NANYANG TECHNOLOGICAL UNIVERSITY
SINGAPORE

CZ4046 (Intelligent Agents)

Assignment 1 Submission

By

Name	Matriculation No.
Muqaffa Al-Afham Bin Kamaruzaman	U2022554C

Date of Submission: 16/03/2022

1 Solving for Given Environment

1.1 Value Iteration

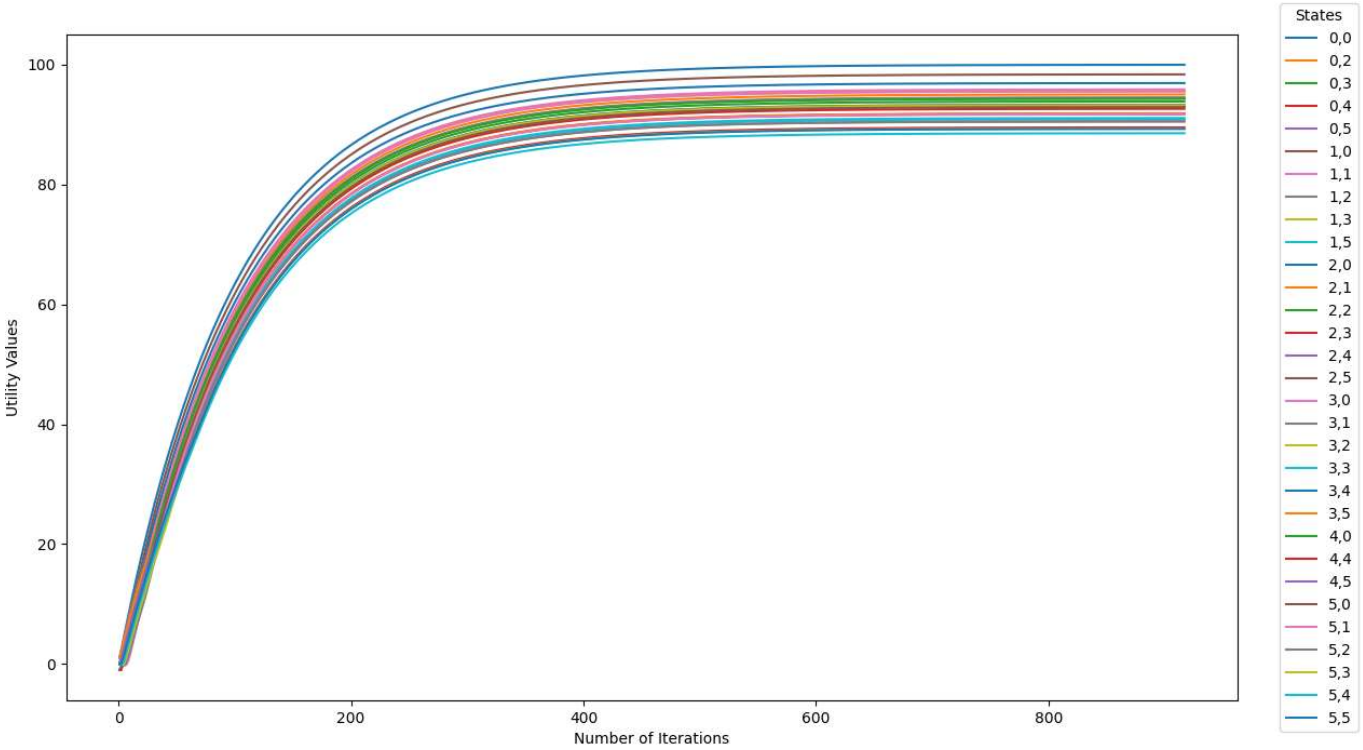
According to the reference book, we must first determine 3 constants, ε, γ, and c, before we can perform value iteration. γ is already given a value of 0.99 in the problem statement. Then we have $\varepsilon = c \cdot R_{max}$. But since R_{max} is just 1, we now have $\varepsilon = c$.

ε is the maximum error allowed in the utility of any state; thus, a smaller c will yield more precise utilities. However, a smaller c also requires more iterations. In this assignment, 0.01 was the value chosen for c, which turns out to require 917 iterations to converge. The resultant utilities and optimal policy for all states are shown below:

Utilities					
99.99		95.036	93.865	92.645	93.319
98.383	95.873	94.535	94.388		90.908
96.939	95.576	93.284	93.166	93.092	91.785
95.544	94.443	93.223	91.105	91.804	91.878
94.303				89.538	90.557
92.928	91.719	90.525	89.346	88.559	89.288

Optimal Policy					
↑		←	←	←	↑
↑	←	←	←		↑
↑	←	←	↑	←	←
↑	←	←	↑	↑	↑
↑				↑	↑
↑	←	←	←	↑	↑

Note that there are many paths that gravitate towards state (0, 0), which has the highest utility. This makes sense because it is the only green square that the agent can stay in forever (by always moving up). Below is the plot of utility values against the number of iterations, for each state:



1.2 Policy Iteration

For simplification and efficiency, modified policy iteration is used instead of the standard policy iteration. The value chosen for k – the number of times the simplified Bellman update is repeated – is 20. The following is an example of a random policy:

Random Policy					
↓		→	→	↓	↓
←	←	←	↑		↓
→	←	↑	↓	→	←
←	←	↑	→	↓	↓
↓				→	↓
←	↓	→	↑	↑	↑

After 7 iterations, the algorithm converges and produces the following utilities and policy:

Utilities					
70.523		**65.568**	**64.398**	**63.177**	**63.800**
68.917	**66.406**	**65.068**	**64.921**		**61.382**
67.472	**66.11**	**63.818**	**63.7**	**63.626**	**62.311**
66.077	**64.976**	**63.756**	**61.638**	**62.337**	**62.404**
64.836				**60.07**	**61.082**
63.461	**62.252**	**61.059**	**59.88**	**59.09**	**59.813**

Optimal Policy					
↑		←	←	←	↑
↑	←	←	←		↑
↑	←	←	↑	←	←
↑	←	←	↑	↑	↑
↑				↑	↑
↑	←	←	←	↑	↑

We can see that the resultant optimal policy is exactly the same as the one obtained in value iteration previously. However, compared to value iteration which required 917 iterations to converge, modified policy iteration is far superior in terms of computational costs. The utility values are of course very different, since modified policy iteration only performed 7 iterations, hence the utility values have yet to converge.

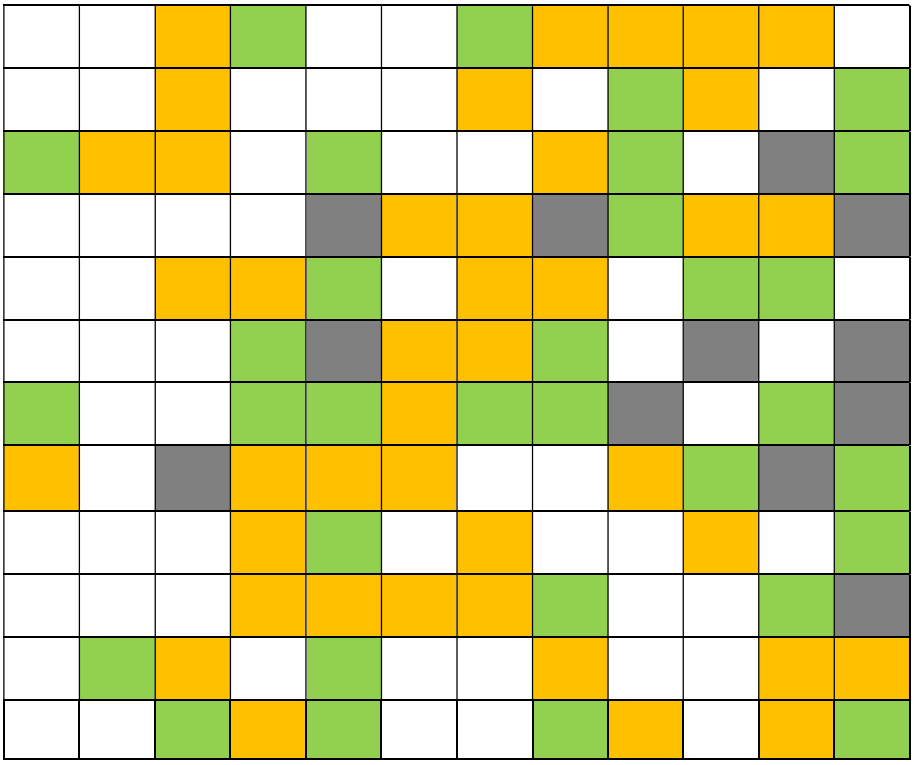
2 Solving for More Complicated Environment

2.1 Maze Generation

To make a more complicated maze, more states are added and the percentage of white cells is reduced as they are less interesting. In this assignment, 12 is selected for the new number of rows and columns, and the desired densities (frequencies) for each type of cell is as follows:

i. White cell: 40%
ii. Green cell: 25%
iii. Orange cell: 25%
iv. Wall: 10%

To simplify the process, a random maze generator is written, which (ideally) follows the aforementioned specification. The maze generator was run multiple times, but the main focus will be on the first maze that it generated, which was later hardcoded in the source file. This maze looks like this:



2.2 Value & Policy Iteration

Value iteration and policy iteration were both applied to this map, with the exact same parameters as before. It turns out value iteration ran 917 iterations, which is exactly the same as before! More random maps were generated, and each ran anywhere from 905 to 917 iterations, suggesting that the complexity of the environment has little to no impact on the number of iterations required to converge. This hypothesis was then tested on different values of C such as 0.001 and 0.0001, and the same observation was seen each time.

As for policy iteration, the number of iterations still remained relatively small, ranging from 3 to 20 iterations. The main factor was the initial random policy.

For each random map, the optimal policy produced by both value and policy iteration were at least *almost* exactly the same, suggesting that the optimal policies found were correct, and that the differences most likely resulted from the fact that a state could have multiple optimal actions.

3 Source Code Remarks

3.1 Programming Languages

The primary language chosen for this assignment is C++, with C++11 onwards being compatible with the project. However, due to lack of strong support for graph plotting, Python3 is also used, but only for graph plotting using the *matplotlib* library.

3.2 Legends (C++)

In the source code, some defined legends are used for convenience:

i. Actions are represented using integers 0 to 3, representing up, right, down and left respectively.

ii. Characters are used to map to cell types:

   a) ‘g’ represents a green cell

   b) ‘w’ represents a white cell

   c) ‘o’ represents an orange cell

   d) ‘0’ represents a wall

3.3 Aliases (C++)

For convenience, some aliases are defined and used for certain data types.

i. *HashMap<A, B>* is an alias for *std::unordered_map<A, B>*

ii. *vect2d<T>* is an alias for *std::vector<std::vector<T>>*

iii. *vect<T>* is an alias for *std::vector<T>*

The aliases are defined in the user-defined header *mdp.hpp*. You can read more on C++ aliases on *https://en.cppreference.com/w/cpp/language/type_alias*.

3.4 Code Separation (C++)

The value and policy iteration algorithms are encapsulated within the *Mdp* class, which is declared in *mdp.hpp* and implemented in *mdp.cpp*. The *Mdp* class also defines a few other functions for modularization. The *main()* function is placed in *IA_Assignment.cpp*.

3.5 Running the Program

Some binary files are already included in the *bin* folder:

i. *part1* uses the default maze provided in the problem statement
ii. *part2-fixed-maze* uses a pre-defined 12-by-12 maze
iii. *part2-random-maze* uses a random 12-by-12 maze

To run them in Linux, simply navigate to the project directory and run *./bin/<binary_file>*. If you wish to make some changes to the code, you can recompile it after you are done using ***g++ -o ./bin/<binary_file_name> ./src/*.cpp***. Of course, ensure that you have GCC installed first, or another C++ compiler of your preference.

Alternatively, or if you are on Windows, you may instead copy the project into a C++ supported IDE such as Visual Studio 2022 or Code::Blocks.

The plot of utility values against number of iterations is done in *plots.py*, which reads data from *value-iteration-output.txt* by default.