Preprocessing Methods in Informed Graph Search

John E. Jones
COMP 364: Artificial Intelligence
December 16, 2010

1. Introduction

Informed graph search is a very practical problem in the field of artificial intelligence, and, as discussed in class and based on research here, it is a field of extensive study. Here, we look at the results of using the traditional A* Search, and look at newer research using techniques such as Bidirectional A* Search, and the ALT set of algorithms. Andrew Goldberg and Chris Harrelson in [1, 2] have completed extensive research in the field of ALT algorithms. ALT refers to A*, Landmark, and Triangle inequality – the principles which the technique is based upon. These ALT algorithms show improvement over other informed search methods based upon the technique's use of graph preprocessing.

All code in these experiments was compiled and executed using the Sun/Oracle Java 6 compiler and runtime (64 bit) on a MacBookPro with a 2.2 GHz Intel Core 2 Duo processor and 4 GB of RAM.

2. Background

A* Search, while a significant improvement over other informed search methods such as Greedy best-first search [3] and Dijkstra's algorithm [2], are too costly and inefficient for practical uses such as real-world mapping systems like Yahoo! Maps and Microsoft MapPoint. [2] A* Search is inefficient because, at each tree-node expansion,

the algorithm must calculate the distance-to-goal heuristic and path distance thus-far for each child node.

Goldberg and Harrelson prove in [1, 2] that ALT algorithms reduce the complexity of A* Search using preprocessing. ALT performs preprocessing by selecting some subset of vertices in the graph, called *Landmarks*, determining the Euclidean distance to every other vertex in the graph, and storing that information per each Landmark. Note that the Euclidean distance between two vertices and path distance between two vertices may not be equal. Because of this preprocessing, there is less work to be performed at algorithm runtime based on the amount of data available to the algorithm. A drawback to this method is the need for increased memory as a result of the preprocessing data. [1, 2]

3. Methods

To test various algorithm implementations, we used graphs of 900 random vertices. These vertices were created by defining a 1000 by 1000 plane, picking 900 unique x coordinates, and then picking any 900 y coordinates. The y coordinates did not need to be unique because the x coordinates ensured uniqueness of the vertices' locations. To define the graph edges, for each vertex, an undirected edge was added to any other vertex within y +/- 100 and within x +/- 100. Then added to the graph are two vertices at (200,200) and (800,800), and edges are added to these nodes using the method described above.

Each algorithm is then instructed to find a path linking to the two nodes defined above on the graph. Results recorded were path size in nodes and pixels, nodes added to

the search tree, and runtime in nanoseconds. A visual result of the graph and algorithm executed was also produced in the form of a PNG image.

An additional facet of the testing were the algorithms' performances on real graph data. The researchers in [2] had the added benefit of access to the Microsoft MapPoint database; here we do not. Therefore real-world data was compiled using pathways and intersections at Dickinson College.

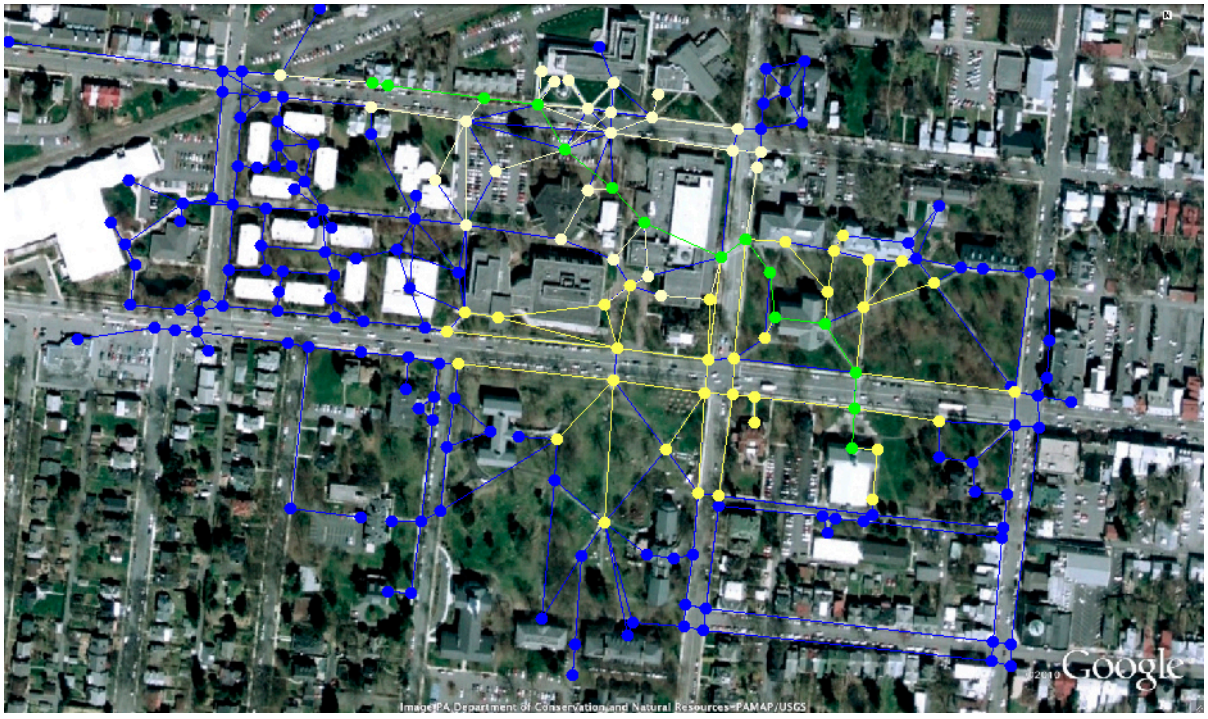


Figure 1. This aerial map of the Dickinson College campus provided a guide to the creation of the pathways and intersections graph. The image is courtesy of Google Earth. In green is the path found by a Bidirectional Symmetric A* Search. The yellow and white paths represent the search trees.

4. A* Search Methods

4.1 Implementation of A Search*

A* Search was implemented based upon the pseudo-code given in Russel and Norvig. [3] Because in both graphs used in the experiment are location based – the edge weights are based upon the vertex values – our frontier node priority function $f(\bullet) = g(\bullet) + h(\bullet)$, is defined as follows for any vertex v in our graph:

- $g(v)$ – The path-distance from the starting vertex to this vertex. Note that this is not a Euclidean distance
- $h(v)$ – The Euclidean distance to the goal vertex.

4.2 Implementation of Bidirectional A Search*

Based upon research in [2], we implemented a form of Bidirectional A* Search known as Symmetric Bidirectional A* Search. Typical bidirectional search as defined in [2] stops as soon as the first two frontiers intersect. However, this solution may not be optimal and a new, more robust termination requirement is necessary. In Symmetric Bidirectional A* Search, the algorithm will constantly check any new solutions against the current solution to see if the new solution is better. If the new solution is better, that becomes the current solution. The algorithm will terminate when there are no more nodes in either forward or reverse frontiers or the heuristic function $g(\bullet)$ for the node at the front of the queue yields a value greater than that current solution. This termination condition exists because if the greatest lower bound offered by the frontier is greater than a true solution, there cannot exist a better solution than the current one.

5. ALT Search Methods

ALT, as stated earlier, is an acronym for A* Search, Landmarks, and Triangle inequality. The concept behind this method is this: for any graph $G = (V,E)$, we take a subset of that of vertices in V and call those *Landmarks*. Then, we compute the distance from each landmark L to every other vertex in V. This distance to L from some vertex v in V is represented by $d(v)$. Because of triangle inequality, it holds that for any two vertices v and w in V, $d(v) - d(w) \leq dist(v,w)$, where $dist(v,w)$ represents the shortest path distance between v and w. [1,2]

Thus, the distances found in the landmark preprocessing offer us a lower bound on the shortest path between two vertices. The ALT algorithm, similar to A* Search, selects nodes using a priority queue where the result of $d(v) - d(w)$ determines a node's priority. In that case, v is the current vertex in the graph and w is the goal vertex. Also, when using the $d(\bullet)$ function in implementation, our code selects four random landmarks and finds the landmark that gives the highest lower bounds to use in the $d(\bullet)$ function.

5.1 Static Landmarks with Random Selection

In order to use the ALT algorithm, we must select some subset of vertices to serve as landmarks. The simplest, most efficient method is to select some subset of vertices at random. Here, we choose 16 random vertices. [1,2]

5.2 Static Landmark with Farthest Selection

To make the landmark choices more meaningful, we also employed the *farthest* landmark selection method. In this method, an algorithm selects any vertex v and adds

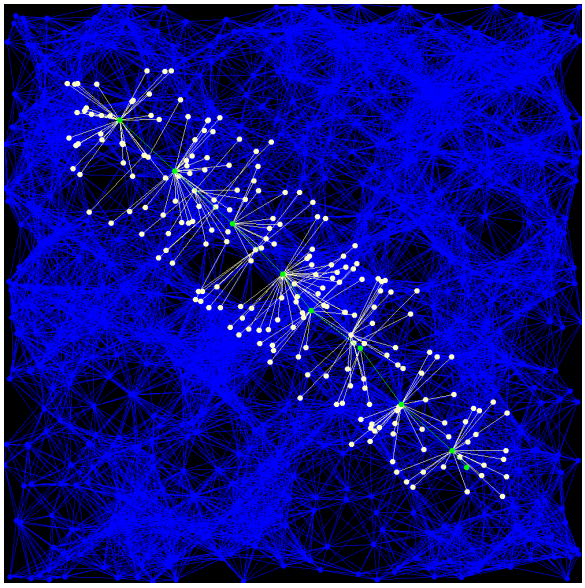
it to the set of landmarks. Then, the algorithm finds the vertex that is farthest-from *v* and adds that new vertex to the set of landmarks. The algorithm then continues to add vertices to the set that are farthest from all of the landmarks until there are the desired amount of landmarks in the set. In farthest, we also choose 16 landmarks. [1,2]

6. Random Graph Results

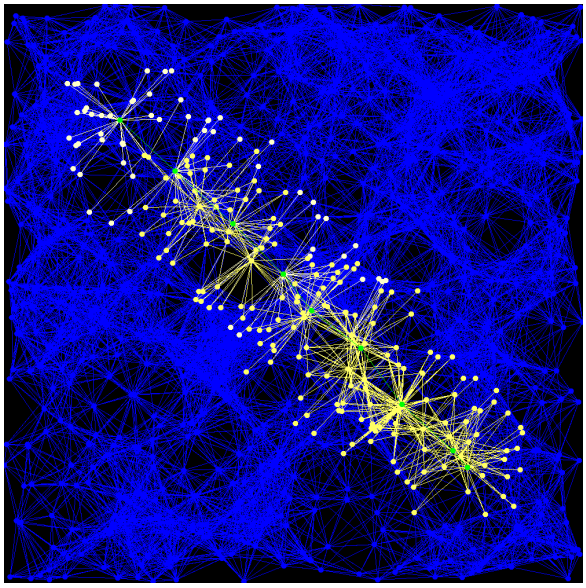
Table 1. The data in this table is based on averages of 30 executions of each algorithm on the same random graph with the same start and target vertices. Note that the runtimes given are in nanoseconds.

Algorithm	Path Nodes	Distance	Tree(s) size	Runtime
A* Search	10	852	262	6501266
ALT (Farthest Landmarks)	12	1092	320	5981633
ALT (Random Landmarks)	10	986	313	2982733
Bidirectional Symmetric A* Search	10	852	275	8368133

A Search Based Results*



A* Search

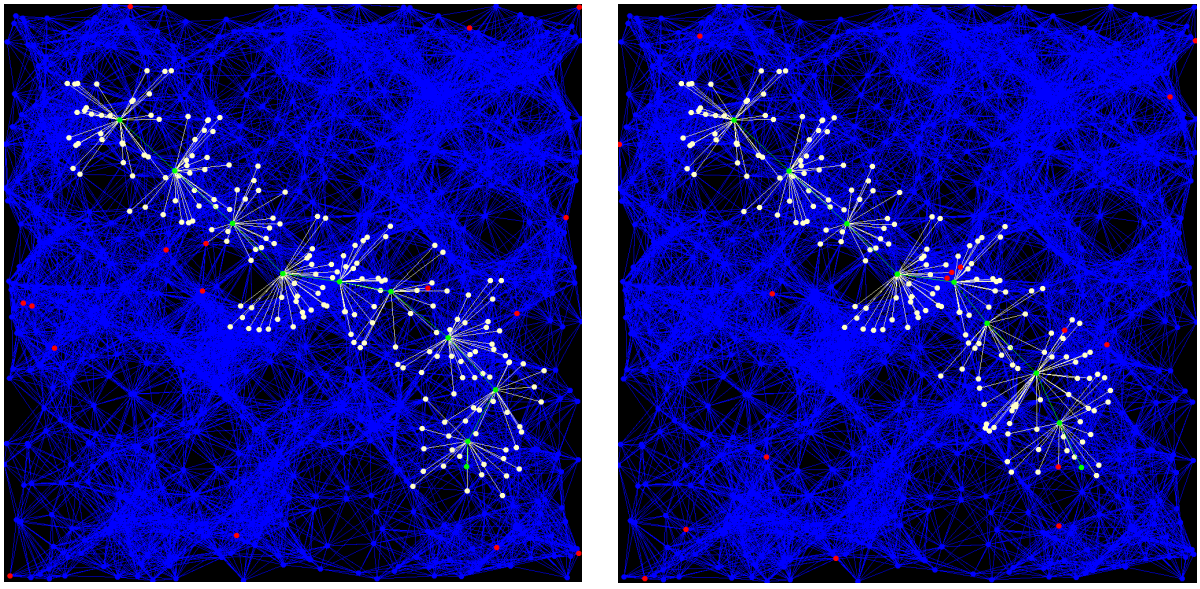


Bidirectional Symmetric A* Search

Figure 2. These renderings represent results of the A* Search and Bidirectional Symmetric A* Search on a graph of 900 vertices.

When executed on the same graph, both A* Search and Bidirectional Symmetric A* Search generally discover the same path with drastic performance differences favoring traditional A* search. According to the data gathered (see Table 1), A* Search, on average, executes in 75% of the time Bidirectional Symmetric A* Search executes. Additionally, in Table 1 and Figure 2, one can see that Bidirectional Symmetric A* Search scans more graph vertices. This is attributed to the fact that Bidirectional Symmetric A* Search cannot terminate upon discovery of the first solution as described in §4.2.

ALT Search Results



Farthest Landmarks **Random Landmarks**

Figure 3. These renderings represent results of the ALT algorithm with landmarks chosen using the *Farthest* method and at random.

The ALT algorithms showed significant time-performance improvement over the A* searches. As Table 1 shows, ALT with *farthest* landmarks has an average execution time of 5981633 ns and ALT with random landmarks has an average execution time of 2982733 ns. ALT with random landmarks showed to be the fastest of all algorithms tested, but also searched the most vertices as seen in Table 1 and Figure 3. This was visible in about half of the trials of that algorithm. In terms of distance, the ALT algorithms found paths that were marginally longer than those found in the A* searches despite their efficiency improvements. However, increased computational efficiency is the goal of ALT search because of the need for fast algorithms on extremely large datasets. [1,2]

7. Real-World Graph Results

Table 2. This data is based upon a single execution of the algorithms on graph data of Dickinson College's campus. Averages were not used because multiple trials in this scenario produced almost uniform results. Also, runtimes are omitted because the classes that interact with the Dickinson graph depend upon a GUI, which interfered with runtimes.

Algorithm	Path Nodes	Distance	Tree(s) size
A* Search	19	1380	113
ALT (Farthest Landmarks)	19	1440	46
ALT (Random Landmarks)	18	1510	48
Bidirectional Symmetric A* Search	17	1348	124

The graph of Dickinson College's campus contained 201 vertices, on a plane of width 1440 and height 852. Our start vertex, the Goodyear building, was located at (6, 46), and our target vertex, the side door of the Weiss center, was located at (1043, 595).

In this scenario, the ALT algorithms scanned less than half the vertices that A* Search and Bidirectional Symmetric A* Search scanned. Also, Bidirectional Symmetric A* Search was able to find the best path out of all four algorithms. We attribute this difference in results to the lesser complexity of the graph.

8. Conclusions

The ALT method of shortest path finding shows significant improvement in computational efficiency despite its need to scan more vertices. As shown in [1,2], this method is very useful for large datasets because of the preprocessing performed. Unfortunately, these large datasets were unavailable in these experiments. Additionally, in [3], the Goldberg and Harrelson also suggest an improved method of dynamic landmark selection as opposed to static landmark searching used here.

Animations of the algorithms studied in this paper are available at:

- http://users.dickinson.edu/~jonesjo/searches/a-star.mov
- http://users.dickinson.edu/~jonesjo/searches/ALTfarthest.mov
- http://users.dickinson.edu/~jonesjo/searches/ALTrandom.mov
- http://users.dickinson.edu/~jonesjo/searches/bidirectional.mov

8. References

1. Goldberg, Andrew V. *Point-to-Point Shortest Path Algorithms with Preprocessing.* Microsoft Research, Microsoft Corporation, Mountain View, CA: Microsoft Research, 2007.
2. Goldberg, Andrew V., and Chris Harrelson. *Computing the Shortest Path: A* Search Meets Graph Theory.* Microsoft Research, Microsoft Corporation, Redmond, WA: Microsoft Research, 2004.
3. Russell, Stuart J., and Peter Norvig. *Artificial Intelligence: A Modern Approach.* Vol. 3. Upper Saddle River, NJ: Pearson Education Inc., 2010.