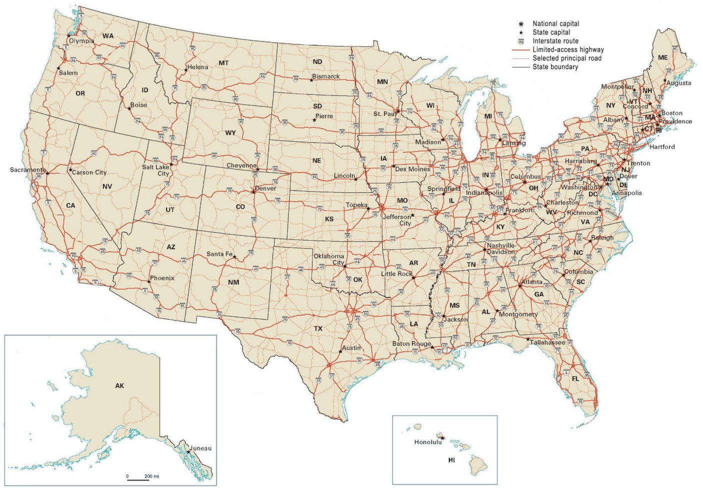
Geographic Data Analysis and Pathfinding System

Kashyap Panda, Jordan Koseski, John Burke, Ritesh Rana

Problem

- The United States is the 5th largest country by land area in the world.
  - Common travel method is road-tripping
  - How can we reach out destination while hitting points of interest?



 [US Road Map: Interstate Highways in the United States - GIS Geography]

Our Dataset

- CSV file, containing multiple fields of information about a list of many cities in the US
  - Read each of these cities into program as a vector of City structs
  - Skip cities in either state “HI” or “AK” as they are not contiguous
- City struct:
  - String name
  - String state
  - Double lat
  - Double lng

Constructing the Graph

- Distance Calculation
  - Every city has its distance to every other city calculated using the Haversine formula
- Graph Creation
  - If the distance between a given pair of cities is less than some threshold, DISTANCE_THRESHOLD_KM, then the distance is set in the adjacency matrix, otherwise there is no path between the two cities
    - This allows for us to control how long edges can be, forcing paths to reroute through intermediate nodes when cities are far apart

Code for Distance Calculation and Graph Creation

```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;
}
```

```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;
}
```

Dijkstra Algorithm

- Selects the node with the shortest distance to the source node
  - The next node to select is tracked in the program using a min-heap priority queue
  - Benefits:
    - Guarantees that the solution will be the exact perfect solution, so that the path is the shortest possible
    - One run of Dijkstra evaluates distances for all nodes to the source node
  - Limitations:
    - Must visit all nodes to evaluate the distances

Dijkstra Code

```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;
}
```

A* Algorithm

- Nearest-neighbor algorithm that uses a BFS to pick the next best neighbor (n_best)
  - Tracks the decision using a queue ordered by min-first priority
  - Benefits:
    - Guarantees result if path is feasible (i.e. nodes reside within the same connected component)
    - Finds efficient path without visiting all nodes
  - Limitations:
    - Less space efficient than Dijkstra
    - Guarantees efficient solution, not necessarily optimal

A* Pseudocode

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

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
repeat
  Pick n_{best} from O such that $f(n_{best}) \leq f(n_O)$ for all $n_O \in O$. ▷ PriorityMinExtract
  Remove n_{best} from O and add it to C.
  Ver. 1 **if** $n_{best} = n_{goal}$ **then**
    Return path using backpointers from n_{goal}. ▷ Path
  end if
  Ver. 2 **if** $g(n_{goal})$ is set **and** $g(n_{goal}) \leq f(n_O)$ for all $n_O \in O$ **then**
    Return path using backpointers from n_{goal}. ▷ Path
  end if
  $S = \text{ExpandList}(n_{best})$
  for all $x \in S$ that are not in C **do** ▷ ExpandElement
    if $x \notin O$ **then**
      Set the backpointer cost $g(x) = g(n_{best}) + c(n_{best}, x)$.
      Set the `backpointer` of x to n_{best}.
      Compute the heuristic $h(x)$. ▷ Heuristic
      Add x to O with value $f(x) = g(x) + h(x)$. ▷ PriorityInsert
    else if $g(n_{best}) + c(n_{best}, x) < g(x)$ **then** ▷ A better path to x has been found
      Update the backpointer cost $g(x) = g(n_{best}) + c(n_{best}, x)$.
      Ver. 2 Update the value of $f(x) = g(x) + h(x)$ in O (might require computing $h(x)$ again). ▷ Heuristic
      Update the `backpointer` of x to n_{best}.
    end if
  end for
until O is empty

Source: 49955523 (blackboardcdn.com) (Roberto Tron ME570)

Testing & Analysis

- Boston, MA to San Francisco, CA (4980.6 km)
- Boston, MA to Chicago, IL (1367.67 km)
- Boston, MA to New York, NY (346.0 km)

Testing & Analysis

- Boston, MA to San Francisco, CA (4980.6 km)
- Route: Boston MA, Fiskdale MA, Winsted CT, Clintondale NY, Watchtower NY, Wallenpaupack Lake Estates PA, Georgetown PA, Salem OH, Oakwood OH, Woodburn IN, Winona Lake IN, La Crosse IN, Cedar Lake IN, Minooka IL, La Moille IL, Le Claire IA, University Heights IA, Williamsburg IA, Newton IA, Bondurant IA, Lake Panorama IA, Audubon IA, Mondamin IA, Snyder NE, Leigh NE, Spalding NE, Sargent NE, Anselmo NE, Tryon NE, Arthur NE, Broadwater NE, Bridgeport NE, Hawk Springs WY, Chugwater WY, Rock River WY, Sinclair WY, Wamsutter WY, Superior WY, Fontenelle WY, Cokeville WY, St. Charles ID, Malad City ID, Malta ID, Oakley ID, Jackpot NV, Wells NV, Elko NV, Carlin NV, Valmy NV, Grass Valley NV, Lovelock NV, Wadsworth NV, Sunnyside-Tahoe City CA, Cameron Park CA, Vineyard CA, Clyde CA, San Francisco CA

- Identical routes?
  - Yes
- Dijkstra’s: 9.09166 seconds
- A*: 9.90502 seconds

Testing & Analysis

- Boston, MA to Chicago, IL (1367.67 km)
  - Route: Boston MA, Fiskdale MA, Winsted CT, Clintondale NY, Watchtower NY, Wallenpaupack Lake Estates PA, Georgetown PA, Salem OH, Oakwood OH, Antwerp OH, Syracuse IN, Lakeville IN, Beverly Shores IN, Chicago IL
- Identical routes?
  - Yes
- Dijkstra’s: 9.09956 seconds
- A*: 5.00597 seconds

Testing & Analysis

- Boston, MA to New York, NY (346.0 km)
  - Route: Boston MA, Millis-Clicquot MA, South Windham CT, Woodmont CT, New York NY
- Identical routes?
  - Yes
- Dijkstra’s: 9.04967 seconds
- A*: 1.115 seconds

Future Work

- Improve graph construction performance
- Read graph from input file
    - Rather than creating graph from CSV data each time
- Input validation
    - Check if start and destination cities exist before calculating route
- Further Experimentation
    - Analyze how these algorithms perform when we remove x% of the connections

Thank you!

Any questions?