GROUP 3 CAPSTONE PROJECT

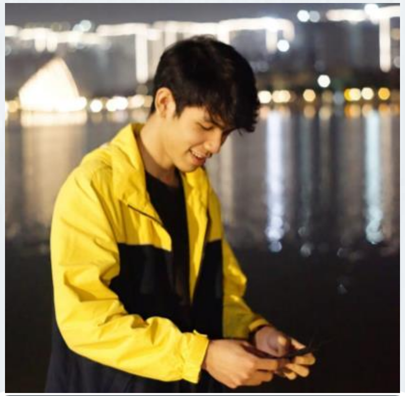
AI GARDENING PROBLEM

Members

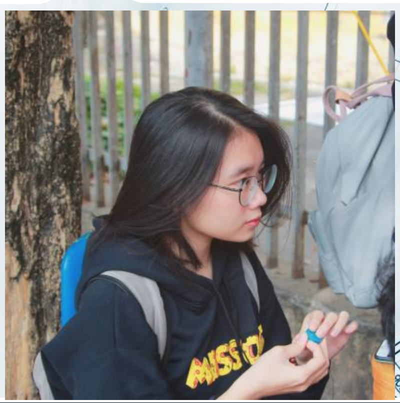
Đỗ Tuấn Minh
20200390

Nguyễn Kim Tuyến
20205196

Hoàng Huy Chiến
20200084



Tô Thái Dương
20205180



Đinh Ngọc Hạnh Trang
20204928

CONTENTS

PROBLEM DESCRIPTION

There is a m×n square garden. We denote the location of a square by (x, y). Each square contains exactly one of the following:

- Scarecrow (Denoted by 0)
- Warehouse (Only at (0, 0), denoted by 1)
- Newly planted seed (Denoted by 2)
- Live tree (Denoted by 3)
- Dead tree (Denoted by 4)

y \ x	0	1	2
0	1	2	3
1	4	0	2
2	4	0	3

PROBLEM DESCRIPTION

An AI machine, starts at the warehouse (0, 0), has following functionalities:

- If the AI machine gets to the newly planted seed, it will water it
- If the AI machine gets to the live tree, it will pick all the fruit and bring it back to the warehouse.
- If the AI machine gets to the dead tree, it will remove the dead tree, bring it back to the warehouse, then take a new seed to the previous spot, plant and water it (plant and water at the same time).
- The AI machine can not go pass the scarecrow.

y \ x	0	1	2
0		2	3
1	4	0	2
2	4	0	3

PROBLEM DESCRIPTION

The goal is to:

- Water all the newly planted seed
- Bring all the fruit back to the warehouse
- Bring all the dead trees back to the warehouse.
- Plant and water all the new tree in the dead tree spot.

With a minimum number of actions.

y \ x	0	1	2
0		2	3
1	4	0	2
2	4	0	3

PROBLEM DESCRIPTION

The goal is to:

- Water all the newly planted seed
- Bring all the fruit back to the warehouse
- Bring all the dead trees back to the warehouse.
- Plant and water all the new tree in the dead tree spot.

With a minimum number of actions.

If it is not possible, print out “Can not find the path”.

y \ x	0	1	2
0		2	3
1	4	2	0
2	4	0	3

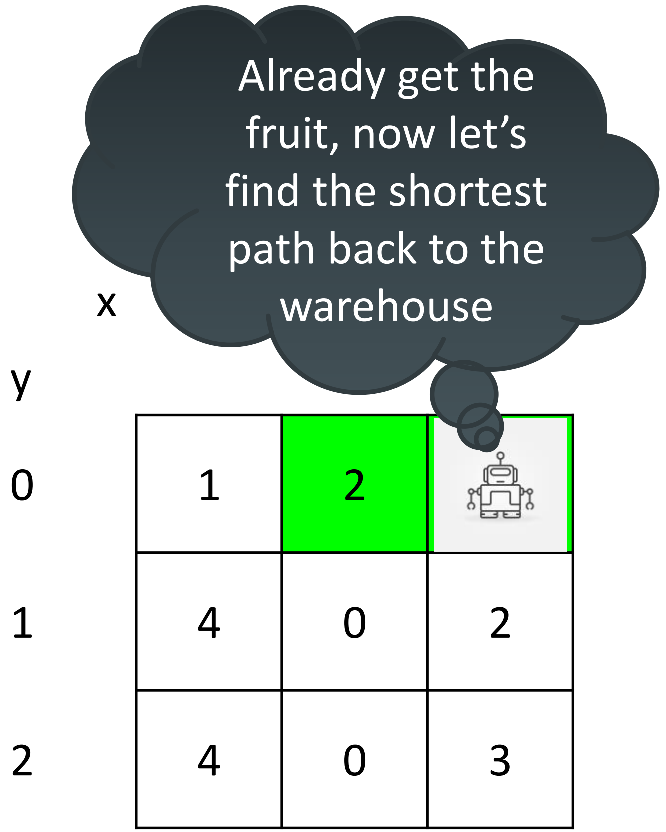
DETAILED DESCRIPTION OF THE PROBLEM

The environment is known, partially observable, deterministic, dynamic and discrete. Thus, the solution is always a fixed sequence of actions.

There are two main parts in this problem to deal with:

- Searching: to find the shortest path in the matrix (m×n square garden).
- Traversing: to visit and do the task in every cell (except the scarecrow).

So, we have to combine at least two algorithms.



A* SEARCH ALGORITHM FOR SEARCHING

```
function heuristic(a, b)
    return abs(a[0] – b[0]) + abs(a[1] – b[1])
function init()
    OPEN = {}
    CLOSED = {}
    add start to OPEN
    set g_cost of start to 0
    set h_cost of start to heuristic(start, target)
    set f_cost of start to g_cost(start) + h_cost(start)
```

A* SEARCH ALGORITHM FOR SEARCHING

We choose heuristic function is the Manhattan distance (the sum of the absolute differences between the two vectors) between two nodes because:

- It is admissible
- Since we can only move the blocks 1 at a time and in only one of 4 directions (right, left, top, bottom), the optimal scenario for each block is that it has a clear, unobstructed path to its goal state

```
function heuristic(a, b)
    return abs(a[0] – b[0]) + abs(a[1] – b[1])
function init()
    OPEN = {}
    CLOSED = {}
    add start to OPEN
    set g_cost of start to 0
    set h_cost of start to heuristic(start, target)
    set f_cost of start to g_cost(start) + h_cost(start)
```

A* SEARCH ALGORITHM FOR SEARCHING

We choose heuristic function is the Manhattan distance (the sum of the absolute differences between the two vectors) between two nodes because:

- It is admissible
- Since we can only move the blocks 1 at a time and in only one of 4 directions (right, left, top, bottom), the optimal scenario for each block is that it has a clear, unobstructed path to its goal state

Initialize a set to store nodes to be traversed (OPEN) and a set to store already traverse nodes (CLOSED).

```
function heuristic(a, b)
    return abs(a[0] – b[0]) + abs(a[1] – b[1])
function init()
    OPEN = {}
    CLOSED = {}
    add start to OPEN
    set g_cost of start to 0
    set h_cost of start to heuristic(start, target)
    set f_cost of start to g_cost(start) + h_cost(start)
```

A* SEARCH ALGORITHM FOR SEARCHING

We choose heuristic function is the Manhattan distance (the sum of the absolute differences between the two vectors) between two nodes because:

- It is admissible
- Since we can only move the blocks 1 at a time and in only one of 4 directions (right, left, top, bottom), the optimal scenario for each block is that it has a clear, unobstructed path to its goal state

```
function heuristic(a, b)
    return abs(a[0] – b[0]) + abs(a[1] – b[1])
```

Initialize a set to store nodes to be traversed (OPEN) and a set to store already traverse nodes (CLOSED).

Then, set all cost to node start.

```
function init()
    OPEN = {}
    CLOSED = {}
    add start to OPEN
    set g_cost of start to 0
    set h_cost of start to heuristic(start, target)
    set f_cost of start to g_cost(start) + h_cost(start)
```

A* SEARCH ALGORITHM FOR SEARCHING

```
function astar(start, target)
    init()
    while OPEN is not empty
        current = node in OPEN with lowest f_cost
        remove current from OPEN
        if current is target
            return
        add current to CLOSED
        foreach neighbor of current
        temp_cost = g_cost(current) + 1
        if neighbor is not Obstacle
            if neighbor in OPEN or CLOSED and g_cost(neighbor) <= temp_cost
                continue
            if neighbor in CLOSED
                remove neighbor from CLOSED
                add neighbor to OPEN
            if neighbor not in OPEN or CLOSED
                add neighbor to OPEN
                set f_cost of neighbor
                set parent of neighbor to current
    return FALSE
```

A* SEARCH ALGORITHM FOR SEARCHING

For each loop:

- Check if node current is the target or not

```
function astar(start, target)
    init()
    while OPEN is not empty
        current = node in OPEN with lowest f_cost
        remove current from OPEN
        if current is target
            return
        add current to CLOSED
        foreach neighbor of current
        temp_cost = g_cost(current) + 1
        if neighbor is not Obstacle
            if neighbor in OPEN or CLOSED and g_cost(neighbor) <= temp_cost
                continue
            if neighbor in CLOSED
                remove neighbor from CLOSED
                add neighbor to OPEN
            if neighbor not in OPEN or CLOSED
                add neighbor to OPEN
                set f_cost of neighbor
                set parent of neighbor to current
    return FALSE
```

A* SEARCH ALGORITHM FOR SEARCHING

For each loop:

- Check if node current is the target or not
- Temporarily store the new g_cost of neighbor one larger than that of current (the distance of two adjacent nodes is one).

```
function astar(start, target)
    init()
    while OPEN is not empty
        current = node in OPEN with lowest f_cost
        remove current from OPEN
        if current is target
            return
        add current to CLOSED
        foreach neighbor of current
        temp_cost = g_cost(current) + 1
        if neighbor is not Obstacle
            if neighbor in OPEN or CLOSED and g_cost(neighbor) <= temp_cost
                continue
            if neighbor in CLOSED
                remove neighbor from CLOSED
                add neighbor to OPEN
            if neighbor not in OPEN or CLOSED
                add neighbor to OPEN
                set f_cost of neighbor
                set parent of neighbor to current
    return FALSE
```

A* SEARCH ALGORITHM FOR SEARCHING

For each loop:

- Check if node current is the target or not
- Temporarily store the new g_cost of neighbor one larger than that of current (the distance of two adjacent nodes is one).
- Compare the old and new g_cost to make decision about neighbor.

```
function astar(start, target)
    init()
    while OPEN is not empty
        current = node in OPEN with lowest f_cost
        remove current from OPEN
        if current is target
            return
        add current to CLOSED
        foreach neighbor of current
        temp_cost = g_cost(current) + 1
        if neighbor is not Obstacle
            if neighbor in OPEN or CLOSED and g_cost(neighbor) <= temp_cost
                continue
            if neighbor in CLOSED
                remove neighbor from CLOSED
                add neighbor to OPEN
            if neighbor not in OPEN or CLOSED
                add neighbor to OPEN
                set f_cost of neighbor
                set parent of neighbor to current
    return FALSE
```

A* SEARCH ALGORITHM FOR SEARCHING

For each loop:

- Check if node current is the target or not
- Temporarily store the new g_cost of neighbor one larger than that of current (the distance of two adjacent nodes is one).
- Compare the old and new g_cost to make decision about neighbor.
- If neighbor is neither in two sets (so haven't has g_cost yet), add it to OPEN, set all cost to it and set its parent node in the path.

```
function astar(start, target)
    init()
    while OPEN is not empty
        current = node in OPEN with lowest f_cost
        remove current from OPEN
        if current is target
            return
        add current to CLOSED
        foreach neighbor of current
        temp_cost = g_cost(current) + 1
        if neighbor is not Obstacle
            if neighbor in OPEN or CLOSED and g_cost(neighbor) <= temp_cost
                continue
            if neighbor in CLOSED
                remove neighbor from CLOSED
                add neighbor to OPEN
            if neighbor not in OPEN or CLOSED
                add neighbor to OPEN
                set f_cost of neighbor
                set parent of neighbor to current
    return FALSE
```

A* SEARCH ALGORITHM FOR SEARCHING

For each loop:

- Check if node current is the target or not
- Temporarily store the new g_cost of neighbor one larger than that of current (the distance of two adjacent nodes is one).
- Compare the old and new g_cost to make decision about neighbor.
- If neighbor is neither in two sets (so haven't has g_cost yet), add it to OPEN, set all cost to it and set its parent node in the path.
- Return FALSE when can not find a path

```
function astar(start, target)
    init()
    while OPEN is not empty
        current = node in OPEN with lowest f_cost
        remove current from OPEN
        if current is target
            return
        add current to CLOSED
        foreach neighbor of current
        temp_cost = g_cost(current) + 1
        if neighbor is not Obstacle
            if neighbor in OPEN or CLOSED and g_cost(neighbor) <= temp_cost
                continue
            if neighbor in CLOSED
                remove neighbor from CLOSED
                add neighbor to OPEN
            if neighbor not in OPEN or CLOSED
                add neighbor to OPEN
                set f_cost of neighbor
                set parent of neighbor to current
    return FALSE
```

BFS TO SEARCHING

```
function BFS(start, target)
    VISITED = []
    QUEUE
    add start to QUEUE and VISITED
    set parent of start to None
    while QUEUE is not empty
        remove front from QUEUE store it to current
        if current is target
            return path
    foreach neighbor of current
        if neighbor not in VISITED
            add neighbor to QUEUE
            set parent of neighbor to current
            add neighbor to VISITED
```

BFS TO SEARCHING

- Make a list to store visited nodes (VISITED) to prevent loops and a QUEUE.

```
function BFS(start, target)
    VISITED = []
    QUEUE
    add start to QUEUE and VISITED
    set parent of start to None
    while QUEUE is not empty
        remove front from QUEUE store it to current
        if current is target
            return path
    foreach neighbor of current
        if neighbor not in VISITED
            add neighbor to QUEUE
            set parent of neighbor to current
            add neighbor to VISITED
```

BFS TO SEARCHING

- Make a list to store visited nodes (VISITED) to prevent loops and a QUEUE.
- Node start doesn't have parent

```
function BFS(start, target)
    VISITED = []
    QUEUE
    add start to QUEUE and VISITED
    set parent of start to None
    while QUEUE is not empty
        remove front from QUEUE store it to current
        if current is target
            return path
    foreach neighbor of current
        if neighbor not in VISITED
            add neighbor to QUEUE
            set parent of neighbor to current
            add neighbor to VISITED
```

BFS TO SEARCHING

- Make a list to store visited nodes (VISITED) to prevent loops and a QUEUE.
- Node start doesn't have parent
- Check if current is target or not

```
function BFS(start, target)
    VISITED = []
    QUEUE
    add start to QUEUE and VISITED
    set parent of start to None
    while QUEUE is not empty
        remove front from QUEUE store it to current
        if current is target
            return path
    foreach neighbor of current
        if neighbor not in VISITED
            add neighbor to QUEUE
            set parent of neighbor to current
            add neighbor to VISITED
```

BFS TO SEARCHING

- Make a list to store visited nodes (VISITED) to prevent loops and a QUEUE.
- Node start doesn't have parent
- Check if current is target or not
- Add unvisited neighbor to QUEUE, VISITED and set its parent.

```
function BFS(start, target)
    VISITED = []
    QUEUE
    add start to QUEUE and VISITED
    set parent of start to None
    while QUEUE is not empty
        remove front from QUEUE store it to current
        if current is target
            return path
    foreach neighbor of current
        if neighbor not in VISITED
            add neighbor to QUEUE
            set parent of neighbor to current
            add neighbor to VISITED
```

DFS FOR TRAVERSING

```
DFS(s)
    update PERCEPT SEQUENCE
    if visited(s) == TRUE:
        return
    visited(s) = TRUE
    if s is Obstacle:
        return
    if distance(last_visited_node, s) != 1
        add shortest_path(last_visited_node, s) to PATH
    else
        add s to PATH
    add move(PATH) to ACTION
    add do_task(s) to ACTION
    print PERCEPT SEQUENCE and ACTION
    foreach neighbor of s
        if visited(neighbor) == FALSE
            DFS(neighbor)
forall cell in MATRIX
    visited(cell) = FALSE
forall cell in MATRIX
    DFS(cell)
```

DFS FOR TRAVERSING

- Update the matrix that the AI machine sees

```
DFS(s)
    update PERCEPT SEQUENCE
    if visited(s) == TRUE:
        return
    visited(s) = TRUE
    if s is Obstacle:
        return
    if distance(last_visited_node, s) != 1
        add shortest_path(last_visited_node, s) to PATH
    else
        add s to PATH
    add move(PATH) to ACTION
    add do_task(s) to ACTION
    print PERCEPT SEQUENCE and ACTION
    foreach neighbor of s
        if visited(neighbor) == FALSE
            DFS(neighbor)
forall cell in MATRIX
    visited(cell) = FALSE
forall cell in MATRIX
    DFS(cell)
```

DFS FOR TRAVERSING

- Update the matrix that the AI machine sees
- If DFS a cell that is not adjacent to the latest one, then find the shortest path between them.

```
DFS(s)
    update PERCEPT SEQUENCE
    if visited(s) == TRUE:
        return
    visited(s) = TRUE
    if s is Obstacle:
        return
    if distance(last_visited_node, s) != 1
        add shortest_path(last_visited_node, s) to PATH
    else
        add s to PATH
    add move(PATH) to ACTION
    add do_task(s) to ACTION
    print PERCEPT SEQUENCE and ACTION
    foreach neighbor of s
        if visited(neighbor) == FALSE
            DFS(neighbor)
forall cell in MATRIX
    visited(cell) = FALSE
forall cell in MATRIX
    DFS(cell)
```

DFS FOR TRAVERSING

- Update the matrix that the AI machine sees
- If DFS a cell that is not adjacent to the latest one, then find the shortest path between them.
- Update ACTION needed to get to the cell through PATH and to finish tasks in the cell.

```
DFS(s)
    update PERCEPT SEQUENCE
    if visited(s) == TRUE:
        return
    visited(s) = TRUE
    if s is Obstacle:
        return
    if distance(last_visited_node, s) != 1
        add shortest_path(last_visited_node, s) to PATH
    else
        add s to PATH
    add move(PATH) to ACTION
    add do_task(s) to ACTION
    print PERCEPT SEQUENCE and ACTION
    foreach neighbor of s
        if visited(neighbor) == FALSE
            DFS(neighbor)
forall cell in MATRIX
    visited(cell) = FALSE
forall cell in MATRIX
    DFS(cell)
```

DFS FOR TRAVERSING

- Update the matrix that the AI machine sees
- If DFS a cell that is not adjacent to the latest one, then find the shortest path between them.
- Update ACTION needed to get to the cell through PATH and to finish tasks in the cell.
- Print out the PERCEPT SEQUENCE and list of ACTION for each run.

```
DFS(s)
    update PERCEPT SEQUENCE
    if visited(s) == TRUE:
        return
    visited(s) = TRUE
    if s is Obstacle:
        return
    if distance(last_visited_node, s) != 1
        add shortest_path(last_visited_node, s) to PATH
    else
        add s to PATH
    add move(PATH) to ACTION
    add do_task(s) to ACTION
    print PERCEPT SEQUENCE and ACTION
    foreach neighbor of s
        if visited(neighbor) == FALSE
            DFS(neighbor)
forall cell in MATRIX
    visited(cell) = FALSE
forall cell in MATRIX
    DFS(cell)
```

DFS FOR TRAVERSING

- Update the matrix that the AI machine sees
- If DFS a cell that is not adjacent to the latest one, then find the shortest path between them.
- Update ACTION needed to get to the cell through PATH and to finish tasks in the cell.
- Print out the PERCEPT SEQUENCE and list of ACTION for each run.
- Continue run DFS on unvisited neighbors

```
DFS(s)
    update PERCEPT SEQUENCE
    if visited(s) == TRUE:
        return
    visited(s) = TRUE
    if s is Obstacle:
        return
    if distance(last_visited_node, s) != 1
        add shortest_path(last_visited_node, s) to PATH
    else
        add s to PATH
    add move(PATH) to ACTION
    add do_task(s) to ACTION
    print PERCEPT SEQUENCE and ACTION
    foreach neighbor of s
        if visited(neighbor) == FALSE
            DFS(neighbor)
forall cell in MATRIX
    visited(cell) = FALSE
forall cell in MATRIX
    DFS(cell)
```

DFS FOR TRAVERSING

- Update the matrix that the AI machine sees
- If DFS a cell that is not adjacent to the latest one, then find the shortest path between them.
- Update ACTION needed to get to the cell through PATH and to finish tasks in the cell.
- Print out the PERCEPT SEQUENCE and list of ACTION for each run.
- Continue run DFS on unvisited neighbors
- Visited check is initial set FALSE to all cells

```
DFS(s)
    update PERCEPT SEQUENCE
    if visited(s) == TRUE:
        return
    visited(s) = TRUE
    if s is Obstacle:
        return
    if distance(last_visited_node, s) != 1
        add shortest_path(last_visited_node, s) to PATH
    else
        add s to PATH
    add move(PATH) to ACTION
    add do_task(s) to ACTION
    print PERCEPT SEQUENCE and ACTION
    foreach neighbor of s
        if visited(neighbor) == FALSE
            DFS(neighbor)
forall cell in MATRIX
    visited(cell) = FALSE
forall cell in MATRIX
    DFS(cell)
```

DFS FOR TRAVERSING

- Update the matrix that the AI machine sees
- If DFS a cell that is not adjacent to the latest one, then find the shortest path between them.
- Update ACTION needed to get to the cell through PATH and to finish tasks in the cell.
- Print out the PERCEPT SEQUENCE and list of ACTION for each run.
- Continue run DFS on unvisited neighbors
- Visited check is initial set FALSE to all cells
- Run DFS on all cell to make sure no cell is unvisited.

```
DFS(s)
    update PERCEPT SEQUENCE
    if visited(s) == TRUE:
        return
    visited(s) = TRUE
    if s is Obstacle:
        return
    if distance(last_visited_node, s) != 1
        add shortest_path(last_visited_node, s) to PATH
    else
        add s to PATH
    add move(PATH) to ACTION
    add do_task(s) to ACTION
    print PERCEPT SEQUENCE and ACTION
    foreach neighbor of s
        if visited(neighbor) == FALSE
            DFS(neighbor)
forall cell in MATRIX
    visited(cell) = FALSE
forall cell in MATRIX
    DFS(cell)
```

SUMMARY OF ALGORITHMS

Criterion	A* search	BFS	DFS
Complete?	Yes	Yes	No
Time complexity	Depends on the heuristic	O(b^n)	O(b^m)
Space complexity	O(b^d)	O(b^d)	O(bm)
Optimal?	No	Yes	No

Q&A?

THANK YOU FOR LISTENING