Dijkstra Interpretation

Cristafalk
4/19/17
A2

Code + parameters test values (arguments)

```
def medium_test():
    test_graph = {
        "A":{"B":1, "C":2},   → "start_node"
        "B":{"C":5},          → (should be in path from ↑ to ↓)
        "C":{"D":1, "A":2},
        "D":{}                → "end_node"
    }
    assert [4,["A","C","D"]] == shortest_path("A","D", test_graph)
```

this graph = "graph"

ISSUE COMES INTO FRUITION WHEN GREEDY ALGO chooses B (path of 1) rather than C (path of 2), THEN B goes to C before going to D, so it is LESS EFFICIENT

```
def shortest_path(start_node, end_node, graph):
    current = start_node          current = "A"
    to_visit = [current]          to_visit = ["A"]
    visited = set()               visited = set(___) #BECAUSE NONE VISITED YET

    dist_from_start = {start_node: 0}   dist_from_start = {'A': 0}
    tent_parents = {}                   tent_parents = {} #BECAUSE ROOT HAS NO PARENTS

    while len(to_visit) > 0:
        current = min([(dist_from_start[val], val) for val in to_visit]   current = min([('A':0[v], for v in ["A"][1]
        if current == end_node:                                            #ALL ESSENTIALLY = "A"
            break                  current ≠ end_node IF "A" ≠ "D"
                                   here is the recursive break (base case)
        if current in to_visit:    IF "A" in ["A"] #YES
            to_visit.remove(current)   ["A"] - "A" = [] #TO_VISIT
            visited.add(current)       set() + "A" = set("A") #VISITED
        edges = graph[current]     edges = test_graph["A"] = {"B":1, "C":2}
        unvisited = set(edges).difference(visited)   unvisited = set({"B"1, "C":2}).difference("A")
        for neighbor in unvisited:   unvisited = ({"B":1, "C":2}) = {'B', 'C'}
            neighbor_dist = dist_from_start[current] + edges[neighbor]   n_d = d_f_s["A"] + edges[B, then C
            if neighbor_dist < d_f_s.get(n_d, float('inf')):   if 1 < inf   n_d = 0 + 1 = 1
                dist_from_start[neighbor] = neighbor_dist      dist_from_start of 'C' = 2   n_d = 0 + 2 = 3
                tent_parents[neighbor] = current               tent_parents of C = A
                to_visit.append(neighbor)                      to_visit = ['C']
                                                               #THEN REPEAT FOR B starting @ WHILE (because to_visit has been added to BECAUSE a neighbor still exists)
    return [neighbor_dist, deconstruct_path(tent_parents, end_node)]   tent_parents = {"C":'B', D:C, B:A}
                                                                       end_node = D
def deconstruct_path(tent_parents, end_node, time=0):
    if end_node not in tent_parents:
        return None
    goal = end_node   goal = D
    path = []
    while goal:
        path.append(goal)   path = [D]   path = [D,C]   path = [D,C,B]   path = [D,C,B,A]
        goal = tent_parents.get(goal)   goal = [C]   goal = B   goal = A   goal = True, so while ends
    deconstructed_node_path = list(path[::-1])   d_n_p = [A,B,C,D]   return deconstructed_node_path
```

returns [

Program ends here → return [neighbor_dist, deconstruct_path(tent_parents, end_node)]