Part 0 - Setup your Environments

1.) Grid[][] is a matrix containing the 101x101 gridworld. Each member of Grid[][] is an object of class Cell. Cell represents a position in the actual gridworld containing h,f,g values, position in gridworld and the block status of the cell. The method initgrid(Start Position, End Position) initializes the 101x101 gridworld

The Function : initgrid(Start Position, End Position)
-Initializes Start Position.
-Initializes End Position.
-Set h values for each cell using Manhattan Distance.
-Set a cell Blocked/Unblocked with 30% probability.

2.) The gridworld grid[][] is stored in a text file using java IO library function storeObject(gridworld, filename). The Function : storeObject(gridworld,filename)

-Initializes file output stream.
-Initializes object output stream.
-Writes object gridworld to the file with name filename.

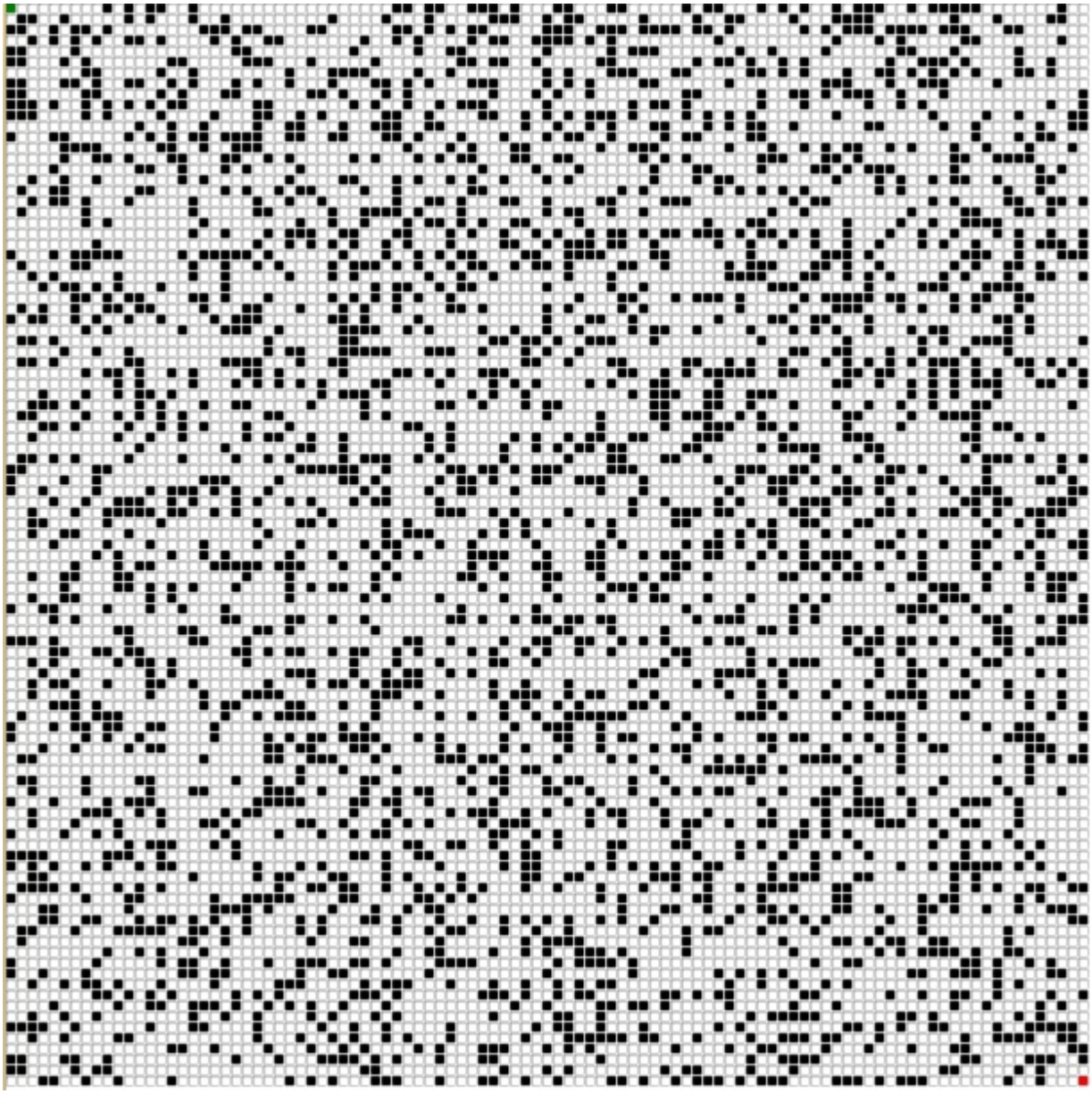
3.) The gridworld can be loaded from the text file using the loadObject(filename) function.

The Function : loadObject(filename)
-Creates file input stream.
-creates object input stream.
-Reads the Gridworld from the file and assigns it to grid[][].

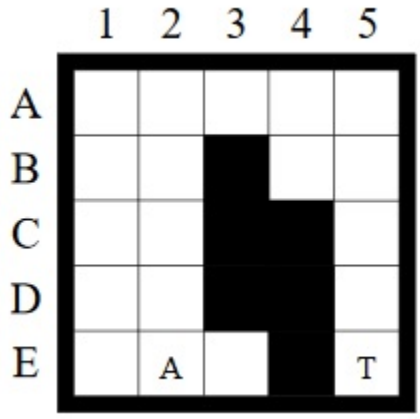
4.) The grid world is displayed using the javafx library.

The function : startstage
-Displays a Green square for start cell.
-Displays a red square for end cell.
-Displays a black square for blocked cells and white square for unblocked cells.

A sample gridworld :



Part 1 - Understanding the methods



a.) In the above gridworld the agent moves to the east and not towards north. Considering A as starting cell and T as target cell, the h value(manhattan distance) for the cell on the east is 2 and for the cell in the north is 4. This is because the agent does not know the blocked cells. As the h value for the cell on the east is less and the cost of moving to east or north is same the agent considers the cell on the east for the shortest path. So the agent moves towards he east first.

b.) The agent is guaranteed to reach the target if it is not separated from it by blocked cells. Consider that there is indeed a path from source to the destination. There are maximum 4 possible movements from a cell. Each cell is placed into the closed list only after considering all the four movements. A cell is dropped only after making sure that it is blocked or does not lead to the target. Hence no possible path is left out and we reach the destination in finite time as the gridworld is finite. Now consider there is no possible path to the target cell. In this case the target cell will never be traversed. Also a cell is traversed only once and as the gridworld is finite, after traversing all the possible cells we can determine that we cannot reach the target in finite time.

Part 2 - The Effects of Ties

Whenever there is a tie among cell regarding the g value, there are two approaches , Maximum G value and Minimum G value.

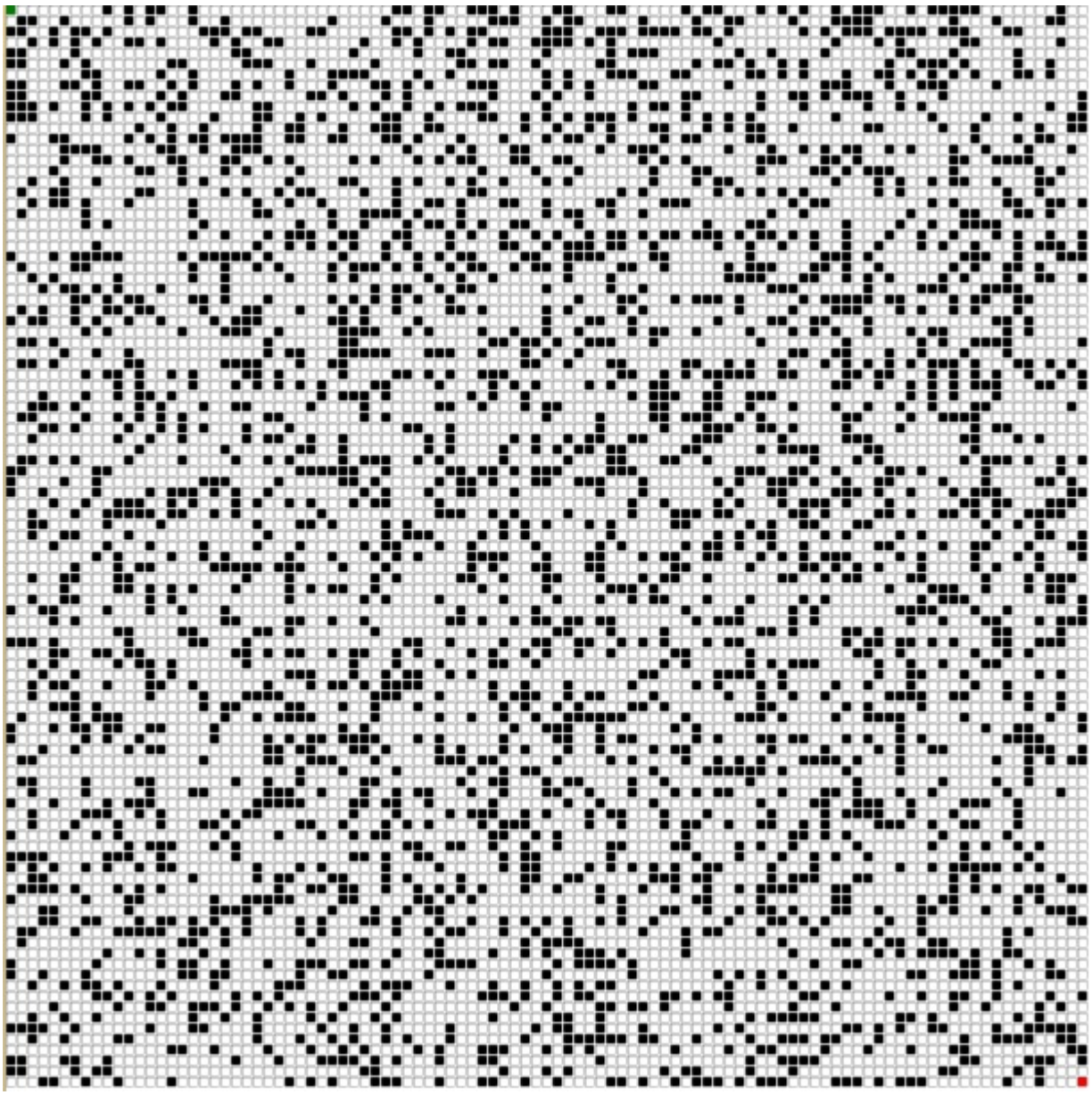
In case of the maximum G value approach, the algorith selects the next cell to be as far from the source cell as possible. This means that we can reach the destination faster. On the other hand if we are pursuing a possibly wrong direction this might mean more deviation from the path.

In case of minimum G value approach we select the cell which is nearest to the source. This can mean a few more movements to reach the destination. But also if we are in a wrong direction, this method deviates less for the optimal path.

Part 3 - Forward vs. Backward

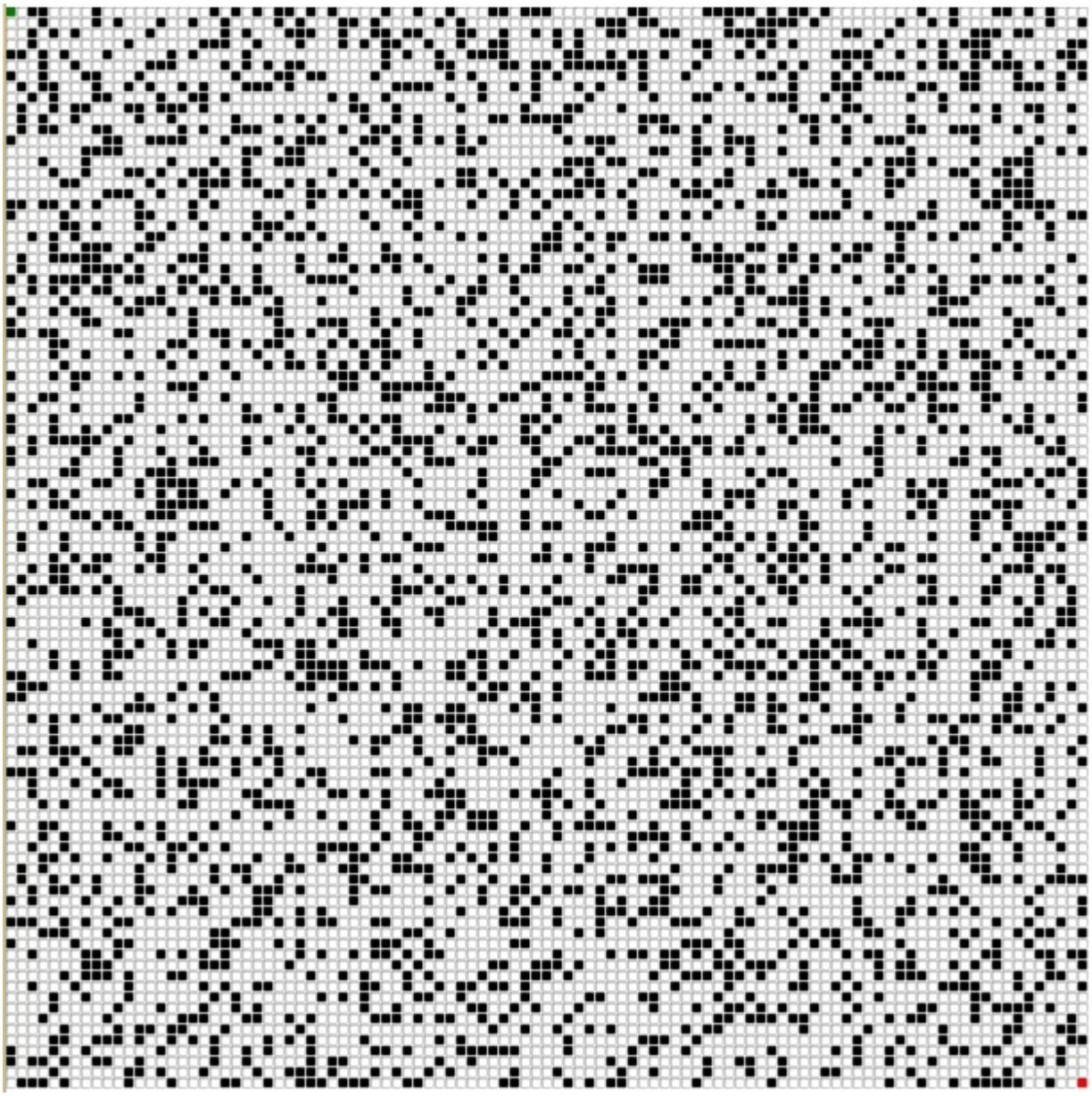
In case of forward repeated A* and backward repeated A*, which is better depends on the grid world and how the blocked states are scattered in the gridworld.

Consider the gridworld in g1.txt. In this case the number of cells visited in forward A* is 6817 and in backward A* is 6914. In this case the density of blocked cells near the starting cell is more so the initial branching of the tree is limited which results in less cells visited. If there is a case where the density of blocked cells near the source is very less than the branching of the traversal tree increases at the start. Also the depth of the tree remains same in both the case as the solution is optimal in either case.



Consider the gridworld in g5.txt. Here there is no path from source to destination. In such cases the number of cells visited depends on where the block is. If the block is near the source cell, as is the case with

g5, the number of cells visited in forward A*(9) is much less than the number of cells visited in backward A*(6843). This happens because the algorithm determines early that there is no path in forward A*.



If the block is near to the target cell (gridworld in g8.txt) the backward A* determines there is no path much faster.

Part 4 - Heuristics in the Adaptive A*

"The Manhattan distances are consistent in gridworlds in which the agent can move only in the four main compass directions.

In a manhattan grid where the total actions provided are the four main compass directions, the distance of the shortest path from one location to another can be calculated by the sum of the number of vertical movements and the number of horizontal movements required to reach the goal. Any path from start state to finish state will require atleast these many movements to reach the goal state, as the total number of actions available are north, south, east or west and no diagonal movements. So assuming the the goal state is 5 path distances to the north of the start state and 3 path distances to the east of the start state, then any path from start to goal will require atleast 5 north movements and 3 east movements.

The h-values hnew(s) ... are not only admissible but also consistent.
Prove that Adaptive A* leaves initially consistent h-values consistent even if action costs can increase.

In our case, we have the action cost for all the four actions equal. Considering the action cost equal to 1, the h values are consistent. If the action cost increases , lets say by a factor of k, then traditional h values(manhatten distances) will also increase by a factor of k as each action requires k cost. As the hnew(s) values are less then the manhatten distances they will be consistent even if the action cost increases.

Part 5 - Heuristics in the Adaptive A*

Repeated A* vs Adaptivr A*

The benefits of adaptive A* compared to Repeated Forward A* is that it reduces the overall runtime and reduces the cost to CPU on finding the shortest path to the goal, Adaptive A* achieves this because it reduces the cost of calculating the heuristic of each visited state, as it does not require the implementation of A* from start for each state but uses imformed heuristics to reduce the computation time for the path search. When we implemented both Repeated Forward A* and Adaptive A* on the same grid, we found that even though the diffence between the number of cells expanded during the two searches was not large, the overall computation time taken by Adaptive A* was lesser than that of Repeated Forward A*. This was because the computation required in Repeated Forward A* had been reduced in Adaptive A*, as it used informed heuristics for each expanded State, which was the difference between the final cost of start state and cost of path from start state to currenct state, hence saving computation time.

Part 6 - Memory Issues

In our problem, every grid has 101x101 cells. Each cell contains 4 integers - h value, f-value and x and y positions in the grid. The implementation of A* requires storage for the open list, closed list, and the list containing the path. Below are some suggestions to reduce the size of the implementation.

-

We can use boolean values or bitset for the closed list. This takes up 1 bit value for each cell. If cell is closed the value is 1 else the value is 0.

We can also use boolean values or bitset for the path list. If the cell is present in the path the value is 1 else it is 0. This reduces the memory usage to 1bit/cell.

Our implementation takes arround 155 KB for each 101x101 grid(16 Bytes per cell). The closed list is boolean and does not take up much space. The open list contains pointers to the original cell locations and does not take up much space. So for a 1001x1001 (10,01,001 cells) grid it takes around 15 MB of memory.

In a memory of 4 MB, a grid world of 512x512 cells can be accomodated. (2,62,144 cells each of 16 Bytes)