Advent of Code - Day 16: Proboscidea Volcanium

Aditya Gupta

February 2023

Introduction

In this assignment, we solve the first part of Advent of Code Day 16.

Part 1

For the first part of the problem we have been told that we are lost inside a cave that is connected to a volcano with a bunch of elephants, and the volcano is about to erupt. We have 30 minutes before the volcano erupts so we need to navigate through the system of pipes in order to escape. We start at the Valve *"AA"* and it takes one minute to open a valve and one minute to go to another directly connected valve from your current position. For this task, we need to find out what the maximum pressure is that we can release while traversing the tunnels.

Approach

At first, the problem seems quite intimidating, but it becomes a lot easier once we try to break it into smaller chunks.

1. The first part of the problem involved parsing the text input. For this part, I just hard-coded my text input into a List of tuples as shown below:

```
# Structured in the following way: {Valve, Flowrate, Connected_Valves}
[
  {"VR", 11, ["LH", "KV", "BP"]},
  {"UV", 0, ["GH", "RO"]},
  {"OH", 0, ["AJ", "NY"]},
  {"GD", 0, ["TX", "PW"]},
  {"NS", 0, ["AJ", "AA"]},
  {"KZ", 18, ["KO", "VK", "PJ"]},
  :
  {"RI", 3, ["NV", "KE", "LN", "XH", "TX"]}
]
```

2. After this we have to generate a list of all unvisited valves with a $flowrate > 0$. We are not interested in valves with 0 flow rate other than if they fall in between the path to other valves that we are interested in. In order for us to be able to calculate the pressure released by each valve we have the following formula:

   pressure = ((Time_Left) - (Time_to_reach_a_valve) - (Time_to_open_a_valve)) * Flowrate

   But, in order to be able to find the value for the pressure we need to know the time it takes for us to go from our current position to any valve we want to go to. In order to do that we can make use of the Floyd-Warshall algorithm to create a matrix containing the distances from all valves to each of the other valves. Then we can just look up the distance between any two nodes for example between the valves "AA" and "VR" instantly.

3. Finally we do a Depth-first search on all unvisited valves to find the max pressure.

Creating a list of all unvisited valves

Now we create the list of only the valves with a non-zero flow rate from the list we created before containing all valves.

```
def create_unvisited_nodes_list(list) do
    Enum.filter(list, fn(x)->{_, rate, _} = x; rate > 0 end)
    |> Enum.map(fn(x)->{letter, rate, _} = x; {letter, rate} end)
end
```

Floyd-Warshall Algorithm

The Floyd-Warshall Algorithm is a dynamic programming algorithm which is used to find the shortest path in a weighted graph. The algorithm runs with a time complexity of $O(n^3)$, where n is the number of vertices in the graph. My first approach towards solving the algorithm was to try and do this with Dijkstra's algorithm instead, but the idea of making all computations for all edges at once seemed more appealing and hence I shifted my approach to the FW Algorithm instead. Here's some pseudo-code for the algorithm:

```
// Pseudo-Code for Floyd Warshall algorithm
// Input: A weighted graph represented by an n x n matrix of edge weights.
// Output: An n x n matrix of the shortest path distances between all pairs of vertices.

function floyd_warshall(graph):
    //let n be the number of vertices in the graph.
    //let dist be an n x n matrix of minimum distances,
    //initially set to the weights of the edges in the graph.
    for k from 1 to n:
        for i from 1 to n:
            for j from 1 to n:
                if dist[i][j] > dist[i][k] + dist[k][j]:
                    dist[i][j] = dist[i][k] + dist[k][j]

    return dist
```

Implementation

I break the algorithm into 3 portions for my implementation:

- Create the matrix which is going to store all the distances
- After this we set the distance value for all immediately adjacent valves
- Finally we perform the nested loop which we see in the pseudo-code

```elixir
def floyd_warshall(list, unvisited, map) do
    matrix = create_matrix(list)
    matrix = init_distance_of_all_immediately_adjacent(matrix, list, map)
    matrix = nested_loop(matrix, length(list))
    dfs_search(unvisited, map, matrix)
end

def create_matrix(list) do
    size = length(list)
    matrix = Matrix.new(size, size, nil)
    Enum.to_list(0..(size-1)) |> Enum.reduce(matrix, fn(x, acc)->
        Matrix.set(acc, x, x, 0) end)
end

def init_distance_of_all_immediately_adjacent(matrix, list, map) do
    weights = get_adjacent_valves(list, map, [])
    Enum.reduce(weights, matrix, fn(x, acc) ->
        {row, col}=x; Matrix.set(acc, row, col, 1) end)
end

def get_adjacent_valves([], _, result) do List.flatten(result) end
def get_adjacent_valves([head|rest], map, result) do
    {from, _, to} = head
    {:ok, from} = Map.fetch(map, from)
        #Fetch row and column index for valve from the map
        new = for item <- to do
            {:ok, tto} = Map.fetch(map, item)
            {from, tto}
        end
    get_adjacent_valves(rest, map, [new|result])
end
```

Depth-First Search on all unvisited valves

Finally, we use a depth-first search in order to recursively find the maximum pressure by looking at all possible combinations to find the path that releases the most pressure. We exit the program for a path if all valves have been visited or the time limit has been reached.

```
def dfs(_, _, _, _, _, _, score) do
    score
end

def dfs_search(unvisited, map, matrix) do
    node = "AA"
    Enum.each(unvisited, fn({key, valverate})->
        IO.puts("Key: #{key}, Rate: #{valverate}") end)
    for unvisitedNode <- unvisited do
        dfs(node, unvisitedNode, unvisited, map, matrix, 30, 0)
    end |> Enum.max()
end

def dfs(current_node, node, unvisited, map, matrix, time, score)
when time > 0 do
    {key, valverate} = node
    {newscore, time} = add_score(current_node, key, valverate, map, matrix, time, score)
    cond do
        time < 0 -> score
        true ->
            unvisited = remove_from_unvisited(unvisited, key)
            case unvisited do
                [] -> newscore
                _ ->
                    for unvisitedNode <- unvisited do
                        dfs(key, unvisitedNode, unvisited, map, matrix, time, newscore)
                    end |> Enum.max
            end
    end
end
```