**CECS 545 Project 2 report**

**Ryan Schron**

**September 9, 2019**

**Introduction**

In Project 1 we implemented a brute-force solution to find the least cost Hamiltonian Cycle in a fully connected, undirected graph. This NP-Hard problem is typically solved in a manner that is optimized for different performance metrics such as run time, or accuracy. In this project, we solve a TSP variant in which the start and end state is fixed, and the graph is directed and no longer fully connected. Two approaches are taken in this implementation.

The first approach is a Breadth-First Search (BFS), which examines the search tree in a wide manner, moving level by level throughout the tree. The second approach is a Depth-First Search (DFS), which attempts to follow a path until the goal is reached or the search must back-track. These methodologies will find the best cost path in the given graph and will be compared for their cost in solving the problem.

**Running the Program**

The program can be run by navigating to the appropriate directory and using the following command in a terminal:

python bfsdfs.py

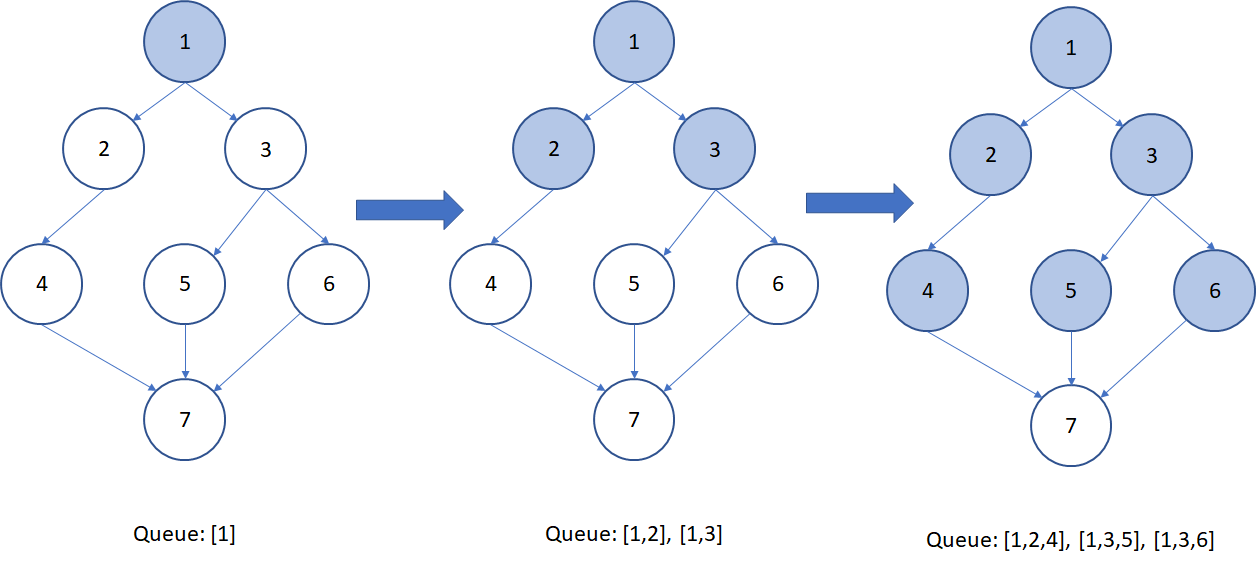
No arguments are required as the graph we are solving is hardcoded in this project. The code should output to the terminal the best path, best found distance, and run time in seconds of both the DFS and BFS solutions to the problem. Transition count is commented out, as it was found to skew run-time.

**Code Description**

To start, an adjacency matrix is created as a two-dimensional numpy array as a representation of the directional graph. This matrix is hard-coded and organized such that adj\_matrix[i][j] represents the path between node i and node j. Each entry contains the distance in the direction i->j or 0 if there exists no path in this direction. All values are stored as 64-bit floating point numbers.

*Breadth-First Search*

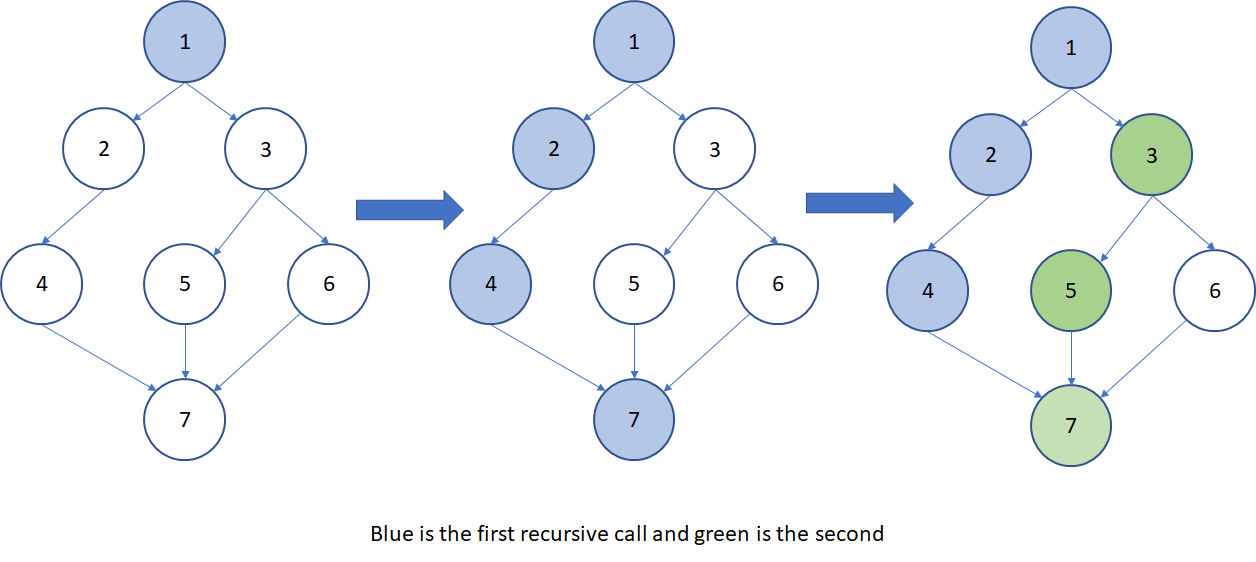
The BFS implementation begins by defining a queue that contains a list with the starting node as a single element. As it searches the graph, it will pop from the queue the first path, and grab the last node within. It will then examine each accessible neighbor from this node and append it to the current path, copying the path for each possible addition. These new paths are then added to the queue for further iterations. This method will check every possible path in a level-by-level manner in the search tree. The following figure illustrates the algorithm’s function:



For each path added, the function checks if it contains the stop node, and if so, checks the path length. If the path length is shorter than the best-found path thus far, both the path length and the path itself are saved for further comparisons.

*Depth-First Search*

The DFS implementation uses a recursive exploration of potential paths. Function DFS() maintains a collection of paths found and an inner function, DFS\_recurse() that handles the exploration of the graph. This inner function identifies the last node of a path passed to it and checks for neighbors. As each neighbor is found (in list order) the function calls itself recursively to discover new neighbors until the end of the path is found. This stepwise action is demonstrated in the following figure:



After discovering all paths, the outer function then examines the collection looking for paths that ends at the desired stop node. These paths are then measured for path length, and the best candidates are stored for further comparisons. Once the best path is found, the function prints the results.

**Code Performance**

Firstly, both approaches were able to identify the correct path of [1, 3, 5, 7, 9, 11] as the best path in this experiment, though the DFS approach may not always guarantee completely accurate solutions. Unfortunately, the DFS approach was slower finding the optimal solution in 0.02399 seconds compared to 0.00099 seconds for the BFS approach, and both solutions involved the same number of transitions at 102. These results are summarized in the table below:

|  |  |  |
| --- | --- | --- |
|  | Breadth-First Search | Depth-First Search |
| Time | 0.00099s | 0.02399s |
| Transitions | 102 | 102 |
| Best Path | [1, 3, 5, 7, 9, 11] | [1, 3, 5, 7, 9, 11] |
| Best Distance | 57.967 | 57.967 |
|  |  |  |

**Conclusions**

Both approaches were able to provide an accurate best path solution to the given directional graph in this project, but the BFS implementation was able to run faster. This result is likely skewed, however, as both approaches took 102 transitions to exhaust all possibilities within the search tree. More importantly, implementing the transition count caused the approaches to begin to converge on running time, indicating that house-keeping operations are skewing the results, and may account for the difference in run-time. As such, a direct comparison of techniques is likely inconclusive on the machine used for testing a graph of this size. Going forward, improvements could be made in memory management by diligently removing any found paths that do not include the goal node and are no longer needed for path generation.