ALGORITHMS PROJECT

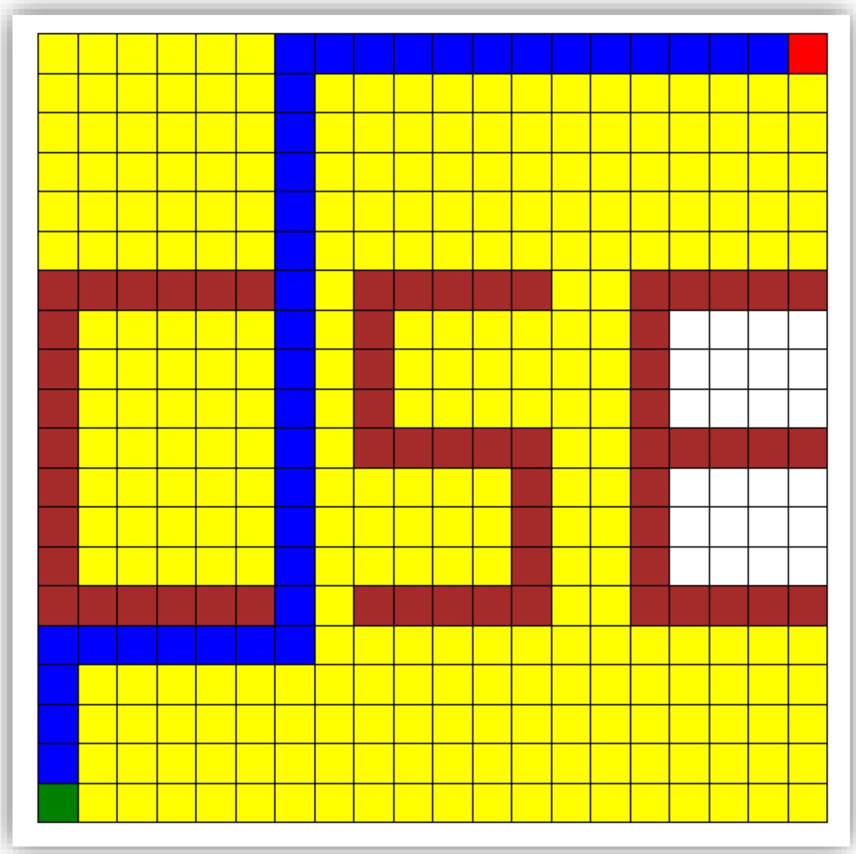
Maze Game

Dr. Al-Shaima Nabil

Team Members

Sec	Name	Code
1	Hassan Mohamed Hassan Ali	20812021101068
2	Mahmoud Galal Ramadan El-Gendy	20812021101328
2	Karim Salah Eldin Abou-mesalam	20812021101453
1	Shehab Shukri AbdelNabi Ibrahim	20812021101006
2	Mohamed Gehad Hussien Metwally	20812021100025
2	Menna Allah Essam Ahmed Salem	20812021200937
1	Shahd AbdelNabi Mahmoud AbdelNabi	20812021200898

Maze Game



A maze game involves navigating a complex maze from start to finish without colliding with obstacles. This is achieved through specific algorithms such as depth-first search, breadth-first search, and Dijkstra.

ANALYSIS

• Breadth-First Search (BFS)

```
const visited = new Set();
 const queue = [[startRow, startCol]];
 const startTime = performance.now();
 let n = 0;
  // Number of cells in the shortest path
 let m = 0;
 // Total number of neighbors
 let yellowVisitedCount = 0;
 // Number of visited cells that are yellow
```

Initially, queue and set were used to implement this algorithm accurately and correctly.

Complexity: O(1)

• processQueue()

```
if (row === endRow && col === endCol) {
        let currentRow = row;
        let currentCol = col;
        while (currentRow !== startRow || currentCol !== startCol) {
            const cell = document.getElementById(`${currentRow}-${currentCol}`);
            if (currentRow !== endRow || currentCol !== endCol){
              cell.classList.add("path");
              ++n;
            }

            const [prevRow, prevCol] = JSON.parse(cell.dataset.prev);
            currentRow = prevRow;
            currentCol = prevCol;

        }
        const endTime = performance.now();
        const runningTime = endTime - startTime; // Calculate the running time
        showData(n,yellowVisitedCount - (n), m, (runningTime / 1000).toFixed(2));
        return;
    }
```

Total running time for this loop:

Iteration * Body

N * 1

Complexity: O(N)

```
visited.add(`${row}-${col}`);
      const neighbors = getNeighbors(row, col);
      for (const [r, c] of neighbors) {
          ++m;
          const key = `${r}-${c}`;
          if (!visited.has(key) && grid[r][c] !== 1) {
              visited.add(key);
              queue.push([r, c]);
              const cell = document.getElementById(`${r}-${c}`);
              if (r !== endRow || c !== endCol) {
                  cell.classList.add("visited");
              }
              cell.dataset.prev = JSON.stringify([row, col]);
              if (cell.classList.contains("visited")) {
                yellowVisitedCount++;
            }
          }
      }
```

Total running time for this loop:

Iteration * Body

M * 1

Complexity: O(M)

Total Running Time: T(n)= (N) + (M) + (constant)

Complexity: O(N+M)

N: the number of cells in the specific path.

M: the neighbors of these cells.

• Depth-First Search (DFS)

```
const visited = new Set();
  const stack = [[startRow, startCol]];
  const startTime = performance.now();
  let n=0;
  let m=0;
  let yellowVisitedCount = 0;
```

In the beginning, stack, and set were used to implement this algorithm accurately and correctly.

Complexity: O(1)

• processStack()

```
if (stack.length === 0) {
        return;
      }

      const [row, col] = stack.pop();
```

If it is empty, this means that the path was specified by the verb, and it emerges from the function.
Else out the element at the top of the stack and then check.

Complexity: O(1)

```
if (row === endRow && col === endCol) {
      let currentRow = row;
      let currentCol = col;
      while (currentRow !== startRow || currentCol !== startCol) {
        const cell = document.getElementById(`${currentRow}-${currentCol}`);
        if (currentRow !== endRow || currentCol !== endCol){
          cell.classList.add("path");
          ++n;
        }

        const [prevRow, prevCol] = JSON.parse(cell.dataset.prev);
        currentRow = prevRow;
        currentCol = prevCol;

      }
      const endTime = performance.now();
      const runningTime = endTime - startTime; // Calculate the running time
      showData(n, yellowVisitedCount - (n), m, (runningTime / 1000).toFixed(2));
      return;
    }
```

Total running time for this loop and complexity

Iteration * Body

N * 1

Complexity: O(N)

```
visited.add(`${row}-${col}`);
    const neighbors = getNeighbors(row, col);
    for (const [r, c] of neighbors) {
      ++m;
      const key = `${r}-${c}`;
      if (!visited.has(key) && grid[r][c] !== 1) {
        visited.add(key);
        stack.push([r, c]);
        const cell = document.getElementById(`${r}-${c}`);
        if (r !== endRow || c !== endCol){
          cell.classList.add("visited");
          ++yellowVisitedCount;
        }
        cell.dataset.prev = JSON.stringify([row, col]);
      }

      // No need to continue exploring neighbor cells if you have reached the End Cell
      if (r == endRow && c == endCol) {
        while (stack.isEmpty === false) stack.pop();
        stack.push([r, c]);
        break;
      }
    }
```

Total running time for this loop:

Iteration * Body

M * 1

Complexity: O(M)

Total Running Time: T(n)= (N) + (M) + (constant)

Complexity: O(N+M)

N: the number of cells in the specific path.

M: the neighbors of these cells.

• Dijkstra

```
const distances = {};
const visited = new Set();
const pq = new PriorityQueue();
const startTime = performance.now();
let n=0;
let m=0;
let yellowVisitedCount = 0;
```

The Total Running Time of This Block:
1 (constant)

Complexity: O(1)

```
for (let i = 0; i < numRows; i++) {
    for (let j = 0; j < numCols; j++) {
      distances[`${i}-${j}`] = Infinity;
    }
  }
```

The Total Running Time of This Block:
(Row * Col) + Constant

Complexity: O(Row*Col)

• processStep()

```
if (pq.isEmpty() || visited.size === numRows * numCols) {
      return;
    }

    const current = pq.pop().element;

    const [row, col] = current.split("-").map(Number);

    if (visited.has(current)) return;
    visited.add(current);
```

The Total Running Time of This Block:
1 (constant)

Complexity: O(1)

```
if (row === endRow && col === endCol) {
      let currentRow = row;
      let currentCol = col;
      while (currentRow !== startRow || currentCol !== startCol) {
        const cell = document.getElementById(`${currentRow}-${currentCol}`);
        if (currentRow !== endRow || currentCol !== endCol) {
          cell.classList.add("path");
          ++n;
        }

        const [prevRow, prevCol] = JSON.parse(cell.dataset.prev);
        currentRow = prevRow;
        currentCol = prevCol;

      }
      const endTime = performance.now();
      const runningTime = endTime - startTime; // Calculate the running time
      showData(n, yellowVisitedCount - (n), m, (runningTime / 1000).toFixed(2));
      return;
    }
```

The Total Running Time of This Block: N + constant

Complexity: O(N)

```
for (const [r, c] of neighbors) {
  ++m;
  if (!visited.has(`${r}-${c}`) && grid[r][c] !== 1) {
    const distance = distances[`${row}-${col}`] + 1;
    if (distance < distances[`${r}-${c}`]) {
      distances[`${r}-${c}`] = distance;
      pq.push(`${r}-${c}`, distance);
      const cell = document.getElementById(`${r}-${c}`);
      if (r !== endRow || c !== endCol) {
        cell.classList.add("visited");
        ++yellowVisitedCount;
      }
      cell.dataset.prev = JSON.stringify([row, col]);
    }

  }
  // give the highest priority to end cell ,
  //no need to explore more cells
  if (r === endRow && c === endCol) {
    pq.push(`${r}-${c}`, -1);
    break;
  }
}
```

The Total Running Time of This Block: (M) + constant

Complexity: O(M)

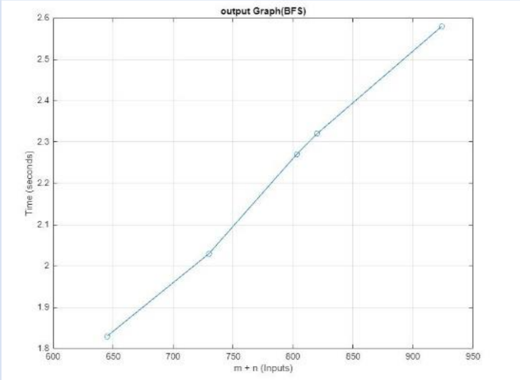
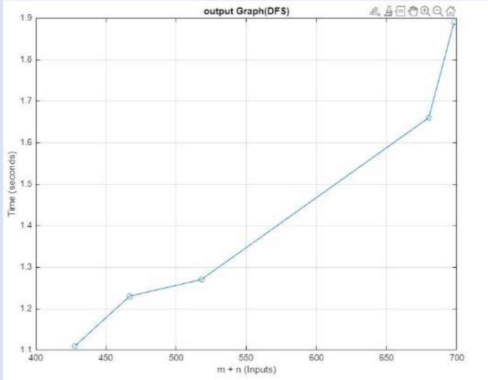
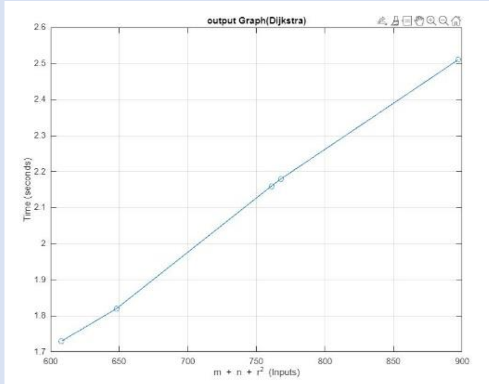
Total Running Time: T(n)= (Row*Col) + (M) + (N) + (constant)

Complexity: O(Row* Col)

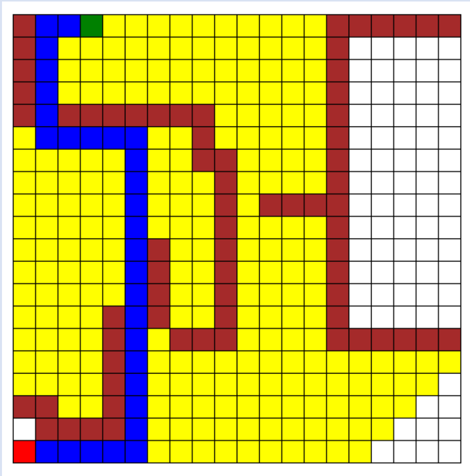
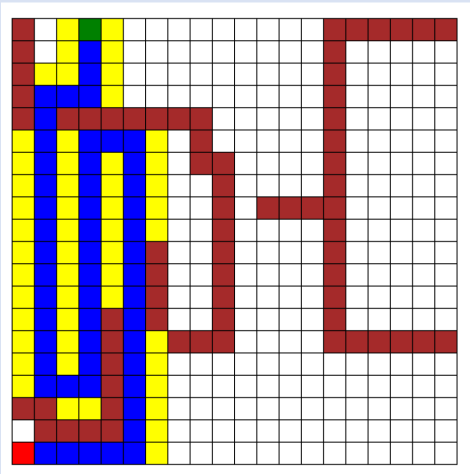
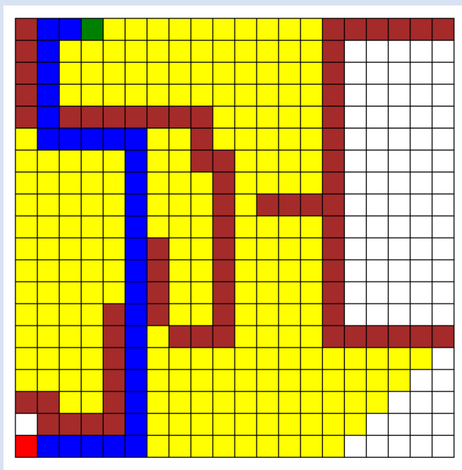
N: the number of cells in the specific path.

M: the neighbors of these cells.

COMPARISONS

	BFS					DFS					Dijkstra				
Graph															
N	55	19	21	21	21	123	81	51	45	51	55	19	21	21	21
M	869	626	799	782	709	557	437	647	422	377	842	589	740	747	627
M + N	924	645	820	803	730	680	518	698	467	428	897	608	761	768	648
T(N)	2.58	1.83	2.32	2.27	2.03	1.66	1.27	1.89	1.23	1.11	2.51	1.73	2.16	2.18	1.82
WORST	O(N+M)					O(N+M)					O(N*2)				
Average	O(N+M)					O(N+M)					O(N*2)				
Best	O(1)					O(1)					O(N*2)				
Stable	Un-Stable					Un-Stable					Un-Stable				
In-Place	No					No					No				

• For Same Obstacles:

	BFS	DFS	Dijkstra
Maze			
No. of cells (path)	29	51	29
No. of cells (visited)	254	102	249
No. of cells (Neighbors)	961	212	941
Running Time	2.76	0.59	2.71

We Obtain the Total Running Time of Three Algorithms As shown, we Note That the DFS is faster than the other algorithms for horizontal End (Start and end point tend to be horizontal) because the mechanism of DFS is (horizontal then Vertical then horizontal), If we change the obstacle so that the end point tends to be Vertical to the start point, we find that BFS is faster.

You can download the code by scanning the following QR code:

SCAN ME