Artificial Intelligence Assignment 1

1 Maze generation and design

I used mazelib (John, 2014) to generate the mazes due to the ease of access and many options for customizing the maze generation algorithm and parameters. I chose the Conway's Game of Life-based Cellular Automaton maze generation algorithm because it is capable of producing mazes with loops and multiple correct paths, which allows me to see the differences between different algorithms in terms of path optimality or propensity to get stuck in loops. I am only generating square mazes because many parameters, including buttons, cell sizes, delays etc are dependent on the maze size and only having one size *n* greatly simplifies this.

I will be comparing the different algorithms and evaluating their performance on a small, medium, and large maze with n = 5, n = 13, and n = 42 respectively. This will allow the different characteristics of each algorithm to show. Other sizes could have also been used, with some amendments: the size n must be larger than 3 due to the restrictions of the maze generator. The DFS tends to reach maximum recursion depth for mazes that are too large. The maximum depth has been increased and most cases up to n = 70 should work, but it is possible the game might crash for specific untested cases. The maze also becomes difficult to visualise for very large sizes. The MDP algorithms quickly become very time-consuming as size increases. The complexity and density parameters of the maze are set to 0.1 and 0.5. These influence the way the maze is formed, and the specific values were chosen because they result in mazes with many loops and some chambers. The A* algorithm uses the Manhattan distance to compute estimated distance to the end. Because a path cannot have diagonal movement, using the Manhattan distance ensures that the estimation is always smaller or equal to the actual path length.

I am using the pygame library to draw the maze and visually show how different algorithms work. The "game" opens on a main menu which can be reaccessed from the game itself. The main menu allows for the modification of the window size (specifically the height, as the aspect ratio is locked at 7:5), maze size n, MDP noise, and MDP gamma, with default values 600, 10, 0.2 and 0.9 respectively. Both the main menu and the game menu have a quit option. In addition, game menu has 5 buttons corresponding to the 3 search and 2 MDP algorithms which trigger the 5 algorithms and generate visualisations. For the three search algorithms, I am showing the animated search process by colouring all the visited cells in the order of visitation in a rainbow gradient. The rainbow colours are mostly cosmetic, but they can show in a single snapshot the progression of the search. The path is then shown in a darker colour. There is a delay between each cell being coloured in order for the whole process to be clearly visible. The delay is inversely proportional to the number of cells n^2 such that all mazes, regardless of size, take similarly long to animate. For the two MDP algorithms, I am visualising, for each cell, the policy using a direction symbol ("^", ">", "v", or "<") and the value by encoding it in the cell brightness. The game menu also has a "Clear" button which redraws the maze. This is automatically done before each of the other 5 visualisations as well.

When running each algorithm I am also printing some performance measurements. For each of the algorithms I am printing the time it took to run the main algorithm - that is, excluding the visualisation. "Path length" and "Nodes visited" are also computed for each algorithm, although they refer to different metrics for the search vs MDP algorithms. The path length of the search algorithms is computed as the length of the optimal path found by the algorithm, shown in the visualisation as a dark line. The number of nodes visited is the number of nodes considered (visited) by the algorithm, shown in the visualisation as rainbow coloured cells. The path length of the MDP algorithms is computed as the optimal path found by always following the direction policy. The number of nodes visited is different from the path length when noise is introduced. It is computed by simulating an agent's run through the maze using the MDP policy and the specified noise. The simulation is run 100 times and the average is taken. Thus, the number of nodes visited will not provide a basis of comparison between search and MDP algorithms since it measures different things - taking into account noise for the MDP which is incompatible with search algorithms, but the path length, which does not use noise, can be used. Finally, for the two MDP algorithms I am printing the number of iterations it took to converge. These performance measures will allow me to discuss the main pros and cons of the algorithms compared to eachother.

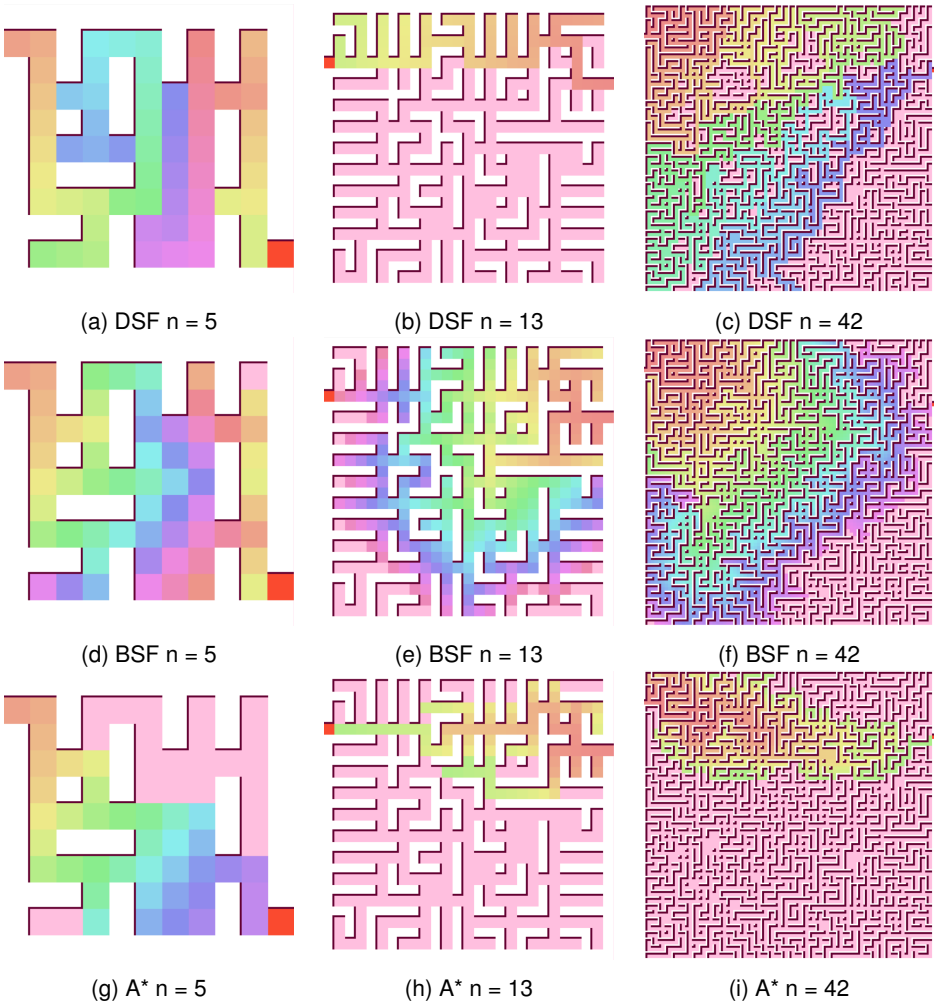
2 Results and discussion

2.1 Search algorithms

Figure 1 shows, for each combination of algorithm and maze size, the cells that were visited in the process of finding a path. Figure 2 shows, for the same combinations, the paths found. Table 1 shows, for each combination, the time it took to find the path, the path length, and the number of cells visited.

	Time (milliseconds)			Path length			Visited nodes		
	n = 5	n = 13	n = 42	n = 5	n =13	n = 42	n = 5	n = 13	n = 42
DFS	1.00	1.00	86.00	40	37	1080	59	73	2473
BFS	0.99	5.73	122.06	19	37	127	59	319	2999
A*	0.00	0.00	30.00	19	37	127	38	89	968

Table 1. *Performance of DF, BF and A* search algorithms*



(a) DSF n = 5 (b) DSF n = 13 (c) DSF n = 42

(d) BSF n = 5 (e) BSF n = 13 (f) BSF n = 42

(g) A* n = 5 (h) A* n = 13 (i) A* n = 42

Figure 1: Visited nodes by each search algorithm for 3 maze sizes

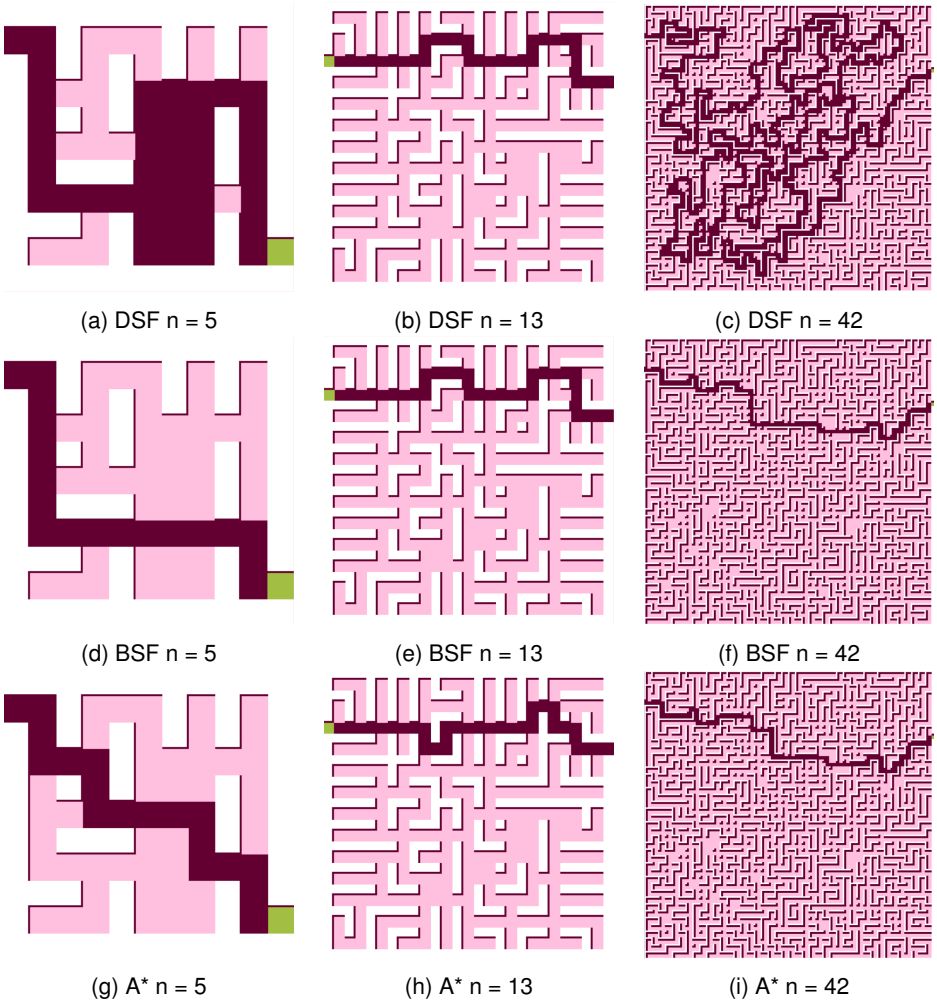
The distinction between BFS and A* is visible in Figure 1 in each of the mazes - while BFS spreads out checking every cell within a "radius", A* spreads only in the direction of the end cell. This results in A* taking the least time to finish in each case, and visiting only a small fraction of the cells compared to BFS. In the small maze, the number of cells visited is similar due to there not be much room for BFS to spread out. In the medium maze, A* visits significantly fewer cells due to there being an almost straight path to the end. The large maze shows a trend which, from other observations, seems to prevail where A* tends to visit about a third of the cells visited by BFS. I believe this could be due to BFS expanding in 3 general directions - towards

the end, which is always on the opposite wall, and to the sides - whereas A* tries to only move towards the end.

The small maze and the large maze both show DFS meandering through the whole or most of the maze in search of a path. It ends up visiting almost as many cells as the BFS, and it takes a similar time to do so.

In The medium maze shows an interesting case for the DFS. Due to chance, the same policy that would normally cause DFS to take forever (iteratively speaking) to find a path leads it straight to the end with a smaller number of visited nodes than even the A*. Another interesting thing to note is that in spite of visiting fewer cells than A*, it still takes longer to find the path.

In each case, A* finds the same path length as BFS. However, Figure 2 shows that this path tends to look slightly more "straight" for A* due to it always looking to go towards the end cell. For DFS, the resulting path is twice as long as the optimal one in the small maze - where it passes through every cell of the chamber- and almost 10 times as long in the large maze - which shows a spectacularly tangled path.



(a) DSF n = 5 (b) DSF n = 13 (c) DSF n = 42

(d) BSF n = 5 (e) BSF n = 13 (f) BSF n = 42

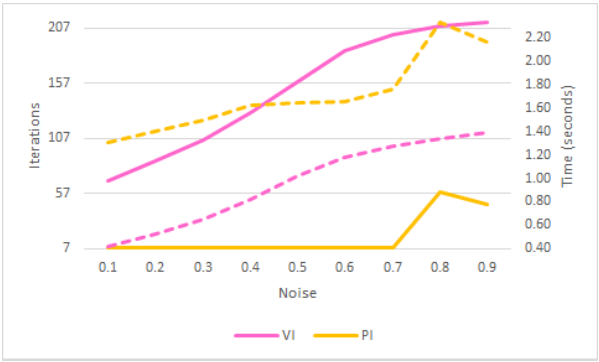
(g) A* n = 5 (h) A* n = 13 (i) A* n = 42

Figure 2: Path of each search algorithm for 3 maze sizes

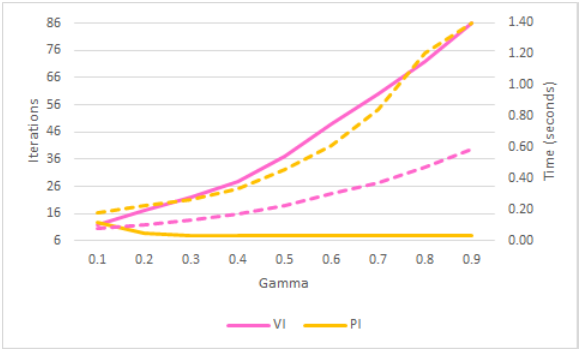
2.2 MDP algorithms

The experiments in Figure 3 were run on mazes with size n = 10. The figure on the left shows the performance of the two MDP algorithms in terms of time and number of iterations for different noise values, with gamma kept equal to 0.9. The solid lines correspond to iterations, and the dashed lines correspond to time. It can be seen that the two yellow lines, corresponding to PI have very similar patterns, as do the two pink VI lines. In order to keep the experiments computationally affordable while still observing the effects of noise, future comparisons will be run on noise equal to 0.2.

Figure 3 (b) shows the performance of the two MDP algorithms in terms of time and number of iterations for different gamma values, with noise kept equal to 0.2. The solid lines correspond to iterations, and the dashed lines correspond to time. Even though for low values of gamma the algorithms converge very fast, VI does not find an optimal if the values are too low. It needs a gamma of 0.5 or larger to find a path in the medium sized maze, and 0.9 for the large maze. Setting gamma to 0.9 is necessary for VI to find an optimal path in all mazes, future experiments will all be run on gamma = 0.9. That being said, it should be noted that for small enough problems, reducing gamma greatly improves the performance of VI in terms of iterations and of PI in terms of speed.



(a) Noise



(b) Gamma

Figure 3: Comparison of the 5 solvers

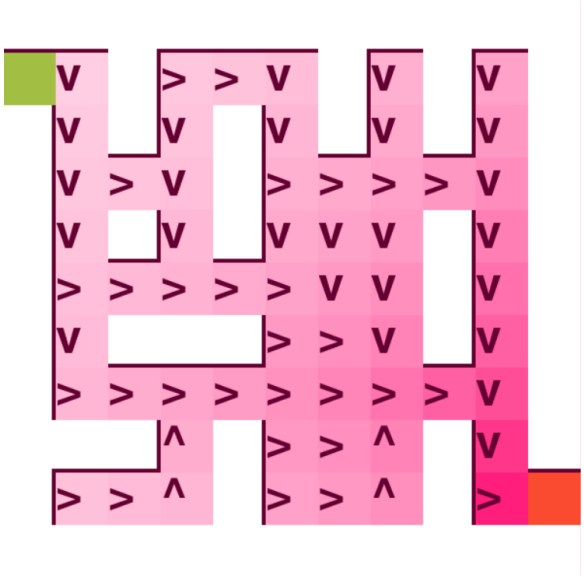
	Iterations			Time (seconds)			Path length			Visited nodes		
	n = 5	n = 13	n = 42	n = 5	n =13	n = 42	n = 5	n = 13	n = 42	n = 5	n = 13	n = 42
VI	64	101	226	0.11	1.08	25.06	19	37	127	23.26	47.56	165.85
PI	11	8	36	0.38	2.40	105	19	37	127	23.00	47.19	164.64

Table 2. *Performance of value iteration and policy iteration search algorithms*

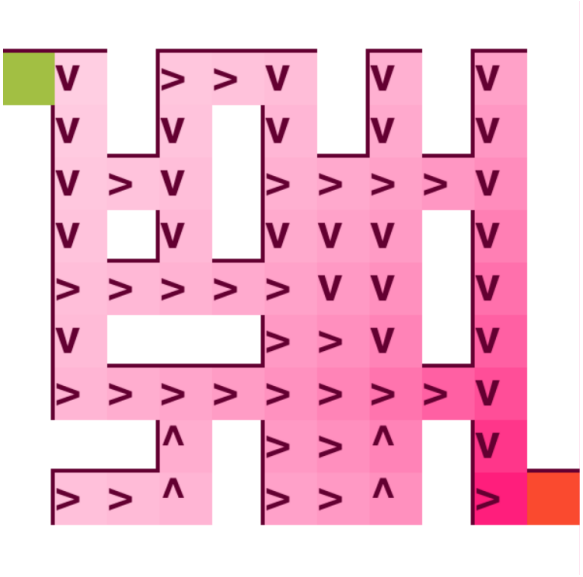
Table 2 shows the performance measurements of VI and PI for each of the three maze sizes. Figure 4 shows the result in terms of policy and values. Policy is shown as an arrow, while value is encoded in the cell brightness. For n = 42, an enlarged portion of the VI maze can be seen. In the PI figures, the cells which differ in policy from the VI counterpart are highlighted in green.

By examining the table and the figures it can be seen that both VI and PI manage to find optimal paths. Moreover, when simulating an noisy agent moving through the maze, both algorithms lead to the same average number of visited paths. The PI averages are slightly smaller than the VI, but given that the simulation only runs 100 times, and that the policies are almost identical, especially along the optimal path, it is most likely that this is just due to random chance.

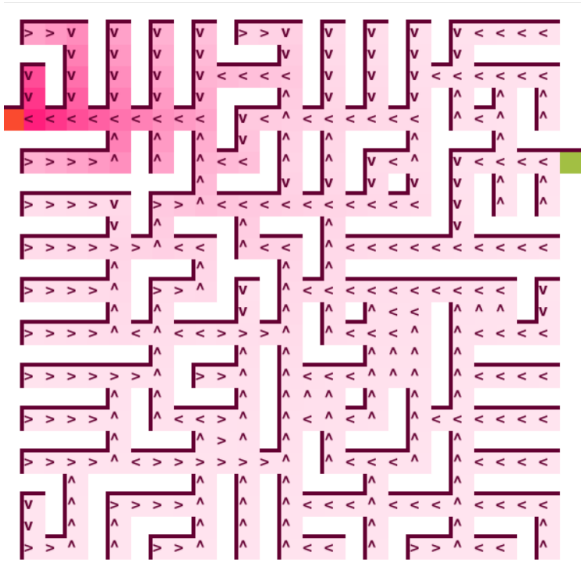
The biggest difference between VI and PI lies in the number of iterations and the time it takes to converge. Due to PI being more complex, requiring the value to converge for a given policy at each iteration, it always end up taking longer to converge than VI - 3 to 4 times longer in the tree mazes observed. However, it manages to do so in only a fraction of the iterations - 6 to 16 times faster in the three mazes.



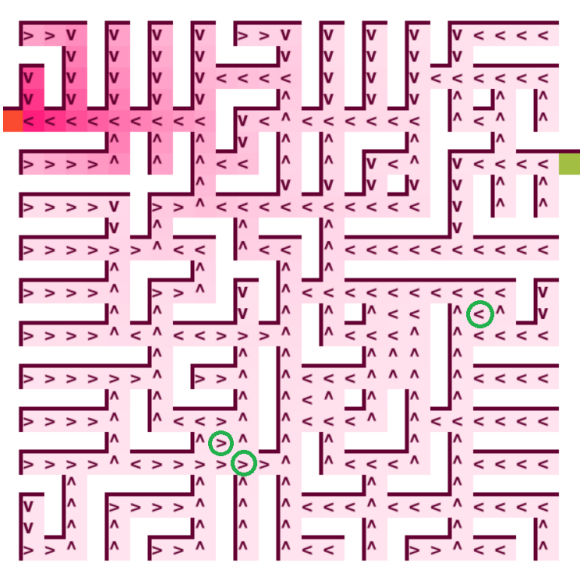
(a) VI n = 5



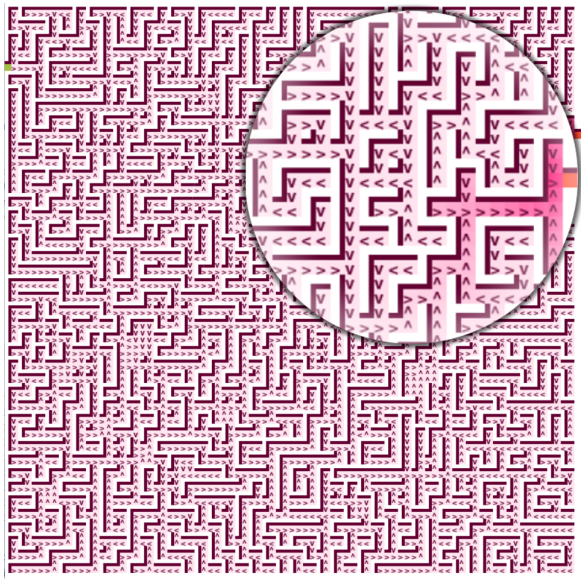
(b) PI n = 5



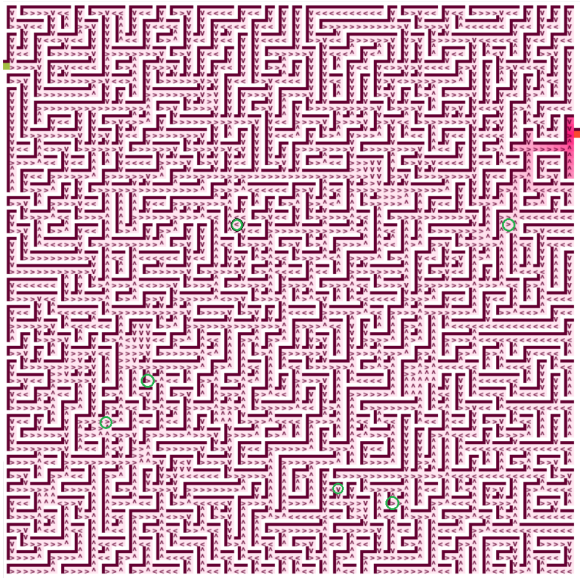
(c) VI n = 13



(d) PI n = 13



(e) VI n = 42



(f) PI n = 42

Figure 4: Policy and values of VI and PI algorithms

2.3 Search and MDP

Table 3 shows the comparison between all 5 solvers. The only one that doesn't manage to find an optimal path is the DFS, with the exception of the case where the maze randomly generates to favour it. In terms of time, the MDP takes much longer to converge that it does the search algorithms to find a path. Not only that, but the time seems to increase faster with maze size for the MDP algorithms than it does for the search algorithms.

	Time			Path length		
	n = 5	n = 13	n = 42	n = 5	n =13	n = 42
DFS	1.00	1.00	86.00	40	37	1080
BFS	0.99	5.73	122.06	19	37	127
A*	0.00	0.00	30.00	19	37	127
VI	110	1080	25060	19	37	127
PI	380	2400	105000	19	37	127

Table 3. *Comparison of search and MDP algorithms*

Figure 5 shows the comparison of the 5 algorithms in terms of time it takes to find a path. It shows how, for each algorithm, the increase in running time is exponential, but it also shows that this happens at a larger order of magnitude for the MDP algorithms.

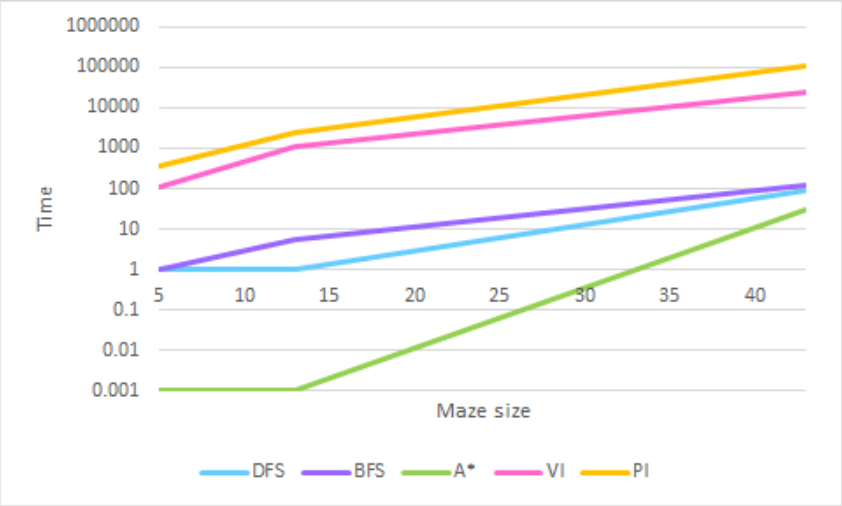


Figure 5: Comparison of the 5 solvers

References

John, S. (2014). mazelib. `github.com/john-science/mazelib.git`.

Appendix A - A* Code

```
def Astar(maze, start, end, visited):
    gCost = {start: 0}
    hCost = {start: abs(start[0] - end[0]) + abs(start[1] - end[1])}
    cost = {start: abs(start[0] - end[0]) + abs(start[1] - end[1])}
    parent = {start: None}
    current = start
    visited = []
    while current != end:
        current = next(iter(cost))
        visited.append(current)
        (row, col) = current
        cost.pop(current)
        neighbours = [(row - 1, col), (row, col - 1), (row + 1, col), (row, col + 1)]
        for neighbour in neighbours:
            (row, col) = neighbour
            if neighbour == end:
                parent[neighbour] = current
                current = neighbour
                break
            if row >= len(maze) or col >= len(maze[0]):
                continue
            if maze[row][col] == 0 and neighbour not in visited:
                parent[neighbour] = current
                gCost[neighbour] = gCost[current] + 1
                hCost[neighbour] = abs(row - end[0]) + abs(col - end[1])
                cost[neighbour] = gCost[neighbour] + hCost[neighbour]
                cost = dict(sorted(cost.items(), key=lambda item: item[1]))
    path = []
    while current != start:
        path.insert(0, current)
        current = parent[current]
    return path, visited
```

Appendix B - BFS Code

```
def bfs(maze, start, end, visited):
    queue = [start]
    parent = {start: None}
    current = queue[0]
    while current != end:
        current = queue[0]
        visited.append(current)
        (row, col) = current
        neighbours = [(row - 1, col), (row, col - 1), (row + 1, col), (row, col + 1)]
        queue.remove(current)
        for neighbour in neighbours:
            (row, col) = neighbour
            if neighbour == end:
                parent[neighbour] = current
                current = neighbour
                break
            if row >= len(maze) or col >= len(maze[0]):
                continue
            if maze[row][col] == 0 and neighbour not in queue and neighbour not in visited:
```

```
                parent[neighbour] = current
                queue.append(neighbour)
    path = []
    while current != start:
        path.insert(0, current)
        current = parent[current]
    path.insert(0, start)
    return path, visited
```

Appendix C - DFS Code

```
def dfs(maze, current, end, list, visited):
    path = []
    (row, col) = current
    neighbours = [(row - 1, col), (row, col - 1), (row + 1, col), (row, col + 1)]
    for neighbour in neighbours:
        if neighbour == end:
            list.append(neighbour)
            return list, visited
        (row, col) = neighbour
        if row >= len(maze) or col >= len(maze[0]):
            continue
        if maze[row][col] == 0 and neighbour not in visited:
            visited.append(neighbour)
            list.append(neighbour)
            path.extend(dfs(maze, neighbour, end, list, visited)[0])
            if path != []:
                break
    list.pop()
    return path, visited
```

Appendix D - PI Code

```
def policyMDP(grid, start, end, noise, gamma):
    converged = False
    grid[end[0]][end[1]] = 0
    mazeValue = np.array([[0.0] * len(grid[0])] * len(grid))
    mazeValue[end[0]][end[1]] = 1
    mazePolicy = np.array([['^'] * len(grid[0])] * len(grid))
    l = 0
    while not converged:
        k = 0
        l = l + 1
        while not converged:
            k = k + 1
            mazePrev = copy(mazeValue)
            for row in range(len(grid)):
                for col in range(len(grid[row])):
                    currentValue = mazePrev[row][col]
                    if (row, col) == end or grid[row][col] == 1:
                        continue
                    up = mazePrev[row - 1][col] if grid[row - 1][col] == 0 and (
                        row - 1, col) != start else currentValue
                    left = mazePrev[row][ col - 1] if grid[row][col - 1] == 0 and (
```

```
                    row, col - 1) != start else currentValue
                down = mazePrev[row + 1][ col] if grid[row + 1][col] == 0 and (
                    row + 1, col) != start else currentValue
                right = mazePrev[row][ col + 1] if grid[row][col + 1] == 0 and (
                    row, col + 1) != start else currentValue
                neighbours = {'^': [up, left, right], '>': [right, up, down],
                'v': [down, left, right],
                             '<': [left, up, down]}
                mazeValue[row][col] = gamma*((1-noise)*neighbours[mazePolicy[row][col]][0]
                + noise/2 * (neighbours[mazePolicy[row][col]][1]
                + neighbours[mazePolicy[row][col]][2]))
        if np.max(np.abs(mazePrev - mazeValue)) < 10 ** (-12) or k > 100:
            converged = True
    for row in range(len(grid)):
        for col in range(len(grid[row])):
            if (row, col) == end or grid[row][col] == 1:
                continue
            else:
                neighbours = [['^', row - 1, col], ['<', row, col - 1],
                ['v', row + 1, col], ['>', row, col + 1],
                              ['^', row - 1, col], ['>', row, col + 1]]
                value = {}
                for i in range(4):
                    value[neighbours[i][0]] = 0
                    for j, p in enumerate([noise / 2, 1 - noise, noise / 2]):
                        if neighbours[i + j - 1][1] >= len(grid)
                        or neighbours[i + j - 1][2] >= len(grid[0]):
                            continue
                        if grid[neighbours[i + j - 1][1]][neighbours[i + j - 1][2]] == 0:
                            value[neighbours[i][0]] = value[neighbours[i][0]]
                            + p * gamma
                            * mazeValue[neighbours[i + j - 1][1]][neighbours[i + j - 1][2]]
                        else:
                            value[neighbours[i][0]] = value[neighbours[i][0]]
                            + p * gamma * mazeValue[row][col]
                value = dict(sorted(value.items(), key=lambda item: item[1]))
                if (list(value.values())[-1] > mazeValue[row][col]
                and not mazePolicy[row][col] == list(value)[-1]):
                    mazePolicy[row][col] = list(value)[-1]
                    converged = False
print('Finished in ',l,' iterations')
return mazeValue, mazePolicy
```

Appendix E - VI Code

```
converged = False
grid[end[0]][end[1]] = 0
mazePrev = np.array([[0.0] * len(grid[0])] * len(grid))
mazePrev[end[0]][end[1]] = 1
mazeValue = copy(mazePrev)
mazePolicy = np.array([['^'] * len(grid[0])] * len(grid))
k = 0
value = {}
while not converged:
    k = k + 1
    mazePrev = copy(mazeValue)
```

```
    for row in range(len(grid)):
        for col in range(len(grid[row])):
            if (row, col) == end or grid[row][col] == 1:
                continue
            else:
                neighbours = [['^', row - 1, col], ['<', row, col - 1],
                ['v', row + 1, col], ['>', row, col + 1],
                             ['^', row - 1, col], ['>', row, col + 1]]
                for i in range(4):
                    value[neighbours[i][0]] = 0
                    for j, p in enumerate([noise / 2, 1 - noise, noise / 2]):
                        if neighbours[i + j - 1][1] >= len(grid)
                        or neighbours[i + j - 1][2] >= len(grid[0]):
                            continue
                        if grid[neighbours[i + j - 1][1]][neighbours[i + j - 1][2]] == 0:
                            value[neighbours[i][0]] = value[neighbours[i][0]]
                            + p * gamma
                            *  mazePrev[neighbours[i + j - 1][1]][neighbours[i + j - 1][2]]
                        else:
                            value[neighbours[i][0]] = value[neighbours[i][0]]
                            + p * gamma * mazePrev[row][col]
                value = dict(sorted(value.items(), key=lambda item: item[1]))
                mazeValue[row][col] = list(value.values())[-1]
                mazePolicy[row][col] = list(value)[-1]
    if np.max(np.abs(mazePrev - mazeValue)) < 10 ** (-12):
        converged = True
print('Finished in ', k, ' iterations')
return mazeValue, mazePolicy
```