AI Project 2: Depth First Search and Breadth First Search of a Finite Tree

Robert Max Williams
CECS
Speed School of Engineering
University of Louisville, USA
r0will21@louisville.edu

1. Introduction

Depth first search and breadth first search are both algorithms for exploring a tree. The tree could be explicit, like a directory, or implicit, like the tree of possible routes though a city. For this report we are exploring a route between cities in 2d Euclidean space, with only a subset of path being allowed. The search space is small enough that it can be searched exhaustively with both approaches, and complete listings of the searches are in the Appendix.

2. Approach

2.1 Depth First Search

Depth first search is naturally implemented as a recursive function, which calls itself with each possible node that can be reached from the current node. Pseudo code for DFS:

```
def DFS (current_node, next_nodes_dict, path, cost):
    if current_node == FINAL_NODE:
        print(path, cost);
    for next_node, edge_cost in next_nodes_dict[current_node]:
        DFS(next_node, next_nodes_dict, path + [current_node], cost + edge_cost)
```

DFS is applicable for exhaustive search, or when a permissible solution is fairly likely within the deepest nodes. For very large search trees, it tends to get stuck. I had this problem with my Bananagrams solver which would get stuck in dead ends. Random restarts are one solution to this problem, but can slow down convergence if there is no mechanism to avoid parts of the search space that have already been explored.

2.2 Breadth First Search

My implementation of breadth first search uses a function `expand-nodes` which takes in a list of (path, cost) pairs and "expands" each one to a list of its successor (path, cost) pairs. Pseudo code for `expand-nodes`:

```
def expand_nodes (path_cost_pairs, next_nodes_dict):
    for path, cost in path_cost_pairs:
        for next_node, edge_cost in next_nodes_dict[path[0]]
            collect ([next_node] + path, cost + edge_cost)
```

This function is called repeatedly on a list of nodes, starting with a list containing only the root node with a cost of zero. Each call to expand nodes gives all paths at one level deeper. So

```
[(root, 0)]
```

contains all nodes at depth zero (just the root), and

```
expand_nodes([(root, 0)])
```

contains all nodes reachable with depth 1, and

```
expands_nodes(expand_nodes([(root, 0)]))
```

Contains all nodes reachable with depth 2. This is done until an acceptable solution is found, or until the empty list is returned if an optimal solution is needed.

Breadth first search is useful because shallower solutions are explored first. The leaves of the tree, where DFS spends most of its time, have far more nodes than the shallower regions, so if the depth is a reasonable heuristic for a good solution then BFS will be very efficient. It is also able to find the shallowest solution first, which can be useful to get a permissible solution.

3. Results

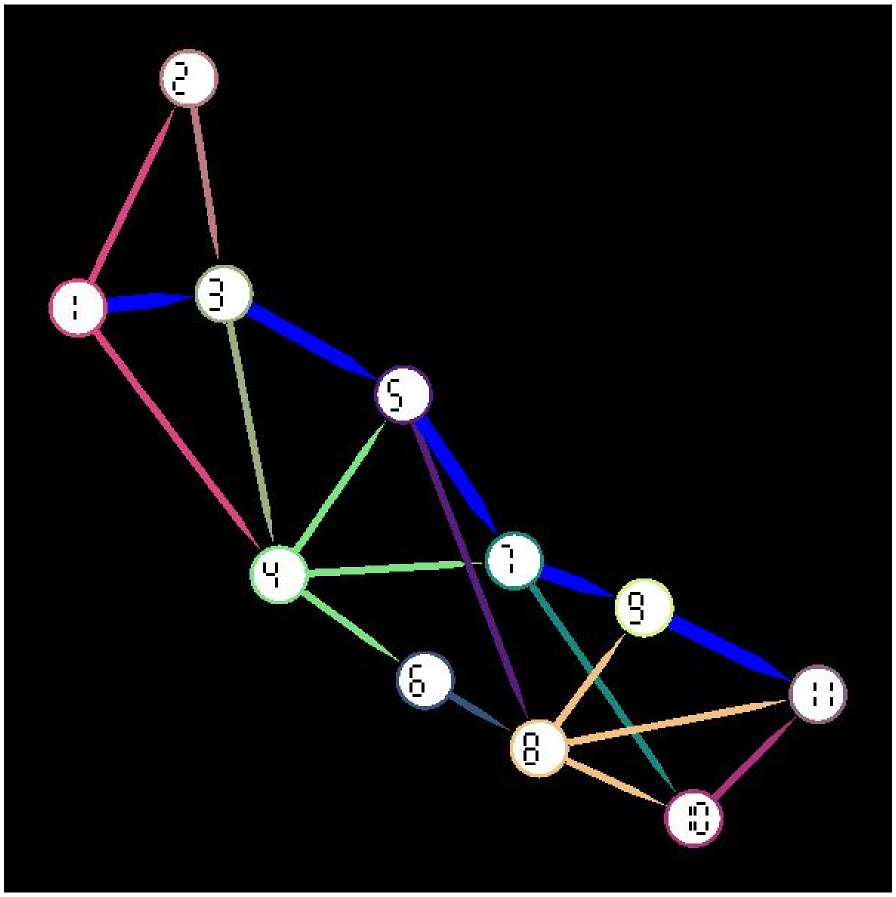

Figure 1: Plot showing all legal paths (outgoing path from each node colored to match that node) and solution path (thick, in blue).

3.1 Data

The provided TSP files was used as input, and the provided connectivity matrix was converted into an associative list which was hard coded into the programs.

3.2 Result

Depth first search's process can be visualized as a tree. If the return from each nested function call is collected, a tree showing all possible paths and the cost at each path terminal is created:

```
(1
 (2
  (3
   (4
    (5 (7 (9 (11 118.55193)) (10 (11 135.53412)))
     (8 (9 (11 139.95969)) (10 (11 137.50194)) (11 130.24113)))
    (6 (8 (9 (11 111.78762)) (10 (11 109.32987)) (11 102.06906)))
    (7 (9 (11 98.596756)) (10 (11 115.57894))))
   (5 (7 (9 (11 89.79618)) (10 (11 106.778366)))
    (8 (9 (11 111.20394)) (10 (11 108.74619)) (11 101.48538)))))
 (3
  (4
   (5 (7 (9 (11 86.722916)) (10 (11 103.7051)))
     ...))))
```

See Appendix 1 for a diagram of the entire tree with the solution path.

Breadth first search gives all paths of a given length, which can be printed as such:

```
Step: 0
 Paths:
  (0 (1))

Step: 1
 Paths:
  (25.950586 (4 1))
  (8.209853 (3 1))
  (21.047512 (2 1))

Step: 2
 Paths:
  (39.076714 (7 4 1))
  (38.249607 (6 4 1))
  ...
...
```

See Appendix 2 for a complete listing of DFS's node expansion process.

4. Discussion

For this problem, both BFS and DFS find a solution in a trivial amount of time. Both find the solution after exploring about 50% of the leaves of the search space, so this problem does not heavily favor one or the other for exhaustive search. Uniform cost search could search much more efficiently, and could be important for more densely connected graphs.

Appendix 1

ASCII art depicting the depth first search tree. '=>' indicates that the goal node, 11, has been reached and is followed by the distance. The optimal path is highlighted in yellow.

```
1
├2
│ └3
│  ├4
│  │ ├5
│  │ │ ├7
│  │ │ │ ├9
│  │ │ │ │ └11=> 118.55193
│  │ │ │ └10
│  │ │ │  └11=> 135.53412
│  │ │ └8
│  │ │  ├9
│  │ │  │ └11=> 139.95969
│  │ │  ├10
│  │ │  │ └11=> 137.50194
│  │ │  └11=> 130.24113
│  │ ├6
│  │ │ └8
│  │ │  ├9
│  │ │  │ └11=> 111.78762
│  │ │  ├10
│  │ │  │ └11=> 109.32987
│  │ │  └11=> 102.06906
│  │ └7
│  │  ├9
│  │  │ └11=> 98.596756
│  │  └10
│  │   └11=> 115.57894
│  └5
│   ├7
│   │ ├9
│   │ │ └11=> 89.79618
│   │ └10
│   │  └11=> 106.778366
│   └8
│    ├9
│    │ └11=> 111.20394
│    ├10
│    │ └11=> 108.74619
│    └11=> 101.48538
├3
│ ├4
│ │ ├5
│ │ │ ├7
│ │ │ │ ├9
│ │ │ │ │ └11=> 86.722916
│ │ │ │ └10
│ │ │ │  └11=> 103.7051
│ │ │ └8
│ │ │  ├9
│ │ │  │ └11=> 108.13068
│ │ │  ├10
│ │ │  │ └11=> 105.67293
│ │ │  └11=> 98.412125
│ │ ├6
│ │ │ └8
│ │ │  ├9
│ │ │  │ └11=> 79.95861
│ │ │  ├10
│ │ │  │ └11=> 77.50086
│ │ │  └11=> 70.24006
│ │ └7
│ │  ├9
│ │  │ └11=> 66.76774
│ │  └10
│ │   └11=> 83.74992
│ └5
│  ├7
│  │ ├9
│  │ │ └11=> 57.967163
│  │ └10
│  │  └11=> 74.94935
│  └8
│   ├9
│   │ └11=> 79.37493
│   ├10
│   │ └11=> 76.91718
│   └11=> 69.65637
└4
 ├5
 │ ├7
 │ │ ├9
 │ │ │ └11=> 79.63069
 │ │ └10
 │ │   └11=> 96.61288
 │ └8
 │   ├9
 │   │ └11=> 101.03846
 │   ├10
 │   │ └11=> 98.58071
 │   └11=> 91.3199
 ├6
 │ └8
 │   ├9
 │   │ └11=> 72.866394
 │   ├10
 │   │ └11=> 70.408646
 │   └11=> 63.14784
 └7
   ├9
   │ └11=> 59.67552
   └10
     └11=> 76.65771
```

Appendix 2

Successive expansions of the root using BFS. Paths are sorted first by nearness to the goal state, then by lowest cost first. Path to optimal solution is highlighted in yellow, and all terminal paths are highlighted in light orange.

```
Step: 0
 Paths:
  (0 (1))


Step: 1
 Paths:
  (25.950586 (4 1))
  (8.209853 (3 1))
  (21.047512 (2 1))


Step: 2
 Paths:
  (39.076714 (7 4 1))
  (38.249607 (6 4 1))
  (21.537903 (5 3 1))
  (43.20143 (5 4 1))
  (33.042805 (4 3 1))
  (40.038864 (3 2 1))


Step: 3
 Paths:
  (63.772064 (10 7 4 1))
  (47.386475 (9 7 4 1))
  (46.941193 (8 6 4 1))
  (53.44973 (8 5 3 1))
  (75.11326 (8 5 4 1))
  (37.36836 (7 5 3 1))
  (46.168934 (7 4 3 1))
  (59.031887 (7 5 4 1))
  (45.341827 (6 4 3 1))
  (50.293648 (5 4 3 1))
  (53.366913 (5 3 2 1))
  (64.87182 (4 3 2 1))


Step: 4
 Paths:
  (59.67552 (11 9 7 4 1))
  (63.14784 (11 8 6 4 1))
  (69.65637 (11 8 5 3 1))
  (76.65771 (11 10 7 4 1))
  (91.3199 (11 8 5 4 1))
  (57.523003 (10 8 6 4 1))
  (62.063705 (10 7 5 3 1))
  (64.03154 (10 8 5 3 1))
  (70.86428 (10 7 4 3 1))
  (83.727234 (10 7 5 4 1))
  (85.69507 (10 8 5 4 1))
  (45.67812 (9 7 5 3 1))
  (54.478695 (9 7 4 3 1))
  (60.577347 (9 8 6 4 1))
  (67.085884 (9 8 5 3 1))
  (67.341644 (9 7 5 4 1))
  (88.74941 (9 8 5 4 1))
  (54.033413 (8 6 4 3 1))
  (82.205475 (8 5 4 3 1))
  (85.27874 (8 5 3 2 1))
  (66.12411 (7 5 4 3 1))
  (69.19737 (7 5 3 2 1))
  (77.99795 (7 4 3 2 1))
  (77.17084 (6 4 3 2 1))
  (82.122665 (5 4 3 2 1))


Step: 5
 Paths:
  (57.967163 (11 9 7 5 3 1))
  (66.76774 (11 9 7 4 3 1))
  (70.24006 (11 8 6 4 3 1))
  (70.408646 (11 10 8 6 4 1))
  (72.866394 (11 9 8 6 4 1))
  (74.94935 (11 10 7 5 3 1))
  (76.91718 (11 10 8 5 3 1))
  (79.37493 (11 9 8 5 3 1))
  (79.63069 (11 9 7 5 4 1))
  (83.74992 (11 10 7 4 3 1))
  (96.61288 (11 10 7 5 4 1))
  (98.412125 (11 8 5 4 3 1))
  (98.58071 (11 10 8 5 4 1))
  (101.03846 (11 9 8 5 4 1))
  (101.48538 (11 8 5 3 2 1))
  (64.61522 (10 8 6 4 3 1))
  (90.81946 (10 7 5 4 3 1))
  (92.787285 (10 8 5 4 3 1))
  (93.89272 (10 7 5 3 2 1))
  (95.86055 (10 8 5 3 2 1))
  (102.6933 (10 7 4 3 2 1))
  (67.66956 (9 8 6 4 3 1))
  (74.43387 (9 7 5 4 3 1))
  (77.50713 (9 7 5 3 2 1))
  (86.30771 (9 7 4 3 2 1))
  (95.84163 (9 8 5 4 3 1))
  (98.914894 (9 8 5 3 2 1))
  (85.86242 (8 6 4 3 2 1))
  (114.03449 (8 5 4 3 2 1))
  (97.953125 (7 5 4 3 2 1))


Step: 6
 Paths:
  (77.50086 (11 10 8 6 4 3 1))
  (79.95861 (11 9 8 6 4 3 1))
  (86.722916 (11 9 7 5 4 3 1))
  (89.79618 (11 9 7 5 3 2 1))
  (98.596756 (11 9 7 4 3 2 1))
  (102.06906 (11 8 6 4 3 2 1))
  (103.7051 (11 10 7 5 4 3 1))
  (105.67293 (11 10 8 5 4 3 1))
  (106.778366 (11 10 7 5 3 2 1))
  (108.13068 (11 9 8 5 4 3 1))
  (108.74619 (11 10 8 5 3 2 1))
  (111.20394 (11 9 8 5 3 2 1))
  (115.57894 (11 10 7 4 3 2 1))
  (130.24113 (11 8 5 4 3 2 1))
  (96.44423 (10 8 6 4 3 2 1))
  (122.648476 (10 7 5 4 3 2 1))
  (124.6163 (10 8 5 4 3 2 1))
  (99.49857 (9 8 6 4 3 2 1))
  (106.262886 (9 7 5 4 3 2 1))
  (127.67065 (9 8 5 4 3 2 1))


Step: 7
 Paths:
  (109.32987 (11 10 8 6 4 3 2 1))
  (111.78762 (11 9 8 6 4 3 2 1))
  (118.55193 (11 9 7 5 4 3 2 1))
  (135.53412 (11 10 7 5 4 3 2 1))
  (137.50194 (11 10 8 5 4 3 2 1))
  (139.95969 (11 9 8 5 4 3 2 1))
```