

A Graph-Based Approach to Action Sequence Optimization in Turn-Based RPGs: A Case Study of Honkai: Star Rail

Lutfi Hakim Yusra 13523084¹
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jl. Ganesha 10 Bandung 40132, Indonesia
13523084@itb.ac.id, luthfihakimyusra@gmail.com

Abstract—This paper presents a computational model to solve the action-sequencing problem within the popular RPG, *Honkai: Star Rail*. We model the game's combat as a deterministic state-space tree, where each node represents a unique game state and each edge represents a possible action. By applying graph traversal algorithms, specifically a search method enhanced with bounding techniques, we can identify the optimal path from an initial state to a victory condition. This approach transforms the abstract challenge of strategic planning into a solvable pathfinding problem. The model was implemented in Java, utilizing simplified but representative character kits and game mechanics to ensure feasibility. Experimental results show that the model successfully generates a strategically coherent and efficient sequence of actions, effectively managing character synergies and resources in a way that mirrors expert-level human strategy.

Keywords—Turn-Based RPG, Pathfinding, State-Space Search, A* Algorithm, Optimization, Honkai: Star Rail

I. INTRODUCTION

The global market for game development is experiencing a period of unprecedented expansion, with both developers and players continually redefining the frontiers of interactive entertainment. Within this dynamic landscape, the turn-based Role-Playing Game (RPG) genre has solidified its position, captivating audiences not with high-speed reflexes, but with deep strategic complexity. Modern iterations of these games have evolved far beyond simple attack-and-defend mechanics, now featuring intricate systems where success hinges on foresight, resource management, and precise planning. The core challenge for a player is often not just deciding what to do, but determining the optimal sequence of actions, as the timing and order of operations can dramatically alter the outcome of a critical encounter.

This paper delves into this complex strategic layer by using the popular turn-based RPG *Honkai: Star Rail* as a comprehensive case study. Combat in this game is orchestrated via an action timeline that dictates when player characters and enemies take their turns, governed primarily by a "Speed" attribute. However, this timeline is not static; it is a fluid battlefield that can be manipulated

through character abilities that inflict delays, grant extra actions, or alter turn priority. This creates an exceptionally rich decision-making space. A single choice—for instance, whether to use a character's ultimate ability now for immediate damage or to save it for a more opportune moment after an ally has applied a defense-reducing debuff—can have cascading effects, making the identification of the most effective action sequence a non-trivial cognitive puzzle.

The central thesis of this paper is that this intricate problem of action sequencing can be rigorously modeled and solved by leveraging the principles of graph theory. We propose a computational model where the entire combat scenario is represented as a vast state-space graph. In this graph, each node is a unique state that encapsulates a complete snapshot of the battle at a single moment: the current turn order, the health and energy levels of all units, and the status of all active buffs and debuffs. An edge connecting one state to another represents a single action—such as a character's attack or an enemy's special move—that transitions the game from the preceding state to the next.

To solve the puzzle of optimization, we will utilize pathfinding algorithms, the cornerstone of navigating graph structures. By assigning a quantitative "weight" or "cost" to each edge—representing resources spent, damage taken, or, most simply, the passage of a single turn—the strategic goal of winning a battle efficiently is transformed into the computational problem of finding the shortest path from an initial state to a victory state. This study will therefore implement established shortest path algorithms to systematically explore the state-space graph and identify a sequence of actions that is demonstrably optimal under the defined cost metric. This deterministic and interpretable approach stands in contrast to more opaque methodologies like machine learning, offering clear, analyzable results that are valuable for both players and developers.

This paper, therefore, aims to develop and validate this

graph-based framework. The initial goal is to construct a robust computational model that can accurately translate the complex mechanics of Honkai: Star Rail's combat into a formal graph structure. Following this, the project will implement pathfinding algorithms to navigate this graph, with the objective of determining the optimal action sequences required to successfully and efficiently complete combat encounters. Ultimately, the broader ambition is to present this analytical framework as a powerful tool. For players, it can offer deeper strategic insights, while for developers, it provides a quantitative method for balancing character abilities, tuning encounter difficulty, and creating more intelligent NPC adversaries. This research contributes to the growing field of game analytics by bridging the gap between intuitive human strategy and the rigorous, data-driven optimizations.

II. THEORETICAL BASIS

A. Graph and Tree Theory

A graph is a fundamental structure in mathematics used to model relationships between objects. It consists of nodes (or vertices) and edges that connect them. For this paper, we will focus on a specific type of graph: a tree. A tree is a graph with no cycles, where any two vertices are connected by exactly one path. It is an ideal structure for representing hierarchical data or sequential decisions. A tree consists of:

- A Root Node: The starting point of the tree.
- Parent and Child Nodes: Each node (except the root) has exactly one parent, and can have multiple child nodes.
- Leaf Nodes: Nodes that have no children, representing endpoints.

In the context of this paper, the turn-based combat of *Honkai: Star Rail* is modeled as a state-space tree, which is a directed tree where nodes represent states and edges represent actions that transition between states.

- The root node of our tree represents the initial game state at the beginning of a battle. This state captures all relevant information, including character health, enemy health, turn order, and any active buffs or debuffs.
- From any given parent node (a game state), child nodes are generated for every possible action that can be taken. Each edge leading to a child node represents a specific action (e.g., Character A uses a skill, Enemy B attacks). This forms a branching structure where each path from the root represents a unique sequence of gameplay events.
- Edge Weights represent the cost of an action. For this analysis, the simplest cost is the progression

of the timeline. An action that advances the turn counter would have a specific weight, allowing the pathfinding algorithm to find the sequence of actions that leads to victory in the fewest turns.

- Leaf nodes represent terminal states of the combat, such as a victory state (all enemies defeated) or a defeat state (all player characters defeated).

By framing the problem this way, the strategic challenge is transformed into finding the optimal path from the root of the state-space tree to a victory leaf node.

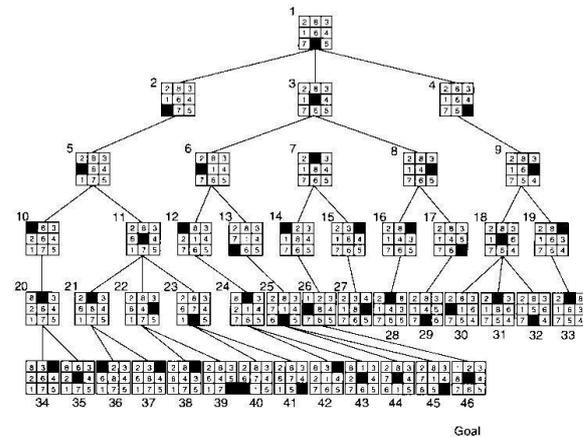


Figure 3.15 Breadth-first search of the 8-puzzle, showing order in which states were removed from open.

Figure 2.1 State Space Tree

<https://i.cs.hku.hk/~kpchan/cs23270/3.problem-solving/problem-solving.html>

B. Pathfinding Algorithm

Pathfinding algorithms are used to find the shortest or optimal route between two nodes in a graph. By modeling combat as a graph, we can use these algorithms to identify the most efficient sequence of actions to achieve victory. This paper will explore several foundational algorithms.

Uniform Cost Search is an algorithm that finds the least-cost path from a source node to a target node in a weighted graph. It is optimal because it always expands the node with the lowest path cost from the source. It can be thought of as Dijkstra's algorithm without a specific target, exploring paths in order of increasing cost. The UCS algorithm works as follows:

1. Initialize a priority queue (min-heap) and add the starting node with a cost of 0.
2. Initialize a set or map to keep track of visited nodes and their costs to avoid redundant processing.
3. While the priority queue is not empty:
 - Extract the node with the lowest cost from the queue. Let this be the current node.

- If the current node is the goal state, the algorithm terminates, and the path is returned.
- For each neighbor of the current node, calculate the new path cost by adding the edge weight to the current node's cost.
- If the neighbor has not been visited or the new cost is lower than its previously recorded cost, add the neighbor to the priority queue with its new cost.

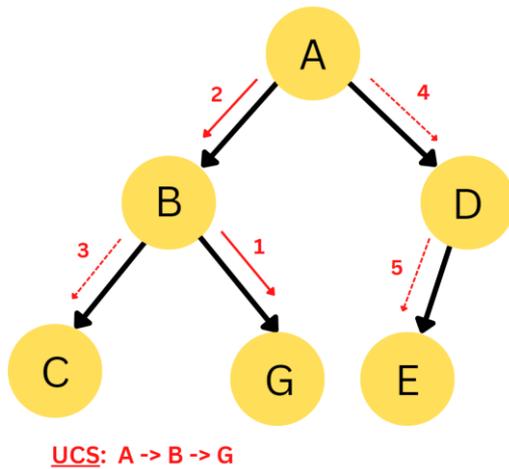


Figure 2.2 Uniform Cost Search

<https://www.naukri.com/code360/library/uninformed-search-algorithms-in-artificial-intelligence>

The next algorithm that will be looked at is A*, which is an extension of UCS that improves efficiency by using a heuristic function. This heuristic estimates the cost from the current node to the goal. A* balances the cost already traveled from the start ($g(n)$) with the estimated cost to the goal ($h(n)$). The algorithm prioritizes nodes with the lowest combined value, $f(n) = g(n) + h(n)$. The A* algorithm works as follows:

1. Initialize a priority queue with the starting node. Its cost is its heuristic estimate, $h(n)$.
2. While the priority queue is not empty:
 - Extract the node with the lowest $f(n)$ value. Let this be the current node.
 - If the current node is the goal state, the path is found.
 - For each neighbor of the current node:
 - Calculate the tentative cost from the start to this neighbor, $g(neighbor)$.

- If this path to the neighbor is better than any previous one, record it.
- Add the neighbor to the priority queue, prioritized by its $f(n)$ value.

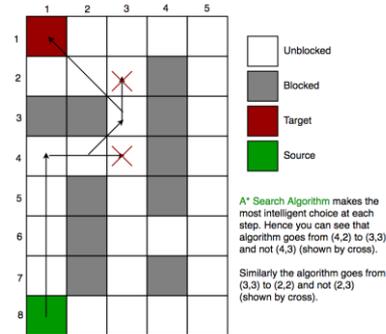


Figure 2.3 A* Algorithm

<https://www.geeksforgeeks.org/a-search-algorithm/>

Branch and Bound is an algorithm design paradigm for optimization problems. While often used for more complex problems, its principles apply here. It systematically explores the state-space graph by breaking the problem into smaller subproblems (branching) and discarding subproblems that are guaranteed not to contain an optimal solution (bounding). The algorithm can be adapted for shortest path finding as follows:

1. Start at the source node. Initialize the best_cost to infinity.
2. Explore a path, recursively or iteratively branching out to neighbors.
3. At each node, check validity with a function.
4. **Bound:** If the current path is deemed invalid, prune this branch.
5. If a path reaches the goal node, compare its total cost to best_cost. If it's lower, update best_cost.

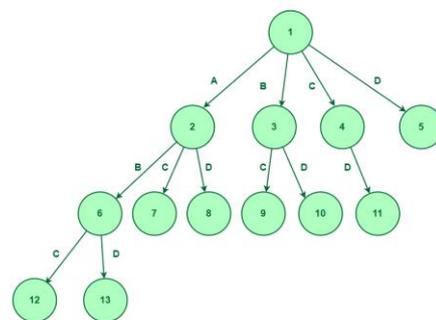


Figure 2.4 Branch and Bound

<https://www.geeksforgeeks.org/introduction-to-branch-and-bound-data-structures-and-algorithms-tutorial/>

These algorithms will be utilized to search the combat state-space graph. The starting node will be the initial state of a battle, and the goal will be a state where all enemies are defeated. By applying UCS, A*, or Branch and Bound, we can systematically discover the sequence of actions (the path) that minimizes the total cost, which is the action value, thereby finding the optimal way to win the encounter. Branches will be pruned when certain characters are unable to perform their most optimal move in a 'state', and the searching will follow UCS or A*.

C. Honkai: Star Rail Calculation

To accurately model the game state and the transitions between states, it is essential to define the core mechanics mathematically. The following sections detail the key formulas that govern combat in *Honkai: Star Rail*.

Action Value (AV) is the core mechanic determining turn order. It represents the "time" a unit must wait for its next turn. The unit with the lowest AV acts next. This will be the main deterministic value for the cost of a state.

- **Formula for Base AV:** A unit's Base AV is inversely proportional to its Speed (SPD). This value is the amount of AV they must recover after an action.

$$\text{Base AV} = 10000 / \text{SPD}$$

- **Turn Progression:** The game progresses by subtracting the AV of the current actor (who has 0 AV) from every unit on the field. This brings the next unit to 0 AV, allowing them to act. After a unit completes its turn, its AV is reset to its Base AV.
- **AV Modification:** Abilities can directly manipulate the AV, which is crucial for turn optimization.

- **Action Advance:** Reduces a unit's current AV, allowing them to act sooner.

$$\text{New AV} = \text{Current AV} * (1 - \text{Percentage Advance})$$

- **Speed Buff/Debuff:** Speed changes do not alter a unit's current AV but will affect their *next* turn by changing their Base AV calculation.

$$\text{Total SPD} = \text{Base SPD} * (1 + \% \text{ SPD Buffs}) + \text{Flat SPD Buffs}$$

The damage dealt by any ability is calculated through a series of multiplicative steps that factor in character stats, enemy defenses, and various buffs and debuffs. The general outgoing damage formula is:

$$\text{Final Damage} = \text{Base Damage} \times \text{Crit Multiplier} \times \text{Damage \% Multiplier} \times \text{DEF Multiplier} \times \text{RES Multiplier} \times \text{Vulnerability Multiplier} \times \text{Toughness Multiplier}$$

- **Base Damage:** The initial damage value, dependent on the character's core offensive stat (ATK, HP, or DEF) and the ability's scaling. $\text{Base Damage} = (\text{Skill Multiplier} \times \text{Scaling Stat}) + \text{Additional Flat Damage}$
- **Crit Multiplier:** If an attack critically hits, the damage is increased by the character's Crit DMG stat. $\text{Crit Multiplier} = 1 + \text{Crit DMG}$
- **Damage % Multiplier:** A sum of all general and specific damage bonuses (e.g., Elemental DMG Bonus, All-Type DMG Bonus). $\text{Damage \% Multiplier} = 1 + \text{Elemental DMG\%} + \text{All-Type DMG\%} + \text{Other specific DMG\% bonuses}$
- **Defense (DEF) Multiplier:** Represents the damage reduction from the enemy's defense. $\text{DEF Multiplier} = (\text{Character Level} + 20) / ((\text{Enemy Level} + 20) * (1 - \text{DEF Reduction}) + (\text{Character Level} + 20))$
- **Resistance (RES) Multiplier:** Represents the damage reduction from the enemy's resistance to the attack's element. $\text{RES Multiplier} = 1 - (\text{Enemy RES} - \text{RES PEN})$
- **Vulnerability Multiplier:** Represents bonus damage taken by an enemy from debuffs. $\text{Vulnerability Multiplier} = 1 + \text{Vulnerability Debuffs}$
- **Toughness Multiplier:** A flat damage reduction applied if the enemy's Toughness bar is not broken. $\text{Toughness Multiplier} = 0.9$ (if Toughness is present) or 1.0 (if Toughness is broken)

III. CODE IMPLEMENTATION

The theoretical model was implemented using the Java programming language, chosen for its strong object-oriented features and robust Collection framework, which provides efficient data structures like PriorityQueue for managing the search algorithm's open set. The primary goal of this implementation is not to create a perfect replica of Honkai: Star Rail, but rather to build a deterministic model that captures its core strategic mechanics—turn order, resource economy, and ability usage—to test the pathfinding hypothesis. Emulating the game's precise, moment-to-moment experience, with its countless variables and random elements, is both impractical and unnecessary for this paper's scope. Therefore, simplifications were made, such as standardizing the outcomes of random events like critical hits. While this may lead to slight deviations from actual

in-game results, the core logic remains intact, ensuring the findings on turn optimization are reliable.

The implementation is structured around several key classes representing the game's combat entities. A Character abstract class serves as the blueprint for all playable units, defining common attributes like HP, Speed, and Energy, along with methods for actions. Four concrete characters were implemented: Archer (a generic damage dealer), Sparkle (a support who manipulates skill points and turn order), Silver Wolf (a debuffer), and Gallagher (a healer). To streamline the problem, the Enemy is a passive "damage sponge" with a large health pool that does not perform actions, focusing the optimization challenge entirely on the player's actions. The most crucial component is the immutable State data structure, which represents a single node in the state-space tree. A State object contains a complete snapshot of the combat, including the status of all units, available skill points, and the total action value consumed to reach that point.

The pathfinding logic is encapsulated within a Solver class, which takes an initial State and explores the state-space tree to find a path to the victory condition: the enemy's health reaching zero. The solver generates the tree by creating a new child State for every possible action a character can take, with the cost of each transition calculated from the action value consumed. To manage the massive potential size of this tree, an effective bounding algorithm is hard-coded into the state expansion logic. This prunes branches that are guaranteed to be invalid or suboptimal. For example, if a character's best action requires a skill point but the current state has none (e.g., Sparkle needing to use her skill), the solver prunes that entire branch before creation. This prevents the algorithm from exploring long, inefficient paths that start with a suboptimal move, thereby significantly narrowing the search to the most promising routes.

```
package genericCharacter;

public abstract class Character {
    private final String name;
    private double currentEnergy;
    private final double maxEnergy;
    private final int skillPointCost;
    private final Skill skill;
    private final Ultimate ultimate;
    private final Stats baseStats;
    private Stats additionalStats;

    public Character(
        String name,
        Stats baseStats,
        double maxEnergy,
        int skillPointCost,
        Skill skill,
        Ultimate ultimate
    ) {
        this.baseStats = baseStats;
        this.additionalStats = baseStats;
        this.name = name;
        this.maxEnergy = maxEnergy;
        this.skillPointCost = skillPointCost;
        this.skill = skill;
        this.ultimate = ultimate;
    }

    public Character(
        String name,
        Stats baseStats,
        Stats additionalStats,
        double currentEnergy,
        double maxEnergy,
        int skillPointCost,
        Skill skill,
        Ultimate ultimate
    ) {
        this.name = name;
        this.baseStats = baseStats;
        this.additionalStats = additionalStats;
        this.currentEnergy = currentEnergy;
        this.maxEnergy = maxEnergy;
        this.skillPointCost = skillPointCost;
        this.skill = skill;
        this.ultimate = ultimate; // No ultimate ability in this constructor
    }

    public String getName() { return name; }
    public Stats getBaseStats() { return baseStats; }
    public Stats getAdditionalStats() { return additionalStats; }
    public Stats getCurrentStats() { return baseStats.plus(additionalStats); }
    public void setAdditionalStats(Stats additionalStats) { this.additionalStats = additionalStats; }
    public void addAdditionalStats(Stats additionalStats) { this.additionalStats = this.additionalStats.plus(additionalStats); }
    public int getSkillPointCost() { return skillPointCost; }
    public Skill getSkill() { return skill; }
    public Ultimate getUltimate() { return ultimate; }

    public double getMaxEnergy() { return maxEnergy; }
    public double getCurrentEnergy() { return currentEnergy; }
    public void setCurrentEnergy(double currentEnergy) {
        if (currentEnergy < 0) {
            this.currentEnergy = 0;
        } else if (currentEnergy > maxEnergy) {
            this.currentEnergy = maxEnergy;
        } else {
            this.currentEnergy = currentEnergy;
        }
    }

    public abstract Character copy();
}
```

Figure 3.1 Character Class Private Documentation

```

public class ActionQueue {
    private PriorityQueue<Action> queue;
    private double actionPassed;

    public ActionQueue() {
        this.queue = new PriorityQueue<>(
            Comparator.comparingDouble(Action::getRemainingAV)
        );
        this.actionPassed = 0;
    }

    public double getActionPassed() {
        return actionPassed;
    }

    public void setActionPassed(double actionPassed) {
        if (actionPassed < 0) {
            throw new IllegalArgumentException("Action passed cannot be negative");
        }
        this.actionPassed = actionPassed;
    }

    public ActionQueue copy() {
        ActionQueue newQueue = new ActionQueue();
        for (Action action : queue) {
            newQueue.addAction(new Action(action.getRemainingAV(), action.getCharacter().copy()));
        }
        newQueue.setActionPassed(this.actionPassed);
        return newQueue;
    }

    public void addAction(Action action) {
        if (action == null) {
            throw new IllegalArgumentException("utils.Action cannot be null");
        }
        queue.add(action);
    }

    private void refreshQueue() {
        if (queue.isEmpty()) {
            return;
        }
        Action firstAction = queue.poll();
        if (firstAction == null) {
            return;
        }
        double avCost = firstAction.getRemainingAV();
        setActionPassed(avCost + getActionPassed());

        PriorityQueue<Action> tempQueue = new PriorityQueue<>(
            Comparator.comparingDouble(Action::getRemainingAV)
        );
        tempQueue.add(new Action(0, firstAction.getCharacter()));
        while (!queue.isEmpty()) {
            Action nextAction = queue.poll();
            if (nextAction != null) {
                double updatedAV = max(0, nextAction.getRemainingAV() - avCost);
                tempQueue.add(new Action(updatedAV, nextAction.getCharacter()));
            }
        }
        queue = tempQueue;
    }
}

```

Figure 3.2 ActionQueue Class Private Documentation

```

package utils;

import enemy.Enemy;
import genericCharacter.Character;

public class DamageFormula {
    public static double getDamage(
        Character c,
        Enemy e,
        double baseDmg
    ) {
        return calculateDmg(
            baseDmg,
            c.getCurrentStats().getCritDmg(),
            c.getCurrentStats().getDmgBoost(),
            e.getDef(),
            e.getRes()
        );
    }

    private static double calculateDmg(
        double baseDmg,
        double critMult,
        double dmgMult,
        double def,
        double res
    ) {
        double critDmg = baseDmg * (1 + critMult);
        double defMult = 1 - (def / (def + 200 + 10 * 80)); // Assuming attacker level is 80
        double finalDmg = (critDmg * (1 + dmgMult) * defMult) * (1 - res);
        return Math.max(finalDmg, 0); // Ensure damage does not go below 0
    }
}

```

Figure 3.3 DamageFormula Class Private Documentation

```

package utils;

import charaImpl.archer.Archer;
import charaImpl.sparkle.Sparkle;
import enemy.Enemy;
import genericCharacter.Character;

import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;
import java.util.Objects;

public class State {
    private State prevState = null; // reference to the previous state
    private String action = null; // action taken to reach this state
    private Character[] characters;
    private Enemy enemy;
    private int skillPoints;
    private ActionQueue actionQueue;
    private List<State> nextStates;

    public State(Character[] characters, Enemy e, int skillPoints) {
        this.characters = characters;
        this.enemy = e;
        this.skillPoints = skillPoints;
        this.nextStates = new java.util.ArrayList<>();
    }

    public State copy(){
        State newState = new State(characters.clone(), enemy.copy(), skillPoints);
        newState.actionQueue = this.actionQueue.copy();
        newState.nextStates = new java.util.ArrayList<>(this.nextStates);
        return newState;
    }
}

```

Figure 3.4 State Class Private Documentation

```

public void fillNextStates() {
    Character currentCharacter = actionQueue.getNextCharacter();
    if (Objects.equals(currentCharacter.getName(), "Archer")) {
        List<Enemy> enemies = new ArrayList<>();
        enemies.add(enemy.copy());
        if (currentCharacter.getSkillPointCost() > getSkillPoints()) {
            System.out.println("Not enough skill points to execute Archer's skill. Pruning this branch.");
            return;
        }
        Character tempCharacter = currentCharacter.copy();
        Character basicCharacter = tempCharacter.copy();
        State basicAttackState = this.copy();
        basicCharacter.setCurrentEnergy(basicCharacter.getCurrentEnergy() + 20);
        basicAttackState.setSkillPoints(basicAttackState.getSkillPoints() + 1);
        basicAttackState.updateCharacter(basicCharacter);
        this.addNextState(basicAttackState, "Archer Basic Attack");
        int currentSkillPoints = getSkillPoints();
        while (currentSkillPoints >= currentCharacter.getSkillPointCost()){
            State newState = this.copy();
            enemies = Arrays.stream(new Enemy[]{newState.getEnemy().copy()}).toList();
            Character temp = tempCharacter.copy();
            temp.getSkill().execute(temp, enemies, newState);
            newState.setSkillPoints(newState.getSkillPoints() - temp.getSkillPointCost());
            currentSkillPoints -= temp.getSkillPointCost();
            temp.setCurrentEnergy(temp.getCurrentEnergy() + 30);
            newState.updateCharacter(temp);
            newState.updateEnemy(enemies);
            this.addNextState(newState, "Archer Skill");
            if (temp.getMaxEnergy() <= temp.getCurrentEnergy()) {
                State ultimateState = newState.copy();
                Character ultimateCharacter = temp.copy();
                ultimateCharacter.getUltimate().execute(
                    ultimateCharacter,
                    enemies,
                    ultimateState
                );
                ultimateCharacter.setCurrentEnergy(5);
                ultimateState.setSkillPoints(ultimateState.getSkillPoints() + 2);
                ultimateState.updateCharacter(ultimateCharacter);
                this.addNextState(ultimateState, "Archer Skill + Ultimate");
                tempCharacter = ultimateCharacter.copy();
            } else {
                tempCharacter = temp.copy();
            }
        }
        Archer.stack = 1;
    }
}

```

Figure 3.5 State Creation from Existing State Private Documentation

```

public class Solver {
    private PriorityQueue<State> queue;

    public Solver(){
        this.queue = new PriorityQueue<>((s1, s2) ->
        Double.compare(s1.getActionQueue().getActionPassed(), s2.getActionQueue().getActionPassed()));
    }

    public State solve(State initialState) {
        queue.clear();
        queue.add(initialState);

        while (!queue.isEmpty()) {
            State currentState = queue.poll();

            if (currentState.isTerminal()) {
                return currentState;
            }
            currentState.fillNextStates();
            List<State> nextStates = currentState.getNextStates();
            for (State nextState : nextStates) {
                queue.add(nextState);
            }
        }
        return null; // No solution found
    }

    public void printSolution(State solution) {
        if (solution == null) {
            System.out.println("No solution found.");
            return;
        }

        List<State> path = new ArrayList<>();
        State current = solution;
        while (current != null) {
            path.add(0, current); // Add to the beginning of the list to reverse the order
            current = current.getPrevState();
        }
        for (State state : path) {
            System.out.println("Action: " + state.getAction());
            System.out.println("Action Queue: " + state.getActionQueue());
            System.out.println("Skill Points: " + state.getSkillPoints());
            System.out.println("-----");
        }
        System.out.println("Action Value: " + solution.getActionQueue().getActionPassed());
    }
}

```

Figure 3.6 Base Solver Class
Private Documentation

IV. EXPERIMENTAL RESULTS

To evaluate the effectiveness of the developed model, a specific test case was designed to simulate a common strategic scenario in Honkai: Star Rail. The setup involved a four-person team tasked with defeating a single, high-health enemy. The initial conditions and character stats were configured programmatically to create a consistent starting point for the solver.

The team was composed of Archer (a primary damage dealer with a base speed of 98), Sparkle (a high-speed Harmony support with 161 speed, designed to manipulate the turn order), Silver Wolf (a very high-speed Nihility debuffer with 181 speed), and Gallagher (a fast healer/support with 171 speed). All characters were initialized with half of their maximum energy, allowing for the potential use of Ultimate abilities early in the sequence. The target was a single, passive Enemy with 1,000,000 HP, acting as a "damage sponge" to test the long-term efficiency of the generated action sequence. The primary objective for the solver was to find the sequence of actions that defeats this enemy in the minimum possible total action value. The state is setup with 3 initial skill points.

```

public class Main {
    public static void main(String[] args) {
        Stats baseStats = new Stats(900, 0.5, 98, 20, 1);
        Character archer = new Archer(baseStats, 220, 2);
        baseStats = new Stats(800, 0.5, 161, 20, 10);
        Character sparkle = new Sparkle(baseStats, 120, 1);
        baseStats = new Stats(900, 0.5, 181, 20, 10);
        Character silverWolf = new SilverWolf(baseStats, 100, 1);
        baseStats = new Stats(800, 0.5, 171, 20, 10);
        Character gallagher = new Gallagher(baseStats, 120, 1);

        archer.setCurrentEnergy(archer.getMaxEnergy()/2);
        sparkle.setCurrentEnergy(sparkle.getMaxEnergy()/2);
        silverWolf.setCurrentEnergy(silverWolf.getMaxEnergy()/2);
        gallagher.setCurrentEnergy(gallagher.getMaxEnergy()/2);

        Enemy enemy = new Enemy("EvilEnemy", 1000000, 1200);
        State initialState = new State(new Character[]{archer, sparkle, silverWolf}, enemy, 3);
        ActionQueue actionQueue = new ActionQueue();
        actionQueue.addAction(new Action(10000 / archer.getCurrentStats().getSpd(), archer));
        actionQueue.addAction(new Action(10000 / sparkle.getCurrentStats().getSpd(), sparkle));
        actionQueue.addAction(new Action(10000 / silverWolf.getCurrentStats().getSpd(), silverWolf));
        actionQueue.addAction(new Action(10000 / gallagher.getCurrentStats().getSpd(), gallagher));
        initialState.setActionQueue(actionQueue);

        Solver solver = new Solver();
        State optimalState = solver.solve(initialState);
        solver.printSolution(optimalState);
    }
}

```

Figure 4.1 Setting Up the Test State
Private Documentation

The test was initiated by creating an initial State object reflecting these parameters and passing it to the Solver class. The solver then executed its pathfinding algorithm, exploring the state-space tree to find the most efficient path to a victory state. The output generated by the solver is a step-by-step sequence of optimal actions. The output of the program is as follows:

```

Action: null
Action: Silver Wolf Skill
Action: Gallagher Basic Attack
Action: Sparkle Skill
Action: Archer Basic Attack
Action: Gallagher Basic Attack
Action: Sparkle Skill
Action: Archer Basic Attack
Action: Gallagher Basic Attack
Action: Sparkle Skill
Action: Archer Skill
Action Value: 93.16770186335404

```

Figure 4.2 Output
Private Documentation

The results demonstrate that the model successfully identifies a strategically coherent sequence of actions that aligns with established player heuristics. The algorithm correctly prioritized the setup actions, using the highest-speed character, Silver Wolf, to apply a crucial defense debuff before the primary damage dealer's turn. It also leveraged the synergistic relationship between Sparkle and Archer, using Sparkle's action-advancing ability to maximize damage output. Furthermore, the sequence shows intelligent resource management, with support characters using basic attacks to restore skill points when their primary abilities were not needed. The bounding mechanism was critical here; for instance, if Sparkle's

turn arrived with zero skill points, the solver would have pruned the invalid "use Skill" branch.

V. CONCLUDING REMARKS AND FUTURE WORK

In summary, this paper has successfully developed and demonstrated a computational model for optimizing action sequences in the turn-based RPG, Honkai: Star Rail. By representing the complex flow of combat as a deterministic state-space tree, we have translated an abstract strategic challenge into a solvable pathfinding problem. The application of graph search algorithms to this model proved effective, with the experimental results showcasing the system's ability to generate a strategically coherent sequence of actions that aligns with established player heuristics. The model intelligently managed character synergies and resource constraints to identify a path to victory, confirming that discrete mathematics provides a powerful framework for analyzing and solving complex gameplay puzzles.

Looking ahead, while this model serves as a successful proof of concept, there are several avenues for significant expansion and refinement. The most critical area for improvement lies in the optimization of the search algorithm itself. The current bounding method is effective but elementary; future work should focus on developing more sophisticated heuristics for an A* search implementation. A well-designed heuristic could drastically reduce the search space by more accurately estimating the "cost" to victory from any given state, leading to substantially faster computation times. Furthermore, the fidelity of the combat simulation could be greatly enhanced. The current model simplifies many game mechanics for feasibility. A more advanced version could incorporate the complexities of character gear (Relics), detailed Light Cone effects, and even probabilistic outcomes like critical hits, which would bring the model's optimal path even closer to true in-game performance.

It is also important to note that the scope of this project was constrained by a tight time limit. With additional time, the aforementioned algorithmic optimizations and simulation enhancements could have been explored more thoroughly, leading to a more robust and comprehensive analytical tool. Nonetheless, this paper lays a strong foundation, demonstrating the viability of using graph theory and pathfinding algorithms to deconstruct and optimize strategic gameplay.

The following is the link to the source code:

<https://github.com/pixelatedbus/hsr-turn-optimization>

The following is the link to the video (BONUS):

<https://youtu.be/nOiK34CwmyQ>

VII. ACKNOWLEDGMENT

The author wishes to express heartfelt gratitude to several parties for their support in the creation of this paper. First, gratitude is extended to God for His guidance throughout the process of learning and writing. The author also acknowledges the invaluable teachings and support of Mr. Rinaldi Munir, the lecturer of ITB's Strategi Algoritma IF2211 course, whose guidance greatly enriched the learning experience. Finally, the author thanks their family and friends for their unwavering support throughout the semester.

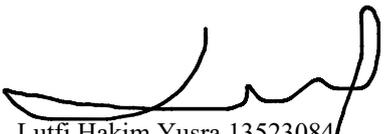
REFERENCES

- [1] P. E. Hart, N. J. Nilsson, and B. Raphael, "A Formal Basis for the Heuristic Determination of Minimum Cost Paths," *IEEE Transactions on Systems Science and Cybernetics*, vol. 4, no. 2, pp. 100–107, 1968.
- [2] E. W. Dijkstra, "A Note on Two Problems in Connexion with Graphs," *Numerische Mathematik*, vol. 1, no. 1, pp. 269–271, 1959.
- [3] M. A. Goodrich and R. Tamassia, *Data Structures and Algorithms in Java*, 6th ed. Hoboken, NJ, USA: Wiley, 2014.
- [4] Prydwen.gg, "Honkai: Star Rail Damage Formula," Prydwen Institute, 2024. [Online]. Available: <https://www.prydwen.gg/star-rail/guides/damage-formula>

PERNYATAAN

Dengan ini saya menyatakan bahwa makalah yang saya tulis ini adalah tulisan saya sendiri, bukan saduran, atau terjemahan dari makalah orang lain, dan bukan plagiasi.

Bandung, 24 Juni 2025



Lutfi Hakim Yusra 13523084