Project 2: Search on Directed graph

Richard Douglas
M.S. Computer Science
Speed School of Engineering
University of Louisville, USA
rcdoug@louisville.edu

1. Introduction

1.1 Goals:

As a first step, we want to implement various solutions to calculating the best path to a target node in a directed graph. This is in essence, the TSP problem. We want to keep track of the runtimes and best path cost for various algorithms to compare them later. From this data we can observe the runtime efficiency of various algorithms for an NP-Complete problem.

1.2 Definitions:

Directed graph - edges in a directed graph go "one way". In other words, an edge from one node to another does not travel in the opposite direction. For the nodes n0 and n1 with an edge (n0, n1), the edge will take you from n0 to n1, but not from n1 to n0. That is not to say there is no possible edge that will take you from n1 to n0, it is simply to say that an edge that does this is denoted as (n1, n0). These two edges would create a loop of sorts, which would make these algorithms slightly more difficult to implement, but certainly not by much. the graph we are working with contains no such loops. In the implementation of the function to generate pseudo random edges, I made efforts to avoid such loops. The results in this report do not include findings from random graphs, but it was fun to puzzle over.

2. Approach

2.1 Concept

2.2 Implementation

2.2.1 Python - I continued with the code from the previous project, modifying it to achieve the goals of this project.

2.2.2 Reusability - I refactored the code, separating the search algorithms into a library containing algorithms to find full traversals of hamiltonian circuits and (the class for this project) finding target nodes in a directed graph.

2.2.2.1 TSP_parser.py - this file is much more slim now, although I made some modifications.

Flags: some new flags were created, most of which currently have default values. Namely, there is a flag that dictates whether the edges in the graph are static (from a dictionary) or are to be semi-randomly assigned.

Node dictionary: this is still populated with values from a tsp file. To ease coding efforts in the rest of the project, the names of the nodes are now integer values one less than the name of the node in the tsp file (for indexing in other functions.

Cycle cost list: dictionaries to store the information computed by the various algorithms upon running them.

2.2.2.2 traversal.py - this contains two classes: FullTraversal and Directed. I will only talk about the functions in the Directed class.

Bfs: this algorithm simply traverses all children of a "head node", searching for the target. After all of the child nodes are searched, one of the children is set to the new head node. this node's children are then searched. This is done repeatedly until the target node is found or all nodes have been exhausted. The best path in terms of both cost and amount of hops is tracked, and so is the amount of hops and cost.

Dfs: the depth first search algorithm traverses a graph until it reaches a node with no children. This implementation is recursive, and the recursive calls search deeper into the graph. Flagging is utilized to set nodes as visited. If they are visited, they are not eligible to traverse to. For back tracking, the start node visited flag is set to false at the end of a recursive loop. In this way, we can travers all terminating paths of a directed graph. along the way, a variable, whose memory address is global to all recursive calls, tracks the current best cost. It is initialized to a very large float and updated when a path cost is lower. this event also triggers an updating of a similarly "global" variable, that stored the path of that cost. Variable of the same nature track the best path in terms of least hops.

Greedy: this is the same code as dfs, however, the costs are sorted, and the lowest cost is selected. Recursion breaks when the target is found. This results in a faster algorithm than dfs, but can't guarantee the best path is found.

2.2.2.3 Plotting.py - I won't go into excruciating details but I used matplotlib, pandas, and seaborn to visualize the data collected in the logs.

2.2.2.4 tsp_driver.py - again, I will refrain from providing overwhelming detail here but the code is available for reference. the major modification to this script is the fact that it uses different path finding algorithms. It is also a little different in that is instantiates a parser object and passes that into the functions that implement the path finding algorithms

2.2.2.5 gui.py - this is the code for animating graph traversal. A parser object is passed in, to instantiate the gui in tkinter. Then that gui object is passed into the various algorithms, which render the traversals as they run. Honestly, its pretty janky, but it works.

3. **Results** (How well did the algorithm perform?)

In a general sense, I achieved the goal of this project. I implemented bfs and dfs as search algorithms in our search space. I also went a little further and implemented a greedy heuristic solution and a gui. I was working on an a* implementation and an IDA* implementation, but - alas - I ran out of time and extra motivation to devote to that.

3.1 Data (Describe the data you used.) :

3.1.1 TSP Files: I used the .tsp files provided in the .zip file Professor Yampolskiy provided. The format is shown below:

3.1.2 Logging results: the visual representation of my results were created with the logs I saved in .csv format from the results of running the algorithms I wrote on the .tsp files. The following are written to the logs in each row. you can check out the results in the csv file:

3.1.3 algorithm,input_size,best_cost,best_cost_path,least_hops,least_hops_path,runtime,tsp_file

3.1.3.1 Algorithm: which algorithm I used

3.1.3.2 Input size: on the iteration for

3.1.3.3 best_cost: the best cost calculated by the algorithm implemented

3.1.3.4 best_cost_path: path with the best cost to reach the target

3.1.3.5 least_hops: least amount of hops to reach the target

3.1.3.6 least_hops_path: path corresponding to least hops

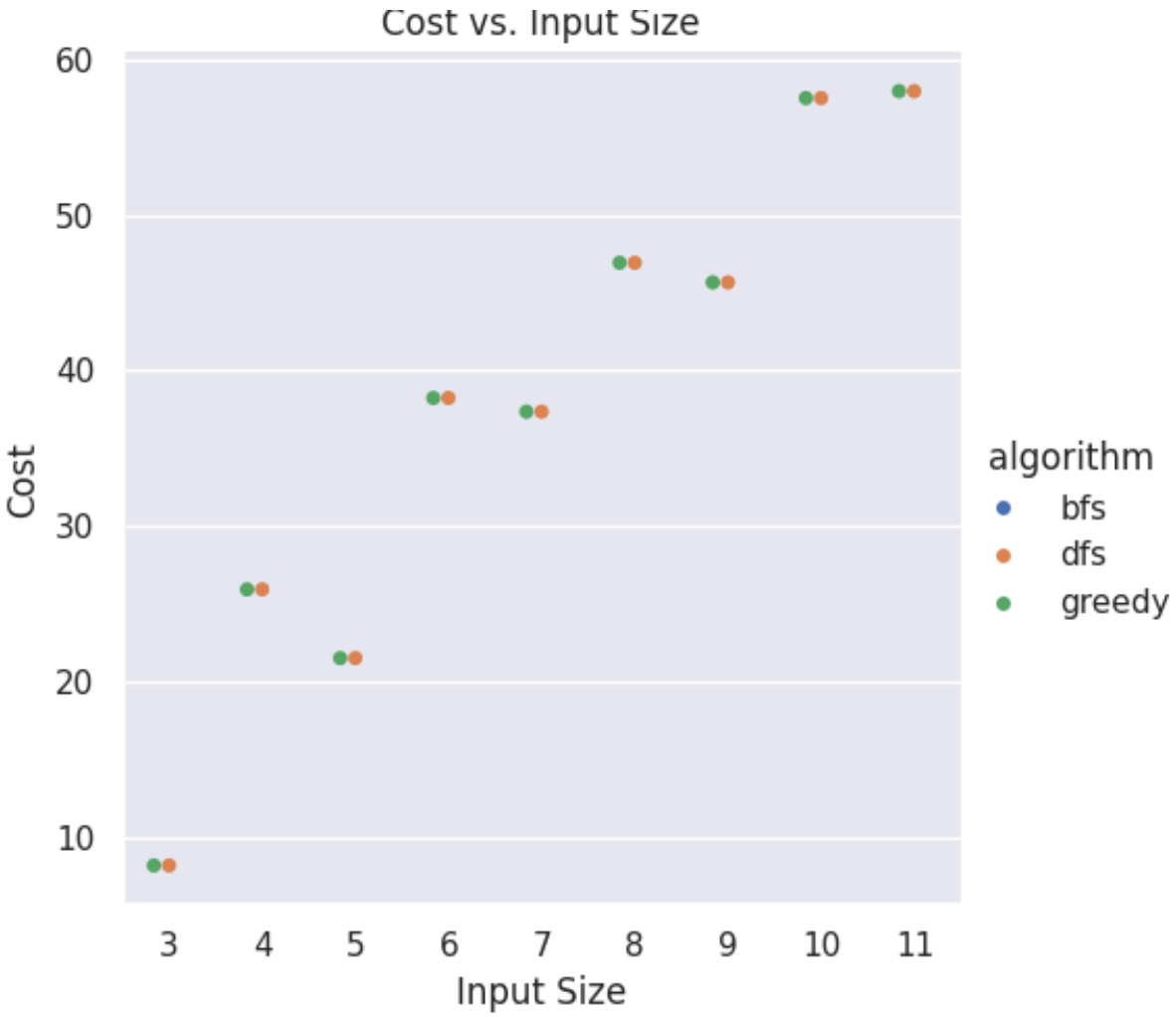
3.1.3.7 Runtime: The measured runtime calculated by the difference between a timestamp taken immediately before the algorithm starts, and one taken immediately after it completes (end - start).

3.1.3.8 File: the TSP file used in making the calculation. Although it isn't germain to this report, it was useful for debugging.

3.2 Results (Numerical results and any figures or tables.) :

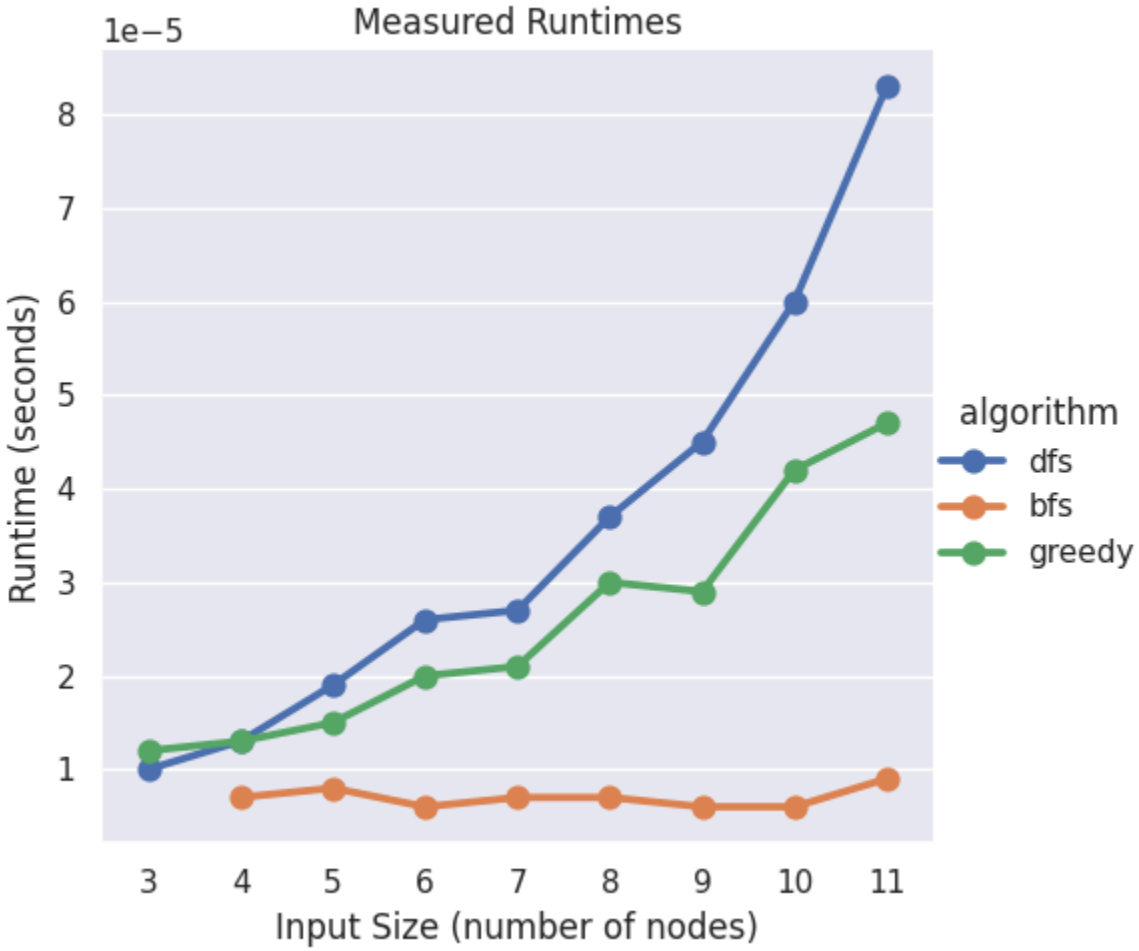
3.2.1 Figure 1: Calculated best path results

This is a categorical box plot of the best costs calculated. there is nothing very special to note in these results. Do notice, however, that the results are the same for all of the algorithms. It is ENTIRELY possible for the greedy algorithm to return results that are not the absolute optimal solution. Depending on your implementations of the other algorithms, they too may not necessarily find the best path in terms of cost, although they are very likely to find the best path in terms of hops. In fact, the BFS algorithm necessarily returns the path with the least hops to a target.

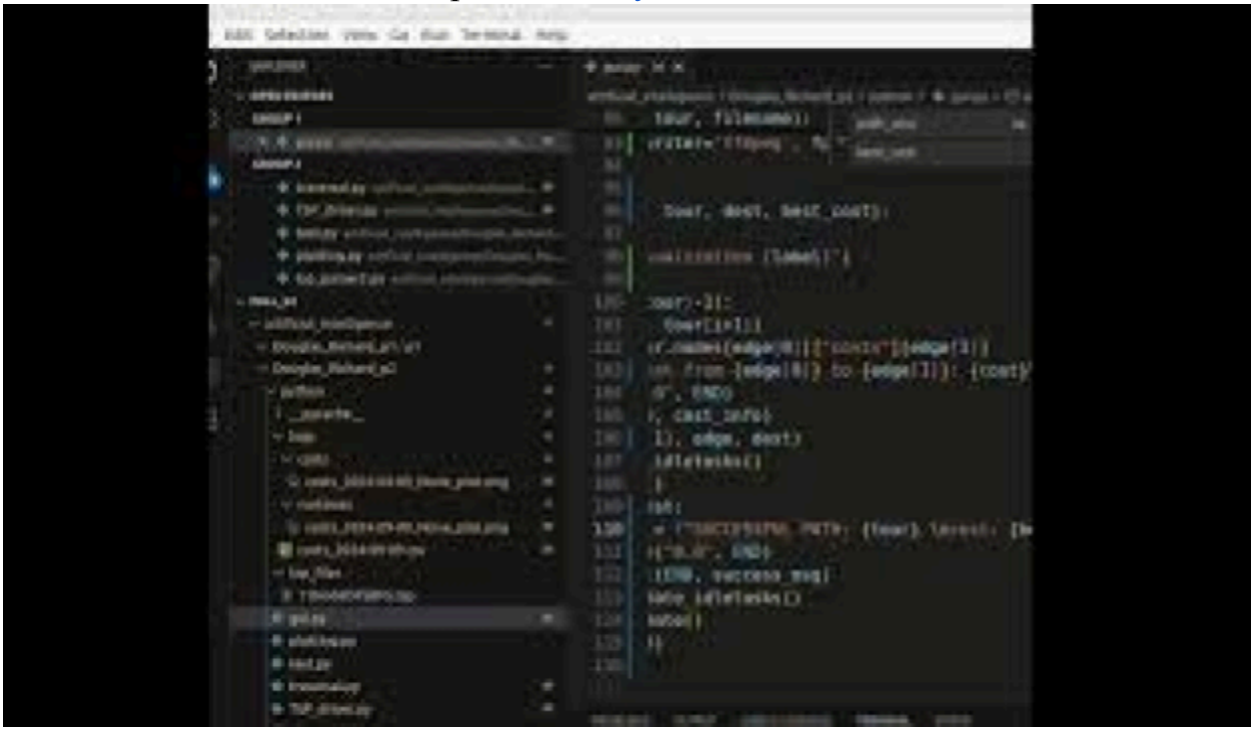


3.2.2 Figure 2: Runtime results

Here we can see that bfs runs the most efficiently. I believe its speed advantage comes from two facts: it inherently has a lower time complexity because it simply expands all of the nodes, searching for the target, and because the other two are implemented recursively. Thus, the recursive calls must be made and return for the recursive tree to return. Finally, we can observe a higher rate of increase in time with input size in bfs and the greedy algorithm. This is no surprise considering the early termination condition of the greedy algorithm - thus, it makes no more recursive calls to search deeper in the graph when the target is found. also note that the first result for the bfs does not appear in this graph. There is just a bug in the logging function I think (I'm noticing this while writing). I'll get around to fixing it, but must submit to time. Besides, the trend is clear regardless of the inclusion of that data point.



Video of visualization:
Not sure if this will play for you. It's a last second, janky workaround for google docs. In case this doesn't work, I posted it on youtube.



4. Discussion

In the future I would like to make an effort to improve my gui. it's fine right now but I have not implemented the functionality to save the animation, which would be useful for imbedding, in lieu of taking a screen capture… I also am a fan of the A* heuristic. it seems so obvious once you know about it, but I didn't quite get around to cramming it into this code base.

5. References

on directed graphs:
https://www.whitman.edu/mathematics/cgt_online/book/section05.11.html#:~:text=A%20directed%20graph%2C%20also%20called,or%20(w%2Cv)

algorithms:
https://www.geeksforgeeks.org/depth-first-search-or-dfs-for-a-graph/
https://www.geeksforgeeks.org/breadth-first-search-or-bfs-for-a-graph/
https://www.geeksforgeeks.org/a-search-algorithm/

python documentation:
https://seaborn.pydata.org/generated/seaborn.catplot.html
https://docs.python.org/3/library/sys.html
https://docs.python.org/3/library/os.html
https://docs.python.org/3/library/datetime.html

https://docs.python.org/3/library/gc.html
https://docs.python.org/3/library/csv.html
https://matplotlib.org/stable/index.html
https://pandas.pydata.org/docs/index.html
https://docs.python.org/3/library/math.html
https://docs.python.org/3/library/itertools.html#itertools.permutations
https://docs.python.org/3/library/random.html