Algorithms 1: Pick your battles

Name: Lucas Jacobs
Pcn: 490692
Class: APS4-RB01
Teachers: Simona Orzan
Date: 12-03-2023
Word count: 1134

Contents

Problem description 3
Problem A 3
Problem B 3
Problem C 4
Problem D 4
Chosen problem D 5
Code 6
Time complexity 7

Problem description

Problem A

For this problem, I want to use the backtracking strategy. We want the optimized path, with this algorithm, it will continue until it found the shortest path. With this approach the backtracking algorithm can start at the beginning and then use the four movements (up, down, left, and right), to find the exit door.
For time complexity, the maze is M * N big, so it will be exponential in the worst-case scenario. We have movements: up, down, left, and right. So the big O notation will be O(4^n), in this n is the length of the shortest path from start to exit.
With data structures, to visualize a maze we need a 2-dimensional array. This array has the values: 0 (wall), 1 (path), and 2 (exit). We can use a stack or a queue to keep track of the visited path. Also, a recursive function to properly implement a backtracking algorithm.

Problem B

With this problem, we can use the backtracking strategy to brute force all the possible ways. Backtracking recursively checks all possible ways that are left, when a solution is found, it will return that solution and stops. We want to assign all the characters to a different value (0-9).
We need to check if the sum of the characters value of String1 + String2 is equal to the value of String3, if this is not the case we start again with backtracking. If the equation is satisfied it will return the solution of the values with assigned characters.
Time complexity will be O(n!). There are in total 10 digits (0-9), and there are n unique characters, so the possible, so there will be 10^n possible ways to combine with these characters. In the worst-case scenario, every character is assigned a value so it needs to try all possible permutations of the 10 for each of the n unique characters. So the worst case when we have 10 digits will be 10! so this means:
10 * 9 * 8* 7 * 6 * 5 * 4 * 3 * 2 * 1 = 3628800 which is the total number of the digits 0 to 9 that can be arranged.
The data structures that I will use are:

- Recursive function: implement the backtracking algorithm
- List: digits that are available for characters and a list of what is in use
- Dictionary: map a character to a digit (to keep track of it)

Note, Permutation: calculation of this number of possible different rankings.

Problem C

This problem can be dynamically programmed, we want to memorize the amount of a minimum number of operations so this can be stored in a list. We can use recursion to calculate the minimum operation to convert X to Y. An expected time complexity will be O(XY). Depending on the state of X and Y we do an operation:

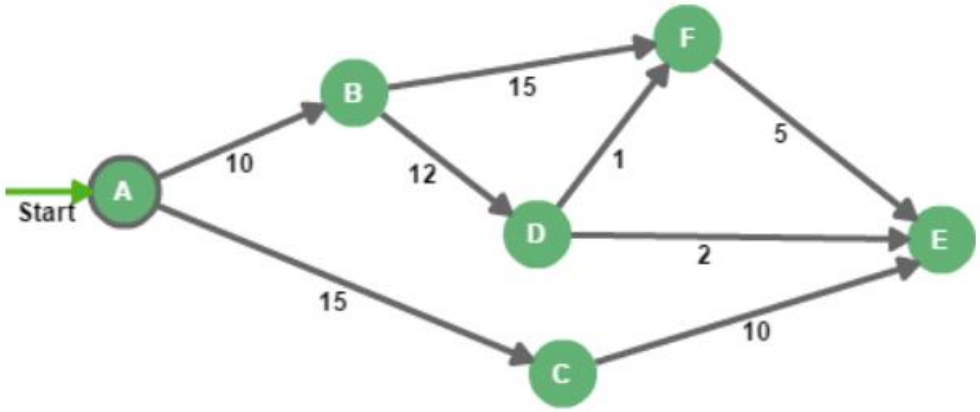
- (1) X = min(X * m, Y * 2)
- (2) X = X – 2
- (3) X = X – 1.

Also for this problem, regarding data structures, I will just use a simple variable to store values. Since we have three options, where the first option increases X and options two and three decreases X. Also we need a list to store the sequence of how we would compute the minimum number of operations.

Problem D

With this problem, the shortest path needs to be found between two vertices. To do this, I want to use Dijkstra's shortest path algorithm. This is an algorithm based on a greedy strategy. It is considered greedy because it will choose the smallest option that is possible at each step.

Let us give an example to get a clear understanding of what we are doing. Given is a directed graph and applies the Dijkstra's shortest path algorithm to go from A to E:



	A	B	C	D	E	F	Selected Edge
Initial	(0, A)	-	-	-	-	-	-
	X	(10, A)	(15, A)				{A, B}
		X		(22, B)		(25 , B)	{A, C}
			X		(25, C)		{B, D}
				X	(24, D)	(23, D)	{D, F}
					~~(28, F)~~	X	{E, D}

Shortest path: A – B – D – E
Weight: 23

Furthermore, for implementing this algorithm, I want to use a priority queue, which is a special queue where every element is associated with a priority value. the time complexity of this algorithm will be at worst O(n^2) because at worst need to check every vertex with each other (n is the number of vertices in the graph). For data structures, I will use an array as the priority queue.

Chosen problem D

For this problem, I will write the code to solve the problem that I mentioned when describing how I would approach this problem. Instead of giving one smallest path, we are asked to program to find the smallest paths from the source vertex to all the other vertices. In this problem, you need to have a matrix with the vertices and the edges between those vertices, also a matrix with the weight of the edges between these vertices. In the problem description, I already explained how I am going to encourage this problem (Dijkstra's shortest algorithm).

Matrix of the picture of the vertices with edges:

```
 A  B  C  D  E  F
[0  1 |1  0  0  0] A
[0  0  0  1  0  1] B
[0  0  0  0  1  0] C
[0  0  0  0  1  1] D
[0  0  0  0  0  0] E
[0  0  0  0  1  0] F
```

Matrix of the weights:

```
 A  B  C   D  E  F
[0  10 15  0  0  0]  A
[0  0  0  12  0  15] B
[0  0  0  0  10  0]  C
[0  0  0  0  2  1]   D
[0  0  0  0  0  0]   E
[0  0  0  0  5  0]   F
```

Code

```
#Making the graph

#matrix to represent the vertices in a graph, where vertice[x][y] = 1 there is a edge, 0 when there is none
verticesInput = [[0, 1, 1, 0, 0, 0],
                 [0, 0, 0, 1, 0, 1],
                 [0, 0, 0, 0, 1, 0],
                 [0, 0, 0, 0, 1, 1],
                 [0, 0, 0, 0, 0, 0],
                 [0, 0, 0, 0, 1, 0]]

#this matrix gives the weight of every edge in the graph, vertice[x][y] is weight of a edge between two vertices, 0 when there is none
edgesInput = [[0, 10, 15, 0, 0, 0],
              [0, 0, 0, 12, 0, 15],
              [0, 0, 0, 0, 10, 0],
              [0, 0, 0, 0, 2, 1],
              [0, 0, 0, 0, 0, 0],
              [0, 0, 0, 0, 5, 0]]

#A, represents the starting vertex
source_vertex = 0
```

```
#we want to find the shortest path from vertext A so index 0, to all the other vertices
def Dijkstra_shortest_path(vertices, edges, source):
    #amount of vertices in graph
    amount_of_vertices = len(vertices[0])

    #Checks for valid graph
    if amount_of_vertices > 1:
        #all vertices get assigned to value infinity
        distances = [float('inf')] * amount_of_vertices

        distances[source] = 0

        #store the shortest paths from source vertex to all the other vertices
        paths = [[] for _ in range(amount_of_vertices)]
        paths[source] = [source]

        #contain all the vertices
        unvisited = set(range(amount_of_vertices))
```

```
        #runs while there are unvisted vertices
        #each iteration it will find the smallest unvisted vertex to the source (shortest distance)
        while unvisited:

            current = min(unvisited, key=lambda x: distances[x])
            unvisited.remove(current)
            for neighbor in range(amount_of_vertices):
                #checks if the vertex is there and if the vertext is in the unvisted set
                if vertices[current][neighbor] == 1 and neighbor in unvisited:
                    distance = distances[current] + edges[current][neighbor]
                    if distance < distances[neighbor]:
                        distances[neighbor] = distance
                        paths[neighbor] = paths[current] + [neighbor]

        return paths
    else:
        return print("not a valid graph")
```

```
# Test case: graph finds the valid paths
expected_output1 = [[0], [0, 1], [0, 2], [0, 1, 3], [0, 1, 3, 4], [0, 1, 3, 5]]
assert Dijkstra_shortest_path(verticesInput, edgesInput, source_vertex) == expected_output1

# Test case: Graph with only one vertex
expected_output2 = "not a valid graph"
assert Dijkstra_shortest_path([[0]], [[0]], 0) == expected_output2
```

This code as explained how I would approach it in the introduction of the problem, the code has a list ‘distances’ (that will store the shortest path from each vertex to the source) and a set ‘unvisited’ (keeps track of the vertices that have not been visited yet).
With the variable: ‘current’, each iteration will be updated with the smallest distance to the source vertex.
Furthermore, with the ‘min’ function, we use a lambda expression to return the vertex with the smallest distance to the source vertex. This also behaves like a priority queue (every element is associated with a value). So instead of using a data structure such as a binary tree, we use the built-in function ‘min’ to find the vertex with the smallest distance.
To conclude, my code doesn’t explicitly use a priority queue, rather it uses a ‘min’ function that acts like a priority queue.

Time complexity

The time complexity of this code (Dijkstra’s shortest algorithm) is O(n^2) where n is the number of vertices in the graph. To tell this from the code, the function Dijkstra has a nested loop. For both the inner loop and outer loop, it will iterate threw the whole adjacency matrix. Therefore n * n = n^2.