11-2012

A System for Bidirectional Robotic Pathfinding

Tesca Fitzgerald
Portland State University, tesca@cs.pdx.edu

A System for Bidirectional Robotic Pathfinding

Tesca K. Fitzgerald

Department of Computer Science, Portland State University
PO Box 751 Portland, OR 97207 USA
tesca@cs.pdx.edu

TR 12-02
November 2012

Abstract

The accuracy of an autonomous robot's intended navigation can be impacted by environmental factors affecting the robot's movement. When sophisticated localizing sensors cannot be used, it is important for a pathfinding algorithm to provide opportunities for landmark usage during route execution, while balancing the efficiency of that path. Although current pathfinding algorithms may be applicable, they often disfavor paths that balance accuracy and efficiency needs. I propose a bidirectional pathfinding algorithm to meet the accuracy and efficiency needs of autonomous, navigating robots.

1. Introduction

While a multitude of pathfinding algorithms exist, pathfinding algorithms tailored for the robotics domain are less common. The stochastic behavior of robots in response to environmental factors warrants pathfinding algorithms specific to this domain. While these environmental factors can be counteracted via the use of precise localizing sensors, the growing demand for inexpensive navigational robots requires accurate navigation without expensive sensors. As low-cost navigating robots need to account for potential inaccuracies without the use of sophisticated localizing sensors, an ideal pathfinding algorithm should favor routes that are efficient while also providing opportunities for landmark use. This paper discusses pathfinding needs found in the robotics domain, criteria for an effective pathfinding algorithm that addresses these

needs, how current pathfinding algorithms apply to robotic pathfinding, and a proposed bidirectional pathfinding algorithm tailored specifically for use in robotics.

Section 2 discusses the requirements for a pathfinding algorithm tailored for use in robotics, and why the robotics domain warrants its own pathfinding algorithm. Section 3 provides an overview of related work in pathfinding algorithms and their relation to robotic pathfinding. Section 4 describes the bidirectional algorithm I developed for the robotic pathfinding domain. Section 5 concludes this paper.

2. Background

The following assumptions are made about the robot and its environment:

1. The robot is surrounded by an environment consisting of known pathways and pathway intersections.

2. The robot may only travel along a defined set of pathways.

3. The robot's origin and destination are both defined intersections of multiple pathways.

4. The robot's movements are stochastic in that environmental variables, such as traffic and flooring surfaces, can alter the robot's actual location.

5. The robot maintains a belief state about its location, but cannot ensure the validity of this belief state due to a limited array of sensors.

6. Various aspects of the environment can be sensed and recognized as landmarks unique to a particular location.

To represent the robot's known environment as a graph, each pathway is represented by a graph edge and each pathway intersection represented by a graph vertex. The robot's origin and destination must both be elements of the set of graph vertices. The pathfinding algorithm must return a sequence of vertices that are visited to reach the destination. The pathfinding algorithm discussed in this paper is not designed to include real-time navigation, such as navigation around unexpected obstacles.

As stated above, it is assumed that the robot's movements are stochastic. This is a result of environmental factors, such as variations in the friction between the robot's wheels and multiple flooring surfaces, that can create discrepancies between the robot's belief state and its actual location. Additionally, factors such as the traffic of a pathway can impact both the efficiency and accuracy of the robot's movement, as a pathway with higher traffic can introduce additional environmental forces furthermore adjusting the robot's position while potentially slowing the robot's movement.

It is also assumed that the robot's precise location and heading cannot be known due to a limited array of sensors. Due to the cost of precise localizing sensors, these expensive sensors may not be feasible for use in an inexpensive robot. The remainder of this paper is presented under the assumption that such precise sensors are unavailable for the robot's use due to cost or other constraints.

Without an exact knowledge of its location, it is possible that the robot maintains a belief state of its position that is false due to the effect of environmental variables on its ability to travel. For this reason, the robot's pathfinding algorithm should provide methods of accounting for environmental forces that

would alter the robot's execution of the pre-determined path.

As stated in the earlier assumptions, the robot's environment should contain aspects recognizable as landmarks to a particular location. An example of such a landmark would be colored tiles in an indoor environment that are located in known, fixed locations. Depending on the type of landmark sensed, the robot could use its sensors to align itself to the landmark, thus correcting the location and direction of the robot. This synchronizes the robot's actual location with its belief state.

During route creation, the pathfinding algorithm should be tailored to a robot's likely environment. As the robot and its destination are not likely to be on opposite sides of the graph, the pathfinding algorithm must be able to prune potential paths that are not directed toward the destination. On the other hand, a path that is very accurate but slightly less efficient than the shortest path should not be pruned, even if it travels away from the destination for a short time. The algorithm must be able to prune unfavorable paths while considering accuracy and efficiency, while also not taking too much time to run for large graph input sets.

Overall, the criteria for a pathfinding algorithm suited for robotic navigation are as follows:

1. Must balance accuracy and efficiency

2. Must prune paths that are not directed toward the destination

3. Must be selective in prunes to prevent pruning of accurate, but slightly less efficient paths

3. Existing Algorithms

3.1 Breadth-First Search

An example of a common path-searching algorithm is called breadth-first search. Breadth first search is straightforward to implement, but inefficient during runtime [2]. Additionally, it does not always produce the "best" path. This algorithm searches a graph by considering all possible branched paths in the search frontier, visiting all vertices adjacent to the starting vertex before expanding its search frontier [3]. While this would thoroughly explore paths that balance the robot's accuracy and efficiency needs, it is unsuitable for use in a large environment without pruning heuristics to limit the number of considered paths. Additionally, quite a number of paths will be considered before the frontier approaches the robot's destination.

3.2 Depth-First Search

The depth-first search algorithm traverses the graph using a stack to track visited vertices. Rather than explore each vertex's neighboring vertices sequentially prior to expanding the frontier, the depth-first search algorithm explores one complete path from the origin to the destination before backtracking and exploring a new path [3]. This algorithm is also unsuitable for discovering paths that balance accuracy and efficiency needs, as paths are not directed toward the destination vertex.

3.3 Dijkstra's Algorithm

Dijkstra's algorithm is a greedy search that also determines the lowest path cost required to reach the goal vertex from the starting vertex. To do this, the algorithm determines the lowest cost to reach every vertex from the starting vertex, and then extracts the cost of traveling to the goal vertex [7].

However, Dijkstra’s algorithm’s “best-first" methodology can cause it to skip or postpone

discovering paths that are scored poorly at first, but later contain aspects that increase its overall accuracy and efficiency rating. If a pathway has a worse score than another available pathway, it is ignored in some circumstances, even if the pathway with the higher score leads to much better route options. This reduces the algorithm's efficiency in finding paths that balance accuracy and efficiency needs.

3.4 A* Search

Another search, known as the A* search, uses a distance heuristic to search routes that are closest to the ending point. The point value of each edge is the sum of G and H, where G is the cost of traversing that edge and H is the estimated cost of traveling from the neighboring vertex to the destination [1]. Dijkstra's algorithm can be thought of as a special case of A* that does not use any heuristic for H [2]. At the starting point, the algorithm considers the surrounding nodes and their point values using the aforementioned G and H heuristics. It then calculates the point value of each surrounding node and uses the node with the lowest point value as its new starting point. When this is run recursively until the path reaches the ending point, the resulting path is very efficient.

A* is a more attractive algorithm for the robotics domain than Dijkstra's algorithm, as paths that are directed toward the destination are explored with higher priority. However, this algorithm has the same setback as Dijkstra's algorithm in that it may avoid paths that are scored poorly in the edges closer to the robot's origin, but later contains edges that are scored very highly in their balance of accuracy and efficiency.

4. Algorithm Definition

My algorithm is based on the concept of a bidirectional search, meaning that two search processes run incrementally. One such search process is run using the robot's origin as its first vertex. The other search process uses the robot's destination as its originating vertex. Each search process in this bidirectional search is a breadth-first search (see section 3.2). Once a path from the robot's origin intersects a path originating from the destination, the two paths are combined to form a complete path. Any path that is not yet a complete path will be referred to as a partial path.

The score assigned to each graph edge is used to determine the accuracy and efficiency that it contributes to or detracts from each path that includes it. This is similar to a graph edge's weight that is used in other search algorithms. Furthermore, the landmark availability and traffic level of a path will be represented by its accuracy rating. As explained in section 2, the pathfinding algorithm must have the ability to balance the accuracy and efficiency (total distance) of a path. The scoring function is not defined in this algorithm, as it should be implemented to suit the situation.

With a bidirectional search being used, this algorithm does not avoid paths that appear less promising closer to the origin vertex, as is potential for other pathfinding algorithms. To prevent paths from branching away from the origin or destination vertices, any partial path with a score worse than that of any complete path found thus far is pruned.

There are five elements involved in this bidirectional search: the two search frontiers; the two path lists, which are represented by a growing list of linked lists; and a reference to the best scored path. The two frontiers are used to determine which paths are to be

expanded, and each contain a reference to a path consisting of its ID and a list of vertices. A single frontier can contain multiple listings for the same node so long as they are associated with separate path IDs. The path lists store each paths' IDs with a list of the vertices that comprise each. Finally, the "best path" reference represents the complete path with the lowest, and therefore best, cost balancing accuracy and efficiency.

The general workflow is as follows:

1. Two path frontiers are created: one from the starting point, and one from the destination.

2. Per iteration, expand each path in the frontiers with their neighboring nodes.

3. When the frontiers intersect, a complete path is formed and a score is calculated representing the accuracy/efficiency tradeoff of that path. The complete path is removed from both frontiers.

4. Score is compared to best score so far, with the better scoring path being kept as the best path.

5. If at any time, a partial (incomplete) path in either frontier has a worst score than the "best path", that path is removed from the frontier.

6. Route building continues until there are no more starting or ending paths to evaluate.

The following Figure 1 contains a pseudocode definition for this pathfinding algorithm. The described algorithm has a worst case time complexity of $O(|V| + |E|)$ and space complexity of $\Theta(|V|^2)$, where V is the set of vertices and E is the set of edges in the graph.

```
Initialize(Vertex V0, Vertex Vdest):
    F0 ← new Frontier, F1 ← new Frontier          /* Create the two frontiers */
    P0 ← new PathList, P1 ← new PathList
    p0 ← new Path, p1 ← new Path
    p0.append(V0), p1.append(Vdest)
    P0.add(p0), P1.add(p1)
    F0.add(p0), F1.add(p1)                        /* Add initial paths to the frontiers */
    b ← empty Path                                /* 'b' will contain best-scored path */
    b.score ← ∞


Search(Vertex V0, Vertex Vdest):
    Initialize(V0, Vdest)
    while F0 or F1 are not empty:
        Increment(F0, P0)
        Increment(F1, P1)
        i ← getIntersections(P0, P1):             /* Gets any completed paths */
        for each Path p in i:
            if p.getScore() ≦ b.getScore():
                b ← p                             /* Replace 'b' if 'p' is better scored */
            F0.remove(p)                          /* Remove 'p' from both frontiers */
            F1.remove(p)


Increment(Frontier F, PathList P):
    for each Path p in F:
        if p.getScore() ≦ b.getScore():
            for each Vertex v adjacent to p.lastVertex():
                pn = p.append(v)
                P.add(pn)                         /* Create a new path for each branch */
                F.add(pn)
        F.remove(p)
        P.remove(p)
```

Figure 1

The following figures illustrate a graph to be traversed by this algorithm.

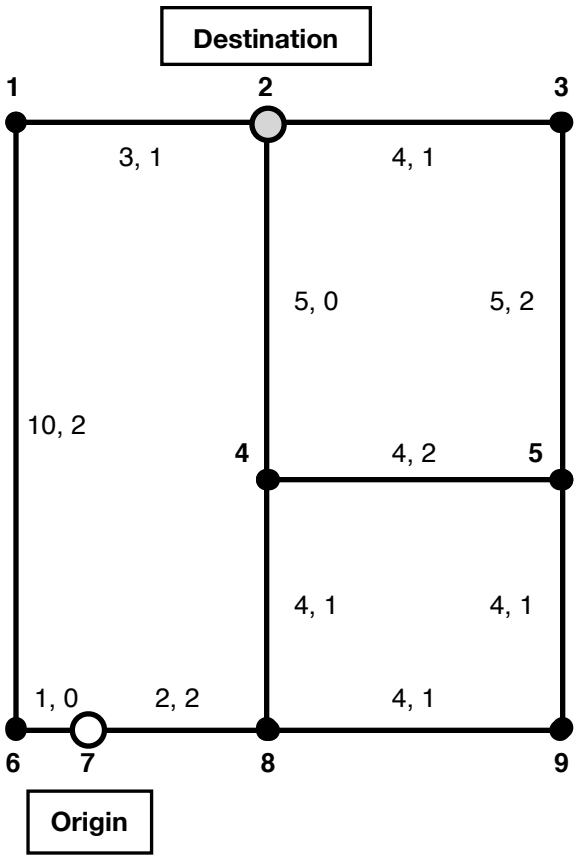

Figure 2

Each edge represents a path traversable by the robot, with vertices being the intersection of two paths. The white vertices represent vertices that have been added to the first frontier. The grey vertices represent vertices that have been added to the second frontier. Visited vertices are represented by thicker black circles. The bold numbers are the reference ID of each vertex. The set of numbers near each edge represent the edge's distance score and accuracy ranking, respectively.

In the example illustrated in Figure 2, the accuracy rating of an edge represents the number of landmarks in that pathway. For this example, the score function is defined as s=d-a, with *d* being the total distance score of a path and *a* being its total accuracy rating. As shown in Figure 2, the algorithm starts with the first frontier containing only the origin vertex (vertex ID 7), with the second frontier containing the destination vertex (vertex ID 2).

The frontier, path list, and "best path" elements corresponding to Figure 2 are as follows.
P_0: [(7)]
P_1: [(2)]
F_0: [(2)]
F_1: [(2)]
b: null

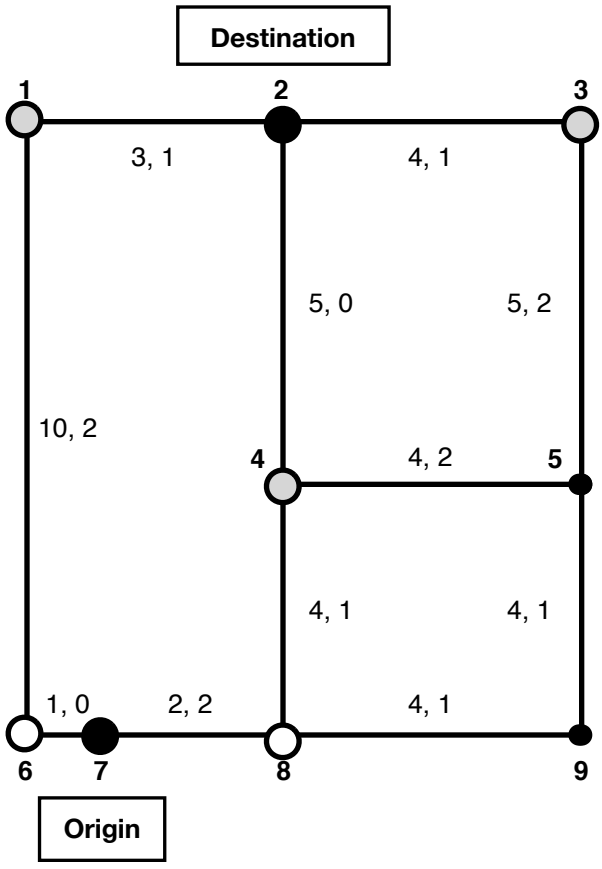

Figure 3

After the first iteration of the algorithm, vertices are added to both frontiers. The result of the second frontier incrementation is illustrated in Figure 3, where the white and

grey vertices represent the vertices newly added to the frontiers.

The frontier, path list, and "best path" elements corresponding to Figure 3 are as follows.
P_0: [(7, 6), (7, 8)]
Scores: [1, 0]
P_1: [(2, 1), (2, 3), (2, 4)]
Scores: [2, 3, 5]
F_0: [(7, 6), (7, 8)]
F_1: [(2, 1), (2, 3), (2, 4)]
b: null

Since no complete path has been discovered yet, *b* is still undefined. Figure 4 illustrates the result of a second frontier incrementation.

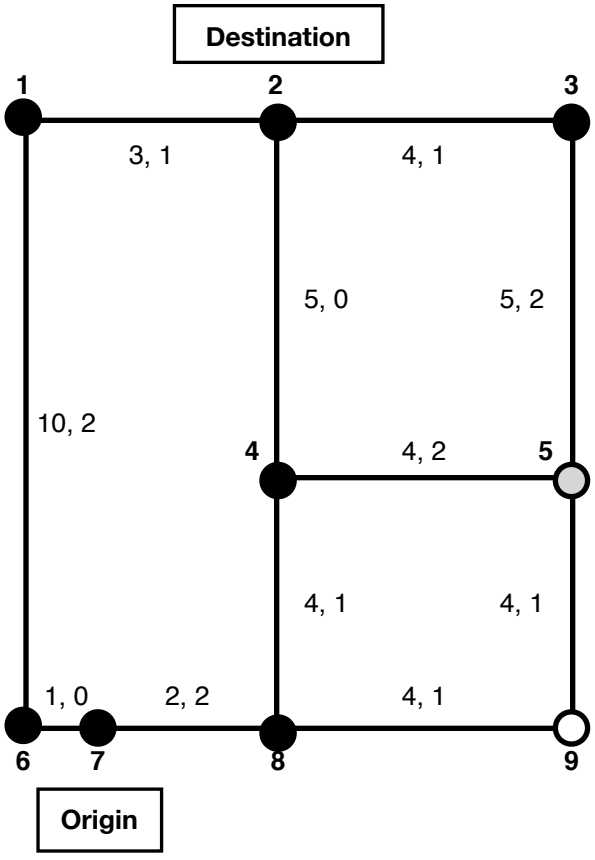

Figure 4

The frontier, path list, and "best path" elements corresponding to Figure 4 are as follows.
P_0: [(7, 6, 1), (7, 8, 4), (7, 8, 9)]
Scores: [9, 3, 3]
P_1: [(2, 1, 6), (2, 3, 5), (2, 4, 5), (2, 4, 8)]
Scores: [10, 6, 7, 8]
F_0: [(7, 8, 9)]
F_1: [(2, 3, 5), (2, 4, 5)]
b: (7, 8, 4, 2)

The path lists P_0 and P_1 intersect over the paths (7, 6, 1, 2) and (7, 8, 4, 2). These two paths correlate to the total scores 11 and 8, respectively. With the path (7, 8, 4, 2) resulting in a lower, and therefore better, score, *b* is assigned to this path.

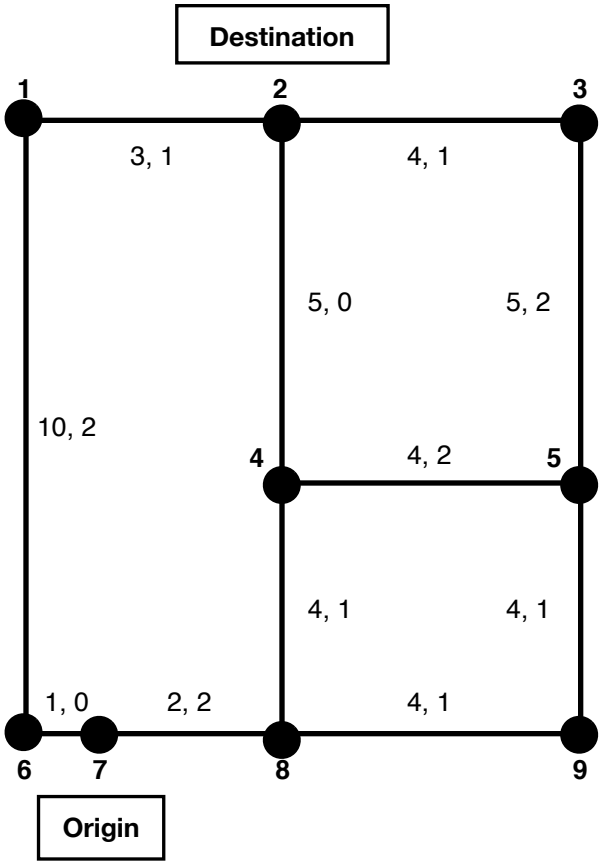

Figure 5

Although two complete paths have been identified in Figure 4, the algorithm does not halt until both frontiers are empty. Figure 5

illustrates the result of a final frontier incrementation.

The frontier, path list, and "best path" elements corresponding to Figure 5 are as follows.
P_0: [(7, 8, 4), (7, 8, 9, 5)]
Scores: [3, 6]
P_1: [(2, 3, 5, 4), (2, 4, 5), (2, 4, 8), (2, 3, 5, 9)]
Scores: [8, 7, 8, 9]
F_0: []
F_1: []
b: (7, 8, 4, 2)

Since the path referenced by *b* correlates to a score of 8, the paths (7, 6, 1) and (2, 1, 6) have been removed from the path lists, as their scores were worse than the score for the best path. Following the final frontier incrementation, the additional complete paths (7, 8, 9, 5, 3, 2), (7, 8, 9, 5, 4, 2), and (7, 8, 4, 5, 3, 2) have been identified. These paths correlate to the total scores 12, 13, and 11, respectively. The previously best path (7, 8, 4, 2) remains the best scored. Since the frontiers F_0 and F_1 are empty, as shown by searched graph in Figure 5, the pathfinding algorithm has finished.

5. Conclusion

For robots with limited location sensors, an algorithm that balances the efficiency and available accuracy features of a path is essential. While current pathfinding algorithms are suitable for other domains, they do not meet the accuracy and efficiency needs of navigating robots. The described bidirectional pathfinding algorithm is suitable for use in autonomous, navigating robots, as it uses bidirectional searching to avoid prioritizing paths more desirable near the robot's originating location (see section 2) and utilizes pruning techniques to limit complexity.

References

[1] Lester, P. (2005). A* Pathfinding For Beginners. Online. *GameDev Website*. *http://www.gamedev.net/reference/articles/article2003*.

[2] Russell, S. J., Norvig, P. (2003). *Artificial Intelligence: A Modern Approach*. Upper Saddle River, NJ: Prentice Hall.

[3] Levitin, A. (2011). *Introduction to the Design and Analysis of Algorithms* (3rd ed., pp. 122-337). Reading, MA: Addison-Wesley.