# BBM405 – Fundamentals of Artificial Intelligence

# Homework 1

Burak ÖZÜESEN – 21827761

## Part 1: Generate your own maze using Randomized DFS

In this part of my work, I implemented an iterative randomized dfs algorithm. Because a large number of programs were starting to create memory errors.

While coding this section, I defined a 2-dimensional array in the given dimensions. Later, thanks to the functions I created, I filled it with "Vertex" type objects. The relevant screenshots are as follows.

```python
def createMaze():
    startVertex = grid[0][0]
    randomizedDFS(startVertex)
    return
```

```python
def randomizedDFS(vertex):
    myStack = []
    vertex.visited = True
    myStack.append(vertex)
    while len(myStack) != 0:
        currentVertex = myStack[-1]
        del myStack[-1]
        nextVertex = randomUnvisitedNeighbour(currentVertex)
        if nextVertex is not None:
            myStack.append(currentVertex)
            currentVertex.connectedCells.append(nextVertex)
            nextVertex.connectedCells.append(currentVertex)

            x_of_edge = abs(nextVertex.row_number - currentVertex.row_number)
            y_of_edge = abs(nextVertex.column_number - currentVertex.column_number)
            x_of_edge += min(nextVertex.row_number, currentVertex.row_number)*2
            y_of_edge += min(nextVertex.column_number, currentVertex.column_number)*2

            printableMaze[x_of_edge][y_of_edge] = 1
            printableMaze[currentVertex.row_number*2][currentVertex.column_number*2] = 1
            printableMaze[nextVertex.row_number*2][nextVertex.column_number*2] = 1

            nextVertex.visited = True
            myStack.append(nextVertex)
```

```python
def randomUnvisitedNeighbour(vertex):
    randomArray = []

    if vertex.column_number != 0:
        leftNeighbour = grid[vertex.row_number][vertex.column_number - 1]
        if not leftNeighbour.visited:
            randomArray.append(leftNeighbour)

    if vertex.row_number != 0:
        topNeighbour = grid[vertex.row_number - 1][vertex.column_number]
        if not topNeighbour.visited:
            randomArray.append(topNeighbour)

    if vertex.column_number != cols - 1:
        rightNeighbour = grid[vertex.row_number][vertex.column_number + 1]
        if not rightNeighbour.visited:
            randomArray.append(rightNeighbour)

    if vertex.row_number != rows - 1:
        bottomNeighbour = grid[vertex.row_number + 1][vertex.column_number]
        if not bottomNeighbour.visited:
            randomArray.append(bottomNeighbour)

    if len(randomArray) == 0:
        return None
    random.shuffle(randomArray)

    return randomArray[0]
```

Another point I want to mention here is how I can visualize after the Grid creation part is over. For this part, I have defined a new 2-dimensional array that will only appear and will not be processed, but this time its dimensions are different. For example, our Grid size is 10x10 and the size of our new maze is 19x19 (2n-1 * 2n-1). This is because I want to determine the non-passable elements by evaluating the elements between 2 vertex as if they were a vertex.

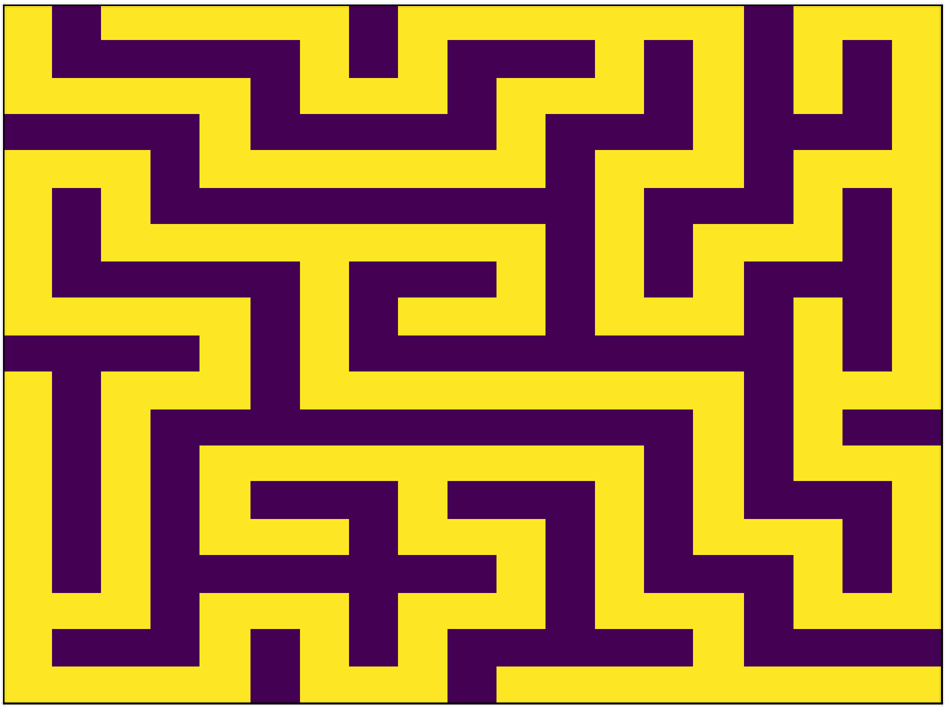
The values I assign to my printableMaze variable in the randomized dfs algorithm you see above are all about this. It does not interfere with any search algorithm.

```
def printMaze(matrix):
    plt.pcolormesh(matrix)
    plt.xticks([])
    plt.yticks([])
    plt.gca().invert_yaxis()
    plt.title("{}x{}".format(rows, cols))
```

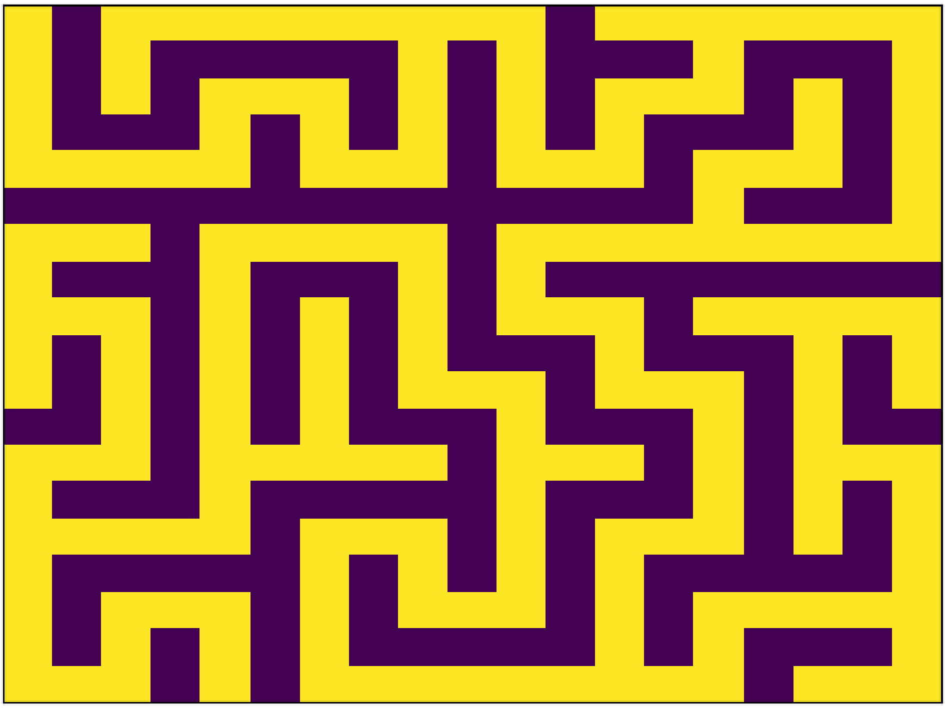
Thanks to this function, I can visualize my labyrinths.

Below are a few screenshots.

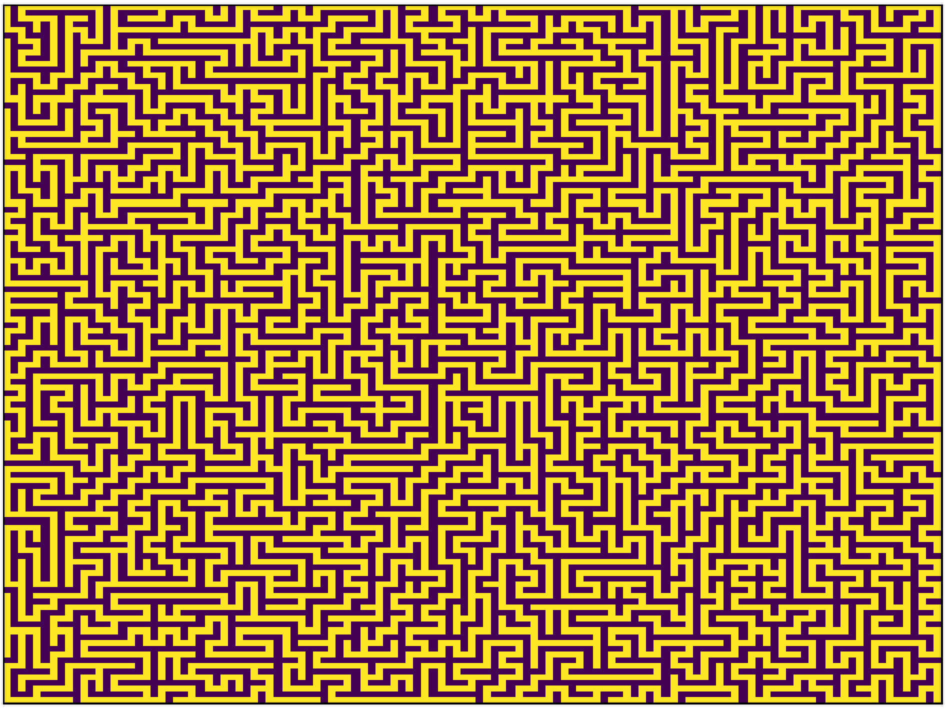
10x10



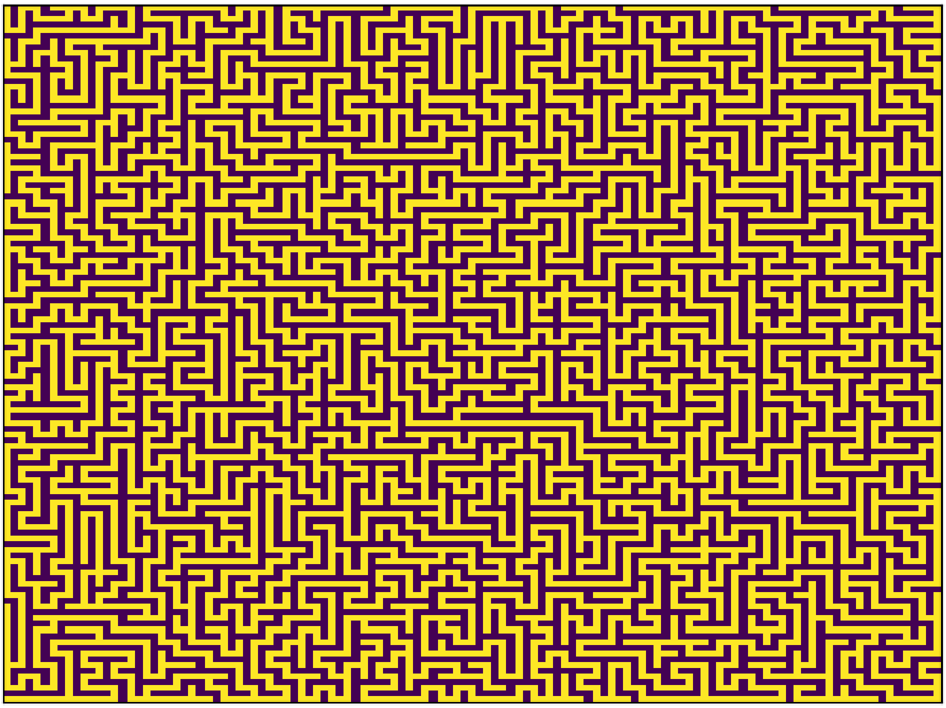
10x10



61x61



61x61



## Part 2: Application of search strategies

In this section, I will try to explain how I implemented search algorithms.

First of all, I want to explain a few helpful functions. I think this is appropriate so that it can be understood more concretely when I talk about it later.

Thanks to this function, I can print the paths I find recursively, and thanks to the other function, I can bring all the elements of the my maze back to their initial state.

```
def print_path(start, goal):
    if start == goal:
        print(goal, end=" ")
        return
    if goal.parent is not None:
        print_path(start, goal.parent)
    print("->", goal, end=" ")
```

```
def clearVisitedStatus(matrix):
    for x in range(rows):
        for y in range(cols):
            matrix[x][y].visited = False
            matrix[x][y].parent = None
```

The last auxiliary function I want to mention is my distanceCalculator function. Thanks to this function, I calculated the heuristic values.

```
def distanceCalculator(matrix):
    row_count = len(matrix)
    column_count = len(matrix[0])
    for x in range(row_count):
        for y in range(column_count):
            x_dist = (row_count - x - 1) ** 2
            y_dist = (column_count - y - 1) ** 2
            matrix[x][y].euclideanDistance = math.sqrt(x_dist + y_dist)
            x_dist = (row_count - x - 1)
            y_dist = (column_count - y - 1)
            matrix[x][y].manhattanDistance = x_dist + y_dist
            matrix[x][y].uniformCostDistance = x_dist + y_dist
```

I've included screenshots of how I implemented my following search algorithms.

```python
def a_star_search_with_Manhattan(start, goal):
    found, fringe, visited, came_from, cost_so_far = False, [(start.manhattanDistance, start)], {start}, {
        start: None}, {start: 0}
    while not found and len(fringe):
        _, current = heappop(fringe)
        if current == goal: found = True; break
        for node in current.connectedCells:
            new_cost = cost_so_far[current] + 1
            if node.visited == False or cost_so_far[node] > new_cost:
                node.visited = True
                node.parent = current
                cost_so_far[node] = new_cost
                heappush(fringe, (new_cost, node))
    if found:
        print("")
        print("A* Search With Manhattan Heuristic Values")
        print_path(start, goal)
        print("")

        return True
```

```python
def a_star_search_with_Euclidean(start, goal):
    found, fringe, visited, came_from, cost_so_far = False, [(start.euclideanDistance, start)], {start}, {
        start: None}, {start: 0}
    while not found and len(fringe):
        _, current = heappop(fringe)
        if current == goal: found = True; break
        for node in current.connectedCells:
            new_cost = cost_so_far[current] + 1
            if node.visited == False or cost_so_far[node] > new_cost:
                node.visited = True
                node.parent = current
                cost_so_far[node] = new_cost
                heappush(fringe, (new_cost, node))
    if found:
        print("")
        print("A* Search With Euclidean Heuristic Values")
        print_path(start, goal)
        print("")

        return True
```

```python
def uniform_cost_search(start, goal):
    print("")

    found, fringe, visited, came_from, cost_so_far = False, [(0, start)], {start}, {start: None}, {start: 0}
    while not found and len(fringe):
        _, current = heappop(fringe)
        if current == goal:
            found = True
            break

        for node in current.connectedCells:
            new_cost = cost_so_far[current] + 1
            if node.visited == False or cost_so_far[node] > new_cost:
                node.visited = True
                node.parent = current
                cost_so_far[node] = new_cost
                heappush(fringe, (new_cost, node))
    if found:
        print("")
        print("Uniform Cost Search")
        print_path(start, goal)
        print("")

        return True
```

```python
def DLS(src, target, maxDepth):
    if src == target: return True
    if maxDepth <= 0: return False
    for i in src.connectedCells:
        if i.visited:
            continue
        i.visited = True
        i.parent = src
        if DLS(i, target, maxDepth - 1):
            return True
    return False


def IDDFS(src, target, maxDepth):
    print("Iterative Deepening Search")
    for i in range(maxDepth):
        clearVisitedStatus(grid)
        if DLS(src, target, i):
            print_path(src, target)
            return True
    return False
```

Some paths are as follows:

*10x10_1*

## *Iterative Deepening Search*

*(0,0) -> (1,0) -> (2,0) -> (3,0) -> (3,1) -> (4,1) -> (5,1) -> (5,2) -> (4,2) -> (4,3) -> (4,4) -> (5,4) -> (5,5) -> (4,5) -> (4,6) -> (4,7) -> (5,7) -> (5,6) -> (6,6) -> (6,5) -> (6,4) -> (7,4) -> (7,3) -> (7,2) -> (6,2) -> (6,1) -> (6,0) -> (7,0) -> (8,0) -> (8,1) -> (8,2) -> (9,2) -> (9,3) -> (8,3) -> (8,4) -> (9,4) -> (9,5) -> (8,5) -> (8,6) -> (9,6) -> (9,7) -> (8,7) -> (7,7) -> (7,8) -> (8,8) -> (9,8) -> (9,9)*

## *Uniform Cost Search*

*(0,0) -> (1,0) -> (2,0) -> (3,0) -> (3,1) -> (4,1) -> (5,1) -> (5,2) -> (4,2) -> (4,3) -> (4,4) -> (5,4) -> (5,5) -> (4,5) -> (4,6) -> (4,7) -> (5,7) -> (5,6) -> (6,6) -> (6,5) -> (6,4) -> (7,4) -> (7,3) -> (7,2) -> (6,2) -> (6,1) -> (6,0) -> (7,0) -> (8,0) -> (8,1) -> (8,2) -> (9,2) -> (9,3) -> (8,3) -> (8,4) -> (9,4) -> (9,5) -> (8,5) -> (8,6) -> (9,6) -> (9,7) -> (8,7) -> (7,7) -> (7,8) -> (8,8) -> (9,8) -> (9,9)*

## *A* Search With Manhattan Heuristic Values*

*(0,0) -> (1,0) -> (2,0) -> (3,0) -> (3,1) -> (4,1) -> (5,1) -> (5,2) -> (4,2) -> (4,3) -> (4,4) -> (5,4) -> (5,5) -> (4,5) -> (4,6) -> (4,7) -> (5,7) -> (5,6) -> (6,6) -> (6,5) -> (6,4) -> (7,4) -> (7,3) -> (7,2) -> (6,2) -> (6,1) -> (6,0) -> (7,0) -> (8,0) -> (8,1) -> (8,2) -> (9,2) -> (9,3) -> (8,3) -> (8,4) -> (9,4) -> (9,5) -> (8,5) -> (8,6) -> (9,6) -> (9,7) -> (8,7) -> (7,7) -> (7,8) -> (8,8) -> (9,8) -> (9,9)*

## *A* Search With Euclidean Heuristic Values*

*(0,0) -> (1,0) -> (2,0) -> (3,0) -> (3,1) -> (4,1) -> (5,1) -> (5,2) -> (4,2) -> (4,3) -> (4,4) -> (5,4) -> (5,5) -> (4,5) -> (4,6) -> (4,7) -> (5,7) -> (5,6) -> (6,6) -> (6,5) -> (6,4) -> (7,4) -> (7,3) -> (7,2) -> (6,2) -> (6,1) -> (6,0) -> (7,0) -> (8,0) -> (8,1) -> (8,2) -> (9,2) -> (9,3) -> (8,3) -> (8,4) -> (9,4) -> (9,5) -> (8,5) -> (8,6) -> (9,6) -> (9,7) -> (8,7) -> (7,7) -> (7,8) -> (8,8) -> (9,8) -> (9,9)*

# *10x10_2*

## *Iterative Deepening Search*

*(0,0) -> (1,0) -> (2,0) -> (2,1) -> (3,1) -> (4,1) -> (5,1) -> (6,1) -> (6,2) -> (5,2) -> (4,2) -> (3,2) -> (2,2) -> (1,2) -> (1,3) -> (2,3) -> (2,4) -> (1,4) -> (0,4) -> (0,5) -> (1,5) -> (1,6) -> (0,6) -> (0,7) -> (1,7) -> (1,8) -> (1,9) -> (2,9) -> (2,8) -> (3,8) -> (3,7) -> (2,7) -> (2,6) -> (2,5) -> (3,5) -> (3,6) -> (4,6) -> (4,5) -> (4,4) -> (5,4) -> (5,3) -> (6,3) -> (6,4) -> (7,4) -> (7,3) -> (7,2) -> (7,1) -> (7,0) -> (8,0) -> (8,1) -> (8,2) -> (9,2) -> (9,3) -> (9,4) -> (8,4) -> (8,5) -> (7,5) -> (7,6) -> (8,6) -> (9,6) -> (9,7) -> (8,7) -> (7,7) -> (7,8) -> (8,8) -> (8,9) -> (9,9)*

## *Uniform Cost Search*

*(0,0) -> (1,0) -> (2,0) -> (2,1) -> (3,1) -> (4,1) -> (5,1) -> (6,1) -> (6,2) -> (5,2) -> (4,2) -> (3,2) -> (2,2) -> (1,2) -> (1,3) -> (2,3) -> (2,4) -> (1,4) -> (0,4) -> (0,5) -> (1,5) -> (1,6) -> (0,6) -> (0,7) -> (1,7) -> (1,8) -> (1,9) -> (2,9) -> (2,8) -> (3,8) -> (3,7) -> (2,7) -> (2,6) -> (2,5) -> (3,5) -> (3,6) -> (4,6) -> (4,5) -> (4,4) -> (5,4) -> (5,3) -> (6,3) -> (6,4) -> (7,4) -> (7,3) -> (7,2) -> (7,1) -> (7,0) -> (8,0) -> (8,1) -> (8,2) -> (9,2) -> (9,3) -> (9,4) -> (8,4) -> (8,5) -> (7,5) -> (7,6) -> (8,6) -> (9,6) -> (9,7) -> (8,7) -> (7,7) -> (7,8) -> (8,8) -> (8,9) -> (9,9)*

## *A* Search With Manhattan Heuristic Values*

*(0,0) -> (1,0) -> (2,0) -> (2,1) -> (3,1) -> (4,1) -> (5,1) -> (6,1) -> (6,2) -> (5,2) -> (4,2) -> (3,2) -> (2,2) -> (1,2) -> (1,3) -> (2,3) -> (2,4) -> (1,4) -> (0,4) -> (0,5) -> (1,5) -> (1,6) -> (0,6) -> (0,7) -> (1,7) -> (1,8) -> (1,9) -> (2,9) -> (2,8) -> (3,8) -> (3,7) -> (2,7) -> (2,6) -> (2,5) -> (3,5) -> (3,6) -> (4,6) -> (4,5) -> (4,4) -> (5,4) -> (5,3) -> (6,3) -> (6,4) -> (7,4) -> (7,3) -> (7,2) -> (7,1) -> (7,0) -> (8,0) -> (8,1) -> (8,2) -> (9,2) -> (9,3) -> (9,4) -> (8,4) -> (8,5) -> (7,5) -> (7,6) -> (8,6) -> (9,6) -> (9,7) -> (8,7) -> (7,7) -> (7,8) -> (8,8) -> (8,9) -> (9,9)*

## *A* Search With Euclidean Heuristic Values*

*(0,0) -> (1,0) -> (2,0) -> (2,1) -> (3,1) -> (4,1) -> (5,1) -> (6,1) -> (6,2) -> (5,2) -> (4,2) -> (3,2) -> (2,2) -> (1,2) -> (1,3) -> (2,3) -> (2,4) -> (1,4) -> (0,4) -> (0,5) -> (1,5) -> (1,6) -> (0,6) -> (0,7) -> (1,7) -> (1,8) -> (1,9) -> (2,9) -> (2,8) -> (3,8) -> (3,7) -> (2,7) -> (2,6) -> (2,5) -> (3,5) -> (3,6) -> (4,6) -> (4,5) -> (4,4) -> (5,4) -> (5,3) -> (6,3) -> (6,4) -> (7,4) -> (7,3) -> (7,2) -> (7,1) -> (7,0) -> (8,0) -> (8,1) -> (8,2) -> (9,2) -> (9,3) -> (9,4) -> (8,4) -> (8,5) -> (7,5) -> (7,6) -> (8,6) -> (9,6) -> (9,7) -> (8,7) -> (7,7) -> (7,8) -> (8,8) -> (8,9) -> (9,9)*

*Note: I think it is appropriate to upload 2 text documents containing the paths of the 61x61 maze to my homework directory, because when I try to show it here, the length of my report file reaches 80-90 pages. That's why I chose this method. I take refuge in your understanding.*

## Part 3: Analysis of the search strategies

| Search Algorithm | Dimensions | Path Length | Expanded Nodes | Max Time |
|---|---|---|---|---|
| IDDFS | 10x10 | 50 | 1641 | 0.0049 |
| IDDFS | 10x10 | 40 | 967 | 0.0029 |
| UCS | 10x10 | 50 | 129 | 0.0019 |
| UCS | 10x10 | 40 | 91 | 0.0009 |
| A* – Manhattan | 10x10 | 50 | 129 | 0.0010 |
| A* – Manhattan | 10x10 | 40 | 91 | 0.0009 |
| A*– Euclidean | 10x10 | 50 | 129 | 0.0009 |
| A* – Euclidean | 10x10 | 40 | 91 | 0.0009 |
| IDDFS | 100x100 | 1430 | 1812970 | 8.3239 |
| IDDFS | 100x100 | 2854 | 8590808 | 22.904 |

| | | | | |
|---|---|---|---|---|
| UCS | 100x100 | 1430 | 5694 | 0.0470 |
| UCS | 100x100 | 2854 | 16209 | 0.1226 |
| A* – Manhattan | 100x100 | 1430 | 5694 | 0.0469 |
| A* – Manhattan | 100x100 | 2854 | 16209 | 0.1304 |
| A* – Euclidean | 100x100 | 1430 | 5694 | 0.0421 |
| A* – Euclidean | 100x100 | 2854 | 16209 | 0.1166 |
| IDDFS | 1000x1000 | N/A | N/A | N/A |
| IDDFS | 1000x1000 | N/A | N/A | N/A |
| UCS | 1000x1000 | 126116 | 1154392 | 12.239 |
| UCS | 1000x1000 | 207980 | 1996813 | 20.169 |
| A* – Manhattan | 1000x1000 | 126116 | 1154392 | 9.7219 |
| A* – Manhattan | 1000x1000 | 207980 | 1996813 | 19.024 |
| A* – Euclidean | 1000x1000 | 126116 | 1154392 | 10.738 |
| A* – Euclidean | 1000x1000 | 207980 | 1996813 | 16.849 |
| IDDFS | 61x61 | 1420 | 2026035 | 4.6079 |
| IDDFS | 61x61 | 590 | 292642 | 1.1211 |
| UCS | 61x61 | 1420 | 6975 | 0.0479 |
| UCS | 61x61 | 590 | 2314 | 0.0159 |
| A* – Manhattan | 61x61 | 1420 | 6975 | 0.0149 |

| | | | | |
|---|---|---|---|---|
| A* – Manhattan | 61x61 | 590 | 2314 | 0.0149 |
| A* – Euclidean | 61x61 | 1420 | 6975 | 0.0448 |
| A* – Euclidean | 61x61 | 590 | 2314 | 0.0159 |
| IDDFS | 761x761 | N/A | N/A | N/A |
| IDDFS | 761x761 | N/A | N/A | N/A |
| UCS | 761x761 | 81668 | 597878 | 4.9300 |
| UCS | 761x761 | 102460 | 688890 | 3.8148 |
| A* – Manhattan | 761x761 | 81668 | 597878 | 3.7978 |
| A* – Manhattan | 761x761 | 102460 | 688890 | 5.9992 |
| A* – Euclidean | 761x761 | 81668 | 597878 | 3.9608 |
| A* – Euclidean | 761x761 | 102460 | 688890 | 4.5305 |

## Part 4: Extending the limits

| Algorithm | Size | Cost | Expanded Nodes | Time |
|---|---|---|---|---|
| UCS | 1500 | 337532 | 2696291 | 19.483 |
| A* – Manhattan | 1500 | 337532 | 2696291 | 21.466 |
| A* – Euclidean | 1500 | 337532 | 2696291 | 21.976 |

When I try to test in a 2000 and 2500 size maze it is now very difficult to avoid memory errors.

## References:

[1] https://www.baeldung.com/cs/maze-generation

[2] https://www.geeksforgeeks.org/uniform-cost-search-dijkstra-for-large-graphs/

[3] https://www.geeksforgeeks.org/iterative-deepening-searchids-iterative-deepening-depth-first-searchiddfs/

[4] https://cyluun.github.io/blog/uninformed-search-algorithms-in-python

[5] https://github.com/chitholian/AI-Search-Algorithms

[6] https://stackoverflow.com/questions/43300179/plotting-an-array-in-python

[7] https://en.wikipedia.org/wiki/Maze_generation_algorithm