Geographic Analysis and Pathfinding System

Kashyap Panda - U40571904 - kpanda@bu.edu
Jordan Koseski - U69593936 - jkoseski@bu.edu
John Burke - U31936503 - jwburke@bu.edu
Ritesh Rana - U80375845 - ritesh19@bu.edu

Abstract

Navigation problems such as the traveling salesman problem (TSP) have been of significant interest for research in algorithms. With the adoption of smartphone technology into common use within the past decade and a half, increasing demands have been made for efficient, robust algorithms that can navigate individuals from one place to another at a click of a button. This paper explores two path planning algorithms - Dijkstra's algorithm and the A* search algorithm - for finding efficient driving routes between cities in the United States. The implementation reads in a dataset of contiguous US cities, and constructs a graph representation where edges connect cities within a distance threshold. Dijkstra's algorithm iterates through all nodes, greedily updating path distances, to find shortest paths from a start city to all destinations. In contrast, A* selectively explores the graph by making informed decisions based on a heuristic function to find a least-cost path from start to goal without considering all nodes. Both algorithms are implemented in C++ and can generate cross-country driving routes by connecting intermediate city stopovers between a given start and end location. Sample results demonstrate the algorithms generating optimal paths between locations such as Augusta, ME and Concord, NH; Austin, TX and Cambridge, MA; Charlotte, NC and New York City, NY. Timing comparisons show A* performing faster than Dijkstra on shorter routes while being slower across country-spanning trips. Runtime efficiency and optimality represent key tradeoffs between the two methods. Overall, the project provides two different approaches to trip planning over real geographic data.

I. Introduction

As the fifth largest country by area in the world, the United States is filled with interesting cities and locations spanning thousands of miles. As a relatively affordable method of travel, road-tripping across the country has become a popular method of transit dating back to the advent of the highway system in the mid-1950's. Although road-tripping gives the freedom to the traveler to make their own itinerary and travel at their own pace, there are inherent challenges faced when planning a route across the United State. Across the many logistical difficulties of routing a trans-continental drive, questions of where to rest and find gas are of the highest priority when considering the safety of the travelers. To that end, we implement two algorithms which connect cities along a route connecting a start and end location to provide intermediate stops for drivers to safely rest along their travels. Given the user's location and desired goal, the algorithm will scan cities in the contiguous United States and output an efficient route on which to travel.

II. Dataset & Graph Construction

<u>Dataset</u>

Our implementation works with a CSV dataset which contains detailed information about a list of cities in the United States [1]. The dataset includes many fields for each city, and the code reads this information into the program, processing it as a vector of 'City' structs. Precisely, we represent the data utilizing the following fields:

1) *Name* - A string indicating the name of the city.
2) *State*- A string representing the state in which the city is located.

3) *Latitude*- A double-precision floating-point number indicating the geographical latitude of the city.
4) *Longitude*- A double-precision floating-point number indicating the geographical longitude of the city.

Additionally, the code skips cities located in the states of Hawaii ("HI") and Alaska ("AK"). We skip these areas as they are not contiguous with the continental United States, and we want to focus on analyzing cities within the contiguous states, to make the idea of road-tripping between locations feasible.

After reading in the dataset, the code constructs a graph from the dataset of cities by calculating distances between each pair of cities using the Haversine formula. This formula is specifically designed for calculating distances on a sphere, such as an approximation of the Earth. The steps involved in constructing the graph are as follows:

Distance Calculation:

- *Haversine Formula:* The Haversine formula calculates the great-circle distance between two points on a sphere, given their latitudinal and longitudinal coordinates. This is essential for determining the distances between cities accurately.
- *Iterative Calculation:* The code calculates the distance using the Haversine formula for each pair of cities in the dataset. The result is the geographical distance between the two cities on the Earth's surface.

Graph Creation:

- *Adjacency Matrix Representation:* The graph is an adjacency matrix, where each row and column correspond to a city, and the matrix elements store the distances between cities.
- *Distance Threshold:* The code introduces a distance threshold constant, *DISTANCE_THRESHOLD_KM*, which acts as a filter for determining whether a direct edge (path) between two cities should be considered in the graph. This threshold controls for the maximum allowable length of any given edge in the graph.
- *Routing Through Intermediate Nodes:* Due to the aforementioned conditional edge creation, the code influences the graph's connectivity. Geographically distant cities will not have a direct edge between them if their calculated distance exceeds the threshold. This forces paths to reroute through intermediate nodes when cities are far apart, adding a level of control to the graph's structure.

```cpp
double toRadians(double degree) {
    return degree * (M_PI / 180.0);
}

double calculateDistance(double lat1, double lng1, double lat2, double lng2) {
    // Haversine formula to calculate the distance between two points on the Earth
    double dLat = toRadians(lat2 - lat1);
    double dLng = toRadians(lng2 - lng1);
    lat1 = toRadians(lat1);
    lat2 = toRadians(lat2);

    double a = sin(dLat/2) * sin(dLat/2) + sin(dLng/2) * sin(dLng/2) * cos(lat1) * cos(lat2);
    double c = 2 * atan2(sqrt(a), sqrt(1-a));

    return EARTH_RADIUS_KM * c;
}
```

Figure 1: Our implementation of Haversine distance calculation in C++

```cpp
unordered_map<string, vector<pair<string, double>>> createCityGraph(const vector<City>& cities) {
    unordered_map<string, vector<pair<string, double>>> graph;

    for (const auto& city1 : cities) {
        vector<pair<string, double>> cityDistances;
        for (const auto& city2 : cities) {
            if (city1.name != city2.name) {
                double distance = calculateDistance(city1.lat, city1.lng, city2.lat, city2.lng);
                if (distance <= DISTANCE_THRESHOLD_KM) {
                    string city1Name = getCityState(city1);
                    string city2Name = getCityState(city2);

                    cityDistances.push_back(make_pair(city2Name, distance));
                }
            }
        }
        // Use concatenated name for graph key
        graph[getCityState(city1)] = cityDistances;
    }

    return graph;
}
```

Figure 2: Our algorithm in C++ for graph construction given a vector of cities

III. Algorithm Design

Dijkstra's algorithm

For our first algorithm to implement in finding shortest path between cities, we elected to implement Dijkstra's As implemented in the provided C++ code, Dijkstra is a graph search algorithm designed to find the shortest paths from a source node to all other connected nodes in a weighted graph. Here, the algorithm operates on a set of cities represented by geographical coordinates. Figure 1 contains our code for implementing Dijkstra's shortest distance algorithm in our project.

The algorithm starts by initializing three data structures: predecessor_list, minHeap, and dist_vec. The predecessor_list is a map that will store the predecessor of each city in the shortest path. The minHeap is a priority queue that will keep track of cities and their distances from the start city, and dist_vec is a map that records the shortest distance from the start city to each city.

The algorithm then sets the initial distances for all cities. For each city in the graph, the initial distance is set to infinity except for the start city, whose distance is set to 0. This initialization prepares the algorithm for the following iterative process of exploring and updating all distances.

The algorithm's main loop runs until the priority queue (minHeap) is empty, signifying that all reachable cities have been explored. Within the loop, the algorithm extracts the city with the minimum distance from the priority queue in each iteration. It then explores the neighbors of the current town, calculating potential new distances through the present city to each neighbor.

If the new distance is shorter than the current recorded distance for a neighbor, the distance is updated, and the neighbor is added to the priority queue with its new distance. The predecessor of each neighbor is also updated to be the current city if a shorter path is found. This process efficiently explores the graph, prioritizing paths with the fastest recorded distances at each step.

The algorithm continues until the priority queue is empty. At the end of its execution, the predecessor_list contains the necessary information to reconstruct the shortest paths from the start_city to all other cities in the graph. The primary function then utilizes this information to rebuild and print the shortest path from the specified start city to the end city.
As implemented in the provided code, Dijkstra's algorithm guarantees an efficient result by systematically exploring the graph and updating distances based on the shortest paths discovered.

```cpp
unordered_map<string, string> dijkstra(unordered_map<string, vector<pair<string, double>>>& graph, string start_city) {

    unordered_map<string, string> predecessor_list;
    priority_queue<pair<double, string>, vector<pair<double, string>>, greater<pair<double, string>>> minHeap;
    unordered_map<string, double> dist_vec;

    for (const auto& entry : graph) {
        dist_vec[entry.first] = numeric_limits<double>::max();
    }


    dist_vec[start_city] = 0;
    minHeap.push({0, start_city});

    while (!minHeap.empty()) {
        string curr_city = minHeap.top().second;
        minHeap.pop();

        for (const auto& neighbor : graph[curr_city]) {
            string neighbor_city = neighbor.first;
            double edge_weight = neighbor.second;

            if (dist_vec[curr_city] + edge_weight < dist_vec[neighbor_city]) {
                dist_vec[neighbor_city] = dist_vec[curr_city] + edge_weight;
                predecessor_list[neighbor_city] = curr_city;
                minHeap.push({dist_vec[neighbor_city], neighbor_city});
            }
        }
    }

    return predecessor_list;
}
```

Figure 3: Our implementation of Dijkstra's Algorithm in C++ for Pathfinding

A* algorithm

For our second algorithm, we opted to implement a version of the A* algorithm, which has been popularized in the field of control theory as an efficient graph search method for connecting two nodes. As seen in Figure 4, the algorithm operates in essence as an exploratory variant of Dijkstra's algorithm. Rather than efficiently iterating over the entire graph, A* probes at each timestep to take the next best action. For simplicity of implementation, our algorithm follows the version 1 structure, which is only marginally less optimized than the version 2 method.

The algorithm begins similarly to Dijkstra's with the creation of a distance vector f(x) and a backpointer vector. For both vectors, each entry corresponds to a node on the graph and all values are initialized such that their starting distance is infinity and their backpointer is None. The algorithm then takes the start node, which is given as input, and pushes it into a minimum-priority queue, stored as f(x) = 0 and a backpointer that is either empty or self-referential.

Algorithm 1 The A* algorithm. **Ver. 1** is a simplified, complete but not optimal version. **Ver. 2** is the full optimal version.

1: Add the starting node n_{start} to O, set $g(n_{start}) = 0$, set the `backpointer` of n_{start} to be empty, initialize C to an empty array. ▷ Initialization
2: **repeat**
3: Pick n_{best} from O such that $f(n_{best}) \leq f(n_O)$ for all $n_O \in O$. ▷ PriorityMinExtract
4: Remove n_{best} from O and add it to C.
5: **Ver. 1** **if** $n_{best} = n_{goal}$ **then**
6: Return path using backpointers from n_{goal}. ▷ Path
7: **end if**
8: **Ver. 2** **if** $g(n_{goal})$ is set **and** $g(n_{goal}) \leq f(n_O)$ for all $n_O \in O$ **then**
9: Return path using backpointers from n_{goal}. ▷ Path
10: **end if**
11: $S = \text{ExpandList}(n_{best})$
12: **for** all $x \in S$ that are not in C **do** ▷ ExpandElement
13: **if** $x \notin O$ **then**
14: Set the backpointer cost $g(x) = g(n_{best}) + c(n_{best}, x)$.
15: Set the `backpointer` of x to n_{best}.
16: Compute the heuristic $h(x)$. ▷ Heuristic
17: Add x to O with value $f(x) = g(x) + h(x)$. ▷ PriorityInsert
18: **else if** $g(n_{best}) + c(n_{best}, x) < g(x)$ **then** ▷ A better path to x has been found
19: Update the backpointer cost $g(x) = g(n_{best}) + c(n_{best}, x)$.
20: **Ver. 2** Update the value of $f(x) = g(x) + h(x)$ in O (might require computing $h(x)$ again). ▷ Heuristic
21: Update the `backpointer` of x to n_{best}.
22: **end if**
23: **end for**
24: **until** O is empty

Figure 4: Pseudocode of the A algorithm [2]*

After initialization, the algorithm begins its iteration phase, which effectively explores along the graph using informed decision making to find a path. After popping the minimum value from the queue, that node is marked as visited, meaning it will no longer be considered in future iteration. This step prevents backtracking if cycles occur within the graph. The next step is to expand the list at the chosen node by running a breadth-first search to find all the immediate neighbors. After this set is determined, the algorithm iterates over all non-visited neighbors and updates our minimum-priority queue. For each node in the set of neighbors, if that node is not already in the queue, we start by updating its weight to equal the weight of its parent plus the cost of the edge between them and change its backpointer to reference the parent node. Next, we calculate the

total value f(x) for the node by adding a calculated heuristic to the cost. As previously mentioned, this algorithm explores using informed decisions, which are made through the use of the heuristic value. Effectively, the heuristic can be seen as a method of tracking the "goodness" of a given node, and it is only calculated when that node is being pushed to the queue. While the heuristic realistically could represent a number of factors, our implementation utilizes the relative distance of the node to the goal location. After calculating this value and adding it to our cost to acquire the f(x) value, the node is then pushed into our minimum priority queue. In the case that our node does already exist within the queue, the algorithm checks to see if the f(x) value obtained through the current intermediate path is lower than the current value, and then updates the f(x) and backpointer accordingly if it is.

The algorithm has two potential termination conditions, the first being that the goal node is found and the second being that our priority queue has no further entries to scan. In the case that our goal node is found, the array of backpointers is returned and used to reverse-construct a route from our source to the destination. However, if the queue empties before the goal is reached, this means that there is no feasible solution to the problem and that our source and destination do not reside within the same connected component.

The benefit to running the A* algorithm is that it guarantees an efficient result if a feasible solution can be reached. Additionally, by making informed decisions along the way, A* can terminate without visiting every node, which can lead to drastic improvements in performance over one-to-all methods such as Dijkstras. Although this method is fast and efficient, it is limited by the fact that it does not guarantee the optimal solution since it does not consider all nodes. Additionally, it is less space efficient than Dijkstra's due to the fact that extra information must be stored about each node to calculate the heuristic.

IV. Instructions for Running Program

The steps for cloning, compiling, and running are code are as follows:

1) Clone the repository: https://github.com/EC504-Pathfinding/geo-data-analysis
    a) This contains our report, input files, output files, source code, as well as our makefile.
    b) Note that "input/full_adjacency_list.csv" is a nearly 200 MB document
2) Execute "make" in the terminal while inside of the cloned git repository directory
3) Execute "./GeoDataAnalysis" in the terminal after executing make, this will run our program

The steps for utilizing the program are as follows:

1) Allow the program to finish constructing the graph from the input file located at "input/full_adjacency_list.csv". This may take up to a minute depending on the speed of your computer.
2) Upon completion of the prior step, the program should prompt the user to enter an initial city. This can be any city present within cities.csv, besides Islamorada, FL as it had improper data formatting.
    a) The way in which this city is entered should be as follows: CITY STATE CODE. This means Boston should be entered as Boston MA and New York should be

entered as New York NY. This allows our city identifiers to be unique across the nation.

3) After entering an initial city the program will prompt for a destination city. The input formatting for this city is the same as the first city input.
4) After entering these two cities, the program will execute and output resulting information for both Dijkstra's and the A* algorithm. It will also write a unique output file for each two cities analyzed to "output/SourceCity DestinationCity.txt"
    a) The output will also be printed to the command line. It will also inform the user about the time it take to analyze the shortest path

V. Sample Results & Discussion

Here are some sample results from example source and destination cities as analyzed by our algorithm:

- Augusta ME → Concord NH
    - Dijkstra and the A* algorithm gave the same resulting path
    - Time taken:
        - Dijkstra - 12.391 seconds
        - A* - 0.916 seconds
    - Path Computed:
        - Augusta ME → Steep Falls ME → Concord NH
- Austin TX → Cambridge MA
    - Dijkstra and the A* algorithm gave the same resulting path
    - Time taken:
        - Dijkstra - 12.812 seconds
        - A* - 13.878 seconds
    - Path Computed:
        - Austin TX → Buckholts TX → Kosse TX → Fairfield TX → Chandler TX → Gilmer TX → Douglassville TX → Oakhaven AR → Donaldson AR → College Station AR → Griffithville AR → Weiner AR → Cardwell MO → Wardell MO → Bardwell KY → Brookport IL → Sturgis KY → Tennyson IN → Livonia IN → Crothersville IN → St. Leon IN → Oakwood OH → Arcadia OH → New London OH → Gloria Glens Park OH → Atwater OH → Oakland PA → Pine Hill NY → Housatonic MA → Easthampton MA → Marlborough MA → Cambridge MA
- Boston MA → Los Angeles
    - Dijkstra and the A* algorithm gave the same resulting path
    - Time taken:
        - Dijkstra - 12.172 seconds
        - A* - 13.218 seconds
    - Path Computed:
        - Boston MA → Fiskdale MA → Winsted CT → Clintondale NY → Watchtower NY → Wallenpaupack Lake Estates PA → Georgetown PA → Salem OH → Oakwood OH → New Haven IN → Akron IN → Francesville IN → Clifton IL → Pontiac IL → Spring Bay IL → Roseville IL → Donnellson IA → Glenwood MO → Lucerne MO → Gentry MO →

Barada NE → Humboldt NE → Steele City NE → Republic KS → Athol KS → Edmond KS → Rexford KS → Goodland KS → Bethune CO → Hugo CO → Ellicott CO → Rock Creek Park CO → Howard CO → Bonanza CO → Cathedral CO → Rico CO → Lewis CO → Tselakai Dezza UT → Halchita UT → Navajo Mountain UT → Big Water UT → Fredonia AZ → Cane Beds AZ → Scenic AZ → Moapa Valley NV → Las Vegas NV → Tecopa CA → Fort Irwin CA → Silver Lakes CA → La Cañada Flintridge CA → Los Angeles CA

- Charlotte NC → New York NY
    - Dijkstra and the A* algorithm gave the same resulting path
    - Time taken:
        - Dijkstra - 12.446 seconds
        - A* - 8.0514 seconds
    - Path Computed:
        - Charlotte NC → Mocksville NC → Stuart VA → Salem VA → Hot Springs VA → Whitmer WV → Jennings MD → Oakland PA → Starrucca PA → Washington Heights NY → Woodbury NY → New York NY
- Lewis CO → Rico CO
    - Dijkstra and the A* algorithm gave the same resulting path
    - Time taken:
        - Dijkstra - 13.0116 seconds
        - A* - 0.9346 seconds
    - Path Computed:
        - Lewis CO → Rico CO

Looking at our sample results, we see that no matter how far or how short the distance is for our two algorithms across the contiguous graph, they yield the same path to get between source and destination. The difference that we are able to see in our analysis of the output is that, depending on the relative location of the source to the destination node within the graph, A* can take either much shorter than Dijkstra's algorithm, or around the same amount of time, or possibly even longer due to the heuristic calculation.

When our algorithms are required to analyze a path which traverses across a majority of the graph, for example the paths from Austin to Cambridge and Boston to Los Angeles, the A* algorithm takes longer than our Dijkstra implementation. The reason for this is likely that when the paths need to be evaluated across such a stretch of the graph, both Dijkstra and A* look at around the same amount of nodes. A* also has to calculate a distance heuristic, so it will have to take slightly longer than Dijkstra in this case to account for those calculations at every step.

When our algorithms are required to analyze a path which traverses a good amount of the country, but not across the entire nation, such as the path from Charlotte to New York City, Dijkstra performs slower than A*. This is because Dijkstra is going to perform with around the same time taken for each path it calculates. Dijkstra will always find every optimal path from the source to all destination nodes, looking at the same number of nodes each time it is called. However, A* is capable of not looking at every single node in the graph, so for a smaller path it

is able to locate the optimal path faster.

A* really shines when we pick very close cities for our source and destination, such as Augusta and Concord as well as Lewis and Rico. For both of these cases, A* took less than a second compared to Dijkstra taking around 13 seconds. This is for the same reason that was stated above. Dijktsrta is going to take the same amount of time for each path analysis given that it is running under the same conditions (hardware). A* is able to find this short path efficiently and then return with the optimal answer without analyzing every node.

VI. References

[1] Nuss, Sergej. "United States Cities Database." Kaggle, 2022, www.kaggle.com/datasets/sergejnuss/united-states-cities-database. Accessed 2 Dec. 2023.

[2] Tron, Roberto. *"ME570 Robot Motion Planning Homework 4," Boston University, Fall 2023.*