Treasure Hunt

A* agent against semi-stochastic Enemies

Jorge Eduardo Aparicio Rivera
Control Systems and Artificial Intelligence Research Group
Universidad Nacional de Ingenieria
Lima,Peru

Manuel Arturo Deza Figueroa
Control Systems and Artificial Intelligence Research Group
Universidad Nacional de Ingenieria
Lima,Peru

***Abstract*—In this paper we propose a Genetic Algorithm to optimize 3 parameters for a videogame based scenario we call 'Treasure Hunt'. Our main goal is to solve the problem of: How do I get to the treasure with the maximum amount of Hit Points, thus modeling it as an Optimization problem. To achieve this we use an XNA videogame framework to program the A* algorithm to our Hero, and Semi-stochastic behavior to model the Enemies.**

Keywords- A Star, Genetic Algorithm, Optimization

I. INTRODUCTION

Path planning is a problem in the realm of Artificial Intelligence where we are given the task of finding the minimal cost of a path from point A to point B. Various methods exist for solving this problem, some of these include : BFS (Breadth-First Search), DFS (Depth-First Search) and Greedy Search [1]. These types of algorithms are frequently used in route optimization, which is easily observable in any high-tech GPS. Moreover, some areas in Robotics use these techniques not only limited to 1 agent, but to various agent, for cooperative or swarm intelligence [5].

We call our paper "Treasure Hunt" because we implement the A* algorithm in a somewhat playful dungeon-game scenario, similar to the Super Nintendo game of "The Legend of Zelda: A link to the past". Our Agent is an avid explorer looking for the optimal route from his initial point A to the treasure (Flag): point B. But the dungeon is infested with monsters wandering around, looking for food. Our unlucky agent must run, and get to the treasure quick before he is seen or detected by the monsters, so he decides to choose the A* search algorithm for the optimal route. Unfortunately, even though the Agent optimizes his path, the Enemies are equipped with an internal Biosensor that tells them the position of the Agent explorer whenever he is close, forcing them to smell his flesh and follow him stochastically in that direction. We call this 'blood seeker' mode. Figure 1 illustrates this scenario.

II. DUNGEON MODEL

A. Map Specifications

We model our problem by taking in the following considerations:

- We have a finite map –modeled as a Matrix - of known dimensions (50 x 25)
- Obstacles/Walls exist in the dungeon. These walls are impenetrable.
- There are 5 Enemies/Monsters in the dungeon and they respawn at the upper left corner only if slaughtered.
- There is only 1 Agent Explorer from the beginning to the end of the game.
- The goal is to make the Agent get to the treasure with the maximum amount of Health, having in mind that fights with enemies decrease Health, as well as a Health penalty for every move.

B. Algorithm

```
Initialize Map and Parameters;
Load Content;
Upload Game Data;
While (Health  > 0)
        If Remapping Rate Achieved
                Remap and Draw Fairy Path;
        Else
                Link.TakeAction();
                Enemy.TakeAction();
                Add action to counter;
                Draw Sprites;
End
```

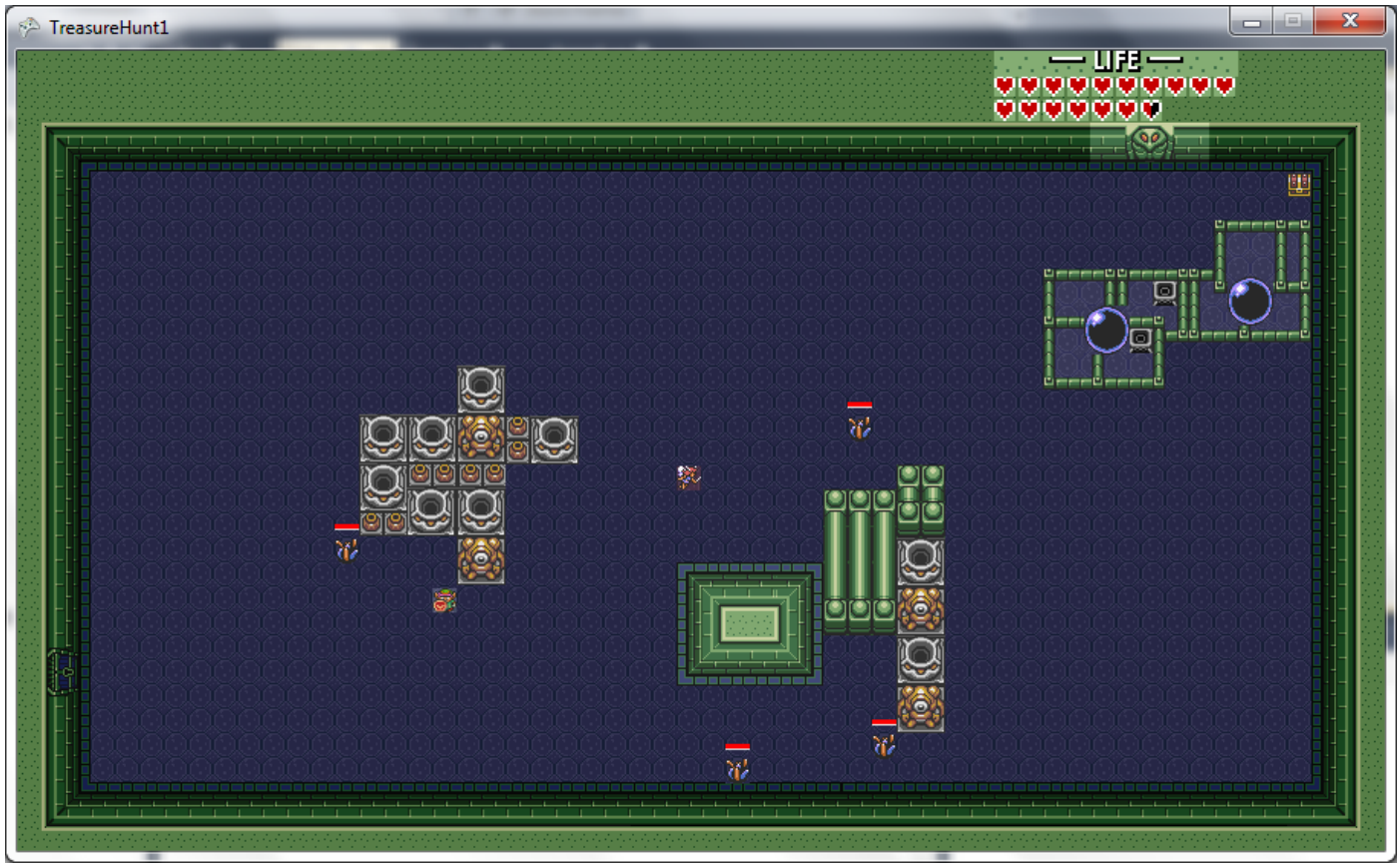

Figure 1: This is our Dungeon map with different sprites representing the characters and objects in the game. Notice Link, our explorer, in the middle; 4 enemies with different health bars; a treasure chest, at the top right corner; and a fairy that glides from the treasure to the explorer. Obstacles in the map are represented in the matrix to make them congruent with the nonuniform textures that appear in the map. Finally, a Life gauge displays Link's Health in direct proportion, being 20 Hearts the initial standard.

C. Parameter Optimization

A Clock represented by a 'Ticks' variable counts the elapsed time and Agent and Enemy speed are controlled by parameters according to the Ticks. The Agent starts with 200 Health and must arrive to the Flag still alive: Health > 0. In some occasions he does not arrive to the Flag due to Enemy persecution and fighting leaving the Agent without any Health left. This happens because our Agent models the enemies as pseudo-obstacles as well, and he not only wishes to arrive to the Treasure, but also wants to avoid getting into fights with the Monsters if his Health is under the HealthThreshold parameter. In other words, we control only 3 parameters manually: Health Threshold, Enemy pseudo-obstacle radius, and Remapping Rate.

Remapping Rate: This last parameter is the Rate at which the Agent re-computes the A* Algorithm, thus "re-freshing" the last trajectory by a new one.

Health Threshold: This is the Health Threshold that if the Agent's health is lower or equal than this, then he will consider Enemies as obstacles in the A* computation.

Enemy Pseudo-obstacle Radius: This is the permissible Enemy radius drawn to simulate obstacles for the Agent in his A* computation for avoidance only if the Agent's Health is lower than the Health Threshold.

If the Agent is in the Enemy Vicinity then Boodseeker mode is activated. More will be commented about this in the Enemies section.

III. AGENT

A. Movement

The A* Search algorithm is implemented by a Heuristic Euclidean distance map from the Agent's initial point to the Agent's final point [3]. Each new step, the Agent analyzes the map and looks for the minimum cost and minimum heuristics in each option. To make the model more realistic, we also made the Agent try to arrive to the Flag/Treasure, but at the same time he tries to avoid the enemies if they start chasing him. To do this, the Agent considers the enemies as pseudo-obstacles in the Map variable swapping labels momentarily. More about this is commented in the Enemies section.

Notice how we represent our Heuristic Map, in figure 2, by an image grayscale gradient. The brightest regions represent lowest Heuristic costs, while the darkest regions represent higher heuristic costs. This is congruent with our route from point A to point B, being point A the lower left corner where our Agent always initializes his position, and point B the upper right corner where the treasure will always wait. In previous experiments we used to add randomized factors to point A and point B, but this was later found to be a disadvantage for the Genetic Algorithm Optimization.

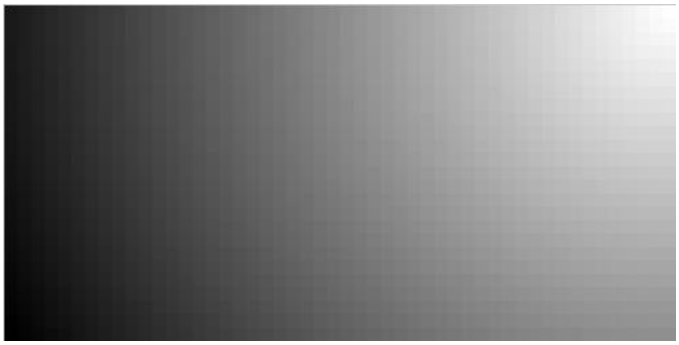

Figure 2: Image Representation of the Heuristics.

To model the turn taking dynamics in the game, every action comes with a time penalty or as we call it in our code "TimeForNextTurn" variable. This variable applies both to the Agent and the Enemies independently, as it is a class member function. Table I has more information on the values.

TABLE I. TimeForNextTurn

	Action	TimeForNextTurn
Agent	Guard Mode	5
	PathFinding	20
	Non-Diagonal Move	10
	Diagonal Move	14
	Fight	10
Enemies	Non-Diagonal Move	10
	Diagonal Move	14
	Fight	10
	Respawn	200

We observe that the operations that take most time are Pathfinding for the Agent and Respawning for the Enemy. The Movement cooldown however in the Enemy is less than the Movement cooldown of the Agent, implying that the Enemy will have a priori information of where the Agent is, increasing the probability of catching him up and encounter him in a fight.

IV. ENEMIES

A. Movement

Enemies are programmed with the following instructions:

- Random Mode: If Agent is "far", then wander randomly around the dungeon.
- Blood Seeker Mode: If Agent is "close", then chase him stochastically.

We use the word "stochastic" because our chasing algorithm is not deterministic, thus even though it knows where the Agent is, the enemy's next move is still random, but with higher probability of getting close to the Agent.

Enemy (i) Movement Algorithm
1 Set Blood Seeker Distance (BSD)
2 Get Euclidean Distance (ED) from Enemy(i) to Agent
3 Compute Relative Row and Column Distance
4 If ED <BSD
5 Generate Random Number 1 and 2;

6 **If** Random Number 1 > Relative Row Distance
7 Move 1 Row closer to Agent;
8 **Else**
9 Don't move Vertically;
10 **If** Random Number 2 > Relative Column Distance
11 Move 1 Column closer to Agent;
12 **Else**
13 Don't Move Horizontally;

14 Else
15 Move randomly to any direction;
16 If position is not-occupied
17 Execute Move;
18 Else
19 Recompute new possible move;

Where the Relative Row Distance RRD is given by:

$$RRD = \frac{Row\ Distance}{Blood\ Seeker\ Distance}$$

And the Relative Column Distance RCD is given by:

$$RCD = \frac{Column\ Distance}{Blood\ Seeker\ Distance}$$

We also mentioned that enemies in the A* algorithm are considered as obstacles, making the Agent dodge them depending on the value of the assigned radius. Figure 3, shows how we propose this criteria.

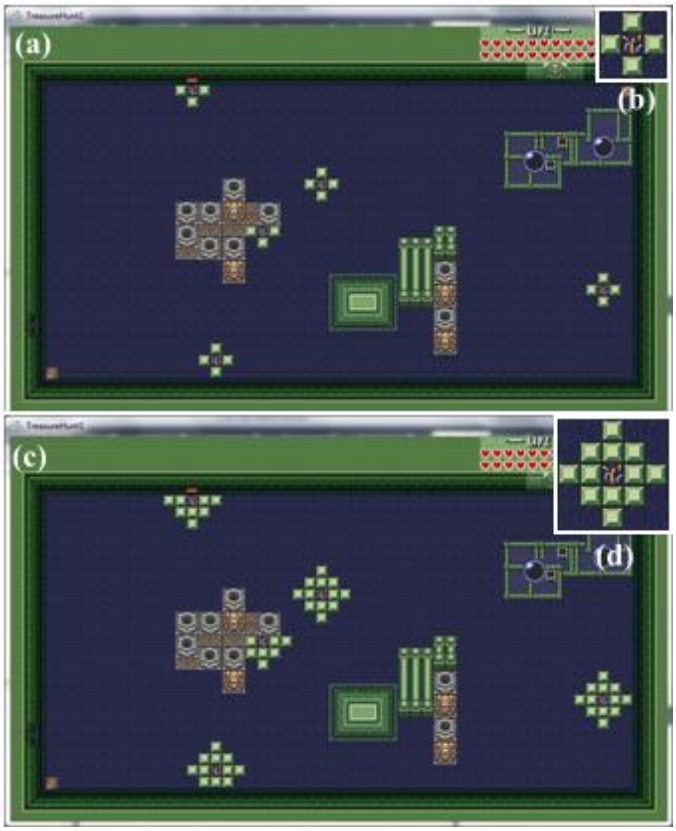

Figure 3: (a) Enemies are surrounded by obstacles of radius 25. (b) A close-up of an enemy surrounded by a radius of 25. (c) Enemies are surrounded by obstacles of radius 50. (d) A close-up of an enemy surrounded by a radius of 50.

In Figure 4, we also verify a distributed configuration of Initial Enemy placement to make the game variable, still within some random context. Getting an average of multiple game trails with the same 3 parameters and different initial enemy placement allows us to infer more robustly the 3 optimized values with a Genetic Algorithm.

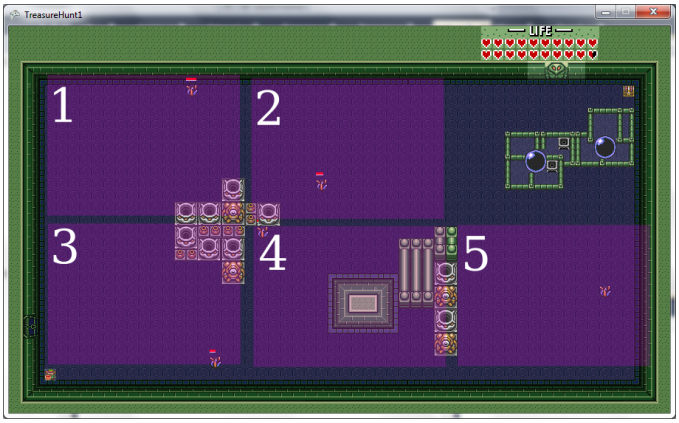

Figure 4: Six sections of equally spaced areas are created where only 5 of them are used for initial Enemy placement. Two random numbers between these intervals generate the initial Enemy position.

V. COMBAT SCENARIOS

When the Agent and Enemy are close enough, they encounter a fight. Then a dice roll determines how much damage is taken to the Agent – Critical hits are also available for both the Agent and the Enemies. Damage taken is important, because our Agent must arrive to the Flag still alive (Health > 0), and we must remember that each move he takes decreases his health in 1 HP. All enemies begin with 30 Health (Figure 6).

As we observe in both Pseudocodes, Link or the Enemy can initiate battle, and the other responds in return. Depending on the remapping rate, battles can be interrupted, but when they are not, these last until the Enemy or Link have drained each others health completely.

After an enemy is slaughtered, a time delay called Respawn time passes until it spawns in the upper left corner, with its full 30 hit points restored.

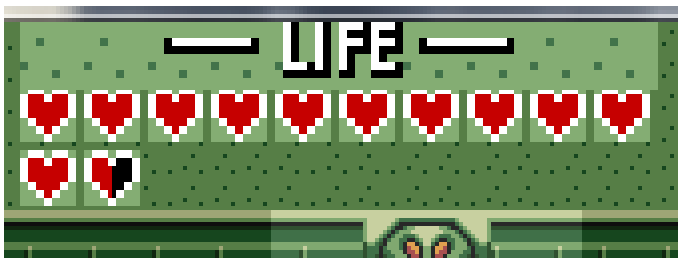

Figure 5: Link/Agent life gauge. 2 x 10 Matrix representing 20 initial hearts.

Figure 5, shows a close up on Link's Health status bar that is discreetly proportional to the initial 200 Health. Figure 6, on the other hand shows a close-up on the Enemy status bar's, which are discreetly proportional to the 30 Hit Points.

Figure 6: Enemy status bar, represented by a red colored bar placed over the Enemy. Each enemy has its own independent status bar. In specific instant of time the Enemy has only half its status bar full.

A. Enemy Fighting Algorithm

```
1  Enemy(i).Fight()
2        Enemy(i) attacks Agent;
3        If Link is Dead
4              Game Over;
5        Else
6              Link fights back: Link.Fight(i);
7        If Enemy is Dead
8              Remove Enemy from Map;
9              Respawn Later;
```

```
10      Else
11          Continue;
12 End
```

B. Agent Fighting Algorithm

```
1 Link.Fight(i)
2       Link attacks Enemy(i)
3       If Enemy(i) is dead
4           Remove Enemy(i) from Map;
5           Respawn Later;
6       Else
7           Enemy(i) Fights back: Enemy(i).fight;
8       If Link is dead
9           Game Over;
10      Else
11          Continue;
12 end
```

VI. GENETIC ALGORITHM

We use a Genetic Algorithm as mentioned previously to optimize these parameters into a fitness function [2]. First of all we have a TreasureHunt function with input values: Remapping Rate, Health Threshold and Enemy pseudo-Obstacle Radius. The output is the final Health of the Agent, which can be a positive number assuming he arrives alive, or 0 assuming that he didn't make it to the treasure. However to make the GA maximization more robust, as what we have is a Health maximization problem, we average 10 computations of the TreasureHunt function, for the same 3 parameters.

A. GA PSEUDOCODE

```
1 Create the population;
2 Random Initialize each population's individual chromosomes
3 For i = 1 : Generation
4       Compute each population's individual fitness;
5       Create empty new population;
6       Put the Elite population individuals in the New
        Population;
7       While New Population size != Population Size
8           Select two individuals of current population,
            using roulette.
9           If (Random < CrossOver rate)
10              Perform single point Crossover;
11              Add their children to new
                population;
12          Else
13              Add selected individuals to new
                population;
14      End
15      For j = Elitism : New Population Size
16          If (Random < Mutation rate)
17              New Population individual mutates;
18      End
19 End
```

Genetic Algorithms are computationally expensive and optimizing the values can take hours or days depending on how we choose the Genetic Algorithm parameters. To save time, we turn off the graphics engine, and only compute relevant values and functions of the computer game.

Once we programmed the Genetic Algorithm Table II, shows us which values we chose to assign to each variable for this Optimization. In addition, Figure 7 shows a visual representation of our Genetic Algorithm GUI that allows us to input these values.

TABLE II. Variables and Parameters range.

	Variable	Minimum	Maximum	Constant
Treasure Hunt	Health Threshold	1	200	-
	Obstacle Radius	1	50	-
	Remapping Rate	3	50	-
Genetic Algorithm	Population	-	-	50
	Generations	-	-	100
	Crossover Rate (%)	-	-	60
	Mutation Rate (%)	-	-	4
	Elitism	-	-	4

ACKNOWLEDGMENT

We would like to thank our Professor M.Sc. David Achanccaray from Universidad Nacional de Ingenieria, for advising our Project. We also like to credit the game Zelda: A Link to the Past, as all image Sprites were modified from this game, and using the Microsoft XNA Videogame Framework [4].

CONCLUSIONS AND FUTURE WORK

We conclude that the Agent's Philosophy is based on:

- Maximizing the Health Threshold, thus ignoring the enemies only on the first mapping then runs.
- Maximizing the Obstacle Radius, when running maximizes the distance from enemy.
- Maximizing the Remapping Rate, doesn't recompute the optimal path often, as it's time consuming.

This can be explained as follows, in the first A* mapping, the agent ignores the enemies. After taking a step its health falls below the threshold, therefore the agent ignores the enemies as he marches towards the treasure, and because the Remapping rate is so high, it makes the Obstacle radius oblivious. And even if the Remapping Rate wouldn't be considered, the Agent would not consider the Enemies as obstacles because of its elevated radius.

For our future work, we propose Blocks so that the Obstacle Radius can be visible to the observer during the game. In addition, we intend to make the game map more complex and labyrinth like, this will make it more entertaining for the spectator, thus making the A* search for different possibilities.

REFERENCES

[1] P. Norvig & S.Russell, Artificial Intelligence: A Modern Approach (2009)

[2] S. Marsland, Machine Learning: An Algorithmic Perspective (2009)

[3] P.Lester, A* Pathfinding for Beginners (2008) http://www.policyalmanac.org/games/aStarTutorial.htm

[4] A. Reed, Learning XNA 4.0 (2011)

[5] R. Brafman, J. C. Latombe, Y. Moses, Y. Shoham. Applications of a logic of knowledge to motion planning under uncertainty. Journal of the ACM (1997) .