S3952320 Oisin Aeonn

Appendix 1: Theoretical Analysis Complementary

Array-Based Maze

Operations	Worst Case	Best Case
__init__()	$\Theta(n^2)$ occurs if rowNum=colNum, meaning we need to initialise a (2n+2)*(2n+2) grid. Simplified: $(2n+2)^2 \in \Theta(n^2)$.	$\Theta(1)$ occurs if rowNum&colNum=1 meaning we initialise a $(2*1+2)^2$ grid. Simplified 4*4 grid = 16 $\in \Theta(1)$.
initCells()	$\Theta(n^2)$ occurs if rowNum=colNum, as we are calling the allWalls() from our MASTER class which has $\Theta(n^2)$ as it contains two nested loops, iterating from 0 to n-1.	$\Theta(1)$ occurs if addWallFlag=false, meaning the function ends meaning it is $\in \Theta(1)$
addWall()	$\Theta(1)$ occurs since addWall() performs checkCoordinates(), isAdjacent(), performs a calculation in the difference between the rows, and columns of each point, and finally checks if a wall already exists, and then otherwise sets the cell value to True. All of these operations are $\in \Theta(1)$ meaning that addWall() is the same for the worst, and best case.	
removeWall()	Same as above, except it checks if a wall exists, and if it does removes it by setting the cell value to False. Similarly, all of these operations are $\in \Theta(1)$ meaning that removeWall() is the same for the worst, and best case.	
hasWall()	Same as the last two, but it instead just returns True if there is a wall, or False if there is no wall. Again, all of these operations are $\in \Theta(1)$ meaning that hasWall() is the same for the worst, average, and best case.	
neighbours()	checkCoordinates() is run first which creates an empty list and, adds corresponding neighbour cells to the list. These operations are $\in \Theta(1)$, meaning that neighbours() is $\Theta(1)$ for the worst case.	
arrayMaze	init() initialises a 2-D list of size $(2 \times n + 2) \times (2 \times k + 2)$ and fills it with boolean values. This requires iterating over each cell in the 2-D grid, resulting in a time complexity $\in \Theta(n \times k)$. All other methods in the ArrayMaze class perform operations in constant time $\Theta(1)$, and thus aren't relevent as they aren't the dominant term.	init() initialises a 2-D list of size $(2 \times n + 2) \times (2 \times k + 2)$ and fills it with boolean values. This requires iterating over each cell in the 2-D grid, resulting in a time complexity $\in \Theta(n \times k)$. All other methods in the ArrayMaze class perform operations in constant time $\Theta(1)$, and thus aren't relevent as they aren't the dominant term.

Figure 1.1: Time Complexity for Array-Based Maze Implementation

Adjacency List Graph Maze

<table>
<tr><th>Operations</th><th>Worst Case</th><th>Best Case</th></tr>
<tr><td>__init__()</td><td colspan="2">$\Theta(1)$ since all we do to initialise the adjList implementation is create an empty dictionary which is an $\in \Theta(1)$.</td></tr>
<tr><td>addVertex()</td><td colspan="2">$\Theta(1)$ addVertex() checks if the vertex $\in$ adjList{}, if the vertex $\notin$ adjList{} it adds it as a key-value pair. Both of these operations $\in \Theta(1)$.</td></tr>
<tr><td>addVertices()</td><td>$\Theta(n)$ addVertices() loops across all vertices calling addVertex() n times, meaning addVertices() $\in \Theta(n)$.</td><td>$\Theta(1)$ if adjList{} is empty then no loop will be run.</td></tr>
<tr><td>addEdge()</td><td colspan="2">$\Theta(1)$ addEdge() appends the adjacency list when both the coordinates $\in$ adjacencyList as it simply just adds a wall and returns True $\in \Theta(1)$. If either or both coordinates $\notin$ adjacencyList then the function returns False $\in \Theta(1)$.</td></tr>
<tr><td>updateWall()</td><td>updateWall() scans for the edge linking the two vertices in the adjacency list. In the worst case, the edge is at the end or $\notin$ adjacency list. This requires iterating through all the neighbours of both vertices, which can be up to $\Theta(n)$ in a dense graph. Therefore, the worst case time complexity of updateWall() $\in \Theta(n)$.</td><td>updateWall() checks if both vertices $\in$ adjacency list. In the best case, the edge connecting the two vertices is found immediately in the adjacency lists. This implies the method only needs to iterate through a single neighbour. All other operations in the updateWall(), are performed in constant time. Therefore, the best case time complexity of updateWall() $\in \Theta(1)$.</td></tr>
<tr><td>removeEdge()</td><td>$\Theta(n)$ removeEdge() occurs when the two vertices are $\in$ adjList{} and are connected by an Edge. Thus removeEdge() will then need to iterate through the list to find all the neighbors of these vertices to remove them, finally returning True meaning $\in \Theta(n)$</td><td>$\Theta(1)$ occurs when either or both vertices $\notin$ adjList{} meaning it just returns False $\in \Theta(1)$.</td></tr>
<tr><td>hasVertex()</td><td colspan="2">$\Theta(1)$ hasVertex() simply checks if the vertice given is present in the adjList{} $\in \Theta(1)$.</td></tr>
<tr><td>hasEdge()</td><td>$\Theta(n)$ hasEdge() occurs if one coordinate is not in the adjList{} then it must scan through the entire list before returning False $\in \Theta(n)$.</td><td>$\Theta(1)$ hasEdge() occurs when the first vertex is not present in the adjList{} thus it just returns False $\in \Theta(1)$.</td></tr>
<tr><td>getWallStatus()</td><td>$\Theta(n)$ getWallStatus() similarly to above occurs if one coordinate is in the adjList{}, but the second coordinate is not connected to it. It must then scan through the entire list before returning False $\in \Theta(n)$.</td><td>$\Theta(1)$ occurs when the first vertex is not present in the adjList{} thus it just returns False $\in \Theta(1)$.</td></tr>
<tr><td>neighbours()</td><td>Since in the maze implementation a maximum number of 4 neighbours can be present, these would be returned, which is still $\in \Theta(1)$.</td><td>The best case occurs when the coordinate $\notin$ adjList{}, thus it just returns an empty list $\in \Theta(1)$.</td></tr>
<tr><td>adjListGraph</td><td>addVertex() needs to check if the vertex $\in$ adjList{} before adding it which requires the traversal of the entire adjList{}. $\forall$ other methods in the adjListGraph Class $\in \Theta(1)$, thus their time complexities are not relevant because the addVertex() is the dominant term. Therefore, adjListGraph $\in \Theta(n)$ in the worst case.</td><td>addVertex() adds the vertex to adjList{} in constant time $\Theta(1)$ when the vertex $\notin$ adjList{}. $\forall$ other methods in the adjListGraph Class $\in \Theta(1)$, thus their time complexities are not relevant because the addVertex() is the dominant term. Therefore, adjListGraph $\in \Theta(1)$ in the best case.</td></tr>
</table>

Figure 1.2: Time Complexity for Adjacency List Graph Maze

S3952320 Oisin Aeonn

Adjacency Matrix Graph Maze

Operations	Worst Case	Best Case
__init__()	$\Theta(n^2)$ occurs when we need to initialise a adjMatrix[] where rowNum = colNum, where all cells need to be set to $0 \in \Theta(n^2)$.	$\Theta(1)$ occurs when the graph is empty, or very small.
addVertex()	$\Theta(n)$ occurs when there is a new coordinate being added, and the adjacency matrix needs to be expanded by iterating through the existing rows, and adding the new element $\in \Theta(n)$.	$\Theta(1)$ occurs when the coordinate is already present in the list, thus no changes are made $\in \Theta(1)$.
addVertices()	$\Theta(n^2)$ occurs when all of the vertices are new, and the matrix needs to be expanded for each vertex.	$\Theta(n)$ occurs as it calls addVertex() n times for every vertex coordinate in vertex{}.
addEdge()	$\Theta(1)$ addEdge() appends the adjacency matrix when both the coordinates $\in$ vertex{} as it simply just adds a wall and returns True $\in \Theta(1)$. If either or both coordinates $\notin$ vertex{} then the function returns False $\in \Theta(1)$.	
updateWall()	$\Theta(1)$ appends the adjacency matrix when both the coordinates $\in$ vertex{} as it simply just adds a wall and returns True $\in \Theta(1)$. If either or both coordinates $\notin$ vertex{} then the function returns False $\in \Theta(1)$.	
removeEdge()	$\Theta(1)$ appends the adjacency matrix when both the coordinates $\in$ vertex{} as it simply just removes the Edge by setting the cell to 0 and returns True $\in \Theta(1)$. If either or both coordinates $\notin$ vertex{} then it returns False $\in \Theta(1)$.	
hasVertex()	$\Theta(1)$ hasVertex() simply checks if the vertice given is present in the vertex{} $\in \Theta(1)$.	
hasEdge()	$\Theta(1)$ queries the adjacency matrix when both the coordinates $\in$ vertex{} and simply returns the vertex $\in \Theta(1)$. If either or both coordinates $\notin$ vertex{} then it returns False $\in \Theta(1)$.	
updateWall()	If either or both coordinates $\notin$ vertex_indices{}, then the function returns False $\in \Theta(1)$.	When both coordinates $\in$ vertex_indices{} as it simply updates the wall status in the adjacency matrix and returns True $\in \Theta(1)$.
neighbours()	neighbours() is $\Theta(n)$ as it needs to iterate through the corresponding row in the adjacency matrix to find all the neighbours $\in \Theta(n)$.	$\Theta(1)$ occurs when the coordinate $\notin$ vertex_indices{}, thus it just returns an empty list $\in \Theta(1)$.
adjMatGraph	init() in the worst case needs to initialize an adjMatrix[] where rowNum = colNum $\in \Theta(n^2)$. addVertex() $\in \Theta(n^2)$ as it needs to expand the matrix by adding new rows and columns. $\forall$ other methods in the adjMatGraph Class $\in \Theta(1)$ or $\Theta(n)$. Therefore, the overall worst-case time complexity of the adjMatGraph Class $\in \Theta(n^2)$.	init() in the best case still needs to initialise an adjacency matrix of size $(n \times k) \in \Theta(n \times k)$. $\forall$ other methods in the adjMatGraph Class perform operations $\in \Theta(1)$ in the best case. Therefore, the best case of adjMatGraph $\in \Theta(n \times k)$.

Figure 1.3: Time Complexity for Adjacency Matrix Graph Maze