Presented to the **Software Technology Department**

De La Salle University-Manila

Term 1, A.Y. 2020 - 2021

In partial fulfillment

of the course

In **CSINTSY S14**

“Into the Unknown: An Implementation of the Breadth-First Search Algorithm”

Submitted by:

Espiritu, Paolo Edni Andryn V.

Perez, James Andrew F.

Submitted to:

Joanna Pauline Rivera

December 10, 2020

I. Introduction

The goal of the Gold Miner program is for the agent to find its way to the gold tile, which contains the pot of gold, with the shortest amount of moves. The shorter the path that the agent traveled, the more rational it is.

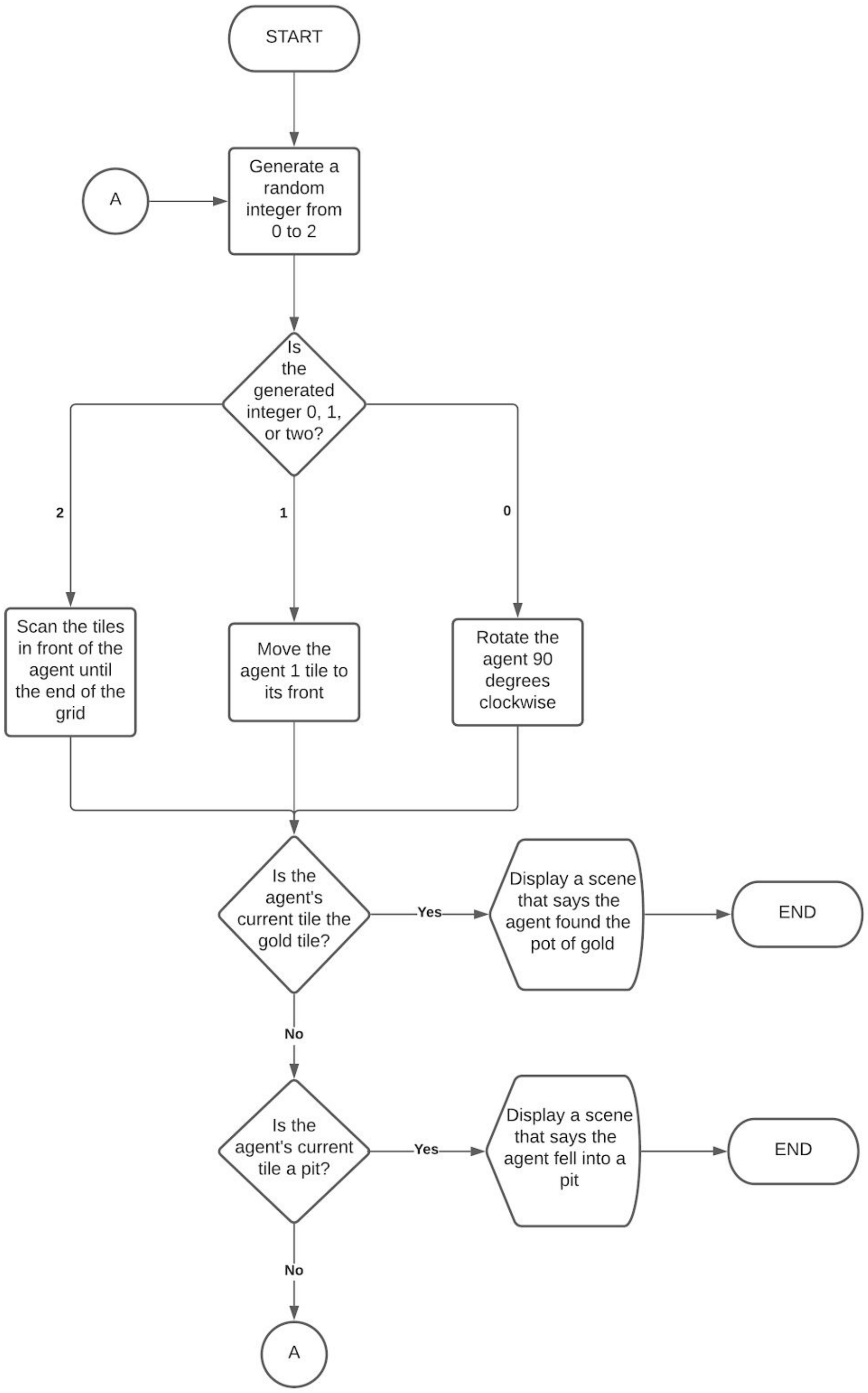
However, there are special tiles that the agent must consider when moving along the mining area. These tiles are pit, beacon, and gold. Whenever the agent belongs to the same coordinate as one of the special tiles, an event occurs. For pit, the game ends, and the agent fails to meet its objective. For the beacon, there is no active event. The beacon is simply an object that has a straight path to the gold without anything in between. This signifies that from the beacon's position, the agent can go directly to the golden tile without falling into a pit. Finally, when the agent moves into the pot of gold or the gold tile, the game ends, the agent meets its objective, and it should return a "search successful" message or something similar.

The agent has 3 main functions which are front, scan, and rotate. The front indicates where the scan and move are performed. The miner moves one tile with respect to its front; however, moving outside the grid should not be possible. The scan is used for the agent to view what is in its front all the way to the edge of the grid. It should return only one of the following special tiles: p for pit, b for beacon, and g for gold. This would depend on what is nearest to the agent. If the agent scanned no notable tiles, then the scan should return a "NULL" value. Lastly, the agent may also perform rotation to alter the direction of its front. This results in a 90 degrees clockwise rotation whenever it is performed.

The main problem to be solved here is what is the shortest path the agent can take to the gold tile. This problem is broken down into two parts. Firstly, finding the gold tile. In the program, the agent does not know where the goal is. Furthermore, the agent also doesn't know any information about the presence of pits or beacons, nor their locations. Finding the correct path involves the agent avoiding pits and using beacons as hints of the gold tile's position. There may also be multiple paths that can be taken. That is the second part of the problem. From the valid paths, the agent must select the shortest/most efficient path. This is the path that requires the least amount of actions to reach the goal. This efficiency is what separates the smart intelligence algorithm from the random intelligence algorithm. This along with the impossibility of the smart agent falling into a pit.

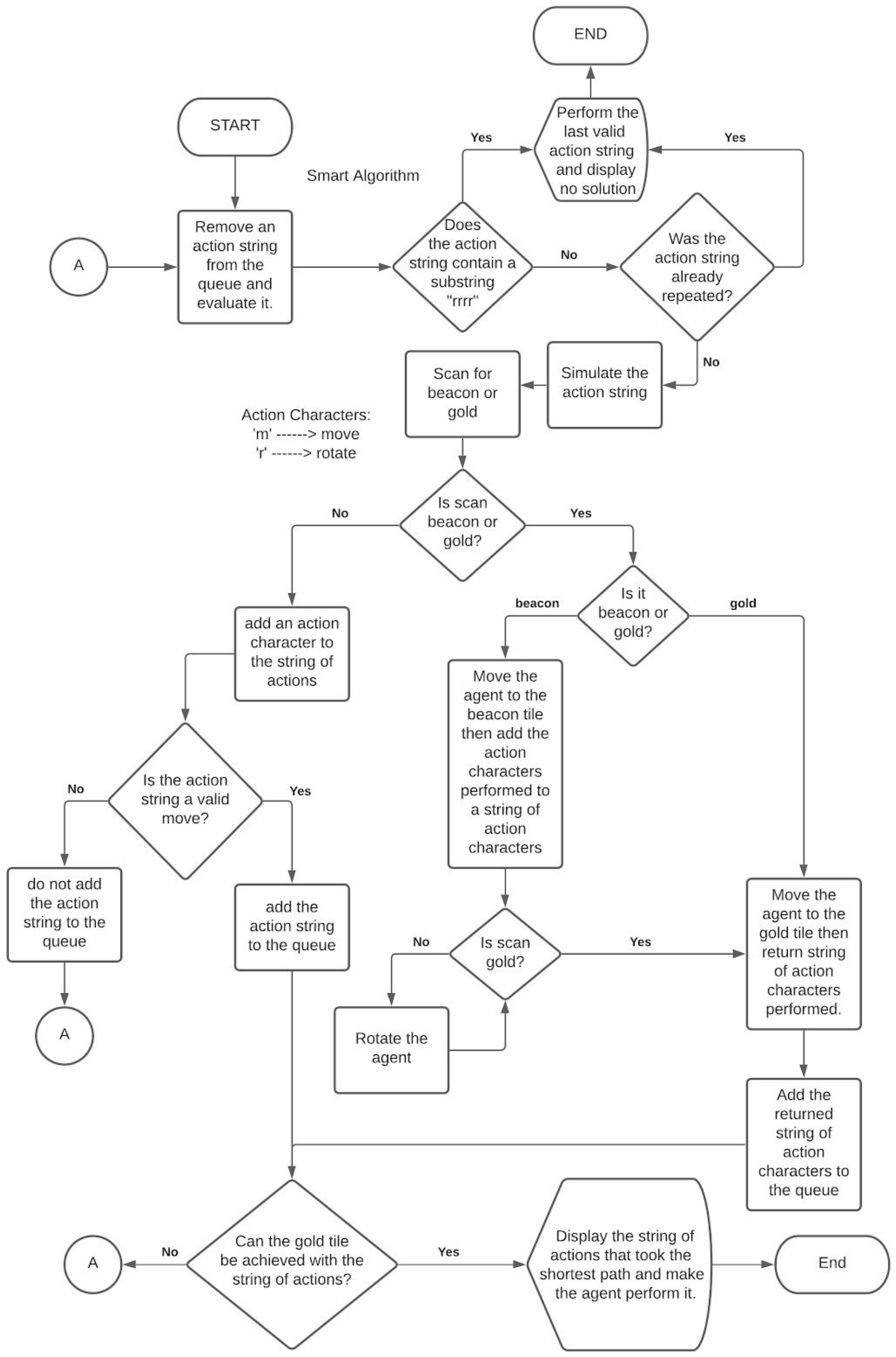
II. AI Features

Random:


START
A
Generate a random integer from 0 to 2
Is the generated integer 0, 1, or two?
2
1
0
Scan the tiles in front of the agent until the end of the grid
Move the agent 1 tile to its front
Rotate the agent 90 degrees clockwise
Is the agent's current tile the gold tile?
Yes
Display a scene that says the agent found the pot of gold
END
No
Is the agent's current tile a pit?
Yes
Display a scene that says the agent fell into a pit
END
No
A

For the random algorithm, we kept generating a random integer from 0 to 2 to determine which method will the bot use. The integer 0 tells the bot to rotate 90 degrees clockwise, 1 tells the bot to move one tile with respect to its front, and 2 tells the bot to scan the tiles in front of it until the end of the grid. The algorithm will end when the agent accidentally steps on the pit or gold tile. If the agent ends up on a gold tile, the agent meets its objective and the GUI displays that it found the pot of gold. On the other hand, if the agent moves into a pit, the agent fails to achieve its goal and the GUI displays that it fell into a pit.

Smart:



END
START
Smart Algorithm
Yes
Perform the last valid action string and display no solution
Yes
A
Remove an action string from the queue and evaluate it.
Does the action string contain a substring "rrrr"
No
Was the action string already repeated?
No
Scan for beacon or gold
Simulate the action string
Action Characters:
'm' ------> move
'r' ------> rotate
No
Is scan beacon or gold?
Yes
beacon
Is it beacon or gold?
gold
add an action character to the string of actions
Move the agent to the beacon tile then add the action characters performed to a string of action characters
No
Is the action string a valid move?
Yes
do not add the action string to the queue
add the action string to the queue
No
Is scan gold?
Yes
Move the agent to the gold tile then return string of action characters performed.
A
Rotate the agent
Add the returned string of action characters to the queue
A
No
Can the gold tile be achieved with the string of actions?
Yes
Display the string of actions that took the shortest path and make the agent perform it.
End

For our smart algorithm, we made use of the Breadth-First Search (BFS) algorithm. We used this algorithm because we wanted our agent to learn from its past movements and eventually reach the goal state after generating and simulating the action strings. To do this, we made a queue of strings which stores all the action strings that the bot can perform without falling into a pit or going out of bounds. In our program, a string of actions means a combination of ‘m’ (move) and ‘r’ (rotate) characters To know which action string should be stored in the queue, it is first evaluated if it is valid (meaning after the agent performs the action string, it should not be out of bounds or standing on a pit). The action string should also be unique, does not contain a substring “rrrr” (rotating four times), and when the sequence of action characters is performed, the agent must not go back to the visited tiles because doing so does not make sense when trying to find the most efficient path. Until the goal state is not met, a new action string is created by adding an action character to the current string of actions.

We also tweaked our BFS algorithm so that we can make use of the scan method of the miner. Our scan method will retrace each valid action string then perform a scan. If the scanned tile in front of the miner is ‘b’ (beacon), or ‘g’ (gold), the agent will move until it is in the same tile as the beacon/gold tile. Then, a number of ‘m’ action characters based on the number of movements made to reach the beacon/gold tile will be added to the action string that is currently being evaluated. If the beacon is scanned first, the agent will go to the beacon tile, then add the ‘m’ action characters. The agent will then perform a scan method, and if it was not able to scan a gold tile, the agent will rotate until it successfully scans a gold tile. This will add a number of ‘r’ action characters based on the number of rotations made to scan a gold tile to the action string. Finally, the agent will go to the gold tile, then add a number of ‘m’ action characters based on the number of movements made to get to the gold tile to the action string. The action string is returned and it will be added to the queue of action strings which will be the next one to be evaluated. In short, the scan method makes the algorithm more efficient by going directly to the goal when spotting a beacon, or the gold tile. This makes our BFS algorithm into a Best-First Search algorithm because the agent chooses the best path towards the goal state.

The agent will perform the action strings in the queue until it executes an action string that will lead him to the gold tile which is the goal of the program. The backend of the program will return the action string that reached the goal state and our GUI will demonstrate how our agent reached the goal state in the most efficient way possible.

If there is no path to the gold tile, the agent performs what the last few valid moves it could calculate before coming to the realization that there is no solution. It will then end the program.

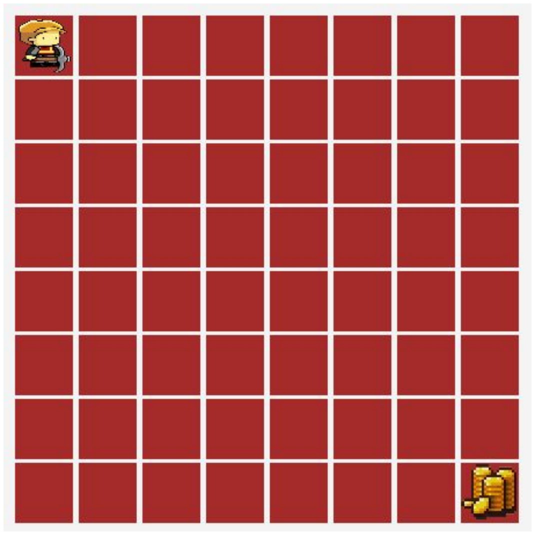
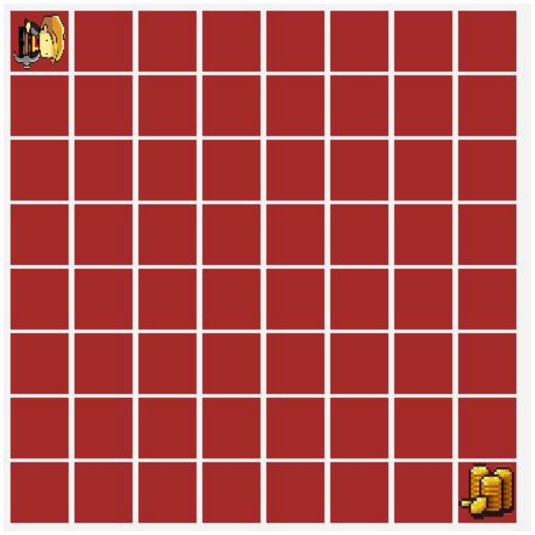
III. Results and Analysis

- Determine the specific states and configurations that the agent operates on.
- Determine the specific actions and the states (configurations) on which the actions are applicable (include illustrations for transition table).
- Specify how the goal state can be determined and detected by the agent.
- Discuss what situations the bot cannot handle. Explain why these situations cannot be handled by the bot. Point out which part of the Artificial Intelligence of the bot made it fail.
- You may include screenshots if need be.

The states of the agent are the different spaces/locations that it can occupy in the grid. For example, the initial state is the miner being in the top left corner, which is the (1, 1) position. The agent always starts at this position. Another state is the agent being in a neutral space. These are spaces that are empty. This, however, can be further classified into two more states: a state where the agent is in the middle of the grid, and a state where the agent is on the edge of the grid. One more state is the agent being in a space that has a beacon on it. This means that the goal state is almost reached. Speaking of which, the goal state is when the miner is on the gold tile. This signifies the success of the algorithm. The fail state, on the other hand, is when the agent steps on a tile that has a pit on it. This signifies the algorithm's failure and is a state that is not possible in the smart algorithm.

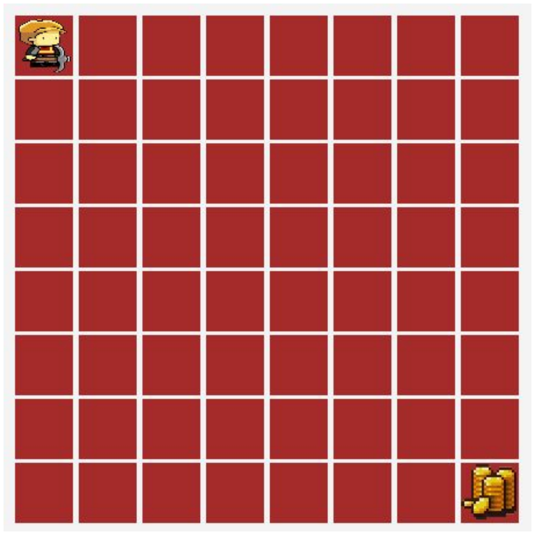
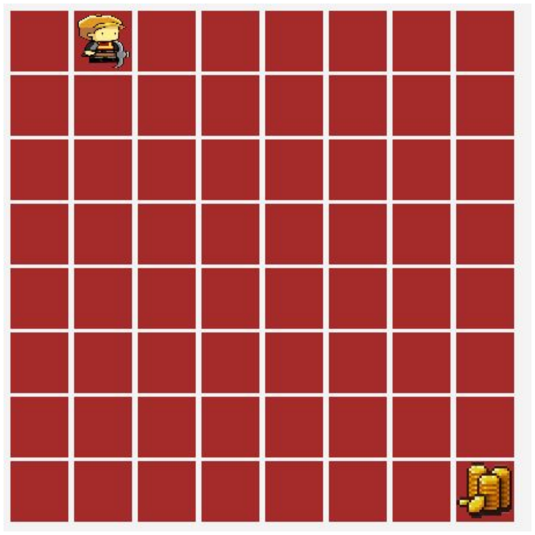
All states, except the fail and goal states, can have up to three applicable actions: rotate, move, and scan. When talking about these actions, it is automatically assumed that they are not performed in the fail, or goal state.

Rotate is where the miner shifts its orientation by 90 degrees clockwise. This action is applicable in any state. It changes the front of the agent according to the new direction it is facing towards.

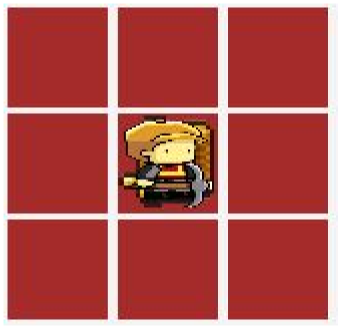



Scan, on the other hand, is an action that does not change what the agent looks like on the screen. It simply returns a value depending on what is in front of the miner. It returns the initial of the nearest special tile from the miner's front up to the edge of the grid. Having no special tile in front of the miner will return nothing or *null*. This action, like rotate, is applicable in any state.

Last is the move action. Move shifts the position of the agent from its current tile to the tile in front of it. There are states where this action is not applicable. For example, in both random and smart intelligence, the move action cannot be performed when the miner is on and facing the edge of the grid. Trying to do this action in that state will result in nothing but a console display of *"Miner moves out of bounds!"*. In the smart algorithm, this action is also not applicable in stats where there is a pit in front of the miner.

The goal state is simply when the agent reaches the gold tile. In other words, this is when the coordinates of the miner matches the coordinates of the gold. You can recognize this in the GUI by having the miner image in front of the gold image.



There are no situations that the bot cannot handle. If the scenario is something that has no solution (e.g. the gold tile being surrounded by pits, the miner being surrounded by pits, or a wall of pits between the miner and the gold tile), the program would simply display a “No Solution” window. There are situations, however, where the program takes so much time to determine the best path, or if there is no path at all. It still works, but it just takes too long. The GUI might also stop responding while the backend is calculating for the path. This usually happens when there are so many pits and the gold tile is far from the miner, or the grid size is large.

IV. Recommendations

The main weakness of the bot is the drastic amount of time it takes in determining the best path, or if there is no path to the gold tile. This causes scenarios involving large grid sizes, and many pits to take too long to calculate an answer. The GUI can crash or stop responding while the backend is in the process of computation. To address this, the Breadth-First Search (BFS) algorithm applied in the program can either be improved (made faster/more efficient), or replaced altogether with a better, more efficient algorithm. One that doesn’t take too long to compute more complex scenarios. The code can also be further optimized to make the program more efficient by implementing a function that will determine until when should the BFS create nodes (mix and matching action characters in a string). If this function is created, the agent should be able to compute a no solution scenario much faster than what we created.

V. References (Follow the APA format)

Tech With Tim. (2019). Python Path Finding Tutorial - Breadth First Search Algorithm [YouTube Video]. In *YouTube*. Retrieved on December 10, 2020 from https://www.youtube.com/watch?v=hettiSrJjM4&ab_channel=TechWithTim

VI. Appendix A: Contribution of Members

Name	Contribution
ESPIRITU, Paolo Edni Andryn V.	• Developed the backend and GUI. • Designed the format of the GUI. • Researched algorithms that can be used for the miner. • Debugged the code. Tested the program. • Technical writeup.
PEREZ, James Andrew F.	• Researched the controller and its interaction with the GUI. • Developed the backend and GUI. • Designed the graphic art of the GUI. • Debugged the code. Tested the program. • Technical writeup.