CITY PATHFINDING

PROJECT REPORT

Introduction to AI, Project 1
Richard Habeeb, Anwesh Tuladhar

I. INTRODUCTION

Optimal pathfinding is an interesting, albeit well-understood, problem. This project is an exercise in designing and implementing software to search for an optimal paths among cities. In this report, we document our experiences and results from this project.

At a high-level our implementation was a front-end, web-based application written entirely in javascript. The application took in a list of city coordinates, a list of connections among the cities, and a number of configuration parameters. The output is a visualization of the optimal path between two cities.

II. BACKGROUND

To design a program to quickly find the shortest path between two cities on a given map, requires us to first build a model of the scenario. Naturally, the network of cities and roads correspond well to a cyclic directed graph, where cities are vertices and roads are edges. When we model the scenario in this way, we gain access to numerous, well-studied algorithms for finding the shortest path between two cities. Per specification, our design used the *A* search algorithm*, detailed in the following section.

The A* Search Algorithm

In the past, computer scientists have shown many ways to systematically find the optimal path between two vertices in a graph; however, the A* algorithm stands out in particular for it's ability to use *knowledge* to improve search performance. At each step in the search, a heuristic function is used to evaluate the quality of a given vertex. The heuristic function varies from application to application, but it always **must** be an underestimate of the remaining distance from some vertex to the destination vertex.

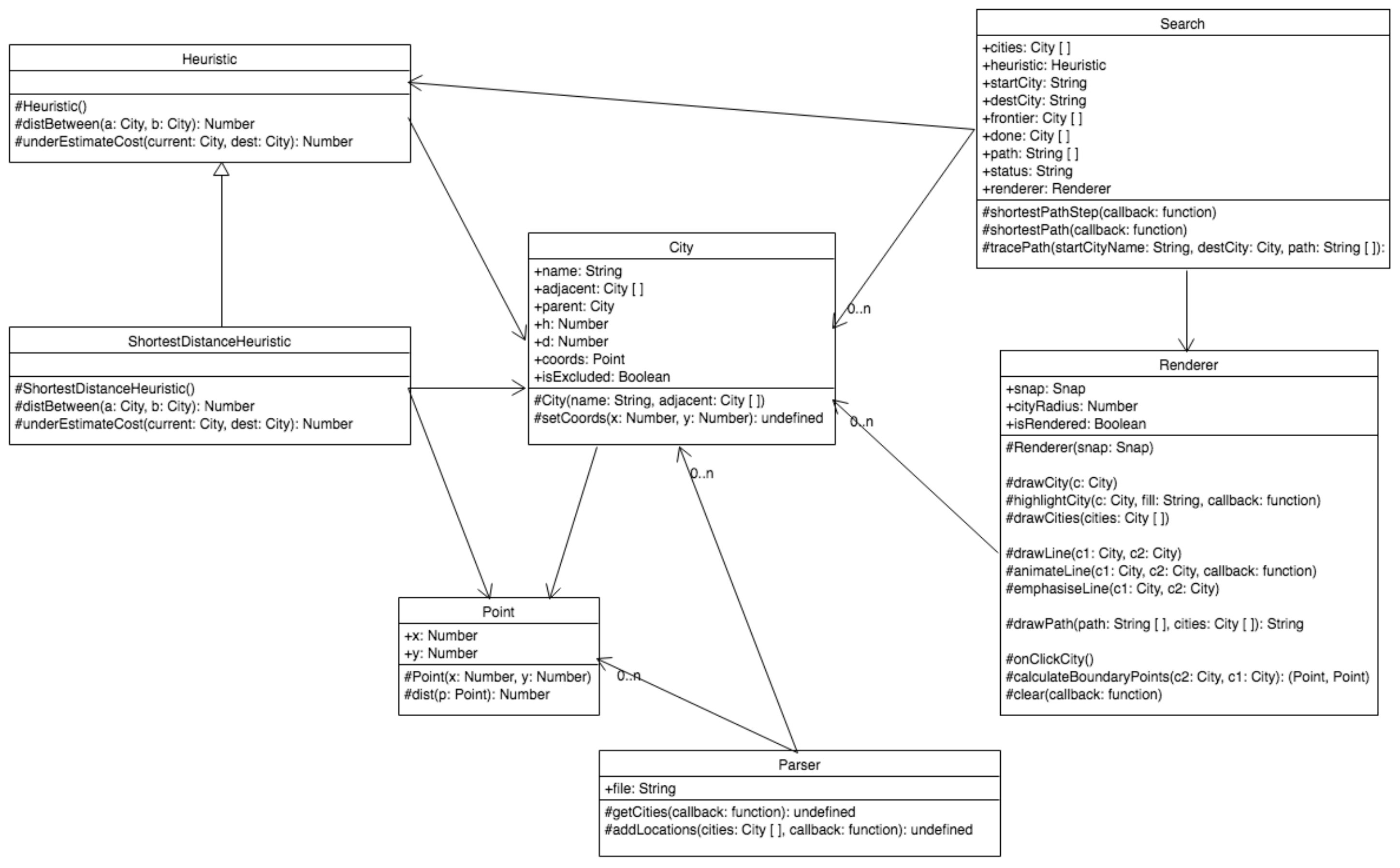
This algorithm works by maintaining three sets of vertices: unexplored, frontier, and explored. During each iteration of the main loop, the nodes on the frontier are evaluated to identify the highest quality searching location. That vertex is selected based on how much distance it has traveled plus the heuristic underestimate. From here the following steps occur:

1. If the current vertex is the destination: stop, search complete.
2. Otherwise: Iterate through each adjacent vertex to the current vertex
    a. If the adjacent node is in the explored set, skip this vertex.
    b. Estimate the distance between this adjacent vertex and the destination.
    c. If this distance is greater than any previous distances through this vertex, skip this vertex.
    d. Add this vertex to the frontier set. And set its parent to the current vertex.

After these actions, we start from the beginning of the main loop and evaluate the next best frontier node. Once the algorithm completes we just need to trace back through the parent references from the destination vertex back to the starting vertex.

III. DESIGN

We wrote seven classes in our design, following relatively conventional object oriented standards. The primary algorithm was included in the *search* class. A UML diagram is given below, followed by documentation of each class and method.


Heuristic
#Heuristic()
#distBetween(a: City, b: City): Number
#underEstimateCost(current: City, dest: City): Number
ShortestDistanceHeuristic
#ShortestDistanceHeuristic()
#distBetween(a: City, b: City): Number
#underEstimateCost(current: City, dest: City): Number
City
+name: String
+adjacent: City [ ]
+parent: City
+h: Number
+d: Number
+coords: Point
+isExcluded: Boolean
#City(name: String, adjacent: City [ ])
#setCoords(x: Number, y: Number): undefined
Search
+cities: City [ ]
+heuristic: Heuristic
+startCity: String
+destCity: String
+frontier: City [ ]
+done: City [ ]
+path: String [ ]
+status: String
+renderer: Renderer
#shortestPathStep(callback: function)
#shortestPath(callback: function)
#tracePath(startCityName: String, destCity: City, path: String [ ]):
Renderer
+snap: Snap
+cityRadius: Number
+isRendered: Boolean
#Renderer(snap: Snap)
#drawCity(c: City)
#highlightCity(c: City, fill: String, callback: function)
#drawCities(cities: City [ ])
#drawLine(c1: City, c2: City)
#animateLine(c1: City, c2: City, callback: function)
#emphasiseLine(c1: City, c2: City)
#drawPath(path: String [ ], cities: City [ ]): String
#onClickCity()
#calculateBoundaryPoints(c2: City, c1: City): (Point, Point)
#clear(callback: function)
Point
+x: Number
+y: Number
#Point(x: Number, y: Number)
#dist(p: Point): Number
Parser
+file: String
#getCities(callback: function): undefined
#addLocations(cities: City [ ], callback: function): undefined
0..n
0..n
0..n
0..n

Class Documentation

- **City**: This class defines all the necessary information for a single city.
- **Heuristic**: This class defines two functions, that allow the search engine to work. By default this is the fewest-links heuristic.
- **Parser**: This is the class that parses connections and locations files. It should produce an array of City objects which have been correctly initialized.
- **Point**: An x-y pair in a coordinate on the cartesian plane.
- **Renderer**: This class in in charge of maintaining the SVG canvas.
- **Search**: This class implements the core A* algorithm.
- **ShortestDistanceHeuristic**: This class inherits the heuristic class and redefines its two functions to be a shortest distance underestimate.

IV. IMPLEMENTATION

The project was implemented as a front-end web application. In addition to each of the classes outlined in our design, we used a number of libraries to help reduce our software engineering efforts. These are described below.

HTML5 Boilerplate

This template is a starting point for our development. It provides a large set of standard code for making front-end web applications. It handles simple normalization between browsers.

Rollup.js

Rollup is a tool used to bundle a large javascript project spanning many files into a single, compressed js file. This tool helped us manage dependencies among all our classes and allowed us to easily modularize all our code into files. This tool’s flexibility would also allow us to easily shift our algorithm to another platform, like a backend Node server for example.

Materialize CSS

This open source library provides us with a strong, clean starting point for the front-end design of our application. The CSS and Javascript framework includes sleek looking buttons and dropdowns that we used design the configuration section of the application. Figure 2 demonstrates our usage of this library.

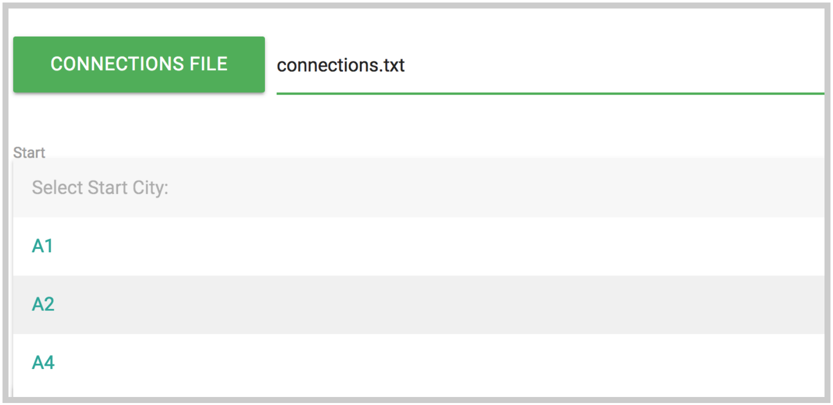


Figure 2: Screen capture of the part-1 configuration interface.

Snap SVG and JQuery

JQuery is a standard part of any front-end javascript application. We used it to efficiently work with the HTML document object model (DOM) from our javascript code. The other DOM library we used was called Snap SVG, this library allowed us to programmatically animate scalable vector graphics (SVG) on the web page. With Snap, drawing the cities and the paths between them was a “snap”.

Broccoli.js

Larger javascript projects often times have a number of steps required to build the final files. Broccoli.js is a task manager that can do exactly this. We used it to automatically run the Rollup.js operations. In the future we could setup Broccoli to automatically run tests on our different modules whenever we build so that we would be alerted to breaking changes.

V. EVALUATION

In our design we included two different heuristics, although more could be easily added in the future. With these two heuristics we get a very clear picture of the performance impact of the heuristic function on the running time of the algorithm. We first implemented the “fewest links” heuristic functions which just attempt to find the fewest number of edges between a the start and end vertices. For this heuristic under estimate it can only safely guess that from a vertex that the end vertex is one edge away. The second heuristic algorithm attempts to minimize the total distance traveled on the cartesian plane. The underestimate here is the “straight-line distance” from a vertex to the end vertex. The performance difference between the two heuristics is quite large we can see from the following two runs.

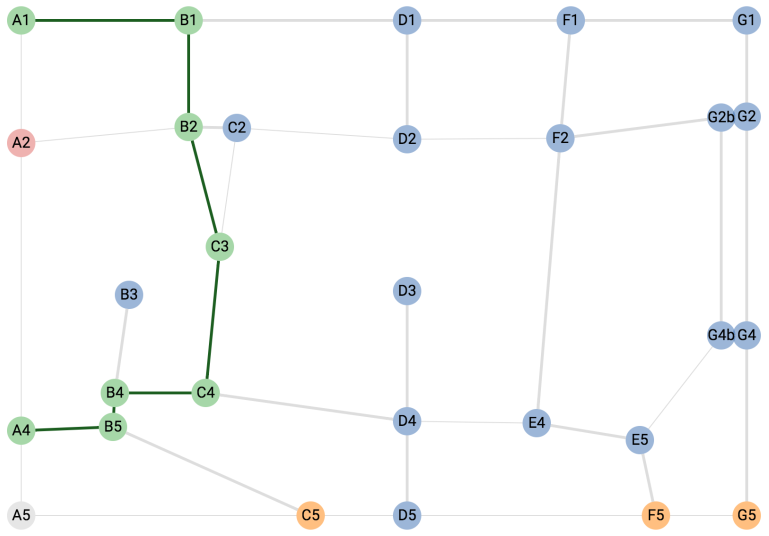


Figure 3: An execution of the A algorithm with the “fewest links” heuristic.*

The green vertices indicate the discovered shortest path. The blue vertices are in the “closed” set, the red vertex is excluded from the search, and the yellow vertices are in the “frontier” set.

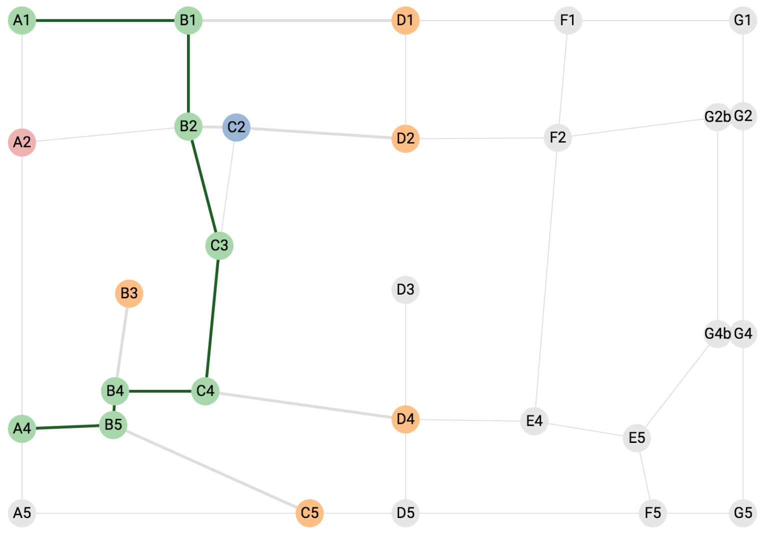


Figure 4: An execution of the A algorithm with the “straight-line path” heuristic.*

We can see that the “straight line distance” run observed only 14 vertices while the “fewest links” observed 27 vertices. This example is consistent with our expectations that the tighter the underestimate is to the true remaining distance then the greater the performance.

VI. EXPERIENCES AND CONCLUSION

Overall the project was a smooth software engineering exercise. Anwesh was in charge of writing the parser, renderer, and main code, and Richard was in charge of writing the city, heuristic, and search classes. We have been sporadically working since October 8th on this assignment. We spent about 20-40 combined hours on this project. We used Slack to coordinate our efforts.

We think this project was a fruitful learning endeavor since it required a deep understand of both A* and front-end software engineering techniques. Even though pathfinding is a fairly simple and well-studied AI problem, this was a positive experience for the both of us.