Contents

1 **Introduction** **2**

2 **Classes** **2**

2.1 UnitedWeStandSearch Class . . . 2
2.2 GenericSearch Class . . . 2
2.3 Organism Class . . . 3
2.4 Node . . . 3

3 **Grid Generation** **4**

4 **Searching** **4**

4.1 Reshape Function . . . 5
4.2 Expansion of a Node . . . 6
4.3 void move(int[] coordinates, int[] action) . . . 6
4.4 Goal Test . . . 7

5 **Strategies** **7**

5.1 Breadth-First Search . . . 7
5.2 Depth-First Search . . . 7
5.3 Uniform Cost Search . . . 7
5.4 Iterative Deepening Search . . . 7
5.5 Greedy Search . . . 8
5.6 A* Search . . . 8

6 **Heuristics** **8**

6.1 First Heuristic . . . 8
6.2 Second Heuristic . . . 8

7 **Path Reconstruction** **9**

8 **Performance** **9**

1 Introduction

This project involves constructing a grid with randomly scattered organisms and obstacles. The objective is to find the optimal path of movement for each organism to ensure continuous collisions, resulting in the formation of a single large organism. The organisms can move in any of the four cardinal directions: north, south, east, and west.

Key elements of the simulation include:

Randomly Scattered Organisms: The grid will initially have organisms placed at random positions. Obstacles: There will be obstacles on the grid that organisms cannot pass through, adding complexity to their movement. Chemical Barrier: The grid will be surrounded by a chemical barrier that kills any organism that collides with it. Collision and Merging: When two organisms collide, they merge into a single larger organism. The primary goal is to develop an algorithm that determines the optimal sequence of moves for each organism to ensure they eventually collide and merge into one large organism, while avoiding obstacles and the deadly chemical barrier.

2 Classes

2.1 UnitedWeStandSearch Class

The `UnitedWeStandSearch` class contains the `main` method that calls the `search` method. This class extends the `GenericSearch` class, which includes the `search` method. Additionally, the `UnitedWeStandSearch` class contains the `genGrid` method, which generates the grid with organisms and obstacles.

2.2 GenericSearch Class

The `GenericSearch` class has attributes for the row, column, and grid, along with additional attributes to store the organisms placed on the grid. It includes an `ArrayList` named *visited* to store the previously visited (explored) nodes, the total cost, and the total number of nodes chosen for expansion after finding the solution.

The output of the `genGrid()` function is sent to the `search` method in the format `search(String grid, String strategy, Boolean visualize)`. In this method, the grid is converted to a proper 2D array using the `stringToGrid` function. This function splits the input string with a semicolon (`;`), using the first two elements as the column (width) and row (height). It then performs another split with a comma (`,`) to extract the coordinates of the organisms, with the first element representing the column and the second element representing the row. These coordinates are inserted into the grid with positive unique values that will function as IDs. The same process is applied to obstacles, but with negative values.

2.3 Organism Class

The `Organism` class represents organisms with the following attributes: `ID`, `size` (indicating the number of micro-organisms that constitute the organism, initially set to 1), `row`, and `column` (representing the coordinates of the organism on the grid). All placed organisms are stored in an `ArrayList` named `organismList` within the `reshape` method for later use.

2.4 Node

At the beginning of each search strategy, a node is created with the following structure:

- *grid*: Represents the initial state as a `HashMap` mapping each organism and obstacle to its coordinates.
- *organismList*: An `ArrayList` representing the remaining organisms on the grid so far.
- *parent*: An attribute of type `Node` representing the parent of the current node (previous state), initially set to `null` in the root.
- *action*: An integer array of size 2 with the first element pointing to the ID of the organism to be moved in this path, and the second element identifying the direction as either 1, 2, 3, or 4 (north, south, east, and west respectively).
- *cost*: An integer representing the accumulative path cost to arrive at this particular node, initially set to 0 and then accumulated as 1 move in any direction costs 1.
- *isDead*: A boolean initially set to `false` but updated to `true` if a node hits a barrier (out of boundary). In this case, it will be a dead node and won't be considered while expanding possible node solutions.

Methods:

- `int organismCheck`: Takes organism coordinates and the grid in the form of a `HashMap` and checks if the coordinates of the organism collide with those of other organisms. Returns the ID of the organism that the given organism will collide with, or `-1` if no collision occurs.
- `boolean obstacleCheck`: Takes the coordinates of the organism and the grid in the form of a `HashMap`, and returns `true` if a collision will occur between the given organism and any obstacle in the grid.
- `boolean boundaryCheck`: Takes the organism coordinates and checks if they are out of grid bounds, returning `true` if so.
- `void move`: This critical method moves an organism based on the given coordinates and action.

The `move` method is explained as follows:

In simple terms, the `move` method works as follows:

1. It takes the current coordinates of the organism and the action (which specifies the direction to move).
2. It calculates the new coordinates based on the specified direction.
3. It checks if the new coordinates are within the grid boundaries. If not, the node is marked as dead.
4. It checks if the new coordinates collide with an obstacle. If so, it updates the organism's position, increases the path cost, and breaks the loop.
5. It checks if the new coordinates collide with another organism. If so, it merges the organisms, removes the collided organism from the list, updates the position and size of the remaining organism, increases the path cost, and breaks the loop.
6. If no collisions occur, it updates the current coordinates and continues moving in the specified direction.

3 Grid Generation

The grid generation process involves creating a 2D array that represents the environment where organisms will be placed. Each cell in this grid can be empty, contain an organism, or hold an obstacle. The function `genGrid()` within the `UnitedWeStandSearch` class is responsible for constructing this 2D array.

The grid is generated with random dimensions, with a maximum limit of 10x10 cells. Within this grid, up to 20 organisms are placed randomly, along with several obstacles. The function also generates a string that contains the coordinates of the grid elements in the following format:
`width;height;organism_coordinates;obstacle_coordinates`. This string is used by other functions for further processing.

4 Searching

The generated string output from `genGrid()` represents the grid configuration and is then passed to a search function. This search function allows specifying the search *strategy* to be used. The available strategies are:

- `BF`: Breadth-First Search
- `DF`: Depth-First Search
- `UC`: Uniform Cost Search
- `ID`: Iterative Deepening Search

- `GRi`: Greedy Search, with $i \in \{1, 2\}$ distinguishing between two heuristics
- `ASi`: A* Search, with $i \in \{1, 2\}$ distinguishing between two heuristics

Additionally, a boolean parameter *visualize* can be set to `true` to display the grid after each suggested action. The search function outputs the suggested path to be taken in order to achieve the goal of merging all organisms into a single large organism According to the parameter `strategy`, different methods are called to perform the search based on the specified strategy. The common scenario across all strategies is as follows:

1. An object of type `Node` is created to represent the root, which contains the initial state of the grid and the initial state of the `organismList` that contains all the organisms.
2. This root node is then stored in a data structure, such as a `Deque` or a `PriorityQueue`, depending on the algorithm used.
3. The algorithm iterates over this data structure until it is empty. In each iteration, a node is popped from the data structure.
4. The `expand` method is used to generate all children nodes of the current node.
5. For each child node, the `move` method is called to update the node based on the action to be performed. The updated nodes are then placed back into the data structure if they are valid (i.e., not redundant and not in a failure state such as hitting a boundary).
6. During each iteration, after generating the children, the algorithm checks if any child node has reached the goal. If so, the last node expanded is returned as the result.

4.1 Reshape Function

The output of `genGrid()` is a string that represents the locations of obstacles and organisms within the 2D grid array. To facilitate efficient retrieval of organism locations, it is preferable to store only the locations of obstacles and organisms in a `HashMap` so in `search(String grid, String Strategy,boolean visualize)` method,the string representation of the grid is first converted to a 2D array through the `StringToGrid` method. This 2D array is then passed to the `reshape` function to convert it to `HashMap`. In this map, the keys represent organism IDs as positive integers (ranging from 1 to the total number of organisms remaining), and obstacles as negative integers (ranging from -1 to the total number of obstacles). The values of the `HashMap` are arrays of size two, representing the row and column coordinates.

```
Map<Integer, int[]> coordinatesMap = reshape(UnitedWeStandSearch.stringToGrid(gridG));
```

4.2 Expansion of a Node

In the `GenericSearch` class, there is a method called `expand` that takes a `Node` object as a parameter. This method is invoked by the `search` method when a node is popped from the data structure and its children need to be generated.

After creating the initial node (root), a data structure, such as a `Deque` or `PriorityQueue`, is used to store nodes that are to be expanded. The choice of data structure depends on the search strategy. Nodes must be checked before being added to the queue using the `checkAddState(Node` *`toBeAddedNode`*`)` method. This method verifies if the node has been expanded before through different paths by hashing its grid using gridHash function and comparing it to hashed values of elements in an already defined `ArrayList` named `visited`. If the node is found in `visited`, it is not added again to the queue. If not found, it is added to both the expansion queue and the `visited ArrayList`.

The queue is explored iteratively by popping one node at a time and adding its children. Each child results from moving each organism in one of the four directions, leading to a maximum of $4\times$ the number of organisms in child nodes.

In the `expand(Node n)` method, the `organismList` provided in the node to be expanded is cloned, along with the grid of the node, to ensure that modifications in the child node do not affect the original list and grid of the parent node. A new node is then cloned to act as the child, initially copying the parent node but with changes in the action array that identifies the organism to be moved and the direction of movement. Subsequently, another method in the `Node` class is called to track the movement.

4.3 void move(int[] coordinates, int[] action)

This method takes the original coordinates of the organism that is to be moved along with the action array that specifies the ID of the organism and the direction to move. It updates the coordinates of the organism according to the action. The method moves the organism in the specified direction one cell at a time, counting the number of moves to later calculate the path cost. This process continues until one of the following three cases occurs:

- **Out of Boundary**: If the organism's coordinates exceed the grid's chemical boundaries the `isDead` variable is set to `true`, and the node resulting from this move is considered dead.

- **Organism Collision**: If the organism occupies a cell already containing another organism, a collision occurs.

- **Obstacle Collision**: If the organism collides with an obstacle, the organism's coordinates are updated to the position before hitting the obstacle. The cost is updated based on the number of moves taken to reach this position, multiplied by the size of the micro-organisms it consists of. The node is then added to the queue for further expansion if it is not the goal state.

4.4 Goal Test

For each node that is popped from the data structure, a goal test is performed before further expanding the node. This test checks whether the node has reached the goal by examining the size of its `organismList`. If the size of the `organismList` is equal to 1, it indicates that all organisms have been merged into a single organism, thus reaching the goal state.

In this case, the total cost of the expansion is set to the cost associated with the node. The last node, which represents the goal state, is returned. To reconstruct the path that led to the goal, the `parent` attribute of each node is used to trace back through the sequence of nodes from the goal to the initial state.

5 Strategies

5.1 Breadth-First Search

The Breadth-First Search algorithm is implemented with an integer parameter to distinguish it from the Depth-First Search algorithm. If this parameter is set to 1, the algorithm performs a breadth-first search by popping nodes from the front of the queue. This ensures that nodes are explored level by level. The number of nodes expanded is tracked and returned at the end of the process. Nodes are expanded using the `expand` method if they do not meet the goal test.

5.2 Depth-First Search

If the integer parameter is set to any value other than 1, the algorithm performs a Depth-First Search. In this case, nodes are popped from the end of the queue, exploring nodes in a depth-first manner.

5.3 Uniform Cost Search

In Uniform Cost Search, nodes are managed using a priority queue to regulate the expansion based on their cost. The `Node` class implements the `Comparable` interface, and the `compareTo` method is overridden to prioritize nodes with lower costs. Consequently, nodes with the lowest cost are expanded first.

5.4 Iterative Deepening Search

The Iterative Deepening Search method involves looping over levels of expansion, starting from 1 up to a maximum depth, which is set to a relatively high value (e.g., 1000) to ensure thorough exploration. At each depth level, a new root node is created, and the `visited ArrayList` is cleared. The node, with a depth of 0 and a limit set to the current maximum depth, is passed to a helper method for further exploration and expansion using recursion. If the node does not meet the goal test and the current depth does not match the limit, nodes are expanded and the depth is incremented until the goal is found. If the goal is not reached, `null` is returned.

5.5 Greedy Search

In Greedy Search, the priority of nodes is determined by two implemented heuristic functions. Nodes are added to a priority queue, and the `compare` method of the priority queue is modified to prioritize nodes with lower heuristic values.

5.6 A* Search

The A* Search algorithm is similar to the Greedy Search but also considers the cost of the node. Nodes are prioritized based on the sum of the node's cost and the value of the heuristic function. The node with the lowest total (cost + heuristic) is given higher priority.

6 Heuristics

6.1 First Heuristic

The first heuristic function is based on the number of organisms remaining in the node after merging. Nodes with fewer organisms are considered more favorable. This is determined by examining the `organismList` attribute of each node, which contains the remaining organisms.

Admissibility: This heuristic is admissible because it always gives a number that is less than or equal to the actual cost needed to reach the goal. It counts the remaining organisms, so it only looks at how many merges are still needed. Since you can't merge organisms in fewer steps than there are merges left, this number will always be the smallest possible estimate of the work left to do. Therefore, it never overestimates the true cost.

6.2 Second Heuristic

The second heuristic function evaluates the proximity of organisms on the grid to be merged. A priority queue named `minDistance` is used to store the minimum distances between each unique pair of organisms. For each organism in the `organismList`, the heuristic function calculates the minimum Manhattan distance to all other organisms. The largest distance is excluded because merging n organisms requires $n - 1$ moves. The resulting sum of all the minimum distances is used as the heuristic value for the node.

For example, if the `organismList` contains the organisms {1, 3, 6, 5}, the heuristic calculates the distance between organism 1 and all other organisms, then selects the minimum distance. This process is repeated for each organism in the list. Finally, the sum of all minimum distances is calculated, excluding the largest distance.

Admissibility: This heuristic is admissible because it provides an estimate of the cost to reach the goal that is always less than or equal to the actual cost. The reasoning is as follows:

1. **Minimum Distance Calculation**: The heuristic calculates the minimum Manhattan distance between each pair of organisms. The Manhattan distance measures the shortest path between two points on a grid by only considering vertical and horizontal moves.
2. **Pairwise Comparison**: For each organism, the heuristic determines the smallest distance to any other organism. This ensures that only the shortest, most direct path is considered for merging each pair of organisms.
3. **Excluding the Largest Distance**: When merging n organisms, you need $n - 1$ moves. By excluding the largest distance, the heuristic accurately represents the minimal necessary moves to merge all organisms. This prevents overestimating the cost since you can't merge all organisms without making at least $n - 1$ moves.
4. **Lower Bound Guarantee**: Since the heuristic only sums the smallest necessary distances and excludes the largest one, it always provides an estimate that is less than or equal to the actual cost. This means it gives a safe, conservative estimate of the work left to do.

7 Path Reconstruction

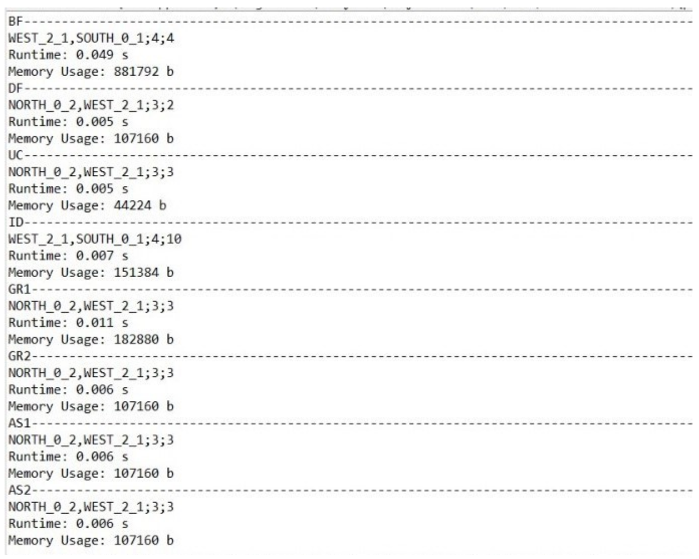
After the goal is found (if it exists), the returned node is used to reconstruct the path that led to it, primarily utilizing the `parent` attribute, which stores its parent node, and the `action` array, which stores both the organism that was moved and the direction it moved.

The reconstruction process begins by creating an empty `ArrayList` named `path` to store the path. For each node, the organism ID of the moved organism is extracted, then used to retrieve its coordinates from the parent node (before it moved). The direction is obtained using a helper method that maps each integer from 1 to 4 to the four directions (North, South, East, and West, respectively). This process is repeated iteratively for each parent node, effectively tracing the path back to the initial state.

Since this iterative process results in the path being recorded in reverse order, we use `Collections.reverse(path)` to correct the order. Finally, the total cost and the number of explored nodes are printed. If no node is returned as an answer to the search, the method returns "nosolution" as the final answer.

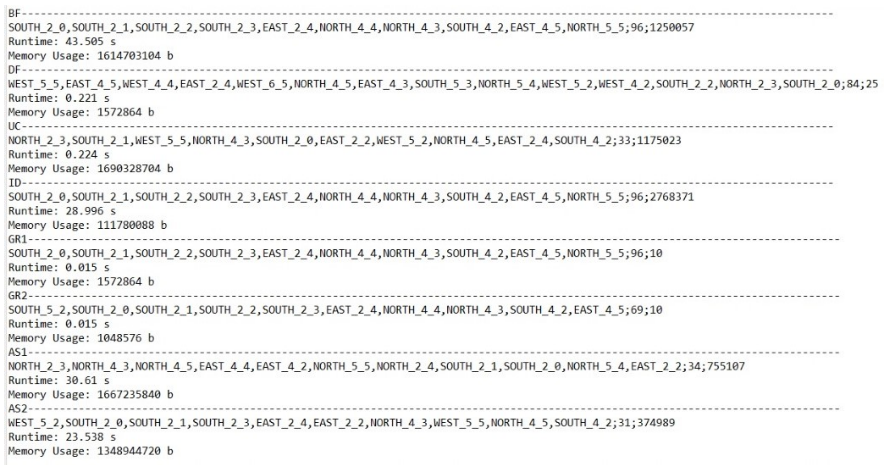
8 Performance

The following section provides snapshots of the memory usage and runtime for executing the search method on two different grids: Grid 0 and Grid 3. Each grid is evaluated using the specified search strategies.



```
BF------------------------------------------------------------------------------------
WEST_2_1,SOUTH_0_1;4;4
Runtime: 0.049 s
Memory Usage: 881792 b
DF------------------------------------------------------------------------------------
NORTH_0_2,WEST_2_1;3;2
Runtime: 0.005 s
Memory Usage: 107160 b
UC------------------------------------------------------------------------------------
NORTH_0_2,WEST_2_1;3;3
Runtime: 0.005 s
Memory Usage: 44224 b
ID------------------------------------------------------------------------------------
WEST_2_1,SOUTH_0_1;4;10
Runtime: 0.007 s
Memory Usage: 151384 b
GR1-----------------------------------------------------------------------------------
NORTH_0_2,WEST_2_1;3;3
Runtime: 0.011 s
Memory Usage: 182880 b
GR2-----------------------------------------------------------------------------------
NORTH_0_2,WEST_2_1;3;3
Runtime: 0.006 s
Memory Usage: 107160 b
AS1-----------------------------------------------------------------------------------
NORTH_0_2,WEST_2_1;3;3
Runtime: 0.006 s
Memory Usage: 107160 b
AS2-----------------------------------------------------------------------------------
NORTH_0_2,WEST_2_1;3;3
Runtime: 0.006 s
Memory Usage: 107160 b
```

Figure 1: grid0



```
BF------------------------------------------------------------------------------------------------------------
SOUTH_2_0,SOUTH_2_1,SOUTH_2_2,SOUTH_2_3,EAST_2_4,NORTH_4_4,NORTH_4_3,SOUTH_4_2,EAST_4_5,NORTH_5_5;96;1250057
Runtime: 43.505 s
Memory Usage: 1614703104 b
DF------------------------------------------------------------------------------------------------------------
WEST_5_5,EAST_4_5,WEST_4_4,EAST_2_4,WEST_6_5,NORTH_4_5,EAST_4_3,SOUTH_5_3,NORTH_5_4,WEST_5_2,WEST_4_2,SOUTH_2_2,NORTH_2_3,SOUTH_2_0;84;25
Runtime: 0.221 s
Memory Usage: 1572864 b
UC------------------------------------------------------------------------------------------------------------
NORTH_2_3,SOUTH_2_1,WEST_5_5,NORTH_4_3,SOUTH_2_0,EAST_2_2,WEST_5_2,NORTH_4_5,EAST_2_4,SOUTH_4_2;33;1175023
Runtime: 0.224 s
Memory Usage: 1690328704 b
ID------------------------------------------------------------------------------------------------------------
SOUTH_2_0,SOUTH_2_1,SOUTH_2_2,SOUTH_2_3,EAST_2_4,NORTH_4_4,NORTH_4_3,SOUTH_4_2,EAST_4_5,NORTH_5_5;96;2768371
Runtime: 28.996 s
Memory Usage: 111780088 b
GR1-----------------------------------------------------------------------------------------------------------
SOUTH_2_0,SOUTH_2_1,SOUTH_2_2,SOUTH_2_3,EAST_2_4,NORTH_4_4,NORTH_4_3,SOUTH_4_2,EAST_4_5,NORTH_5_5;96;10
Runtime: 0.015 s
Memory Usage: 1572864 b
GR2-----------------------------------------------------------------------------------------------------------
SOUTH_5_2,SOUTH_2_0,SOUTH_2_1,SOUTH_2_2,SOUTH_2_3,EAST_2_4,NORTH_4_4,NORTH_4_3,SOUTH_4_2,EAST_4_5;69;10
Runtime: 0.015 s
Memory Usage: 1048576 b
AS1-----------------------------------------------------------------------------------------------------------
NORTH_2_3,NORTH_4_3,NORTH_4_5,EAST_4_4,EAST_4_2,NORTH_5_5,NORTH_2_4,SOUTH_2_1,SOUTH_2_0,NORTH_5_4,EAST_2_2;34;755107
Runtime: 30.61 s
Memory Usage: 1667235840 b
AS2-----------------------------------------------------------------------------------------------------------
WEST_5_2,SOUTH_2_0,SOUTH_2_1,SOUTH_2_3,EAST_2_4,EAST_2_2,NORTH_4_3,WEST_5_5,NORTH_4_5,SOUTH_4_2;31;374989
Runtime: 23.538 s
Memory Usage: 1348944720 b
```

Figure 2: grid3