entire map. If a start or goal are inside a regions a Dijkstra search is performed to find a path quickly out of the polygon to a point visible from the goal, and this path is connected with other paths to create a solution path. This connecting of path segments can result in high-cost solution paths if optimal subpaths do not combine to an optimal global path.

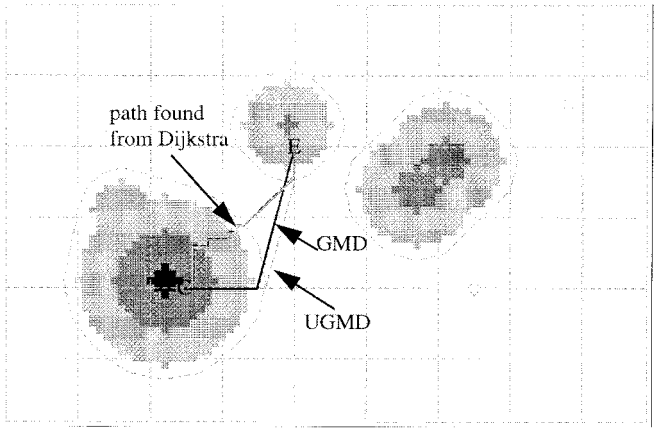

Fig. 14. Start and end inside polygons.

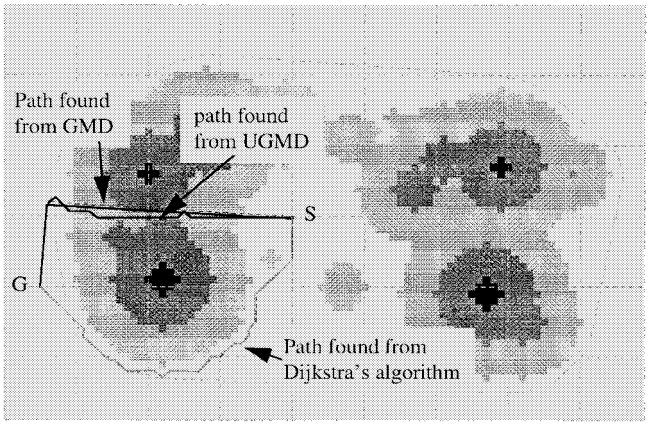

Fig. 15. Search with large high-cost regions.

In maps with many regions the geometrically guided search proved a dramatic improvement by decreasing the number of nodes that must be explored. This is because only nodes on the perimeter of polygons that are in the direction of the goal can be labeled, resulting in a tremendous reduction in the number of nodes explored.

XXV. Postprocessing

The path found in step-two of the geoguided search algorithm may consist of many locally optimal paths, but may be far from globally optimal. The algorithm will find a locally optimal path out of a high-cost region to the boundary of that region. Next, the algorithm will find a locally optimal path from the boundary to the goal. The combined resultant path though, may be far from globally optimal. The solution is a postprocessing step that examines the entire path, and were possible, improves the global path.

In Fig. 16 a path was quickly found out of the high-cost region to the boundary, and from the boundary to the goal. This path is far from optimal. The path-improvement step is applied. This step prunes center nodes out of ordered, consecutive triples of path points, if the resultant path is shorter.

The postprocessing step was found particularly beneficial on terrain maps that have large concave regions (Fig. 17). On these maps, the postprocessing step will straighten the global paths through the low-cost portions of the bounded region. Additionally, the postprocessing step will prune the path, skirting the perimeter of a polygon or passing over small ridges inside a bounded region, if a start or goal is inside the region. These path improvements were found to improve the quality of the subsequent paths, sometimes resulting in a globally optimal path.

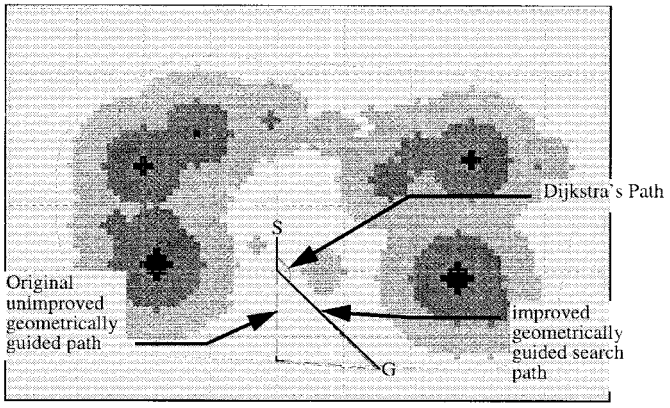

Fig. 16. Example of postprocessing.

XXVI. Conclusion/Recommendations for Future Work

Searching on topographical maps continues to be an interesting problem. The problem offers many challenges due to the diversity of the earth's terrain, man-made modifications to the natural terrain, and the applications for which a path is being found. Current research continues to investigate the shortest path problem in many fields of study.

The three-step approach applied in this paper proved very effective. From the experimental results the paper can be summarized as follows:

- A quick preprocessing step provides the main search routine with a small amount of very important global information that can dramatically reduce the search effort and time.
- The additional space needed for the preprocessed information is offset by the reduced amount of space needed during the main search routine. The preprocessed path costs eliminated repetitive searches through rough terrain that is very costly to search.
- The main search routine jumps over flat terrain and does not waste time and space searching flat terrain. This jumping across flat terrain saves tremendously on the search effort.
- The postprocessing step provides global path improvement and path straightening. The solution from the postprocessing may take a significant smaller amount of space since it transforms straight paths of nodes to a line segment of two nodes.

A possible direction for future work is precomputing implicit roads. The motivation behind precomputing implicit roads is that on certain terrain, paths gravitate toward locations that are easy to travel. These paths are similar to trails made in the wilderness, where man and beast tend to travel on certain types of terrain. The implicit roads algorithm used a precomputation step that recursively partitions the landscape into smaller and smaller pieces along these implicit roads. The solution of the precomputing created a network of implicit roads similar to skeletonization. When a path is needed between two points, the algorithm finds a short path from the source to a road, travel through these precomputed implicit roads, and compute a path from a road to the goal.

This paper introduced many other unanswered questions that are open to the inquisitive researcher. These areas include:

⇒ **Conversion to Contour Intervals**—By repeated calls to the precomputing step for an increasing value of δ, the elevation array topographical map is transformed into a plane with concentric, closed (simple or complex) contour intervals. These intervals bound regions whose elevations are greater than the

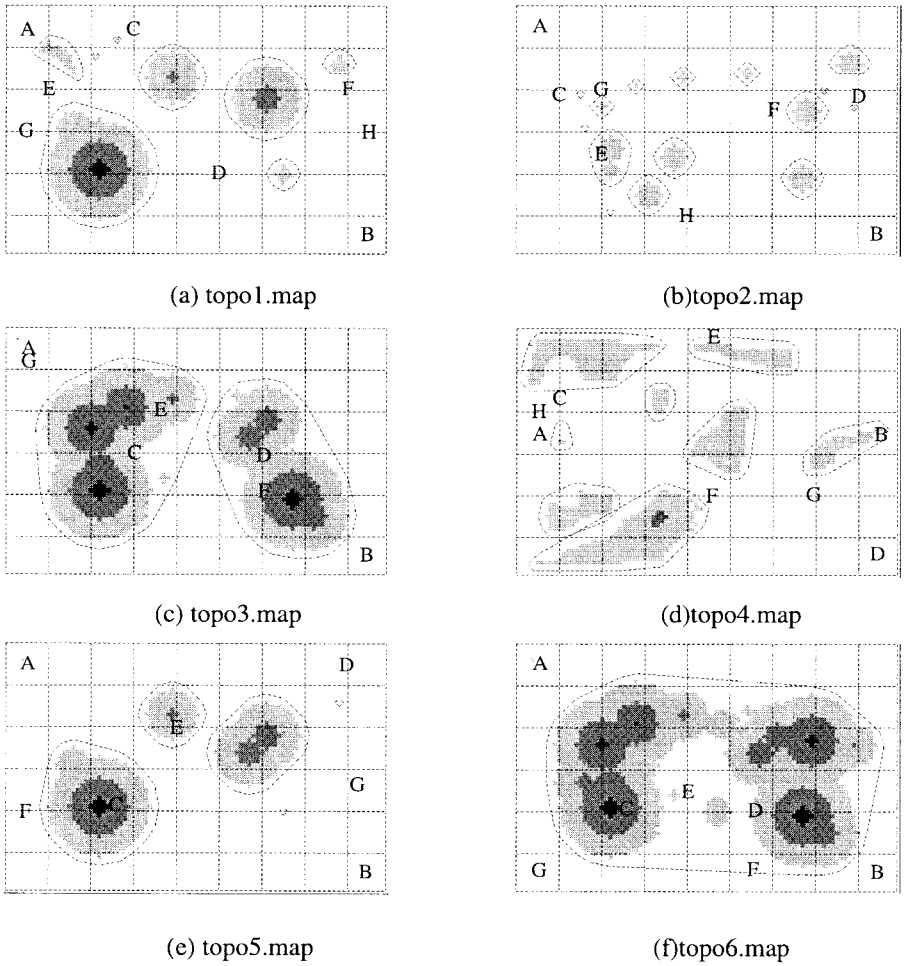

Fig. 17. Terrain maps.

contour value of δ. The search does not use the elevation array, so after precomputing it can be permanently removed from memory. The solution to a path query is solved as a computational geometry problem.

⇒ **Shortest Path, Intermediate Point Data Structure**—Any path segment contained inside a shortest path is itself a shortest path. Using this observation, it makes sense to store solution paths into an efficient data structure that can be quickly searched. When solving a query the data structure is searched and returns the solution if found. If the solution is not found then the problem is solved and the data structure is updated with the solution. All paths cannot be stored efficiently as mentioned earlier ($O(N^2)$ for N points on a map), so it makes sense to store special paths, perhaps implicit roads. Using this data structure, not only the shortest path costs, but also the actual paths could be stored, eliminating any Dijkstra search through polygons.

⇒ **Link Metric**—Many applications are constrained by solution paths that have a minimal number of bends or maximum angles of bends. Autonomous robot motion planning and high-performance aircraft flight planning are examples of these applications. Examination and comparison of algorithms based on the number and extent of bends in the paths provides a rich area of research that is just beginning to be explored.

⇒ **Additional Topographical Map Data**—The input for this research was simplified to create a manageable search problem. Additional man-made terrain features can be included to increase the complexity of the problem. Typical terrain features include roads, rivers, bridges, fences, and tunnels.

⇒ **Preprocessing on Demand**—Rather than preprocess all shortest paths through polygons, process them as needed and store them. This would resemble the system learning the shortest paths through polygons and remembering them for future use. This would eliminate the preprocessing costs, and slow initial searches, but would result in searches only performed for actual paths needed.

⇒ **Point inside Polygon**—The current heuristics do not perform well all of the time when a start or goal are inside a polygon. There are many difficulties that arise on real terrain when computing the shortest path with large regions, providing an additional area for future research.

REFERENCES

[1] C. P. Bonnington, *The Foundations of Topological Graph Theory*. New York: Springer-Verlag, 1995.

[2] K. C. Clarke, *Analytical and Computer Cartography*. Englewood Cliffs, NJ: Prentice-Hall, 1990.

[3] A. Gross and L. Longin, "Digitizations preserving topological and differential geometric properties," in *Proc. SPIE, Vision Geometry III*, Nov. 2–3, 1994, vol. 2356, pp. 34–35.

[4] G. F. Luger, *Artificial Intelligence, Strategies and Structured for Complex Problem Solving*. New York: Benjamin Cummings, 1993.

[5] J. S. B. Mitchell, "An algorithm approach to some problems in terrain navigation," *Artif. Intell.*, vol. 37, pp. 171–201, Dec. 1988.

[6] K. Fujimura, *Motion Planning in Dynamic Environments*. New York: Springer-Verlag, 1991.

[7] W. L. Winston, *Operations Research: Applications and Algorithms*. Boston, MA: Duxbury, 1987.

[8] S. Russel and P. Norvig, *Artificial Intelligence, A Modern Approach*. Englewood Cliffs, NJ: Prentice-Hall, 1995.

[9] S. Tanimoto and A. Klinger, *Structured Computer Vision—Machine Perception Through Hierarchical Computation Structures*. New York: Academic, 1980.

[10] F. P. Preparata and M. I. Shamos, *Computational Geometry an Introduction*. New York: Springer, 1985.

[11] J. Pearl, *Heuristics, Intelligent Search Strategies for Computer Problem Solving*. Reading, MA: Addison-Wesley, 1984.

[12] Manzini and Giovanni, "BIDA*: An improved perimeter search algorithm," *Artif. Intell.*, vol. 75, pp. 347–360, 1995.

[13] R. Agrawal and H. V. Jagadish, "Algorithms for searching massive graphs," *IEEE Trans. Knowledge Data Eng.*, vol. 16, pp. 225–238, Apr. 1994.

[14] J. Latombe, *Robot Motion Planning*. Norwell, MA: Kluwer, 1991.

[15] H. P. Moravec, *Robot Rover Visual Navigation*. East Lansing, MI: Univ. Michigan Press, 1981.

[16] K. Fan and L. Po-Chang, "Solving find-path problem in mapped environment using modified A^* algorithm," *IEEE Trans. Syst., Man, Cybern.*, vol. 24, pp. 1390–1397, Sept. 1994.

[17] J. L. Crowley, "Navigation for an intelligent mobile robot," *IEEE J. Robot. Automat.*, vol. RA-1, pp. 31–41, Mar. 1985.

[18] J. O'Rourke, *Computational Geometry in C*. Cambridge, U.K.: Univ. of Cambridge Press, 1993.

[19] J. T. Schwartz, M. Sharir, and J. Hopcroft, *Planning, Geometry, and Complexity*. New York: Ablex, 1987.

[20] J. T. Schwartz and M. Sharir, "A survey of motion planning and related geometric algorithms," *Artif. Intell.*, vol. 37, pp. 157–169, Dec. 1988.

[21] Y. K. Hwang, "Gross motion planning—A survey," *ACM Comput.*, vol. 24, pp. 219–291, Sept. 1992.

[22] A. Kelley and I. Pohl, *A Book on C*. New York: Benjamin Cummings, 1995.

[23] J. A. Storer, "Shortest paths in the plane with polygonal obstacles," *J. ACM*, vol. 41, pp. 982–1012, 1994.

[24] J. F. Dillenburg and P. C. Nelson, "Perimeter search," *Artif. Intell.*, vol. 65, pp. 165–178, 1994.

[25] Environmental Systems Research Institute, *Cell-Based Modeling with GRID*, 1991.

[26] M. Gondran and M. Minoux, *Graph and Algorithms*. Paris, France: Wiley, 1984.

[27] R. Kimmel and N. Kiryati, "Finding shortest paths on surfaces by fast global approximation and precise local refinement," *SPIE Vision Geometry III*, vol. 2356, pp. 198–209, 1994.

[28] R. Korf, "Linear space best-first search," *Artif. Intell.*, vol. 62, pp. 41–78, 1993.

[29] J. Reif, "A single-exponential upper bound for finding shortest path on three dimensions," *J. ACM*, vol. 41, pp. 1013–1019, Sept. 1994.

[30] T. Cormen, *Introduction to Algorithms*. Cambridge, MA: MIT Press, 1990.