CS 575 Final Project

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

Sean Kunz

Problem

- Geospatial data can be applied in a variety of different contexts
- Two notable areas are storage and path-finding
- I implemented two solutions to these problems
  - Storage - Red-Black Tree
  - Path-Finding - A* Search Shortest Path Algorithm
- To demonstrate the implementation of these problems, I used all U.S. urbanized areas with over 10,000 people as my data input

Algorithms - Red-Black Tree

- Solution 1: Red-Black Tree
- A red-black tree is a form of a self-balancing binary search tree
- A self-balancing tree can minimize search time by preventing excessive depths
- In a red-black tree, each node is marked as either red or black
- A red-black tree must maintain four properties based on the nodes’ colors to ensure that it balances the tree correctly

Algorithms - Red-Black Tree Properties

- Properties
  - 1. Each node is either red or black
  - 2. All NULL values are considered black
  - 3. A red node does not have a red child
  - 4. Every path from a given node to any of its descendants goes through the same number of black nodes
- These properties are maintained through six different insert cases and six different delete cases

Algorithms - A* Search Algorithm

- Solution 2: A* Search Algorithm
- Single-source shortest path algorithm
- Set of nodes (cities) and weighted edges (city connection distances)
- The algorithm maintains a set of paths originating from the source until the destination node is found
- Uses a heuristic - in my case, this is city population
- As nodes are discovered they are placed into a priority queue and the nearest node + heuristic is removed

Software Design and Implementation

- The project was broken down into four major subcomponents
  - 1. Data Scraping
  - 2. Red-Black Tree
  - 3. A* Shortest Path Algorithm
  - 4. Shortest Path Visualization
- The code was written in a mixture of Golang, Python, JavaScript, and HTML

Implementation - Data Scraper

- Divided into two scripts - node generator and edge generator
- Provide the necessary information used as inputs to the two core programs
- Write results to a MongoDB database and a CSV (for testing)
- Both scripts written in Python

Implementation - Node Generator

- Receives a list of all U.S. urbanized areas above 10,000 residents (~1,500)
- Geocodes all cities to retrieve latitude and longitude coordinates
- Uses the ArcGIS REST API to geocode these cities

Implementation - Edge Generator

- Takes generated nodes as input
- Calculates neighbor nodes from inputs
- Neighbor nodes are nodes within a certain Euclidean distance
- Distance is dependent on the city’s state’s population density
  - High density: ~100 km (calculated via decimal degrees)
  - Mid density: ~150 km
  - Low density: ~250 km

Implementation - Red-Black Tree

- Written in Golang - better performance than Python and works better as a system level language
- Loads a CSV file of nodes initially
- Users can perform four actions
  - 1. Insert a node
  - 2. Delete a node
  - 3. Print the tree
  - 4. Write to a CSV

Implementation - Red-Black Tree

- Insert and delete are typical implementations
- Printing the tree prints it in level-order form
- Writing the tree to a CSV writes the nodes in in-order form
- Nodes are sorted by their FIPS (Federal Information Processing Standard) code

Implementation - Red-Black Tree

[DEMO]

Implementation - A* Search Algorithm

- Also written in Golang (for similar reasons as RBT)
- Loads in nodes and edges as inputs
- Exists as a standalone, command-line application as well as a module for a web interface
- Users input a starting city and an ending city and a path is returned

Implementation - A* Search Algorithm

- The algorithm uses a priority queue to fetch the “nearest” node
- Golang does not have a priority queue in its standard library, so it needed to be developed from the heap module
- The algorithm uses population as a heuristic - larger cities are more ideal as connecting points
    - Reasoning: A path from NYC to LA would not go through mostly rural areas. It would pass through larger cities within reason to maximize resource utilization. Highways and railroads follow this idea

Implementation - Visualization

- Written in Golang (backend), JavaScript, and HTML (frontend)
- Allows users to input a starting city and destination city and retrieve the shortest path
- This path is then drawn onto a map using the ArcGIS JavaScript API
- Results are cached in a MongoDB database (overkill for this small graph, but can be useful for large graphs)

Implementation - Visualization

[DEMO]

Possible Changes/Improvements

- Integrate MongoDB into the red-black tree
- Using more points for the input - urban areas are calculated down to 2,500 residents
- Creating a pop-up for the path on the website if clicked that details the whole path