Assignment 1: Finding Our Way - Comparing Breadth-First and Depth-First Search

Ali Hassan
Student ID: F22BDOCS1E02011

March 9, 2025

1 Introduction

Imagine you've got a map with a bunch of towns connected by roads, and you need to figure out the best way to get from one town to another. That's pretty much what this assignment is about. We're using a computer to find routes between places, and we're trying out two different methods: Breadth-First Search (BFS) and Depth-First Search (DFS). Think of BFS as exploring layer by layer, and DFS as following one path as far as it goes before trying another. We're interested in finding the shortest route, especially when we know the distances between towns.

2 Breadth-First Search (BFS) Solution

So, BFS is like checking all your immediate neighbors before moving on. It's like if you're looking for a lost item in a house, you'd check every room on the first floor before going upstairs.

Here's the Python code I used:

```
# Name: Ali Hassan
# Student ID: F22BDOCS1E02011
# Assignment: Assignment 1 - BFS.ipynb

def find_shortest_way(places, start_point, end_point):
  check_these_places = [(start_point, [start_point])] # places to look at later
  places_i_already_went = set() # don't go back!

  while check_these_places:
    this_place_now, the_path_i_got = check_these_places.pop(0) # first one in, first one out?

    if this_place_now not in places_i_already_went:
      if this_place_now == end_point:
        return the_path_i_got # found it!

      places_i_already_went.add(this_place_now)

      for next_place_to_go in places.get(this_place_now, []):
        check_these_places.append((next_place_to_go, the_path_i_got + [next_place_to_go])) # add to the

  return None # guess there's no way

my_place_connections = [
  ('S', 'A', 3), ('S', 'B', 6), ('S', 'C', 2),
  ('A', 'D', 3), ('B', 'E', 2), ('C', 'E', 1),
  ('D', 'F', 5), ('F', 'G', 5), ('E', 'H', 5),
  ('H', 'G', 5), ('B', 'G', 9), ('D', 'B', 4),
  ('E', 'F', 6)
]

connections_map = {} # make a thing to hold the places
for place_from, place_to, distance_between in my_place_connections:
  connections_map.setdefault(place_from, []).append(place_to)
  connections_map.setdefault(place_to, []).append(place_from) # both ways!

start_of_trip = 'S'
end_of_trip = 'G'
```

```
the_real_route = find_shortest_way(connections_map, start_of_trip, end_of_trip)

if the_real_route:
  print("Found␣the␣shortest␣way!␣It's:", the_real_route) # yay!
else:
  print("Couldn't␣find␣a␣route␣:(") # oh no
```

When I ran this, it gave me the shortest route between 'S' and 'G'. The cool thing about BFS is that it guarantees to find the shortest path if you're considering the number of steps or, in our case, the distance values we provided. It's pretty reliable for finding the most efficient way to get somewhere.

The downside? If the map gets huge, BFS might take a while because it has to check so many places at once.

3 Depth-First Search (DFS) Solution

Now, DFS is more like going down one rabbit hole at a time. It's like if you're exploring a maze, you'd pick a direction and keep going until you hit a dead end, then backtrack and try another way.

Here's the DFS code I used:

```
# Name: Ali Hassan
# Student ID: F22BD0CS1E02011
# Assignment: Assignment 1 - DFS.ipynb

def find_a_route(place_list, start_place, end_place):
  to_look_at = [(start_place, [start_place])] # make a list of places to check
  already_seen = set() # keep track of where i went

  while to_look_at:
    current_spot, the_route_so_far = to_look_at.pop() # get the last one?

    if current_spot not in already_seen:
      if current_spot == end_place:
        return the_route_so_far # found it!

      already_seen.add(current_spot)

      for next_place in place_list.get(current_spot, []):
        to_look_at.append((next_place, the_route_so_far + [next_place])) # add more to check

  return None # guess there's no route?

my_places_and_roads = [
  ('S', 'A', 3), ('S', 'B', 6), ('S', 'C', 2),
  ('A', 'D', 3), ('B', 'E', 2), ('C', 'E', 1),
  ('D', 'F', 5), ('F', 'G', 5), ('E', 'H', 5),
  ('H', 'G', 5), ('B', 'G', 9), ('D', 'B', 4),
  ('E', 'F', 6)
]

connections_between_places = {} # make a thing to hold the connections
for place1, place2, distance in my_places_and_roads:
  connections_between_places.setdefault(place1, []).append(place2)
  connections_between_places.setdefault(place2, []).append(place1) # both ways!

start_trip_here = 'S'
end_trip_there = 'G'

the_path_we_need = find_a_route(connections_between_places, start_trip_here, end_trip_there)

if the_path_we_need:
  print("Found␣the␣path!␣It's:", the_path_we_need) # yay!
else:
  print("Couldn't␣find␣a␣path␣:(") # oh no
```

DFS finds *a* path, but it doesn't guarantee the shortest one. It's good if you just need to find *any* way to get from A to B, and it can be faster than BFS in some cases. However, it can get stuck exploring long paths that don't lead anywhere.

4 Comparison and Conclusion

When I compared the results, BFS gave me the shortest route based on the distances. DFS gave me a route, but it wasn't necessarily the most efficient. This makes sense because BFS is designed to find the shortest path, while DFS is more about exploring all possible paths.

For this assignment, BFS is definitely the better choice if we care about finding the shortest route based on distances. It's reliable and gives us the optimal solution. DFS could be useful if we just needed to find *any* route, but for efficiency, BFS wins.

In conclusion, both BFS and DFS are useful tools for exploring connections between places, but they have different strengths and weaknesses. Understanding these differences helps us choose the right tool for the job.