Big O Complexity

Patrick Devanney – Tracking on a Graph

Search

search – $O(n^3)$

`def search(distance_to_every_node, distance_to_target, visited_nodes):`	
`    current_tower = 0`	O(1)
`    possible = []`	O(1)
`    for tower in distance_to_every_node:`	O(n)
`        possible_nodes_ind = []`	O(1)
`        for node in tower:`	O(n)
`            if (tower[node] == distance_to_target[current_tower]) and (node not in visited_nodes):`	O(n)
`                possible_nodes_ind.append(node)`	O(1)
`        possible.append(possible_nodes_ind)`	O(1)
`        current_tower += 1`	O(1)
`    possible = [x for x in possible if x != []]`	O(n)
`    if len(possible) == 0:`	
`        return []`	O(1)
`    elif len(possible) == 1:`	
`        return possible[0]`	O(1)
`    else:`	
`        setlist = []`	O(1)
`        for arr in possible:`	O(n)
`            setlist.append(set(arr))`	O(1)
`        return list(set.intersection(*setlist))`	O(n) ???

➔ $O(1) + O(1) + O(n)(O(1) + O(n)(O(n)(O(1))) + O(1) + O(1)) + O(n) + O(1) + O(n)$

➔ $O(1) + O(n)(O(n)(O(n)) + O(1)) + O(n)$

➔ $O(1) + O(n)(O(n^2) + O(1)) + O(n)$

➔ $O(1) + O(n)(O(n^2)) + O(n)$

➔ $O(1) + O(n^3) + O(n)$

➔ $O(n^3)$

is_found – $O(n^3)$

`def is_found(graph, target_location, tower_locations, visited, distances):`	
`    distance_to_target = []`	O(1)
`    for t in tower_locations:`	O(n)
`        distance_to_target.append(current_distance_to_target(graph, t, target_location))`	O(n log n)
`    s = search(distances, distance_to_target, visited)`	$O(n^3)$
`    return len(s) == 1`	O(1)

- ➔ $O(1) + O(n)(O(n \log n)) + O(n^3) + O(1)$
- ➔ $O(n^2 \log n) + O(n^3)$
- ➔ $O(n^3)$

Distance

current_distance_to_target – O(n Log n)

`def current_distance_to_target(G, To, Ta):`	
`    try:`	
`        return len(nx.dijkstra_path(G, To, Ta)) - 1`	O(n Log n)
`    except nx.NetworkXNoPath:`	
`        return -1`	O(1)

- ➔ O(1) + O(n Log n))
- ➔ O(n Log n)

populate_distance_table – O(n^2 Log n)

`def populate_distance_table(graph, tower_location):`	
`    tower_distance = {}`	
`    for node in graph.nodes():`	O(n)
`        try:`	
`            tower_distance[node] = len(nx.dijkstra_path(graph, tower_location, node)) - 1` `        except nx.NetworkXNoPath:`	O(n Log n)
`            tower_distance[node] = -1`	O(1)
`    return tower_distance`	

- ➔ O(1) + O(n)(O(n Log n)) + 0(1)
- ➔ O(1) + O(n^2 Log n) + O(1)
- ➔ O(n^2 Log n)

Tower

Random Tower

`def initial_position(graph, tower_count):`	
`    return random.sample(graph.nodes, tower_count)`	O(n^2) ??

- ➔ O(n^2)

Heuristic Tower

`def initial_position(graph, tower_count):`	
`    unique_distances = []`	O(1)
`    for node in graph.nodes:`	O(n)

`        unique_distances.append(`
`                len(set((populate_distance_table(graph,`
`                node)).values())))`
	O(n^2 log n)

`    heuristic = np.argpartition(unique_distances, -` `        tower_count)[-tower_count:]`	O(n) ??
`    return heuristic`	

- ➔ O(1) + O(n)(O(n^2 Log n)) + 0(n)
- ➔ O(n^3 Log n)

Target

Random Target

`def initial_location(graph, tower_locations):`	

`number_of_nodes = len(graph.nodes)`	O(1) ??
`random_location = random.randrange(0, number_of_nodes)`	O(1)
`while random_location in tower_locations:`	O(n)
`    random_location = random.randrange(0, number_of_nodes)`	O(1)
`return random_location`	O(1)

- ➔ O(1) + O(1) + O(n)(O(1)) + O(1) + O(1)
- ➔ O(1) + O(n)
- ➔ O(n)

`def next_move(possible_moves, turn=-1):`	
`    return possible_moves[random.randrange(0, len(possible_moves))]`	O(1)

- ➔ O(1)

Heuristic Target

`def initial_location(graph, tower_locations):`	
`    common_distances = []`	O(1)
`    for node in graph.nodes:`	O(n)
`        common_distances.append(len(set(populate_distance_table (graph, node)).values())))`	$O(n^2 \log n)$
`    index = np.argmin(common_distances)`	O(1)
`    while index in tower_locations:`	O(n)
`        common_distances[index] = len(graph.nodes) + 1`	O(1)
`        index = np.argmin(common_distances)`	O(1)
`    return index`	O(1)

- ➔ O(1) + O(n)($O(n^2 \log n)$) + O(n)(O(1) + O(1)) + O(1)
- ➔ O(1) + $O(n^3 \log n)$ + O(n) + O(1)
- ➔ $O(n^3 \log n)$

`def heuristic_target_next_move(graph, towers, visited, distances, possible_moves):`	
`    one_step = []`	O(1)
`    two_step = []`	O(1)
`    for move in possible_moves:`	O(n)
`      if not is_found(graph, move, towers, visited, distances):`	$O(n^3)$
`        one_step.append(move)`	O(1)
`          for neighbour in [x for x in graph.neighbors(move) if x not in visited]:`	$O(n^2)$
`            if not is_found(graph, neighbour, towers, visited+[move], distances):`	$O(n^3)$
`              two_step.append(move)`	O(1)
`              break`	O(1)
`     if len(one_step) > 0:`	O(1)
`       if len(two_step) > 0:`	O(1)

`return random.choice(two_step)`	O(1)
`return random.choice(one_step)`	O(1)
`return random.choice(possible_moves)`	O(1)

- ➔ $O(1) + O(1) + O(n)(O(n^3) + O(n^2)(O(n^3) + O(1) + O(1))) + O(1)$
- ➔ O(1) + O(n)(O(n^3) + O(n3)(O(n2) + O(1) + O(1)) + O(1)
- ➔ O(1) + O(n)(On^3) + O(n^5)) + O(1)
- ➔ O(1) + O(n^6) + O(1)
- ➔ O(n^6)

Optimal Functions

optimal_is_found – $O(n^3)$

`def optimal_is_found(target_location, visited, distances, distance_to_target):`	
`    target_distance = [item[target_location] for item in distance_to_target]`	O(n)
`    s = search(distances, target_distance, visited)`	$O(n^3)$
`    return len(s) == 1`	O(1)

- ➔ $O(n) + O(n^3) + O(1)$
- ➔ $O(n^3)$

build_tree – $O(n^4\ n!)$

`def build_tree(graph, tree, node, parent, distance_table, tower_locations, distance_to_target):`	
`    v = [int(n) for n in parent.split(',')]`	O(n)
`    for t in tower_locations:`	O(n)
`        v.append(t)`	O(1)
`    possible_moves = []`	
`    for n in graph.neighbors(node):`	O(n)
`        if n not in v:`	O(n)
`            possible_moves.append(n)`	O(1)
`    for n in possible_moves:`	O(n)
`        ident = parent + "," + str(n)`	O(1)
`        tree.create_node(n, ident, parent=parent)`	O(log n)
`        if not optimal_is_found(n, v, distance_table, distance_to_target):`	$O(n^3)$
`            build_tree(graph, tree, n, parent + "," + str(n), distance_table, tower_locations, distance_to_target)`	O(n!)

- ➔ $O(n) + O(n)(O(1)) + O(n)(O(n)(O(1))) + O(n)(O(1)+O(\log n)+O(n^3)+ O(n!)$
- ➔ O(n) + O(n) + O(1) + O(n^2) + O(n)(O(n^3)(O(n!)))
- ➔ O(n) + O(n^4 n!)
- ➔ O(n^4 n!)

optimal_path – $O(n^4\ n!)$

`def optimal_path(graph, target, tower_locations, distance_to_target):`	
`        all_path = Tree()`	O(1)
`        all_path.create_node(target, str(target))`	O(log n)
`        distance_table = []`	O(1)
`        for t in tower_locations:`	O(n)
`            distance_table.append(populate_distance_table(graph,t))`	$O(n^2 \log n)$
`        possible_moves = []`	O(1)
`        for i in graph.neighbors(target):`	O(n)
`            if i not in tower_locations:`	O(n)
`                possible_moves.append(i)`	O(1)
`        if not optimal_is_found(target, tower_locations, distance_table, distance_to_target) and len(possible_moves) > 0:`	$O(n^3)$
`            build_tree(graph, all_path, target, str(target), distance_table, tower_locations, distance_to_target)`	$O(n^4\ n!)$
`        leaves = all_path.leaves()`	O(1)
`        depth = {}`	O(1)
`        for leaf in leaves:`	O(n)
`            length = all_path.depth(leaf) + 1`	O(1)
`            if length in depth.keys():`	O(n)
`                depth[length].append(leaf.identifier)`	O(1)
`            else:`	
`                depth[length] = [leaf.identifier]`	O(1)
`        longest_path = []`	O(1)
`        longest_path_list = depth[max(depth)]`	O(1)
`        for longest_path_string in longest_path_list:`	O(n)
`            longest_path.append(list(map(int, longest_path_string.split(','))))`	O(1)
`        return max(depth), longest_path`	O(1)

➔ $O(1) + O(\log n) + O(1) + O(n^3 \log n) + O(n^2) + O(n^3) + O(n^4\ n!) + O(1) + O(n^2) + O(1) + O(n) + O(1)$

➔ $O(n^4\ n!)$