Motion Planning for Mobile Agents[1]

Hayward Little
Senior, Department of Computer Science and NASA ACE
North Carolina Agricultural and Technical State University
Greensboro, NC 27401
hayward@ncat.edu
Dr. Albert Esterline, advisor

Abstract

A mobile agent is an autonomous entity that travels from a starting point (S) to a destination point (G) without human intervention or collision. Our agents must operate in a static environment, where all the obstacles are stationary. The agent generates a visibility graph, which represents all the possible paths from S to G. From this visibility graph, the shortest path is determined.

1. Introduction

This paper relates to undergraduate research conducted for the NASA-supported project "Motion Planning in a Society of Intelligent Mobile Agents." Agents in this context can be considered as robots. Mobile agents are capable of traveling collision-free from a start point (*S*) to a destination point (*G*) without the intervention of a human. We cover low-level path planning in two contexts, static environments and dynamic environments. A static environment consists of stationary obstacles. Here determining a shortest, collision-free path from *S* to *G* requires knowledge of all pairs of points visible from each other (without an intervening obstacle), so a visibility graph is created. Dynamic environments, however, pose a more difficult challenge since we must deal with obstacles that are in motion. Since the obstacles are in motion, the beginning path may no longer be collision free after a certain period of time. This problem is addressed by using an accessibility graph to determine whether the current path is still valid or must be changed. This paper concentrates on techniques we have implemented for finding the shortest path in the static case. These same techniques, with some modification and considerable enhancement, also supply a foundation for the dynamic case.

2. Abstractions

Our abstractions (classes) include the following: *Point, Line, Polygon,* (graph) *Node, Binary Search Tree, Priority Queue,* and *Undirected Graph*. Both *Binary Search Tree* and *Priority Queue* are used for searching in the line intersection function. For the shortest path, an adjacency structure is created to implement the *Undirected Graph*. A *Point* holds *x* and *y* coordinates, along with a name field. A *Line* holds the index values of

[1] This research was supported by grant NAG 2-1150, "Motion Planning in a Society of Mobile Intelligent Agents," from NASA Ames Research Center.

two *Points* and a name field. A *Polygon* contains the number of edges, a linked-list of the *Points* in counterclockwise order, and a name field. Our *Binary Search Tree* refers to the index of a *Point* and its *x* coordinate. The *Priority Queue* contains elements, each with an index field and *y* coordinate field. And finally, our *Undirected Graph* stores a set of *Nodes*; its edges connect pairs of *Nodes*.

3. Obstacles

Obstacles are assumed to be polygons. Our approach assumes that all the polygons are convex, and this must be verified at input. A polygon is simple if there is no pair of nonconsecutive edges sharing a point. A convex polygon is a simple polygon with a convex set, that is to say, any two points in the polygon can be connected without going outside the polygon [PS85]. This validation is achieved by testing each polygon's convexity. The convex hull algorithm implemented is Graham's scan. (Ironically, Graham is the name of the Computer Science Department building at A&T.) It works in three phases:

1. Find an extreme point. This point will be the pivot and is guaranteed to be on the hull. It is chosen to be the point with largest *y* coordinate.
2. Sort the points in order of increasing angle about the pivot. We end up with a star-shaped polygon (one in which one special point, in this case the pivot, can "see" the whole polygon).
3. Build the hull, by marching around the star-shaped polygon, adding edges when we make a left turn, and backtracking when we make a right turn.

If all the points of the polygon are on the convex hull, then the test is passed (see Figure 1a). Refer to Figure 1b for examples of nonconvex polygons. The last obstacle test is to ensure that no polygons overlap. A search is done to ensure that the endpoints of polygons do not fall into the range of the other polygons (see Figure 1b).

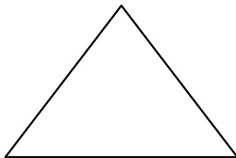

Figure 1a. Convex polygons.

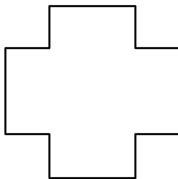
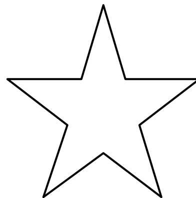
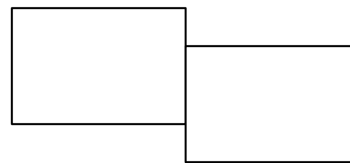

Figure 1b. Nonconvex polygons and overlapping polygons.

A data file is used to store the points, which serve as the start, destination and obstacles. These points are stored in separate arrays, one for the individual lines and the other for corresponding points for each line. This is advantageous because the index can be used to convert from point-to-line and vice versa.

4. Visibility Graph

The visibility graph (see Figure 2) is created by:

1. Reading in S, G, and the obstacles.
2. Generating all paths possible from S to G.
3. Removing those paths obstructed by obstacles.
4. Determining the shortest path (see Note).

Note: Once the *Line* array only contains those lines that are visible from one another, we can determine the shortest path from *S* to *G*. An adjacency structure contains *Points* that are used to represent the visibility graph. We implemented Dijkstra's algorithm to find the shortest path.

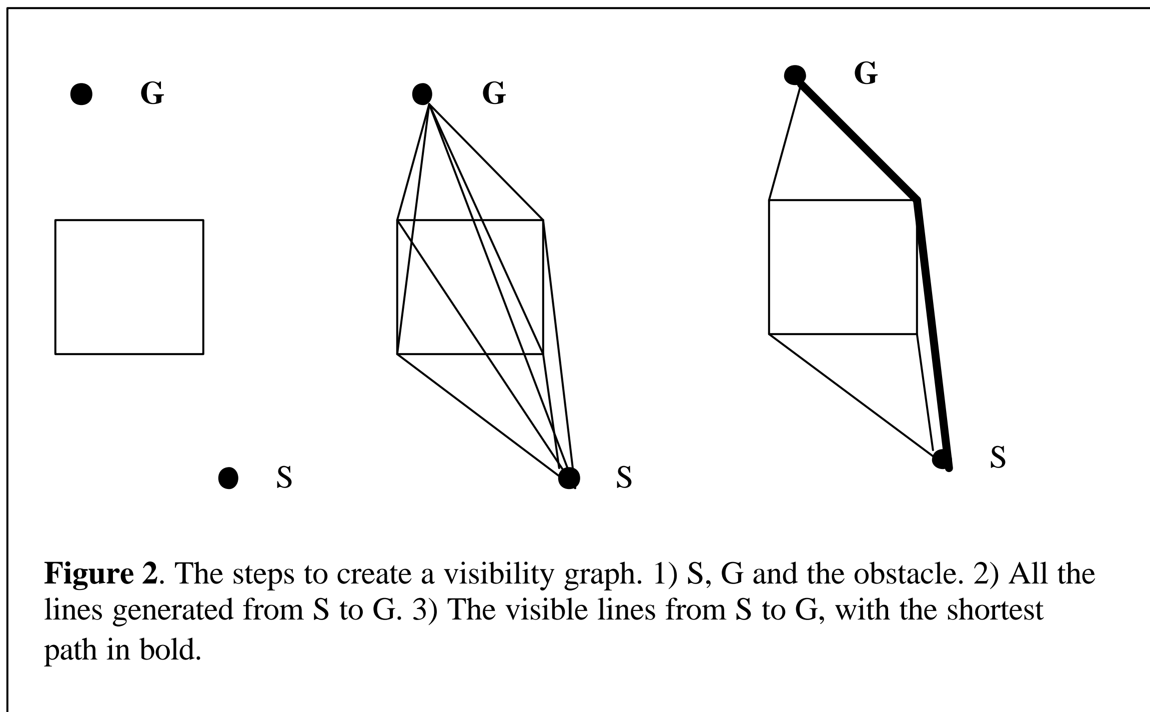

Figure 2. The steps to create a visibility graph. 1) S, G and the obstacle. 2) All the lines generated from S to G. 3) The visible lines from S to G, with the shortest path in bold.

5. Line Intersection

To create the visibility graph, all of the points need to be connected together. This is done with a brute force method that starts with *S* and connects it with each point; we continue, connecting each pair of points (See Figure 3). Given the number of points, *N*, connecting the graph is done in $O(N^2)$ time. Special care must be taken to ensure that no lines lie within polygons.

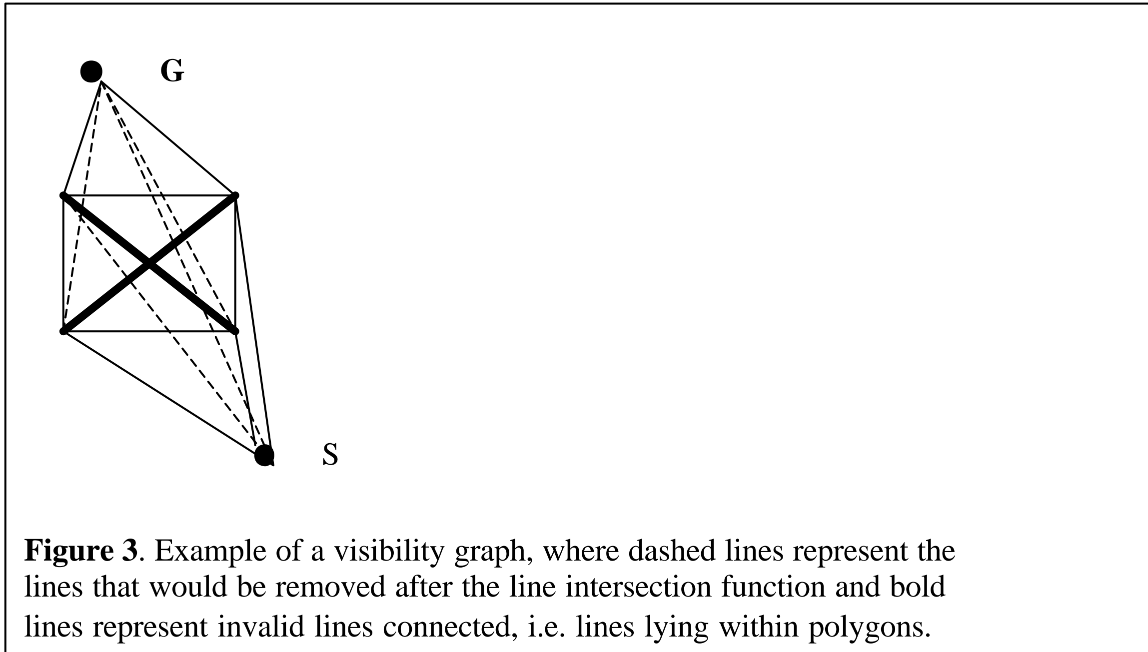

Figure 3. Example of a visibility graph, where dashed lines represent the lines that would be removed after the line intersection function and bold lines represent invalid lines connected, i.e. lines lying within polygons.

Once we have an initial graph, we must remove any lines that intersect with a polygon. We accomplish this by scanning through the graph in a bottom-to-top method. In order to simulate scanning in lines, we use a priority queue, binary search tree and linked-list. The binary search tree is used to insert/delete the points of the lines in a bottom-to-top fashion, while the linked-list stores intersections found within the tree. By using a binary search tree, as opposed to nested for loops, our search algorithm runs in $O(N \log_2 N)$ versus $O(N^2)$ time. To ensure that the lowest point is used for insertion/deletion, a priority queue is used.

Inserting all the ordinates of the lines generates the priority queue (which is ordered in nondecreasing order of the ordinates). By removing all the points from the priority queue and inserting/deleting from our binary search tree accordingly, all intersections are found. Insertion into the binary search tree is done when the lowest point of a line is inserted. Once the line's other endpoint is found, it is deleted from the tree (see Algorithm 1).

Algorithm 1. Pseudo code to simulate a "sweep" of the graph. This creates the priority queue and inserts/deletes from the binary search tree.

```
  procedure insert()
begin j : = 1 ;
      while j ≤ N  do
       begin y1 : = lines[ j ].p1.y ;          y2 : = lines[ j ].p2.y ;
```

```
            pqinsert( j , y1 ) ;
            if y1 ≠ y2  then pqinsert( j , y2 ) ;
            j : = j + 1 ;
        end
end
 procedure scan()
begin
       while pqsize ≠ 0  do
        begin j : = pqremove() ;
            x1 : = lines[ j.i ].p1.x ;          y1 : = lines[ j.i ].p1.y ;
            x2 : = lines[ j.i ].p2.x ;          y2 : = lines[ j.i ].p2.y ;
            if j.c = y1  then bstinsert( j.i , x1 ) ;
            if j.c = y2  then bstdelete( j.i , x2 )
        end
end
```

Since our binary search tree is based on line orientation, we use a counterclockwise function to determine if a line should be in the left subtree or right subtree of another line in the tree. While this function can determine which subtree of another line a given line belongs to, it also detects if the two lines intersect. Intersections are found by checking the slopes of two lines. The slopes tell whether line 1 is the left of line 2 or if line 1 is to the right of line 2. Where line 1 is neither to the left or right of line 2, the two lines must intersect. We treat intersections as a special case and automatically insert the line at the right node in the binary search tree. When an intersection is found, the points and indices are stored in a linked-list. Special consideration must be used when deleting a line. Necessary precautions and conditions must be met to verify the tree is properly shaped after a line is removed. This is because, once a line is removed, it is not guaranteed that its children can be linked to the parent node of the line deleted, since the generalized comparison operation is not transitive (see Figure 4) [Se92]. For these cases, the children are reinserted into the tree.

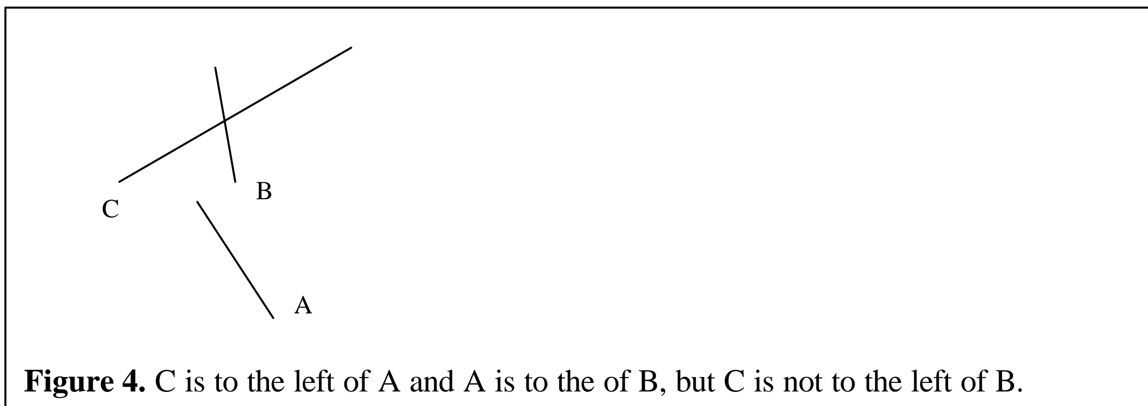

Figure 4. C is to the left of A and A is to the of B, but C is not to the left of B.

6. Accessibility Graph

An accessibility graph is a generalized visibility graph. If all the obstacles in a dynamic environment had a zero velocity, then the accessibility graph would reduce to a visibility graph. Basic assumptions are that our agent is a point and moves faster than all the obstacles. These assumptions are necessary to create time-minimal motion, which can be represented as sequence of edges in the accessibility graph [Fu95]. A path in a dynamic environment is known as a *trajectory*, which refers to velocity and direction along a path. It is essential that our agent is capable of developing *destination points* within a specific time frame. A *destination point* is a goal point in motion. These destination points become various steps necessary to reach our final goal.

7. Conclusion

By developing the necessary classes to generate a visibility graph, it is possible for agents to travel from S to G in a stationary environment. The classes are *Point, Line, Polygon,* (graph) *Node, Binary Search Tree, Priority Queue,* and *Undirected Graph*. Basic assumptions are that obstacles are convex polygons and do not overlap or intersect. Our shortest path algorithm considers those paths unobstructed. Our software was implemented in Java. By taking an object-oriented approach, we have code that can be efficiently reused and utilized in future projects.

In the future, we plan on developing the necessary modifications for our agent to maneuver in dynamic environments. This requires additional functions (methods) and abstractions (classes) that will work with obstacles in motion.

References

[Fu91] Fujimura, K., *Motion Planning in Dynamic Environments.* Tokyo: Springer-Verlag, 1991.

[PS85] Preparata, F. and Shamos, M., *Computational Geometry: An Introduction.* New York: Springer-Verlag, 1985.

[Se92] Sedgewick, Robert, *Algorithms in C*. Reading, MA: Addison-Wesley Publishing Company, 1992.