DMU Final Project Report: Creating an Algorithm that Plays a Simplified Version of Temple Run

Yang Lee*, Colton Ord†, Seif Said‡

I. Introduction

Decision making algorithms have become an integral and powerful tool for producing autonomous capabilities in various systems and scenarios. It is currently a very active area of research. Decision making algorithms attempt to remove the need for human intervention and provide autonomous systems with the ability to handle highly dynamic scenarios and environments in order to complete their tasks or objectives. Thus, the project presented here attempts to assess the capability of a decision making algorithm in a procedurally generated map. For this project, a simplified version of the mobile game Temple Run is played as a Markov Decision Process (MDP). The MDP solver being evaluated here is Monte Carlo Tree Search (MCTS). The objective of this game is to survive and reach the end of the maps that are generated randomly. In those maps are obstacles and boundaries that the agent must avoid to reach the objective. The agent has three actions that it can take, move right, move left, or move down. Furthermore, at all the times, the agent is constrained to a choice of transitioning between 2 discrete blocks.

A. Levels of Success

- **Level 1:** The first level is the minimum level of success for this project. At this level, the requirement is to have MCTS solve a small constrained map. This level was successfully achieved.
- **Level 2:** The second level of success required having MCTS solve a procedurally generated map. To clarify, the maps were not infinite, but of varying sizes. This level was also successfully achieved in the project.
- **Level 3:** The third level which is considered the hardest would introduce moving enemies to the map as well new types of obstacles. The enemies would be constrained to a row in the map and would alternate between the 3 blocks in a specific row every time the agent transitions. Additionally, the new obstacles would introduce new actions that the agent would have to take. There would be two new types introduced. The first one can only be passed over by the jump action. The second would only be passed by performing the crouch and slide action. Unfortunately, this level was not achieved for the project due to time constraints.

II. Background and Related Work

The initial step for this project was to determine the algorithm of choice for the temple run MDP. Multiple options were examined that included both online and offline methods. However, only two were examined thoroughly, Deep Q Learning and MCTS. Reinforcement learning methods have proven to be quite capable of handling known popular video games such as video pinball, boxing, and breakout [1]. Therefore, DQN was tested and examined for this MDP. The constrained map described my level 1 was used to assess DQN's capability in finding a solution. Unfortunately, the group was unable to produce an implementation that was capable of solving the MDP map. It was also concluded not to be a proper choice for the MDP as DQN is an offline solver; This would require the algorithm to train over every new map that is generated as the Q values change. Thus, any Q value that the solver converges on for one map might not necessarily be equivalent in a new generated one. This proved to be an issue for using DQN. Therefore, DQN was abandoned for MCTS due to the limited time available.

On the other hand, MCTS is an online planner and balances between two strategies, exploitation and exploration. It does not require full knowledge of the environment to return an action it estimates to be the best. There are many variations that have been employed [2]. those variations have been tested on various games to assess its capability. Jacobsen et al, 2015, utilized a version of MCTS that used mixmax backups and partial expansion to play the popular

*Aerospace Engineering Graduate Student, CU Boulder SID:107054520
†Aerospace Engineering Graduate Student, CU Boulder SID:106104552
‡Aerospace Engineering Graduate Student, CU Boulder SID:106981440

platform game, Mario; the authors were able to achieve a near optimal behavior[3]. Schadd et al, 2008, developed a variant of MCTS called single-player MCTS, specifically, for single player games; The authors deployed the algorithm on a puzzle game known as SameGame which was able to produce a high performance [4]. Several other modifications have also been investigated and implemented. These studies show that MCTS maintains a strong versatility in its ability to be modified and used for different games [5]. However, after examining the MDP for this project, it was decided that the original MCTS with the the upper confidence bound would be the best variation due to the lower level of complexity this MDP presents in comparison with the video games utilized in previous studies.

A. Map Generation

For our project, the map generated needed to be random each time it was created and be able to be infinite. Since we were coding in Julia, and the given grid worlds that Julia has could not achieve our desired end-goal, we needed to create our own function to do this. The result was a function that took in three arguments and returned a matrix of the map and an array of which spots in the map were terminal and not. The idea was that this function could be called as many times as needed until the agent failed the map. This was possible because the function would always start and end at the same position on the map. To also aid in the stitching of different maps, the map always used a constant row length of 27 blocks. The first argument of the function was determining how far in total this portion of the map would be. The second argument of the function was to determine the maximum number of discrete blocks until a turn. The actual distance from each turn was randomly selected from 2 to that max number. The third argument was a Boolean value that would turn the last row in the map matrix into all non-terminal states. The Boolean value was added for level 1. The returned matrix map from the function was a matrix that contains the row and column for each block. The returned array was an array that indexes corresponded to the matrix map identifying if it was a terminal state or not by using either a one for non-terminal or a zero for a terminal state. The path that the agent could take stays at a constant row length of 3 blocks. An example map is shown in figure 1. In the figure, white is non-terminal while black is terminal. The figure 1 shows the two different kinds of turns implemented: single direction and junctions. Due to the limitation of the map being a constant row length, all turns cause the agent to continue moving in the same direction before doing the turn.

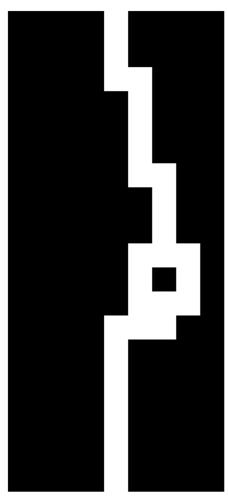


Fig. 1 Output of Map Generation Function

The function would start by creating the map matrix and terminal array. All discrete blocks within the map are first declared terminal. The code then creates the starting path for the agent which is 3 blocks by 3 blocks at the start of the map and in the middle of the row length. The code then goes into a while loop that goes row by row in the map matrix creating the path. The while loop will only stop once it has looped through each row in the map matrix. The first thing that is checked within the while loop is how close the current row is to the end of the map matrix. If it was within 25 blocks of the end, the code then created one final turn that causes the path to go to the starting column. Once it reaches it, the function creates a straight path until the end of the matrix. If the current row isn't close to the end of the map, the code then generates a straight path forward by a random number of blocks. At the end of the straight path is where the turn is generated at. The code picks randomly between creating a single direction turn or a junction. When creating a single direction turn, the code again picks randomly between whether or not it will be a left or right turn. The turn itself is created by extending the straight path's either most left block or right block by 3 blocks. This extension then happens for three rows. The turn is then done and created and the new leftmost position is saved so it can be used for the next turn. This concept is also used when creating junctions with the row extending in both directions rather than one. One key difference is that since the path will end in the same location as it started before the junction, the left-most block does not change and thus does not need to be updated. Due to coding limitations and that the map was generated before the agent is placed in it, junctions were required to always rejoin after their initial. The split then causes each path to be extended itself forward by 3 blocks. The two paths are then rejoined. Once the map is fully generated and complete, before it returns the map matrix and terminal array, it checks if the last row needs to be turned into all ones and does it.

III. Problem Formulation

This project formulates the problem as an MDP. Each grid on the map represents a state. The state-space consists of the entire procedurally generated map. The agent can take one of the three actions in the action space: left, right, and down. The problem explicitly defines the transition function, with a twenty percent probability of taking a random action. The transition is simple: the agent moves down when taking the down action, the agent moves left when taking the left action, and the agent moves right when taking the right action. If the agent hits a wall, the agent returns to its original state prior to hitting it. When the agent arrives at a valid state (i.e., not a wall), the agent receives a reward equal to the current row number plus one. When the agent runs into a wall, it receives a reward of negative ten. The agent also receives a reward equal to ten times the length of the map when it completes the map. Below is the MDP formulation of the randomly generated map.

$$S \in R^2 \tag{1}$$

$$A \in (right, left, down) \tag{2}$$

$$T = \begin{cases} 0.8 & s' = (x, y+1), a = right \\ 0.8 & s' = (x, y-1), a = left \\ 0.8 & s' = (x+1, y), a = down \\ 0.2 & s' = valid, a = any \\ 0 & s' = invalid, a = any \end{cases} \tag{3}$$

$$R = \begin{cases} 0 & s' = valid, (a = right, left) \\ x(s') + 1 & s' = valid, a = down \\ 10x(s') & s' = terminal, a = any \\ -10 & s' = invalid, a = any \end{cases} \tag{4}$$

In the above MDP formulation, x represents the row and y represents the column of the matrix to indicate the current state on the map. Furthermore, a valid state is a state that the agent is permitted to transition; an invalid state would be one that is outside of the bounds of the map or an obstacle location.

IV. Solution Approach

A. Heuristic Policy

The group implemented a simple heuristic policy as a benchmark test for the other algorithms. The heuristic policy works as follows: if going down yields a positive reward, then go down. If going down hits a wall (i.e., yields a negative reward), pick left or right based on the highest reward.

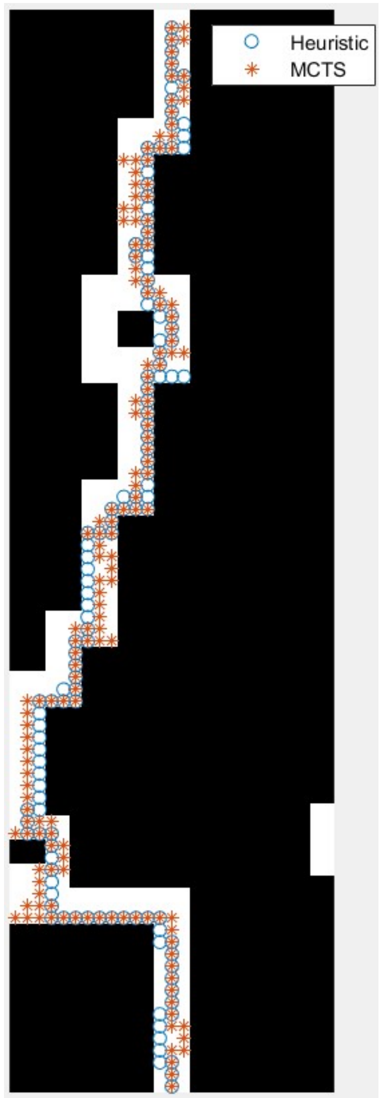
B. Monte Carlo Tree Search

After formulating the appropriate MDP, the code uses Monte Carlo Tree Search to determine the policy in each state. The solution implemented in Julia utilizes the MCTS package and the QuickPOMDP package developed by Zachary Sunburg et al. The original plan was to develop and implement a unique MCTS algorithm like one of the assignments. Unfortunately, the effort was unsuccessful. Consequently, the group modified the original formulation of the MDP code to make it compatible with the QuickPOMDPs package. Furthermore, the QuickMDP function enables the group to structure and use the MDP code with various MDP solvers. Once the MCTS solver is fully defined, the code creates a planner based upon the MCTS solver and the MDP formulation. At every step of the simulation, the code outputs an action using the solution calculated by the solver. The transition and reward functions then use this action to determine the next state and the associated reward of the agent. This loop runs continuously until the agent reaches a terminal state. In the case of the procedurally generated map, the agent never hits a terminal state. In order to demonstrate the function of the algorithm, the code ran a trial with a finite number of procedurally generated maps. The result from the trial proves that the algorithm works as expected.

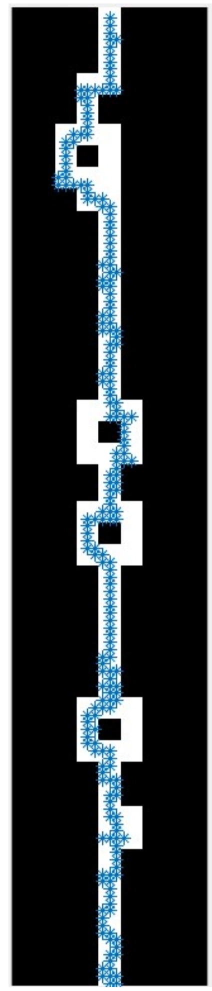
V. Results

The resulting plots show that the MCTS code implemented is fully capable of navigating the agent through the map. The performance of MCTS is comparable to the performance of the heuristic policy. Both are fully capable of navigating through the given map. Figure 2a shows the comparison between MCTS's trajectory and the trajectory when using the heuristic policy. It should be noted that the state transition is not deterministic: there is a 20 % chance of the agent taking a random action.

Figure 2b shows the agent navigating through a procedurally generated map. When the agent reaches the end of a map, another map is generated using the map generation function. The new map is then concatenated with the previous map, forming a longer map. The agent then uses the previous end state as the initial state, and continues to navigate through the new map. Figure 2b only shows an agent navigating through a map concatenated with three maps. If this process were to be repeated in an infinite while loop, the agent is fully capable of navigating through an infinitely long map. Due to time limitations, the code was only bench marked for 10 runs. MCTS had an average reward of 5582 while the heuristic policy had an average reward of 4963. MCTS had an average run time of 0.27 seconds and the heuristic policy had a value of 5.5 seconds. Both also achieved a 100% success rate.



(a) Comparison between MCTS and a Heuristic Policy



(b) Example of the Agent Navigating Through Procedurally Generated Maps

Fig. 2 Examples of the Agent Navigating Through the Maps

VI. Conclusion

The map generation code is a decent beginning to this problem, but it still has its own limitation. Further work needs to be done to add obstacles and missing parts of the map. DQN was investigated initially as a solver for this MDP. However, it proved to be problematic for this type of problem where the reward function changes with every new generated map. The Q values are not always the same for every map. Furthermore, in the scenario of an "infinite" map, DQN would fail as it would need to train over every possible state and action tuple to converge over a Q value for the map. Thus, it seems that an online planner is better suited for an infinite horizon MDP. The implementation of MCTS to this problem proves to be mostly successful. Although the agent sometimes gets stuck between states, this issue is mainly avoided by having a 20% chance of taking a random action. The reward function may be modified to penalize going back and forth between states in the future. Moreover, more obstacle types such as moving enemies may be implemented in the future. This project successfully achieved most of the targets set initially and effectively serves as a good starting point for further studies and explorations into the world of DMU.

VII. Contributions and Release

- Yang Lee: Convert code to work with QuickMDP; implement MCTS; implement heuristic policy; wrote the problem formulation, solution approach, and results sections
- Colton Ord: Tried to modify current GridWorld for our project, created algorithm and implemented the map generation code
- Seif Said: Investigated DQN, created the transition and reward function for the environment, wrote the introduction and background section as well as some parts of the conclusion.

The authors grant permission for this report to be posted publicly.

References

[1] Mnih, Volodymyr, et al. "Human-Level Control through Deep Reinforcement Learning." Nature, vol. 518, no. 7540, 2015, pp. 529–533., https://doi.org/10.1038/nature14236.

[2] Fu, Michael C. "Monte Carlo Tree Search: A Tutorial." 2018 Winter Simulation Conference (WSC), 2018, https://doi.org/10.1109/wsc.2018.8632344.

[3] Jacobsen, Emil Juul, et al. "Monte Mario." Proceedings of the 2014 Annual Conference on Genetic and Evolutionary Computation, 2014, https://doi.org/10.1145/2576768.2598392.

[4] Schadd, Maarten P., et al. "Single-Player Monte-Carlo Tree Search." Computers and Games, 2008, pp. 1–12., https://doi.org/10.1007/978-3-540-87608-3_1.

[5] Frydenberg, Frederik, et al. "Investigating Mcts Modifications in General Video Game Playing." 2015 IEEE Conference on Computational Intelligence and Games (CIG), 2015, https://doi.org/10.1109/cig.2015.7317937.

Appendix

A. Map Sample Examples

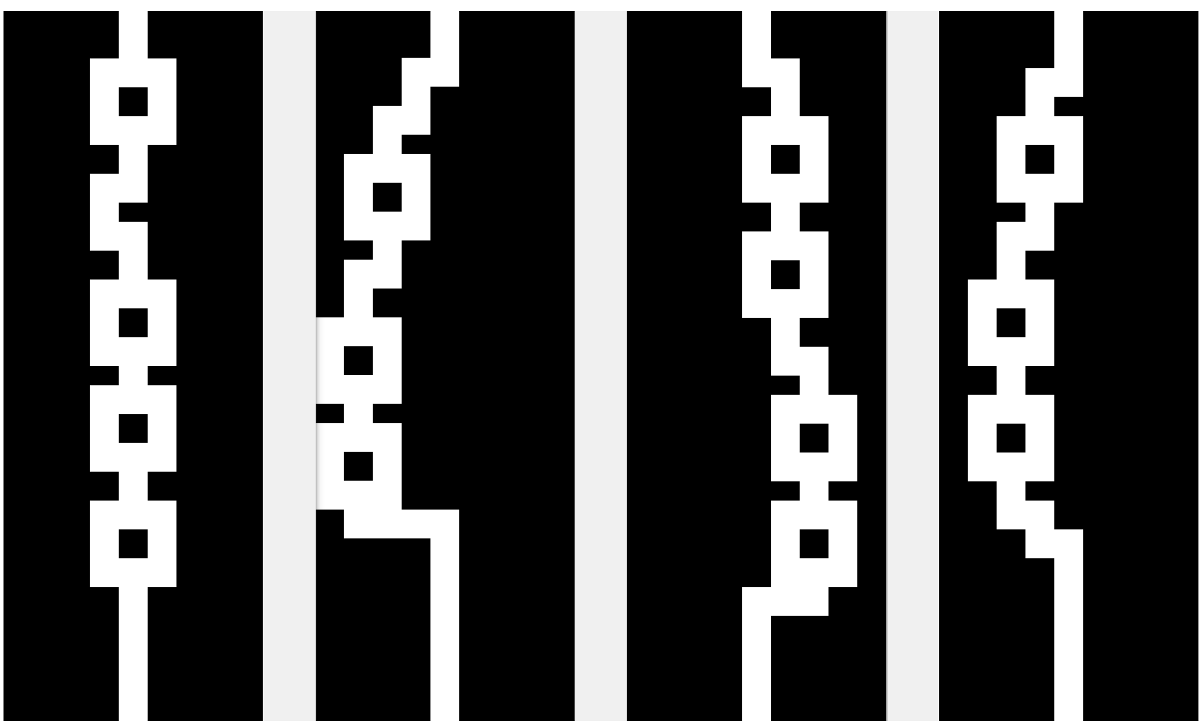


Fig. 3 Output of Map Generation Function with Max Map Distance of 75 and Distance between turns of 3

Code

1. Over All Simulation Code

```
using Random
using POMDPModels: SimpleGridWorld
using POMDPModelTools
using POMDPs: POMDPs, Solver, Policy, states, actions, isterminal, @gen, statetype, solve
using POMDPPolicies: FunctionPolicy
using LinearAlgebra: I
using CommonRLInterface: render, act!, observe, reset!, AbstractEnv, observations, ...
    terminated, clone
using SparseArrays
using QuickPOMDPs
using StaticArrays
using Statistics: mean
using Interact
using Plots
using Cairo
using Fontconfig
using MCTS: MCTSSolver, action

function createmap(n, maxfar,lastrow)
    # Overall, the first index in the matrix is moving forward while the second index is ...
        about moving left to right. Going left is negative while right is positive
    # Creating the actual matrix so that we have an actual grid to ensure is valid
    mat = [(i,j) for i=1:n for j=1:27]
    # This will tell if the square are valid, so if it is a one, it is valid. The idea is ...
        that the reward should increase by one each time it gets a valid square
    matvalid = zeros(length(mat))
    counter = 0#length(mat)
    ind = 0
    # Note it goes by 1 to 9 then goes to a new row, so to access a part of the matrix, so ...
        to access any column (c) row index (r), use this (r + (27 * (c-1)))
    # This is creating
    for i in 1:3
        ind = 13+(27*(counter))
        matvalid[ind:ind+2] = [1,1,1]
        counter += 1
    end
    lastpos = [ind:ind+2]
    # Note that I use counter to help keep track of where in the matrix we are, meaning to ...
        get the latest position, in the last equation c is changed to counter + i
    # Another note, we are only adding to the right, if we cannot add to the right, we do ...
        not add
    while ((n - counter) ≥ 1)
        #NOTE THIS NUMBER FOR THE FIRST IF IS SOMETHING I NEED TO PLAY WITH, RIGHT NOW 20 ...
            WILL WORK
        if ((n - counter) ≥ 25) # This means we have can make a turn and not have to ...
            recenter it
            # Going forward a random amount before generating a turn
            howfar = rand(2:maxfar) # number of spaces
            for i in 1:howfar
                #counter += 1 # Think the last one is correct placement
                ind = mat[lastpos[1][1]][2]+(27*(counter))
                num = lastpos[1][2] - lastpos[1][1] + 1
                matvalid[ind:ind+num] = ones(num+1)
                counter += 1 # Move all to first line in for and remove the +i in the next ...
                    line if time
            end
            #numright = min(27-ind[1][1],2)
            #numleft = min(ind[1][1]-1,2)
            num = lastpos[1][2] - lastpos[1][1] + 1
            lastpos = [ind:ind+num]
            # Now we can create the turns

```

```

        numa = 3
        numb = 3
        madeone = false
        cannot = false
        while(!madeone)
            oneortwo = rand(1:2) # To determine how many turns we are making
            if oneortwo == 1 # Only one turn
                whichside = rand(1:2) # Determine if left or right turn (1 is left 2 ...
                    is right)
                if whichside == 1 # Left turn which means we care about the first ...
                    position in lastpos
                    if ((mat[lastpos[1][1]][2] - numa) > 0) # We can make the turn
                        for i in 1:numa
                            # Note the constant of three, need to figure out what to ...
                                do with the other right side
                            ind = mat[lastpos[1][1]][2]+(27*(counter))
                            num = lastpos[1][2] - lastpos[1][1] + 1
                            matvalid[ind-numa:ind+num] = ones(numa+num+1) # MAYBE ADD ONE
                            counter += 1
                            # Since the turn is written from the center's on the left ...
                                perspective, the lastpos has to too until the turn is done
                            lastpos = [ind:ind+num] # Maybe not needed, but better to ...
                                have for now
                        end
                        lastpos = [ind-numa:ind-1]
                        madeone = true
                    else # We cannot make the turn
                        if numa ≤ 1
                            numa = 1
                        else
                            numa -= 1
                        end
                    end
                else # Right turn
                    if ((mat[lastpos[1][2]][2] + numb) < 27) # We can make the turn
                        for i in 1:numb
                            # Note the constant of three, need to figure out what to ...
                                do with the other right side
                            ind = mat[lastpos[1][1]][2]+(27*(counter))
                            num = lastpos[1][2] - lastpos[1][1] + 2
                            matvalid[ind:ind+numb+num-1] = ones(numb+num) # MAYBE ADD ONE
                            counter += 1
                            lastpos = [ind:ind+num] # Maybe not needed, but better to ...
                                have for now
                        end
                        lastpos = [ind+num:ind+numb+num-1]
                        madeone = true
                    else # We cannot make the turn
                        if numb ≤ 1
                            numb = 1
                        else
                            numb -= 1
                        end
                    end
                end
            else # Means we make two turns
                done = false
                while (!done)
                    if (mat[lastpos[1][1]][2] - numa) > 0
                        done = true
                    else
                        if numa ≤ 1
                            numa = 1
                        else
                            numa -= 1
                        end
                        done = false
```

```
                        end
                        if (mat[lastpos[1][2]][2] + numb) < 27
                            done = true
                        else
                            if numb ≤ 1
                                numb = 1
                            else
                                numb -= 1
                            end
                            done = false
                        end
                        if (numb ≤ 1) | (numa ≤ 1)
                            done = true
                            cannot = true
                        end
                    end
                    if (!cannot)
                        howmany = min(numa,numb) # This is so we have a symmetrical junction
                        for i in 1:howmany # The first split
                            ind = mat[lastpos[1][1]][2]+(27*(counter))
                            num = lastpos[1][2] - lastpos[1][1]
                            matvalid[ind-howmany:ind+num+howmany+1] = ...
                                ones((2*howmany)+num+2) # MAYBE ADD ONE
                            counter += 1
                            lastpos = [ind:ind+num+1]
                        end
                        for i in 1:howmany # Creating the two different paths
                            ind = mat[lastpos[1][1]][2]+(27*(counter))
                            num = lastpos[1][2] - lastpos[1][1]
                            matvalid[ind+num+2:ind+num+howmany+1] = ones(howmany) # MAYBE ...
                                ADD ONE
                            matvalid[ind-howmany:ind-1] = ones(howmany) # MAYBE ADD ONE
                            counter += 1
                            lastpos = [ind:ind+num+1]
                        end
                        for i in 1:howmany # Joining them back up
                            ind = mat[lastpos[1][1]][2]+(27*(counter))
                            num = lastpos[1][2] - lastpos[1][1]
                            matvalid[ind-howmany:ind+num+howmany+1] = ...
                                ones((2*howmany)+num+2) # MAYBE ADD ONE
                            counter += 1
                            lastpos = [ind:ind+num+1]
                        end
                        madeone = true
                    end
                    # I AM GOING TO HAVE TO MAKE THEM JOIN BACK UP
                end
            end
        else # This means we are getting to the end and need to center the road so we can ...
            join it up in the next one
            todes = 13 - mat[lastpos[1][1]][2]
            if todes < 0 # The center is to the left
                for i in 1:3
                    ind = mat[lastpos[1][1]][2]+(27*(counter))
                    num = lastpos[1][2] - lastpos[1][1] + 2
                    matvalid[ind+todes:ind+num-1] = ones(num-todes)
                    counter += 1
                end
                lastpos = [13:15]
                for i in 1:(n-counter)
                    ind = mat[lastpos[1][1]][2]+(27*(counter))
                    num = lastpos[1][2] - lastpos[1][1] + 1
                    matvalid[ind:ind+num] = ones(num+1)
                    counter += 1
                end
            elseif todes == 0 # We are aligned with the center
                for i in 1:(n-counter)
                    ind = mat[lastpos[1][1]][2]+(27*(counter))
```

```
                        num = lastpos[1][2] - lastpos[1][1] + 1
                        matvalid[ind:ind+num] = ones(num+1)
                        counter += 1
                    end
                else # The center is to the right
                    for i in 1:3
                        ind = mat[lastpos[1][1]][2]+(27*(counter))
                        num = lastpos[1][2] - lastpos[1][1] + 2
                        matvalid[ind:ind+num-1+todes] = ones(num+todes)
                        counter += 1
                    end
                    lastpos = [13:15]
                    for i in 1:(n-counter)
                        ind = mat[lastpos[1][1]][2]+(27*(counter))
                        num = lastpos[1][2] - lastpos[1][1] + 1
                        matvalid[ind:ind+num] = ones(num+1)
                        counter += 1
                    end
                end
            end
        end
    if lastrow
        matvalid[(27*n)-26:n*27] = ones(27)
    end

    return mat, matvalid
end

function Generate(matvalid::Vector, mat::Vector, s::Tuple, action::Symbol)
    xx = s[1]
    yy = s[2]
    s = [xx,yy]
    #Grid = reshape(mat, (length(mat)/27, 27)))
    x_p = 0
    y_p = 0
    reward = 0
    Prob = rand()
    if action == :right
        if Prob < 0.2
            choice = rand([1, 2])
            if choice == 1
                x_p = -1
            else
                y_p = 1
            end
        else
            x_p = 1
        end
    elseif action == :left
        if Prob < 0.2
            choice = rand([1, 2])
            if choice == 1
                x_p = 1
            else
                y_p = 1
            end
        else
            x_p = -1
        end
    elseif action == :down
        if Prob < 0.2
            choice = rand([1, 2])
            if choice == 1
                x_p = -1
            else
                x_p = 1
            end
        else
```

```
                y_p = 1
            end
        end

        sp = [s[1] + y_p, s[2] + x_p]
        sp_temp = (sp[1], sp[2])
        index = findall(x -> x == sp_temp, mat)
        #@show sp_temp
        #@show index
        ########################################
        ########################################
        # if termi(matvalid, mat,sp) #s[1] == Int(length(mat) / 27)
        xxx1 = sp[1]
        xxx2 = sp[2]
        xxx = tuple(xxx1,xxx2)
        if termi(xxx) #s[1] == Int(length(mat) / 27)

            reward = sp[1]*10
            aa = sp[1]
            bb = sp[2]
            ruty = tuple(aa,bb)
            return (ruty, reward)
        end
        ########################################
        ########################################

        if isempty(index)
            reward = -10
            aa = s[1]
            bb = s[2]
            ruty = tuple(aa,bb)
            return (ruty, reward)
        end

        if matvalid[index[1]] == 1
            if action == :down
                reward = s[1] + 1
            else
                reward = 0
            end

            aa = sp[1]
            bb = sp[2]
            ruty = tuple(aa,bb)
            return (ruty, reward)
            @show "Here"
        else
            reward = -10
            aa = s[1]
            bb = s[2]
            ruty = tuple(aa,bb)
            return (ruty, reward)
        end

    end

    function reward_fun(s::Tuple, action::Symbol)
        global matvalid,mat
        snext, reward = Generate(matvalid, mat, s, action)

        return reward
    end

    function trans_fun(s::Tuple, action::Symbol)
        global matvalid,mat
        snext, reward = Generate(matvalid, mat, s, action)
        return Deterministic(snext)
    end
```

```
function test_valid(mat,matvalid,snext)
    index = findall(x -> x == snext,mat)
    valid_val = matvalid[index]
    if valid_val == 0.0
        return :notvalid
    else
        return :valid
    end
end

function heu_pol(mat,matvalid,current_s)

    snext, reward = Generate(matvalid, mat, current_s, :down)
    if reward > 0.0
        return :down
    else
        action_vec = [:right, :left]
        snext, reward_right= Generate(matvalid, mat, current_s, :right)
        snext, reward_left = Generate(matvalid, mat, current_s, :left)
        ind = argmax([reward_right,reward_left])
        return action_vec[ind]

    end
end

# function termi(matvalid::Vector,mat::Vector, s::Tuple)
function termi(s::Tuple)
    global matvalid, mat
    final_row = mat[end][1]
    current_row = s[1]
    if current_row == final_row
        return true
    else
        return false
    end

end

################################################
################################################
## Creating Map ################################
################################################
################################################

n = 90
maxfar = 10
global mat, matvalid = createmap(n,maxfar,false)

## Initialize Initial Condition
global current_s = tuple(1,14)

################################################
################################################
## Saving Trajectory ###########################
################################################
################################################

fil = open("C:\\Users\\User\\Desktop\\julia_stuff\\map_text.txt", "w")
fil2 = open("C:\\Users\\User\\Desktop\\julia_stuff\\traj.txt", "w")
fil3 = open("C:\\Users\\User\\Desktop\\julia_stuff\\MCTS_traj.txt", "w")
fil4 = open("C:\\Users\\User\\Desktop\\julia_stuff\\MCTS_traj_proc.txt", "w")
fil5 = open("C:\\Users\\User\\Desktop\\julia_stuff\\proc_map.txt", "w")
#fil2 = open("states.txt", "w")

for i in 1:length(matvalid)

    write(fil, string(matvalid[i]))
```

```
    write(fil, " ")
    if i % 27 == 0

        write(fil, "\r")

    end

end

close(fil)
paths = []

###############################################
###############################################
## Simulate Heuristic Policy #################
###############################################
###############################################
global total_r_heu = 0
@time begin
while true
    global current_s, total_r_heu
    action_sym =  heu_pol(mat,matvalid, current_s)
    snext, reward = Generate(matvalid, mat, current_s, action_sym)
    if current_s[1] == n
        break
    end
    current_s = snext
    total_r_heu += reward
    push!(paths,current_s)
    # println(action)
    # println(reward)
    # println(current_s)
end
end
println("Total Reward Using Heuristic Policy")
println(total_r_heu)


theone = paths[end]

for i in 1:length(paths)

    write(fil2, string(paths[i][1]))
    write(fil2, " ")
    write(fil2, string(paths[i][2]))
    write(fil2, "\r")

end

close(fil2)

##########################################
##########################################
##### MCTS Stuff #########################
##########################################
##########################################
solver = MCTSSolver(n_iterations = 300, depth = 10, exploration_constant = 1.0)

m = QuickMDP(
    states = mat,
    actions = [:left,:right,:down],
    discount = 0.1,
    initialstate = (1,14),
    isterminal = termi,
    transition = trans_fun,
    reward = reward_fun
)

```

```
planner = solve(solver,m)

current_s = tuple(1,14)
path2 = []
global total_r_mcts = 0

@time begin
while 1 == 1
    global current_s,total_r_mcts
    action_sym = action(planner,current_s)
    snext, reward = Generate(matvalid, mat, current_s, action_sym)
    if current_s[1] == n
        break
    end
    current_s = snext
    total_r_mcts += reward
    push!(path2,current_s)
    # println(action)
    # println(reward)
    # println(current_s)

end
end
println("Total Reward Using MCTS")
println(total_r_mcts)


for i in 1:length(path2)

    write(fil3, string(path2[i][1]))
    write(fil3, " ")
    write(fil3, string(path2[i][2]))
    write(fil3, "\r")

end

close(fil3)

##########################################
##########################################
##### MCTS Procedural Map Gen ############
##########################################
##########################################


# global mat_new = []
# global matvalid_new = []
global mat = []
global matvalid = []
global current_s = tuple(1,14)
global path3 = []
global stop = 0

# for i = 1:3
for i = 1:1
    # global mat_new,matvalid_new,mat,matvalid, current_s
    global mat,matvalid, current_s, stop
    n = 45
    maxfar = 10
    mat_new, matvalid_new = createmap(n,maxfar,false)
    for j = 1:length(mat_new)
        temp_x = copy(mat_new[j][1])

        temp_y = copy(mat_new[j][2])
        temp_x = copy(temp_x)+(i-1)*n
        mat_new[j] = (temp_x,temp_y)

        temp_x = []
```

```
        temp_y = []
    end

    mat = [copy(mat);mat_new]
    matvalid = [copy(matvalid);matvalid_new]
    global stop = 0

    while stop == 0
        global current_s
        action_sym = action(planner,current_s)
        snext, reward = Generate(matvalid, mat, current_s, action_sym)
        if termi(current_s) == true
            stop = 1
        end
        current_s = snext
        # total_r_mcts += reward
        push!(path3,current_s)
        # println(action)
        # println(reward)
        # println(stop)
        # println(current_s)
    end
    # x = current_s[1]
    # y = current_s[2]
    # x += 1
    # current_s = tuple(x,y)
end



for i in 1:length(path3)

    write(fil4, string(path3[i][1]))
    write(fil4, " ")
    write(fil4, string(path3[i][2]))
    write(fil4, "\r")

end

close(fil4)

# mat = mat_new
# matvalid = matvalid_new
for i in 1:length(matvalid)

    write(fil5, string(matvalid[i]))
    write(fil5, " ")
    if i % 27 == 0

        write(fil5, "\r")

    end

end

close(fil5)
```