Stochastic Search and Graph Techniques for MCM Path Planning

Christine D. Piatko, Christopher P. Diehl, Paul McNamee, Cheryl Resch and I-Jeng Wang[*]

The Johns Hopkins University Applied Physics Laboratory, Laurel, MD 20723-6099

ABSTRACT

We have been developing path planning techniques to look for paths that balance the utility and risk associated with different routes through a minefield. Such methods will allow a battlegroup commander to evaluate alternative route options while searching for low risk paths. A risk management framework can be used to describe the relative values of different factors such as risk versus time to objective, giving the commander the capability to balance path safety against other mission objectives. We will describe our recent investigations of two related path planning problems in this framework. We have developed a stochastic search technique to identify low risk paths that satisfy a constraint on the transit time. The objective is to generate low risk paths quickly so that the user can interactively explore the time-risk tradeoff. We will compare this with the related problem of finding the fastest bounded-risk path, and the potential use of dynamic graph algorithms to quickly find new paths as the risk bound is varied.

Keywords: mines, path planning, risk management, optimization

1. OVERVIEW

Mine and minefield data is probabilistic due to many factors such as position uncertainty, detection probabilities, classification confidence levels, and classification errors. In the face of this uncertainty, a commander needs a meaningful way to measure risk in order to determine the best way to cross a potential minefield, or decide that the risk to cross is too great. Effective path planning techniques can provide valuable information in a decision aid for mine countermeasures (MCM).

Path planning is a well-studied problem in the setting where there is a single criterion to optimize. It is well known how to find the path of shortest distance, and well known how to find the path of least cost (risk). However, to optimize both criteria at once is a challenging problem. Instead of single path quality of length/time-to-objective, we want to include other optimization criteria that describe the desirability of paths such as level of risk. In addition, it is desirable for command and control to have alternative routes presented to help make a decision. Therefore, besides a single optimal path we would like to find several choices of paths to display as alternatives. The most common path quality attribute is path length or time to goal, as a model of the time required to achieve a successful landing. Another important quality that we model is the probability of a successful landing, given the path risk from mines.

In previous work (Piatko et al. 2001) we described initial efforts on this problem using an A* planner. In this paper we present our recent implementation experience using Dijkstra's algorithm for this problem. We also investigate the use of dynamic graph algorithms to efficiently update the shortest path when simulating the removal of a single mine. We have developed methods to address a new problem of minimizing risk while meeting a hard constraint on time to goal.

1.1 Grid graph model for path planning

We discretize the minefield into a graph, with vertices evenly distributed over the minefield. We assume a start vertex *s* and a goal vertex *g*. Risk values r_i are associated with each gridpoint vertex *i* according to its proximity to mines. Factors to consider for developing a risk model can be found in (Piatko et al. 2001 and Wang et al. 2001). Lengths l_{ij} represent the estimated length/time to traverse between two adjacent gridpoints *i* and *j*. Eight edges emerge from each vertex. The edges are directional, and are defined by the "from" vertex, the "to" vertex, and the weight, or cost of traversing the edge. The weight of each edge is found using the following equation:

[*] Christine.Piatko@jhuapl.edu; phone 1 443 778-6584; Chris.Diehl@jhuapl.edu; Paul.McNamee@jhuapl.edu; Cheryl.Resch@jhuapl.edu; and I-Jeng.Wang@jhuapl.edu.

$$W_{ij} = \alpha\, r_{\text{to}} + (1-\alpha)\ l_{ij}\ ,$$

where r_{to} is the risk value at the "to" vertex, and l_{ij} is the distance from the "from" vertex to the "to" vertex. This distance is $\sqrt{2}$ for diagonal edges, and 1 otherwise. The parameter α is set by the user, and is a risk tolerance factor. Setting α to zero corresponds to searching for the shortest distance path; setting α to one corresponds to searching for the lowest risk path without regard to distance.

1.2 Linear combination of time and risk

One approach to making multicriteria path planning problems tractable is to reduce them to single metric problems. For example, we can use a linear combination of the path quality metrics of path length and risk, and find the best path using this single metric.

Given a starting location *s* and a desired goal *g*, this corresponds to finding the path p that minimizes the value $O(p) = \sum_{ij\in p} W_{ij}$. This can be computed optimally in polynomial time using Dijkstra's algorithm for fixed α.

1.3 Fixed constraint on risk

Another approach to making multicriteria path planning problems tractable is to fix side criteria as constraints. The user or application can then explore various values for the side constraints. For mines path planning, one such constrained problem is to find the shortest length path with risk less than a fixed bound *R*. This corresponds to finding the path *p* that minimizes the value $O(p) = \sum_{ij\in p} l_{ij}$ (in this case (1-α) is zero), such that each vertex along the path has risk $r_i \leq R$.

This can also be computed optimally in polynomial time using Dijkstra's algorithm, by using the subgraph of the full grid graph that does not consider any edges that violate the risk threshold constraint.

1.4 Fixed constraint on time/length

An alternative way to fix a side criterion for mines path planning is to find the shortest path with risk less than a fixed time constraint *T*. Assuming constant velocity, fixed time corresponds to a fixed length *L*. For this version, we wish to find the path *p* that minimizes the value $O(p) = \sum_{ij\in p} r_{ij}$ (in this case α is zero) such that the total path length $l(p) = \sum_{ij\in p} l_{ij}$ is less than *L*. This is harder than the fixed constraint on risk problem above, because the side constraint is on a sum rather than on the individual edges of the path. While this version of the problem is NP-hard in general, using a restricted version of the grid graph that uses only horizontal and vertical edges (and thus unit lengths) we can solve this problem in polynomial time. We also present a faster stochastic search algorithm that finds locally reasonable paths.

1.5 Waypoints and multiple goals

Each of the above path-planning problems can be generalized to include specified intermediate waypoints $w_1 ... w_m$ along the path. In this case, we plan the shortest path from *s* to the intermediate goal w_1 , from w_1 to w_2 , and so on, until reaching w_m and then planning to final goal *g*. Similarly, we can generalize to plan paths to multiple goalpoints.

2. DIJKSTRA'S ALGORITHM FOR PATH-PLANNING

We can use Dijkstra's algorithm to find the shortest path between waypoints. In previous work, we described our efforts using a Java implementation of an A* planner for the linear combination and fixed constraint on risk problems. We have redone this original planner to use a static Dijkstra's implementation. In the next section, we also describe our results from coding a dynamic version to update Dijkstra shortest paths as the graph is modified. To date, we have found A* and static Dijkstra to be comparable in terms of running time. Our particular implementation of Dijkstra seems to produce more aesthetic paths than our A* implementation.

First, the graph for the minefield must be built and all the weights for the edges calculated for the particular choice of α. Dijkstra's algorithm then proceeds as follows (see also Cormen, Leiserson, and Rivest 1997 and Dijkstra 1955). The vertex closest to the “starting” waypoint is first considered, and given a path cost of zero. This “starting” vertex is put then in a set of vertices for which cost has been calculated (this set has only one vertex in it at this point). Next, all the vertices connected by an edge to that vertex are put in a set to be considered. The vertex with the lowest cost, i.e., the lowest edge weight, is assigned a cost equal to the edge weight between the source and that vertex. That vertex is then put in the set of vertices for which cost has been calculated. Next all vertices connected to the vertex for which the cost was just calculated are put in the set to be considered. Again, the vertex with the lowest cost from the source is considered next. The cost is the sum of the edge weights from the source to the vertex. This vertex is assigned that cost and put in the set of vertices for which the cost has been calculated, and all vertices connected to the vertex are added to the set of vertices to be considered, unless the vertex is already in that set. The algorithm ends when each vertex has been considered, i.e., the cost from the source to each vertex has been calculated. Since this algorithm computes the shortest path to all vertices, it can be used to compute the shortest path to multiple goal points. The process can also be terminated if there is just a single goal *g* when that particular vertex is reached. Figure 1 shows an instance of Dijkstra’s algorithm. For each vertex, the sum of the edge weights along the shortest path from the source can be saved, as well as the previous vertex along the path from the source to that vertex. This gives an encoding of the shortest path tree from the start vertex that can be used to reconstruct the vertices along the shortest path. The path from the “starting” waypoint to the “goal” waypoint is found by locating the vertex closest to the goal, then stepping through the previous vertices until the source is reached.

Figure 2 shows a snapshot of a path through a minefield calculated using Dijkstra’s algorithm. Dijkstra’s algorithm can be implemented to have a running time of O((V + E) log V), where V is the number of vertices and E is the number of edges.

3. DYNAMIC GRAPH UPDATES TO SIMULATE MINE NEUTRALIZATION EFFECTS

We can simulate the removal or neutralization of a single mine by recomputing the risk of affected edges in the neighborhood of a mine, and recomputing the shortest path from the start to the goal. This enables interactive exploration of which mines or minefields are best to neutralize. However, recomputing the shortest path using Dijkstra’s algorithm can be expensive for a large grid graph. Fortunately, Dijkstra needs to be run in full just once for each set of waypoints and each value of α. Recent results have shown how to efficiently maintain shortest paths in a dynamic setting, where sets of edge insertions, deletions or cost changes occur. An algorithm by Ramalingam and Reps [Ramalingam and Reps 1996] can be used to update the shortest paths efficiently. Only those vertices whose shortest path cost changes because of the change in the risk field due to the removed mine need to be updated. Such methods can also be used to support dynamic changes on bounds of acceptable risk for a path. They can also support other dynamic aspects of path planning, including incorporating updated information about mines.

When a mine is removed, the weight of each edge in that affected area of the risk field is recalculated. The “to” vertex of each edge whose weight has changed is put into a set of vertices to be updated. With each of these vertices is saved the old cost of the shortest path from the source to that vertex, and the new cost of the path when the cost of just that one edge is changed. These vertices are considered according to the lowest new path cost from the source to that vertex, much the same way as in Dijkstra’s algorithm. As vertices are considered, their lowest path cost is updated, and they are removed from the set of vertices to be considered. After the vertex is considered, the cost of the shortest path to all vertices connected by an edge to the considered vertex is compared to the cost that would result from a new path through the updated vertex. If the new cost to that vertex is lower than the previous saved cost, that vertex is added to the set of vertices to be updated. When the set of vertices to be updated is empty, the algorithm terminates. Thus, only vertices whose shortest path changes are updated, instead of all vertices. Figure 3 shows an instance of Ramalingam and Reps’ algorithm.

This updating algorithm has a running time of O(A log A), where A is the sum of affected vertices and the edges into these vertices. The closer the removed mine is to the source, the larger A is, and the longer the running time of the algorithm.
We implemented Ramalingam and Reps’ algorithm to update the shortest path when a single mine is removed, and compared this to the cost of redoing Dijkstra’s algorithm for the entire graph each time. Figures 4 and 5 show some results from these experiments with our implementations.
We randomly generated 10 different minefields for each of 6 different grid resolutions (250 by 250, 300 by 300, and so on up to 500 by 500). Each minefield contained up to 10 randomly generated minelines, each with 5 to 15 mines. The

simulated minefield also contained up to 100 randomly placed “noise” points (with lower mean posterior probabilities than the mines). We also generated a random start vertex to the “left” of the minefield and a random goal to the “right.” We used the linear combination metric with fixed α of 0.1 for these experiments. We computed the shortest path from the start to goal using Dijkstra’s algorithm. We simulated two kinds of mine deletion/neutralization, a mine near the start vertex and one farther away. This captures the effect of different numbers of affected vertices on the running times.

Figure 4 shows the 120 individual data points of these experiments, showing the time to update the shortest paths as a function of the distance to the source for various graph sizes. The trend is that removing a mine close to the source takes longer to update, since more of the shortest path tree is affected.

Figure 5 compares using the full Dijkstra’s algorithm versus the graph updating algorithm. It also shows more clearly the effect of increasing graph size. The upper curves show the mean and standard deviation of the running times to compute the initial shortest path using Dijkstra’s algorithm. We did not time using the full Dijkstra algorithm to compute the two mine deletion paths, but those running times would be similar. The lower curves show the mean and standard deviation of the update running times (using the same 120 experiments as in Figure 4). Note that there is a substantial time advantage to updating rather than computing the shortest path from scratch.

4. ALGORITHMS FOR FIXED TIME-TO-GOAL

The previous techniques can be used to identify paths that satisfy constraints on the acceptable amount of risk, or linear combinations of time-to-goal and risk.

We are also interested in the alternative constrained problem of finding low risk paths that satisfy a constraint on the transit time. Such methods would identify low risk paths through a minefield that allow a ship to arrive at its destination at a specified time. (We assume the ship traverses the path at a constant speed in order to simplify this planning problem.) That is, given a fixed bound L on the length of the path (fixed time-to-goal), we wish to find the lowest risk path.

4.1 Optimal algorithm for fixed time-to-goal

In general, the problem is NP-hard. However, in the grid case, with just 4 directions, L is discretized to exactly unit increments. The lowest risk path of length L can then be computed by using a modified version of a Bellman-Ford algorithm (Cormen et al. 1997). As described above, path risk is used for the objective (distance) function. Several modifications are necessary. We need to make *L* passes over the graph. As opposed to the usual asynchronous approach of Bellman-Ford (which overwrites distance information in the middle of each pass), we need to maintain both the previous phase *i-1* and current phase *i* distance information to each vertex. The previous distance information records the lowest risk path from the source *s* with *i-1* or fewer links to each vertex. Initially all such distances are set to infinity, except the source vertex distance is to itself is set to zero. At pass *I*, we initialize the *i* link distance to each vertex *v* with the value of the previous (*i-1* or fewer links) lowest risk path to *v*. Then, for each edge (*u*,*v*), we test if using this edge results in a lower risk path to *v* with *i* links; i.e., if the lowest risk path with *i-1* or fewer links to *u* plus the cost of edge (*u*,*v*) is less than the value of the current lowest risk path with *i* or fewer links to *v*. If so, we record the value of this new path of *i* or fewer links to *v*. After all edges have been considered, we have found the shortest risk path with *i* or fewer links to each vertex. (We have no negative cost edges and thus do not need to worry about negative cost cycles). After L such passes, if we have reached the goal vertex *g*, we have found the shortest risk path that takes <= L steps. If we wish to reconstruct the path, we also need to keep an auxiliary size *L* data structure at each vertex to record the predecessor vertex for each link value *i* (since predecessors are different for different values of *i*). This modified Bellman-Ford approach can be implemented in O(*k*E) time and O(*k*V) space for *k* phases. For our mines problem, this corresponds to O(LE) and O(LV) space.

This approach becomes even more expensive when adding more directions to the grid vertices. For 8 directions (adding the diagonals), a diagonal step adds $\sqrt{2}$ length, which are not equal to the vertical/horizontal steps. One option is to discretize each edge into finer increments and distribute the risk of the original edge proportionally on these finer edges. The goal is make each "step" small enough to be a unit increment as above. Since this is a bigger grid, this computation will be even slower. Another option would be to discretize the minefield into a hex grid. In this case, all steps would be of unit length.

4.2 Stochastic search for fixed time-to-goal

Since this optimal algorithm can be so computationally expensive, we developed a fast, randomized method finding the minimum risk path where there is a constraint on time-to-goal. In this case our objective is to define a path planner that generates low risk paths quickly so that the user can interactively explore the time-risk tradeoff. We also require the planner to produce alternatives to the lowest risk path as well as intermediate solutions when the planning process cannot be completed due to time constraints.

We use a population-based, multiresolution stochastic search to generate a number of low risk paths through a minefield. The basic idea is to generate random paths, locally improve them, and save the best ones. This type of approach is also being used for more general types of motion planning (Hocaglu and A. C. Sanderson 1998 and 2001). When the planner is executed, an initial population of candidate paths is created by randomly sampling the space of paths that deliver the ship to the destination at the appropriate time. The paths are encoded as symbol strings that indicate the sequence of moves through the grid overlaid on the minefield. As the planning proceeds, the resolution of the grid is incrementally increased. At each resolution level, the current population of paths is first randomly perturbed to explore the path space. Then the paths are deterministically perturbed into local minima. This process is repeated for a number of iterations at each resolution level. Then, the best paths in the current population are saved if they yield improvements over any of the lowest risk paths discovered previously. The advantage of this approach is that it quickly generates paths, including intermediate results. The main disadvantage is that it provides no guarantees of optimality on the resulting path.

Figure 6 shows a path generated by this planner coded in MATLAB. Initial results have been encouraging; many reasonable paths can be generated very quickly. It is also possible to use this method to generate alternative choices of paths. We also ported this planner to C++ to investigate the effectiveness of the process on larger planning tasks, and found that it scaled well.

5. CONCLUSIONS

We have developed efficient methods for several versions of mines path planning problems with multiple constraints. This work has demonstrated the effectiveness of such algorithms for mines path planning.

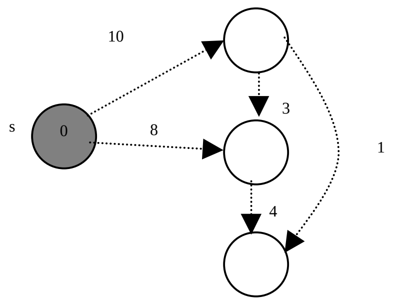
In future work, we would like to implement the polynomial time algorithm for the fast time-bounded problem to compare how close the stochastic approach comes to finding optimal paths. We also plan on using the Ramalingam and Reps algorithm to create a “risk slider” that uses batches of updates to allow a user to vary the risk threshold interactively. We also seek to develop methods to address the very interesting problem of deciding which *multiple* mines would be most effective to remove. Finally, there are a number of very recent results that use different techniques for constrained shortest path planning that could also prove valuable to use for MCM path planning applications (Mehlhorn and Ziegelmann 2001).

6. ACKNOWLEDGEMENTS

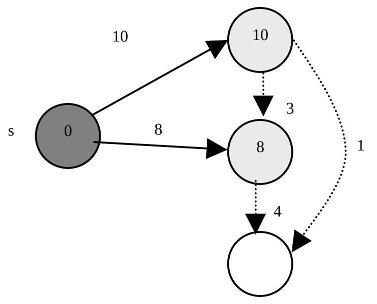
This work is partially supported by Office of Naval Research Contract N00024-98-D-8124. The authors thank Fernando Pineda of JHU/APL for his help developing the risk model. The authors also thank Richard Chang for his insights on modifying the Bellman-Ford algorithm.

7. REFERENCES

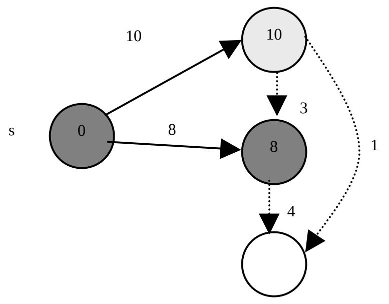
1. T. H. Cormen, C. E. Leiserson, and R. L. Rivest, *Introduction to Algorithms*, McGraw-Hill, New York, 1997.
2. E. W. Dijkstra, "A note on two problems in connexion with graphs." In *Numerische Mathematik*, Vol. 1, pp. 269-271, 1955.
3. C. Hocaglu and A. C. Sanderson, "Evolutionary Path Planning using Multiresolution Path Representation." In Proceedings of the 1998 IEEE International Conference on Robotics and Automation, pp. 318-323, May 1998.
4. C. Hocaglu and A. C. Sanderson, "Planning Multiple Paths with Evolutionary Speciation." In IEEE Transactions on Evolutionary Computation, Vol. 5, No. 3, June 2001.
5. K. Mehlhorn and M. Ziegelmann, "CNOP – A Package for Constrained Network Optimization." In ALENEX 2001, Algorithm Engineering and Experimentation: Third International Workshop (ALENEX 2001), Lecture Notes in Computer Science, Volume 2153, Spring Verlag, 2001.
6. C. D. Piatko, C. Priebe, L. Cowen, I.-J. Wang, and P. McNamee, "Path Planning for Mine Countermeasures Command and Control." In Proceedings of the SPIE Volume 4394, Detection and Remediation Technologies for Mines and Minelike Targets VI, Orlando, Florida, pp. 836-843, 2001.
7. G. Ramalingam and T. Reps, "An Incremental Algorithm for a Generalization of the Shortest-Path Problem." In Journal of Algorithms, Vol. 21, No. 2, pp. 267-305, 1996.
8. I-J. Wang, C. D. Piatko, and F. Pineda, "Risk Models for Mine Countermeasures," manuscript, 2001.



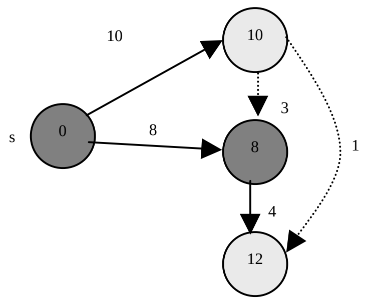
Step one - set cost to source to zero and put in set of vertices for which path is calculated



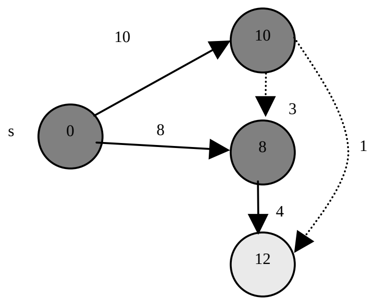
Step two - add vertices connected to source to set to be considered



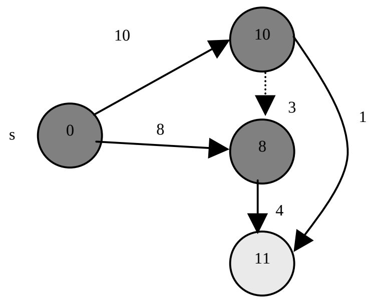
Step three - set cost for vertex with lowest value



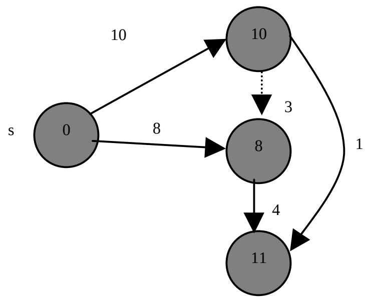
Step four - add for consideration vertices adjacent to the vertex just calculated



Step five - set cost for next lowest vertex



Step six - add for consideration vertices adjacent to the vertex just calculated



Step seven - set cost for next lowest vertex

Figure 1. An Instance of Dijkstra's Algorithm

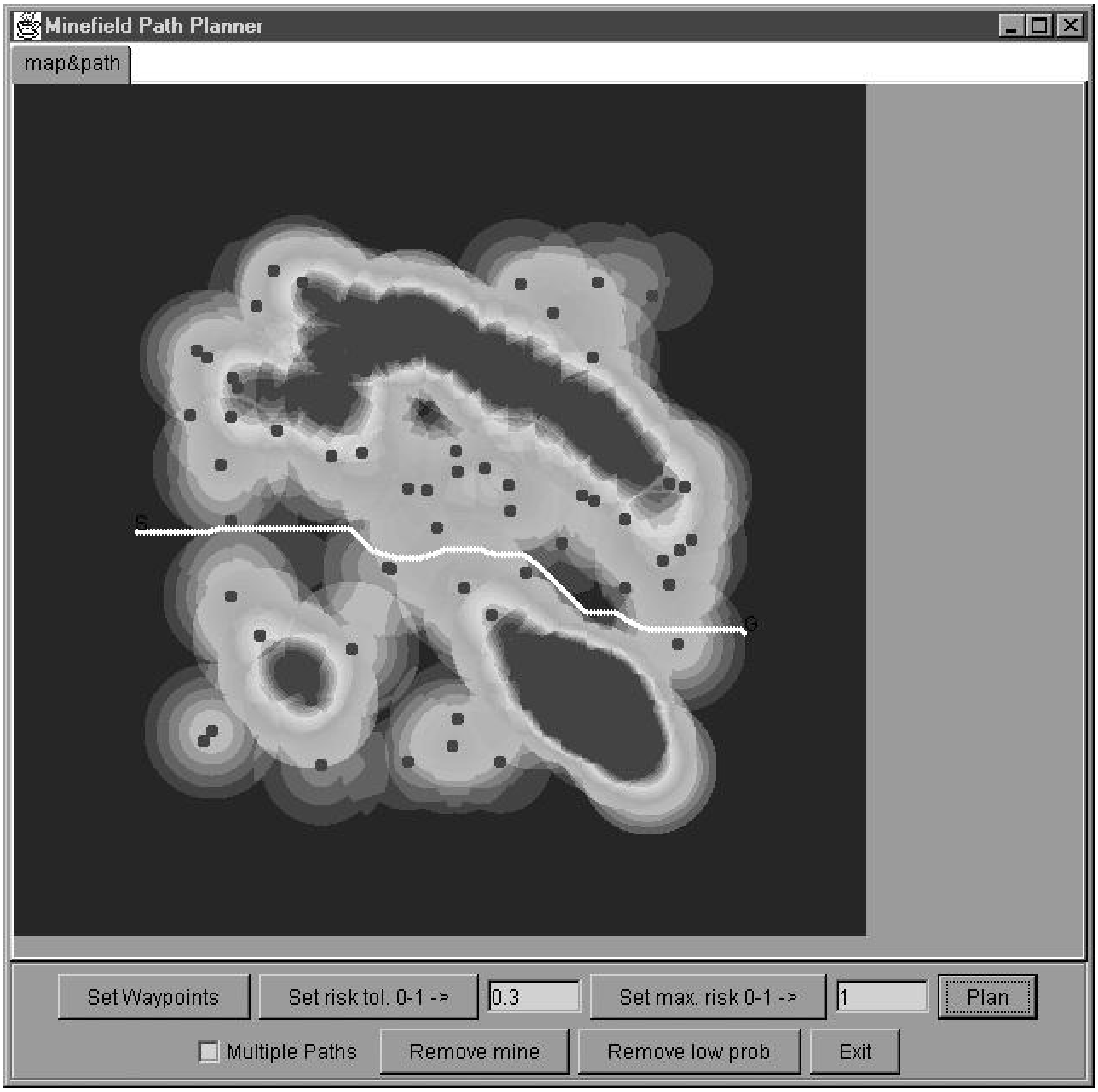
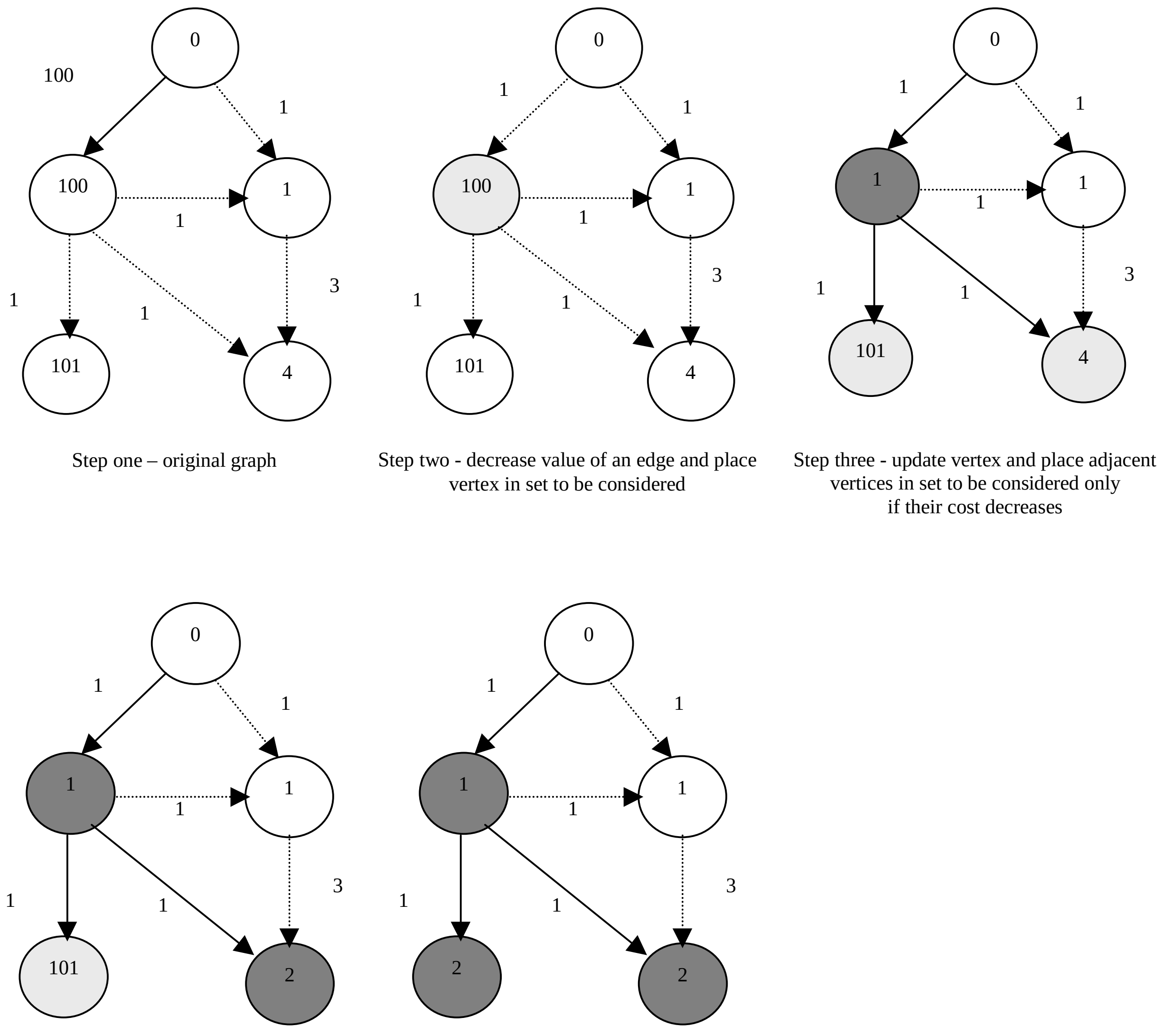


Figure 2. A snapshot of the Dijkstra mines path



Step one – original graph

Step two - decrease value of an edge and place vertex in set to be considered

Step three - update vertex and place adjacent vertices in set to be considered only if their cost decreases

Step four - update vertex with lowest new value and add adjacent vertices to set, if necessary

Repeat Step four until set to be considered is empty

Figure 3. An instance of Ramalingam and Reps' Algorithm

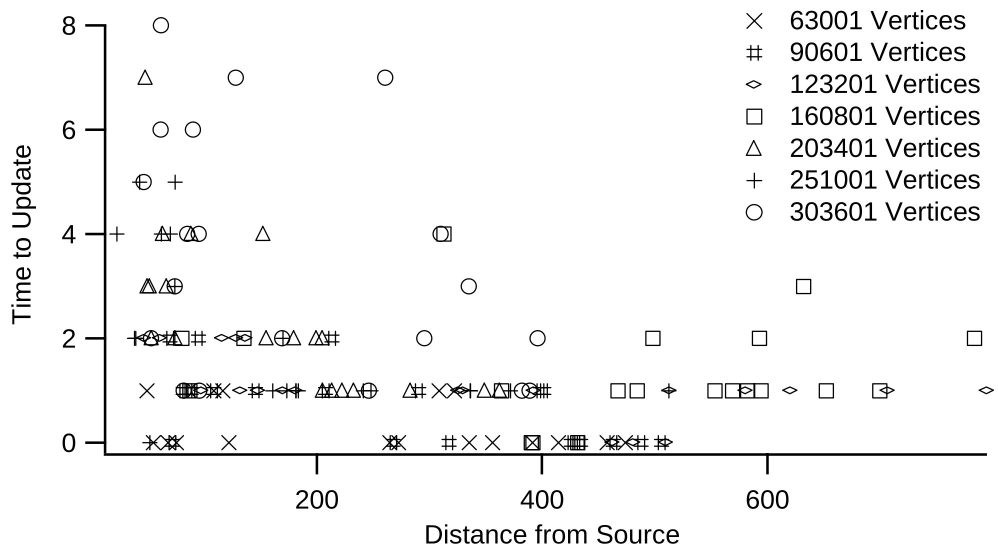


Figure 4. The effect of mine location on update time.

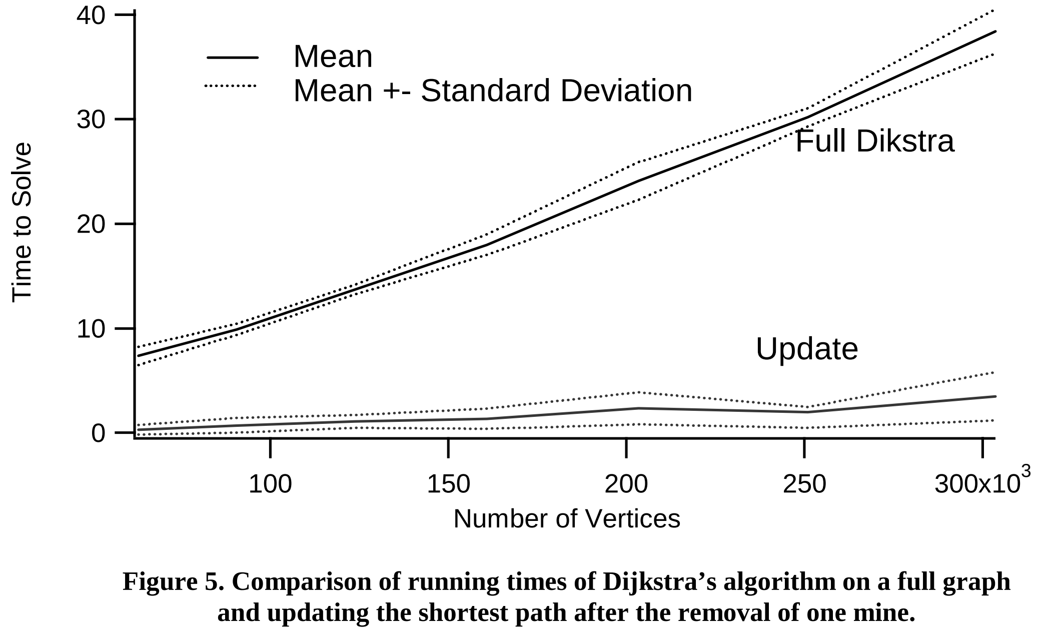


Figure 5. Comparison of running times of Dijkstra's algorithm on a full graph and updating the shortest path after the removal of one mine.

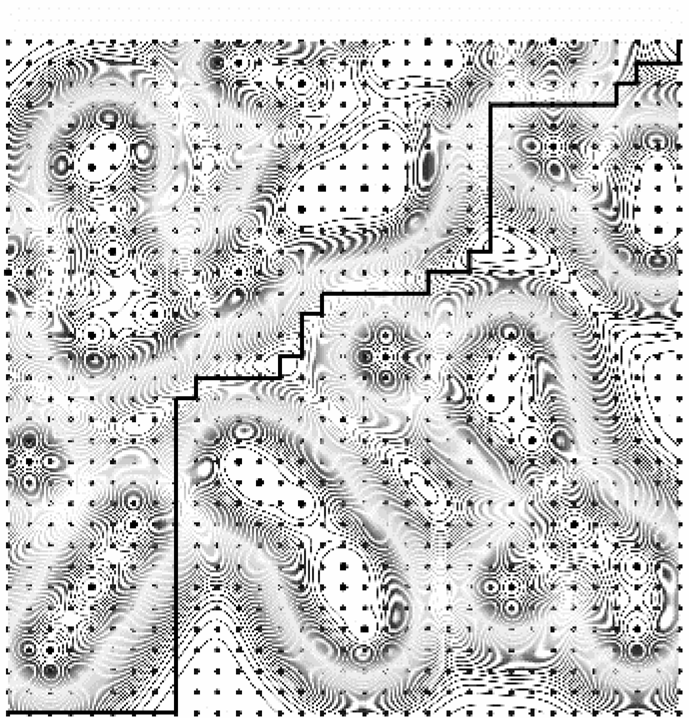


Figure 6. A snapshot of the stochastic search mines path planner.