Modify BFS

Recipe Name: BFS_DFS

Inputs:

graph, a directed graph

rac_class, a restricted access container

start_node, a node in *graph* representing the starting point

end_node, a node that is being searched for

Outputs:

parent, a mapping associating each visited node with its parent node

Steps:

1. Initialize *dist* to an empty mapping
2. Initialize *parent* to an empty mapping
3. For each *node* in *graph*, do
    a. $dist_{node}$ ← infinity
    b. $parent_{node}$ ← *null*
4. $dist_{start\_node}$ ← 0
5. if *rac_class* is a queue, then
    a. initialize *queue* to an empty queue
    b. push *start_node* onto *queue*
    c. while *queue* is not empty, do
        i. pop *node* off of *queue*
        ii. for each neighbor *nbr* of *node*, do
            1. if $dist_{nbr}$ is infinity, then
                a. $dist_{nbr}$ ← $dist_{node}$ + 1
                b. $parent_{nbr}$ ← *node*
                c. push *nbr* onto *queue*
            2. if *nbr* is *end_node*, do
                a. empty the queue
                b. end the for loop
6. if *rac_class* is a stack, then
    a. initialize *stack* to an empty stack
    b. push *start_node* onto *stack*
    c. while *stack* is not empty, do
        i. pop *node* off of *stack*
        ii. for each neighbor *nbr* of *node*, do
            1. if $dist_{nbr}$ is infinity, then
                a. $dist_{nbr}$ ← $dist_{node}$ + 1
                b. $parent_{nbr}$ ← *node*

        c. push *nbr* onto *stack*
    2. if *nbr* is *end_node*, do
        a. empty the queue
        b. end the for loop
7. Return *parent*

Implement Recursive DFS:

Base case 1:

When *start* has no neighbors that are not already in *parents*, return.

Base case 2:

When a *start* is *end*, return.

Recursive case:

When *start* has neighbors that are not already in *parents*, for each neighbor of *start* that is not in parents, map the parent of neighbor to start and run dfs(graph, neighbor, end, parents).

Recipe Name: dfs

Inputs:

graph, a directed graph

start_node, a node in *graph* representing the starting point

end_node, a node that is being searched for

parent, a mapping associating each visited node with its parent node

Steps:

1. For each neighbor *nbr* of *start_node*, do
    a. If *start_node* is *end_node*,
        i. Return
    b. If *nbr* is not in *parent*,
        i. $parent_{nbr} \leftarrow start\_node$
        ii. dfs(*graph*, *nbr*, *end_node*, *parent*)
2. Return

Implement A*

Recipe Name: astar

Inputs:

graph, a directed graph

start_node, a node in *graph* representing the starting point

end_node, a node that is being searched for

Outputs:

parent, a mapping associating each visited node with its parent node

Steps:

1. Initialize *parent* to an empty mapping
2. Initialize *openset* as an empty mapping
3. Initialize *closedset* as an empty mapping
4. $parent_{start\_node}$ ← null
5. $openset_{start\_node}$ ← 0
6. While *openset* is not empty, do
    a. Initialize *lowest_value* as -1
    b. For each *key*, *value* in *openset*,
        i. If *value* + the straight line distance between *key* and *end_node* on *graph* is lower than *lowest_value*, then
            1. *current_node* ← *key*
            2. *lowest_value* ← *value*
    c. If *current_node* is *end_node*,
        i. Return *parent*
    d. Otherwise,
        i. $closedset_{current\_node}$ ← $openset_{current\_node}$
        ii. Remove *current_node* from *openset*
        iii. For each neighbor, *nbr*, of *current_node*, do
            1. If *nbr* is not in *openset* and *nbr* is not in *closedset*, then
                a. $openset_{nbr}$ ← $openset_{current\_node}$ + the distance of the edge between *current_node* and *nbr* on *graph*
                b. $parent_{nbr}$ ← *current_node*
            2. Otherwise, if $openset_{nbr}$ is less than $openset_{current\_node}$ + the distance of the edge between *current_node* and *nbr* on *graph*, then
                a. $openset_{nbr}$ ← $openset_{current\_node}$ + the distance of the edge between *current_node* and *nbr* on *graph*
                b. $parent_{nbr}$ ← *current_node*
7. Return *parent*

Python code:

https://py3.codeskulptor.org/#user305_d2KF7OYPdU_29.py

Discussion

1. DFS with a stack almost always has a lower amount of unnecessarily traveled lines than DFS with recursion. While both produce random-looking pathways traveling in multiple directions, recursive DFS always traverses every node in the graph due to how recursion works. Thus, since the recursive version must spend an unnecessary amount of time traversing the graph and both produce nowhere near the shortest routes, the stack version is better.
2. BFS and A* seem to produce similar routes, but across longer distances there is a distinct delay in A*'s route production. Thus, BFS is the better algorithm for producing routes. This is because it is significantly more thorough than the other algorithms, approaching every possibility and finding the shortest pathway after finding every possibility. A*, on the other hand, spends an unnecessary amount of time with nested loops and so lags.
3. Recursive DFS seems to produce the worst routes because it is constantly going down every possibility, but without actually carefully finding shorter pathways. Thus, the routes it produces meander and typically take up entire chunks of the map.
4. Because google has had years if not decades of experience in finding the best possible algorithms for routes. Additionally, their algorithms are likely significantly longer and more complex and they have access to significantly more data than a first year computer science class.