CSC263 Assignment #3

Divij Sanjanwala, Vaibhav Holani

April 15 2021

Question 1

1a)

The solution algorithm uses two data structures, one to represent a block in the maze and the other to represent the maze.

Data Structure 1: MazeNode (Represent a block in the node):

The data structure consists of the following attributes:

1. position: (y, x) position in the given maze
2. directions: a list of strings showing the directions in which the maze block pointing to. The list is a subset of the following directions. [N, NE, E, SE, S, SW, W, NW]
3. color: the color of the arrows
4. visited: keeps track of all the d values with which the node has been visited
5. goal: identifies if the current node is the goal node or not

Data Structure 2: Maze (Represent the entire maze):

The data structure consists of the following attributes:

1. length: the length of one side of the Maze
2. nodes: a 1D list containing all the nodes
3. matrix: represent the matrix as a list of lists with each list containing length elements
4. startNode: the index (in nodes) where the startNode is stored

Algorithm:
The implementation is based on the BFS algorithm. BFS has been used to ensure that the smallest path (the one that uses the minimum number of operations) to the goal block can be found.

Traversal and structure:
The algorithm considers each block of the maze combined with a certain step-size as a vertex in a graph: one block may be treated as a different vertex when visited with a different step-size. Moreover, The maze has been stored in a $n \cdot n$ matrix where n is length of one side of the matrix. This matrix representation is used to efficiently find

the neighborhood of a vertex visited with a certain step size. Thus, the data structure does not store a adjacency list for any vertex and instead calculates the adjacent vertices using the directions of the arrows and the step size. If the destination node is outside of the matrix, it is ignored and not en-queued.

Coloring and revisiting:
Each time a node is visited, the current step-size (d) is noted and the node is marked gray for the visit with step size d by adding the d to a attribute list **visited**. Thus, a node can only be visited once with a certain d value.

Path Tracking and step size increments:
As we are only interested in finding the path to goal vertex. The BFS queue is modified to store the path from start node to node being en-queued. Moreover, as the step size may differ for different paths being explored from the startNode, each node is en-queued with certain d value. The step size in increased when a node with red arrows is encountered and the step size is reduced when an node with yellow arrows is discovered.
This is implemented by en-queuing a tuple that contains information in the following format :
`(MazeNode, stepSize(d), [path from start to curr]`

Execution and GoalNode:
Each MazeNode stores a property which identifies the goal node. Once the goal node is found, the execution of the BFS is finished and the path is printed along with it's length. However, if the BFS execution finishes and the goal node been found, in that case, there is no solution to the maze which, accordingly, prints a message.

1b)

The text representation has been organised in the following format:

1. Line 1 stores the **side length** of the maze.
2. Line 2 stores the **index of the startNode** in the sequence provided in the file.
3. The rest of the file contains a representations for each of the MazeNodes in order. **Directions** in which the Node can travel in are represented by the **first parameter**, the **second parameter** identifies the **color** of the arrows and the **third parameter identifies** whether or not a given node is the **goal node**.

1d)

The following tests have been added:

1. The **example maze** given in question checks the algorithm for the correct functioning of the following features:

   (a) Correct increment and decrements of step size on encountering nodes with red and yellow arrows respectively as the only way to get to the goal is through going from the first node in row 1 of the matrix to the 3rd node in row 1 of the matrix where the first node has red arrows and the other node has yellow arrows.

   (b) Ignoring the destination nodes that are not within in the maze matrix. Whenever, a node has a destination which is not in the matrix, it causes an IndexError in python. The algorithm catches that error to let the algorithm execute properly.

   (c) A node cannot be visited twice with the same step size. When the right direction of the startNode is explored, it forms a loop when the 2nd node in the 3rd row points towards the 3rd node in the second row which has already been discovered with the same d value. However, the node is not enqueued else it would form an infinite loop and the function would never the queue would never be empty.

   (d) Keeping track of correct path from the startNode to the GoalNode, if exists.

   (e) **Expected Output:**
   (Node: (2, 0), d: 1) → (Node: (1, 0), d: 1) → (Node: (0, 0), d: 1) → (Node: (0, 2), d: 2) → (Node: (1, 1), d: 1) → (Node: (0, 1), d: 1)

   (f) **Actual Output:** Found!

   Number of Steps: 6

   1. Node: (2, 0), d: 1
   2. Node: (1, 0), d: 1
   3. Node: (0, 0), d: 1
   4. Node: (0, 2), d: 2
   5. Node: (1, 1), d: 1
   6. Node: (0, 1), d: 1

2. <u>**No solution Maze - no_solution.txt:**</u>
   The test checks the behavior of the algorithm when the maze is not solvable. A simple maze of $2 \cdot 2$ has been used where none of the nodes that are not goal nodes are pointed towards the goal node, thus having no path to the goal node.

   (a) **Expected Output:**
   No solution

   (b) **Actual Output:**
   No solution

3. <u>**Twice visit Node for Goal - twice.txt:**</u>

   The test checks if the algorithm behaves correctly when one needs to visited twice in order to reach the GoalNode.

   The following maze has been used to test the functionality:

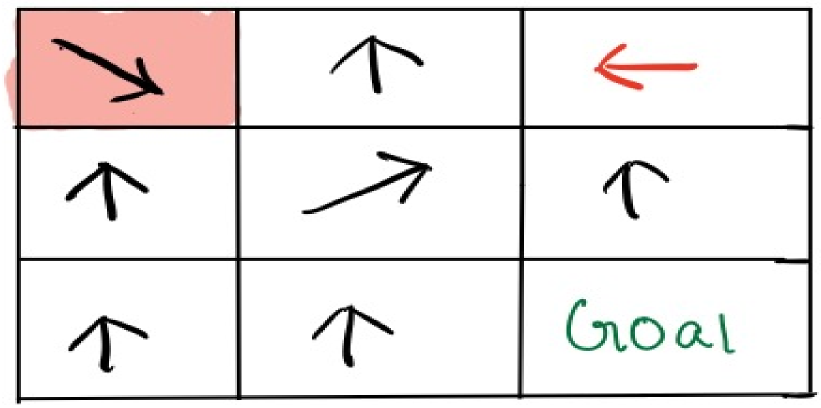

Figure 1: **This maze has been used to test the functionality**

   (a) **Expected Output:**
   (Node: (0, 0), d: 1) → (Node: (1, 1), d: 1) → (Node: (0, 2), d: 1) → (Node: (0, 0), d: 2) → (Node: (2, 2), d: 2)

   (b) **Actual Output:** Found!
   Number of Steps: 5
   1. Node: (0, 0), d: 1
   2. Node: (1, 1), d: 1
   3. Node: (0, 2), d: 1
   4. Node: (0, 0), d: 2
   5. Node: (2, 2), d: 2

4. **Difference Solutions:**

   Checks if the algorithm returns the smallest solution. Here there exists another solution to the maze as well which takes more steps, but the smallest solution is output. The other solution goes from :
   Node: (0, 0), d: 1 → Node: (1, 0), d: 1 → Node: (2, 0), d: 1 →
   Node: (2, 1), d: 1 → Node: (1, 2), d: 1 →

   (Node: (2, 2), d: 1) which is longer.

   (a) **Expected Output:**
   (Node: (0, 0), d: 1) → (Node: (1, 1), d: 1) → (Node: (0, 2), d: 1) →
   (Node: (0, 0), d: 2) → (Node: (2, 2), d: 2)

   (b) **Actual Output:** Found!
   Number of Steps: 5
   1. Node: (0, 0), d: 1
   2. Node: (1, 1), d: 1
   3. Node: (0, 2), d: 1
   4. Node: (0, 0), d: 2
   5. Node: (2, 2), d: 2

Question 2

2a

Assume we have a graph $G = (V, E)$ where $|V| = 2n$ for some $n \in \mathbb{N}$.

Step 1: If $n \leq 2$, then return `False`.

Step 2: Initialise the global variable the variable `num_cycle = 0` to keep track of the number of cycles in the tree, `deg_3_vertex = 0` and `deg_1_vertex = 0` to keep track of the number of vertices encountered with degree 3 and 1 respectively . Additionally, we define a function `adj(i)` which takes a vertex as an argument and **returns 1** if its adjacency list only has one vertex or **returns 3** if 3 vertices else **returns -1** if the adjacent list has 0, 2 or more than 3 vertices.
Note: `adj(i)` makes at most 4 loop iterations. Thus executes in $\mathcal{O}(1)$

Modifications: The algorithm which will be used is a modified version of the `DFS` algorithm provided in the notes. The modifications are described below.

1. Replace the for loop in line 6-8 in `DFS(`G`(V, E))` with the line `DFS-VISIT(`G`, v)`. Effectively, running `DFS-VISIT` on only one vertex.

2. Each time `DFS-VISIT(`G`, u)` is called, where `u` $\in V$, the number of vertices in the adjacency list of `u` is checked with a call to `adj(u)` and the following checks are **added on line 1** of `DFS-VISIT(`G`, u)` .

```
num_vertex = adj(u)
if (num_vertex == 3 and deg_3_vertex < n):
    deg_3_vertex += 1
elif (num_vertex == 1 and deg_1_vertex < n):
    deg_1_vertex += 1
else:
    return False
```

3. Additionally, there would be another addition in `DFS-VISIT(`G`, v)` where there will be an **else** component added to the **if** statement in **line 4** to check if there is a cycle in the tree and if there is exactly one back/forward/cross edge in the tree that too only between vertices of degree 3.

```
    if colour[v] == white:
        parent[v] = u
        DFS-VISIT(G, v)
    else:
        if (num_vertex != 3 and adj(v) != 3) and (num_cycle != 0):
            return False
        else:
```

```
                    num_cycle += 1
```

Please note that `num_vertex = adj(u)`.

Step 3: Pick the first vertex `v` stored $\in V$ and run the modified DFS on the vertex `v`.

Step 4: If the call to DFS in Step 3 does not return False, then check the finish time of `v`. If `f[v] = 4n` then return True else return False.

2b

The algorithm operates on the following deductions:

1. As any graph that is a sun with `2n` vertices must have only one cycle of length n and as each vertex that is a part of the cycle (nicknamed cycle-vertex) is connected to exactly one vertex of degree 1, each cycle-vertex must have degree 3. Thus, as there as n cycle vertices, there must be exactly `n` vertices in the graph with degree 3.
2. Similarly, there must be `n` vertices of degree 1 which the cycle-vertices are connected to. Thus, in total there must n vertices with degree 1 and n vertices with degree 3. Moreover, as there are only 2n vertices there must be no vertex with degree other than 1 or 3.
3. The algorithm ensures that the tree contains only one cycle as there can be at most one non-tree edge and that too between vertices of degree 3. Thus enforcing the constraint of only one cycle in the tree.
4. We also need to ensure that the graph is connected. If we run DFS on a connected graph, the finish time of the starting vertex must always be twice the number of vertices in the graph. Moreover, as in DFS, each vertex has an associated start and finish time, checking if the finish time be twice the number of vertices in the graph, would mean all the vertices in the graph were discovered. Thus, implying the graph is connected.

2c

The analysis is broken up into parts corresponding to steps described in the algorithm:

Part 1: Runs in $\mathcal{O}(1)$

Part 2: Runs in $\mathcal{O}(1)$ as `adj(i)` runs in constant time as noted before.

Modifications: The modified algorithm in the worst case runs $\mathcal{O}(n)$. The aforementioned claim is true as there can be at most 2n calls to `DFS-VISIT(`G`, i)` (because are there only 2n vertices) and each call runs in constant time as the loop in line 3 of the original `DFS-VISIT` would iterate at most 3 times as any vertex with degree more than 3 will fail the degree check added in the modified version of the algorithm and return before the loop is executed.

Part 3: Runs the DFS with the time complexity described above.

Part 4: Runs in $\mathcal{O}(1)$
Thus, the described algorithm runs in $\mathcal{O}(n)$.

Question 3

3a)

Let initial capacity = 10
Let Capacity increment = 10
Insert amortized cost = \$5
Insert actual cost = \$1
Cost of shifting an element = 2 array access = \$2

- After 10 inserts, each of the 10 elements in the array would have a \$4 credit.
- After 20 inserts, each of the first 10 elements would have credit \$2 (as \$2 was used up when shifting to the bigger array on the 11^{th} insert) and each of the last 10 elements will have \$4 credit.
- After 30 inserts, credit in each of:
  - First 10 elements : \$0 (due to shifting)
  - Middle 10 elements : \$2 (due to shifting)
  - Last 10 elements: \$4
- On the 31^{st} insert operation, the additional credit in the last 10 elements is used to shift the first 10 elements.
- Thus after 40 inserts, credit in each of:
  -
    - First 30 elements: \$0 (due to shifting)
    - Last 10 elements: \$4
  - Total credit : \$4 x 10 = \$40 credit
- On the 41^{st} insert, we would need to shift 30 elements which would require minimum \$60 credit. However, as data structure only has \$40 in credit, it will be in debt on the 41^{st} insert.
- Thus, it would require the 41 insert operations to run out.

3b)

Let initial-capacity = b
Let capacity increment = b
Actual cost per array access = \$1
Initial amortized cost = \$3
Amortized cost increment after each b inserts = \$2

Let n $\in \mathbb{N}$ be the number of elements in the array.

<u>**CREDIT INVARIANT:**</u> Each time the array within n $\in \mathbb{N}$ elements needs to be expanded, each of the last b elements would have $2(\lfloor \frac{n}{b} \rfloor)b$ credit.

Proof of credit invariant:

Proof using mathematical induction on the number of times the array has been shifted.

Let k be the number of times the array has been shifted.

<u>**Base case:**</u>

k = 0
If the array has not been expanded even once, the number of elements in the array are less than or equal to b. Thus, the cost of each insert must be \$3, leaving a \$2 credit with each of the inserted elements.

The first time the array will be shifted would be on the (b + 1)th call to insert, at this point there must be b elements in the list with \$2 credit each.

Thus, there would be \$2b credit, since

$$= 2(\lfloor \tfrac{b}{b} \rfloor)b$$

<u>**Inductive Hypothesis:**</u>

Let k $\in \mathbb{N}$
Assume $k > 0$
Assume Credit invariant is true when the array has been expanded k times.

<u>**WTP:**</u> Credit invariant is true when the array needs to be expanded for the $(k+1)^{th}$ time.

As the array has been expanded k times, we know that there must be more than k x b elements in the array, and also that each element inserted into the array after the (k x b)th had an amortized cost of 3 + 2(k), which would leave 2 + 2(k) credit with each element.

Moreover, when the array would need to be expanded for the $(k + 1)^{th}$, there must be $(k+1) * b$ elements in the array and as each of the new b elements inserted into the array stores a credit of 2 + 2k, in total we would have (2 + 2k) * b = 2(k + 1) * b = 2 * number of elements.

Thus, by M1, the CI is true.

By CI,

As every-time the array needs to be expanded, the data structure has 2 * (number of elements) credit. It will never be in debt.

3c)

Initial and increment size = b

Cost of an array access = $1

Let $m \in \mathbb{N}$

In a sequence of m insert operations, there must be m array access to insert an element, and additional cost of 2 x (number of elements) at that point each time the array is shifted.

Thus, the total cost =

$$m + \sum_{n=0}^{\lfloor \frac{m-1}{b} \rfloor} (2)(b)(n)$$

$$m + (2b) \sum_{n=0}^{\lfloor \frac{m-1}{b} \rfloor} (n)$$

$$m + (2b)(0 + 1 + 2 + 3 + ... + \left\lfloor \frac{m-1}{b} \right\rfloor)$$

$$m + (2b)(\frac{\lfloor \frac{m-1}{b} \rfloor (\lfloor \frac{m-1}{b} \rfloor + 1)}{2})$$

$$m + (b)\left\lfloor \frac{m-1}{b} \right\rfloor^2 + (b)\left\lfloor \frac{m-1}{b} \right\rfloor$$

$$\textbf{The Total Cost is in } \Theta(2m + \left\lfloor \frac{(m-1)^2}{b} \right\rfloor - 1)$$

$$\textbf{Thus, it is in } \Theta(m^2)$$

Thus, the amortized complexity of an insert is $\Theta(m)$ on a sequence of m inserts.

Question 4

4a)

`n` = 8

The difference between T1 and T2's implementation is path compression. As path compression is only possible for a tree with height ≥ 2, we need a T1 and T2 to have at-least height 2. In addition, it is not possible to create a tree with height 2 when using union by weight with 3 or less singleton sets. Thus, we need at-least 4 singleton sets which counts as **4 makeset operations**.
Furthermore, we need to make at-least 3 union calls to add all the singleton sets to the same tree, which counts as **3 union operations**. Lastly, let the node in the tree at height 2 be `i` and let the representative/root of the tree be `x`. Now, call the operation `UNION(x, i)` and this would change the node `i's` pointer to `x` in T2 but will not make a change in T1, producing 2 trees that are not equivalent. Thus, in total, making **8 operations**.

4b)

`n = 9`

The difference between T2 and T3's implementation is the property by which they union: weight v/s rank. As weight keep tracks of the number of elements that are in the set tree and rank keeps track of the maximum possible set tree height, calling union on trees of same height with different number of elements can be used to produce trees that are not equivalent as weight would not be the same as the rank.

Firstly, we need to build up two trees with the same height and different number of elements. We cannot use trees of height 0 as they would always have the same number of elements, thus we need trees of height at-least 1. The most efficient way to produce different number of elements is to have a tree with 1 element at height 1 and another tree with 2 elements at height 1. This would require at-least 5 singleton sets and at-least 3 union calls to make 2 trees of height 1 with different number of elements, which counts for **8 (5 make-sets and 3 union) operations**.

At this point, let the representative of the tree with 2 elements be `i`, where `i` weight 2 and rank 1 and let the representative of the tree with 3 elements be `x`, where `x` would have weight 3 and rank 1. Thus, now if we perform `UNION(i, x)`. The representative of the new tree in T2 is `x` and representative of the new tree in T3 is `i`, thus producing trees that are not equivalent. In total, making **9 operations**.