A RECURSIVE APPROACH TO ROADMAP-BASED PATH PLANNING

David W. Dougall
Electrical and Computer Engineering
Brigham Young University
Provo, Utah, USA
ddougall@ee.byu.edu

James K. Archibald
Electrical and Computer Engineering
Brigham Young University
Provo, Utah, USA
jka@ee.byu.edu

ABSTRACT

Many techniques have been developed to plan paths for mobile robots. The need to provide reasonable paths for mobile robots is critical in the development of useful robots. Many researchers have dismissed roadmap approaches to path planning because of computational inefficiencies and scalability challenges; it is assumed that crude paths result if the problem must be solved in reasonable time. This paper proposes a new approach to path planning that uses recursion to overcome the limitations of other roadmap methods. The JARB algorithm is highly optimized for speed in sparse environments while also providing smooth paths in densely populated environments. By joining recursive branching with the ability to ignore irrelevant obstacles, high quality paths can be produced with reduced computation. We describe the algorithm in detail and present experimental results.

KEY WORDS
Path planning, mobile robots, obstacle avoidance.

1 Introduction

One of the major challenges in mobile robotics is determining an appropriate and efficient path by which a robot can reach its desired goal. The operating environment may range from a simple two-dimensional world with static obstacles to a complex three-dimensional dynamic world with moving obstacles. Because of environment and vehicle complexity, a variety of approaches have been developed over the years to solve the path planning problem. In this paper we propose a new roadmap-based approach suitable for two dimensional environments with sparse obstacles.

Most roadmap based planners such as that discussed by Ibarra-Zannatha and colleagues [1] execute in an iterative fashion, enumerating all possible paths from each edge of an obstacle to all other edges and the destination. From this entire set, a final list of feasible or desirable paths is determined. This method is inefficient in two notable ways. First, the algorithm does not produce a complete path until all possible links have been enumerated. Secondly, the final step of optimization to determine the best path must deal with a significant number of duplicate or near-duplicate links.

JARB, the algorithm we present in this paper, is based on a recursive method for path planning. Intended for application in the relatively sparse world of low-altitude UAVs (uninhabited aerial vehicles), the JARB algorithm begins by charting the straight-line path from start to goal. If the straight-line path intersects an obstacle, the algorithm adds new edges around the obstacle. In our initial implementation, obstacles are assumed to be circular. Circular obstacles provide advantages of computational efficiency while providing sufficient spatial accuracy as discussed by Martinez-Salvador and Pobil [2].

By solving the problem recursively, the algorithm produces a path that can be simplified or augmented even after the original solution is complete without solving the entire problem again. Additionally, because of the roadmap approach, there may be multiple possible solutions from which an optimization may be performed.

The remainder of this paper is organized as follows. Section 2 discusses previous work relating to path planning. The path planning algorithm is presented in Section 3. Section 4 details the major advantages of JARB, while Section 5 discusses boundary conditions or special cases that must be addressed in applying JARB. Section 6 summarizes results of experimental simulations evaluating the path planning algorithm, and Section 7 presents conclusions and directions for future work.

2 Previous Work

Path planning has been studied from many perspectives and a variety of different approaches have been suggested. Latombe [3] provides a thorough overview of many of the approaches to path planning such as roadmap, potential field, and cell decomposition.

The roadmap path planning method involves tracing distinct links throughout the configuration space that become the potential paths. These links are defined differently based on the specific implementation, but they may link obstacle edges, follow a Voronoi Diagram, or use a more sophisticated method such as the silhouette method discussed by Canny [4].

Many roadmap planners create excessive numbers of links. In general, it is difficult to extend the approach beyond two dimensions because of the exponential increase in the number of links. The silhouette method was critical in this regard because it reduces three dimensional roadmap searches to a two dimensional problem.

More recently, randomized and probability based planners have received much research attention. Both PRM (probabilistic road maps) [5] and RRT (rapidly-exploring random trees) [6] algorithms plan paths using randomization techniques. This can lead more quickly to a solution, but the solution is not necessarily optimal nor can solutions be generated in arbitrary environments. It can be argued that path planning is an NP hard problem that cannot be solved in a scalable fashion without randomization techniques, but there are many heuristics that can be applied to significantly reduce the complexity and result in more practical solutions.

The novelty of the algorithm presented in this paper is primarily in its recursive structure. Goel et al [7] discuss a path planner which employs recursive searching, but this is a minor part of their overall algorithm. With JARB, the entire procedure is recursive allowing the complexity of the problem to be adjusted depending on the exactness of the solution desired. A solution can be detailed and precise as the recursion depth increases.

3 Algorithm Analysis

Given full knowledge of the location of obstacles including initial and goal locations, the procedure commences in a recursive fashion. First, a line is created from the initial position to the goal. The algorithm determines if this line intersects any obstacles. If an obstacle is crossed by the line, JARB finds two points just outside the obstacle on either side that are the closest to the original line. New links are then created from the initial point to both new points. These two lines will be referred to as the *first legs* of their respective sides of the obstacle. Two additional links are then created from both new points to the goal point. These two lines will be referred to as *second legs*. The algorithm recursively checks each of the four legs for intersection with obstacles.

Thus, each line that crosses an obstacle will create two new potential paths and each of those have two legs as defined above. If, during a recursive function call, the link does not cross or intersect any obstacles, an obstacle-free link has been found and the function call simply returns.

A limit to the recursion is essential, since at some point there will be little to gain by continuing down a difficult path. There may also be situations where a solution is not possible, where no obstacle-free path around the right side of an obstacle can be constructed, for example. (Additional cases are discussed in Section 5.) In our experiments, the recursion level was limited to 20 levels. No benefit was observed by allowing the recursion past this point. This number may be adjusted depending on the number of obstacles and the size of the search space. It may be necessary to increase the recursion if the number of obstacles increases significantly.

```
jarb_main(init,goal)
{
   obstacle = FindIntersectingObstacle();
   if (obstacle)
   {
      /* find pt on side 1 of obstacle */
      newvertex = GetClosestVertex(1);
      result1 = jarb_main(init,newvertex);
      result2 = jarb_main(newvertex,goal);
      if ((result1 && result2) == SUCCESS)
      {
         return_lists=result1 + result2;
      }
      /* find pt on side 2 of obstacle */
      newvertex = GetClosestVertex(-1);
      result1 = jarb_main(init,newvertex);
      result2 = jarb_main(newvertex,goal);
      if ((result1 && result2) == SUCCESS)
      {
         return_lists+=result1 + result2;
      }
   }
   else
   {
      /* no obstacles: valid path */
      return (init,goal);
   }
   return return_lists;
}
```

Figure 1. Basic JARB Algorithm

Since each call to the planner completes when an unobstructed path is found, each level of recursion corresponds to a new leg in a potential path. If the algorithm fails to find a free path within the permitted number of recursions, the complementary leg (second if it was first, and vice versa) is discarded.

The current implementation is restricted to using circular obstacles and the method to determine intersections with an obstacle is done with a simple geometric calculation as illustrated in Figure 2. The line segment terminating at points P1 and P2 is being checked to see if it intersects the obstacle centered at point P3. The first step is to locate the point on the line segment that is closest to the center of the obstacle. This is represented as point P4. Using simple line equations, this point can be found as follows. Assume (x_i, y_i) are the coordinates for point Pi.

In two-point form, the equation of the original line is given by

$$y = \frac{y_2 - y_1}{x_2 - x_1}(x - x_1) + y_1. \tag{1}$$

The equation of the line perpendicular to the original and going through point P3 is therefore:

$$y = \frac{x_1 - x_2}{y_2 - y_1}(x - x_3) + y_3. \tag{2}$$

Solving Equation 1 for x and substituting into Equation 2 yields

$$x_4 = \frac{m^2 x_3 - m y_3 + m y_1 + x_1}{1 + m^2} \tag{3}$$

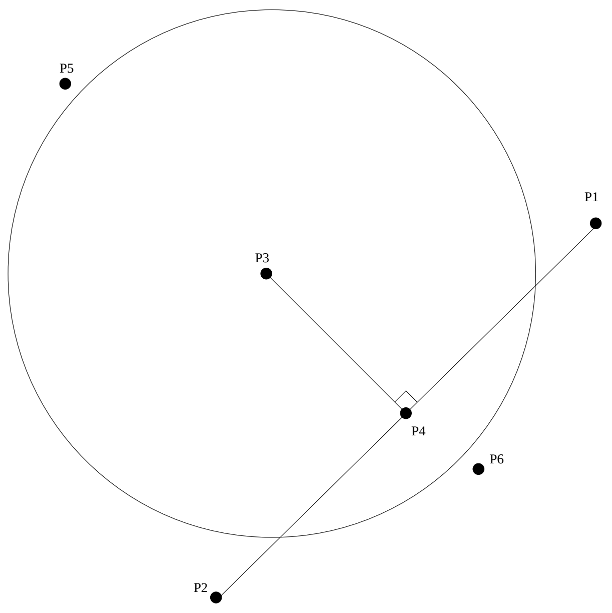

Figure 2. Geometric Layout of Intersection Testing and New Point Determination

where m is the slope of perpendicular line, given by

$$\frac{x_1 - x_2}{y_2 - y_1}. \tag{4}$$

Equation 3 provides the value of the x-coordinate of the point P4 where the two lines intersect. The value of y_4 can be calculated by substituting the value of x_4 into either of the two line equations. Obviously vertical lines must receive special treatment since they have infinite slope.

Once the coordinates for P4 are determined, it is a simple matter to compute the distance from P4 to the center of the obtacle. If the distance is greater than the radius of the circle then the line falls outside of the obstacle and there is no intersection with this obstacle. If the distance is smaller than the obstacle radius, then the original line intersected the obstacle. This function is relatively straightforward to execute, but it must be called for each obstacle as each new edge is tested for obstacle intersection. We note that it is possible to reduce the number of function calls by representing obstacles in a data structure that corresponds to spatial layout. (This allows distant obstacles to be ruled out without first computing the distance to each.) The results presented in this paper do not assume this optimization: every obstacle is tested for intersection with each possible new leg.

The most important aspect of this algorithm is determining what to choose for the ”new points.” Our approach is to simply follow the line orthogonal to the intersecting line and to select points that are on this line and just outside the obstacle. From the center of the obstacle, the points would be radius + Δ. The value of this delta specifies how far from the obstacle edge the new points are located. The smaller the value, the less detour is required around the obstacle. If the value is allowed to be larger, there will be fewer intersections with the same obstacle and hence less recursive steps to perform. Our implementation used a delta value of 1. These new points are depicted as P5 and P6 in Figure 2. These points create something close to the least amount of detour for the original line. Figures 3 and 4 show examples of the point that is chosen to one side of the obstacle. It is the point that is close to the center of the obstacle while lying just outside of its boundary.

4 Strengths of JARB

Because each recursive step creates two new potential paths, this algorithm may create an entire tree of possible paths from initial to goal. An optimizing algorithm such as Dijkstra’s [8] may be executed as a post-processing step to determine the optimal solution from all of these paths. There may be situations, however, where it is desirable to produce multiple possible paths. Our approach can be used to create several possible paths before a task is to be performed. Then, during execution, if a certain path becomes unusable the robot could select an alternative path in real-time without recalculating the entire tree.

Another benefit of JARB is that on large problems that are too intense for a single processor, this algorithm can be easily computed in a parallel fashion on a cluster of processors. Since each new level of recursion creates two or four new threads which are completely independent of any other threads, multiple processors could be computing independent paths at the same time.

The recursive nature of this algorithm allows for a solution to be returned at any level required. If a more general solution is desired, less important obstacles can be abstracted out of the solution to produce a faster result. If a more detailed solution is needed in certain areas at a later point in time, the recursion can continue in that area without requiring the entire search space to be explored again. This later problem is reduced to solving a path between two already determined vertices. Because each new recursive step is essentially an entirely seperate problem to solve, it is also possible to change the obstacle definition with each step. This could be for abstraction reasons as discussed above, or it could be the result of moving obstacles, or it could be because a different set of requirements is needed in certain areas.

Finally, this algorithm is tremendously efficient at sparse search spaces. If obstacles do not intersect each other, the algorithm is even simpler. An issue relating to intersecting obstacles is the order in which obstacles are searched. Given the same obstacles, the JARB algorithm may find slightly different paths depending on the order in which the obstacles are searched to find an intersection. In the current implementation, if a line intersects two obstacles, the resulting ”new points” will depend on which intersecting obstacle is detected first. In practice, we did not find this issue to be a concern because the resulting paths were so similar. We ran several tests to ascertain the impact of searching obstacles in different orders when multiple obstacles intersected a line. Alternatives included selecting the largest obstacle or selecting the closest obstacle. Test results did not provide any conclusive evidence that a particular search order of obstacles consistently produced a better result.

5 Boundary Cases

Certain boundary conditions need to be dealt with to provide a robust solution. The search space is assumed to be bound by some finite size. Paths must not overlap these boundaries. If an obstacle overlaps the boundary or if a new point chosen would be outside of the defined search space, the links that would be terminated on the outside of that obstacle will fail as unpassable. This would return a failure condition to previous recursion levels.

A detail that arises from our specific implementation is dealing with intersecting obstacles. With polygonal obstacles, multiple intersecting obstacles can be dealt with as a single obstacle, however, the current implementation of the algorithm deals only with circular obstacles. As discussed previously, the algorithm chooses a new point that lies outside the obstacle as close to the original line as possible. If this point falls within another obstacle, the obstacles are either intersecting or are close enough to be considered so. This will begin an iterative process to find the next best location for the new point. The process follows the line orthogonal to the original line until it no longer lies within any obstacles. This is represented in Figure 3 by the bold black line inside the smaller obstacle. After detecting the intersection with the larger obstacle, a new point was chosen. This new point, however, fell within the second smaller obstacle. The algorithm moved outward away from the original obstacle's center by small iterations in the same direction until it was free from any more obstacles. The recursion then continues using this point as the new vertex for the new legs.

Lastly, a more complicated boundary condition occurs when the angle from the initial point to its new point is steep enough to cause the new leg to intersect the same obstacle again. If this occurs, we do not want to continue the recursion on both sides of the same obstacle. We only recurse to the side for which this line was initially created. If this check is not performed, it can create a cascading effect of useless recursions producing duplicate paths around and around the obstacle. Figure 4 shows a situation where this would occur. After the new point is chosen and the two new legs are assigned on that side, the leg on the upper side is still intersecting the same obstacle. The desired result is merely to "bend" this leg so that it avoids the obstacle. By executing this single-sided recursion, it has the effect of bending the path so it moves smoothly around obstacle without attempting to create an unnecessary path on the other side that is already being searched by another thread.

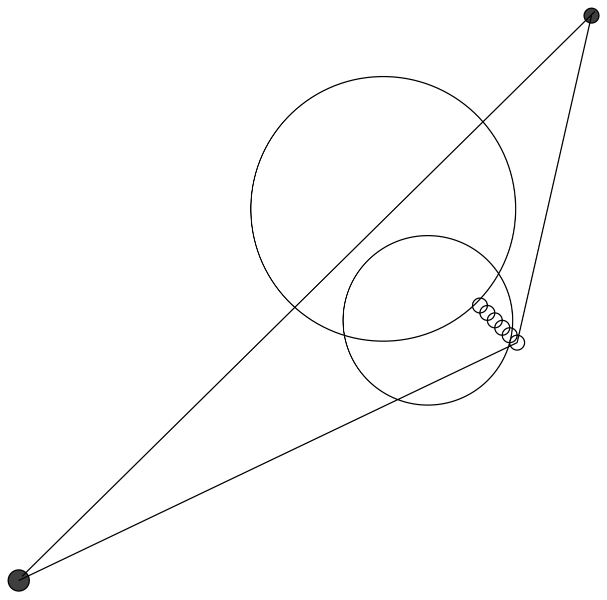
Figure 3. New Point Lies Within Another Obstacle

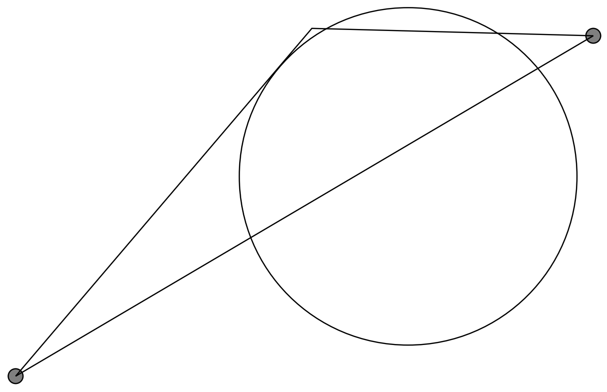
Figure 4. New Line Intersects Same Obstacle

6 Experimental Results

We constructed a suite of 64 separate world scenarios, each with 10 randomly sized and randomly placed obstacles in a 400x400 configuration space. For each run, the initial point and goal were (10,250) and (300,10) respectively. Of the 64 trials, two correctly found that no solution was possible and the remaining 62 correctly found a valid path.

Figure 5 shows a simple yet interesting solution to one of the trials. The process to find this solution included determining that only one side of the first obstacle was passable. All paths to the left of the obstacle (from the reader's perspective) are blocked by additional overlapping obstacles. The algorithm would have either iterated to the edge of the configuration space or recursed too deeply without finding a solution and returned a failure condition for that leg.

Figure 6 shows another path even simpler than the

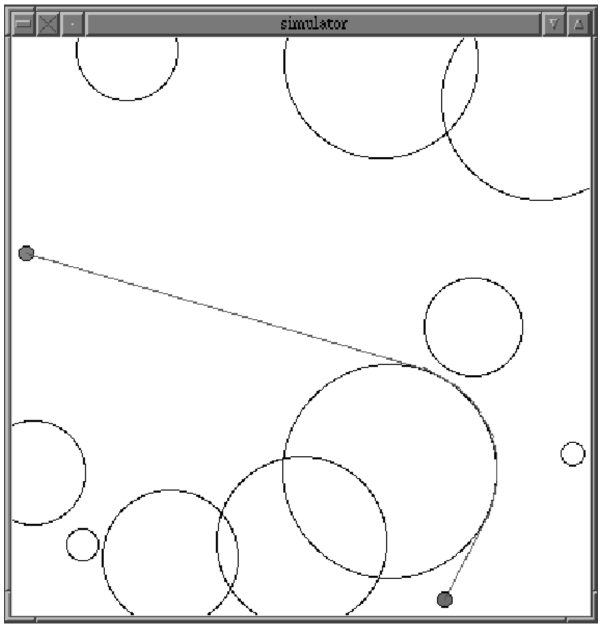

Figure 5. One Sided Solution

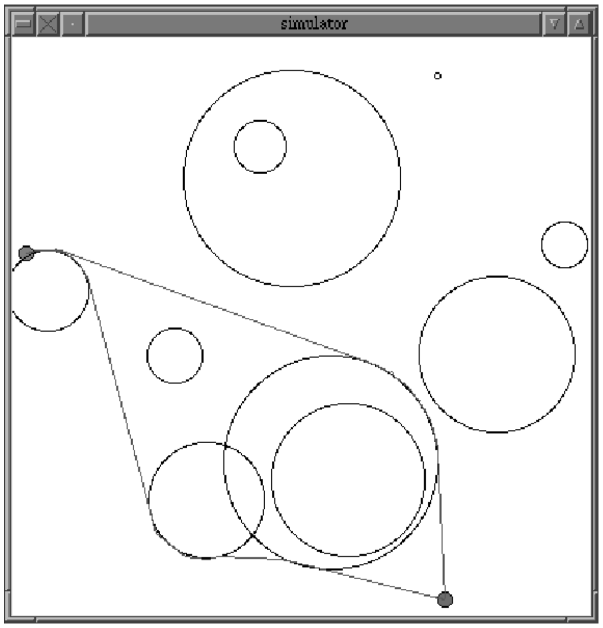

Figure 6. Simple Solution

first. Of note in this figure is the smooth trajectory that is found. In the open areas, the algorithm produces a straight line, but closer to the obstacles it will make a smooth path. Since JARB takes the smallest possible turn to avoid an obstacle, it will have to take many iterations to get around a large obstacle. This may increase the computation time, but creates a path that is more realistic for a nonholonomic robot to follow.

Figure 7 shows one of the more complicated scenarios from the 64 trials. As can be seen, there is only a small passage available for the robot to pass from start to goal. JARB successfully determines this path and provides an optimal path. Unfortunately, there were several stray paths that the algorithm also included. These extra paths were technically correct, but unfortunately completely redundant. This is analogous to creating detours around distant obstacles. Those distant paths are legitimate, but would never be used. In the case of this scenario, all of the extra paths eventually lead back to the small opening and will be ignored by an optimizing algorithm.

7 Conclusions and Future Work

We have presented a new roadmap path planning algorithm that employs recursion to avoid the computational overhead typically associated with other roadmap approaches. Paths generated by this approach typically have a smoother trajectory than previous roadmap planners. We feel that the basic algorithm can be extended in many ways.

By allowing recursion to arbitrary depths, we believe the algorithm will find a valid path in any solveable two-dimensional configuration space. We do not believe, however, that this property would hold for a three-dimensional extension of the basic algorithm. Proving these properties, as well as an implementation in three dimensions, are important areas of focus of our future work.

Circular obstacles reduce the complexity of the problem, but certain environments are better modeled by polygonal representation. We intend to extend the algorithm to allow polygonal obstacles. This increases the required computation significantly—each edge of a polygon requires roughly the same computation as a single circular obstacle.

So far in experiments, this algorithm has only been used to plan paths prior to robot execution. However, if we expand the implementation to allow the addition, removal, or motion of obstacles, we believe this algorithm is well suited for applications with severe real-time constraints.

References

[1] J. M. Ibarra-Zannatha, J. H. Sossa-Azuela, and H. Gonzalez-Hernandez, "A new roadmap approach to automatic path planning for mobile robot navigation," *1994 IEEE International Conference on Systems, Man, and Cybernetics, 1994. 'Humans, Information and Technology'*, vol. 3, pp. 2803–2808, October, 1994.

[2] B. Martinez-Salvador and A. P. del Pobil, "A hierarchy of detail for general object representation," in *Practical Motion Planning in Robotics: Current Approaches and Future Challenges*, K. Gupta and A. P. del Pobil, Eds. John Wiley and Sons, 1998, pp. 225–242.

[3] J.-C. Latombe, *Robot Motion Planning*. Boston: Kluwer Academic Publishers, 1991.

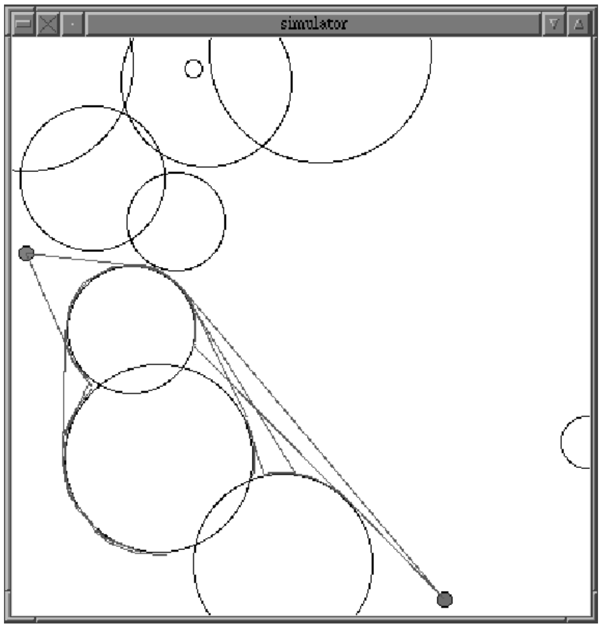

Figure 7. Difficult Solution With Redundant Links

[4] J. F. Canny, *The Complexity of Robot Motion Planning*. Cambridge, MA: The MIT Press, 1998.

[5] L. Kavraki and J. Latombe, “Probabilistic roadmaps for robot path planning,” 1998. [Online]. Available: citeseer.ist.psu.edu/article/kavraki98probabilistic.html

[6] S. LaValle, “Rapidly-exploring random trees: A new tool for path planning,” 1998. [Online]. Available: citeseer.ist.psu.edu/lavalle98rapidlyexploring.html

[7] A. K. Goel, K. S. Ali, M. W. Donnellan, A. G. de Silva Garza, and T. J. Callantino, “Multistrategy adaptive path planning,” *IEEE Expert*, vol. 9, no. 6, pp. 57–65, December, 1994.

[8] E. W. Dijkstra, “A note on two problems in connexion with graphs,” *Numerische Mathematik*, vol. 1, pp. 269–271, 1959.