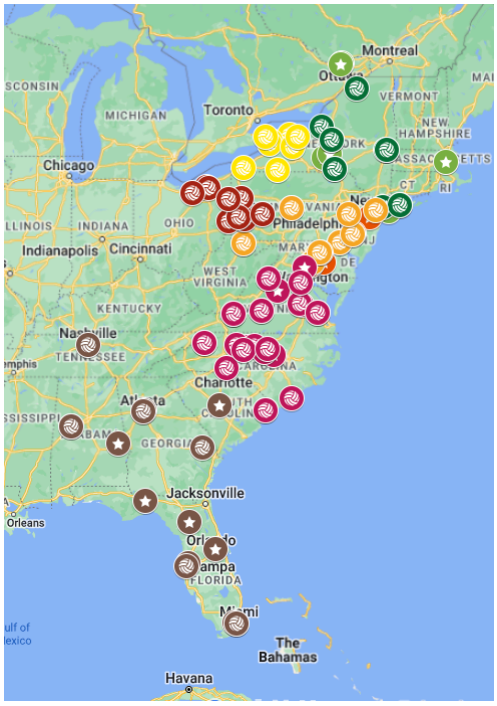
Owen Spolyar and Jon LeFrois

Academic Showcase Writeup

For the first part of our project, we looked into finding optimal host locations for tournaments for our club volleyball league, the ECVA, or Eastern Collegiate Volleyball Association. This league spreads across the entire East coast and is broken up into two divisions, Division I and Division II, as well as four regions, North, South, Central and Southeast. This even gets broken down further as some regions like the North get broken down into the Northeast and Northwest. 108 total teams make up these regions and divisions spreading across 19 states, DC, and even into Canada, and they participate in 19 sanctioned league tournaments, including one league championship which brings together the top 16 teams in each division. The picture shown to the right shows the breakdown of where all of the teams are and are color coded based on subregions. The data we had was sorted alphabetically by team name and in order to begin finding the best college in each subregion to host tournaments, we first had to sort the data according to these subregions. To do this, we made use of a comparator and an ArrayList of Team objects, which was a new class created in order to store all of the information about each team in multiple instance variables, which was done for easier information access. The comparator sorted all of



the teams based on subregion and division pairings, alphabetically, making our best host algorithm much easier to implement next.

The best host algorithm determines the best hosting school for a tournament using the logic where the best host is the one where the least cumulative distance is traveled to get to that location. This could have been also interpreted as the most convenient location for all teams, which would help any schools which have locations as outliers and are relatively far from every other school, but that was not the logic we used. We created a brute force algorithm to ensure that our host is the ideal host. This algorithm runs in $\theta(n^2)$ due to its brute force nature. The way it works is that for each team in a subregion and division pairing, it calculates the total cumulative distance that every other school would take to get there and through doing this using every team as a sample host, it finds and returns the one with the smallest total. Our data had longitude and latitude numbers for each location, so we used a distance formula getDist to convert these longitude and latitude pairs into usable distances. The pseudocode for our algorithm follows this logic:

```
bestHost(teamList){
        best, host = -1
        for(i from 0 to teamList.size){
                dist = 0
                for( j from 0 to teamList.size){
                        dist = dist + getDist(i, j)
                }
                if(dist < best || best < 0){
                        best  = dist
                        host = i
                }
        }
        return host;
}
```

Since there are two league tournaments in each subregion, we altered our code to get not only the best host, but also the second best host in order to get two different schools to be able to host. The results of our algorithm gave us this output:

Central 1: Maryland and George Washington

Central-East 2: Loyola and UMBC

Central-West 2: Youngstown State and Franciscan

North 1: Syracuse and Binghamton

North-East 2: Clarkson and Binghamton - B

North-West 2: St. Bonaventure and Buffalo - B

South 1: App State and North Carolina

South 2: App State - B and North Carolina - B

When looking at the map and realizing that even if a school looks overall central, it may not be the best host with our algorithm because of the logic we chose, the results make a lot of sense. Many of the schools chosen have many other schools in a really close proximity, which results in the algorithm favoring these schools even if there are a few that are much farther away. Had we chosen logic that would factor in the most optimal location overall for every school, there would have been different results most likely. Instead, our algorithm favors schools that are in close proximity to most of the other schools in their subregion.

A* Writeup

For the second part of our project we looked into the A* search algorithm. The A* search algorithm shares a lot of similarities with Dijsktra's algorithm which was covered in class. The main difference between the two is that A* also has a directional aspect that is added to the calculations. When choosing the next node to consider with Dijsktra's algorithm the node with the lowest cost of reaching is selected. For the A* algorithm, the node selected is the one with the lowest cost of reaching plus the estimated cost of reaching the goal from said node. Thus nodes that are closer to the end location are more likely to be selected. The way we decided to create our estimated cost was by taking the diagonal distance from the current node to the goal. This results in an underestimate of what the true cost ends up being. We utilized the Dijkstra algorithm code as a base for our calculations, our main change being in the Priority Queue list additions. The code below has the new additions to the lists:

```
pq.add(new PQEntry(e.length + crossDist, e, dest));
pq.add(new PQEntry(nextPQ.totalDist + e.length + crossDist, e, dest));
```

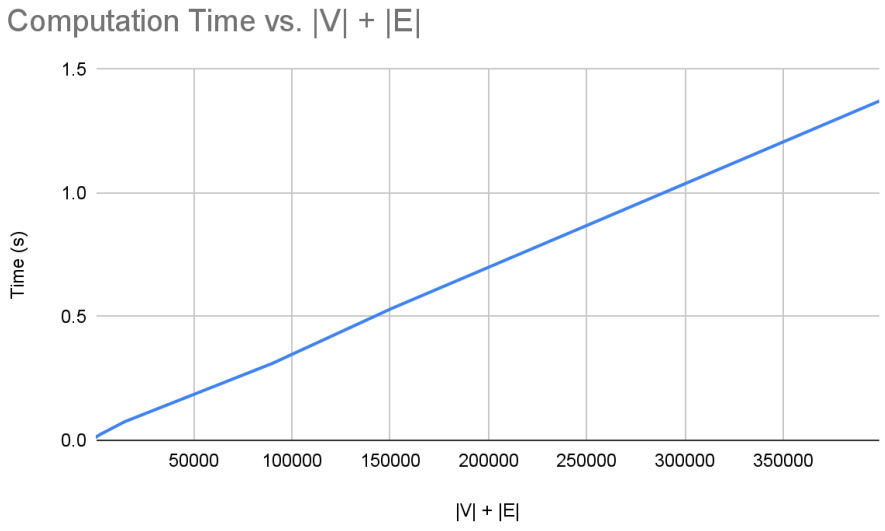
We then did an empirical analysis on the A* algorithm using the METAL data as a test set. The environment used to test these methods is a Dell Precison, with a 2.90 GHz Intel(R) Core(TM) i7-10700. There is 16 GB of RAM. It is running on Windows 10 v 22H2 and Java v 19.0.2. According to the limited literature we could find, the expected performance of our algorithm should run in $\theta(|V| + |E|)$ time, where |V| is the number of

vertices and |E| is the number of edges. We ran the A* algorithm 10 times on seven different maps, each time it had a different random start and end location that was reachable from one another.

The full results are available in the file `A_Data.xslx` in this write-up's github. Each test gave an output with six values such as

```
siena50-area.tmg 1237 1457 0.015 NY9L@CR54 I-787/NY7@8thSt
```

The first is the name of the file that is being tested by the program, followed by the number of vertices in the file, then edges. The computation time in seconds is the fourth variable. The last two are the start and end vertices respectively. Of these we were mainly interested in the correlation between the sum of the vertices and edges and the computation time.



Looking at the figure above and in depth at the numbers in the data, we can see we got a runtime consistent with O(|V| + |E|), which we expected. Keeping in mind that

Dikstras has a runtime of O(|V|^2), A* is far less time-complex. In fact, it can be shown that the worst case of A* is Dijkstra's algorithm.

$$|E| \leq |V|(|V| - 1)$$

$$|E| \leq |V|^2 - |V|$$

$$|E| + |V| \leq |V|^2.$$

In conclusion, we found the A* algorithm matched the expectations. And that A* is a linear search algorithm that can be created from a modified Dijkstra's algorithm.

Sources:

https://cs.stackexchange.com/questions/56176/a-graph-search-time-complexity

https://www.hackerearth.com/practice/algorithms/graphs/shortest-path-algorithms/tutorial