MCO1 : Gold Miner

A major course output
for the course on
Introduction to Intelligent Systems
(CSINTSY)

Submitted by:

Capunitan, Jonaviene De Guzman

Tolentino, John Enrico Grijaldo

Villarica, Matthew James Del Pilar

Dr. Merlin Teodosia Suarez
Teacher

Dec 1, 2021

Link to Google Drive :
https://drive.google.com/drive/folders/1VvymIGSDP3W_-eRFDN_UbNqeYiylYDzq

I. Introduction

The goal of the GOLD MINER agent is to find a path to reach the pot of gold unit on a grid of N x N squares. The GOLD MINER must be a rational agent, avoiding any pits and preferably reaching the pot of gold while minimizing the path cost. The initial state is when the GOLD MINER starts at position (1,1) in the upper left portion of the mining area. For each run of the program, the initial front of the GOLD MINER and the locations of the pot of gold, pits and beacons are randomized. Additionally, the GOLD MINER agent doesn't have prior knowledge about the grid before finding a path.

II. AI Features

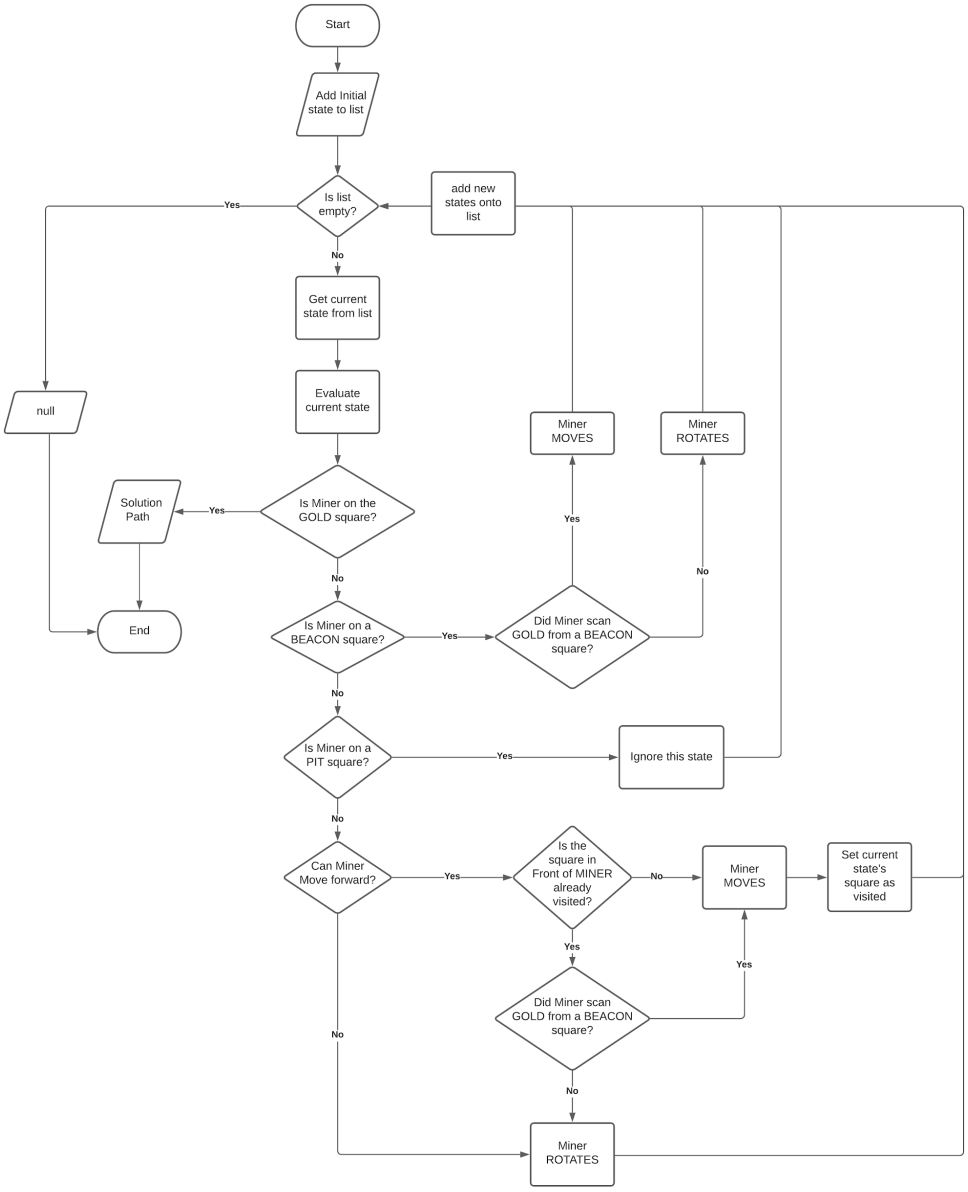

Figure 1. Agent Flowchart for Level R

Implemented Rules

The following actions are the Miner's possible moves for finding the Pot of Gold.

- Moving - The Miner displaces itself 1 tile from its current front given that the index is within 0 and N-1.

- Rotating - Using a HashMap, the Miner's possible fronts (value) are bound to integers (keys) divisible by three that are between 3 and 12. The Miner rotates by

adding three to the Miner's current front, a separate integer variable that is used to determine the Miner's current front by retrieving it from the HashMap.

- Scanning - Acts as the sensor for Miner. Scanning works by continuously checking the row or column of the Miner's position for objects, returning the object if one is detected, returning EMPTY otherwise.

For Miner's Decision Making in R-level search, the following rules are applied:

- The Miner cannot return to a square that is already visited.
- If the Miner lands on a PIT square, then it is considered a leaf node, and the agent will proceed on evaluating the next node on the list.
- If the Miner lands on a BEACON square, then it knows that the GOLD is reachable within a horizontal or vertical direction. Hence, it will continue to rotate and scan until it sees GOLD square, and proceed to move towards it until it is reached.
- If the Miner lands on the GOLD square, then the algorithm will return the solution node, which includes the current position and front of the Miner, as well as the path it took to reach the goal.

Algorithms

Two levels of rational behavior are implemented for the Miner agent as it traverses the board. Assuming that there is always a valid path to the Pot of Gold, the search algorithms applied are as follows:

- *Level R (Random)*
  Uninformed/Blind Search: Breadth-First Search Algorithm

  This algorithm is utilized for the randomized rational behavior of the Miner. Breadth-First Search utilizes the queue data structure (FIFO). The algorithm traverses the tree node by expanding a parent node then enqueueing its children nodes. It will then evaluate all nodes in the current depth level before moving to the nodes of the next level. In the context of the game, the algorithm operates by initially taking the first position of the Miner agent and explores each subsequent square until it reaches the GOLD square.

  This search algorithm will not utilize Miner's scan as much, since the Miner agent will randomly explore each square as opposed to finding the shortest path.

Assuming that there is at least one valid path, BFS will always return a solution path. However, it will not always return the most optimal path, but rather the first path it finds.

The algorithm will start by taking the initial position of Miner agent and expand its children nodes. The children node/s can be either of the two actions: ROTATE and MOVE. If Miner can move forward, then a MOVE node is enqueued. A ROTATE Node is also enqueued.

If the Miner lands on a PIT square, then it is considered a leaf node, and the algorithm will evaluate the next node on the fringe.

If the Miner lands on a BEACON square, then it will remove all the queues from the fringe and will continually create a ROTATE node until the Miner's scan returns GOLD. The Miner will proceed to MOVE until it reaches the GOLD square.

If the Miner lands on the GOLD square, then the algorithm will return the solution node, which includes the current position and front of the Miner, as well as the path it took to reach the goal.

Every move and rotate action will create a new state and will be added to the list of states to be explored.

- *Level S (Smart)*
  Informed/Smart Search : A* Search Algorithm

This algorithm is utilized for the smart rational behavior of the Miner. Similarly to the Breadth-First Search, it uses the Queue data structure although the BFS algorithm utilizes linked lists whereas the A*S utilizes the priority queue.

The A* Search Algorithm finds the path to the goal node with the least amount of a given cost such as distance, time, etc. In the context of the Gold Miner, it aims to find the most efficient path to the GOLD square. The algorithm uses the A* node instead, containing the g(n), the path cost from a node to another one and the h(n), the estimated path cost from the current position to the GOLD square. The sum of g(n) and h(n) is the basis of the ordering of nodes in the priority queue.

The algorithm traverses the tree by expanding a node, prioritizing the ones with the least amount of path cost, configuring its children's g(n) and h(n) costs then enqueueing the states where the action is valid and the miner's coordinates are on a tile that has not been visited yet. The priority queue is ordered in ascending order, which means the node at the head of the queue will always have the least amount of path cost and is more likely to lead to the goal node.

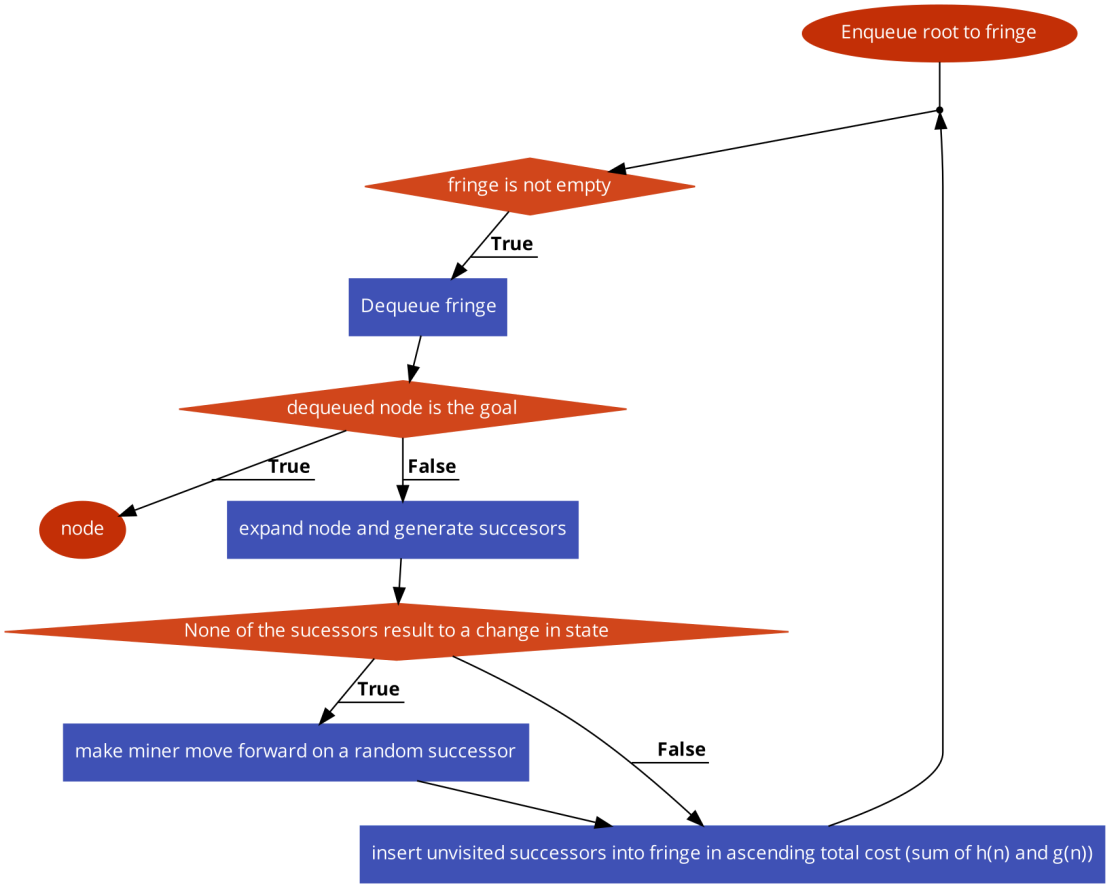

Figure 2. Agent Flowchart for Level S

The agent expands and enqueues the generated nodes into the fringe, with the least cost nodes being prioritized, until it reaches the goal node which is returned as a solution. In rare instances, the agent expands and generates a list of successors that did not result in a change of state. In the context of Level S, a change of state means the miner has not moved from the expanded node's original UNIT. This can happen because the miner has sensed pits in all 4 directions, or is cornered by 2 pits on the corner of the grid. In order for the agent to continue looking for a path, it has to move forward on some random successor in the list, which can be a risk.

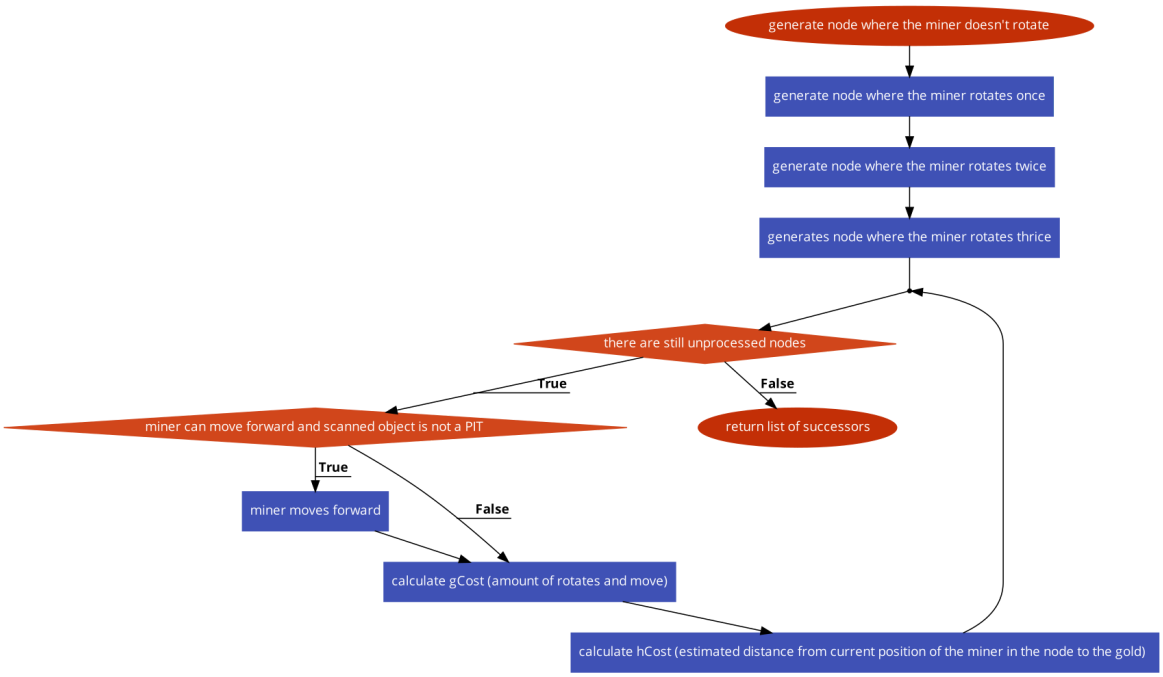

Figure 3. Flowchart for expansion of a node in smart search.

Expansion of a node will result into 4 children, but only unvisited nodes are enqueued into the fringe. The g(n) cost is the path cost, which is incremented by 1 for each rotation and move, and the h(n) cost is the heuristic.

Both search algorithms utilize a search tree data structure. In order to get the path, each node contains a String of the Miner's actions before reaching said node. 'R' stands for rotate, and 'M' stands for MOVE.

III. Results and Analysis

States, Configurations and Actions

1.) Random Search State Space (Breadth-first search Algorithm)

The specific states the agent operates on for random search are the different squares the miner can traverse on the grid. A state is the miner's current position on the grid, the type of UNIT the miner is on, and the direction the miner is facing. There are two actions that generate new states: ROTATE and MOVE. A new node is generated if the miner rotates once or moves once. In the case of an 8 x 8 grid, there are 64 squares on the grid the miner can be on, and the miner can be facing in 4 different directions on

each square, which results in a state space of 256. To streamline the navigation of the state space in the miner's search for the gold, the agent that used random rationality can minimize children generated and clear the fringe of unexplored nodes as soon as it trips on a beacon, which will be used to find the gold.

Figure 4. Possible actions miner can execute to change state.

In Figure n, the miner is at the starting position, and can not move forward because it's initial direction is set to NORTH, which is outside the bounds of the grid. It can only rotate, so a single ROTATE node child is generated. The miner more commonly generates two children, which is the ROTATE node and MOVE node.

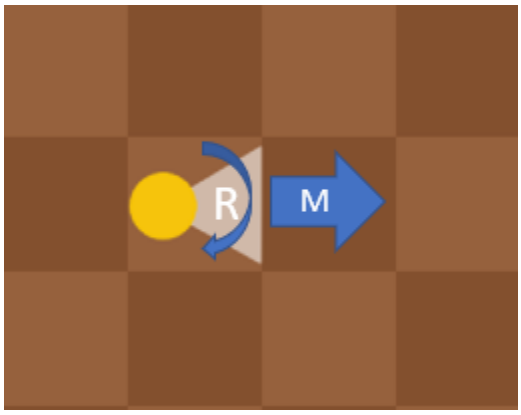

Figure 5. Possible actions miner can execute to change state.

This makes the branching factor at most 2 nodes. Although this is the usual case, there are special cases when the miner will only generate one child. For instance, when the miner trips on a beacon and has found the location of the pot of gold or another beacon, it will only generate one MOVE node from then on and clear the fringe of unnecessary nodes.

2.) Smart Search State Space (A* Search Algorithm)

In the case of A* search, an overall smaller state space is used which makes it fast, in addition to the heuristic used. A new node is generated only if the miner has

moved, rotating does not generate a new state. To transfer to another state, the miner has to move. For instance, on an 8 x 8 grid, the state space that the miner can navigate through a combination of rotating and moving has 64 states.

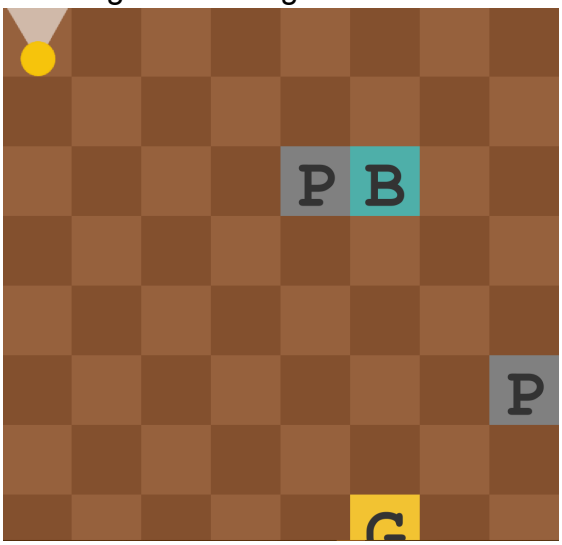

Figure 6. 8 x 8 grid with a state space of 64

In Figure 1, the miner is in the starting position randomized to be facing NORTH. There are two sets of actions that the miner can do to move to the next state: rotate, move (RM) and rotate, rotate, move (RRM). They have a path cost of 2 and 3 respectively.

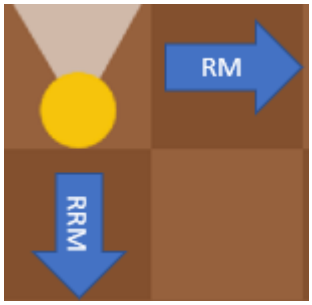

Figure 7. Actions miner can execute to move to the next state.

In this state, the miner can not move NORTH or WEST because that is outside the bounds of the grid. When the node of this particular state is expanded, only 2 nodes are generated. This state has a less amount of valid moves that would change the state compared to a state where the miner was not on the edge of the grid. This is why the branching factor is not a constant, and the max is 4.

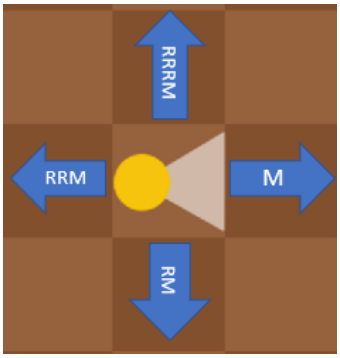

Figure 8. Actions miner can execute to move to the next state.

In this state, the miner can move NORTH, EAST, WEST and SOUTH and the actions the miner can do to reach these states are indicated in Figure 3. Although 4 children are generated after the expansion of this node, the UNITS that have already been visited are not enqueued anymore into the open list to avoid redundancy and to save time and space. So technically, only 3 nodes are generated if the resulting state of RRM has already been visited by the agent.

Detection of Goal State

For the bot with a random level of rationality that utilizes breadth-first search, it continuously expands every node that has not been visited before in a brute force manner, since nodes are explored by depth and path cost is not considered. When it trips on a beacon, it clears the fringe and finds the gold by scanning all directions and moves towards the gold. Since a beacon has already been detected, the agent no longer needs to expand any other nodes from the fringe aside from the node with the beacon and its successors. This prevents breadth-first search from consuming an impractical amount of memory and time. Once it detects that it is on the pot of gold, the node with that path is returned as a solution.

The smart bot that uses the A* search algorithm has a heuristic, which is the Manhattan distance from the current position of the miner to the pot of gold. Each node is assigned a cost, which is the sum of the Manhattan distance and the path cost to reach that node. The formula used for the Manhattan distance is |x2 - x1| + |y2 - y1|, where (x1, y1) and (x2, y2) are the positions of the miner and the gold on the grid. Any node enqueued to the open list is inserted into the list in ascending order, so nodes that have the least path cost and that are more likely to lead to the gold are prioritized and expanded first. Since the Manhattan distance is an admissible heuristic, it returns the optimal path, minimizing the path cost and distance travelled.

Limitations

Since both search algorithms use the search tree data structure, each node cannot point back to previously existing nodes. Therefore, multiple nodes consisting of

the same states can be generated for the tree. While the agent can find a solution path with the given range of 8-64, this could cause an overheap error for bigger board sizes.

To prevent the above scenario, each square is marked to be visited. Once the Miner visits a square on the board, it cannot revisit that square again. However, this prevents Miner from back tracking. If the Miner finds itself stuck in a dead-end and there are no other nodes to be explored, it cannot go back to its previous position. This will cause the agent to rotate indefinitely, causing an overheap error.

The S level algorithm's response to the expansion of a node not leading to a list of successors that are unexplored states can also be a problem. If not a single successor has resulted to the location of the miner changing, it means the miner is in a rare instance where it has scanned a pit in all 4 directions, or it is cornered by 2 pits in the corner of a grid. Since the miner cannot detect how far exactly the pit is from its current location, it has to randomly choose a node where it executes the action MOVE. This can end up with the miner falling into a pit.

IV. Recommendations

Utilizing graphs instead of trees is recommended to reduce the size of the data structure for the search algorithms. This will also allow the agent to backtrack to previously created nodes, and set the visit flag into the nodes itself rather than the squares on the grid. This way, each state (front + position) can be explored by the agent, instead of only checking if the position is visited.

The aforementioned problem of the S-level agent falling into a pit can be solved through backtracking. The agent only moves forward if it has scanned anything but a pit in that specific direction. If the miner finds himself in a state where it has scanned 4 pits in all 4 directions, the agent can backtrack to a state where the agent is not in that position anymore, which guarantees that the agent will not fall into a pit.

References

Peachpit. (2003, 12 September). Game Character Path Finding in Java. Retrieved From: https://www.peachpit.com/articles/article.aspx?p=101142&seqNum=2

V. Appendix A: Contribution of Members

Name	Detailed Contributions
Capunitan, Jonaviene De Guzman	Coding of classes and search algorithms, GUI, technical report

Tolentino, John Enrico Grijaldo	Coding of classes and search algorithms, UML diagram, technical report
Villarica, Matthew James Del Pilar	Coding of classes and search algorithms, technical report