METHODOLOGY AND IMPLEMENTAION

2.1 Floodfill Methodology and Implementation

Pathfinding.cs

```
Queue<Tile> FloodFill(Tile start, Tile goal)
{
    Dictionary<Tile, Tile> nextTileToGoal = new Dictionary<Tile,
Tile>();
    Queue<Tile> frontier = new Queue<Tile>();
    List<Tile> visited = new List<Tile>();

    frontier.Enqueue(goal);

    while(frontier.Count > 0)
    {
        Tile curTile = frontier.Dequeue();

        foreach(Tile neighbor in _mapGenerator.Neighbors(curTile))
        {
            if (visited.Contains(neighbor) == false &&
frontier.Contains(neighbor) == false)
            {
                if (neighbor._TileType != Tile.TileType.Wall)
                {
                    frontier.Enqueue(neighbor);
                    nextTileToGoal[neighbor] = curTile;
                }
            }
        }
        visited.Add(curTile);
    }

    if (visited.Contains(start) == false)
        return null;

    Queue<Tile> path = new Queue<Tile>();
    Tile curPathTile = start;
    while(curPathTile != goal)
    {
        curPathTile = nextTileToGoal[curPathTile];
        path.Enqueue(curPathTile);
    }

    return path;
}
```

The algorithm starts by initializing a dictionary nextTileToGoal and a queue frontier. The nextTileToGoal dictionary is used to keep track of the next tile to visit to get to the goal tile, and the frontier queue is used to keep track of the tiles that need to be visited.

The frontier queue is initialized with the goal tile, and then the algorithm enters a loop that continues until the frontier queue is empty. In each iteration of the loop, the algorithm dequeues a tile from the frontier queue, and then checks each of its neighbors to see if they can be added to the frontier queue.

If a neighbor has not been visited before and is not already in the frontier queue, and it is not a wall tile, it is added to the frontier queue and its parent is stored in the nextTileToGoal dictionary.

The algorithm keeps track of visited tiles in the visited list. Once the algorithm has visited all the tiles that can be reached from the goal tile, it checks if the start tile has been visited. If the start tile has not been visited, it means that there is no path from the start tile to the goal tile, and the algorithm returns null.

If the start tile has been visited, the algorithm constructs the path from the start tile to the goal tile by following the parent tiles stored in the nextTileToGoal dictionary. The path is stored in a queue path, which is then returned by the algorithm.

2.2 Dijkstra Methodology and Implementation

Pathfinding.cs

```
Queue<Tile> Dijkstra(Tile start, Tile goal)
{
    Dictionary<Tile, Tile> NextTileToGoal = new Dictionary<Tile,
Tile>();//Determines for each tile where you need to go to reach the
goal. Key=Tile, Value=Direction to Goal
    Dictionary<Tile, int> costToReachTile = new Dictionary<Tile,
int>();//Total Movement Cost to reach the tile

    PriorityQueue<Tile> frontier = new PriorityQueue<Tile>();
    frontier.Enqueue(goal, 0);
    costToReachTile[goal] = 0;

    while (frontier.Count > 0)
    {
        Tile curTile = frontier.Dequeue();
        if (curTile == start)
            break;

        foreach (Tile neighbor in _mapGenerator.Neighbors(curTile))
        {
            int newCost = costToReachTile[curTile] + neighbor._Cost;
            if (costToReachTile.ContainsKey(neighbor) == false ||
newCost < costToReachTile[neighbor])
            {
                if (neighbor._TileType != Tile.TileType.Wall)
                {
                    costToReachTile[neighbor] = newCost;
                    int priority = newCost;
                    frontier.Enqueue(neighbor, priority);
                    NextTileToGoal[neighbor] = curTile;
                    neighbor._Text =
costToReachTile[neighbor].ToString();
                }
            }
        }
    }

    //Get the Path

    //check if tile is reachable
    if (NextTileToGoal.ContainsKey(start) == false)
    {
        return null;
    }

    Queue<Tile> path = new Queue<Tile>();
    Tile pathTile = start;
    while (goal != pathTile)
    {
        pathTile = NextTileToGoal[pathTile];
        path.Enqueue(pathTile);
    }
    return path;
}
```

This algorithm implements Dijkstra's shortest path algorithm for finding the shortest path between two points on a tile-based map. The input to the algorithm is the starting tile and the goal tile. The algorithm uses a priority queue to keep track of the tiles with the lowest movement cost, and it uses two dictionaries: "NextTileToGoal" and "costToReachTile".

The "NextTileToGoal" dictionary determines for each tile where you need to go to reach the goal, and the "costToReachTile" dictionary keeps track of the total movement cost to reach the tile.

The algorithm starts by enqueueing the goal tile in the priority queue with a priority of 0 and a cost of 0. It then continues to dequeue tiles from the priority queue until it reaches the starting tile or until there are no more tiles left in the queue.

For each dequeued tile, the algorithm examines each of its neighboring tiles and calculates the movement cost to reach each neighbor tile. If the neighbor tile has not been visited yet or if the new cost to reach it is lower than the previous cost, the algorithm updates the "NextTileToGoal" and "costToReachTile" dictionaries for the tile and enqueues it in the priority queue with its new priority value.

Once the algorithm has found the shortest path to the starting tile, it backtracks through the "NextTileToGoal" dictionary to find the path and returns it as a queue of tiles. If the starting tile is unreachable, the algorithm returns null.

2.3 A* Methodology and Implementation

Pathfinding.cs

```
Queue<Tile> AStar(Tile start, Tile goal)
{
    Dictionary<Tile, Tile> NextTileToGoal = new Dictionary<Tile,
Tile>();//Determines for each tile where you need to go to reach the
goal. Key=Tile, Value=Direction to Goal
    Dictionary<Tile, int> costToReachTile = new Dictionary<Tile,
int>();//Total Movement Cost to reach the tile

    PriorityQueue<Tile> frontier = new PriorityQueue<Tile>();
    frontier.Enqueue(goal, 0);
    costToReachTile[goal] = 0;

    while (frontier.Count > 0)
    {
        Tile curTile = frontier.Dequeue();
        if (curTile == start)
            break;

        foreach (Tile neighbor in _mapGenerator.Neighbors(curTile))
        {
            int newCost = costToReachTile[curTile] + neighbor._Cost;
            if (costToReachTile.ContainsKey(neighbor) == false ||
newCost < costToReachTile[neighbor])
            {
                if (neighbor._TileType != Tile.TileType.Wall)
                {
                    costToReachTile[neighbor] = newCost;
                    int priority = newCost + Distance(neighbor,
start);
                    frontier.Enqueue(neighbor, priority);
                    NextTileToGoal[neighbor] = curTile;
                    neighbor._Text =
costToReachTile[neighbor].ToString();
                }
            }
        }
    }

    //Get the Path

    //check if tile is reachable
    if (NextTileToGoal.ContainsKey(start) == false)
    {
        return null;
    }

    Queue<Tile> path = new Queue<Tile>();
    Tile pathTile = start;
    while (goal != pathTile)
    {
        pathTile = NextTileToGoal[pathTile];
        path.Enqueue(pathTile);
    }
    return path;
}
```

This is an implementation of the A* algorithm for finding the shortest path between two points on a grid. It uses a priority queue to explore the most promising paths first, based on an estimate of the remaining cost to the goal (heuristic). In this case, the heuristic is the straight-line distance between the current tile and the goal. The algorithm also keeps track of the total cost of reaching each tile, which is the sum of the cost of moving from the start tile to the current tile and the heuristic cost to the goal. Finally, it backtracks from the goal tile to the start tile using the NextTileToGoal dictionary to construct the shortest path.

2.4 Q-Learning Methodology and Implementation

```
Queue<Tile> QLearning(Tile start, Tile goal)
{
    // Set hyperparameters
    float learningRate = 0.1f;
    float discountFactor = 0.99f;
    int maxEpisodes = 10000;
    int maxStepsPerEpisode = 100;

    // Create the Q-Table
    Dictionary<Tile, Dictionary<Tile, float>> qTable = new
Dictionary<Tile, Dictionary<Tile, float>>();
    foreach (Tile tile in _mapGenerator.GetAllTiles())
    {
        qTable[tile] = new Dictionary<Tile, float>();
        foreach (Tile action in _mapGenerator.Neighbors(tile))
        {
            qTable[tile][action] = 0f;
        }
    }

    // Run episodes
    for (int episode = 0; episode < maxEpisodes; episode++)
    {
        Tile curTile = start;
        int step = 0;
        while (curTile != goal && step < maxStepsPerEpisode)
        {
            // Choose the action with the highest Q-value
            Tile nextTile = null;
            float maxQValue = float.MinValue;
            foreach (Tile neighbor in
_mapGenerator.Neighbors(curTile))
            {
                if (qTable[curTile][neighbor] > maxQValue)
                {
                    maxQValue = qTable[curTile][neighbor];
                    nextTile = neighbor;
                }
            }

            // Update the Q-value of the current state-action pair
            float reward = 0f;
            if (nextTile == goal)
            {
                reward = 100f;
            }
            float oldQValue = qTable[curTile][nextTile];
            float newQValue = oldQValue + learningRate * (reward +
discountFactor * maxQValue - oldQValue);
            qTable[curTile][nextTile] = newQValue;

            // Move to the next state
            curTile = nextTile;
            step++;
        }
    }

    // Follow the policy determined by the Q-table to get the path
    Queue<Tile> path = new Queue<Tile>();
    path.Enqueue(start);
    Tile curPathTile = start;
    while (curPathTile != goal)
    {
        Tile nextPathTile = null;
        float maxQValue = float.MinValue;
        foreach (Tile neighbor in
_mapGenerator.Neighbors(curPathTile))
        {
            if (qTable[curPathTile][neighbor] > maxQValue)
            {
                maxQValue = qTable[curPathTile][neighbor];
                nextPathTile = neighbor;
            }
        }

        curPathTile = nextPathTile;
        path.Enqueue(curPathTile);
    }

    return path;
}
```

This code implements the Q-learning algorithm to find a path from a start tile to a goal tile in a map. The Q-learning algorithm is a reinforcement learning algorithm that learns an optimal policy by updating Q-values for state-action pairs. The Q-value represents the expected cumulative reward for taking a particular action in a particular state.

The code initializes a Q-table that stores the Q-values for each state-action pair. The Q-values are initially set to zero. Then, the code runs a specified number of episodes, where each episode consists of taking actions and updating Q-values based on the observed rewards. In each step of an episode, the code chooses the action with the highest Q-value and moves to the next state. After each action, the code updates the Q-value for the current state-action pair based on the observed reward and the maximum Q-value for the next state.

After all episodes have been run, the code follows the policy determined by the Q-table to generate a path from the start tile to the goal tile. The policy is determined by choosing the action with the highest Q-value in each state until the goal tile is reached. The resulting path is returned as a queue of tiles.