# *An Artificial Intelligence Simulated Game Based on Uniform Cost Search and Greedy Search in a Maze Environment*

Shyam Sundar Somasundaram[1], Ganesh Rakate[2], Faris Al Afif[3]

[1,3]Department of Computer Science and Engineering
[2]Department of Electrical and Computer Engineering
University of Minnesota - Twin Cities
Minneapolis, Minnesota 55455, US
Email: {somas009, rakat001, afifx001}@umn.edu

***Abstract*— Maze solving algorithm is used to find the shortest path between the source and target point in a given set of conditions of the maze. Maze solving techniques have been consistently evolved as they have always been immensely popular. In this paper, we have studied various standard Artificial Intelligence search strategies for the application of maze solving problems. We consider a particular maze solving problem statement which consists of a thief trying to find the treasure and exit point located inside a maze while trying to avoid the security. We have described our approach for solving these kinds of maze problems. We applied various standard informed and uninformed search strategies for this particular maze solving problem. The simulation results lead us towards a conclusion about the nature, behaviour, and efficiency of these algorithms. Upon considering all the regulating factors which can alter the performance of an algorithm, some proposals have been drawn. Our results show that the best suitable strategy for this kind of maze solving problem will be Uniform Cost Search and Modified Greedy Search.**

***Keywords*- Uniform-cost Search, Greedy Search, Heuristic Function, Maze, Simulation Game**

## I. INTRODUCTION

The problem of maze solving has always been an interesting topic among Artificial Intelligence researchers. Even before the field of Artificial Intelligence progressed fast in late nineteenth century, people were always fascinated about various maze solving methodologies. Sadik et al [1] defined a maze as a puzzled way which consists of different branch of passages where the aim of the solver is to reach the destination by finding the most efficient route within the shortest possible time. Video games industry is growing very fast in twenty-first century. So it is always interesting to design autonomous agents which can do the task of a human player in a video game. Regular attempts have been made to design such intelligent agents to play various games like Pac-Man [10][11].

Artificial Intelligence plays a vital role in finding the best possible way of solving any maze effectively. Various developed search strategies are being applied for solving various kinds of mazes. The graph theory also appears as an efficient tool while designing proficient maze solving techniques. Graph is a representation or collection of sets of nodes and edges and graph theory is the mathematical structure used to model pair wise relations between these nodes within the graph. By proper interpretation and mathematical modelling, it is possible to figure out the shortest distance between any of the two nodes. This concept is deployed in solving unknown maze consisting of multiple cells. Depending on the number of cells, maze dimension may be 8x8, 10x10, or 36x36. The cells of maze are considered as nodes which are isolated by walls or edges. This is similar as the interpretation of graphs onto a maze. This analogy between graph and maze provides the necessary foundation in developing maze solving algorithms.

Search strategies are divided into non-information-search (also known as blind search) ones and information search (heuristic) ones. While the shortest path search strategy is to find the most direct and shortest path from the starting point to the target point according to the terrain and obstacles in the map.

Our paper focuses on designing a search algorithm for an intelligent agent which can solve a maze problem. The maze description is explained in detail in section III. We have designed a modified search algorithm by combining Uniform Cost Search and Greedy search. The problem consists of a 10*10 maze consisting of walls and empty cells placed randomly. A thief (the intelligent agent) tries to find the treasure and exit cell while avoiding the security officer. So our algorithm tries to find the best possible algorithm for this task. We studied various informed and uninformed search strategies like Breadth-First search, Iterative deepening Search, Uniform Cost Search, Depth-First search, Depth-Limited Search, Bi-directional Search for the movement of the thief. For the movement of security, we studied various informed search strategies like A-star search, Greedy Search, etc. Based on various assumptions, calculation and testing, we finalized on Uniform-cost Search method for movement of thief and a modified Greedy Search method for movement of security officer.

The rest of this paper is organized as follows. In the following section, we discuss some related work with regard to various maze solving methodologies and commonly used Artificial Intelligence algorithms for this task. Section III describes the maze structure and problem statement in detail.

In this section we also discuss various assumptions made in problem statement. In section IV, we describe the Uniform Cost Search and Greedy search algorithm in detail. We also discuss how these algorithms are applied to solve the given problem statement. We have also mentioned the pseudo code of our total algorithm. We then discuss why Uniform Cost Search and Greedy search algorithm are more suitable as compared to other search strategies for the problem statement. Section V describes the software implementation of our algorithm. In section VI, we have discussed the experimentation on various randomly generated mazes. Then we discuss the results with respect to number of steps required to solve the problem statement. Finally section VII discusses the conclusion of our work.

## II. PREVIOUS WORK

The theory and analysis of maze solving algorithms is fairly well-developed in the Artificial Intelligence literature, and is not discussed in detail here. For any maze solving technique, it is required to minimize the number of steps taken by the agent to solve the particular task. A complete search algorithm will in finite time always find a solution path if one exists, and report failure (also in finite time) if no solution path exists.

The earliest maze solving algorithms were based on a simple principle of wall follower [12]. The wall follower, the best-known rule for traversing mazes, is also known as either the left-hand rule or the right-hand rule. If the maze is simply connected, that is, all its walls are connected together or to the maze's outer boundary, then by keeping one hand in contact with one wall of the maze the player is guaranteed not to get lost and will reach a different exit if there is one; otherwise, he or she will return to the entrance having traversed every corridor in the maze at least once. One of the oldest maze solving algorithm was the flood fill algorithm [15]. The development of Artificial Intelligence field included various informed and uninformed search algorithms. The shortest path algorithm [12] was based on finding the shortest path to the goal based on an admissible heuristic. When a maze has multiple solutions, the solver may want to find the shortest path from start to finish. One possible algorithm finds the shortest path by implementing a breadth-first search, while another, the A* algorithm, uses a heuristic technique. The breadth-first search algorithm uses a queue to visit cells in increasing distance order from the start until the finish is reached. Each visited cell needs to keep track of its distance from the start or which adjacent cell nearer to the start caused it to be added to the queue. When the finish location is found, follow the path of cells backwards to the start, which is the shortest path.

About solving a particular type of maze, various standard search algorithms were modified or combined with other algorithms. Sadik et al [1] discussed about solving the maze problem by implementation of graph theory. They worked on solving a particular type of three-dimensional maze using graph and non graph theory algorithms. Kuffner [2] explained how to exploit the structure of optimal paths on Euclidean-cost grids and lattices in order to reduce the number of neighbouring nodes considered during a node expansion step. The result is a moderate reduction in the total nodes examined, which reduces the overall memory requirements and computational cost of the search.

Micromouse is a robot which solved the maze. The maze solving algorithms have been widely applied to design faster Micromouse. Wyard-Scott and Meng [3] demonstrated methods of assigning and manipulating the artificial potentials to provide locally optimized path choices while maintaining the integrity of the potentials. The basic algorithm was improved by retaining information of the number of decisions that have been made. Kazerouni et al [4] designed a new method based on variable priority for the purpose of maze solving.

Various video games have been developed for maze solving. So it was obvious that Artificial Intelligence researchers tried to design algorithms for autonomous intelligent agents which would play those games. The Pac-Man game [10][11] is one of the most popular video games based on maze solving. For Pac-Man game, Bell et al [5] described a rule-based system, which involves utilizing Dijkstra's shortest path algorithm and a Benefit- Influenced Tree Search algorithm.

Dobrowolski [6] discussed about The Applicability of various Uninformed and Informed Searches to Maze Traversal. Liu et al [7] explained various modifications done to A star algorithm for fulfilling the task of search and rescue trapped persons in some dangerous situations, which can be abstracted as a maze. Zhang et al [8] designed an improved algorithm based on existing mathematical model inspired by an amoeboid organism, Physarum polycephalum, to solve maze solving problems. They adopted the positive feedback mechanism in the mathematical model.

## III. PROBLEM DESCRIPTION

This section describes the problem statement along with all the assumptions in detail.

### *A. Description of Maze*

There is a 10x10 maze, as shown in Figure 1, consisting of empty cells and walls placed at random locations. A thief enters the maze from one corner, tries to take the treasure placed at a random location in the maze and exit the maze through an exit point that usually located at the boundary of the maze.

The actual maze is an 11x11 matrix (including the outermost rows and columns which are considered as walls) consisting a total of 121 cells that is randomly generated each time. It consists of empty cells, walls, a cell that contains the treasure and a cell that denotes the exit point. The starting point of the thief is fixed at cell (1, 1). Each maze has a different location for treasure, exit point and starting point of security. The thief can move and see through empty cells. Walls restrict the movement and the vision of the thief. The number of empty cells and walls differs for each new maze

generated. The thief is represented by blue, treasure by yellow, exit point by orange, empty space by white, wall by grey, and later visited cell by light blue.

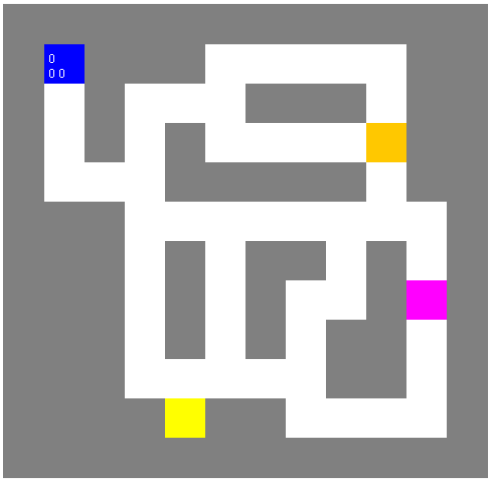

Figure 1. Structure of a typical Maze.

### B. Problem Statement

We need to develop an algorithm in which a thief tries to collect the treasure and to escape while saving himself from the security officer. The security tries to catch the thief. The algorithm should be able to perform the task with low number of steps.

### C. Assumptions

The following are the assumptions about the problem statement.

1) The thief can see up to 3 cells in all four directions.
2) The thief cannot see the cells behind the walls.
3) The thief can move and see only in horizontal and vertical direction.
4) The thief can only move one cell in a unit time.
5) The starting point for thief is fixed at cell (1, 1).
6) The exit point is always placed at the outermost row or column.
7) The outermost rows and columns (eleventh row and eleventh column) are considered as walls.
8) The security officer has complete knowledge about the maze.
9) The thief moves continuously without stopping till he finds the treasure and then the exit point or until he is caught by the security officer.
10) The directions are defined as standard wind directions: North, East, South and West.

## IV. SEARCH METHODOLOGY

In this section, we will be discussing the various search algorithms employed by us and how they are particularly applied to solve the search problem. Finally we will explain why we did not select other algorithms for this problem statement.

### A. Uniform Cost Search

Uniform-cost search (UCS) [9][14] is a tree search algorithm used for traversing or searching a weighted tree, tree structure, or graph. The search begins at the root node. The search continues by visiting the next node which has the least total cost from the root. Nodes are visited in this manner until a goal state is reached.

Typically, the Uniform cost search algorithm involves expanding nodes by adding all unexpanded neighbouring nodes that are connected by directed paths to a priority queue. In the queue, each node is associated with its total path cost from the root, where the least-cost paths are given highest priority. The node at the head of the queue is subsequently expanded, adding the next set of connected nodes with the total path cost from the root to the respective node. The uniform-cost search is complete and optimal if the cost of each step exceeds some positive bound ε. The worst-case time and space complexity is $O(b1 + C^*/\varepsilon)$, where $C^*$ is the cost of the optimal solution and b is the branching factor. When all step costs are equal, this becomes $O(bd + 1)$.

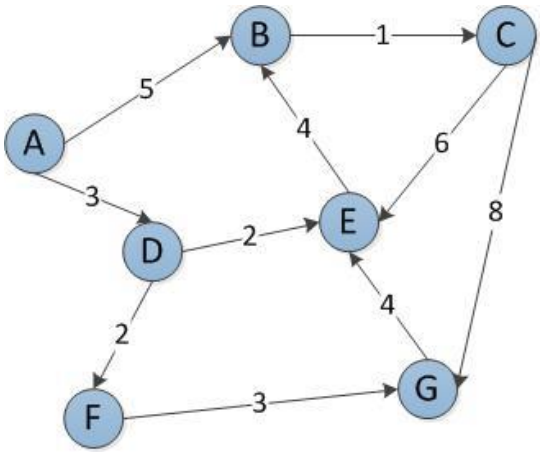


Figure 2. An example of Uniform Cost Search Algorithm.

Dijkstra's algorithm, which is perhaps better-known, can be regarded as a variant of uniform-cost search, where there is no goal state and processing continues until all nodes have been removed from the priority queue, i.e. until shortest paths to all nodes (not just a goal node) have been determined. As in Dijkstra's algorithm, Uniform cost search guarantees that (if all edge weights are non-negative) the shortest path to a particular node has been found once the node is extracted from the priority queue.

Uniform-cost search is a special case of the A* search algorithm if it's heuristic is a constant function. If A* is used with a monotonic heuristic, then it can be turned into a uniform cost search by subtracting from each edge cost the decrease in heuristic value along that edge. Breadth-first search (BFS) is a special case of uniform-cost search when all edge costs are positive and identical. Where BFS first visits the node with the shortest path length (number of nodes) from

the root node, Uniform cost search first visits the node with the shortest path costs (sum of edge weights) from the root node. Uniform-cost search is a variant of best-first search.

*B. Greedy Search*

A greedy best -first search algorithm [9][13] is a strategy based on a heuristic function which intends to find the optimal choice at each level and eventually find the goal. Greedy search does not always produce an optimal solution. However it is very useful to find a local optimal solution that approximates a global optimal solution in a reasonable amount of time.

A greedy search algorithm can be implemented using the queue data structure as follows:

1) Expand the first child of the parent. If the heuristic of the first child is better than the parent, then the child is set at the front of the queue and the loop is restarted.
2) Else the child is inserted into the queue in a location depending upon its heuristic function value and the other children of the parent are examined.

Greedy search fails to find the optimal solution almost every time because it does not examine all the nodes exhaustively. It can make commitments to certain choices too early which prevent them from finding the best overall solution later. It chooses certain paths which prevents it from finding the best overall solution later on. However it is still very useful as it can give some approximations to the optimal solution.

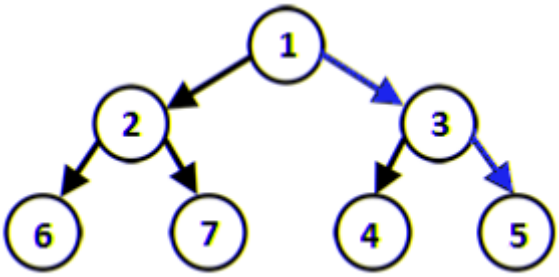


Figure 3. An example of Greedy Search Algorithm.

In Figure 3, if the goal was to reach the largest number, the greedy algorithm chooses 3 over 2 and chooses 5, but the actual largest number is 7.

*C. Search Strategy for Thief*

The movement of the thief emulates the uniform cost search algorithm. The thief uses Uniform Cost search to first locate the treasure and then locate or go to the exit point. A security officer has a job to catch the thief and guard the treasure. The security officer uses Greedy search to locate and catch the thief. While searching for the treasure and the exit point, the thief must also avoid the security officer.

At each step, the thief uses a weight evaluation function to choose the direction in which he moves. The thief moves in the direction with highest evaluation function value. The thief has direction preferences as - South, East, North and West. If two or more directions have same value from weight evaluation function, the thief will choose the direction based on the preference.

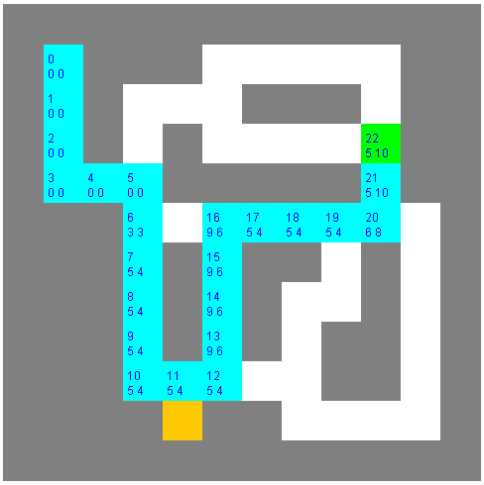


Figure 4. Search Strategy for Thief.

In searching the treasure, the thief moves one cell at a time in one of the four directions. The thief can see the three cells ahead of his position in all four directions, so there are 12 visible cells from the position of the thief. In the algorithm, there is a weight evaluation function using the information of visible cells to produce one best direction for the next movement.

1) Suppose three cells located to the North direction as N1, N2, and N3. We assigned the same condition to the other directions. So, we also have {E1, E2, E3}, {S1, S2, S3}, and {W1, W2, W3}. The best direction for the next movement is the direction that has the largest accumulated weight value of its three cells. So, we are comparing the value of (N1+N2+N3), (E1+E2+E3), (S1+S2+S3), and (W1+W2+W3).
2) However, we are creating a tendency that the thief by default should not go backward (reversing direction). The thief is only allowed to go backward only if the cell that is located forward, right, and left of him is a wall. So, actually now there are 3 default next directions that will depends on the current direction. For example, if the current direction is N, the possible next directions are: W, N, E. If the accumulated weight value is equal between any two directions, we choose the following as movement priority: move forward, move to the right, move to the left. For example, if current direction is N, the order of priority for the next direction is: N, E, W.
3) During the movement, thief will remember any branch he encounter and record all unexplored path (unselected adjacent cell). This means that we want to explore one branch as the best possible path first and we will always be able to go back (backtrack) and explore the other unexplored path later. So, in case of dead end, the thief will go back to the latest

encountered branch as the storing of the branch is simulated as a stack.

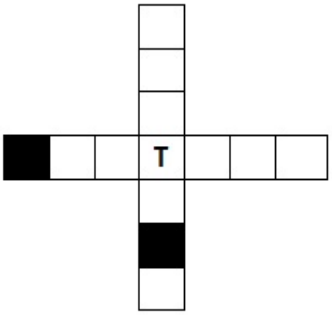


Figure 5. The vision of the thief on all four directions before he chooses the best path using UCS algorithm.

The important step is in assigning the weight value for each cell. We have 6 different cells that need to be assigned weight value. These cells are: the treasure cell, exit cell, empty cell, visited cell, wall cell, and security officer cell. First we grouped which cell should be considered as to be visited or to be avoided. We assigned positive value for the cells that are considered as to be visited: treasure cell, exit point cell, empty cell, and visited cell. We assigned negative value for the cells that are considered as to be avoided: wall cell and security officer cell. Then, we established the order of weight. We applied the following order: Treasure cell (Or Exit Cell) > Empty Cell > Visited Cell > Wall Cell > Security Cell.

*D. Search Strategy for Security*

The movement of the security emulates Greedy Best-First Search. At each step the security has 4 paths to choose from viz. North, South, East, and West. He uses greedy search algorithm to select the path which will take him closest to the thief. The heuristic function used is the minimum number of tiles from the security to the thief.

h(n) = Minimum number of tiles from the security to the thief

Direction preferences are given to help the security choose when more several paths have the same value of heuristic.

There are few limitations of the above strategy.

1) Assigning direction preferences could result in loops and hinder the movement of the security. In order to break of the loop, the security is forced to move out of the loop ignoring the heuristic function.

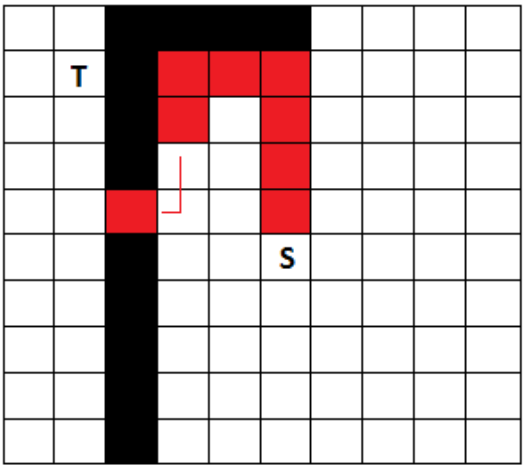


Figure 6. The security tries to move out of loop using blind movement.

2) While the security is trying to break out of the loop, the thief could move in a direction which will put him in and advantageous position.

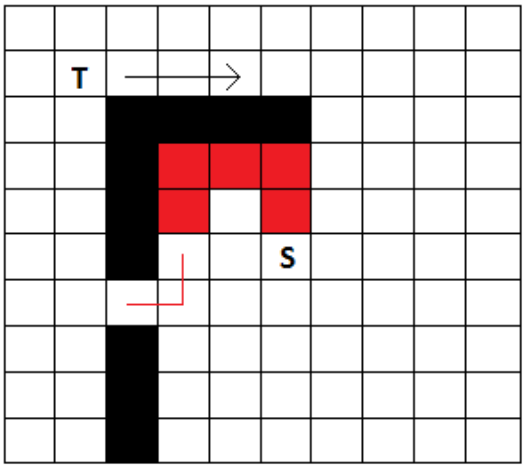


Figure 7. The thief gains an advantage as he moves east while the security tries to break out of loop.

In Figure 7, if the thief moves east while the security tries to break out of the loop, the security loses ground on the thief and the thief can escape easily.

3) Another scenario is when the security hits a dead end trying to chase the thief.

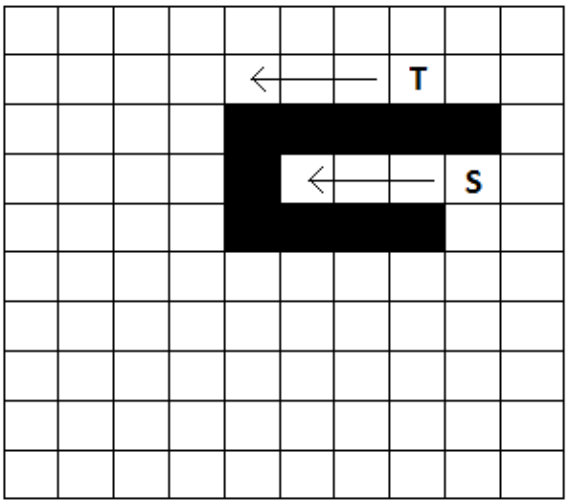


Figure 8. The security getting caught in a dead end while trying to chase the thief.

In Figure 8, the thief moves west, the security also moves west trying to catch the thief and hits a dead end. The security has to backtrack to come out of this situation and this allows the thief to escape easily.

*E. Pseudocode of our Algorithm*

The following pseudocode is used by the thief in searching the treasure.

**function** FIND-TREASURE(*problem*) **returns** a solution, or failure
*cell* ← a cell of thief's current position
*adjacent-cell* ← four adjacent cells of the current cell
*next-cell* ← a selected cell to be visited for the next thief's move
*previous-direction* ← previous direction of the thief's move
*updated-direction* ← direction to be used for the next thief's move
*branch* ← a stack of stored branch of cells that is not the best cell according direction selection function, used for continuing maze exploration and backtracking.
*visited* ← an empty set at initial, and later used to store visited cells during the search

**loop do**
**if** EMPTY?(*branch*)
**then return** failure
**if** CAPTURED(*cell*.STATE)
**then return** failure
/* Captured by security officer */
**if** EXIT-FOUND(*cell*.STATE)
**then** STORE-EXIT(*cell*.STATE)
/* Storing exit point */
*cell* ← POP(*branch*)
**if** *problem*.GOAL-TEST(*cell*.STATE, *treasure*)
**then return** SOLUTION(*cell*)
/* Treasure is found */
add *cell*.STATE to *visited*

**for each** action **in** *problem*.ACTIONS(*cell*.STATE)
**do**
*adjacent-cell* ← ADJACENT-CELL(*problem*, *cell*, *action*)
**if** all *adjacent-cell*.STATE is *visited*
**then**
THIEF-BACKTRACK*( POP(branch) )*

/* Perform backtrack */
**else** *next-cell*, *updated-direction* ← UPDATE-DIRECTION( *adjacent-cell*, *previous-direction* )
**then**
THIEF-MOVE( *next-cell*, *updated-direction* )
PUSH( *branch*, *adjacent-cell* )

The second pseudocode is used by the thief in searching the exit point and executed after the treasure has been found.

**function** FIND-EXIT(*problem*) **returns** a solution, or failure
*cell* ← a cell of thief's current position
*adjacent-cell* ← four adjacent cells of the current cell
*next-cell* ← a selected cell to be visited for the next thief's move
*previous-direction* ← previous direction of the thief's move
*updated-direction* ← direction to be used for the next thief's move
*branch* ← a stack of stored branch of cells that is not the best cell according direction selection function, used for continuing maze exploration and backtracking.
*visited* ← an empty set at initial, and later used to store visited cells during the search

**loop do**
**if** EMPTY?(*branch*)
**then return** failure
**if** CAPTURED(*cell*.STATE)
**then return** failure
/* Captured by security officer */
**if** EXIT-FOUND-CHECK()
**then** GO-TO(*exit-cell*)
/* Directly to exit point */
**else**
*cell* ← POP(*branch*)
**if** *problem*.GOAL-TEST(*cell*.STATE, *exit*)
**then return** SOLUTION(*cell*)
/* Exit is found */
add *cell*.STATE to *visited*

**for each** action **in** *problem*.ACTIONS(*cell*.STATE)
**do**
*adjacent-cell* ← ADJACENT-CELL(*problem*, *cell*, *action*)
**if** all *adjacent-cell*.STATE is *visited*
**then**

```
        THIEF-BACKTRACK( POP(branch) )

        /* Perform backtrack */
        else next-cell, updated-direction ← UPDATE-
        DIRECTION( adjacent-cell, previous-direction )
         then
        THIEF-MOVE( next-cell, updated-direction )
        PUSH( branch, adjacent-cell )
```

The last pseudocode is used by the security to capture the thief.

```
function FIND-THIEF(problem) returns a solution, or failure
cell ← a cell of security's current position
adjacent-cell ← four adjacent cells of the current cell
next-cell ← a selected cell to be visited by the next security's
move
previous-direction ← previous direction of the security's
move
updated-direction ← direction to be used for the next
security's move

loop do
        if EMPTY?(branch)
        then return failure
        if problem.GOAL-TEST(cell.STATE, thief)
        then return SOLUTION(cell)
        /* Thief is captured */

for each action in problem.ACTIONS(cell.STATE)
         do
        adjacent-cell ← ADJACENT-CELL(problem, cell,
        action)
        else next-cell, updated-direction ← UPDATE-
        DIRECTION( adjacent-cell, previous-direction )
        then
        SECURITY-MOVE( next-cell, updated-direction )
```

*F. Other Algorithms*

1) Breadth-First Search and Iterative Deepening Search

Breadth-first Search [9] is a simple strategy in which the root node is expanded first, then all the successors of the root node are expanded next, then their successors, and so on. In general, all the nodes are expanded at a given depth in the search tree before any nodes at the next level are expanded.

Iterative deepening search or Iterative Deepening Depth-First Search [9] is a general strategy often used in combination with depth-first search that finds the best depth limit. It does this by gradually increasing the limit until a goal is found. Iterative deepening combines the benefits of depth-first and breadth-first search.

If the thief uses BFS or IDS, there will be a repetition of steps and this in turn would increase the overall step cost. For instance in Figure 9, the thief would try to move to one of the red squares and then to the other red square by going through to the square in which he is currently located. In general when a node has more than one child, it would result in visiting the parent node several times depending on the number of the child nodes.

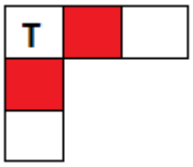


Figure 9. Movement pattern of BFS and IDS algorithms.

2) Depth First Search and Depth Limited Search

Depth-First search [9] is a strategy that always expands the deepest node in the current fringe of the search tree. The search proceeds immediately to the deepest level of the search tree, where the nodes have no successors. As those nodes are expanded, they are dropped from the fringe, so then the search "backs up" to the next shallowest node that still has unexplored successors.

To avoid the problem of unbounded trees we can supply depth-first search with a predetermined depth l. That is the nodes at depth l are treated as if they have no successors. This approach is called depth-limited search. The depth limit solves the infinite-path problem.

If the thief uses DFS or DLS, he might give preference to depth and ignore the better paths. In Figure 10, the thief sees the treasure but continues to move towards South since he gives preference to depth. But this is not an efficient move.

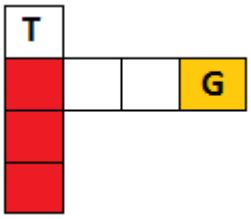


Figure 10. Movement using DFS or DLS algorithms.

3) Bidirectional Search

Bidirectional search [9] is a strategy which involves running two simultaneous searches – one forward from the initial state and the other backward from the goal, stopping when the two searches meet in the middle.

Bidirectional search is not suitable because the thief and the security are not supposed to meet.

## V. PROGRAM IMPLEMENTATION

The program is implemented in Java 1.7 with Java Applet for the graphical interface. Several mazes with different configuration are created and used for experimentation. Certain parameters such as colour of the objects, id of input maze, and movement speed can be edited in the .html applet

launcher file. This will allow changes when experimenting with the program without need to re-compile the java program.

There are two main part of the program. The First part is for the thief to search and find the treasure. After the treasure is found, second part of the program is executed as the thief may continue the mission now to search and find the exit point. During those two missions, security officer will always try to capture the thief thus the part of the program which represent the function of finding thief is always executed.

## VI. EXPERIMENTATION & RESULTS

First, we would like to compare the number of step required by thief to complete the mission of finding treasure and exit point. There are three algorithms used in comparison: exhaustive search, our algorithm (initial), and our algorithm (improved). We run these algorithm for Maze A, B, and C, shown by Figure 11, Figure 12, and Figure 13. Due to limited space, only the result of the third algorithm is shown. The result can be seen in Figure 14 to Figure 21.

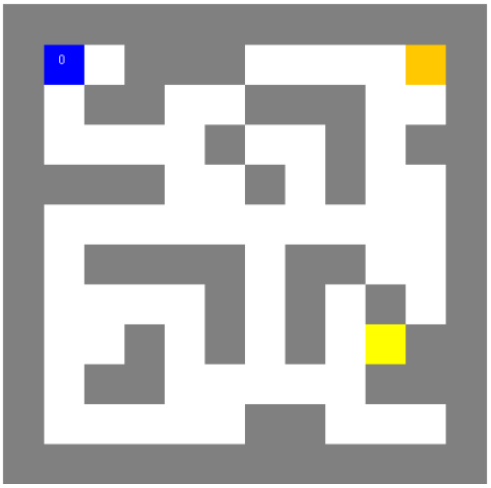

Figure 11. Configuration of maze A.

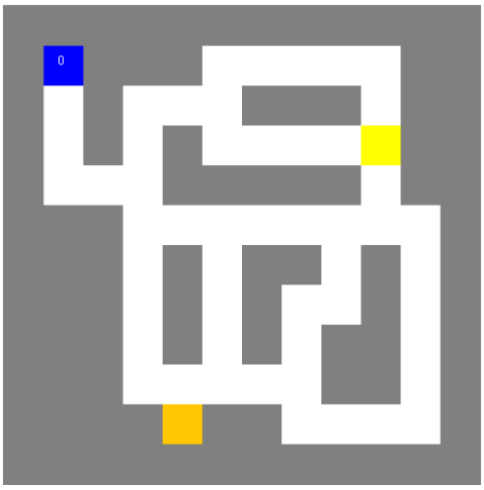

Figure 12. Configuration of maze B.

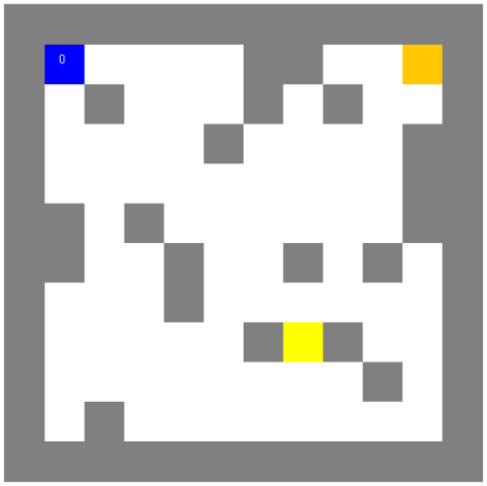

Figure 13. Configuration of maze C.

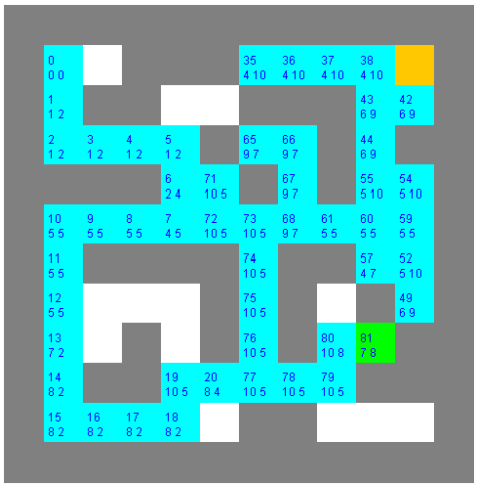

Figure 14. Maze A after the treasure is found.

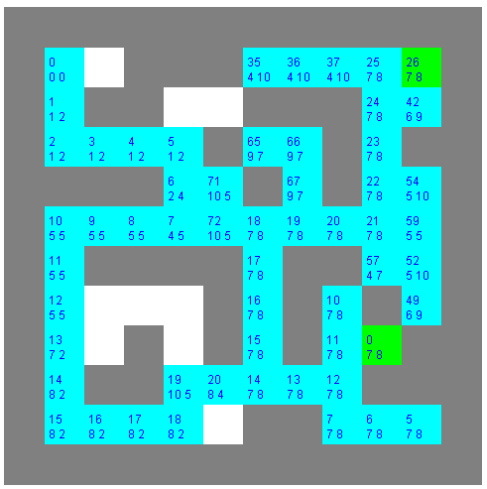

Figure 15. Maze A after the exit point is found.

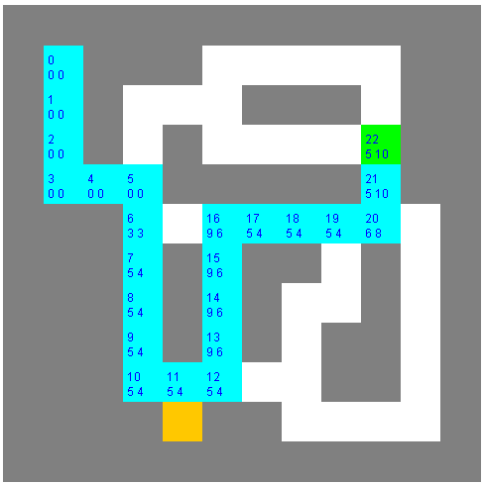

Figure 16. Maze B after the treasure is found.

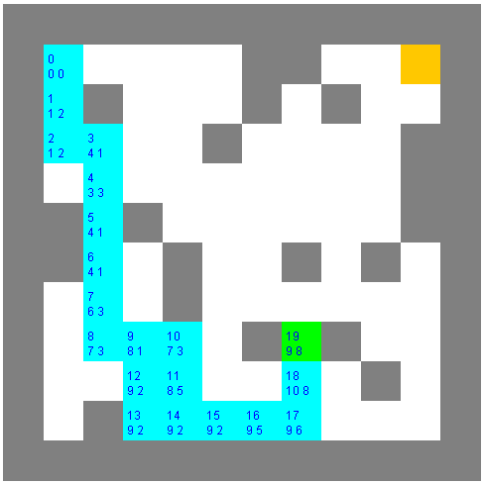

Figure 19. Maze C after the treasure is found.

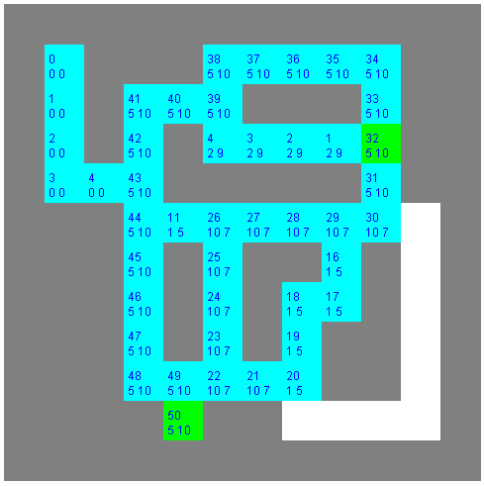

Figure 17. Maze B after the exit point is found.

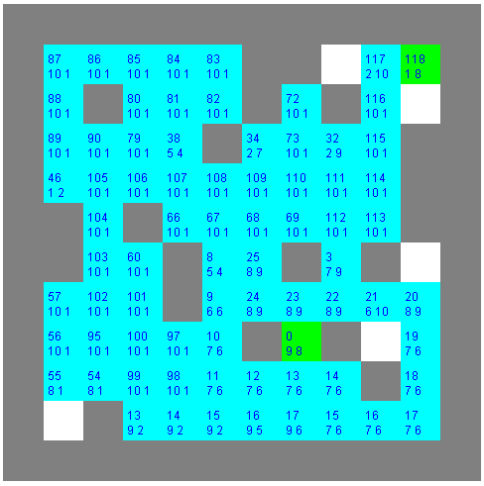

Figure 20. Maze C after the exit point is found.

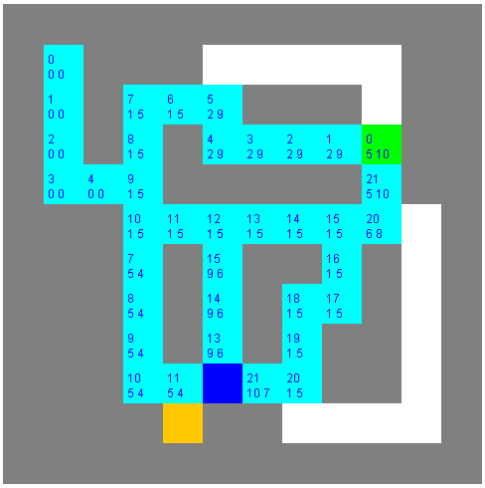

Figure 18. Behaviour of thief in maze B.

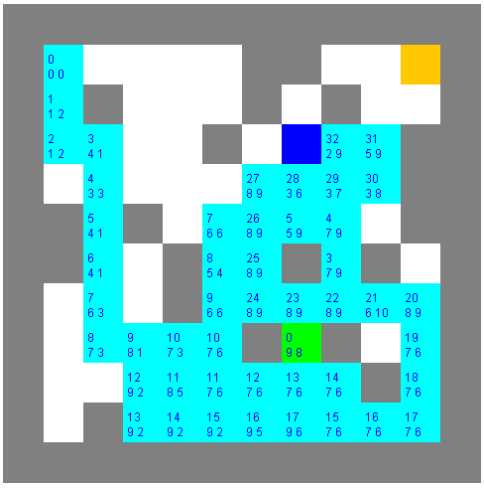

Figure 21. Behaviour of thief in Maze C.

Table 1. Comparison of required number of steps between three algorithms.

| No. | Maze | Number of steps required | | | | | |
|---|---|---|---|---|---|---|---|
| | | Exhaustive Search | | Our algorithm (initial) | | Our algorithm (improved) | |
| | | Trea-sure | Exit | Trea-sure | Exit | Trea-sure | Exit |
| 1. | A | 23 | 28 | 109 | **26** | 81 | **26** |
| 2. | B | 28 | **28** | **22** | 50 | **22** | 50 |
| 3. | C | 53 | 22 | 23 | 132 | **19** | 118 |

In finding the treasure, exhaustive search takes fewer steps than our algorithm for maze A. This is because the default preference of the exhaustive search algorithm that suit with the configuration of maze A thus making maze A more favourable for exhaustive search algorithm. Exhaustive search takes more steps than our algorithm for maze B and C. In all maze, our improved algorithm is able to reduce the required steps from the previous algorithm, such as from 109 to 81 and from 23 to 19.

In finding the exit point, exhaustive search takes fewer steps than our algorithm for maze B and C. This is also caused by configuration of maze B and C that is more favourable for exhaustive search related to its default movement preference. However, exhaustive search takes more steps than our algorithm for maze A. In addition our algorithm with backtrack is more effective in reducing branch, guarantee a completeness of searching in the maze, and later able to provide a capability of avoiding the security officer.

Next we want to show two different scenarios to test the overall implementation of thief's function (searching for treasure and exit point) and security officer's function (searching for thief). Maze B is used for first scenario while maze C is used for second scenario. Initial configuration of each scenario is shown in Figure 22 and Figure 24.

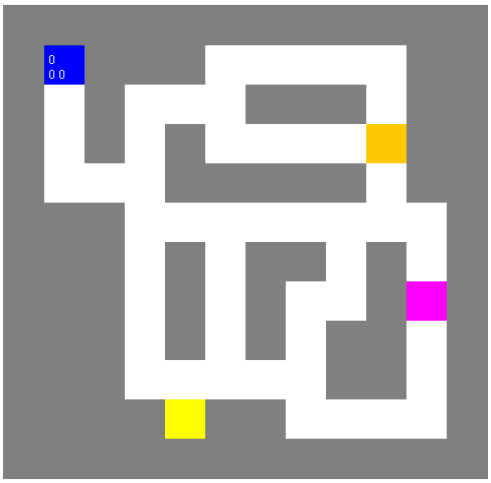

Figure 22. Initial configuration of maze B with the security officer.

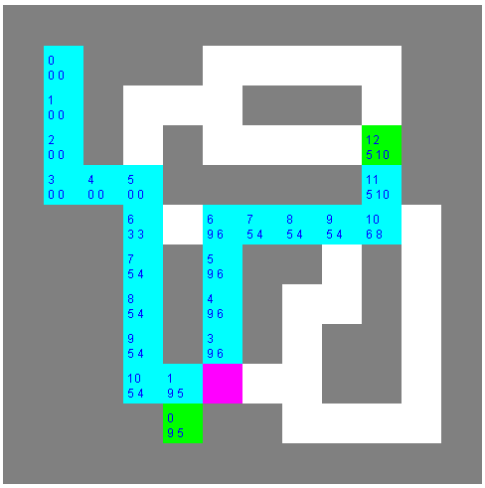

Figure 23. Final configuration of Maze B when the program is terminated.

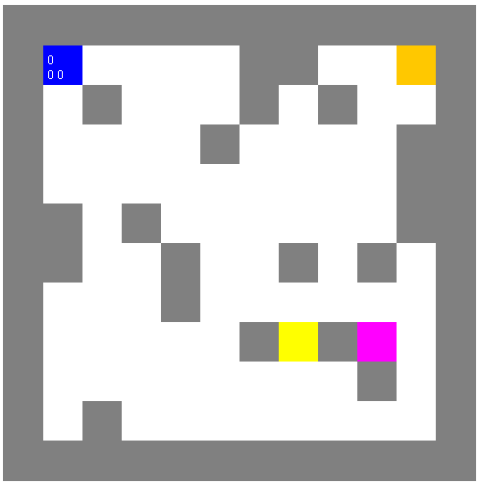

Figure 24. Initial configuration of maze C with the security officer.

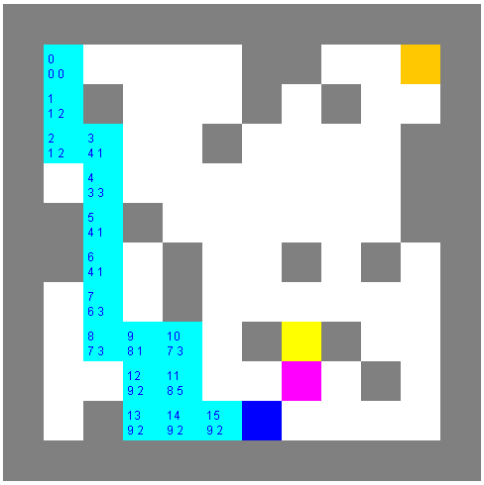

Figure 25. One step before program is terminated in Maze C.

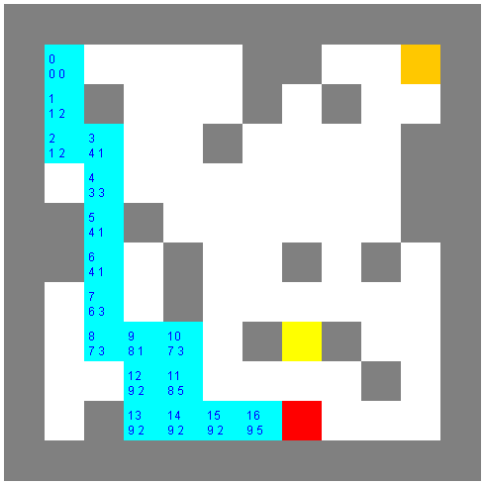

Figure 26. Final configuration of Maze C when the program is terminated.

From Figure 22, we can see that initially the thief (indicated by blue color) has a mission to find the treasure (indicated by yellow color). After that, the thief should go to the exit point (indicated by orange color). Both missions should be completed without being captured by the security officer (indicated by purple color). We can see from Figure 23 that the thief was able to find the treasure and the exit point while avoiding the security. This scenario is one of the examples that show the thief is able to avoid the security officer and the thief can successfully find the treasure and exit point.

From Figure 24, similar to the previous scenario, we can see that the thief has a mission to find the treasure and exit point. We can see from Figure 26 that the thief was captured by security and unable to find the treasure and the exit point. This is one of the conditions that show the thief is unable to complete the mission because the thief is captured by the security officer. Specifically, for this condition, this is because the thief has only limited vision, i.e. to the four directions (vertical and horizontal). As shown in Figure 25, the thief comes from the direction west to east, while the security is approaching from north to south. Because the thief has no diagonal vision and the security is able to capture the thief in the cell indicated by red color.

## VII. CONCLUSION

We have described in detail two search algorithms suitable for the thief and the security depending upon assumptions and their individual goals. Uniform Cost Search has been used to help the thief decide which path is best to proceed and also avoid the security. It encourages the thief to avoid visiting previously visited cells and visit unvisited cells and helps him explore the map better. The thief also has a vision up to 3 cells in each direction which will help him spot the treasure from a distance and move away from the security when he sees him.

Greedy Search algorithm has been used for the security to help him catch the thief. At each step the security chooses the path which takes him closer to the thief. Direction preferences are given to help the security to choose between multiple optimal paths.

Several problems were encountered during the implementation of the search algorithms. The movement of the thief and the security was hindered by loops. These were address by allowing the thief to backtrack and the security to use blind movement (ignoring heuristics) to come out of the loop situations. In addition we have also compared our algorithms with other informed and uninformed search strategies and the algorithms we have chosen have proven to be successful.

Finally we would like to extend our work in the future by trying new algorithms such as Poly-Iteration and Value-Iteration Algorithms which are based on probability rather than weights. We would also like to test our search algorithms on larger and more complex mazes involving more than one security.

## ACKNOWLEDGMENT

We would like to thank to our project guide Prof. Nikolaos Papanikolopoulos for his enormous encouragement and guidance through the course of the project.

## REFERENCES

[1] Sadik, A.M.J. ; Dhali, M.A. ; Farid, H.M.A.B. ; Rashid, T.U. ; Syeed, A.; “A Comprehensive and Comparative Study of Maze-Solving Techniques by Implementing Graph Theory”; International Conference on Artificial Intelligence and Computational Intelligence, Oct. 2010, Vol.1, pp.52-56

[2] James J. Kuffner; “Efficient Optimal Search of Uniform-Cost Grids and Lattices”; IEEE/RSJ International Conference on Intelligent Robots and Systems, 2004.(IROS 2004). 2004; volume-2; pp: 1946-1951.

[3] Wyard-Scott, L. ; Meng, Q.-H.M.; “A potential maze solving algorithm for a micromouse robot”; IEEE Pacific Rim Conference on Communications, Computers, and Signal Processing 1995, pp.614-618

[4] Kazerouni, B.H. ; Moradi, M.B. ; Kazerouni, P.H.; “Variable Priorities in Maze-Solving Algorithms for Robot's Movement”; IEEE International Conference on Industrial Informatics, 2003, Aug. 2003, pp.181-186

[5] Bell, N. ; Xinghong Fang, Xinghong Fang ; Hughes, R. ; Kendall, G. ; O'Reilly, E. ; Shenghui Qiu, Shenghui Qiu; “Ghost direction detection and other innovations for Ms. Pac-Man”; International Conference Computational Intelligence and Games, Aug. 2010, pp.465-472

[6] Nicole Dobrowolski; “The Applicability of Uninformed and Informed Searches to Maze Traversal”; Dept. of Computer Science, University of Rochester

[7] Xiang Liu ; Daoxiong Gong ; “A Comparative Study of A-star Algorithms for Search and rescue in Perfect Maze”; International Conference on Electric Information and Control Engineering (ICEICE), 2011; pp: 24 - 27.

[8] Zhang, Ya Juan ; Zhang, Zi Li ; Deng, Yong; “An Improved Maze Solving Algorithm Based on An Amoeboid”; 2011 Chinese Control and Decision Conference, May 2011, pp.1440-1443

[9] Russel, Stuart and Norvig, Peter; “Artificial Intelligence A Modern Approach”; 2010; 3rd Edition, Pearson Education, New Jersey.

[10] “The Pac-Man Projects”, 2010; Department of Computer Science, University of California, Berkeley.

[11] http://www.freepacman.org/

[12] http://en.wikipedia.org/wiki/Maze_solving_algorithm

[13] http://en.wikipedia.org/wiki/Greedy_algorithm

[14] http://en.wikipedia.org/wiki/Uniform-cost_search

[15] http://en.wikipedia.org/wiki/flood_fill