CS225 Final Project Goals

Dataset

The Dataset we are choosing to use is the “OpenFlights'. OpenFlights is an open source database where one can map their flights in all sorts of ways and use the data to create a project of their own. You can find all this information from the following link: https://openflights.org/data.html.

From the different datasets provided by OpenFlight we have decided to use the “Route” and “Airport” databases, to ensure we can get the names of the airports and the different routes from different airports. The airport dataset provides the data in a CSV format where we would use the latitude and longitude to calculate the distances for weighting our graph's edges. We will use the airports as the nodes of the graph. The route dataset provides us with source and destination airport identification, which will be converted to edges of the graph.

Traversal: BFS

We will implement a Breadth First Traversal for this project.

Implementation Steps:

1. Add root node to queue
2. Label queue as visited
3. While queue is not empty:
    a) Dequeue
    b) If dequeued value is goal, return
    c) Enqueue all non-visited adjacent edges of the dequeued node
    d) Mark all enqueued edges from c as visited and go back to a)

Covered Algorithm: Dijkstra

The first algorithm we are using is the Dijkstra’s algorithm for finding the shortest path between the two airports. Dijkstra’s algorithm is used to find the shortest path tree from a source node by building a set of nodes that has the minimum distance from the source.

Implementation Steps:

1. Set all vertices distances = infinity except for the source vertex, set the source distance = 0.
2. Push the source vertex in a min-priority queue in the form (distance , vertex), as the 3. comparison in the min-priority queue will be according to vertex distances.
4. Pop the vertex with the minimum distance from the priority queue (at first the popped vertex = source).

4. Update the distances of the connected vertices to the popped vertex in case of "current vertex distance + edge weight < next vertex distance", then push the vertex with the new distance to the priority queue.
5. If the popped vertex is visited before, just continue without using it.
6. Apply the same algorithm again until the priority queue is empty.

Non-covered Algorithm: Landmark Path

The landmark path algorithm decreases time to calculate the shortest distance by preprocessing the data to create landmarks. Landmarks are points whose distance from every other point in the data is stored. The shortest path is then calculated when queried by finding the shortest path from queried point a to queried point b amongst the landmark paths.

Implementation Steps:
1. Determine Landmarks:

    We will attempt several methods to find optimal landmarks:
    a) Random nodes
    b) Nodes with most connected edges
    c) Self-selected nodes (this will probably be similar to b)

    We will also play with the number of landmarks selected. Too many landmarks will lead to a large amount of disk usage, as well as a longer final path calculation time, as paths through each landmark will have to be compared. Too few will lead to suboptimal paths forced through far-away landmarks.

2. Build Landmarks:

    We will pre-calculate the distance from each landmark to every other node in the graph using Dijkstra's algorithm.

3. Query

    Compare the paths through each landmark and use the shortest one.
    If we decide to advance the algorithm more we can select only landmarks that are likely to contain a good path then compare these ones.

Complexity:
(Taken from source 1)
Landmark Building:

    Time: O(m + log(n)), where m is number of edges and n is number of nodes
    Space: O(n)

Query:

    Time: O(rk), where r is number of landmarks and k is length of path.

Sources:

1. LPI: Approximating Shortest Paths using Landmarks, by Kevin Grant and David Mould
https://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.333.2161&rep=rep1&type=pdf

2. Dijkstra's shortest path algorithm: https://en.wikipedia.org/wiki/Dijkstra%27s_algorithm

3. BFS: https://en.wikipedia.org/wiki/Breadth-first_search