Hierarchical Graphs for GPS Routing Devices

January 31, 2011

1 Problem Definition

When using GPS devices, it is necessary for them to be able to compute the shortest path between two points very quickly. Improving the ability to answer queries from the user quickly is essential for a good device, but the size of the stored map can be a big hindrance. Therefore methods to reduce the number of visited nodes would be a good way to speed up the entire process. The graph has some properties that can be used to achieve this goal. In particular, cities have a higher density of nodes and arcs, many of which do not need to be considered while searching for a path to pass through. We would like to test the performance of an algorithm that precomputes the shortest path to cross a subgraph, like a city, creating a new and more sparse graph that can be used to quickly find a way to cross cities without visiting the thousands of nodes composing them.

2 Related Work

In [Song and Wang, 2010], Song and Wang experimented with three algorithms to find a shortest path in a hierarchical graph, where subgraphs called "communities" are extracted. Higher levels of the graph are created, with each community represented on the higher level only by their frontier nodes, and new arcs connecting each of them with the precomputed shortest path. The higher level graph is then connected with the lower one with new dummy arcs, in order to keep the connection between different levels. The algorithms have a good performance, from two to three times faster than A^* for the first two, and reaches a speed-up of 10 with the last one, which unfortunately finds an approximated solution.

In [Jagadeesh et al., 2003], instead, a two-level graph is generated, with only the major roads in the upper level. The hierarchical algorithm used, called "node promotion hierarchical algorithm", finds a route between the starting point and nodes connecting the bottom with the top level. Unfortunately, those are not the frontier nodes as in [Song and Wang, 2010], but are just nodes connecting major roads to a specific block of minor roads, resulting in a loss of precision: the generated path in the hierarchical graph is on average 2.8% longer than the optimal solution, reduced to 0.25% with the heuristic presented in the paper.

3 Implementation

Our implementation can be divided into two different processes: the online and offline computations. The offline computation includes the graph simplification and the hierarchical graph generation through community (or cluster) extraction. The online portion is the hierarchical search algorithm to find the optimal path between any two nodes in the generated graph.

3.1 Input Graph simplification

We retrieved our test data from [OSM, 2011, GF, 2011], which is more oriented to visualization than it is to planning. For this reason, we had to transform the graph in order to fit our needs. In particular, we had to remove all of the paths that were not representing roads or streets (such as buildings, rivers, lakes). This was done by computing the connected components of the graph and keeping the largest one, which was the network of roads. Additionally, we removed nodes that were in the middle of roads, but not intersections, which are needed only for visualization purposes and not for planning. For this reason, every node in the graph has at least a degree of three. This left us with a graph containing nodes that only represented possible sources and destinations for someone to travel between.

3.2 Hierarchical Graph generation

The point of the hierarchical graph generation is to find areas that can be simplified. These areas have very dense connections within them, but fewer nodes that connect them to the rest of the map. This is crucial for the performance of the search algorithm, because it impacts both the offline and online computation for the reasons explained in the following sections.

3.2.1 Cluster Identification

We did not implement the methods outlined in the related work, due to time constraints. There are many ways to extract communities from a graph, among them using linear or dynamic programming. However, we chose to use a less sophisticated approach that still resulted in significant improvements from the unedited graph.

Our method was inspired in part by simulated annealing [Russell and Norvig, 2009]. To extract a cluster from the graph, we start from a node that has a high degree of connectivity (65% of the maximum degree of any node in the graph). We mark this node as being in a cluster and expand to its neighbors, adding them into the same cluster if their connectivity also passes a slightly lower threshold. The threshold is decreased by five percent with each step away from the starting node to a minimum of thirty percent. We also test neighbors with low connectivity that are less than two steps away from the cluster in order to allow areas in the cluster that are not highly connected. This was done because not every street in a city intersects the same number of streets.

3.2.2 High-Level Generation

Having computed the clusters in the graph, we then find each cluster's frontier nodes. We run a modified version of Dijkstra's algorithm, which uses multiple targets. For each node on the frontier, we compute the shortest path to every other frontier node. Computationally speaking, this is the same as computing the shortest path from the source node to the furthest of the target nodes. Then the shortest path between the frontier nodes is stored as a new arc in the graph. The nodes in the cluster that are not on the frontier are removed from the high level graph. Note that if the clusters have been correctly identified, the number of new edges added to the graph is small, while the number of nodes removed is consistent. This will considerably reduce the extra space required by the heirarchical graph and increase the performance of the search algorithm.

Graph Nodes	3461
Graph Arcs	9261
Hierarchical Graph Nodes	1867
Hierarchical Graph Arcs	8159
Hierarchical Graph Space	+74.52%
Test Cases	50,000

Table 1: The Testing Set used to evaluate the implemented algorithms.

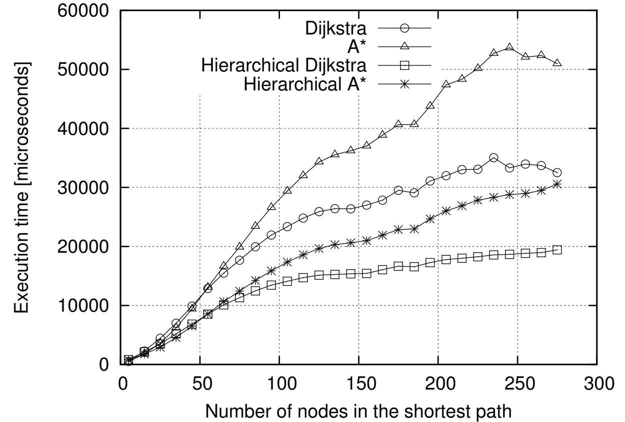

Figure 1: Execution time of the four algorithms on path length.

3.3 Hierarchical Search Algorithm

The search process for our graph focuses on the high-level portion. This way, unless a destination is within a cluster, we are guaranteed the shortest path through any cluster we pass through en route. If the source or the target are within an identified community, another step is required before the search can be completed. The modified Dijkstra algorithm is executed on the offending node, finding the shortest path to every frontier node of its cluster. A temporary node is then added to the high-level graph, connected to each of the frontier nodes by a new arc storing the shortest path. The normal search algorithm can then be run to find the optimal solution. To do this, we chose to implement Dijkstra's algorithm and A^* with the following heuristic (h):

$$a = sin^2(\Delta Lat_{s,t}) + cos(Lat_s) \cdot cos(Lat_t) \cdot sin^2(\Delta Lon_{s,t})$$
$$h = 6,371km \cdot 2 \cdot atan2(\sqrt{a}, \sqrt{1-a})$$

which computes the distance (in kilometers) between two points given their latitude and longitude [Veness, 2010].

4 Evaluation

In order to evaluate our method, we acquired from [GF, 2011] a graph that contained 3461 nodes connected by 9261 arcs. When our hierarchical graph was formed, the graph contained only 1867 nodes and 8159 arcs. This required 75% more storage between the high-level graph and the clusters, as can be seen in Table 1.

We chose 50,000 random pairs of nodes in the original graph to serve as the source and destination nodes for the tests. For each node pair, we ran four search algorithms: The Dijkstra and A^* algorithms on the original graph and

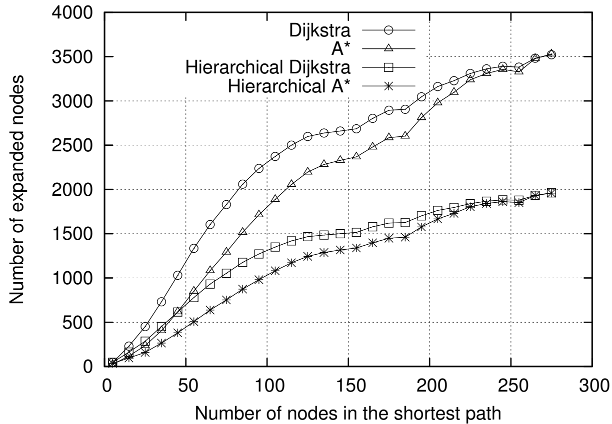

Figure 2: Number of nodes expanded on path length.

the Dijkstra and A^* algorithms on the hierarchical graph. The metrics for evaluating these algorithms included the route they found from the source to the destination node (whether or not the optimal path was found), how long the search took in microseconds, and how many nodes were expanded in the search process.

The first of these metrics was a non-factor, as all four algroithms always found the same, optimal route. Figure 1 shows the results of the time based on the length of the shortest path. The regular Dijkstra and A^* searches consistently took more time than their hierarchical counterparts and both A^* algorithms took more time to run than the Dijkstra algorithms did. In Figure 2, we show that, despite the increased time the A^* algorithm takes, it is expanding fewer nodes in its search.

5 Discussion

Looking at the results of our tests, we can see that our hierarchical graph was an improvement over the original one. The algorithms were faster on our graph and arrived at the same, optimal solution as those on the original graph did. However, it is interesting to note that the A^* search algorithm was, in both cases, slower than Dijkstra's algorithm at times. While Dijkstra's algorithm expands far more nodes, it does so in the time it takes the A^* method to choose which node to expand. This indicates that our heuristic for A^* was, while admissible, too large of a computation, negating part of the benefit of having a heuristic. On a larger graph, the A^* algorithm should overtake Dijkstra's algorthim in terms of speed as the number of nodes that Dijkstra's algorithm expands will be too great for it to make up the difference with A^*'s efficiency. In terms of storage, we believe that a 75% increase is tolerable, as GPS devices tend to be more limited on time than space.

Overall, our experiments were successful, although areas remain that we would like to explore further. Obviously, we would like to find a quicker heuristic function that is still a reasonably good estimate of the distance between nodes. In addition, we would like to compare some methods of determining where communities are located in the graph. Enough experimentation was done to find decent parameters for our current method, but they are not necessarily the optimal parameters. It would be interesting to combine this work with a genetic algorithm and more real world information to find optimal settings. Finally, we were not satisfied with the size of the map we tested on. A map of Italy was more desirable, at approximately four million nodes, but memory resources were rather limited on our test machines.

References

[GF, 2011] (2011). Geofabrik. `http://download.geofabrik.de/osm/`. A set of dumps of the OpenStreetMap database:.

[OSM, 2011] (2011). Openstreetmap. `http://www.openstreetmap.org/`.

[Jagadeesh et al., 2003] Jagadeesh, G., Srikanthan, T., and Quek, K. (2003). Heuristic techniques for accelerating hierarchical routing on road networks. *Intelligent Transportation Systems, IEEE Transactions on*, 3(4):301–309.

[Russell and Norvig, 2009] Russell, S. and Norvig, P. (2009). *Artificial intelligence: a modern approach*. Prentice hall, 3rd edition.

[Song and Wang, 2010] Song, Q. and Wang, X. (2010). Efficient routing on large road networks using hierarchical communities. *Intelligent Transportation Systems, IEEE Transactions on*, PP(99):1–9.

[Veness, 2010] Veness, C. (2010). Calculate distance, bearing and more between latitude/longitude points. `http://www.movable-type.co.uk/scripts/latlong.html`.