The algorithm is divided into two parts, finding the gold and returning home. When finding the gold, the robot senses information from its environment and thinks about its next logical move to make in order to find the gold. The robot receives no new information when navigating home.

Finding the Gold

In order to get the highest chance of finding the gold, the robot must always move into safe tiles and discover enough of the world that a return path can be found. By avoiding any potentially dangerous tiles the robot is guaranteed to not fall into a pit or walk into the wumpus.

The world is represented as a 4x4 array

(0,3)	(1,3)	(2,3)	(3,3)
(0,2)	(1,2)	(2,2)	(3,2)
(0,1)	(1,1)	(2,1)	(3,1)
(0,0)	(1,0)	(2,0)	(3,0)

The robot will always start in *(0,0)* and is guaranteed to be safe and have no senses coming from it. Starting from *(0,0)* we can map all the potential moves the robot can make from the starting position to the upper right of the world *(3,3)*. All potential moves/paths can be seen in figure 2.

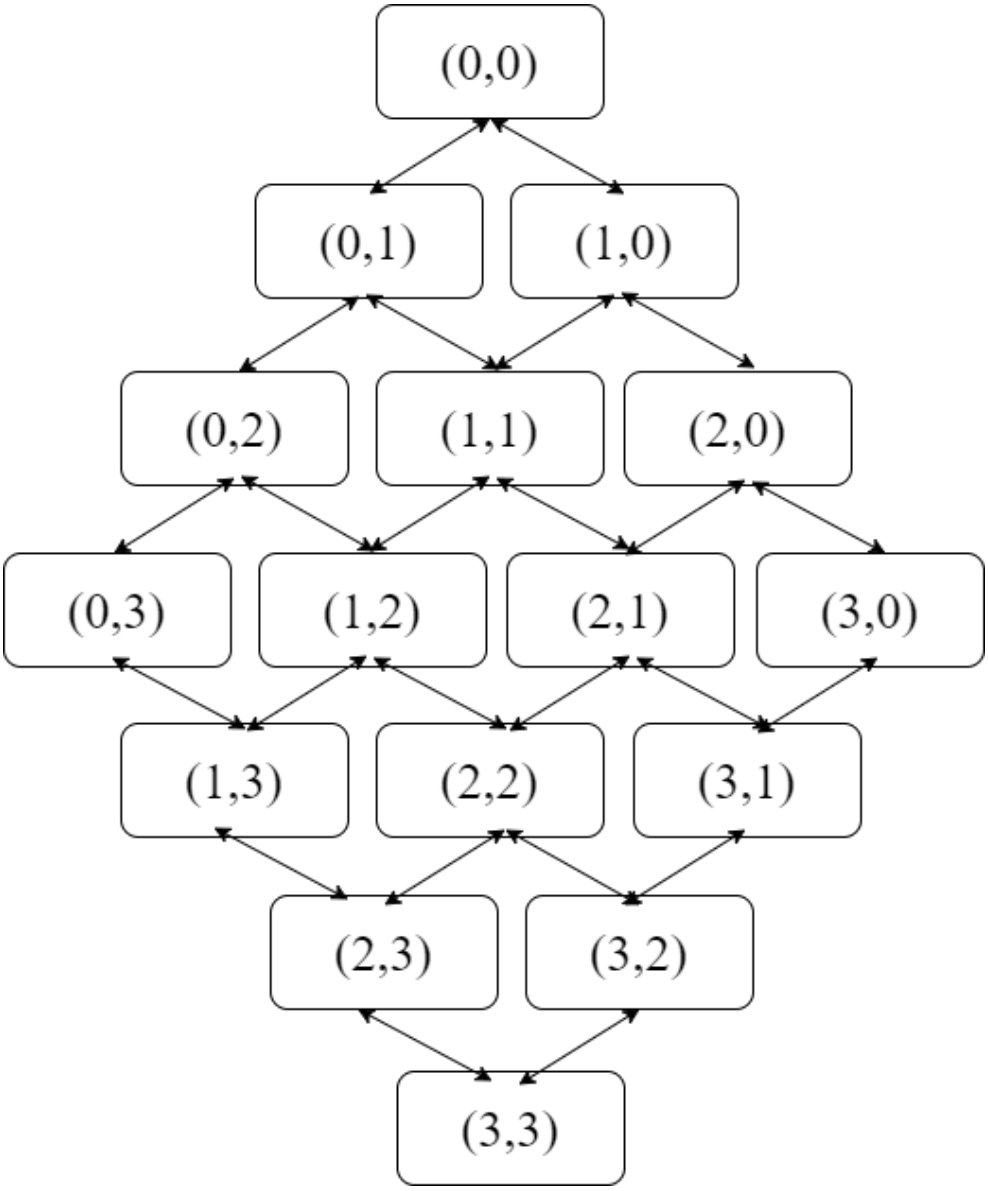
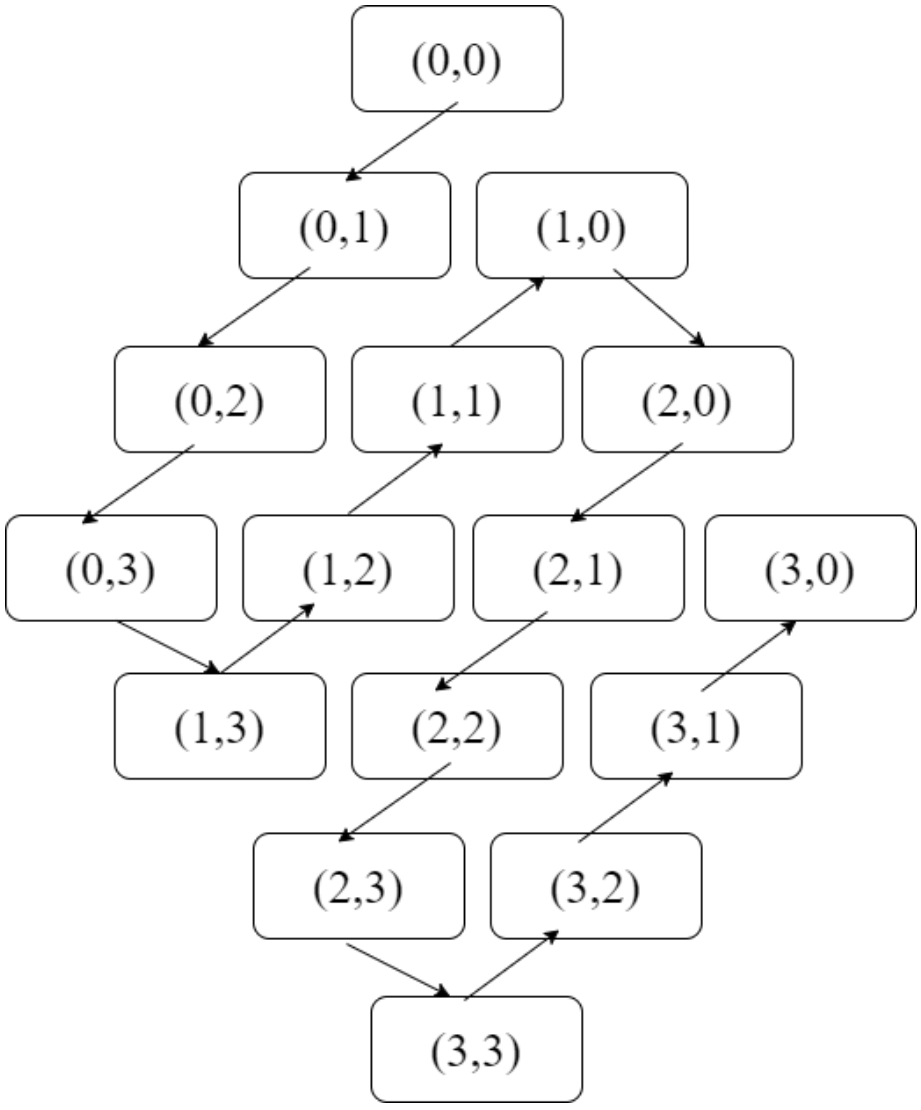


Figure 2

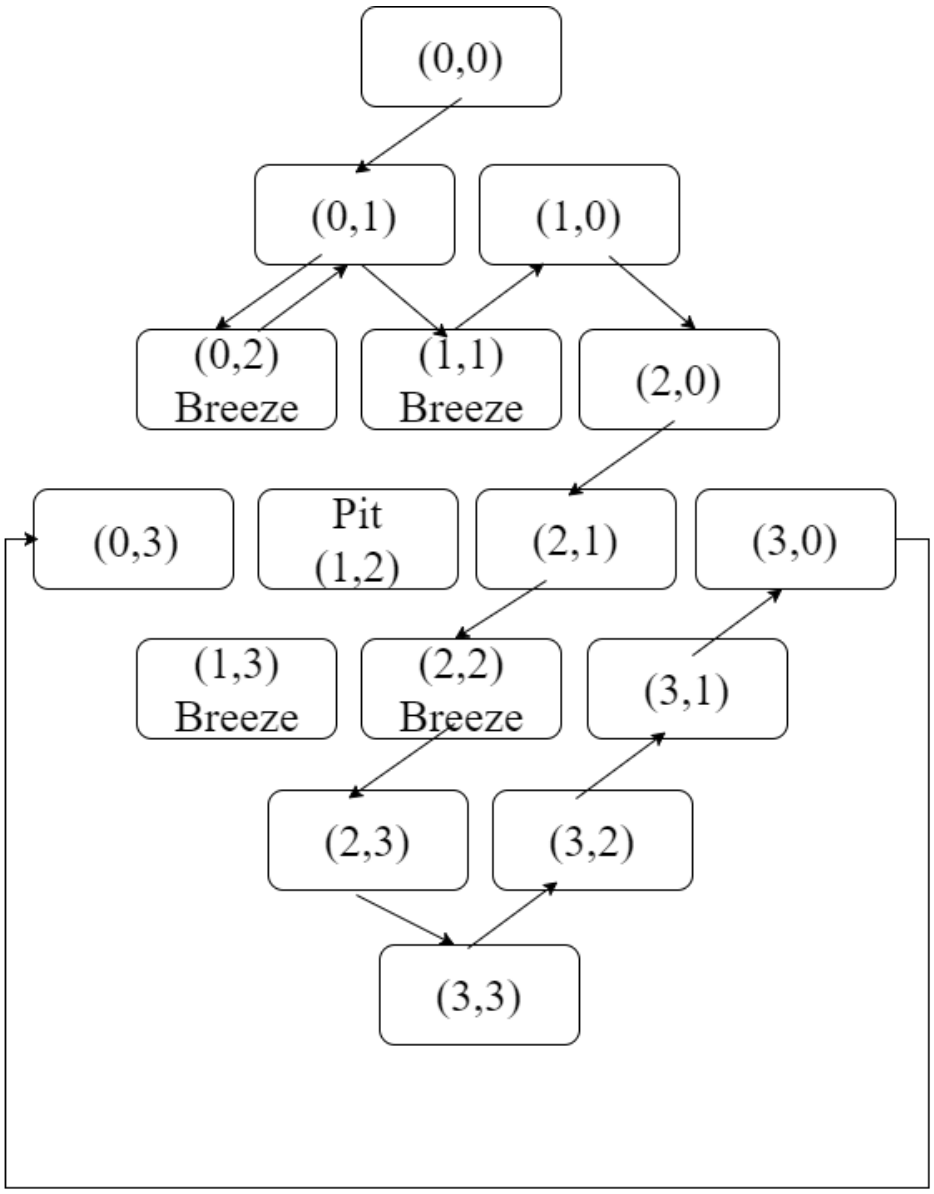
Notice the similarity to a ternary tree, the max number of children a node can have is 3. However, in order to reduce the number of potential paths the robot can take the robot may move the node it came from (following the path from the child node to its parent) and children nodes can also share parent nodes to reduce the number of paths.

The robot should only search immediate nodes that are safe and not already discovered. The desired tree traversal is thus travelling down all left nodes and then going up the right parent when reaching a node with no left child. Once we hit a node with no right parent, the algorithm will start again traveling once to the right child then following the left nodes all the way down. This modified breadth first search (BFS) looks like:



An advantage of this representation is that there will always be a path present even if a node is marked unsafe. Once unsafe, the BFS will investigate nearby nodes that

will confirm whether a node is a threat or not. A disadvantage is that if the robot does not find the goal after avoiding the obstacle, it will need to take a very long path to any undiscovered nodes.

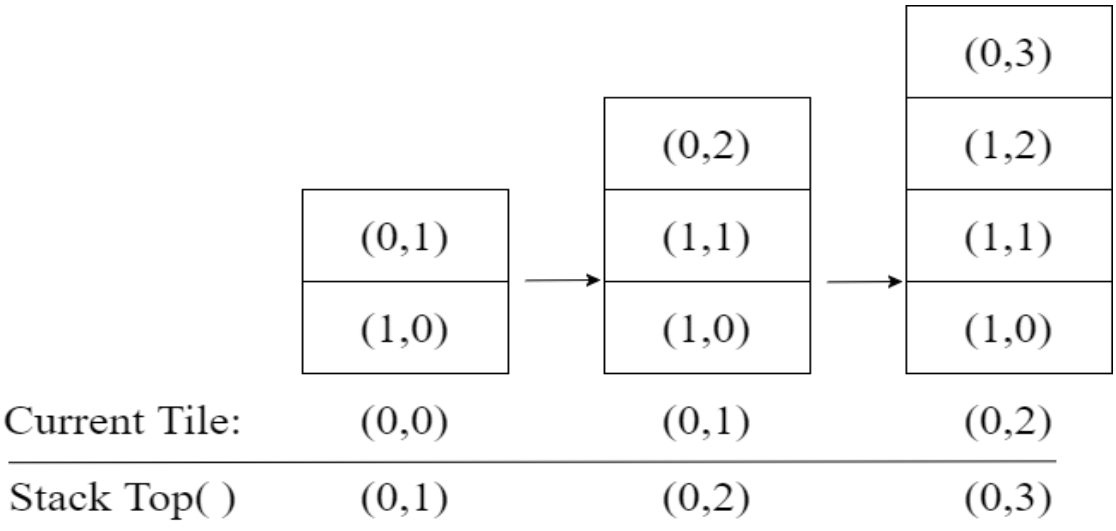


This traversal works exceptionally well when entities on the board are spaced out. If entities are clumped together, this will cause the robot to search for more information before making a decision, taking significantly longer to complete its goal.

Algorithm Optimization

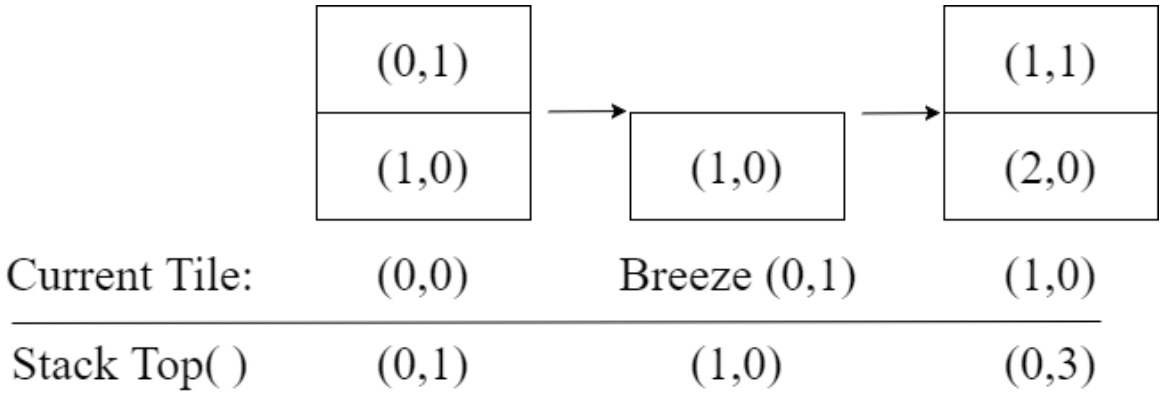
It might be tempting to immediately represent the board as a ternary tree, marking nodes that were visited and moving between children and parent nodes using pointer arithmetic, but implementing such a data structure in C is a complex task. After testing tree developing in C, it was found that separating the world representation from processing decisions worked the best. This means that the world was represented as a 4x4 array within the memory of the robot and processing information was dependent only on the BFS. How do we perform a BFS without a tree? Since the only thing that matters are the immediate moves that may be performed and the order of which new moves are found, there is no need to represent every single path and move from the current position; we only need to look at the immediate ones. This can be done be implementing a stack of potential moves.

A potential move is defined as a tile that is safe to move onto but has not been explored yet. Traveling to an already explored tile to get to the next move is valid also.



(A stack is an array where elements can only be added and removed from the top)

Now consider the stack method when faced with a sense:



A breeze has been detected on (0,1) therefore no new moves can be made from that tile and nothing has been added to the stack. The only valid move now is to go to tile (1,0). At (1,0) the robot senses no new data and therefore can mark (1,1) as safe and will add it to the stack.

Notice that this method gives the same moves as the Ternary tree does but at a fraction of the processing and memory cost. A stack is also notably easier to program as it is just an array with specific restrictions.

Algorithm Analysis:

Processing information is guaranteed constant. Accessing the next move and adding new moves to the stack are both ***O(1)***.

The worst case/average memory usage is currently the representation of the array along with the memory usage of the stack which is ***O(n + n) = O(n)***, where ***n*** is the number of tiles

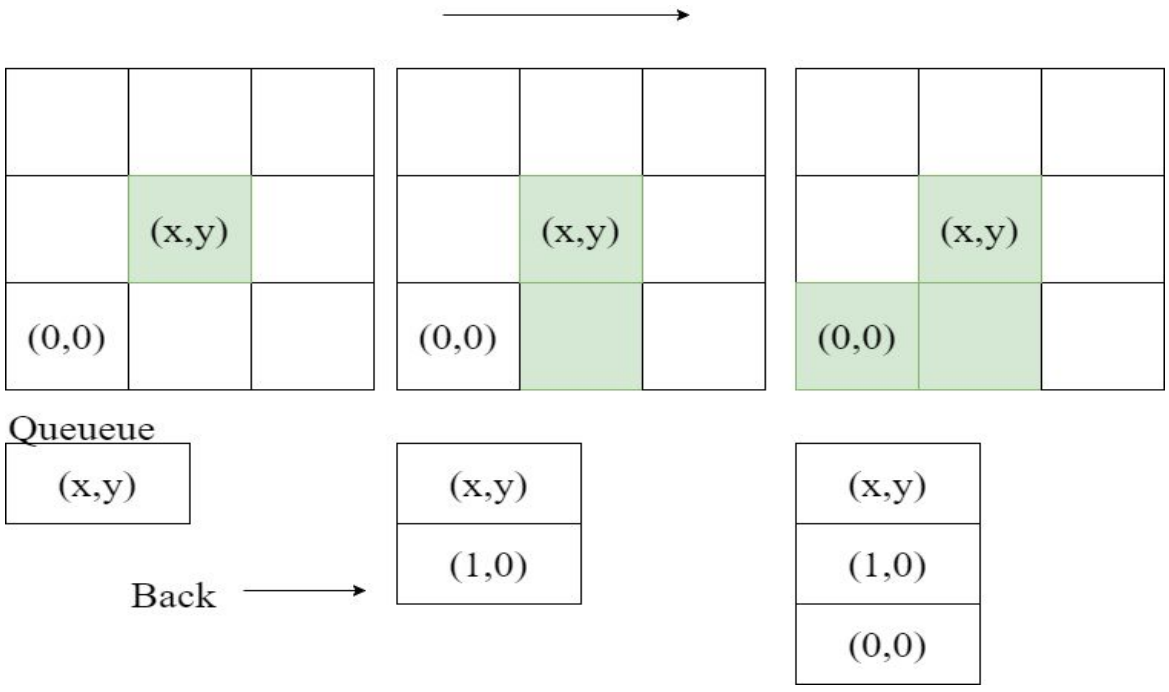
Returning Home

Part two of the challenge occurs when the robot has collected the gold and needs to find its way back to (0,0).

The robot does this by first creating a path to (0,0) from its memory of safe tiles and then following that path back. In order to create a path, a hashtable equal to the size of the width times the height is created. All spots are marked with -1 except with the final location which is marked with -2. The final position can be marked with the equation: (This will always be index 0 for returning to (0,0), but this equation is used to move in between moves for the first part)

$$\text{path}[\, width \bullet X_{fnal} + Y_{final} \,] = -2;$$

The algorithm then uses a queue to find a path back. Going from the current position, tiles are added to the back of the queue. The robot then determines the general direction of the target position - if the target is either above or under it and then if it to the left or right of it. This is down each cycle until the queue includes the target position.



The general direction of the target position is found by using the index of the path.

$$n = (width \bullet y_{current}) + x_{current}$$

$$X_{next} = \text{path}[\, n \,] \,\%\, width$$

$$Y_{next} = \text{path}[n]/\text{width}$$

The Hash Function (above)

If $X_{next} = X_{current} + 1$ then the target is to the right

If $X_{next} = X_{current} - 1$ then the target is to the left

If $Y_{next} = Y_{current} + 1$ then the target is above

If $Y_{next} = Y_{current} - 1$ then the target is below

Once the queue is calculated, the frontmost move is acted upon and the robot will end up at the target position. If the robot has the gold at this time, it will terminate, if the robot is just using the moveto algorithm in between tiles to search the world it will then resume normal runtime in part 1.

Analysis:

The memory of the hash table will be a worst case of ***O(n)***. Entering and accessing elements will be ***O(1)*** or guaranteed constant since the hash function itself is constant. Memory usage and processing power for the queue is ***O(n), O(1)***, respectively.

Overall the processing time of the entire algorithm is guaranteed constant- *O(*1)
The memory usage is *O(n),* where n is the number of tiles. (A stack will never exist alongside the hash table or queue which will maintain the linear memory usage)