NANYANG TECHNOLOGICAL UNIVERSITY

SCHOOL OF COMPUTER SCIENCE AND ENGINEERING

CZ4046 Intelligent Agents

Assignment 1 Report

By: TEO WEI JIE (U1822263C)

28 March 2021

Table of Contents

Table of Contents **i**

1 Introduction **1**

1.1 Source code structure 1
1.2 Compilation & execution 2
1.3 Defining the arena 2

2 Value Iteration **4**

2.1 Implementation 4
2.2 Optimal policy & utilities of all states 7
2.3 Utility estimates plot 9

3 Policy Iteration **10**

3.1 Implementation 10
3.2 Optimal policy & utilities of all states 12
3.3 Utility estimates plot 16

4 Bonus Question **19**

4.1 Implementation 19
4.2 Relationship between complexity & convergence 20
4.2.1 Complexity limit 20

1. Introduction

A TornadoFX (Kotlin port of JavaFX) application has been developed for this project. It contains the algorithms for optimal policy finding via value iteration and policy iteration, and a user interface for ease of interaction and visualisation.

1.1. Source code structure

The submitted source code is a Gradle project, which contains several files and folders related to Gradle to facilitate builds and dependencies. As such, only the structure of the code's main module, located in "src/main/kotin", is described below:

- "algorithm" package:
  - Iteration.kt: Abstract implementation of the value and policy iteration algorithms, containing common methods and variables used by both algorithms.
  - PolicyIteration.kt: Algorithm for policy iteration, extending Iteration.
  - ValueIteration.kt: Algorithm for value iteration, extending Iteration.
- "arena" package:
  - Arena.kt: Represents and manages the arena environment. Each grid of the arena represents a state.
- "enum" package:
  - Action.kt: Enumerator for actions that can be taken by the agent.
  - Grid.kt: Enumerator for the type of grids that can be represented in the arena, along with the rewards associated to the grid type.
- "views" package:
  - ArenaView.kt: UI view to display the arena and the computed policy and utilities for each state.
  - BonusConfigView.kt: UI view to configure the arena for the bonus question.
  - BonusResultsView.kt: UI view to display results pertaining to the bonus question.
  - ConfigView.kt: UI view to configure parameters for the iteration algorithms.
  - MainView.kt: The main UI view container containing all the above views.
- Constants.kt: Contains constants used by multiple classes.
- TornadoApp.kt: The entry point of the application.

1.2. Compilation & execution

Table 1 below lists the components required for compiling the source code and for running the application.

Table 1: Compilation & execution requirements

Component	**Version**
Java Development Kit & Runtime	11.0.2
JavaFX Runtime (optional, see below)	11.0.2
Kotlin plugin-capable IDE	Kotlin 1.4.10

The JavaFX Runtime libraries are bundled with the source code in the “javafx” directory. This eliminates the need to download the libraries separately. To run the application, the following JVM options need to be set:

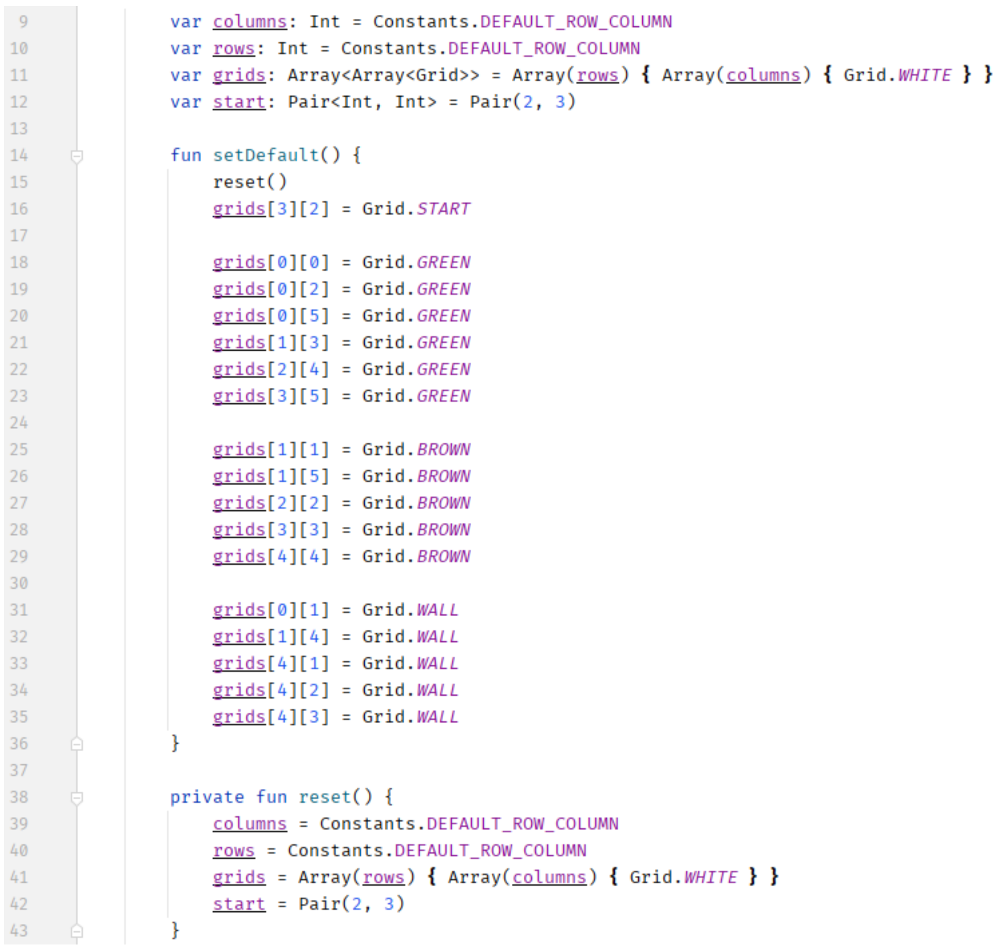
```
--module-path javafx/lib --add-modules javafx.controls,javafx.fxml
```

1.3. Defining the arena

The arena environment is represented in the application using a 2-dimensional array. Each element in the array, which represents a grid / state, is initialised with the following additional information:

- Grid type: White, Green, Brown, Wall, or Start (same parameters as White)
- Rewards associated: -0.04 for White, 1.0 for Green, -1.0 for Brown, 0 for Wall

The code snippet for initialising the arena environment is provided in Figure 1 below.



```
9       var columns: Int = Constants.DEFAULT_ROW_COLUMN
10      var rows: Int = Constants.DEFAULT_ROW_COLUMN
11      var grids: Array<Array<Grid>> = Array(rows) { Array(columns) { Grid.WHITE } }
12      var start: Pair<Int, Int> = Pair(2, 3)
13
14      fun setDefault() {
15          reset()
16          grids[3][2] = Grid.START
17
18          grids[0][0] = Grid.GREEN
19          grids[0][2] = Grid.GREEN
20          grids[0][5] = Grid.GREEN
21          grids[1][3] = Grid.GREEN
22          grids[2][4] = Grid.GREEN
23          grids[3][5] = Grid.GREEN
24
25          grids[1][1] = Grid.BROWN
26          grids[1][5] = Grid.BROWN
27          grids[2][2] = Grid.BROWN
28          grids[3][3] = Grid.BROWN
29          grids[4][4] = Grid.BROWN
30
31          grids[0][1] = Grid.WALL
32          grids[1][4] = Grid.WALL
33          grids[4][1] = Grid.WALL
34          grids[4][2] = Grid.WALL
35          grids[4][3] = Grid.WALL
36      }
37
38      private fun reset() {
39          columns = Constants.DEFAULT_ROW_COLUMN
40          rows = Constants.DEFAULT_ROW_COLUMN
41          grids = Array(rows) { Array(columns) { Grid.WHITE } }
42          start = Pair(2, 3)
43      }
```

Figure 1: Code snippet for initialising the arena

Lines 9 - 12: Initialising the given variables pertaining to the arena.

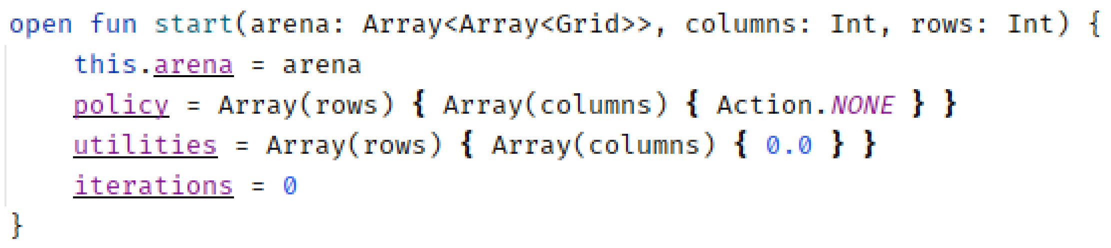
Lines 16 - 35: Setting the green spaces, brown spaces, walls, and starting grid on the initialised arena.

Lines 39 - 42: Resetting the arena to the initial state, which can be modified during the bonus question.

2. Value Iteration

2.1. Implementation

Two 2-dimension arrays, representing the best calculated utility and optimal action for each state, is first initialised. The utility array is initialised to zero across all elements while the action array is initialised to “none” across all elements (none is used for walls, but is also a placeholder for initialisation). The code snippet is provided in Figure 2 below.



```
open fun start(arena: Array<Array<Grid>>, columns: Int, rows: Int) {
    this.arena = arena
    policy = Array(rows) { Array(columns) { Action.NONE } }
    utilities = Array(rows) { Array(columns) { 0.0 } }
    iterations = 0
}
```

Figure 2: Code snippet for initialising the utility and action for each state

The algorithm then goes through the following steps:

1. Calculate the best utility and optimal action for each state based on current utility values.
2. Update utility and action array for each state.
3. Repeat steps 1 and 2 until convergence.

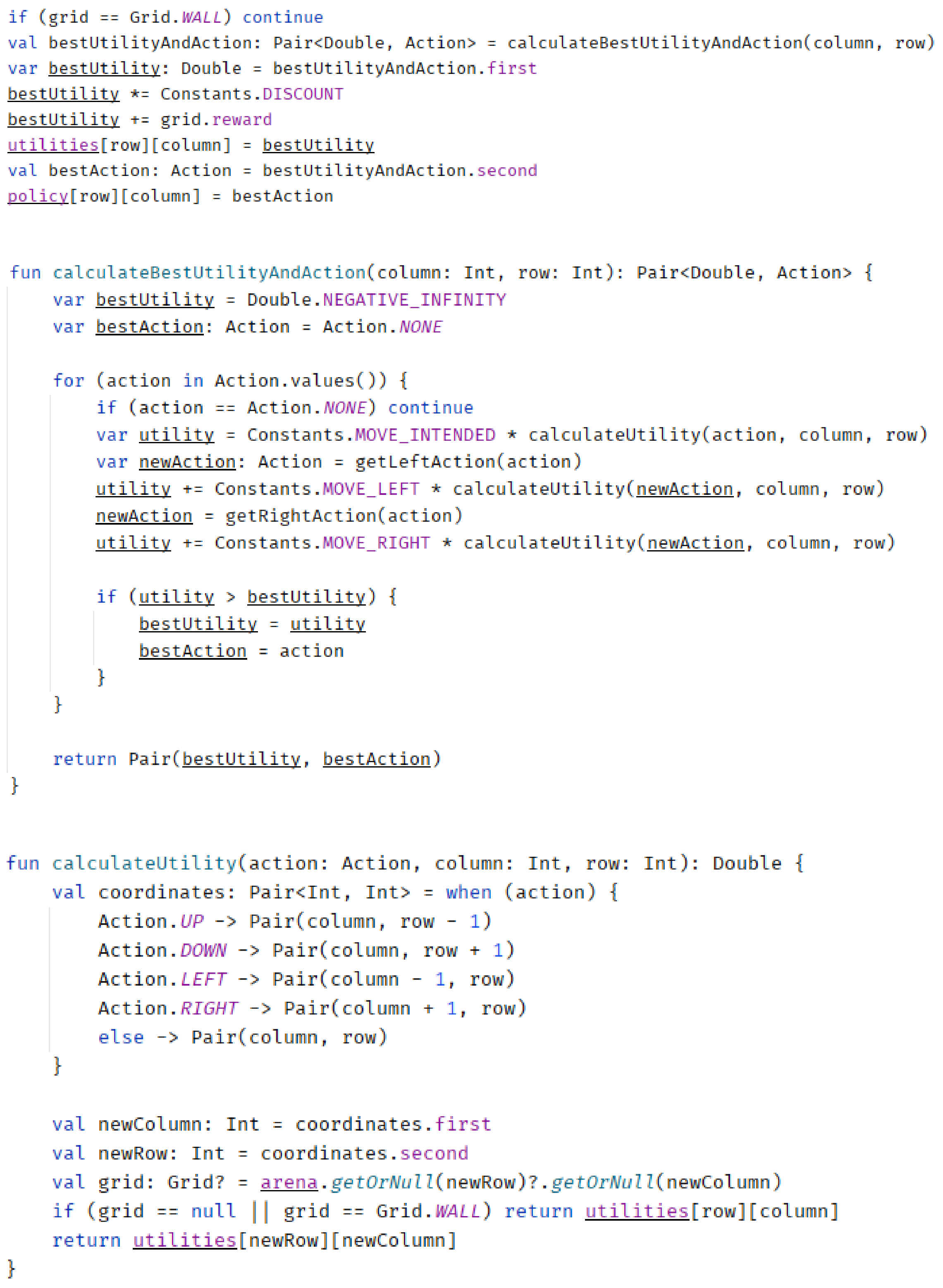
The algorithm calculates the best utility and optimal action for each state based on the current utility values stored in the utility array. The utility is calculated using the Bellman update formula:

$$U_{i+1}(s) \leftarrow R(s) + \gamma \times \max_{a \in A(s)} \Sigma_{s'} P(s'|s,a) \times U_i(s')$$

where:

- $R(s)$ is the reward for the state: -0.04 for white spaces, 1.0 for green spaces, -1.0 for brown spaces
- γ is the discount value of 0.99
- $\Sigma_{s'} P(s'|s,a)U_i(s')$ is the sum of all utilities for an action that can be taken in that state and its given probability: 0.8 to move in intended direction, 0.1 to move left of intended, 0.1 to move right of intended
- $\max_{a \in A(s)} \Sigma_{s'} P(s'|s,a)U_i(s')$ is the best utility out of all possible actions $A(s)$ in that state

The calculated best utility is then updated into the utility array, and its associated action is updated into the action array as the optimal action for that state. The code snippet is provided in Figure 3 below.



```
if (grid == Grid.WALL) continue
val bestUtilityAndAction: Pair<Double, Action> = calculateBestUtilityAndAction(column, row)
var bestUtility: Double = bestUtilityAndAction.first
bestUtility *= Constants.DISCOUNT
bestUtility += grid.reward
utilities[row][column] = bestUtility
val bestAction: Action = bestUtilityAndAction.second
policy[row][column] = bestAction


fun calculateBestUtilityAndAction(column: Int, row: Int): Pair<Double, Action> {
    var bestUtility = Double.NEGATIVE_INFINITY
    var bestAction: Action = Action.NONE

    for (action in Action.values()) {
        if (action == Action.NONE) continue
        var utility = Constants.MOVE_INTENDED * calculateUtility(action, column, row)
        var newAction: Action = getLeftAction(action)
        utility += Constants.MOVE_LEFT * calculateUtility(newAction, column, row)
        newAction = getRightAction(action)
        utility += Constants.MOVE_RIGHT * calculateUtility(newAction, column, row)

        if (utility > bestUtility) {
            bestUtility = utility
            bestAction = action
        }
    }

    return Pair(bestUtility, bestAction)
}


fun calculateUtility(action: Action, column: Int, row: Int): Double {
    val coordinates: Pair<Int, Int> = when (action) {
        Action.UP -> Pair(column, row - 1)
        Action.DOWN -> Pair(column, row + 1)
        Action.LEFT -> Pair(column - 1, row)
        Action.RIGHT -> Pair(column + 1, row)
        else -> Pair(column, row)
    }

    val newColumn: Int = coordinates.first
    val newRow: Int = coordinates.second
    val grid: Grid? = arena.getOrNull(newRow)?.getOrNull(newColumn)
    if (grid == null || grid == Grid.WALL) return utilities[row][column]
    return utilities[newRow][newColumn]
}
```

Figure 3: Code snippet for calculating best utility and optimal action

The algorithm then repeats the calculation in a loop until the largest absolute difference between the utilities calculated in the current iteration and the previous iteration is less than a convergence threshold. The convergence threshold is determined as such:

$$\epsilon \leftarrow c \times R_{max}$$

$$\tau \leftarrow \frac{\epsilon \times (1 - \gamma)}{\gamma}$$

where:

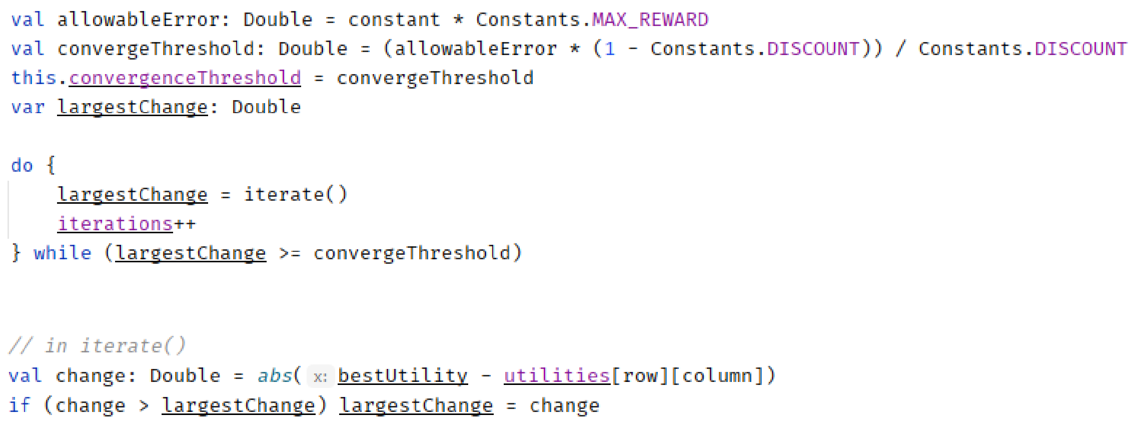
- ϵ is the maximum allowable error after k iterations
- c is a constant value, determined based on the reference book chapters provided
- R_{max} is the maximum reward available: 1.0

The constant value, c is determined after making observations based on the values provided in the reference book chapters. Table 2 below illustrates the observations made.

Table 2: Observations from using different c values

c	**Convergence Threshold**	**Iterations to Converge**
0.1	0.0010101010101010101011	688
0.01	0.0001010101010101010101	917
0.001	0.0000101010101010101010	1146
0.0001	0.0000101010101010101010	1375

As the constant value gets smaller, the number of iterations needed for convergence increases. As such, the value of 0.1 is selected as the final c value as it requires the least iterations to converge while yielding the same optimal policy for the same arena. The code snippet for determining convergence is provided in Figure 4 below.



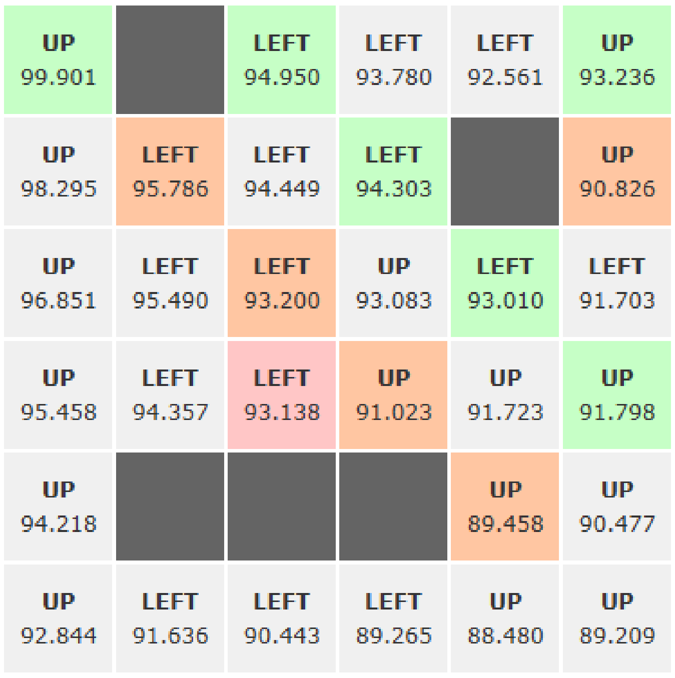
```
val allowableError: Double = constant * Constants.MAX_REWARD
val convergeThreshold: Double = (allowableError * (1 - Constants.DISCOUNT)) / Constants.DISCOUNT
this.convergenceThreshold = convergeThreshold
var largestChange: Double

do {
    largestChange = iterate()
    iterations++
} while (largestChange >= convergeThreshold)


// in iterate()
val change: Double = abs( x: bestUtility - utilities[row][column])
if (change > largestChange) largestChange = change
```

Figure 4: Code snippet for determining convergence

The final utility and action array after the algorithm terminates represents the utilities for all states and the optimal policy respectively.

2.2. Optimal policy & utilities of all states

Figure 5 below illustrates the optimal policy and the utilities for all states for the given arena, using the parameters $\gamma = 0.99$ and $c = 0.1$.



UP 99.901		LEFT 94.950	LEFT 93.780	LEFT 92.561	UP 93.236
UP 98.295	LEFT 95.786	LEFT 94.449	LEFT 94.303		UP 90.826
UP 96.851	LEFT 95.490	LEFT 93.200	UP 93.083	LEFT 93.010	LEFT 91.703
UP 95.458	LEFT 94.357	LEFT 93.138	UP 91.023	UP 91.723	UP 91.798
UP 94.218				UP 89.458	UP 90.477
UP 92.844	LEFT 91.636	LEFT 90.443	LEFT 89.265	UP 88.480	UP 89.209

Figure 5: Optimal policy and utilities for all states (value iteration)

Textual views of the optimal policy and utilities are illustrated in Tables 3 and 4 below, respectively.

Table 3: Optimal policy obtained via value iteration (walls represented by '-')

	(0,0)	**(0,1)**	**(0,2)**	**(0,3)**	**(0,4)**	**(0,5)**
(0,0)	UP	–	LEFT	LEFT	LEFT	UP
(1,0)	UP	LEFT	LEFT	LEFT	–	UP
(2,0)	UP	LEFT	LEFT	UP	LEFT	LEFT
(3,0)		LEFT	LEFT	UP	UP	UP
(4,0)	UP	–	–	–	UP	UP
(5,0)	UP	LEFT	LEFT	LEFT	UP	UP

Table 4: Utilities for all states obtained via value iteration

	(0,0)	**(0,1)**	**(0,2)**	**(0,3)**	**(0,4)**	**(0,5)**
(0,0)	99.901	–	94.950	93.780	92.561	93.236
(1,0)	98.295	95.786	94.449	94.303	–	90.826
(2,0)	96.851	95.490	93.200	93.083	93.010	91.703
(3,0)	95.458	94.357	93.138	91.023	91.723	91.798
(4,0)	94.218	–	–	–	89.458	90.477
(5,0)	92.844	91.636	90.443	89.265	88.480	89.209

2.3. Utility estimates plot

The utility estimates plot as a function of the number of iterations for Figure 5 is illustrated in Figure 6 below.

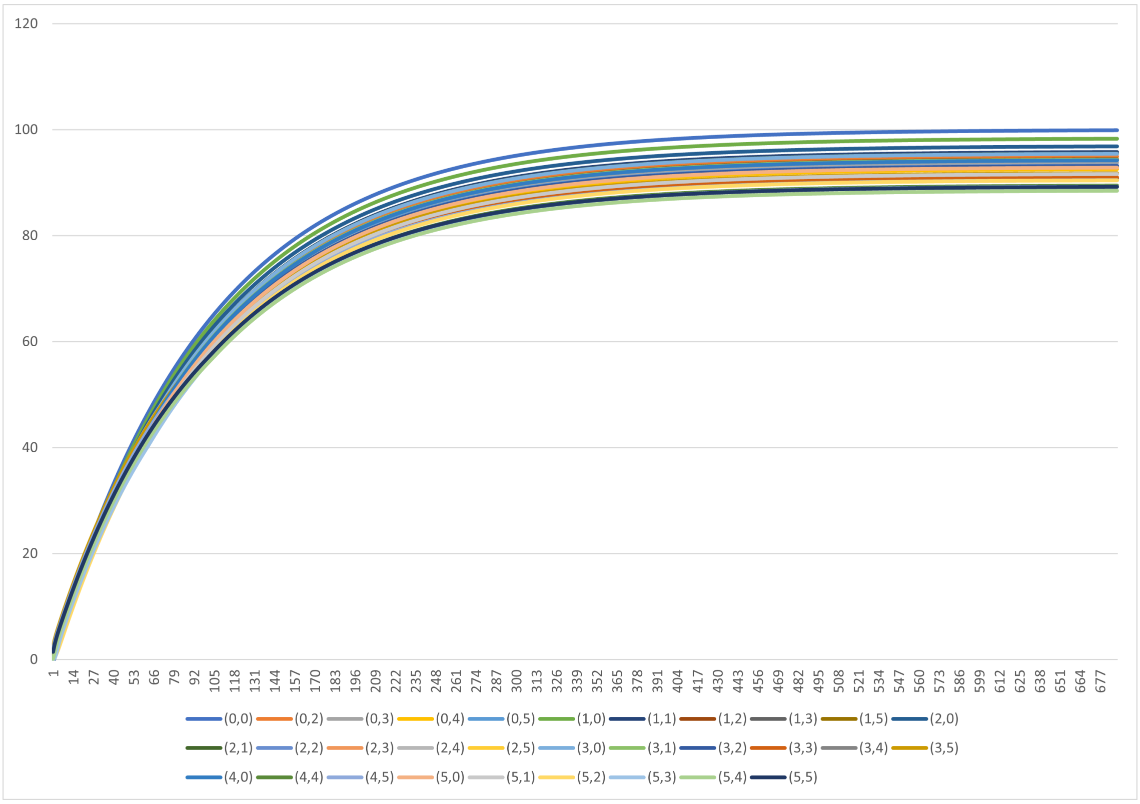
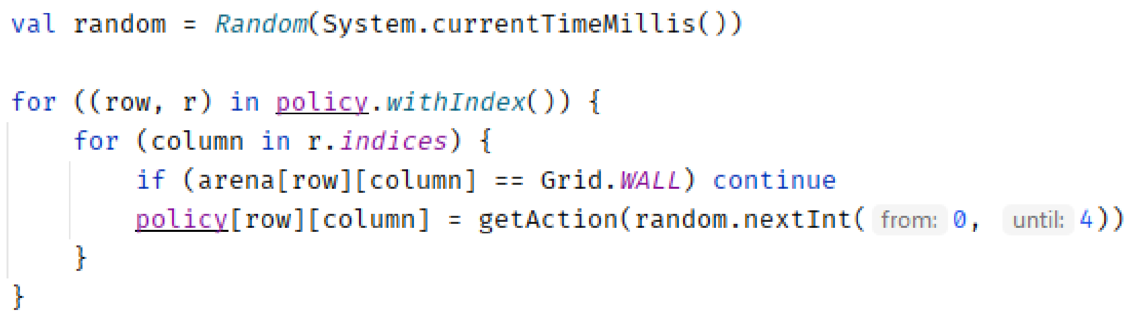


Figure 6: Utility estimates plot as a function of the number of iterations (value iteration)

3. Policy Iteration

3.1. Implementation

A utility array, representing the best calculated utility for each state, is similarly initialised as that discussed in section 2.1. An action array is then initialised similarly, but with the initial action values randomly set between “UP”, “DOWN”, “LEFT”, and “RIGHT” for non-wall grids. The code snippet for initialising the action array is provided in Figure 7 below.



```
val random = Random(System.currentTimeMillis())

for ((row, r) in policy.withIndex()) {
    for (column in r.indices) {
        if (arena[row][column] == Grid.WALL) continue
        policy[row][column] = getAction(random.nextInt( from: 0,  until: 4))
    }
}
```

Figure 7: Code snippet for policy evaluation

The algorithm then goes through the following steps:

1. Evaluates the current policy based on current actions in the action array.
2. Improves the current policy based on current utilities in the utility array.
3. Checks if improved policy is the same as original policy.
4. Repeat steps 1 - 3 if they are not the same policies.

The current policy is evaluated using a simplified Bellman update formula:

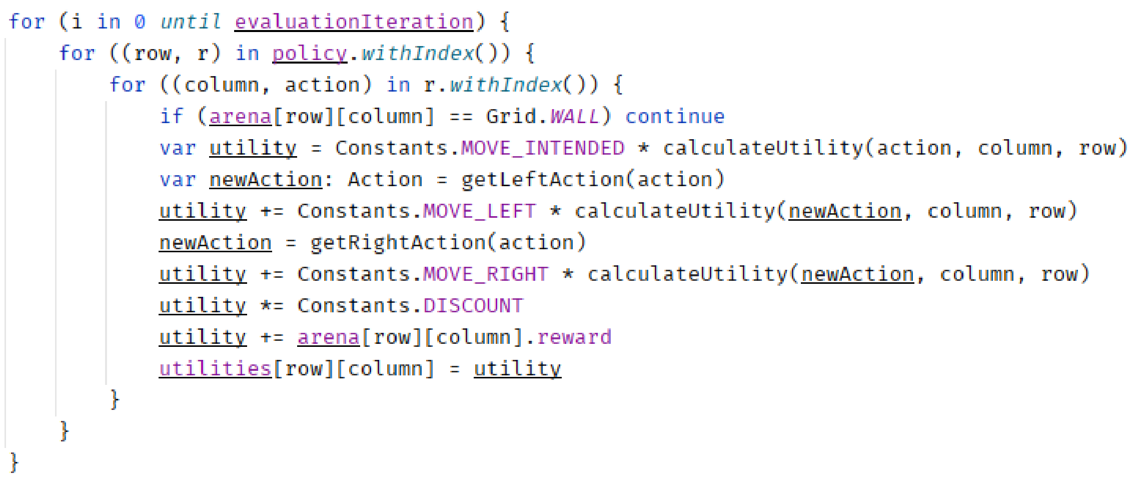
$$U_{i+1}(s) \leftarrow R(s) + \gamma \times \Sigma_{s'} P(s'|s, \pi_i(s)) \times U_i(s')$$

with the parameters similar to that as discussed in section 2.1. The policy is evaluated over a small amount of iterations, N, to estimate the utility of the policy. A handful of N values were tested over 10 runs each, and the observations made are illustrated in Table 5 below.

Table 5: Observations from using different N iteration values, each tested 10 times

N	**Iterations to Converge**
25	12 to 14
50	7 to 9
75	5 to 9
100	5 to 9
125	5 to 9

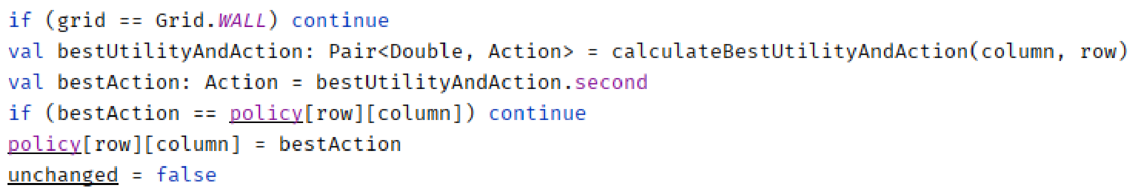
As the iterations to converge plateaus at $N = 75$, each policy evaluation is performed over 75 iterations. The code snippet for the policy evaluation is provided in Figure 8 below.



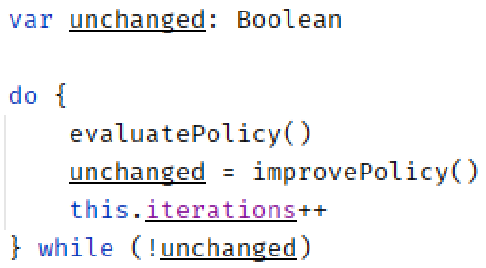
```
for (i in 0 until evaluationIteration) {
    for ((row, r) in policy.withIndex()) {
        for ((column, action) in r.withIndex()) {
            if (arena[row][column] == Grid.WALL) continue
            var utility = Constants.MOVE_INTENDED * calculateUtility(action, column, row)
            var newAction: Action = getLeftAction(action)
            utility += Constants.MOVE_LEFT * calculateUtility(newAction, column, row)
            newAction = getRightAction(action)
            utility += Constants.MOVE_RIGHT * calculateUtility(newAction, column, row)
            utility *= Constants.DISCOUNT
            utility += arena[row][column].reward
            utilities[row][column] = utility
        }
    }
}
```

Figure 8: Code snippet for policy evaluation

The policy is then improved by calculating the best utility amongst all available actions for each state based on the current utilities, similar to the calculation process for value iteration. However, the calculated best utility is not updated here. Only the calculated optimal action is updated in the action array. The code snippet for policy improvement is provided in Figure 9 below.



```
if (grid == Grid.WALL) continue
val bestUtilityAndAction: Pair<Double, Action> = calculateBestUtilityAndAction(column, row)
val bestAction: Action = bestUtilityAndAction.second
if (bestAction == policy[row][column]) continue
policy[row][column] = bestAction
unchanged = false
```



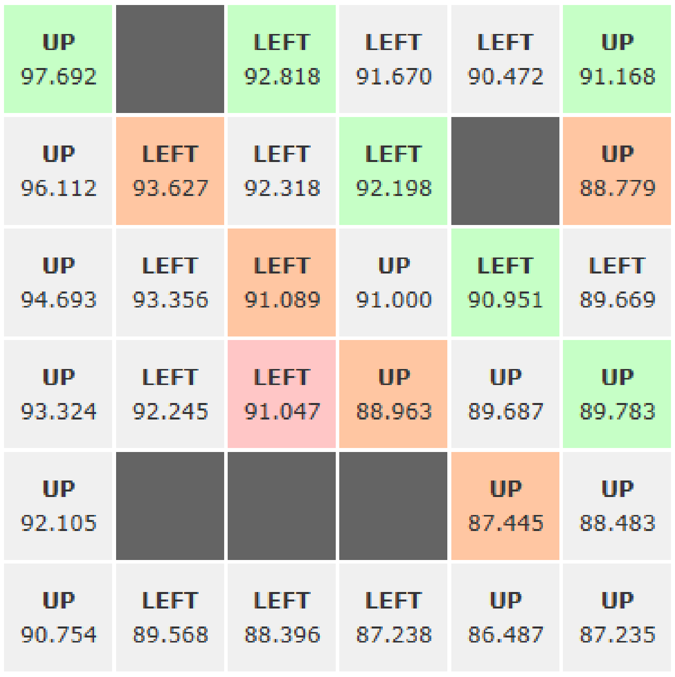
```
var unchanged: Boolean

do {
    evaluatePolicy()
    unchanged = improvePolicy()
    this.iterations++
} while (!unchanged)
```

Figure 9: Code snippet for policy improvement

Note that the `calculateBestUtilityAndAction` method is the same as that provided in Figure 3. The `unchanged` variable, initialised to `true`, is used to determine if the new policy is the same as the previous. If there is a action replacement (second last line), the variable is set to `false`. If unchanged is `false`, the algorithm repeats starting from the policy evaluation process, which will evaluate the new policy and update the utilities accordingly for improvement.

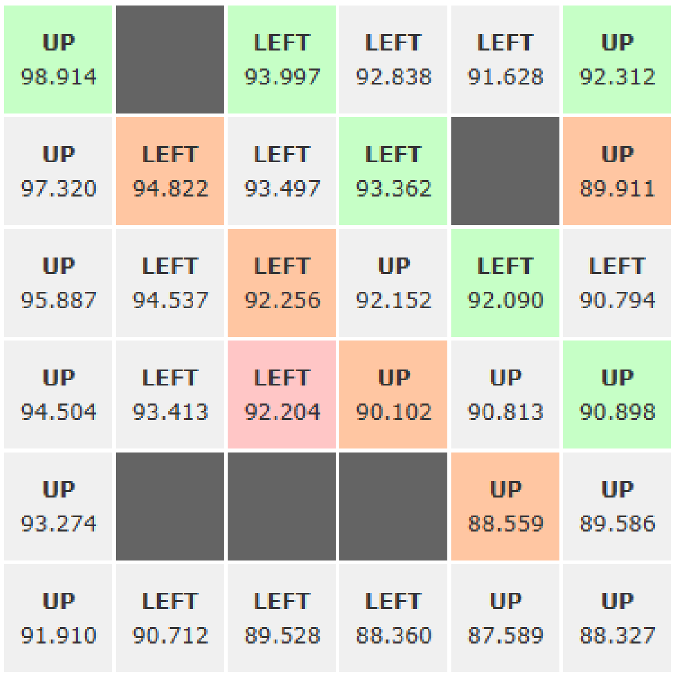
3.2. Optimal policy & utilities of all states

Figures 10, 11, 12, 13, and 14 illustrates the optimal policy and utilities for all states for $N = 75$, convergence iterations = 5, 6, 7, 8, and 9 respectively.



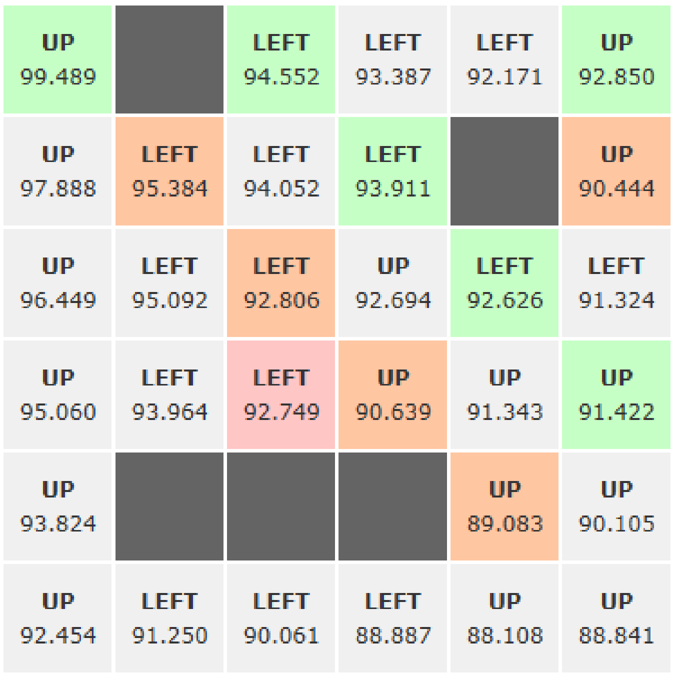
UP 97.692	⬛	LEFT 92.818	LEFT 91.670	LEFT 90.472	UP 91.168
UP 96.112	LEFT 93.627	LEFT 92.318	LEFT 92.198	⬛	UP 88.779
UP 94.693	LEFT 93.356	LEFT 91.089	UP 91.000	LEFT 90.951	LEFT 89.669
UP 93.324	LEFT 92.245	LEFT 91.047	UP 88.963	UP 89.687	UP 89.783
UP 92.105	⬛	⬛	⬛	UP 87.445	UP 88.483
UP 90.754	LEFT 89.568	LEFT 88.396	LEFT 87.238	UP 86.487	UP 87.235

Figure 10: Optimal policy and utilities for all states (policy iteration = 5)



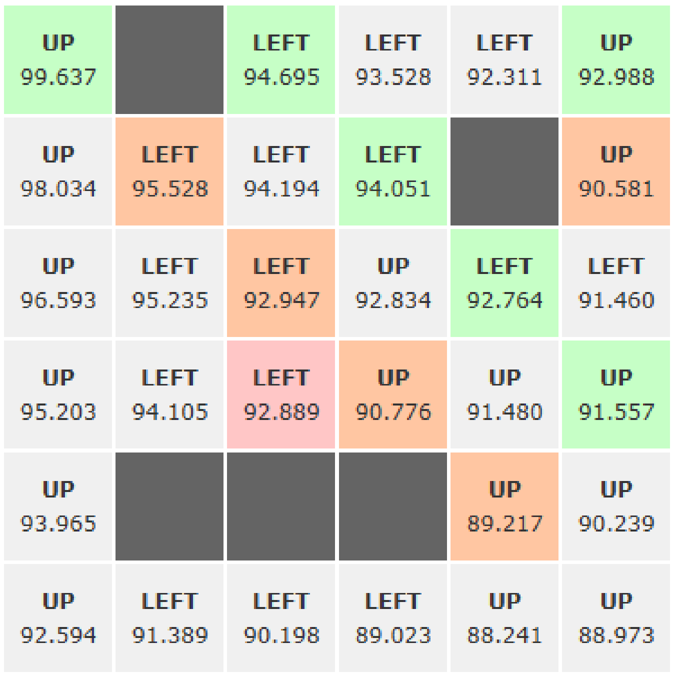
UP 98.914	⬛	LEFT 93.997	LEFT 92.838	LEFT 91.628	UP 92.312
UP 97.320	LEFT 94.822	LEFT 93.497	LEFT 93.362	⬛	UP 89.911
UP 95.887	LEFT 94.537	LEFT 92.256	UP 92.152	LEFT 92.090	LEFT 90.794
UP 94.504	LEFT 93.413	LEFT 92.204	UP 90.102	UP 90.813	UP 90.898
UP 93.274	⬛	⬛	⬛	UP 88.559	UP 89.586
UP 91.910	LEFT 90.712	LEFT 89.528	LEFT 88.360	UP 87.589	UP 88.327

Figure 11: Optimal policy and utilities for all states (policy iteration = 6)



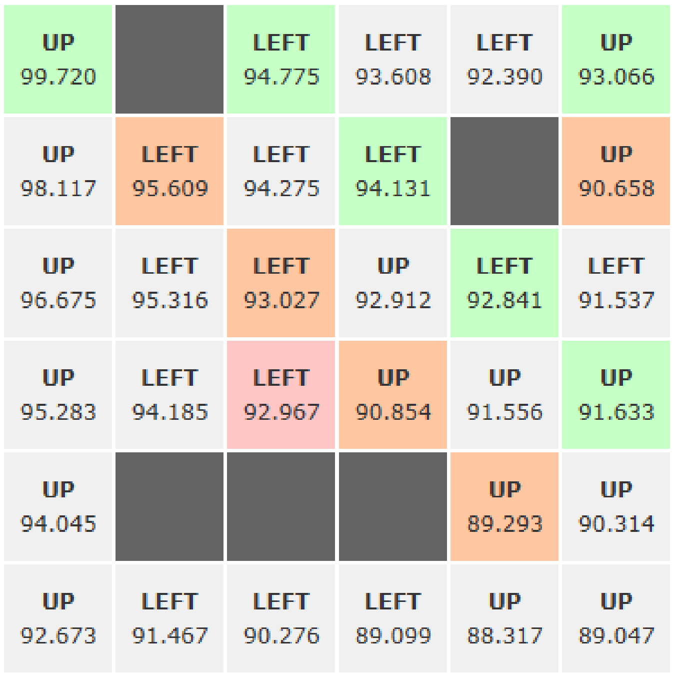
UP 99.489		LEFT 94.552	LEFT 93.387	LEFT 92.171	UP 92.850
UP 97.888	LEFT 95.384	LEFT 94.052	LEFT 93.911		UP 90.444
UP 96.449	LEFT 95.092	LEFT 92.806	UP 92.694	LEFT 92.626	LEFT 91.324
UP 95.060	LEFT 93.964	LEFT 92.749	UP 90.639	UP 91.343	UP 91.422
UP 93.824				UP 89.083	UP 90.105
UP 92.454	LEFT 91.250	LEFT 90.061	LEFT 88.887	UP 88.108	UP 88.841

Figure 12: Optimal policy and utilities for all states (policy iteration = 7)



UP 99.637		LEFT 94.695	LEFT 93.528	LEFT 92.311	UP 92.988
UP 98.034	LEFT 95.528	LEFT 94.194	LEFT 94.051		UP 90.581
UP 96.593	LEFT 95.235	LEFT 92.947	UP 92.834	LEFT 92.764	LEFT 91.460
UP 95.203	LEFT 94.105	LEFT 92.889	UP 90.776	UP 91.480	UP 91.557
UP 93.965				UP 89.217	UP 90.239
UP 92.594	LEFT 91.389	LEFT 90.198	LEFT 89.023	UP 88.241	UP 88.973

Figure 13: Optimal policy and utilities for all states (policy iteration = 8)



UP 99.720		LEFT 94.775	LEFT 93.608	LEFT 92.390	UP 93.066
UP 98.117	LEFT 95.609	LEFT 94.275	LEFT 94.131		UP 90.658
UP 96.675	LEFT 95.316	LEFT 93.027	UP 92.912	LEFT 92.841	LEFT 91.537
UP 95.283	LEFT 94.185	LEFT 92.967	UP 90.854	UP 91.556	UP 91.633
UP 94.045				UP 89.293	UP 90.314
UP 92.673	LEFT 91.467	LEFT 90.276	LEFT 89.099	UP 88.317	UP 89.047

Figure 14: Optimal policy and utilities for all states (policy iteration = 9)

From the above figures, it is observed that the difference in utilities across all five iterations is small and the general trend of the utilities across the arena is similar. Textual views (tables) of the above figures are omitted due to space constraints.

3.3. Utility estimates plot

The utility estimates plots as functions of the number of iterations, for the above figures, are illustrated in Figures 15, 16, 17, 18, and 19 below.

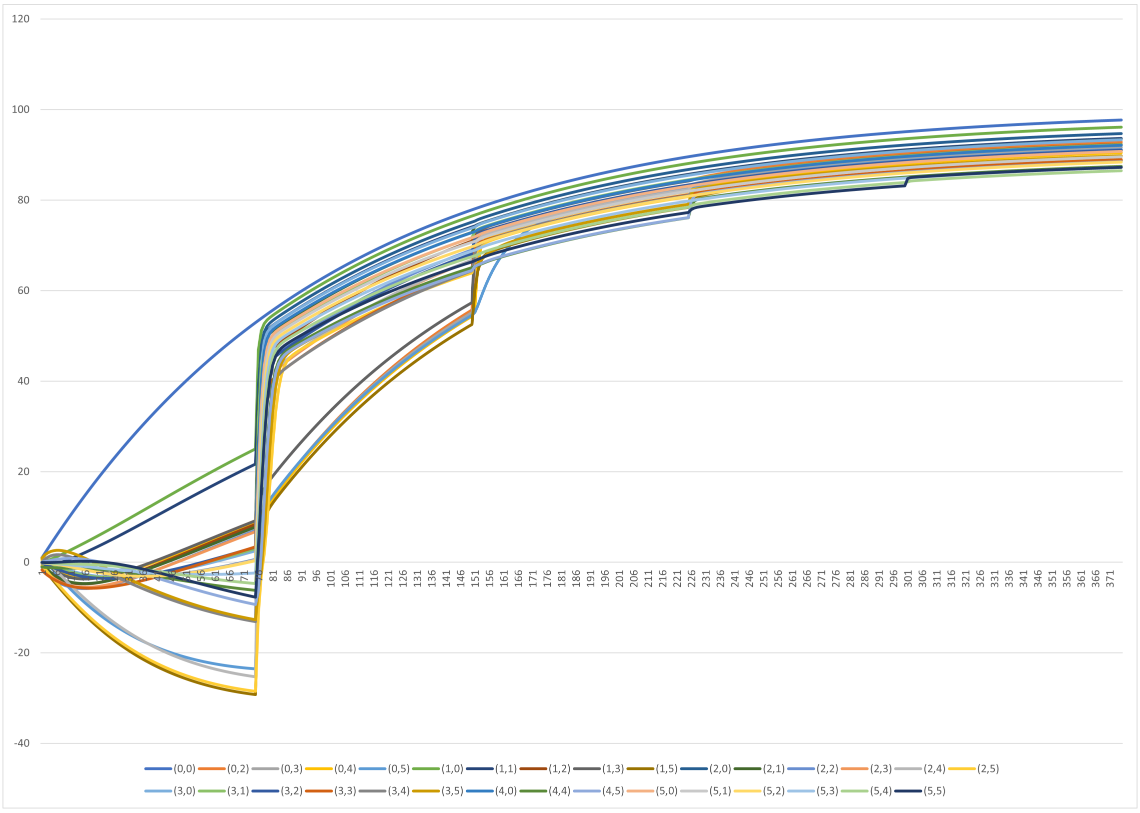


Figure 15: Utility estimates plot as a function of the number of iterations (policy iteration = 5)

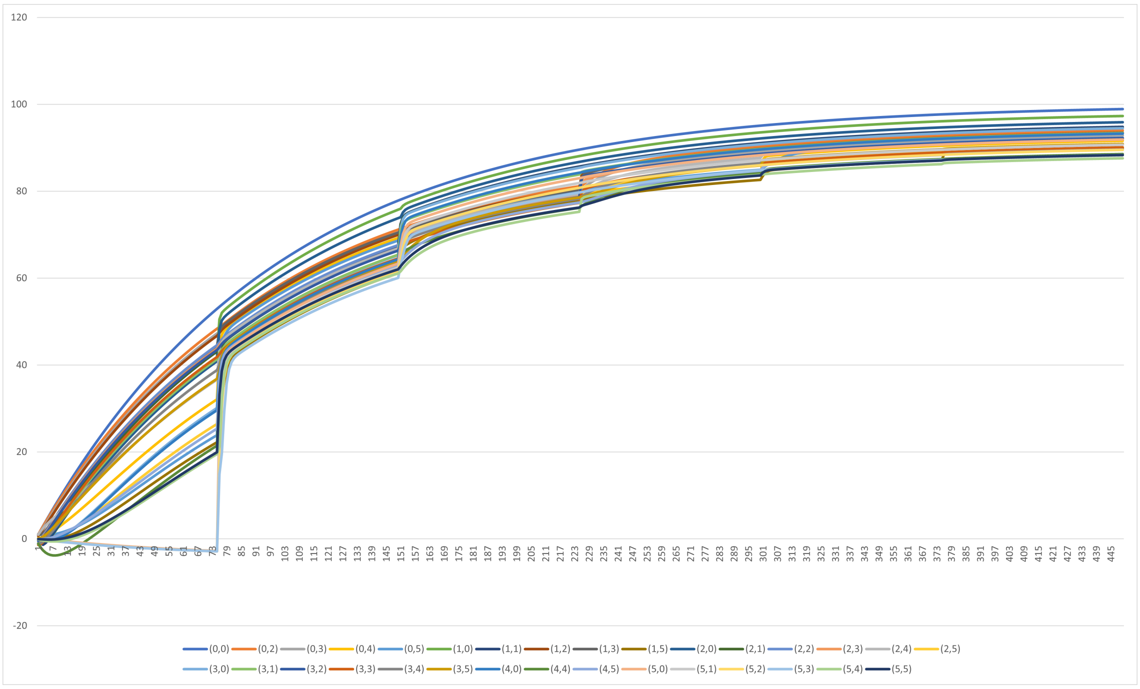


Figure 16: Utility estimates plot as a function of the number of iterations (policy iteration = 6)

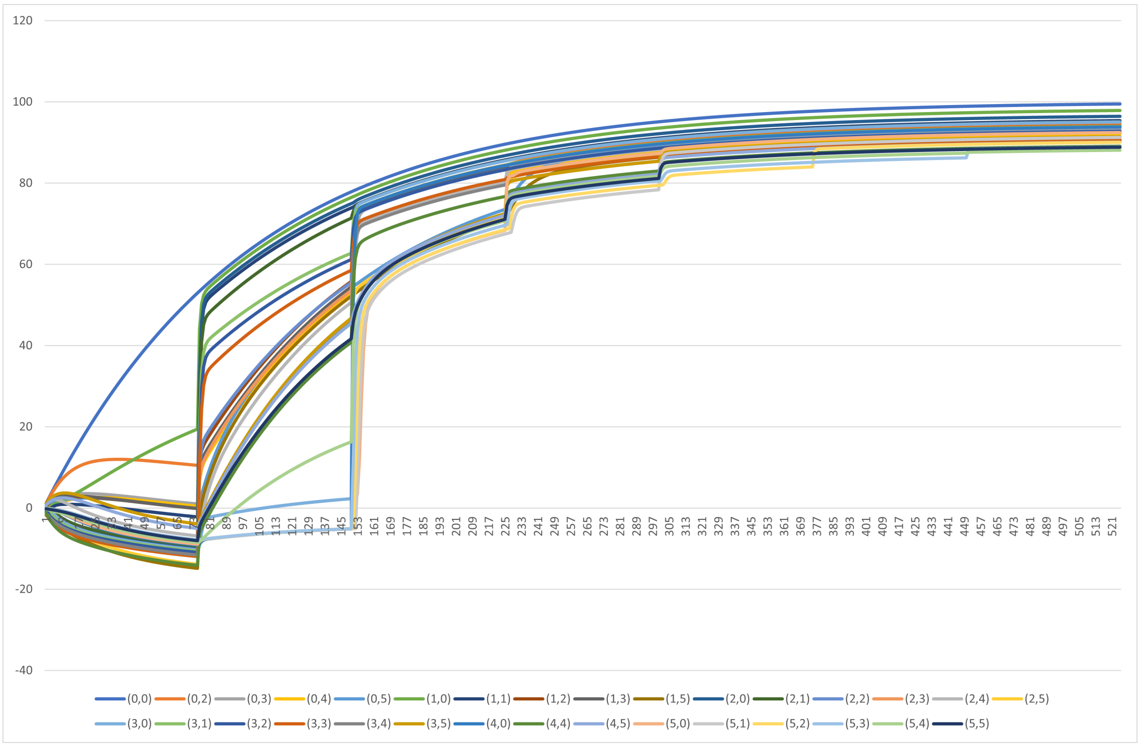


Figure 17: Utility estimates plot as a function of the number of iterations (policy iteration = 7)

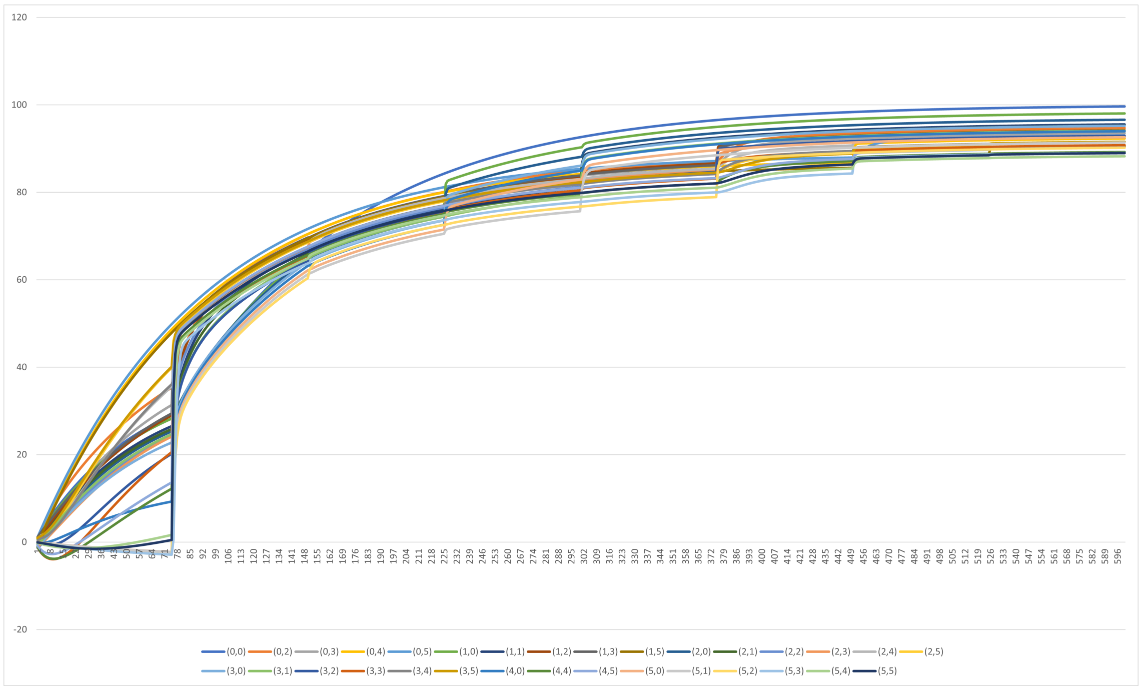


Figure 18: Utility estimates plot as a function of the number of iterations (policy iteration = 8)

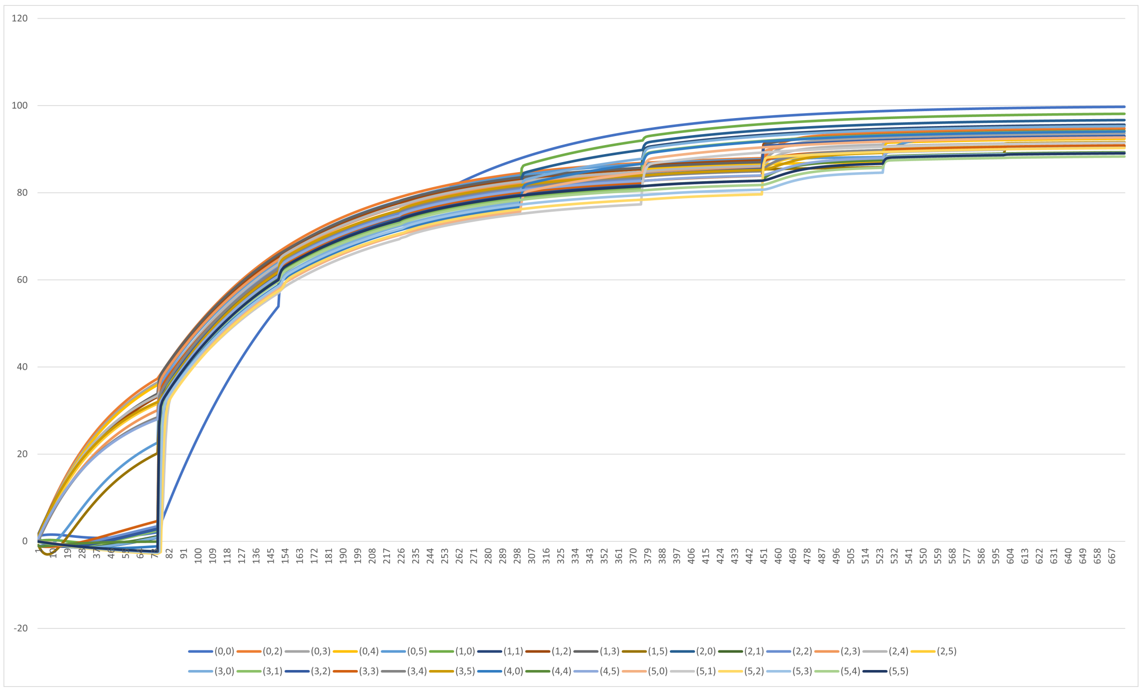


Figure 19: Utility estimates plot as a function of the number of iterations (policy iteration = 9)

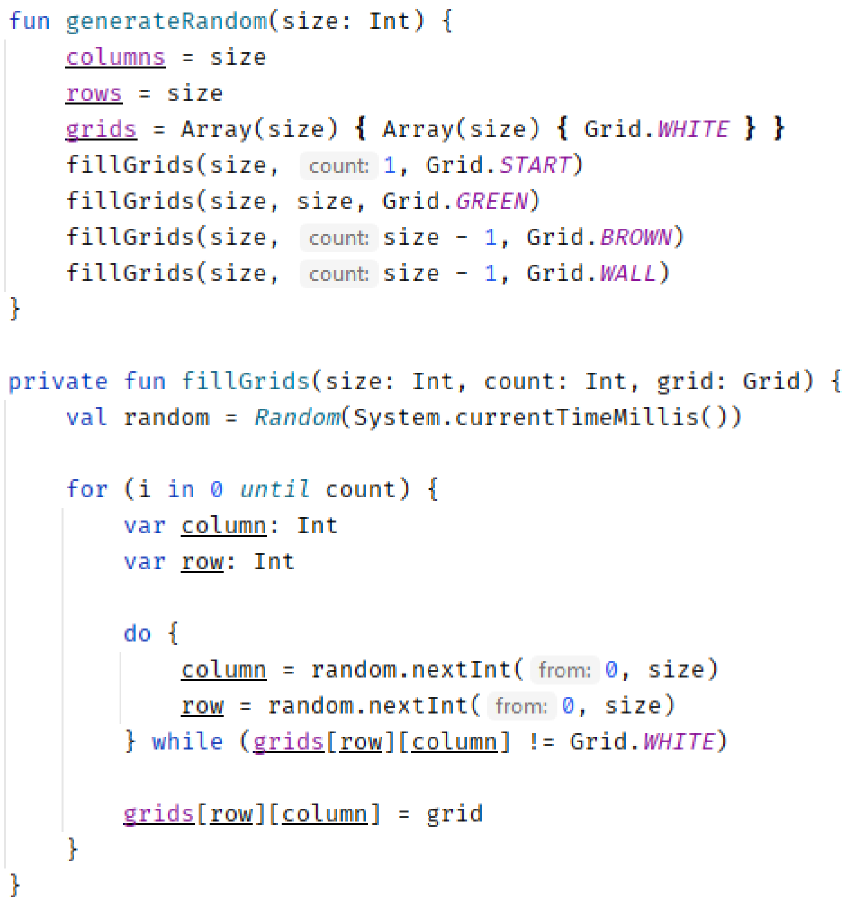
4. Bonus Question

4.1. Implementation

The application is capable of randomly generating a square-sized arena given an input size. For example, given input size = 10, the generated arena will be of size = 10 × 10. The start point, green spaces, brown spaces, and walls are randomly allocated across the arena with the following constraints:

- Start point randomly allocated on a white space
- No. of green spaces = input size (row = column)
- No. of brown spaces = input size - 1
- No. of walls = input size - 1

The code snippet for generating the arena is provided in Figure 20 below.



```
fun generateRandom(size: Int) {
    columns = size
    rows = size
    grids = Array(size) { Array(size) { Grid.WHITE } }
    fillGrids(size,  count: 1, Grid.START)
    fillGrids(size, size, Grid.GREEN)
    fillGrids(size,  count: size - 1, Grid.BROWN)
    fillGrids(size,  count: size - 1, Grid.WALL)
}

private fun fillGrids(size: Int, count: Int, grid: Grid) {
    val random = Random(System.currentTimeMillis())

    for (i in 0 until count) {
        var column: Int
        var row: Int

        do {
            column = random.nextInt( from: 0, size)
            row = random.nextInt( from: 0, size)
        } while (grids[row][column] != Grid.WHITE)

        grids[row][column] = grid
    }
}
```

Figure 20: Code snippet for random generation of arena

Arenas of various sizes were generated. Value iteration and policy iteration are both used to run on the generated arenas and the iterations needed to converge for both are recorded. For each arena, 10 tests are done for value iteration and policy iteration together.

4.2. Relationship between complexity & convergence

The observations made are illustrated in Table 6 below. Note that the parameters used for value iteration and policy iteration are the same as those described in chapters 2 and 3 (part 1 of the assignment).

Table 6: Relationship between arena complexity & convergence

Arena Size	**Iterations to Converge (Value Iteration)**	**Iterations to Converge (Policy Iteration)**
6 × 6	453 to 688	5 to 9
10 × 10	481 to 688	6 to 14
20 × 20	486 to 688	10 to 15
50 × 50	562 to 688	14 to 21
100 × 100	565 to 688	21 to 31
200 × 200	568 to 688	27 to 32
500 × 500	568 to 688	36 to 55

It is observed that the iterations to converge for value iteration does not exceed 688 as the complexity increases, while the iterations to converge for policy iteration steadily increases. From the test results illustrated in Table 6, it can be concluded:

- Value iteration: complexity slightly affects the lower bound for convergence as it increases, with no change observed for the upper bound.
- Policy iteration: convergence gradually increases as complexity increases

4.2.1. Complexity limit

At size = 500, the test runs each (value + policy iterations) took a significantly long time (> 2 minutes) to complete. While it is not indicative that the algorithm has failed to learn the right policy, it shows that the computational time for the algorithms is getting unfeasible. It is possible that the efficiency of the algorithm could be a limiting factor. Regardless, it can be concluded that for the implemented algorithms, they can feasibly learn the right policy for arenas with complexity $\leq$ 500 × 500.