Write Up 7

Zhe Yu | zy19@rice.edu

- Algorithm: bfs_dfs
  Input: Undirected graph *graph*, start node *start* in the *graph*, end node *end* in the *graph*, restricted access container class *rac_class*
  Output: $parent_{node,}$ for each *node* in *graph*: the parent of *node* in the exploration of the *graph* starting from *start*

  1. Initialize *rac* to be an instance of *rac_class*;
  2. Initialize *dist* to be an empty mapping;
  3. Initialize *parent* to be an empty mapping;
  4. **foreach** *node* in *graph* **do**
     a. $dist_{node} \leftarrow \infty$;
     b. $parent_{node} \leftarrow null$;
  5. $dist_{start} \leftarrow 0$;
  6. push *start* onto *rac*;
  7. **while** *rac* is not empty **do**
     a. pop *node* off from *rac;*
     b. **if** *node = end,* **then**
        i. **return** *parent*
     c. **foreach** neighbor *nbr* of *node* **do**
        i. **if** $dist_{nbr} = \infty$, **then**
           a. $dist_{nbr} \leftarrow dist_{node} + 1$;
           b. $parent_{nbr} \leftarrow node$;
           c. push *nbr* onto *rac*;
  8. **return** *parent*

- Recipe for recursive_dfs
  Base Case:
  If start node equals end node, then return true as we found the end node in the graph.
  If start node has no neighbors that have not been explored yet, then the function should return.

  Recursive Case:
  If start node has neighbors that have not yet been explored, then we will do the following for each of such neighbors *nbr* of start node:
  1. Update parent mapping so that the parent of *nbr* will be the start node
  2. Perform DFS with same graph, *nbr* as new start node, same end node, updated parent mapping

Algorithm: DFS (recursive version)
Input: Undirected graph *graph*, start node *start* in the *graph*, end node *end* in the *graph*, $parent_{node,}$ for each *node* in *graph*: the parent of *node* in the exploration of the *graph* starting from *start*
Output: If *end* is found in *graph*, return *true*; otherwise, return *false*

1. **if** *start* = *end* **then**
    a. **return** *true*
2. **foreach** *nbr* of *start* **do**
    a. **if** *nbr* is not in the mapping *parent* **then**
        i. $parent_{nbr} \leftarrow start$;
        ii. DFS (*graph, nbr, end, parent*), where *nbr* is as start node in calling DFS
3. **return** *false*

- Algorithm: A*
  Input: Undirected graph *graph*, start node *start* in the *graph*, end node *end* in the *graph*; function edge_distance (node1, node2, graph) which returns the actual distance required to travel from node1 to node2 in graph, where node1 and node2 are neighbors; function straight_line_distance (node3, node4, graph) which returns heuristic distance from node3 to node4 in graph, where node3 and node4 are not necessarily neighbors.
  Output: $parent_{node,}$ for each *node* in *graph*: the parent of *node* in the exploration of the *graph* starting from *start*

1. Initialize *f_cost* to be an empty mapping;
2. Initialize *g_cost* to be an empty mapping;
3. Initialize *h_cost* to be an empty mapping;
4. Initialize *parent* to be an empty mapping;
5. Initialize *openset* to be an empty set;
6. Initialize *closedset* to be an empty set;
7. **foreach** *node* in *graph* **do**
    a. $parent_{node} \leftarrow null$
8. $g\_cost_{start} \leftarrow 0$;
9. $h\_cost_{start} \leftarrow$ straight_line_distance (*start*, *end*, *graph*);
10. $f\_cost_{start} \leftarrow h\_cost_{start} + g\_cost_{start}$;
11. **add** *start* to *openset*;
12. **while** *openset* is not empty **do**
    a. $current \leftarrow null$;
    b. $f\_lowest \leftarrow \infty$;
    c. **foreach** *node* in *openset* **do**
        i. **if** $f\_cost_{node} < f\_lowest$ **then**
            1. $f\_lowest \leftarrow f\_cost_{node}$;
            2. $current \leftarrow node$;
    d. **if** *current* = *end* **then**
        i. **return** *parent*;
    e. **remove** *current* from *openset*;

f. **add** *current* in *closedset*;

g. **foreach** *nbr,* neighbor of *current* **do**

i. **if** *nbr* is not in *openset* nor *closedset* **then**

1. $parent_{nbr} \leftarrow current$;
2. $g\_cost_{nbr} \leftarrow g\_cost_{current} +$ edge_distance (*current, nbr, graph*);
3. $h\_cost_{nbr} \leftarrow$ straight_line_distance (*nbr*, *end*, *graph*);
4. $f\_cost_{nbr} \leftarrow h\_cost_{nbr} + g\_cost_{nbr}$;
5. **add** *nbr* in *openset*;

ii. **if** *nbr* is in *openset* only **then**

1. **if** $g\_cost_{nbr} > g\_cost_{current} +$ edge_distance (*current, nbr, graph*) **then**
   a. $g\_cost_{nbr} \leftarrow \mathrm{g\_cost}_{\mathrm{current}} +$ edge_distance (*current, nbr, graph*);
   b. $f\_cost_{nbr} \leftarrow h\_cost_{nbr} + g\_cost_{nbr}$;
   c. $parent_{nbr} \leftarrow current$;

13. **return** *parent*

- Discussion

problem 1: They are the same in concepts. Both will traverse along a deep path firstly. Yet, the recursive version calls itself (utilize python) to keep track of the state while bfs_dfs version uses stack to keep track of the state. bfs_dfs version looks at every neighbor of each node it explores in order to utilize stack while recursive version does not look at the next neighbor. bfs_dfs version is better because once we arrive at one node away from the destination, we are done while recursive version is done only when we actually reach the destination.

problem 2: A* gives the best routes. Only A* takes the weight of edge (distance) into account among all.

problem 3: DFS gives the worst routes. Because DFS does not try to find the shortest path while traversing the graph, it circles around the destination and takes much effort to finally reach the target. And between two version of DFSs, recursive version is worse because we have to reach the destination as said in **problem1.**

problem 4: Although A* algorithm may give the shortest path in terms of distance and Google does not generate shorter path, A* does not give the shortest path in terms of time. However, Google considers other factors such as traffic, road work, etc., to generate the path that takes least time (among other options such as least road turns, etc.).