Computer Vision Guided Autonomous Docking Rover

By Matthew Chang, ECE '19

Introduction

Autonomously navigating a rover from one point to another is not difficult to do. However, with obstacles blocking the straight-line path to the destination, pathfinding becomes necessary. Given a 2D grid that maps the locations of the obstacles, the current position, and the destination, how can a rover find a path to the destination? The most naïve algorithm is to continuously move in a straight line towards the destination until an obstacle is in the way, in which case the rover will attempt to steer clear of the obstacle before proceeding towards the destination. This approach, while computationally inexpensive because it only reacts when the rover's progress is impeded, often does not result in the shortest path. A more convoluted pathfinder, on the other hand, plans ahead and does not necessarily move the rover straight towards the destination. This method requires more computation but promises to always deliver the shortest path to the destination. Balancing the tradeoffs between the two approaches would result in an algorithm that extracts the best features from both algorithms [1].

Uninformed Search

Breadth-First Search

The concept of path planning algorithms originated from graph search algorithms. The idea of graph search is to look through a graph, like the one in Figure 1a, to find a certain node B, starting from some node A. Pathfinding has a similar purpose: to go from point A to point B. Therefore, graph search algorithms can easily be modified for pathfinding. As seen in Figure 1b, the nodes can represent locations (squares) while the connections represent possible moves.

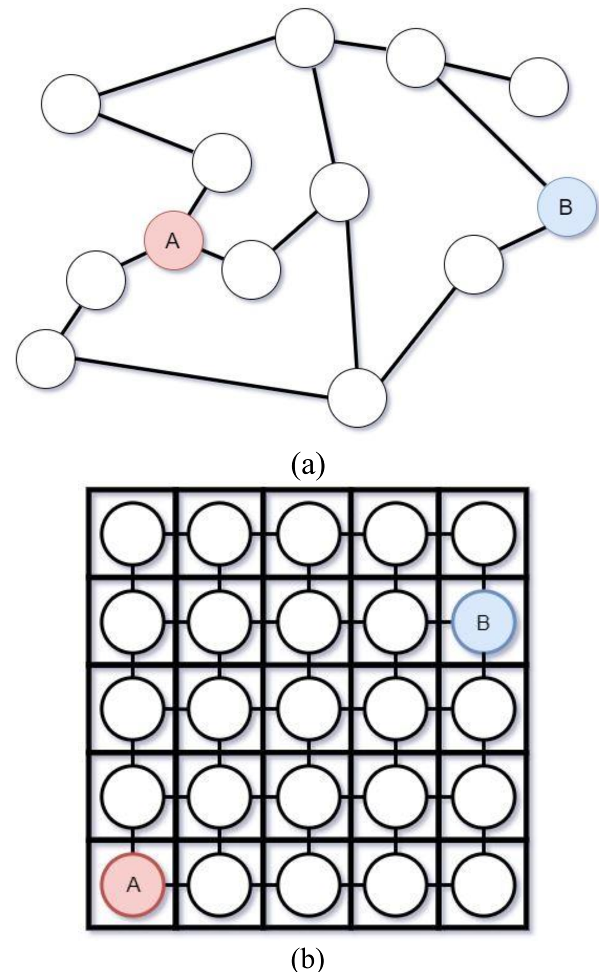

Figure 1. (a) A Graph (b) Graph Representation of a Grid

The simplest graph search algorithm is breadth-first search (BFS). BFS is an algorithm that can be characterized by "an expanding ring called the frontier" [2]. This ring-like search pattern is shown in Figure 2. First, the neighbors of the starting node A (the red nodes) are checked to determine whether any of them is the destination. The neighbors of a node are defined as nodes that have a direct connection to the node in question. Once the red nodes are checked, all their neighbors (the yellow nodes) are checked. Then all their neighbors (the green nodes) are checked. The "frontier" expands by repeatedly checking all neighbors of nodes that have been checked. The frontier will slowly expand until the destination has been checked.

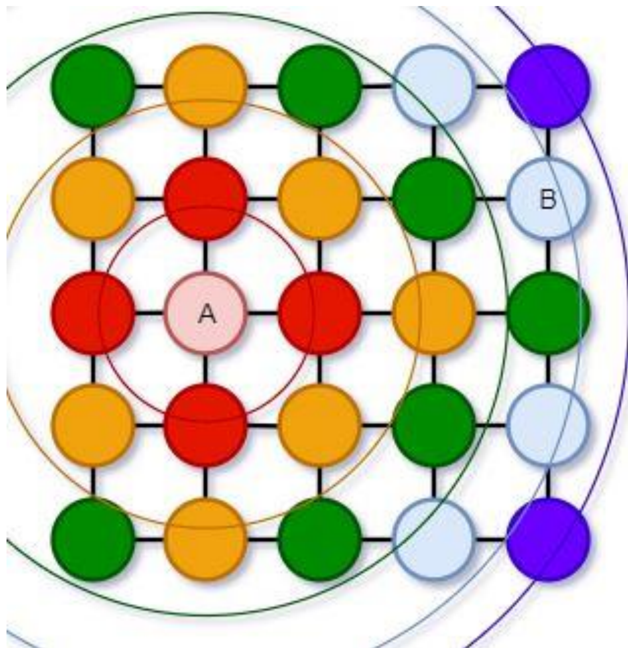

Figure 2. The Ring Search Pattern of BFS

This algorithm belongs to a group of algorithms known as uninformed search algorithms. The defining characteristic of the group is the lack of estimated cost functions. This means that these algorithms do not use information from their current positions to choose a better path [3]. Another flaw for this algorithm is its assumption that all paths are the same, meaning the shortest path is the one that goes through the least number of nodes. In cases where the cost (time or resource) of moving to each position is different, this algorithm will need to be improved. This improvement is known as Dijkstra's Algorithm.

Dijkstra's Algorithm

Dijkstra's Algorithm a pathfinding algorithm that always finds one of the shortest paths to the destination (there could be multiple shortest paths of equal distance). It can, furthermore, find the shortest path to all positions in the grid at the same time. Similar to breadth-first search, the algorithm works by checking the unvisited neighbors of visited nodes. However, it does not check all unvisited neighbors. The algorithm prioritizes certain neighbors, checking nodes with the least path cost from the starting node [1]. The algorithm allows paths to have different costs and keeps track of the smallest cost from the start to each node to determine the next node to be examined. By continually branching out from the closest node, the shortest path from the start to the destination will eventually be found.

The search pattern of the algorithm does not take on the symmetric ring-like shape. The algorithm does not check all neighbors of visited nodes; it prioritizes the nodes with the smallest path cost from the start. Therefore, the "frontier" of checked nodes is not expanded evenly in all directions. In Figure 3, where the paths have different weights (not shown), the frontier grows in different directions.

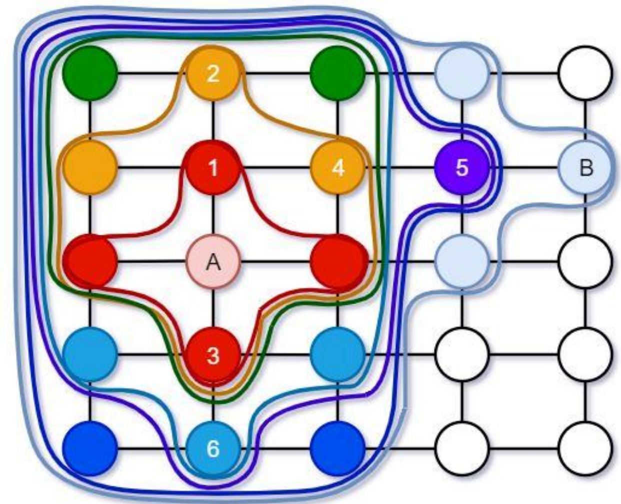

Figure 3. The Uneven Search Pattern of Dijkstra's Algorithm

The biggest flaw with this algorithm is its run-time. Dijkstra's algorithm is even slower than breadth-first search because of the large number of nodes it checks. Such an algorithm would be impractical in

real-time processing. To be able to process video streams, like in our project, a much faster path planning algorithm is needed. One of the fastest path planning algorithms is the greedy best-first-search algorithm.

Informed Search

Greedy Best-First-Search Algorithm

The greedy best-first-search algorithm is the implementation of the naïve algorithm mentioned previously. It tends to select the path that leads the rover directly to the destination without considering obstacles ahead. It is far less computationally intense compared to Dijkstra's Algorithm, because Dijkstra's Algorithm tends to examine more possible steps that lead to the destination. The reason why it is called a "greedy" algorithm is because of its tendency to move toward the destination even when the path is not the shortest [1]. The algorithm estimates the distance between the current position and the destination. Because the algorithm is "greedy," it simply selects the next step that minimizes the estimated distance. This behavior can be observed in Figure 4. It goes straight towards the goal and ignores the cost of the path.

100	100	100	100	100	1	1	1	1	1
100	100	100	100	1	1	100	100	100	B
100	100	100	1	1	100	100	100	100	100
1	1	1	1	100	100	100	100	100	100
1	100	100	100	100	100	100	100	100	100
1	1	100	100	100	100	100	100	100	100
100	1	100	100	100	100	100	100	100	100
1	1	100	100	100	100	100	100	100	100
1	100	100	100	100	100	100	100	100	100
A	100	100	100	100	100	100	100	100	100

Figure 4. Greedy Best-First-Search Path (in red). Optimal path shown in green.

As a result, the algorithm does not always find the shortest path to the destination, even though it requires very little computation relative to other pathfinders. Clearly, an algorithm with a mixture of fast processing speed and shortest path selection is needed. The A* algorithm is a pathfinder designed to take advantage of the best features from both algorithms.

A Algorithm*

The A* algorithm is one of the most popular methods for pathfinding. The algorithm was developed to combine estimation approaches like best-first-search and formal approaches like Dijsktra's algorithm [4]. The resulting algorithm has a computational load and speed comparable to the greedy best-first-search algorithm while still producing one of the shortest paths to the destination like Dijkstra's Algorithm. This performance is achieved by combining the preferences of the two algorithms to produce a cost evaluation function that considers the distance between the current position and the starting point as well as the estimated distance between the current position and the destination. The unvisited neighbor with the smallest cost calculated based on the evaluation function will be selected. The resulting search pattern (Figure 5) is a blend of that of Dijkstra and best-first-search. It has the uneven shape of Dijksta's search pattern but is much more directed towards the destination. The evaluation function allows the algorithm to explore much fewer nodes but still find the best path.

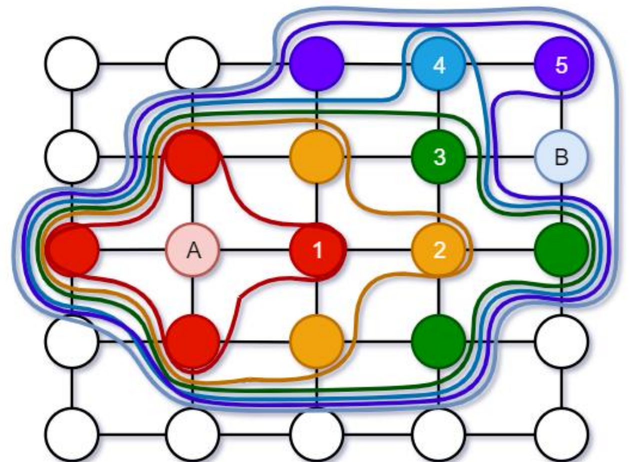

Figure 5. The Directed Search Pattern of A Algorithm*

One critical note about the evaluation function of the A* algorithm is that it must be “admissible”: it must never overestimate the actual path cost. This is critical to help the algorithm rule out other paths when it finds a path whose actual cost is less than the estimated costs of other paths. Because the costs of the other paths are underestimations, the path found can safely be declared as the shortest path [5].

Application

The original plan was to use a variant of the A* algorithm known as D* to navigate the rover to the destination. As the stereo camera mounted on the rover may not have visual of the entire docking area, we needed an algorithm that has the ability to adjust its current path or recompute the whole path. The D* algorithm could produce “an initial plan based on known and assumed information, and then incrementally repairs the plan as new information is discovered about the world” [6]. However, the probability of having moving obstacles on the docking area is very low. Without obstacles, all paths essentially have equal cost, so the shortest path is the straight-line path to the destination. We opted for the best-first-search algorithm for its speed and low computational complexity.

Conclusion

A* algorithm seemed like the best general pathfinding algorithms among the ones explored in this paper. It provides a good balanced of speed and optimality. Yet, the algorithm that best suited our purposes was best-first-search. Because of the lack of obstacle, we can place more emphasis on the speed of the algorithm.

References

1. A. Patel, "Introduction to A*," Stanford University, 24 November 2018. [Online]. Available: http://theory.stanford.edu/~amitp/GameProgramming/AStarComparison.html. [Accessed 14 December 2018].

2. A. Patel, "Introduction to A*," Red Blob Games, 7 November 2018. [Online]. Available: https://www.redblobgames.com/pathfinding/a-star/introduction.html. [Accessed 15 December 2018].

3. H. Choset, "Robotic Motion Planning: A* and D* Search," Carnegie Mellon University, [Online]. Available: https://www.cs.cmu.edu/~motionplanning/lecture/AppH-astar-dstar_howie.pdf. [Accessed 16 December 2018].

4. J. McCullock, "Path-Finding A*," Mnemosyne Studio, [Online]. Available: http://mnemstudio.org/path-finding-a-star.htm. [Accessed 14 December 2018].

5. M. Welling, "A* Search," UC Irvine, [Online]. Available: https://www.ics.uci.edu/~welling/teaching/ICS175winter12/A-starSearch.pdf. [Accessed 16 December 2018].

6. T. Stentz, "Real-Time Replanning in Dynamic and Unknown Environments," Carnegie Mellon University, [Online]. Available: http://www.frc.ri.cmu.edu/~axs/dynamic_plan.html. [Accessed 17 December 2018].

7. S. M. LaValle, "Stentz's Algorithm (D)," University of Illinois at Urbana–Champaign, 20 April 2012. [Online]. Available: http://planning.cs.uiuc.edu/node616.html. [Accessed 21 December 2018].