Code Keeper - The Hero Simulation Design Document

LS-CodeRocket, Quaz, JJoe

March 16, 2023

1 Simulation

To form the game state, we will make use of a simple class or struct (I will refer to it as a class unless needed). There will also be several variables that I will reference. In the code these variables may have names like **PlayerData** but in this sheet it will be (most-likely) referenced as a shorter variable like P. This sheet is not for documentation, it is for planning. Additionally, the notation will be lax. In a further edition I plan to make the notation fixated. This class will have five components. The player data P, the map (dungeon) data D, the monsters' data M, and the items I. The last value is the position of the exit.

$$S = (P, D, M, I, [E_x, E_y])$$

1.1 Player Data

The player data will have five components. P_x which is the player's x position and P_y is the player's y position. The maze is 16 wide and 12 high which provides a bound for x and y. Additionally, C_H, C_S and C_B are the remaining charges for the hammer, scythe and bow respectively. Then, the final component will be the player's health P_H. If the player chooses a movement that moves the player in an invalid direction, nothing will happen.

$$P = \begin{bmatrix} P_x \\ P_y \\ C_H \\ C_S \\ C_B \\ P_H \end{bmatrix}$$

1.2 Monster Data

Each monster's data will have five components. M_x which is the monster's x position and M_y is the monster's y position. The maze is 16 wide and 12 high which provides the same bound for x and y as the player bounds. Additionally, M_V is the view range of the monster. This shows how far it can see the player in any direction. M_R is the attack or damage range of the monster. This shows the minimum distance the monster has to be away from the player to attack him. M_D is the actual numerical amount of damage that the monster does. Lastly, M_H is the monster health.

If it reaches zero, the monster disappears. In this model, it never will however we will discount monsters with no health in calculations.

$$M = \begin{bmatrix} M_x \\ M_y \\ M_V \\ M_R \\ M_D \\ M_H \end{bmatrix}$$

1.3 Item Data

Each item's data will have six components. I_H, I_S, I_B, I_P and I_T are all binary flags. These determine whether the item is a hammer charge, scythe charge, bow charge, potion or treasure. The flag becomes 1 in the corresponding slot when it is of a certain type. The last two components are I_x and I_y which show the position of the item.

$$I = \begin{bmatrix} I_H \\ I_S \\ I_B \\ I_P \\ I_T \\ I_x \\ I_y \end{bmatrix}$$

1.4 Action Data

An action that the player selects before moving will be stated as a three component class (or struct). T is the action type. There are two types of actions, moves and attacks. T is a vector with two components. If the first value in the vector is set to 1, then the move is a move operation. If first component of the vector is 0, then it is an attack operation. T_2 will be 0 when choosing the sword, 1 when choosing the hammer, 2 when selecting the scythe and 3 with the bow.

$$A = (T = [T_1, T_2], x, y)$$

1.5 State Evolution

We will define a function called **NextState** which takes in the current state $S(n)$ and the action for the turn A and turns it into the next state $S(n+1)$. The function actually is a set of three functions (technically four but not really) that convert the first state into the second. $P(n+1) = f_p(P(n), A, I)$ is a function that, when given the current player data and the chosen action, outputs the next player state. This function will also take into account items that the player is currently over and update the corresponding values. $M(n+1) = f_M(f_p(P(n), A), A, M(n))$ is a function that takes in the player data and the action and outputs the next monster state. $I(n+1) = f_I(f_p(P(n), A), I(n))$ is a function that takes in the next player state and the current item state and determines whether certain items disappear.

$$\textbf{NextState}(S(n), A) = S(n+1) = (f_p(P(n), A, I), D, f_M(f_p(P(n), A), A, M(n)), f_I(f_p(P(n), A), I(n)))$$

1.6 Helpful Functions

We will also define a multiplexing function. It will take the input number and then output a unit vector in the dimension direction as the numbered input.

$$\omega(n) = \vec{e}_n$$

We will also define two functions δ and θ which will help with creating conditional statements in the equations without repeated use of piecewise functions in Latex. Basically, ease of writing and programming. $\delta_n(x)$ is a function that takes in an element x and if it's equal to zero (or the zero vector), it returns 1. $\theta_n(x)$ is the opposite. Whenever x is zero, it returns zero. Else, it returns one. We will abbreviate $\delta_1(x)$ as $\delta(x)$ and $\theta_1(x)$ as $\theta(x)$.

$$\delta_n(x) = \begin{cases} 1 & \text{if } x = \vec{0}_n \\ 0 & \text{else} \end{cases}$$

$$\theta_n(x) = 1 - \delta_n(x)$$

We will also define a range function $\phi(a, x, b)$ to determine whether or not x is between a and b inclusive.

$$\phi(a, x, b) = \begin{cases} 1 & \text{if } a \leq x \leq b \\ 0 & \text{else} \end{cases}$$

1.7 Player Update 1

The player update function will be separated into two conditions. If the player did not move this turn, the move part of the function goes to zero. This means the move part will be multiplied by T_1. Next, if the player is not going to attack this turn, the attacking part of the function will go to zero. The left-most term on the right side just is the player data that is started with. The next term to the right is the movement term which finds the distance between the current player position and the intended player position and then adds the x and y values to the player value if the movement flag is enabled. Then the last term subtracts the charge based on the given weapon. The delta term is to make sure that it runs when the move flag is not enabled. The theta term is to make sure that sword charges are not decreased because there are infinitely many of them. Additionally, the 1 part on the function shows that it is only the FIRST function. We will apply this function first and then the player damage function next.

$$f_p^1(P, A = (T, x, y)) = P + \begin{bmatrix} P_x - x \\ P_y - y \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix} (T_1) - \omega(T_2 + 2)\delta(T_1)\theta(T_2)$$

1.8 Monster Update

The monster update function will be separated into two different updates (the rest will remain the same. The first function will update the monster's position. The inside component with the minimizing function finds the quickest path through the obstacles to reach the player. The enemies only move when the player is in their view. Therefore a V_M^i term needed to be multiplied to make sure the actual moving term did not result in a false movement. The minimization condition in the argmin just says the enemy has to find the shortest path through the maze such that the enemy doesn't leap over voids or other enemies. We define O to be the obstacle set.
The second part of the entire thing is the **CalculatePlayerDamage** function. This damage function will compute the damage done to this specific enemy based on the action that was completed. I will specify what this means later (or perhaps not even in this document).

$$V_M^i = \phi(-M_V^i + M_x^i(n), P_x(n+1), M_V + M_x(n)))\phi(-M_V^i + M_y^i(n), P_y(n+1), M_V + M_y(n)))$$

$$O = \{(x,y)|D_{xy} = 0 \text{ or } \exists M^i | M_{xy}^i = (x,y)\}$$

$$f_{M_i}(f_p(P,A), M_i, D) = \begin{bmatrix} M_{xy} + V_M^i(M_{xy} - [\arg\min_{\gamma(t)|\gamma(0)=M_{i,xy},\text{ not } \exists\gamma(t)\in O \text{ and } \gamma(t_f)=P_{xy}} t_f](1)) \\ M_V \\ M_R \\ M_D \\ \max(M_H - \textbf{CalculatePlayerDamage}(A,M), 0) \end{bmatrix}$$

1.9 Player Update 2

We can now define the second part of the player update function. We need to analyze the monster movements to make sure that the enemies attack when they're in range. Because of this, we define **CalculateMonsterDamage** to calculate the damage done by monsters after the player's movements. We also define **ItemLogic** to add to the player's state based on the item that the player is standing on (if any).

$$f_p^2(P,M) = P - \textbf{CalculateMonsterDamage}(P,M) + \textbf{ItemLogic}(P,I)$$

1.10 Item Update

Now, we define an itme update function which checks to see if the ith item is coming in contact with the player. If it isn't in contact with the player, it just stays the same.

$$f_I^i(P,I_i) = \delta_2(I_{i,xy} - P_{xy}) \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix} + \theta_2(I_{i,xy} - P_{xy})I_i$$

1.11 Multithreading

Multithreading will be dealt with by adding multiple instances of this program at the same time. The data log generated from these will be separated into folders and files.

1.12 Optimization

There will be a numerous number of constants in the optimization solution of this game. Therefore, we will repeatedly test multiple functions in given ranges at the beginning. It will be solved using gradient descent or just a brute force algorithm with a certain step size to keep things simple and be able to look for a more global minimum.

This marks the end for the simulation section.

2 Algorithm

To choose an action, we need to decide between two different types of actions. We need to decide between moving and attacking. We will move whenever enemies are not agro'd on us. If an enemy cannot reach the player, we will still try to attack them. The Δ determines whether to move or to attack. The x^* and y^* are the coordinates of the attacking square or the square to move to.

$$\Delta = \delta\left(\sum_i |\phi(0, M_{ix} - P_x, 3)\phi(0, M_{iy} - P_y, 3)|\right)$$

$$\textbf{SelectAction}(P, D, M, I) = \left(\begin{bmatrix}1-\Delta \\ \Delta W\end{bmatrix}, x^*, y^*\right)$$

2.1 Movement

Assuming that the player will move, we will find a destination (final) square F to return to. We will define what that square is later. To find this next square in the path, we first compute the path $\gamma(t)$ in the beginning. Then, we just run through the path every frame so we don't have to keep computing it. When we reach the next square in the sequence, we delete that node from the path. When the path has ended, we recalculate the path for the next square.

$$(x^*, y^*) = [\arg \min_{\gamma(t)|\gamma(0)=P_{xy}, \text{not } \exists \gamma(t)\in O \text{ and } \gamma(t_f)=F} t_f](1)$$

To compute F, we look at enemy locations, potion locations and treasure locations. We define a cost function J that determines the loss made from going to a certain square. Actually, we also define an **TotalItemCost** function to determine the cost for stepping on an item. Sometimes potions can have a positive cost depending on how much health the player has. Otherwise, items will usually have a negative cost (meaning positive gain) which will help the player when they're low on charges. Additionally, we will have a term that determines the amount of cost given to monsters that are agro'd. **MVS** stands for **MonstersViewingSquare** (not Most Valuable Square). The last component has to do with the player having to exit the map when time is running out. t_E is a constant where if we set it to something like 75, that part of the function will start to return positive values.

$$J(x,y) = k_{\text{Monsters}} \left(\sum_{M \in \mathbf{MVS}(x,y)} k_H M_H + k_D M_D + k_V M_V \right)^{\alpha_1} + k_{\text{Items}} \left(\mathbf{TotalItemCost}(P, I, x, y) \right)^{\alpha_2} +$$

$$k_{\text{Time}} \delta_2((E_x, E_y) - (x,y)) \left(t_E - t\right)^{\alpha_3}$$

$$F = \arg \min_{(x,y) \in \{(x,y) | (x,y) \text{ is next to a } -1 \text{ or its next to map edge cell in } D_{ij}\}} J(x,y)$$

2.2 Weapon Damage

To choose the weapon, we just brute force an argmin of a cost function. This cost function calculates the damage that a player WOULD do to the monsters if the attack was to go through. We then add onto that a cost for the weapon because a charge is removed if you use it.

$$W = \arg \min_{(w,x,y)} k_{\text{Damage}}([\max(V - \mathbf{WeaponDamageMatrix}(w,x,y,P), 0)]_{ij}, V, A_w)^{\alpha_1} + k_{\text{Counter}}(C_{\max} - C_w)^{\alpha_2}$$

3 Functions I Used but Didn't Define

3.1 **CalculatePlayerDamage**(A, M)

This function uses another very important function that I have yet to define:

$$Z = \mathbf{WeaponDamageMatrix}(w, x, y, P)$$

CalculatePlayerDamage(A, M) will check if the passed-in monster is in viewing range of the player, it will take that monster's position and find out how far away it is from the player in both dimensions by subtracting the monster position from the player position. We add 3 to this quantity to find the corresponding Z value. It then returns that Z value.

3.2 **CalculateMonsterDamage**(P, M)

This is far more simple. Basically for all the enemies in the monster set, if the player is within damaging range, they will attack. Therefore it sums up the enemy damages (of the enemies that are in damage range of the player) and returns that sum.

3.3 **ItemLogic**(P, I)

This function checks if an item is intersecting a player and if so, it applies its effect to the player.

3.4 $\arg\min_{\gamma(t)}$

To find a gamma that travels somewhere, we will use a breadth first search and we will apply a bias to search through paths that decrease the distance between the squares to improve efficiency.

3.5 **TotalItemCost**(P, I, x, y)

Of all the items that we know are on the ground, we assign a negative (or sometimes positive for potions) cost for getting the item. The more benefitial the item, the more likely we are to go for it. We have to make sure that (x, y) is actually that item however.

3.6 **MVS**(M, x, y) / **MonstersViewingSquare**(M, x, y)

This returns an array of all the monsters that have viewing access to this square (x, y).

3.7 **WeaponDamageMatrix**(w, x, y, P)

This returns a 7-by-7 matrix describing the damage that the player would do if the weapon w is selected and attacking (x, y). There is a sub-function for every w. It's kinda just a chain of if statements that generates the matrix. That matrix is returned.

4 Conclusion

Yeah I'm pretty much done.