Optimal Strategy generator for Tower of the Sorcery Using Language Model

Danny Xu, *Author*

***Abstract*—This project aims to find optimal paths and strategies for Tower of the Sorcery, a 2-D platform matrix game in which the goal is to beat the Demon with the potions and enhanced weapon picked up along the path. The strategies will be stored and calculated using the Genetic Algorithm(GA). Tthe project aims to generate creative routes that could defeat the target and give analysis on the pros and cons for certain strategy. This could further be developed to solve for optimal routes for other complex matrix games.**

***Index Terms*—Genetic Algorithm(GA), Tower of the Sorcery, optimal route, path-finding**

I. Introduction

THE optimal solution for Tower of the Sorcery is defined as the score after defeating the Boss, which is directly proportional to the maximum health remains. In this game, the main character starts with some number of health, attack damage, and defense power. The game is in a 2-D matrix spaces filled with **obstacles**, **enemies**, **supplies**, and **special props**. The traditional way for finding optimal routes involves testing different paths to find the local maximum on certain area of the matrix before moving on manually, which is both time consuming and players often fall into pitfalls. This project introduces a novel predictive framework employing **genetic algorithm(GA)** to analyze the feasibility of the routes, combining data structures such as **breadth first search(BFS)**, aiming to help players quickly eliminates the non-optimal solutions, thus being able to complete the game with a higher score.

II. GAME LOGICS

A. Main Logic

The main logic behind the game is to collect items, defeat enemies along the way, and ultimately beat the boss. The Hero can be numerified as a class with the following properties: **health**, **attack**, **defense**, **shield(mdef)**, **keys**, and **money**. The enemy can be expressed as a similar character. The damage from an enemy is calculated by the formula:

$$\text{Damage} = (E_{\text{atk}} - H_{\text{def}}) \times \frac{E_{\text{hth}}}{H_{\text{atk}} - E_{\text{def}}}$$

Where:

- E_{atk}: Enemy's attack power.
- H_{def}: Hero's defense power.
- E_{hth}: Enemy's health.
- H_{atk}: Hero's attack power.
- E_{def}: Enemy's defense power.

The function $F(\text{hero}, \text{map\_state})$, where `map_state` represents the places the hero has been to, should calculate the maximum health left for the hero after traversing the map and battling enemies.

The result of $F(\text{hero}, \text{map\_state})$ must always be non-negative.

B. Map Traversal

The game is played on a two-dimensional grid. The hero can move in four directions: **up**, **down**, **left**, or **right**. If the hero lands on a grid containing an item, the item will be collected. Walls, enemies, and doors act as obstacles. The hero requires specific keys to open different types of doors. The hero must defeat an enemy to move past it and cannot pass through walls.

Stairs in the game appear in pairs. An upstairs on floor i connects directly to the downstairs on floor i+1.

III. PROPOSED METHOLOGY

The project will employ a multi-phased approach, which includes data acquisition, model development, interface design, testing, and deployment, which were discussed in the proposal. This report would only discuss the progress on each step

- **Data Acquisition**: A JS executable was created to parse the data from the game website. User would have to manually collect the output from console, and save it to a local file for model training. The console displays the parsed data for user to better understand what the data looks like.
- **Model Development**: The tower matrix was turned into 2-D adjacency lists. **Breadth first search(BFS)** was employed to find all the connected blocks. Edges were scanner thoroughly to get the optimal path.
- **Testing and Evaluation**: The model was tested on a few simple towers. It was able to find the optimal solution for each of them. However, when the level of tower exceed 3, a significant slowdown was observed.

IV. DATA ACQUISITION

To retrieve data, A **javascript** was created to parse data from the website. The **main.floors['floorId'].map** contains the map for the game. The script parsed that into a readable format and feed them into the project files. Unfortunately, there isn't a unified function for the enemy informations. Each author creates different function to, so the script would only work for towers created by some authors, but not the others.

Department of Computer Science, San Jose State University, San Jose , California.

V. MODEL DEVELOPMENT

This is a brief description of how each class functions to develop the model.

- **Graph.java**: The Graph class represents the 2-D platform. It contains nodes, edges, and various elements like keys, doors, and enemies. It uses the input from users to build and merge nodes to transform the game into a graph. It also includes a method to run BFS for finding solution.
- **Node.java**: The Node class represents a connected group of nodes in the graph. It contains information for hero, enemies, and items you can on this node. Nodes are linked to each other to form the graph structure.
- **State.java**: The State class represents a certain state of the game. It keeps track of the current node, visited nodes, and the route taken. The states were compared to find the optimal paths to beat the boss.
- **Main.java**: The Main class is the class uses to run the entire program.
- **Hero.java**: The Hero class saves the current status of the hero.
- **Enemy.java**: The Enemy class saves the status of an Enemy.
- **Item.java**: The item class represent a set of items in a certain area. It was saved in the Node class and updated once two node got connected.
- **Prop.java**: The Prop class saves the actual values for each type of item.
- **Util.java**: The Util class saves the core logic of the game, in this case the formula for damage.
- **Chromosome.java(under construction)**: The chromosome class saves the genetic information that was used in the GA algorithm
- **GeneticAlgorithm.java(under construction)**: The GeneticAlgorithm class represents the logic behind the GA algorithm. It takes the graph as input and apply the parameters on each path to estimate its value, and performs reproduction over generation to find the paths the yields the best result.

The behind the model is that the input would be taken from user. The data would be use to initialize **hero**, **enemy**, **props**, and **graph**. The map is build upon the completion of inputs. During map building, each nodes were initialized. Nodes is merged after successful building of the map. The nodes that hero can access **without the need to open door/decrease health** will be linked together. After that, the node that contains hero will be placed in a **priority queue**. For each iteration, the node in the top of the queue will attempt to link with each of its neighbor. If the resulting remaining health is greater than the previous health on the state, the node will merge and be pushed into the queue. The process continues until **max iteration is hit** or **the queue is empty**. When the terminate condition is met, the program will exit whether a solution is found or not.

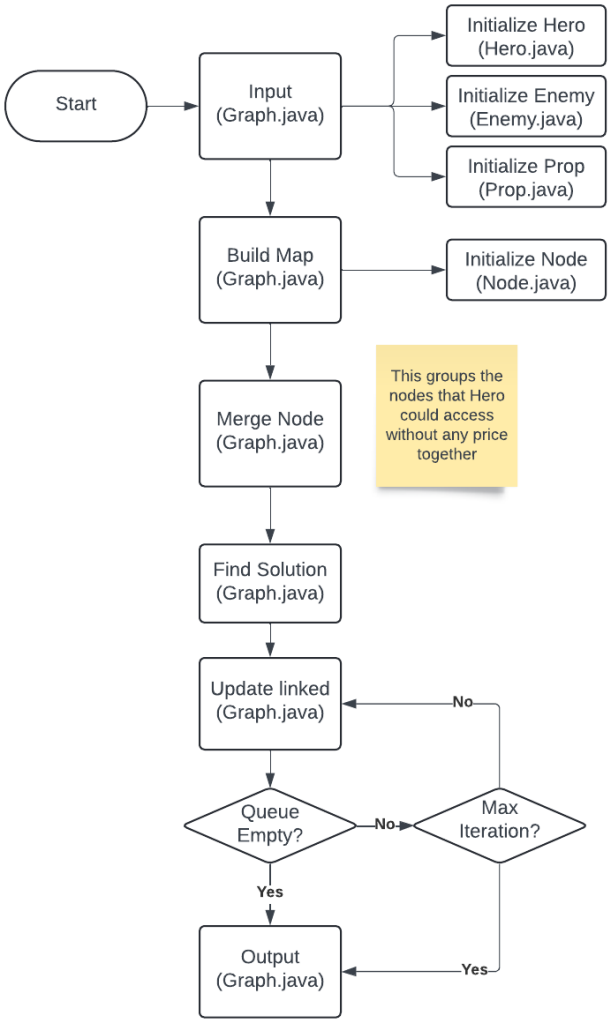

Fig. 1. Enter Caption

VI. CHALLENGES AND SOLUTIONS

- **Data Heterogeneity**: Since each tower constructed by different author might have different special props, and the map would be slightly different in turns of how they are shown. Adjusting the script for parsing data is a simple yet complicated process. To fully address the problem, a image should be build to recognize the map.
- **Model Building**: The data for Tower of Sorcery is very limited, the data provided initially could be very far from the optimal strategy. Instead of building traditional data like transformers or GPTs, which requires lots of data, the project is better off with heuristic algorithms like ACO or GA.

VII. EVALUATION METERS

Since this project has only one goal: to maximize the health after defeating the Boss, the evaluation factor for a solution is completely based on the health value.

However, there are a few other factors that could come into play when evaluating the middle layers of a strategy: player's

current attack value, defense value, and the number of keys player has. These values would help the model better evaluate whether it is going on the right track.

VIII. REAL WORLD VALIDATION

Real world validation involves validating the accuracy of the model in towers constructed by other authors. These testsets could easily be find on the website h5mota.com.

The next step for this is creating a Javascript that takes the strategy as input, and runs corresponding steps on the website.

IX. REQUIRED RESOURCE

Since it turns out that there is no need to train LLMs, a computer CPU is enough.

X. EXPECTED OUTCOME

The completed model should be able to provide a solution for the input matrix that approaches the theoretical maximum remaining health. Users should also be able to request a solution for some local area of the matrix. Furthermore, this model should be able to help tower constructors in determining if their tower is playable, and create more challenging towers based on the machine

XI. LEGAL COMPLIANCES

Since the project is designed to generate solutions for single-player non-profit gaming platform, there is no such compliances necessary.

REFERENCES

[1] Robertson, G., Watson, I. (2014). A Review of Real-Time Strategy Game AI. AI Magazine, 35(4), 75-104. https://doi.org/10.1609/aimag.v35i4.2478

[2] Hafiz Adhiyasa Pratama, Adila Alfa Krisnadhi, Representing Dynamic Difficulty in Turn-Based Role Playing Games Using Monte Carlo Tree Search, 2018 International Conference on Advanced Computer Science and Information Systems (ICACSIS), 10.1109/ICACSIS.2018.8618167, (207-212), (2018).

[3] C.K., Mota-js: Open-source turn-based role-playing game framework, [Online]. Available: https://github.com/ckcz123/mota-js. [Accessed: Dec. 8, 2024].