CS575: Final Project Report

Project Title: Viewing U.S. Urbanized Areas Through the Lens of Data Structures and Algorithms

Team Member(s): Sean Kunz

I. Problem

In this project, I tackled two problems. The first of these problems involved managing user data. User data in this case was a list of all urbanized areas in the United States with populations over 10,000 [5]. Overall, almost 1,500 entries are included. This dataset can be used by many different specialists, such as transportation planners, demographers, and spatial analysts. While not quite as large as some other datasets, any code using this data could be improved by using a faster query method than iterating over a list. To accomplish this goal, a self-balancing red-black tree can be used. A red-black tree can improve query times by limiting the depth of a tree through balancing techniques.

The second problem I solved is finding the shortest path between two points. There are many different ways of doing this, with most solutions falling into two categories: single-pair shortest path algorithms and all-pairs shortest path algorithms. In this project, I worked towards modeling an ideal form of the United States Interstate Highway System, which is best suited to a single-pair shortest path algorithm, as all-pairs shortest path algorithms would include smaller areas that don't necessarily need highway/train stops, resulting in unnecessary computation. I implemented the A* search algorithm to assist in this modeling, where users can input any two cities as input and receive a path between the two as an output.

Lastly, I needed to obtain data for all of the nodes and edges for the graph used in the shortest path algorithm. I also used the node data as an input to the red-black tree. I utilized several APIs and algorithms such as Euclidean distance to accomplish this task.

II. Algorithms

When working on this project, I focused on implementing the red-black tree data structure and the A* Shortest Path Search Algorithm. I derived my implementation for the red-black tree from Cormen's Algorithm book [1], Wikipedia [2], and the University of South Florida [3]. I used Cormen's Algorithm book [1] and Wikipedia [4] for the A* Shortest Path Search Algorithm implementation.

A. *Red-Black Tree*

A red-black tree is a form of a self-balancing binary search tree. In the act of self-balancing, the tree can maintain close to ideal search conditions. To do this, the tree marks each node as either red or black while inserting and deleting from the tree. These nodes can change color throughout the process.

There are four properties that must be maintained by the tree: each node is either red or black, all NIL values are considered black, a red node does not have a red child, and every path from a given node to any of its descendants goes through the same number of black nodes [2]. These properties are maintained through six different insert cases and six different delete cases.

B. *A* Search Algorithm*

The A* Search Algorithm is a single pair shortest path algorithm. The algorithm finds the shortest path by maintaining a set of paths originating from the start until it finds the destination node. In this way, it is fairly similar to Dijkstra's Shortest Path Algorithm, but differentiates itself from the use of heuristics. The heuristic simply adds a value to the already calculated distance to further estimate cost. All nodes at the frontier of the search are stored in a priority queue and the node with the smallest distance + heuristic value is taken from the queue [4].

III. Software Design and Implementation

A. *Software Design*

The project can be broken down into four major software components: the data scraper, the red-black tree, the A* search algorithm, and the visualization.

The first of these components, the data scraper, provides the necessary information that serves as inputs to the other three parts. Two main scripts make up the whole of the scraper. The first script reads in a list of all urbanized areas in the United States with a population of over 10,000 [5]. It then geocodes these place names to receive latitude and longitude coordinates. Cities, in the case of this project, serve as nodes for the red-black tree as well as nodes for the graph that the A* search algorithm traverses. The second script generates edges from the city data. The Euclidean distance is calculated for each city in the dataset. All nodes within a certain distance are considered neighbors to the search node and an edge is generated between the two. The distance used varies by state and is dependent on population density. This helped to bridge gaps in rural states where cities are more than 200+ kilometers apart.

The second component is the red-black tree. As stated previously, it takes a list of cities as inputs to build a tree. Once the initial dataset is loaded, users can choose four options: insert a new node, delete a node, print the tree, and write to a CSV. Inserting and deleting work as one would expect and continue to maintain the integrity of the tree. Printing the tree gives a level-order traversal view of the tree. Writing out to the CSV gives a file with nodes in an in-order manner. Nodes (cities) are sorted by their Federal Information Processing Standard code.

The third and fourth component relate to the A* search algorithm. The algorithm reads in the computed cities and edges and generates a graph. Users can call a standalone program or use the visualization tool, which is a web interface for the search algorithm. After inputting a starting node and a destination node, the shortest path is calculated. The graph is weighted for distance. The heuristic used considers the population of the destination city. While this might make a path longer by distance it produces a more "realistic" path. For example, a real-life path between New York City and Los Angeles would not go through a series of small towns, but it would rather take a slightly longer route to ensure it passes through other cities. This search algorithm implementation aims to achieve the same goal.

B. *Implementation and Tools Used*

I used a variety of languages and tools to build my project. The node and edge generation scripts were written in Python. The node generation script also used the geocoding tool in the ArcGIS REST API. This allowed me to quickly and efficiently get latitude/longitude coordinates for all input cities. Results were then written to a MongoDB database and a CSV file (for testing). The edge script read in data from the CSV/MongoDB database and calculated the Euclidean distance and wrote results back out to another database/CSV file.

The red-black tree was written in Golang. I thought Go was a good choice here because it has better performance than Python and is more of a "systems" appropriate language. The implementation was fairly traditional and allowed users to insert or delete a node, print the tree, and write to a CSV. The red-black tree was built completely from the ground up, with no existing tools used.

The shortest path algorithm was also written in Golang. My thought process for using Go over Python in this case was similar to that of the red-black tree. While my graph wasn't large enough for performance to play a large impact (~1,500 nodes and 10,000+ edges) it would be important for the search to perform well in a real-world situation. This is because it serves as the backend to my website and users would likely click away from the site if it ran too slowly. The implementation existed as a standalone program as well as a library for the web interface. The base algorithm was implemented entirely from scratch.

The website/visualization tool was written in Golang as well. I chose Golang here because I have limited experience with web development and wanted to choose a language I was already familiar with, rather than jumping into something like JavaScript. However, some parts of the code were written in JavaScript/HTML. The HTML was necessary for display purposes and the JavaScript was necessary for the use of mapping visualizations.

This website allows users to input a starting city and an ending city from within the valid city set to find a shortest path. After submitting the appropriate values, users are redirected to another screen with a map. If a path exists between the two cities, it is drawn on the map. This map visualization was created with the ArcGIS JavaScript API, but the rest of the site was created from scratch.

Overall, I am happy with how this project turned out. If I had more time, I would integrate MongoDB more into the red-black tree data. For example, instead of reading in and writing out to a CSV, I could query the database for points. One other change I would make would be adding points (representing cities) that the user could click on to retrieve more information along the shortest path visualization. Right now, the path is represented by a line. The user can zoom in and see the city labels on the basemap, but I think it would be better to label all stops along the path somewhere.

IV. Project Outcomes

- Code: https://github.com/skunz42/CS575
- Video: https://youtu.be/X4iOLOEfxrY
- Slides: https://github.com/skunz42/CS575/blob/main/docs/CS%20575%20Final%20Project.pdf

References

[1] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein, Introduction to Algorithms, Third Edition, The MIT Press, 2009.

[2] Wikimedia Foundation. (2022, April 13). *Red–Black Tree*. Wikipedia. Retrieved from https://en.wikipedia.org/wiki/Red%E2%80%93black_tree

[3] University of South Florida. (n.d.). *Red/Black Tree*. Red/Black Tree Visualization. Retrieved from https://www.cs.usfca.edu/~galles/visualization/RedBlack.html

[4] Wikimedia Foundation. (2022, April 12). *A* search algorithm*. Wikipedia. Retrieved from https://en.wikipedia.org/wiki/A*_search_algorithm

[5] US Census Bureau (2021, October 8). *2010 census urban and Rural Classification and Urban Area Criteria*. Census.gov. Retrieved from https://www.census.gov/programs-surveys/geography/guidance/geo-areas/urban-rural/2010-urban-rural.html