Christopher Anderson (014105585)

Dr. Markus Eger

CS 4200 Artificial Intelligence

Lab 1

Lab 1 Report

**Part I- Algorithm Implementation**

**For all:**

Created dictionary for back pathing

* Was in the form of dict[node\_id] = (came from node\_id, edge)

Created path, frontier and expandedNodes lists

* Path was used at the end alongside dictionary
* Taking end node as key, appending the edge it came from to path and applying its cost to length
* Make current key = last key tuple[0] element which was the node where it came from and repeat until starting node is current key.

Used while loop until goal was found- exception to this was A\* which also checked the rest of the frontier for better pathing.

For all but A\*, node was popped, if not in expanded list then add to expanded, extract neighbors, removed already expanded neighbors, checked if already in the frontier- if not then add to frontier and update dictionary.

A\* differed by adding an additional if, if they were already in the frontier then check that versions current cost and heuristic value to see if it should be updated with the new one.

**BFS:**

Main difference in BFS was creating a for loop for frontier length, this loop allowed all current frontier nodes to be processed in groups creating the level by level tree processing of BFS. This was done using a list/queue with .pop(0)

**DFS:**

Much like BFS but without the forloop and used a stack applying .pop(-1).

**Greedy:**

Implemented similar to DFS but used .pop(0) for queue style and pre-sorted frontier using frontier.sort(key = heuristic, reverse = False).

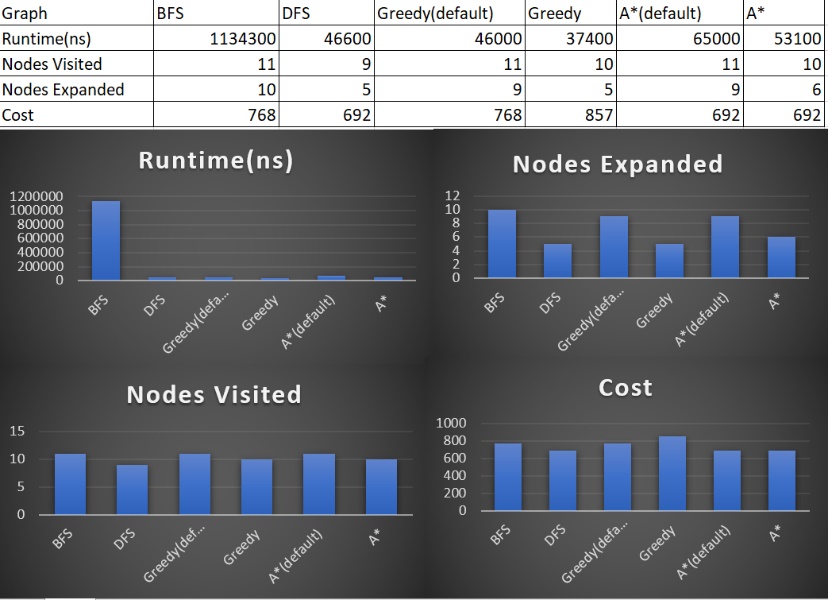
**A\*:**

Similar to greedy but created a custom sort function key frontier.sort(key = aStarKey, reverse = False).

A second dictionary was used to store the current cost to get to that node with the node\_id as its key. This would be updated when they were added or edited into the frontier.

The custom sorting function returned costDict value + heuristic value for sorting.

**Part II- Performance:**

**Austria:**

Best performances

Runtime: Greedy

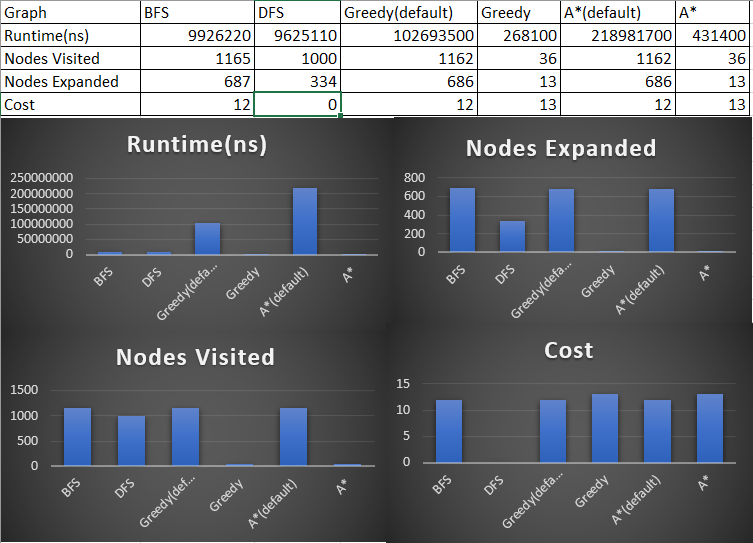
Visited: DFS

Expanded: DFS, Greedy

Cost: DFS, A\* both versions

For Austria, most algorithmns performed roughly the same in run time except for BFS which took a very long time in comparison as seen above.

Overall DFS and A\* performed the best in cost, but DFS came out slightly ahead due to getting lucky on pathing.

**Inf Graph(simple):**

Best performances

Runtime:Greedy

Visited: Greedy, A\*

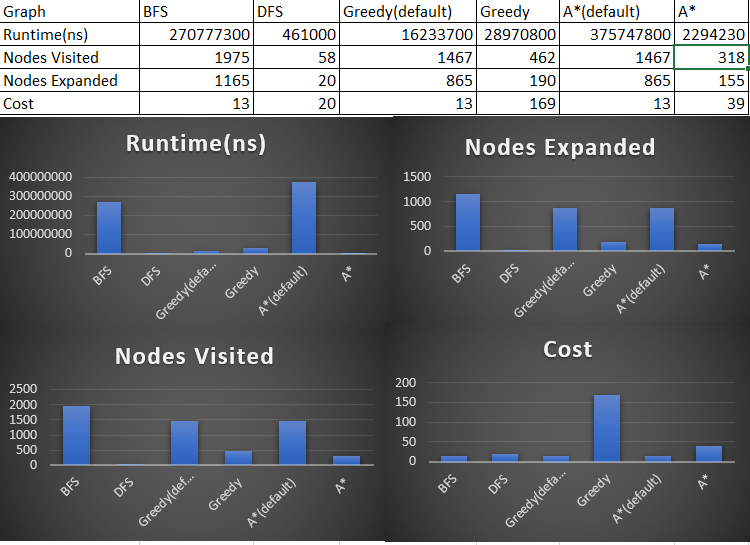
Expanded: Greedy, A\*

Cost: BFS, Greedy(default), A\*(default)

DFS failed this one.

From the data we can see using heuristics greatly reduced runtime, nodes expanded and visited.

All versions were relatively close in cost besides DFS which failed to find a path.

**Inf Graph(Multi):**

Best performances

Runtime: DFS

Visited: DFS

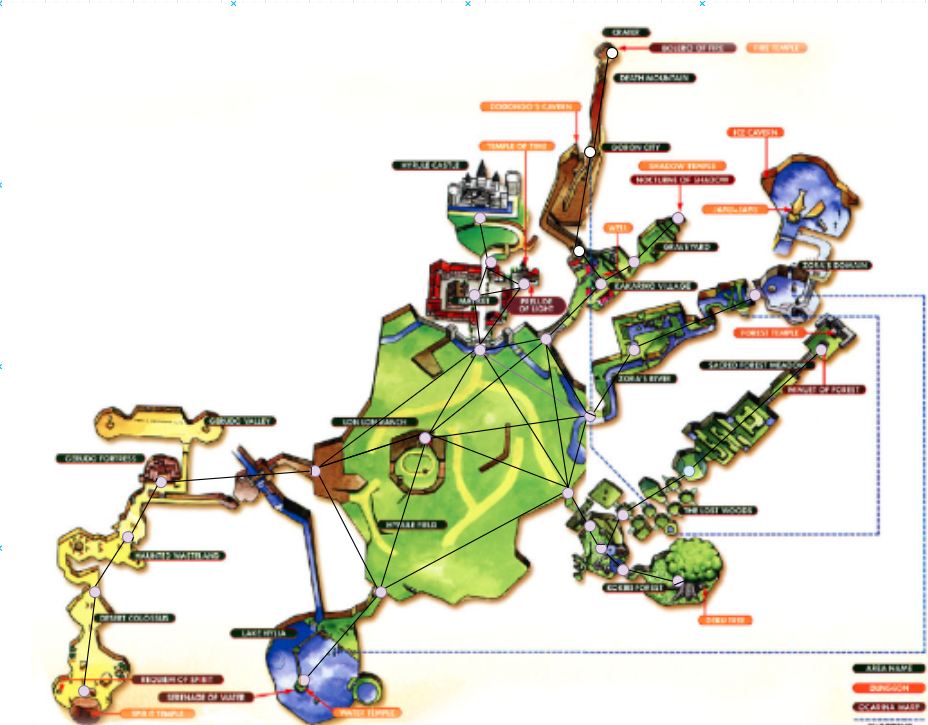
Expanded: DFS

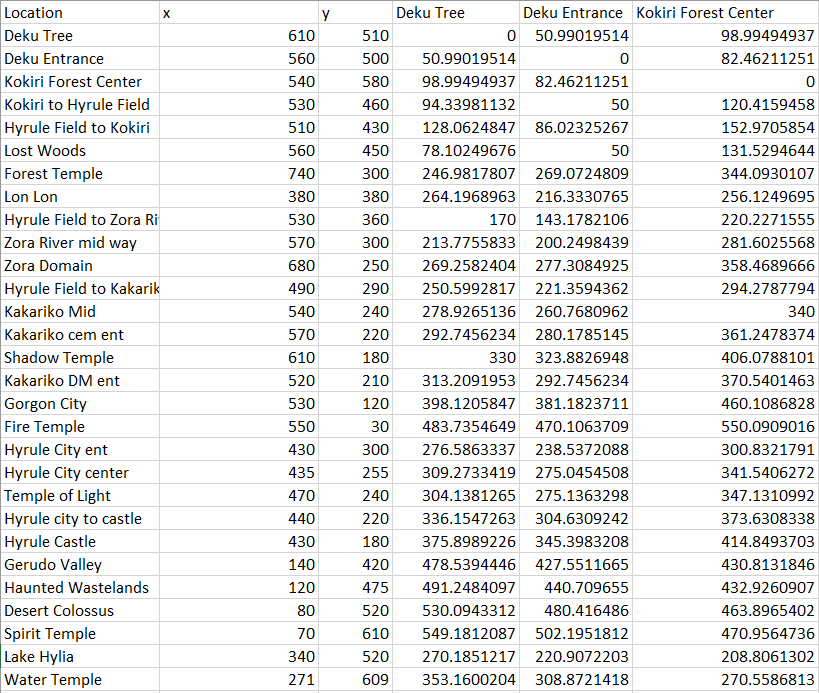
Cost: BFS, Greedy(default), A\*(default)

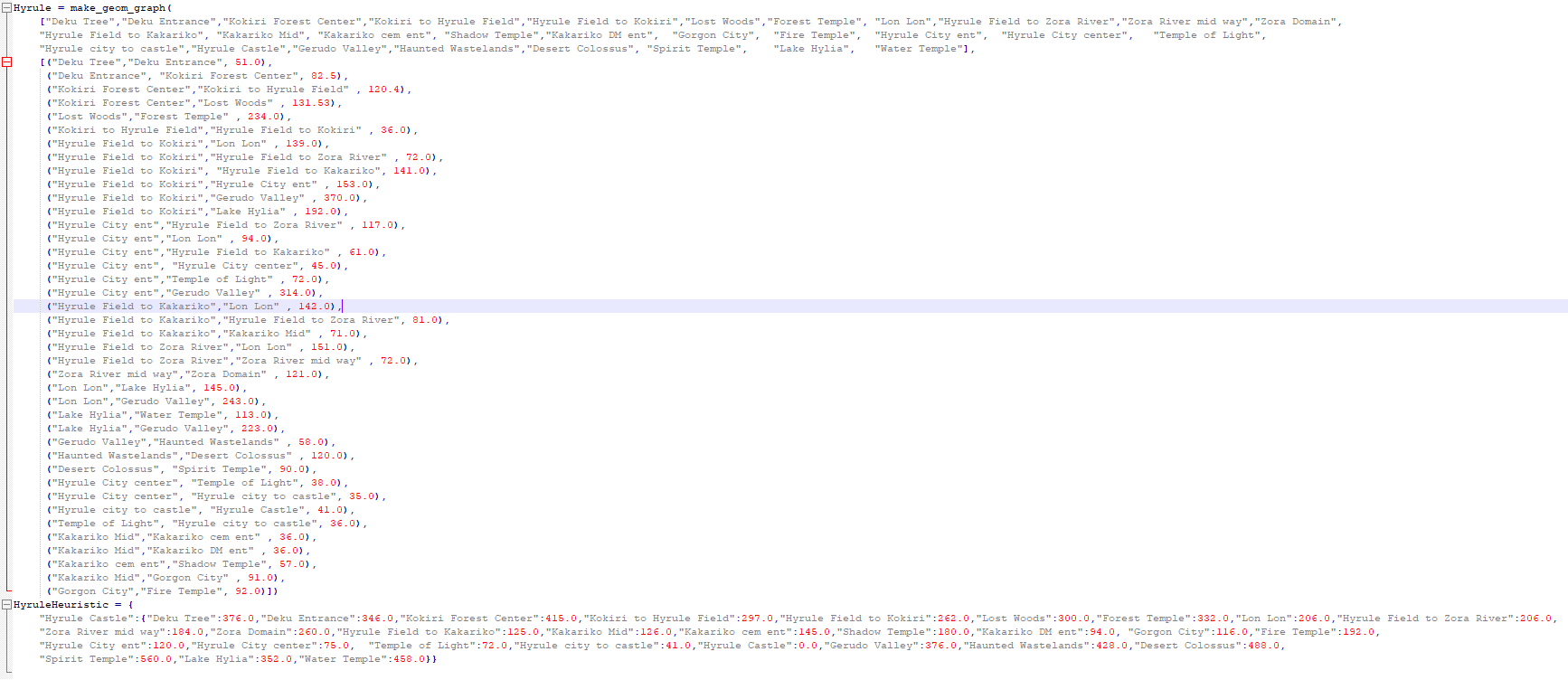
For runtime both BFS and A\* using default heuristics took exceptionally long compared to the others.

DFS, appears to be the best overall for this test, but it could of also just got lucky on pathing as it did before- needs further testing on different graphs.

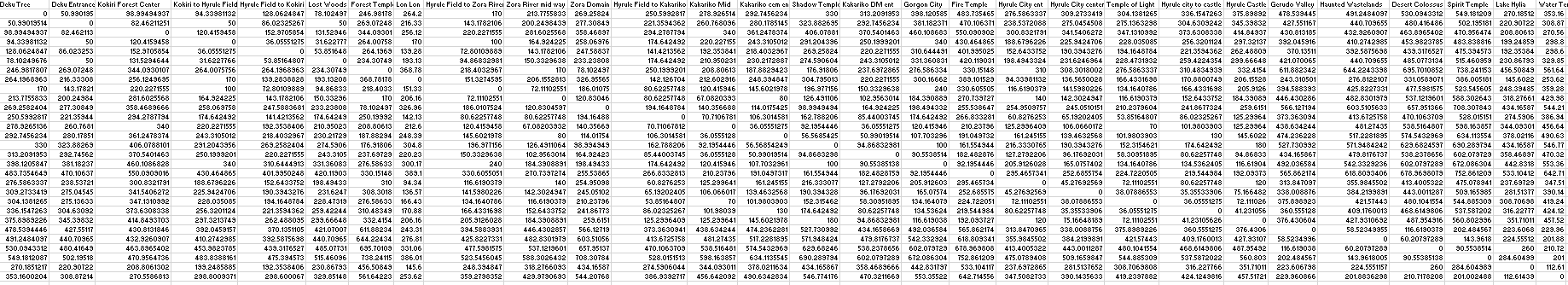
The biggest surprise in this test was how the graphs using non-default heuristics did, while they did a better job going through fewer nodes, the cost they output wasn’t nearly as good as default.

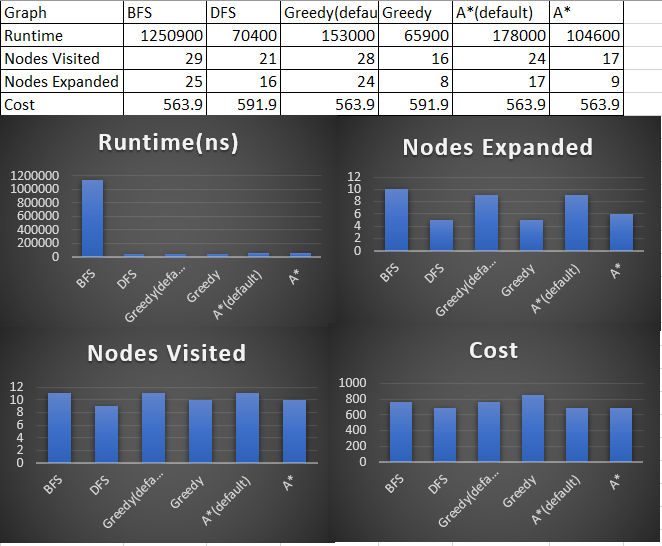
**Part III- Custom Graph – Hyrule from Legend of Zelda: Ocarina of Time**

For this graph, draw.io was used and I drew over a clipped picture I inserted. Attempting to move circles gave x, y coordinates which I input into excel.

Here is in example of the excel spread sheet, using the distance formula to find the differences between nodes to generate edge length and heuristic straight line distance.

I then built the graph using the same style used for Austria, for the Heuristic I only included one for Hyrule castle since that is what I was testing on- and the excel sheet was massive to include all 28 x 28 values.



Hyrule performance:

Best performances

Runtime: Greedy

Visited: A\*

Expanded: Greedy

Cost: BFS, Greedy(default), A\* both versions

My algorithms performed very similar to the Austria graph. This time DFS didn’t get as lucky but BFS ended up tying in total cost with the heuristics but at the cost of a much greater runtime. All algorithmns actually performed decent with roughly 30 units difference in their cost.

The main thing I was testing in this graph compared to the Austria graph was the many dead ends available.