IMPROVING COMPANION AI IN SMALL-SCALE ATTRITION GAMES

by

Shuo Xu

School of Computer Science
McGill University, Montreal

October 2015

A THESIS SUBMITTED TO MCGILL UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF
MASTER OF SCIENCE

Abstract

Artificial Intelligence (AI) has been widely used in modern video games for creating interactive non-player characters (NPC) and opponents. Although the design of NPC enemy AI has been studied for years and has many commercial implementations such as StarCraft, World of Warcraft, a good AI for NPC companions is still under-analyzed. In this thesis we investigate several approaches for solving companion decision problems in small-scale attrition games that involve two teams competing to eliminate the other. Then by introducing an action oriented analytical model, we analyze specific combat choices and improve the existing greedy heuristics. Our experimental results show that the improved heuristics indeed achieve better performance under various combat scenarios.

Résumé

L'intelligence artificielle (IA) est largement utilisée dans les jeux vidéo afin de créer des personnages non-joueurs (PNJ) et des adversaires interactifs. Bien que la conception du PNJ ennemi IA a été étudiée pendant de nombreuses années et a maintes applications commerciales dans des jeux tels que StarCraft et World of Warcraft, un bon IA pour les PNJ compagnons reste encore sous-analysé. Dans cette thèse, nous examinons plusieurs approches pour résoudre les problèmes de décision de compagnon dans les jeux d'usure à petite échelle impliquant deux équipes qui s'affrontent pour éliminer la partie opposante. Ensuite, en introduisant un modèle analytique orienté vers l'action, nous analysons des choix de combat spécifiques et améliorons les heuristiques gloutonnes déjà existantes. Nos résultats expérimentaux démontrent que, effectivement, les heuristiques améliorées performent mieux dans quantité de situations de combat.

Acknowledgements

I would like to express my deepest appreciation to my supervisor, Professor Clark Verbrugge, who constantly encourages and helps me explore the AI area for modern computer games. He also instructs me to write this thesis in a clean, concise and formal way. Without his guidance this thesis would not have been completed.

I would also like to thank my fellow student, Jonathan Tremblay, who has helped me investigate several AI strategies including Rapidly Exploring Random Tree and Monte Carlo Search.

Finally, I would like to thank to my parents and friends, who continuously support me in pursing my M.Sc degree.

Table of Contents

1 Introduction 9

1.1 Contributions 11

1.2 Thesis Roadmap 11

2 Analytical Game Model 13

2.1 Introduction to the Game Model 14

2.2 Basic Elements 15

2.2.1 Players and Teams 15

2.2.2 Moves 17

2.3 Game Type 19

2.4 Agent Decision Process 20

2.5 Combat Configurations 21

2.5.1 Fixed Assumptions 22

2.5.2 Optional Assumptions 23

2.6 Combat Flow 23

2.7 Combat Result Evaluation 24

2.8 Experimental Model 27

2.8.1 The *Pokémon* Game and Combat 27

2.8.2 Experimental Setup 29

3 Search Approaches 33

3.1 Search Process 34

3.2 Search Tree 34

3.3 Brute-Force Approach 36
3.4 RRT Approach 40
3.5 Limitations and the Next Step 45

4 Heuristic Approach for Sleep Moves **46**
4.1 Sleep Analysis 47
4.1.1 Time-Damage Chart 47
4.1.2 Trade-off Analysis 50
4.2 Sleep Decisions 54
4.3 Experiment 55

5 Heuristic Approach for Heal Moves **62**
5.1 Heal Move Setting 62
5.2 Heal Analysis 63
5.3 Heal Decisions 65
5.4 Experiment 66

6 An Experiment with All Move Types **75**
6.1 Combined Smart Heuristic 75
6.2 Experimental Setup 76
6.3 Result 77

7 Related Work **83**
7.1 Analyzing Attrition Games 83
7.2 Tree Search Strategies 84
7.3 Other General AI Approaches 85

8 Conclusions and Future Work **86**
8.1 Conclusions 86
8.2 Future Work 87

List of Figures

2.1 A battle scene screenshot in Final Fantasy IV 14

3.1 A tree link connecting two nodes 35

3.2 Relation of K, average time cost, and the probability of a solution 44

4.1 Time-damage chart for c_1 against e_1 48

4.2 Time-damage chart for c_1, c_2, c_3 against the enemy team 48

4.3 Time-damage chart for c_1, c_2, c_3 against the enemy team with sleep involved 49

4.4 Time-damage chart for e_1, e_2, e_3 against the companion team with sleep involved 50

4.5 Total remaining health of companions by each sleep strategy 58

4.6 Averaged total number of sleep casts per combat of companions 59

4.7 Percentage win-rate for the companion team over all 1000 test cases by each sleep strategy 61

5.1 Total remaining health of companions by each healing strategy 68

5.2 Average total number of rounds spent in combat of "vs. 2 enemies" 69

5.3 Average number of healing casts 70

5.4 Total damage dealt by companions by each healing strategy 71

5.5 Average number of healing casts for vs. 8 enemies case 72

5.6 Percentage win-rate for companions over all 1000 test cases by each healing strategy 74

6.1 Percentage win-rate for companions by each combined strategy 78

6.2 Total remaining health of companions by each combined strategy 80
6.3 Average number of each move casts 81

List of Tables

2.1 List of companion pokemons 30
2.2 List of enemy pokemons 30

3.1 The running time of brute-force search for different player and move settings 38
3.2 Running time of a brute-force search for different player and move settings, with maximum sleep and heal move allowance PP_{max} increased to 5 39
3.3 Comparison of the RRT search approach and the brute-force search approach 43

Chapter 1

Introduction

Artificial Intelligence (AI) has been widely used for Non-Player Characters (NPC) in modern video games, especially for a combat or battle environment. We see a number of examples in Real-Time Strategy (RTS) games, Role-Playing games (RPG), as well as other popular genres where AI plays an important role. In *StarCraft* (an RTS game), the AI controls enemy agents to play against the human players in order to train human skills for humans to compete in the more challenging Player versus Player (PvP) game mode. In the *Pokémon* series (an RPG game) and *Counter Strike* (an FPS game) on the other hand, AI is implemented for both enemy agents and allies (companions) during combat, where companion agents are designed to help players win against enemies. Research on designing and optimizing AI for NPC enemies has gone quite deep nowadays and is recognized as capable of providing players an interesting and challenging game experience. However, the design of good AI for NPC companions who support human players is still under-analyzed.

A typical way for many commercial game companies to implement the AI for NPC companions in combat is to use a scripted *behavior tree* [Ogr12]. Game designers hard-code each behavior by observing how humans play, and then organize the sequence of behavior executions to produce the final NPC companion AI. This experience-based approach works well to a certain extent; without further analyzing how AI agents' behaviors are related to the actual game results theoretically, however, it becomes hard to maintain the good quality of such heuristics when we scale up the game and increase the complexity. Moreover, even for veteran players it is possible that their perception of "correct" actions to take may not be the optimal decision to help them win the game in the end.

In this thesis, we investigate the AI combat decisions for NPC companions in the context of small-scale *attrition games*—games in which two teams compete to eliminate the other. We first build an analytical game model that defines the game elements and combat flows that simulate attrition games. Inside the model we inject AI agents who decide on actions for one of the teams, giving us a problem space to represent combat decisions that need to be made. An experimental framework is then used to test our solutions and evaluate the theoretical result in more realistic game scenarios.

Our initial approach for solving decision problems is to apply search algorithms. Search algorithms have been heavily used in solving traditional games like Go, Chess, etc. For attrition games, player actions are more diverse in that an action may somehow change the target's combat status such as lowering the defense value, or preventing the target from attacking ("sleep" mode), as well as the basic "attack", "defend" and "movement" in space. This results in a large action branching factor so that even if search algorithms can solve attrition games of small size, they fail rather quickly in term of the time and memory cost as we add more action choices and players to the game. Heuristic algorithms on the other hand run very fast and are fairly easy to implement in general, and so are much preferred in interactive, real-time game environments. However, if the heuristic depends purely on a player's experience and feedback without further theoretical analysis, maintaining and properly designing an AI becomes difficult.

Based on investigations of the search approaches, we thus propose an action-specific enhancement for heuristic strategies. We focus on evaluating two specific move types, sleeping and healing, which can be often seen in many games including *League of Legends*, *Pokémon*, *World of Warcraft*, etc. They represent two different genres of actions: sleeping weakens the enemy by preventing its actions in future game turns, while healing conversely increases the chance of ally survival. By applying our game model, we analyze the benefit and cost of using each type of move under various combat scenarios and give suggestions on move decisions accordingly. We test our heuristics against some common strategies taken by modern games and discuss the performance and restrictions. Our goal in this work is to improve the quality of companion AI strategies in a realistic game environment.

1.1 Contributions

The contributions of this thesis include:

- **An analytical attrition-game model**: We define a game framework that describes key elements and the basic combat flows of small-scale attrition games. The framework supports AI players implementing a variety of strategies for making move decisions. It also allows configurations to be added as assumptions for creating and simulating different combat scenarios.

- **Search approaches**: We have proposed search methods using both brute-force and *RRT* game-tree growing techniques to solve companion decision problems. With experiments we have identified the corresponding the complexity and limitations of such search approaches in our context.

- **Move impact analyses**: Search approaches are limited in their ability to scale, and so we further investigate heuristic approaches, analyzing strategies for using two actions core to common game combats, the sleeping move and the healing move. We give a formal analysis of the benefit and cost for each move type and discuss how each move could impact the players and influence the game results.

- **Heuristics of using specific moves**: Based on the move analysis, we define heuristics for choosing move types and move targets wisely under different combat scenarios.

- **Experiments and evaluations**: To augment and validate our theoretical model, we perform experiments based on real game data. We explore the application of our sleep and healing heuristics on battles based on the *Pokémon* game and test the performance of enhanced heuristics. The result show improvements of AI strategies over other scripted greedy strategies made by modern game designers.

1.2 Thesis Roadmap

This thesis contains eight chapters in total, including this introductory chapter. In *Chapter 2* we build up analytical game model that defines key components, flows and an experimental model for attrition game combats. In *Chapter 3* we analyze the search oriented approaches for solving companion AI decision problems. In *Chapter 4* we focus on the sleeping move, investigating the move benefit and cost in combat while proposing heuristics on using it. Experiments are presented for performance comparisons with other strategies. In *Chapter 5* we follow a similar analysis on the healing action and define corresponding healing heuristics, again with experimental evaluation. In *Chapter 6*, we combine the heuristics developed in previous chapters for sleeping and healing together, and take an experiment to test the performance of the new strategy in combats allowing multiple actions. *Chapter 7* provides an overview of related work, while *Chapter 8* gives conclusions for the entire thesis and identifies possible future work.

Chapter 2

Analytical Game Model

Nowadays, thousands of titles have implemented AI-vs-AI battle gameplays. Some are of a pure combat type, including most real-time strategy (RTS) games such as *StarCraft*, and the recent, also quite popular multiplayer online battle arena (MOBA) games such as *League of Legends*. Others include typical role-playing games (RPG), allowing human players to build their team with non-player character (NPC) allies in order to win battles against NPC enemies or enemy teams. Examples can be found in the *Final Fantasy* series, the *Pokémon* series, etc. Variations among all these genres are huge in terms of the actual game experience. However, it is feasible to generalize and abstract the key components of the "combat" concept from real games.

In this chapter, we form an analytical game model that includes the most important factors in attrition game combat. After an introduction in Section 2.1, we start with giving definitions to the basic components, including players, moves, and teams in Sections 2.2 and 2.3. Then, within this game framework we define the decision process of NPC AI agents to simulate the thinking of real game players in Section 2.4. We add configurations to combat settings in Section 2.5 so that the broad, complex strategy problems can be divided into simpler sub-problems. Section 2.6 describes the overall game flow, and we discuss different types of game result evaluations in Section 2.7. Lastly, in Section 2.8 we describe the experimental model and specific environment settings on which we will test the AI strategies. Throughout the entire thesis we use the model developed in this chapter as a basis for our analyses and discussions.

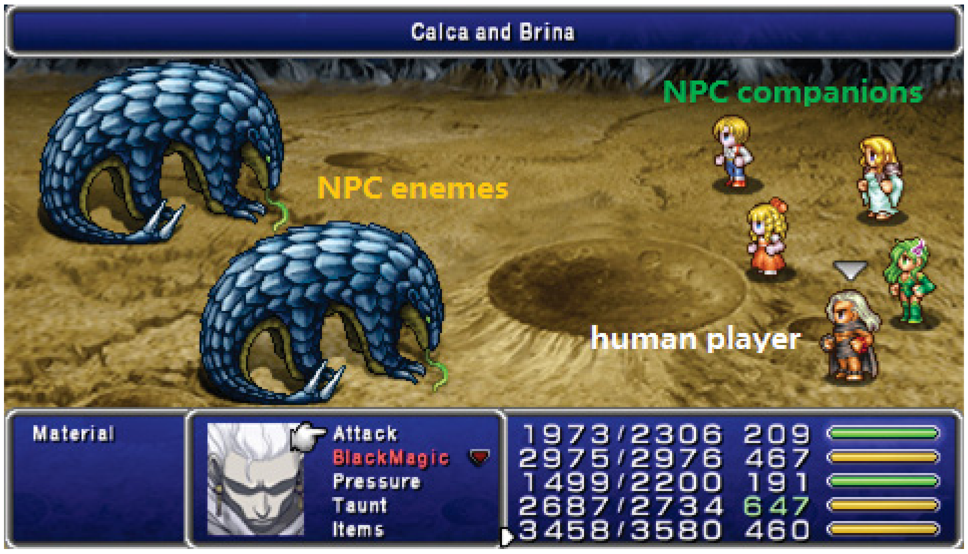


Figure 2.1: A battle scene screenshot in Final Fantasy IV

2.1 Introduction to the Game Model

The analytical model aims to describe and simulate real video games of different genres. It provides a framework to test, analyze, and evaluate behaviors and performance of AI agents who control the participating players in the game combat.

To motivate and give intuition to the model, we first we look at a combat example taken from *Final Fantasy IV*, as shown in Figure 2.1. We observe two teams which are trying to eliminate the other. The player highlighted in white on the right-hand side is controlled by a human while his companions and enemies are controlled by AI programs. Each player has a list of combat moves or actions to choose from (note that "move" and "action" will be used interchangeably in rest of this thesis), shown in the center of the bottom blue bar, and a health bar indicating their current status, shown on the right of the bottom bar.

Therefore, our model starts with the basic elements, **players** and **moves**. Each player has a set of moves they are allowed to use. Players are assumed to form teams (a team needs at least 1 player) and each team has a game goal to be set. A game goal could be eliminating all the other teams, or staying alive in the combat for as much time as possible, depending on the combat scenarios we simulate. Each combat scenario may require certain configurations of the model, including assumptions about each team's strength, limitations on the move set or use, etc. The choice of configurations impacts generality of our model; we will discuss configuration issues in more depth in Section 2.5.

Then after we have defined all the elements and settings, the model initiates the combat. Teams fight against each other by their players using moves on specific targets, until eventually one team wins the game—at least one player in the team is alive and all other teams are eliminated. In the course of players using moves, there is a "brain" behind each player telling them which move should be used, and how the move should be used (on which target). Typically in real games, this **decision process** is either done by a human being (who literally controls a particular player) or by programmed AI agents. In our thesis, we focus on improving the companion AI and how the companion agents pick correct strategies in combat, and so human actions are generally not considered, although the possibility of complex human-agent cooperation does exist and could be an interesting direction for future work.

When the combat ends, the model evaluates the game result, such as in terms of how much damage each team has done, etc. Evaluation criteria will be discussed in Section 2.7. Based on the evaluation, we will then be able to analyze the behavior and performance of our AI agents and make an effort to improve them.

2.2 Basic Elements

In this section, we provide specific detail on the basic model elements, players, teams and moves. We define each element individually and clarify their relations to each other.

2.2.1 Players and Teams

Players are entities involving in the combat. A player p is defined as a 6-tuple:

$$p = \langle h_{max}, h, a, \textit{state, debuffTime}, M \rangle$$

- $p.h_{max} \in \mathbb{N}^+$ is a positive integer marking the initial and maximum health value setting of p

- $p.h \in \mathbb{Z}$ is an integer representing the real-time health of p. When p gets attacked or healed, this real-time health is decreased or increased, but may not increase above maximum health: $p.h \leq p.h_{max}$.
- $p.a \in \mathbb{N}^0$ is a non-negative integer representing the attack power of p. In general this is the amount of damage p can inflict on an opponent in a single attack move.
- $p.state \in \{$*Healthy, Dead, InSleep, ...*$\}$ is a discrete value marking the state of p. When p is alive with $p.h > 0$, its state is simply $p.state =$ *Healthy*. When p is killed and $p.h \leq 0$, then $p.state =$ *Dead*. Note that if an action deals a special effect on p, $p.state$ will be altered to that specific effect, such as sleeping ($p.state =$ *InSleep*). The set of states is extensible.
- $p.$*debuffTime* $\in \mathbb{N}^0$ is the remaining time of a harmful effect existing on player p. For example, in a turn-based combat when someone uses *sleep* to cause p to sleep ($p.state =$ *InSleep*) and $p.$*debuffTime* $= 2$, this means p will wake up in 2 turns. For players in *Healthy* state, $p.$*debuffTime* is always zero. Note that a player can only have at most one "debuff" (harmful effect) at any given point. Also note that we use a non-negative integer value, as we will focus on turn-based games, but in general this could be of a float type to model continuous time in real-time games.
- $p.M$ is the set of move types available for p. Moves and move types will be discussed in the next section.

A **team** is composed of one or many players. In this thesis, we set our model to allow exactly **2 teams** in the combat. This is the case in most attrition games such as the *Pokémon*, the *Final Fantasy*, the *League of Legends* where fights occur between the player team and the enemy team. For combat involving more than two teams, our analyses may not be applied and the model would require further extensions such as to include multidimensional ally-enemy relationship.

As mentioned in the previous section, the actual human player is not considered for companion AI strategy analyses in our model. The human's team, however, does define the protagonists versus the antagonists, giving us two teams, a Companion Team C (presumed to include the human player) and an Enemy Team E. We label the players in the companion team as $c_1, c_2, c_3, \ldots$, and the players in the enemy team as e_1, $e_2, e_3, \ldots$ If C has n players, and E has m players, we will have:

- Companion team C: $\{c_1, c_2, c_3, ..., c_n\}$
- Enemy team E: $\{e_1, e_2, e_3, ..., e_m\}$

2.2.2 Moves

Moves are actions allowed to be used by players during the game. Each move instance **m** is a 3-tuple:

$$m = \langle \textit{user, target, type} \rangle$$

.

- ***m.user*** is the player who uses the move m. For companions, *m.user* $\in C$, and for enemies, *m.user* $\in E$.
- ***m.target*** is the player (target) who is to be affected by the move m. It could be in either companion team or enemy team depending on the *type* of move.
- ***m.type*** is a discrete value defining the genre (or name) of the specific action. In the simplest attack-defense combat games the basic *type* is *attack*. When an *attack* move is used, the user deals damage to the target and directly reduces the target's health. In modern video games, there are moves that are more complex than the simple attacking one. In *World of Warcraft* combat, for example, players may use "state-modifying" moves such as a *sleep* move, or a *freeze* move to put the target into a state such that that the target's future moves are restricted. In League of Legends, player can also use a *heal* move in order to heal a wounded ally player whose current health is low. In *Pokémon* combat there also exists so-called "attack-modifying" moves to double or half the attack strength of a player for a certain range of time.

  During our research, we will include the following three representative move types:

  $$m.\textit{type} \in \{\textit{attack}, \textit{sleep}, \textit{heal}\}$$

In addition, we define a *noaction* move type in the model for situations where a player runs out of move choices, because of all the opponents being dead for example. The *noaction* type should be used only when no other moves are available. This makes the entire set of move types in our research be:

$$m.\mathit{type} \in \{\mathit{attack}, \mathit{sleep}, \mathit{heal}, \mathit{noaction\ (restricted)}\}$$

The type category is also extensible.

Execute the move

When a player uses a move on a target, we calculate health h, attack power a and other move-specific attributes to update the game state. Below we show the execution of the three move types we use in this thesis (*noaction* type is trivial):

- ***attack*** move execution
```
$move.target.h \leftarrow move.target.h - move.user.a$
if $move.target.h \leq 0$, then
    $move.target.h \leftarrow 0$
    $move.target.state \leftarrow Dead$
```

- ***sleep*** move execution
```
if $move.target.state \neq Dead$ and $move.target.state \neq InSleep$, then
    $move.target.state \leftarrow InSleep$
    $move.target.debuffTime \leftarrow SLEEP\_DURATION$
```

- ***heal*** move execution
```
if $move.target.state \neq Dead$
    $move.target.h \leftarrow move.target.h + HEAL\_AMOUNT$
    if $move.target.h > move.target.h_{max}$, then
        $move.target.h \leftarrow move.target.h_{max}$
```

Here *SLEEP_DURATION* and *HEAL_AMOUNT* are globally defined integers representing the effective time for a *sleep* move and the portion of health increased by a *heal* move. Their values are discussed in the experimental setup in Section 2.8.

The **cost** of a move is another important factor in modern games. When a player makes a move, it is usually accompanied with consumption of certain resources to restrict the number of move cast within a designed time frame. This is particularly important for powerful moves, such as *sleep* and *heal*. There are many implementations of the cost, and most games have some form of "mana" resource and consumption. In our model, we will be using a move cast counter "PP" taken from the *Pokémon* games, and introduced in Section 2.8.

2.3 Game Type

Strategy games are widely known to be divided into two types, turn-based and real-time. "The key measure of how games proceed is the manner in which time is concerned" [Sha13]. Turn-based games make players execute their actions as a strict chain of events, following certain defined rules and turn orders. Real-time games on the other hand are more close to real life battles, in which actions from both sides are taken asynchronously, and the player executes its move as long as the move is ready. Real-time games still process player moves discretely, and typically include a limited range of "Cool-Down" (CD) values that constrain how frequently an attack can be performed, and thus can in principle be reduced to turn-based at some granularity. In our research, we thus consider only turn-based combat games. A turn-based combat game is split into **rounds**, where in each round each team is processed, allowing every team member to execute a move. In most games the player side (companion team) moves first, in order to give the player an advantage. In our model, for example if we have two companions c_1, c_2 in the companion team and two enemies e_1, e_2 in the enemy team, then a possible game flow would be:

Round 1: c_1 *attack* e_2
c_2 *attack* e_2
e_1 *heal* e_2
e_2 *attack* c_1
Round 2: . . .

2.4 Agent Decision Process

As mentioned in Section 2.1, in our model we assume players are controlled by AI agents ("brains"). The agent makes decisions each turn on how the player chooses the best move type and on which target the move is to be used, under fast-changing combat situations. As the main challenge of this entire thesis, the decision making process of AI agents is presented in two steps, shown below. Note, however, that these steps are not entirely independent, and while we present and analyze them separately, there is overlap. In this we suppose a player c_1 is to make a move, labelled $c_1.\mathit{nextMove}$, and

$$c_1.\mathit{nextMove} = \langle c_1, \mathit{target}, \mathit{type} \rangle$$

Step 1: Choose the move type

As player c_1 has multiple move choices available,

$$c_1.\mathit{nextMove}.\mathit{type} \in \{\mathit{attack}, \mathit{heal}, \mathit{sleep}, \mathit{noaction}\}$$

the choice of move comes down to a benefit-trade-off comparison among the set of possible moves. Generally, the move that can maximize the player's profit towards its team's goal is preferred, such as to maximize the team's total health left so that there is more chance for the team to stay alive.

There are several possible approaches to pick the best choice. One approach could be to generate all possible game states by trying each move and build a search tree, then using search algorithms to back-propagate the best solution. Such an approach has been used widely in solving board games like Chess. Alternatively, we could make a greedy approach by evaluating each move's benefit individually based on the current game state, designing a proper benefit formula, and then choosing the one with highest benefit value.

Either approach has challenges. The first has heavy CPU and memory requirements in creating and evaluating the entire decision tree (with all possible moves) for current and future rounds, while the second is somewhat ambitious in that the proper benefit formula may be hard to come up with. The analyses on these approaches will be discussed in detail in the following chapters.

Step 2: Choose the move target

Choosing the proper target is known as the "targeting problem" [TDV14] in the game area. In our case it involves assigning a specific enemy to the *target* field of our move object:

$$c_1.\mathit{nextMove}.\mathit{target} \in \{e_1, e_2, e_3, ...\}$$

When a companion uses a move on an enemy, the selection of the correct enemy based on the enemy's attributes (attack power a, health h, etc.) may influence the eventual game result, and is thus a non-trivial decision. Similar to Step 1, in this step either search methods or a greedy method could be applied to find the best target of the move to maximize the profit for the current player or team. Tremblay *et al.* introduced a nice greedy approach in the context of combat with only the *attack* move type: choose the target that has the highest *threat* [TDV14], with threat positively related to enemy attack, and inversely related to enemy health. In our thesis we extend the targeting problem to consider more complicated moves including the *sleep* and the *heal* types. To provide perfect solutions for such problem may be hard, yet the aim is to give insight into trade-offs of different targeting decisions for handling various complex combat scenarios.

Note that in neither step does the AI agent know what actions players on the opposing team will do.

2.5 Combat Configurations

Before the game starts, we impose another list of settings and assumption options on the model. The purpose is to map each combat configuration to a real specific situation, such as the beginner phase of most games where enemies are designed to be relatively weak. Additionally, these assumptions divide the broad decision strategy problems into a set of sub-problems with lower complexity, making it feasible to analyze. However, by doing this we inevitably lose some generality in our analyses and conclusions. This drawback will be discussed specifically after each assumption description in this section.

2.5.1 Fixed Assumptions

The following fixed assumptions will be applied to all the combat cases in this thesis.

[1] ***The companion team C always moves first***:
Moving first is an obvious advantage for the companion team in turn-based game setups. The impact of this assumpption is enlarged as the length of the combat becomes shorter, and in special cases where in a single round either the companion team or enemy team could eliminate the other, this assumption directly decides on a combat winner. In our analysis we try to keep the combat duration relatively long (at least four to five rounds) to avoid the influence on game balance as much as possible.

[2] ***Moves should be used only on targets that are alive (move.target.state != Dead)***:
Generally in attrition games, dead player units are immediately removed from the combat. The setting of using moves on only living targets also implies that AI agents for both companions and enemies should make real-time decisions within a round depending on the changing situation. For example, if c_2 moves after c_1 in a certain round and c_1 kills enemy e_1 in that round, c_2 will be aware of the e_1's death and will not attack e_1 any more. The drawback of this assumption is trivial, as the existence of "wasted" moves attacking an enemy already killed is easy to avoid in turn-based contexts, and commonly disallowed in commercial games.

[3] ***Players within one team move in fixed order***:
If the companion team is built as $\{c_1, c_2, c_3\}$, then within one round c_1 always moves first, then c_2 moves and c_3 moves last (same for enemies). This assumption is to simplify the game flow. However, the impact is that it implies no order variation and any further analyses and conclusions based on this cannot be applied to combat that either has randomization on the move order or uses attributes such as *speed* or *initiative* to define the move order specifically.

[4] ***Enemy AI uses deterministic heuristics***:
Most modern games implement the NPC enemy AI in a deterministic fashion so that the enemies follow scripted strategies to make move choices each time. This allows a

player to learn best strategies, treating combat as another kind of game puzzle which they can eventually master. In other, more sophisticated games enemies may themselves adapt to companion behaviors and improve their strategies at runtime. From an automated companion's perspective, however, while changes in enemy strategies could influence the actual combat result, the choice of combat move is assumed to depend on the current game state, irrespective of enemy evolution.

2.5.2 Optional Assumptions

We also consider optional assumptions in specific cases. These can heavily affect the combat simulation, and potentially causing greater loss of generality.

[1] ***The number of move types is restricted***:
When the number of move types is restricted, analyses of trade-offs among the different types are simpler, but do not necessarily generalize to situations in which more move types are available. We consider restricted move situations in order to manage the complexity of game analysis, although these situations are also interesting in being representative of partial stages of an entire combat, and/or as part of a specific level setup such as the tutorial combat.

[2] ***Predicting and assuming enemy strategies***:
Knowing enemy next moves makes it possible for companion agent to build a look-up game tree so that future moves and states of combat are searchable, and a possible solution might be found. Although applying this assumption may influence the decision quality in practice due to the accuracy of enemy behavior prediction, it allows evaluations on various search-based approaches that are less dependent on specific move definitions, especially useful for games whose elements and logic are complex.

2.6 Combat Flow

With all basic elements (players, teams, moves) and configurations ready, we can now incorporate combat simulation into the model.

Our combat begins with initialization of players in both two teams. For each player $c \in C$ and $e \in E$,

$$c.h \leftarrow c.h_{max}, \quad e.h \leftarrow e.h_{max}$$

As the companion team moves first, each $c \in C$ decides on its next move (*type* and *target*) independently, in a fixed sequence, which we assume follows player indexing order. After every companion completes its move, the turn switches to the enemy side, and each enemy AI agent decides and executes a move. After all enemy moves are complete, the round ends. The model then verifies if the game is over in that either all companions or all enemies have been killed. If so, the combat ends and we may evaluate the result. If not, the game proceeds to the next round, and continues the previous procedures.

The pseudocode in Algorithm 1 on page 25 shows the entire combat flow process algorithmically. The *AIAgentC* and *AIAgentE* are agents making move decisions for companions and enemies respectively (referring to Section 2.4). Function *ExecuteMove()* updates a player's health and states as described in Section 2.2.2, while *UpdateDebuffTime()* decreases the player's debuff remaining time by 1 each round if it is in an unhealthy state like *InSleep*.

2.7 Combat Result Evaluation

After the combat ends, we could evaluate the result (referring to *EvaluateCombatResult()* in Algorithm 1) in different ways. Most games use the attrition evaluation, where winning is defined by surviving—the only outcome is either "win", "lose", or "tie" and the strategy that leads to the highest probability of winning is preferred. In our model, this means we are optimizing the move decision strategy towards the likelihood of companions successfully eliminating all enemies. However, some games evaluate the combat result by also considering how much total health the player team has left at the end, as this can be an advantage in subsequent combats if health is not automatically restored, or has a resource cost to do so. In this case, even if the possible outcomes are all "win"s, we are looking for the win that has the highest sum of all companion's health (as healthy as possible). We could also have special cases of evaluation such as when the game has a primary player $p_{important}$

Algorithm 1 Combat Flow

```
procedure StartCombat
    Initialize team $C$, team $E$, AIAgentC, AIAgentE
    round $\leftarrow 1$
    while not IsGameOver() do
        for $c$ in $C$ do
            if c.state $=$ Healthy then
                move $\leftarrow$ GetNextMove(AIAgentC, $c$)
                ExecuteMove($move$)
            UpdateDebuffTime($c$)
        for $e$ in $E$ do
            if e.state $=$ Healthy then
                move $\leftarrow$ GetNextMove(AIAgentE, $e$)
                ExecuteMove($move$)
            UpdateDebuffTime($e$)
        round $\leftarrow$ round $+ 1$
    EvaluateCombatResult()
function GetNextMove(AIAgent, player)
    Initialize nextMove
    nextMove.user $\leftarrow$ player
    nextMove.type $\leftarrow$ AIAgent.GetNextMoveType(player)
    nextMove.target $\leftarrow$ AIAgent.GetNextMoveTarget(nextMove.type)
    return nextMove
```

whose health and alive status is paramount, while the state of other players is negligible. Modern RPG games with companion teams typically strive to have players emotionally invest in their companions, and so we do not model the latter, and assume the companion team members are equally important.

Games can also have combat situations where it is impossible to eliminate the enemy completely. For example, in "boss fights" of *World of Warcraft* the enemy leader may not be supposed to be killed. In these situations the goal is to do as much damage as possible before players die or within some time period. Since any time period can be modeled by a fixed damage rate to players, we thus also consider a decision strategy that aims to produce the highest damage before players expire.

In our thesis, the choice of combat result evaluation depends on the specific problem we try to solve. We summarize the basic types of combat result evaluation below:

- **P_CWIN - *Probability of combat winning***
  We will repeat the combat simulation for a number of K times and count the number of times companion team wins as C_{win}. Then we calculate $\frac{C_{win}}{K} * 100$ to obtain a percentage win-rate as the result.

- **SUM_H - *Sum of companion health***
  Given companion team $C = \{c_1, c_2, c_3, ..., c_n\}$, we measure the sum of the remaining health of each living companion after combat terminates as $\sum_{i=1}^{n} c_i.h$. Note that this includes all combats, whether or not the companions win.

- **SUM_D - *Sum of damage dealt to enemies***
  Given enemy team $E = \{e_1, e_2, e_3, ..., e_m\}$, we measure the sum of damage companions have dealt as $\sum_{j=1}^{m} e_j.h_{max} - \sum_{j=1}^{m} e_j.h$. This evaluation uses the indirect measure of looking at remaining enemy health, and so assumes no enemy healing. This is primarily aimed at evaluating simulations where enemies are unbeatable.

2.8 Experimental Model

To test and verify the result of our analyses, we would like to take a real commercial game and form an experimental model. Many games can be appropriate, such as *Pokémon* or *Final Fantasy*, which have the advantages of simplicity and well-supported online database resources for detailed game information. In our work, we take the *Pokémon* game and its basic combat setup to perform our experiments. We will use moves, players (with attribute settings) and part of the scripted AI strategies defined in *Pokémon*. First, however, we give an introduction to the *Pokémon* game.

2.8.1 The *Pokémon* Game and Combat

The *Pokémon* game is a role playing game (RPG) consisting of turn-based combat between different classes of creatures—"pokemons" (pocket monsters). The human player encounters wild pokemons while exploring a virtual world, and collects them while increasing their skills and skill ranking (level) in various pokemon combats, until finally he or she is able to challenge the *Pokémon League* champion with a last pokemon combat.

Definition of "pokemon"

During the combat, the player controls his or her pokemons (as the Player Team) to compete against the game NPC's pokemons (the Enemy Team). The combat involves 1 to 3 pokemons on each team at the same time. The game includes a total of 721 different pokemons as of the year 2015, with each pokemon having a set of base integer attributes

$$\langle \textit{ hp, attack, defense, sp.atk, sp.def, speed } \rangle$$

and a set of possible moves pre-defined. Here *hp* stands for health, namely h_{max}. The attributes *sp.atk* and *sp.def* stand for special attack and defense. Together with *defense* and *speed*, they are used for extra combat damage calculation specific to the *Pokémon* game and will not be considered in our core game model.

A pokemon example would be "Bulbasaur," indexed 001 in the pokemon database list [pok] with base attributes:

$$\langle \textit{hp (45), attack (49), defense (49), sp.atk (65), sp.def (65), speed (45)} \rangle$$

and available moves:

$$\{\text{Growl } (\textit{decrease attack}), \text{ Vine Whip } (\textit{attack}), \text{ Synthesis } (\textit{sleep}), ...\}$$

Note that moves are usually variations of the same type with different names. All the definitions and data can be found at the pokemon wiki website [pok].

Moves

The *Pokémon* game has a total number of 621 moves. Besides the three basic moves (*attack, heal, sleep*) we have set in our model, the game also includes "state-modifying" and "attack-modifying" moves mentioned in Section 2.2.2 to increase the fun of gameplay. Moreover, the three moves *attack, sleep* and *heal* in *Pokémon* are implemented in a slightly different way:

- ***attack*** move has its own "power" attribute for the game to use a more complex damage calculation formula during combat. This influences the design of attribute values for each player, although the basic idea of an attack action resulting in a loss to enemy health does not change.
- ***sleep*** move inflicts *InSleep* state for a random duration of 2 to 4 rounds.
- ***heal*** move heals half of the ally target's maximum health.

On the other hand, the cost of moves in *Pokémon* is implemented in the so-called "Power Points (**PP**)" mechanism. Each type of move has a defined constant PP_{max} (varying from 5 to 40) indicating the maximum number of move casts the player can use within one combat. Healing types of moves usually have $\text{PP}_{max} = 15$ while sleeping types of moves in general have $\text{PP}_{max} = 10$. Normal attacking moves have $\text{PP}_{max} = 40$, although the player seldom runs out of *attack* moves before the game terminates. Initially, the PP value (remaining move casts) is set to PP_{max} for every move. Whenever a move type is used, its PP is decreased by 1. For example, if in Round 1 c_1 uses *sleep* on e_1, then remaining casts of *sleep* for c_1 is 9 out of 10.

Combat

The combat of *Pokémon* is similar to our game flow, except that combat order respects the *speed* attribute: the pokemon (player) with the highest *speed* value moves first, and the rest move in order of decreasing *speed*. We will not include this setting in our experiments, following the assumptions made in Section 2.5.1 [1] and [3].

AI Behaviors

The AI that controls NPC enemy behavior in *Pokémon* combat follows a fixed decision procedure. By considering each player's items (not in our model), moves allowed, and the opponent states and attributes, the AI picks the "best" action choice for the NPC enemy for the current round [ess]. Although the exact action selection process is not officially released by Nintendo (the developer of *Pokémon*), many clues of the move choosing criteria can be found in player communities [bul]. We list some below.

- An enemy only uses healing moves (if available) for its allies whose health is below 25% of the maximum *hp*.
- An enemy uses state-modifying (such as sleeping, paralyzing) moves at the beginning with a probability of 70%.

In general, the *Pokémon* game uses custom scripted heuristics to implement intelligence for the enemies. The NPC companion AI of *Pokémon* follows similar decision rules as the NPC enemy AI.

2.8.2 Experimental Setup

In this subsection, we adapt the *Pokémon* game to our lightweight analytical model by mapping the elements we already have. We then assign real values from the *Pokémon* game to the variables in our framework and define the environment for testing AI strategies. Although we do not support all the features in the original game such as *sp.atk, sp.def* and *speed*-driven moving order, the *Pokémon* game is still a good benchmark for testing basic elements and combat strategies with a reliable source of data, as well as in providing a realistic sense of how the various move parameters should be scaled for a good simulation.

Player Setup

The players are pokemons whose data comes from the *Pokémon* database. A typical team setup example is shown in Table 2.1 (companions) and Table 2.2 (enemies), in which pokemons are selected randomly from Index 001 to Index 721. For simplicity we use this setup as a *default* team composition, which is subject to changes by varying the team sizes, pokemon choices, etc., according to specific experimental cases.

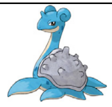
Companion Choices	c_1	c_2	c_3
pokemon name	Lapras	Chandelure	Gardevoir
pokemon index	131	609	282
pokemon icon			
base health h_{max}	130	60	65
base attack a	27	45	21
moves	{*attack, sleep*}	{*attack*}	{*attack, sleep, heal*}

Table 2.1: List of companion pokemons

Enemy Choices	e_1	e_2	e_3
pokemon name	Vaporeon	Mightyena	Gengar
pokemon index	134	262	094
pokemon icon			
base health h_{max}	130	70	60
base attack a	21	27	18
moves	{*attack, heal*}	{*attack*}	{*attack, sleep*}

Table 2.2: List of enemy pokemons

Move Setup

In our experiment, the default settings for the three moves (*attack, sleep, heal*) are as follows:

- ***attack*** deals damage equal to the user's attack power, the same as in Section 2.2.2.
- ***sleep*** causes the opponent in sleep for *SLEEP_DURATION* rounds. *SLEEP_DURATION* is a variable with default value 3.
- ***sleep*** heals the target by *HEAL_AMOUNT* health. *HEAL_AMOUNT* is a variable with default value $target.h_{max}/2$.

The move cost setup will follow the same "PP" cost implementation as mentioned in the *Pokémon* game in the previous subsection.

Decision Strategies Setup

We apply a non-evolving deterministic intelligence for our enemies, as much the same as the original game as we can determine from online sources. By not interfering with the enemy AI performance too much, we would like to simulate the NPC enemy behaviors as close to the real *Pokémon* game as possible. Thus we choose some of the tactics (again not officially confirmed) summarized by human players [bul] that appear similar to the actual game experience. The implicit assumption of using deterministic AI of NPC enemies may refer to Section 2.5.1 [4]. Note that the companions are by default blind to the strategies the enemies are using, regardless of its determinism.

Below we specify the details of our *EnemyAgent* in experiment:

1. ***EnemyAgent.GetNextMoveType()*** (move choice decision):
   return *sleep* in the first round with a probability of 70%
   return *heal* if one ally's health is below 25%
   return *attack* otherwise

2. ***EnemyAgent.GetNextMoveTarget()*** (targeting decision):
   return random target on using *sleep*
   return ally with the lowest health h on using *heal*
   return opponent with the lowest health h on using *attack*

Meanwhile for *CompanionAgent*, we test several decision strategies discussed in the rest of the thesis. Knowing the exact strategies of companions in *Pokémon* game would be impossible, and thus we will also compare our strategies to some commonly-used companion tactics applied in general attrition games [TDV14].

Evaluation Setup

We will use the combat evaluations described in Section 2.7.

Software and Hardware Setup

Our test programs are written in Java SE(1.7.0) and run inside Windows 7 64-bit operating system with Intel(R) Core(TM) i5-2500K CPU @ 3.50GHz and 8 GB of memory. This environment setting remains the same for all experiments in the thesis.

Chapter 3

Search Approaches

In some commercial games, players may have foreknowledge of how the enemies would act in combat, or are more or less able to predict enemy behaviors. Examples can be seen in the game *World of Warcraft* where players quickly learn the enemy AI either by practicing or through online forum discussions. In our experimental model *Pokémon* there are also many observable routine strategies, as partially listed in Section 2.8. For such games, knowing the enemy behaviors does not imply the players could win the game easily, however, due to the complexity of numerous move types, combat rules, team sizes, etc. A proper decision making to select appropriate moves is still critical for players in order to win or achieve a better score.

In this chapter, we propose tree search methods to solve the move decision problems for companions in the games where enemy behaviors are predictable. In Section 3.1 a brief description of the search process is presented. In Section 3.2 we define the basic components of our search tree. In Section 3.3 we show a brute-force method that grows the search tree completely, together with experiments to test its time-cost in real game scenarios. Then in Section 3.4 we propose another, more efficient search method based on the *Rapidly Exploring Random Tree* (RRT) approach that grows the search tree selectively and wisely. We then present experiments and compare RRT with the brute-force method in terms of both time complexity and search precision. Finally we summarize the limitations of search approaches in Section 3.5 including the loss of generality by assuming predictable enemy behaviors.

3.1 Search Process

A search-based approach to move selection involves constructing and traversing a data structure representing the possible game states. Before both teams make any move in combat, the companion AI agent builds a *game tree* whose root represents the combat start state. The tree then grows and adds future game states generated by possible player moves. The growing process could be implemented in several ways (as discussed in Section 3.3 and Section 3.4) and finishes when all the leaf nodes are game end-states (with one team winning) or the search process reaches a time limit. The AI agent then finds the best end node in the produced tree using one of the combat result evaluations (see Section 2.7) and back-propagates the move path to construct a series of moves from the root to the best end node. The moves in the path are eventually picked by companions accordingly in each turn during the combat.

Different from making a move decision in each round, the AI agent pre-produces the solution in advance, a strategy we can employ because we have assumed enemies have deterministic behaviors, and so the actual combat flow is guaranteed to follow the generated move path. Note that building and searching a game tree itself does not necessarily require enemy behaviors to be deterministic or predictable. The game-tree searching can be extended to non-deterministic behaviors with minimax [BW84b] or alpha-beta [BW84a] algorithms. In terms of understanding whether search may be a feasible approach, however, assuming behaviors of enemies is sufficient to give us a sense of baseline performance.

3.2 Search Tree

In this section we define the basic components of the search tree that represents the move choices and states of the combat.

Node

A tree node is a game state that contains information of all the players at a given point of combat. In our model:

$$node = \langle \{c_1, c_2, ..., c_n\}, \{e_1, e_2, ..., e_m\} \rangle$$

The node keeps track of the players' updated attributes such as current health *h*, current debuff time *debuffTime*, etc. The **root** node is defined as the start of the combat where every player is in its initial state.

Link

The link that connects two nodes will be a set of moves made by all players in a single round. This means each level of the tree represents the boundary of exactly one round. For example in Figure 3.1, we have shown one possible link between two game states. Since

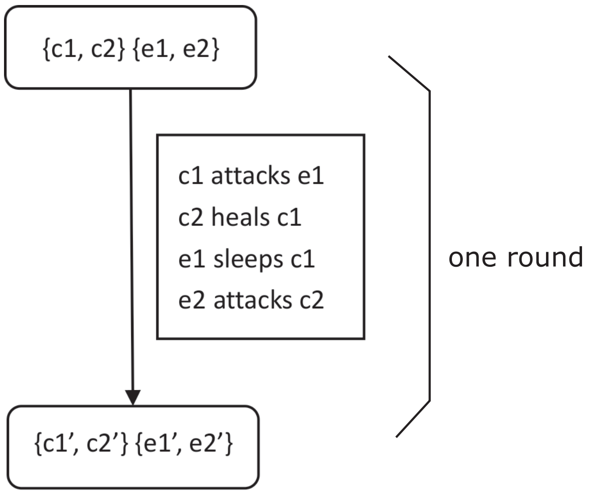


Figure 3.1: A tree link connecting two nodes

each companion could have multiple action choices, we will end up with a large set of such move combinations. On the other hand, the enemy move choices are deterministic (or predicted) as the main assumption of this chapter (may also refer to assumption 2.5.1[4]). We discuss situations where enemy moves are unpredictable and unknown in Section 3.5.

The cardinality of the move combination set defines the branching factor of the entire game tree. The more move or target choices we have, the more links and possible children can be generated.

Note that both companion moves and enemy moves are aggregated into one link. This greatly reduces the tree size over a more naive tree where each edge represents just a single agent move, and is possible given a fixed combat sequence and deterministic behavior of the enemy.

3.3 Brute-Force Approach

Rooted at the combat start, the search tree can be grown in various ways. We begin by attempting a brute-force approach that grows the tree by executing every possible move combination (*link*) of the companions and generating the subsequent children nodes with deterministic enemy moves. We recursively repeat this process on the child nodes, round by round, until we have finally reached the end of game at every leaf. With all the leaf nodes stored in memory, we find the best leaf and back-propagate the move decision path throughout the entire combat from it. The pseudocode is shown in Algorithm 2 in the next page.

The advantage of this approach is that given knowledge of enemy behaviors, we can search the entire space of game states and guarantee the optimal solution by foreseeing all possible future game states. This works regardless of how each individual move is defined. However, by considering every possible move combination choice, we introduce a risk that the tree branching factor will become too large, resulting in unacceptable time complexity as we increase the number of moves allowed.

Time Performance

To determine time performance, we compute the running time of the search process measured in milliseconds, including generating the tree, finding the best node, and returning the move decision. In modern video games, the NPCs are often required to deliver a fast response to output so that human players can have a good, responsive and interactive game experience. This means the AI thinking process needs to be unnoticeable (within fractions of a second)—Churchill et al. [CSB12] suggest a single decision frame be less than 50ms in order for the human player to not notice game delays. Note that we expect that during the combat the companion AI will be able to fetch the move decision from the searched so-

Algorithm 2 Brute-force Move Decision Search

```
function GENERATEBESTMOVEPATH()                                    ▷ Entry point
    root ← initial game state
    root.leaves ← new List<Node>
    BRUTEFORCEGROWTREE(root)
    bestEndNode ← FINDBEST(root.leaves)
    movePath ← BACKPROPAGATEPATH(bestEndNode, root)
    return movePath
function BRUTEFORCEGROWTREE(root)
    if ISENDOFGAME(root) then root.leaves.Add(root) return
    for node in GETCHILDREN(root) do
        node.prev ← root
        root.children.Add(node)
        BRUTEFORCEGROWTREE(node)
function GETCHILDREN(node)
    links ← Find all move combinations of the companion team
    for link in links do
        for $e$ in node.enemies do
            incorporate enemy e's next move (given or predicted) into the link state
    children ← map each link in links to node and create child nodes
    return children
function FINDBEST(nodeList)
    switch Evaluation_Method          ▷ refer to the last two evaluations in Section 2.7
    case SUM_H return node with highest companions' health in nodeList
    case SUM_D return node with lowest remaining enemies' health in nodeList
function BACKPROPAGATEPATH(node,root)
    $i \leftarrow 0$, path ← new List<Move>[]
    while node.prev $\neq$ root do
        path[i] ← node.link.moves
        node ← node.prev
        $i \leftarrow i + 1$
    return path.reverse()
```

lution for each companion more or less instantly; our main interest is in the time required to grow the game tree and run the search at beginning of combat, which can be much larger than 50ms, but must still be less than a second or so in order to not overly delay the start of combat.

Experiment 1: Brute-force Performance

In order to determine whether the brute-force search method has a reasonable scalability and time cost for the size of problems we need to consider, the following experiment is made. We use the default experimental setup (including teams) defined in Section 2.8.2. Additionally, we manipulate the following variables:

- Number of move types allowed for each player changes.
- Move cost (restriction) is implemented as maximum cast count PP$_{max}$ (see Section 2.8.1). Initially, PP$_{max}$ is set to a small number **2** for both *sleep* and *heal*. We will increase this number as the experiment moves on.

To select the best node we use **SUM_H** evaluation (highest companions total health score), as defined in Section 2.7. The result is shown in Table 3.1.

Allowed move set	$\{c_1\}$ vs. $\{e_1\}$	$\{c_1, c_2\}$ vs. $\{e_1, e_2\}$	$\{c_1, c_2, c_3\}$ vs. $\{e_1, e_2, e_3\}$
{*attack*}	2 ms	4 ms	72 ms
{*attack, sleep*}	2 ms	33 ms	1483 ms
{*attack, heal*}	2 ms	25 ms	588 ms
{*attack, sleep, heal*}	4 ms	1871 ms	∞ (out of heap)

Table 3.1: The running time of brute-force search for different player and move settings

From Table 3.1 we observe that in the trivial 1 vs 1 case, the running time required for brute-force search approach does not vary too much for different move restrictions. When the number of players in combat gets increased, however, the execution time grows dramatically and soon becomes far beyond a tolerable AI response time (say within a second). We can also note that the running time is also proportional to the number of move types allowed, with some variation due to move interactions. In the 3 vs 3 case, for example, "*attack+sleep*" takes more time than "*attack+heal*", because *sleep* potentially reduces *attack* moves for both companions and enemies and thus increases the number of rounds needed to terminate the combat while *heal* would only affect companions' *attack* moves. Further impacting time is the excessive memory cost—in the last row where all the three moves are allowed, we are not even able to compute a solution due to lack of memory.

Next, we verify how the move cost setting could affect the searching time. The initial PP$_{max}$ was 2 for both *sleep* and *heal* move types, which is fairly low. By increasing it to **5** for both, we allow each of *heal* and *sleep* to be used 3 more times in one game, based on the length of combat being around 4 to 5 rounds on average in the previous test.

Move set	$\{c_1\}$ vs. $\{e_1\}$	$\{c_1, c_3\}$ vs. $\{e_1, e_3\}$	$\{c_1, c_2, c_3\}$ vs. $\{e_1, e_2, e_3\}$
{*attack*}	2 ms	4 ms	72 ms
{*attack, sleep*}	2 ms	34 ms	5978 ms
{*attack, heal*}	2 ms	808 ms	∞ (out of heap)
{*attack, sleep, heal*}	4 ms	4714 ms	∞ (out of heap)

Table 3.2: Running time of a brute-force search for different player and move settings, with maximum sleep and heal move allowance PP$_{max}$ increased to 5

As we can see in Table 3.2, if we allow more casts for using *sleep* and *heal* move, the running time rises. The *heal* move for example, may be used again and again, allowing a companion to stay alive in combat longer, and increasing the number of rounds to finish the game. This leads to a larger height of the search tree and an exponential growth in the number of possible children nodes. The time for "*attack+sleep*" does not increase as much as "*attack+heal*", though, due to the restriction of *sleep* move itself that it can be only used on non-sleeping targets.

A 3 vs 3 scenario is a reasonable, but still small size for combat. A multi-second delay in combat start for this scale of problem is thus a concern, and in combination with the out of heap errors that occur in both experiments indicates that a brute-force method does not scale very well, and would require multiple order-of-magnitude improvements in time and memory to be useful. When it comes to more complex games with a larger set of move types and more players involved, the AI would need unacceptable time and space resources to search the game space and make a decision. Although our experimentation here is quite limited and many improvements are possible, it strongly suggests that an exhaustive, brute-force strategy is not viable in a real-time environment.

3.4 RRT Approach

As we are looking for a faster and more efficient algorithm to grow the search tree wisely, we investigate a heuristic tree search strategy known for its rapid execution. In 1998, Steven M. LaValle suggested a new search algorithm, called Rapidly Exploring Random Tree (RRT) [Lav98] for solving path planning problems. This approach was originally designed to find the best path from a current position to a goal position in a search space. The idea is to construct a search tree by randomly sampling points in the search space. When a sample is picked, the algorithm finds its nearest node in the current tree. If the connection between these two nodes is feasible, the sample node is added to the tree. The process repeats and the tree grows until a goal is found, or until it reaches the preset maximum number of iterations K.

We modify his approach to adapt to our game search tree. As mentioned previously, each tree node contains the current health for each companion and enemy. We set the search space to be the nodes whose health value is between 0 and the corresponding max health value h_{max} for each companion and enemy. The tree begins with adding the root node. Then we randomly sample a node, labeled N_{sample}, with possible health values for every c_i and e_j (as a potential game state). We search for all current nodes in the tree and find the "closest" node, labeled N_{close}, to the sample node using a distance function D that looks at the difference between health values of corresponding players:

$$D(\mathit{node1}, \mathit{node2}) = \sum_{i=1}^{n} |\mathit{node1}.c_i.h - \mathit{node2}.c_i.h| + \sum_{i=1}^{m} |\mathit{node1}.e_i.h - \mathit{node2}.e_i.h|$$

For example,

$$D((c_1.h = 10, e_1.h = 3), (c_1.h = 5, e_1.h = 1)) = |10 - 5| + |3 - 1| = 7$$

Even if it is closest to our sample node, it is not necessarily true that an action from state N_{close} can take us to state N_{sample}. For example, if health of a player in N_{sample} is higher than in N_{close}, but all heals have already been used, it is not possible to connect these nodes. Thus we then search all possible children of N_{close} to find a reachable state that gets as close as possible to N_{sample}. Once we find the child with the shortest distance to the sample node N_{sample}, labeled N_{new}, we add it to the search tree (connect it to N_{close}) if N_{new} is not already in the tree. Because a child is created by certain actions from the parent node, we ensure the feasibility of this newly added node N_{new}. The tree keeps growing until it reaches the maximum iteration count K. The pseudocode is shown in Algorithm 3 in the next page.

The nice property of this approach is that the tree is constructed rapidly and the game space is explored fast. By selecting the best explored node, however, one may obtain a sub-optimal move decision for the companion. When the game becomes complex (for example, with more move types or players added), the actual game space grows, and with a constant iteration limit K the chance to find an acceptable game end node (N_{new}) in the generated tree drops quickly. Without any explored node representing a combat finish, the move decision obtained is considered local (foreseeing only few rounds ahead) and is less useful. We are thus interested in the time improvement RRT provides over brute-force, but also in relative quality of solution found.

The following experiment helps us test the time complexity and performance of the RRT approach compared to the brute-force one. We here use the subset of experimental settings which brute-force was able to solve, focusing on the more interesting case when all 3 of our move kinds are allowed.

Experiment 2: RRT vs Brute-force

Setup:

- 1 vs 1 and 2 vs 2 combat cases included (players defined in Section 2.8.2)
- All the three moves (*attack, sleep, heal*) are allowed.
- PP$_{max}$ is set to 2 for *sleep* and *heal*
- Node selection uses SUM_H evaluation

Algorithm 3 RRT Move Decision Search

```
function GenerateBestMovePath()                              ▷ Entry point
    root ← current game state
    RRTGrowTree(root)
    bestNode ← FindBest(root.getLeaves())
    movePath ← BackPropagatePath(bestNode, root)      ▷ Same as Algorithm 2
    return movePath
function RRTGrowTree(root)
    $k \leftarrow 0$
    while $k < K$ do
        $N_{sample}$ ← SampleNode()
        $N_{close}$ ← FindClosestNode(root.allnodes, $N_{sample}$)
        childrenset ← GetChildren($N_{close}$)
        $N_{new}$ ← FindClosestNode(childrenset, $N_{sample}$)
        $N_{new}$.prev ← $N_{close}$
        $N_{close}$.children.Add($N_{new}$)
        $k \leftarrow k + 1$
function FindClosestNode(pool, target)
    min ← Integer.Max
    $N_{closest}$ ← pool[0]
    for node in pool do
        distance ←D(node, target)                      ▷ Distance function D
        if distance < min then
            $N_{closest}$ ← node
            min ← distance
    return $N_{closest}$
```

We run both the brute-force search and the RRT search at the beginning of the combat. The brute-force search is executed only once as it is deterministic. The RRT search is executed 100 times (repeated from the initial state each time) and we take the averaged combat result calculated using SUM_H evaluation. Note that since the RRT search does not guarantee to find a game end node because of the random sampling, in the cases where a game end node is not found, we consider the result to be 0. The test data is shown in Table 3.3, with standard deviations for RRT data less than 5%.

Combat with $\{c_1\}$ vs. $\{e_1\}$	Brute-force	RRT ($K = 5000$)
Running time in milliseconds	4	408 averaged
Combat best result in SUM_H	130	129 averaged

Combat with $\{c_1, c_2\}$ vs. $\{e_1, e_2\}$	Brute-force	RRT ($K = 5000$)
Running time in milliseconds	1871	409 averaged
Combat best result in SUM_H	195	186 averaged

Table 3.3: Comparison of the RRT search approach and the brute-force search approach

Table 3.3 shows that in 2 vs 2 combat, the RRT approach with K set to 5000 can search for a solution much faster than the brute-force approach, with little sacrifice to the game result (186 health on average compared to 195 health). In the simpler 1 vs 1 case, the RRT comes even closer in terms of solution quality, but retains the higher runtime of the larger situation due to the constant K numbers of iterations. (Note that even in the 1 vs 1 case, the entire search space can have over 10000 nodes in our test, depending on the players' health and attack.)

We also note from the these results that an increase in the game size, while keeping the boundary K unchanged, correlates with the RRT search producing a worse combat result (186/195 as opposed to 129/130). This is expected, as there is decreasing chance of finding a game end in RRT (and we consider the result as 0 in this case) as the search space increases but our maximum RRT tree size stays constant. We verify this trend further in the next experiment where the iteration boundary K is modified and we investigate the relationship between the running time, K and the probability of finding a game end node.

Experiment 3: Relation of K, Running Time, and Reachability of RRT
In this experiment the game size increases to 3 vs 3 by adding c_3 and e_3. We test on different values of K, and observe the probability of getting at least one game end node. Again for each value of K, we run the search for 100 times and take the averaged result. The result is shown in Figure 3.2.

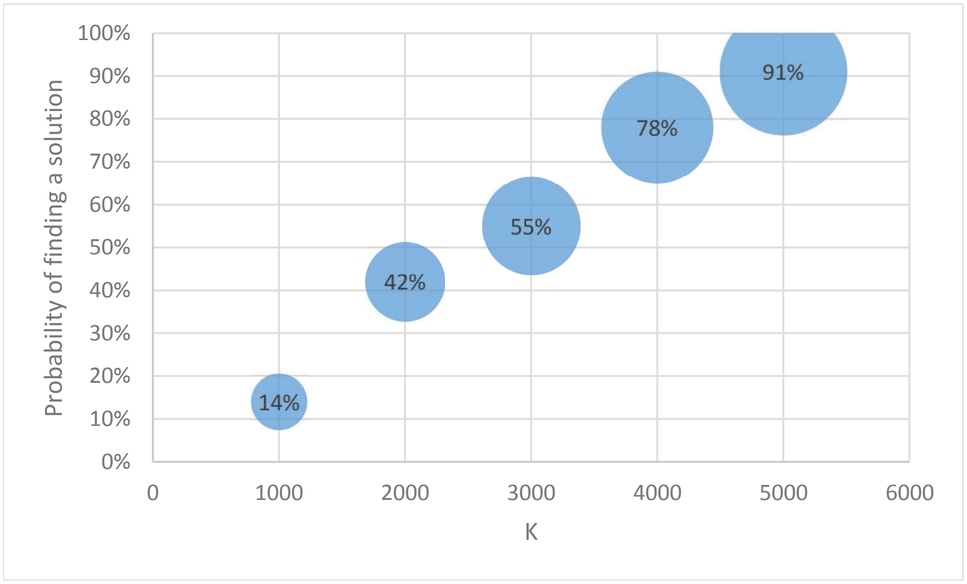


Figure 3.2: Relation of K, average time cost, and the probability of a solution

The diameter of the blue spots represents the averaged execution time in each scenario. The smallest one with $K = 1000$ represents a time of 80ms while the largest one with $K = 5000$ takes 413ms. On the y-axis we mark the probability of having a game end node explored in each case, computed as the ratio of successful searches in 100 search runs. We can see that although the time required for the search to complete reduces to an acceptable 80ms with $K = 1000$, we have a very low (14%) chance of finding a suitable game end node. When the search space gets even bigger as we add more players, this probability will be further reduced. The RRT approach is thus a promising replacement for the brute-force method in being able to solve larger problems, but as game size increases we are faced with the difficulty of trying to maintain the quality of the searched result while still keeping the running time low enough.

3.5 Limitations and the Next Step

Given the knowledge of the enemy AI, the brute-force and the RRT search approaches could provide decent move solutions to the companion in small-scale combat. Nevertheless both of them have limitations on the scalability. When the combat contains a large number of players and move choices, even our limited experiments show that these approaches will struggle to reach a solution in a reasonable time frame due to the fast growth of the tree size.

The assumption that enemy strategies can be predicted at the beginning of this chapter further restricts the use of the search methods. When the enemy decisions are unknown, generating the tree link is meaningless because the enemies may not behave deterministically with respect to certain companion moves. To fix this, a possible solution is to use a minimax method assuming that each enemy uses the move maximizing its profit each turn and thus makes its behavior partially predictable again. This approach has been profitably applied to many turn-based combinatorial games, such as Chess, but still faces scalability concerns, especially in real-time game combat situations.

In the rest of the thesis, we focus on a more analytical approach, considering each move type individually. This analysis is aimed at determining an AI strategy that heuristically makes decisions based on the current game and move information, is able to produce good results, and does not require an expensive search process or foreknowledge of enemy movements.

Chapter 4

Heuristic Approach for Sleep Moves

Heuristic oriented approaches focus on evaluating the move impact on the current combat situation, providing move decisions that may not be optimal, but help in gaining sufficient benefit for players. By calculating the benefit and cost of selecting each move type and target, the AI agent outputs a solution that guarantees minimum advantages over other choices. Different from search approaches, the heuristic strategies do not explore future game states and are supposed to run fast so that they should be practical for AI agents (with less thinking time) in real game scenarios and consoles with limited CPU power. We attempt to tackle the problem by investigating each move type individually.

In this chapter, the *sleep* move is discussed. Based on the game model in Chapter 2, the two major decision tasks for the AI agent are to choose which move type the companion should use, and to find the best target for the move. To simplify the problem at the first stage we restrict the move types to have only the *attack* and *sleep* moves, with the assumption of limiting the number of moves specified in Section 2.5.2.

Recall that the basic effect of *sleep* is to put the target into the *InSleep* state, preventing it from making any moves for *SLEEP_DURATION* numbers of round. The cost of *sleep* is based on the PP implementation (referring to Section 2.8.1) so that one player can only use *sleep* a maximum of PP_{max} times. We set $\text{PP}_{max} = 10$ by default for *sleep* as the numerical setting in *Pokémon* game.

In Section 4.1 we analyze the trade-off between *sleep* and *attack* moves, based on *time-damage charts* which link the benefit and cost of moves with the actual damage in combat.

In Section 4.2 discussions and suggestions on move type and move target selection are provided. In Section 4.3 we undertake experiments to examine our analyses in real combat scenarios inspired by the *Pokémon* games.

4.1 Sleep Analysis

In this section, we formalize the problem of using *sleep* by giving a trade-off analysis. First, a *time-damage chart* is introduced to illustrate and motivate the benefit and cost of different move types in terms of the **damage** dealt by players. We use this to develop equations for the trade-off calculation, and discuss different scenarios case by case.

4.1.1 Time-Damage Chart

The time-damage chart plots a (discrete) curve for each player p, $p.\mathit{damage} : \mathbb{N}^{+} \rightarrow \mathbb{N}^{0}$ mapping rounds to total damage dealt during the entire combat. The x-axis of the chart indicates the round number of the combat and the y-axis is the total damage the player has dealt so far. When different types of moves get involved and used, the chart could change with respect to the specific move type.

Figure 4.1 illustrates a basic example of the time-diagram chart for companion c_1 in a 1 vs 1 combat against e_1 (c_1 and e_1 refer to definitions in Section 2.8.2). In this simple example only the *attack* move is used by both players. The chart tells that c_1's *attack* is dealing 27 (equal to $c_1.a$) damage each round and kills e_1 in Round 5 (so e_1 has maximum health in [109–135]).

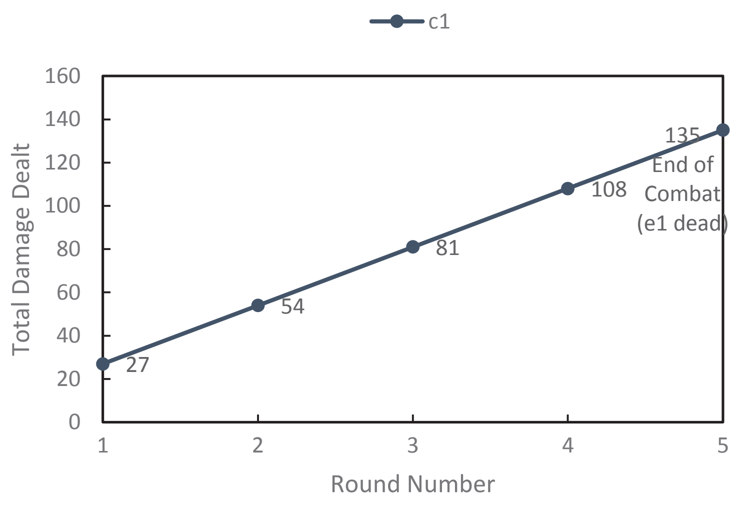


Figure 4.1: Time-damage chart for c_1 against e_1

Now if there are more players in one team, the chart will have several lines representing the damage of each player. Figure 4.2 shows the time-damage chart of one possible 3 vs 3 combat scenario for c_1, c_2, c_3 against e_1, e_2, e_3. Again only *attack* is used and targets are randomly picked for demonstration purposes.

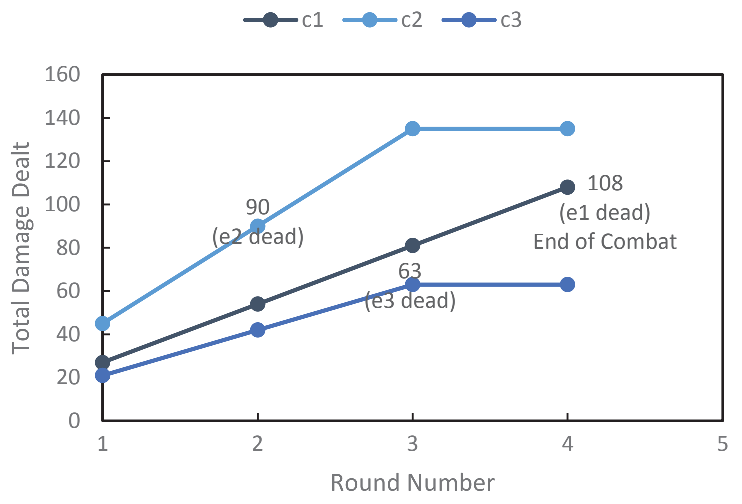


Figure 4.2: Time-damage chart for c_1, c_2, c_3 against the enemy team

Based on the chart, we can calculate the total damage the companion team has dealt so far at any given point of the combat by adding up the y-axis values of each line. For example, in Round 3, the total team damage is:

$$\sum_i c_i.\mathit{damage} = 81 + 135 + 63 = 279$$

Next, the *sleep* move is added and used in the combat. Suppose *SLEEP_DURATION* $=$ 2 and companion c_1 uses *sleep* on enemy e_1 in Round 2, the change to the chart is shown in Figure 4.3.

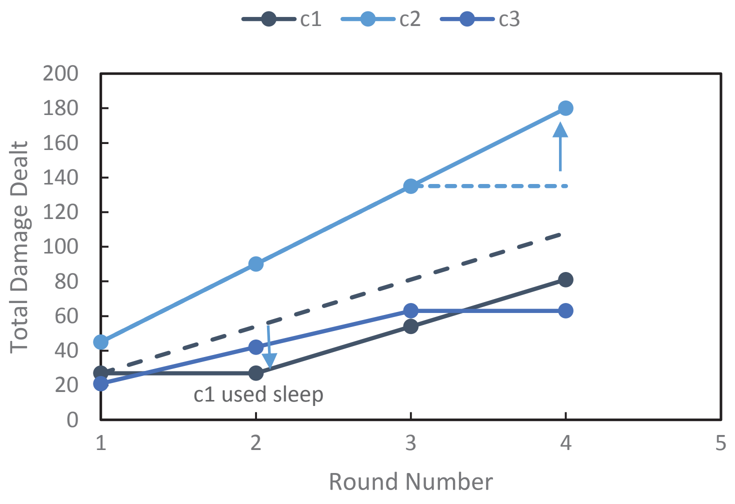


Figure 4.3: Time-damage chart for c_1, c_2, c_3 against the enemy team with sleep involved

The dashed lines represent the damage companions dealt in the previous case without using *sleep*, and the arrows indicate the changes to the damage-dealing after *sleep* is used. As the figure shows, when c_1 uses *sleep* in Round 2, it foregos an *attack* move and thus loses a portion of damage to the enemy team. To compensate this damage drop by c_1, companion c_2 makes extra damage in Round 4 in order to kill the last enemy.

On the other hand, the enemy team also experiences damage changes due to the *sleep* move, as illustrated in Figure 4.4.

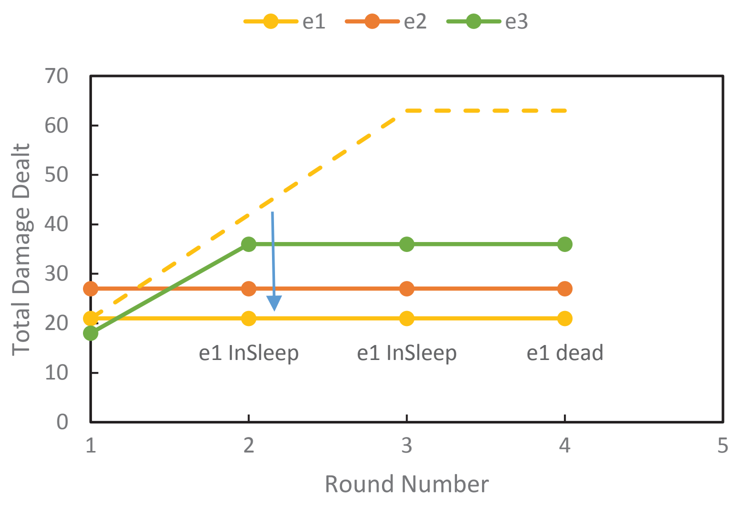


Figure 4.4: Time-damage chart for e_1, e_2, e_3 against the companion team with sleep involved

When e_1 gets slept in Round 2, it makes no damage in that round and the total damage dealt decreases by the end of the combat.

4.1.2 Trade-off Analysis

The time-damage charts imply that using a *sleep* move has both benefits and cost with respect to using a normal *attack* move. In this subsection, we formalize the benefit-cost of using *sleep* and discuss the trade-offs.

Benefit

The benefit of using *sleep* for companions is the damage drop of the target enemy (labeling it e_{slept}) during the time it cannot attack. This decreased portion of damage is essentially the product of e_{slept}'s attack power and the length of sleeping rounds (namely *SLEEP_DURATION*). Note that this product is also the maximum loss for the enemy team (equivalently the maximum benefit for companions) since after *SLEEP_DURATION* rounds the slept enemy recovers to normal, if it is still alive. Thus we have,

$$\textit{benefit}_{\textit{max}} = e_{\textit{slept}}.a * \textit{SLEEP\_DURATION} \tag{4.1}$$

For example in Figure 4.4 where e_1 gets slept and $e_1.a = 21, \textit{SLEEP\_DURATION} = 2$, the damage drop for e_1 is $21 * 2 = 42$.

However, when $e_{\textit{slept}}$ is attacked and dies early, within the sleeping rounds, the damage drop is less than the value Equation (4.1) gives. Instead, the benefit becomes,

$$\textit{benefit} = e_{\textit{slept}}.a * \textit{number of rounds survived after being slept}$$

To know the exact time the $e_{\textit{slept}}$ survives after being slept is not possible because at this deciding moment the companion does not know the future moves by other companions, and thus cannot necessarily estimate the death of $e_{\textit{slept}}$. Nevertheless, we can calculate the minimum time of $e_{\textit{slept}}$'s future survival, labeling it $\mathbf{r_{min}}$. To kill $e_{\textit{slept}}$ as quickly as possible to obtain this $r_{\textit{min}}$ means that every companion is using *attack* against $e_{\textit{slept}}$ until $e_{\textit{slept}}$ dies (except the *sleep* caster for the current round). This gives the following equation where c_k stands for the *sleep* caster (companion),

$$r_{\textit{min}} = 1 + \left\lceil \frac{\max(0, e_{\textit{slept}}.h - \sum_{i \neq k} c_i.a)}{\sum_{i} c_i.a} \right\rceil \tag{4.2}$$

assuming $e_{\textit{slept}}.h > 0$ and that sleep takes 1 round to cast.

Note that $r_{\textit{min}}$ could be either greater or smaller than *SLEEP_DURATION*, depending on other companions' attack power (denominator in Equation 4.2). So we write the minimum benefit of using *sleep* as,

$$\textit{benefit}_{\textit{min}} = e_{\textit{slept}}.a * \min(r_{\textit{min}}, \textit{SLEEP\_DURATION}) \tag{4.3}$$

Interestingly, when the target enemy has enough health to make $r_{\textit{min}} > \textit{SLEEP\_DURATION}$, then

$$\textit{benefit}_{\textit{min}} = \textit{benefit}_{\textit{max}}$$

Combining Equation (4.1) and (4.3), we have the benefit range as follows,

$$e_{\textit{slept}}.a * \min(r_{\textit{min}}, \textit{SLEEP\_DURATION}) \leq \textit{benefit} \leq e_{\textit{slept}}.a * \textit{SLEEP\_DURATION} \tag{4.4}$$

Cost

The direct cost of using *sleep* is the damage loss by the casting companion foregoing an *attack* move (as well as the PP cost). The amount of loss is equal to the attack power of the current companion c_k. Under our assumption that sleep casting requires one round, this gives,

$$cost = c_k.a \tag{4.5}$$

Trade-off

Subtracting the cost from benefit we get the trade-off of using *sleep*,

$$\textit{trade-off} = \textit{benefit} - \textit{cost}$$

If the result is positive, it means using *sleep* brings more profit to the companion team than using *attack* and the AI agent should consider *sleep* as the next move type. Otherwise the *attack* is used. Based on Equation (4.4), without knowing or predicting when e_{slept} dies, we could not quantify the precise benefit for using *sleep* and thus could not compare the benefit and cost exactly. Accurately predicting the death of enemies requires considering future game states, implying the decision problem resort to the heavy-weighted searching solutions discussed in Chapter 3 again (unless the combat is the trivial 1 vs 1 case).

Therefore, instead of calculating the exact trade-off value, we investigate the boundary values case by case.

- Case A:
  If $e_{slept}.a$ is low or the *SLEEP_DURATION* is short such that

  $$e_{slept}.a * \textit{SLEEP\_DURATION} < c_k.a,$$

  regardless of what r_{min} is, the largest benefit is lower than the cost by Equation (4.1) and (4.5) so that the trade-off is negative.

- Case B:
  If $e_{slept}.a * \textit{SLEEP\_DURATION} > c_k.a$ and,

  $$r_{min} > 1 \text{ , } e_{slept}.a > c_k.a$$

then by multiplying both sides together we have

$$r_{min} * e_{slept}.a > c_k.a$$

The left-hand side is the minimum benefit in Equation (4.3), greater than the cost in the right-hand side, indicating that the trade-off is positive.

- Case C:
  If $e_{slept}.a * \mathit{SLEEP\_DURATION} > c_k.a$ and e_{slept} has low health with

  $$r_{min} = 1$$

  then it is possible to kill e_{slept} by other companions in one round. This greatly reduces the value of sleep, although in a general sense some positive trade-off may be possible as the actual benefit depends on which enemies each companion attacks.

- Case D:
  If $e_{slept}.a * \mathit{SLEEP\_DURATION} > c_k.a$ and,

  $$1 < r_{min} < \mathit{SLEEP\_DURATION}\ ,\ e_{slept}.a < c_k.a$$

  then the relation between benefit and cost is ambiguous. In this case we require multi-round attention from other companions in order to eliminate $e_{slept}.a$. This may be difficult to ensure, and so the death point of $e_{slept}.a$ will likely vary, and the actual benefit falls into the range

  $$[e_{slept}.a * r_{min}, e_{slept}.a * \mathit{SLEEP\_DURATION}]$$

  and thus may be either greater or smaller than the cost. For example, assuming a SLEEP_DURATION of 3, $e_{slept}.a = 10$ and $c_k.a = 25$, we have a fixed cost of 25. If e_{slept} survives just 2 rounds after being slept and dies during sleep then we have a benefit of $e_{slept}.a * 2 = 20$, which is net negative, whereas if e_{slept} survives at least 3 rounds after being slept then we have a benefit of $e_{slept}.a * \mathit{SLEEP\_DURATION} = 30$, and thus leading to a positive trade-off.

- Case E:
  If $e_{slept}.a * \mathit{SLEEP\_DURATION} > c_k.a$ and,

$$r_{min} \geq \mathit{SLEEP\_DURATION}$$

then the enemies suffer from the maximum damage loss making the benefit for companions as large as possible,

$$\mathit{benefit} = \mathit{benefit}_{max} = e_{slept}.a * \mathit{SLEEP\_DURATION}$$

This is greater than $c_k.a$, thus we have a positive trade-off of using *sleep*.

Summary
The trade-off calculation shows that the players' health, attack power, as well as the duration of sleep moves are the three key factors in deciding whether to use *sleep*. Different games may have different value settings for combat players and moves so that the actual gain of using *sleep* could vary greatly. If the cost model of moves is more complex than the PP count model in *Pokémon*, such as based on a global resource consumption that fuels multiple kinds of moves, then the trade-off would need to be associated with more factors besides damage. Such a more advanced move cost model is discussed and left as future work in Chapter 8.

4.2 Sleep Decisions

In this section, based on the trade-off analysis of using *sleep*, we discuss and suggest decisions for both move type selection and move targeting problems.

Move Type Decision
According to the case studies in the trade-off analysis, if the current combat state satisfies case B, D or E for any of the enemy targets, then the companion has significant potential for benefit from using *sleep* and would consider *sleep* as the next move type. If for all the enemies the situation falls into case A or C, then the *attack* is chosen as the next move type.

Move Target Decision

Once a *sleep* move type decision is made, we ought to find the best target to maximize the profit for companions. If there is only one enemy who satisfies the condition of using *sleep*, certainly it is the right target. If there are multiple candidates, then the problem gets complex.

Expression (4.1) showed the maximum benefit one companion could obtain is

$$e_{slept}.a * \mathit{SLEEP\_DURATION}$$

So choosing the enemy whose attack a is high would probably be preferred. On the other hand, Equation (4.3) reminds us that an enemy with high health could guarantee the lower bound, and so ensure the minimum benefit for companions, and thus could also be a decent choice.

Therefore, the enemy having both large health and attack is potentially a better target for companions to use *sleep*. This matches the common intuition that strongest enemy should be restricted as much as possible. In the following section we conduct experiments to further confirm this conclusion.

4.3 Experiment

In this section, we test our theoretical result to see how the derived smart *sleep* strategy (described below) performs in practice, compared with some common strategies seen in many games. These simulations are based on the experimental model described in Section 2.8. Additionally, we include various team setups to simulate different combat scenarios.

Setup

The companion team consists of **3** pokemons as the default choices (c_1, c_2, c_3) defined in Section 2.8.2. The enemy team, on the other hand, is based on different sizes in order to simulate games of different difficulty, and to see how effectively *sleep* plays under various situations, giving us the test scenarios below:

- *Against 3 Enemies* - To simulate the combat where enemies are equal to companions

- *Against 4 Enemies* - To simulate the combat where enemies are slightly stronger
- *Against 5 Enemies* - To simulate the combat where enemies are much stronger
- *Against 6 Enemies* - To simulate the combat where companions are very unlikely to win

Enemy pokemons are selected at random from the entire *Pokémon* monster database (introduced in Section 2.8.1). Additionally, we want each individual enemy and companion to be equally competitive so that the tests of different team sizes are valuable, therefore the following criteria is also applied when randomly choosing enemy pokemons. We take the enemy's average individual attack power $\frac{1}{|E|} * \sum\limits_{e \in E} e.a$ and average individual health $\frac{1}{|E|} * \sum\limits_{e \in E} e.h_{max}$, and make sure they differ at most $\pm 20\%$ from those averaged attributes of the companion team:

$$\frac{1}{|E|} * \sum_{e \in E} e.a \in \left[\frac{0.8}{|C|} * \sum_{c \in C} c.a,\ \frac{1.2}{|C|} * \sum_{c \in C} c.a\right]$$

$$\text{and}\quad \frac{1}{|E|} * \sum_{e \in E} e.h_{max} \in \left[\frac{0.8}{|C|} * \sum_{c \in C} c.h_{max},\ \frac{1.2}{|C|} * \sum_{c \in C} c.h_{max}\right]$$

The *SLEEP_DURATION* is set to 3 as the default value in *Pokémon*. For combat result, we use both **SUM_H** (remaining health score) and **P_CWIN** (percentage win-rate of companions over all test cases) evaluation methods in this experiment.

Strategies

We have implemented 3 strategies in deciding the move type between *sleep* and *attack*, and 4 targeting strategies for the chosen move type. The move type selection strategies for companion AI in test are:

- *Move Type - Smart Sleep:* uses *sleep* if the current combat satisfies Case B, D or E in Section 4.1.
- *Move Type - Random Sleep:* uses *sleep* or *attack* randomly (around 50% each) as long as the move is valid (*sleep* is only used on a *Healthy* enemy).

- *Move Type - No Sleep:* no *sleep* is used (attack only). Comparison with this baseline strategy will show whether sleeping is at all useful.

The targeting strategies (once move type is decided) are as follows:

- *Highest Health Targeting:* chooses the enemy who has the highest remaining health.
- *Lowest Health Targeting:* chooses the enemy who has the lowest remaining health.
- *Highest Attack Targeting:* chooses the enemy who has the highest attack power.
- *Lowest Attack Targeting:* chooses the enemy who has the lowest attack power.

Result

We simulate each strategy combination (move type selection and move targeting) and each scenario ("against 3 enemies", "against 4 enemies", etc.) for 1000 times. In each run, the choices of enemies are randomized (from Index 001 to Index 721 in database) to create a specific team size, and then becomes fixed for all strategy tests within that run to reduce noise in comparing strategies. When combat terminates, the averaged **SUM_H** health score as well as the **P_CWIN** win-rate score are taken for each strategy.

The **SUM_H** result is shown in Figure 4.5, with error bars indicating ± 1 standard deviations. In the "vs. 3 enemies" scenario, we can see that *Smart Sleep* performs roughly the same as *Random Sleep*. When there are equal numbers of companions and enemies, the setting that companions always move first results in huge disadvantage to enemies, and individual enemies have a high likelihood of being killed by companions in a single round: companions barely suffer any damage as long as some *sleep* is cast. The average number of times *sleep* is used per combat is plotted in Figure 4.6. In this, for the "vs. 3 enemies" case, it can be observed that the *Smart Sleep* strategy uses less *sleep* according to the sleeping condition of Case C in Section 4.1. This implies *Random Sleep* actually wastes some unnecessary *sleep* casts on enemies while achieving similar SUM_H scores as *Smart Sleep*.

As the combat becomes more difficult for companions in the "vs. 4 enemies", "vs. 5 enemies" and "vs. 6 enemies" cases, *Smart Sleep* starts to show advantage over the *Random Sleep*. In these situations some companions die during the combat, and enemies

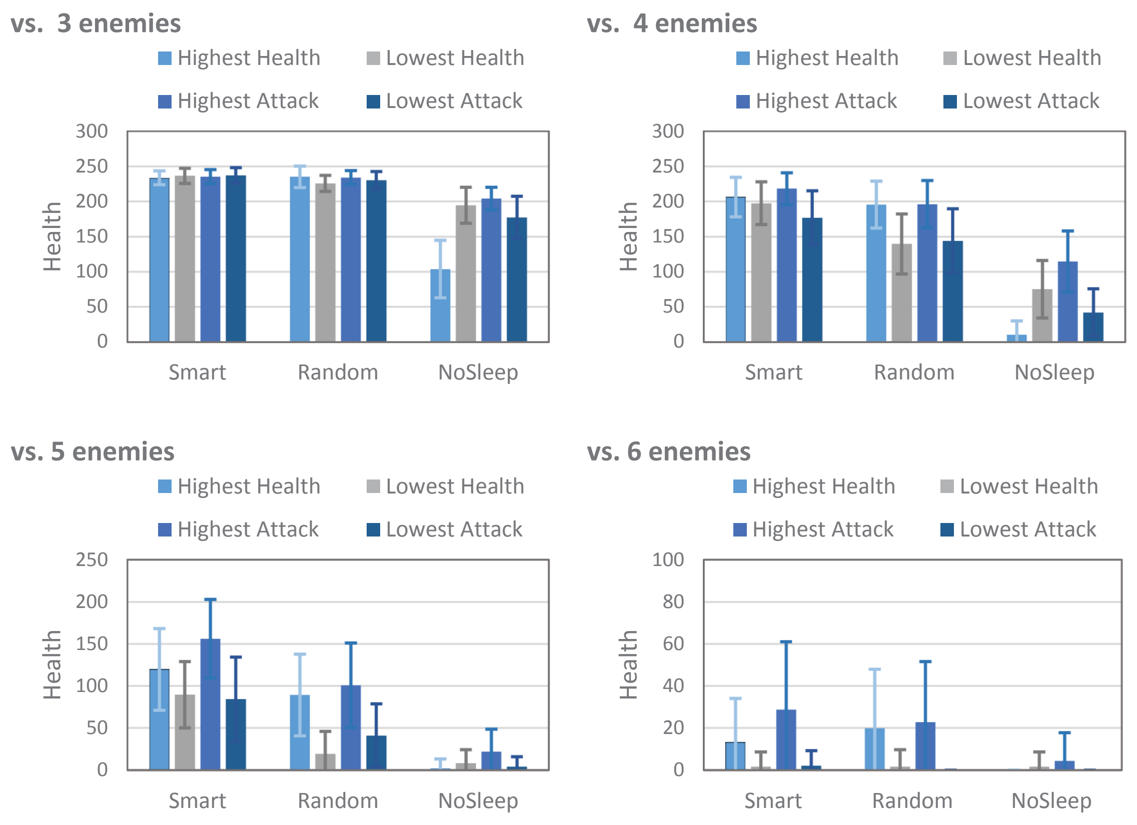


Figure 4.5: Total remaining health of companions by each sleep strategy

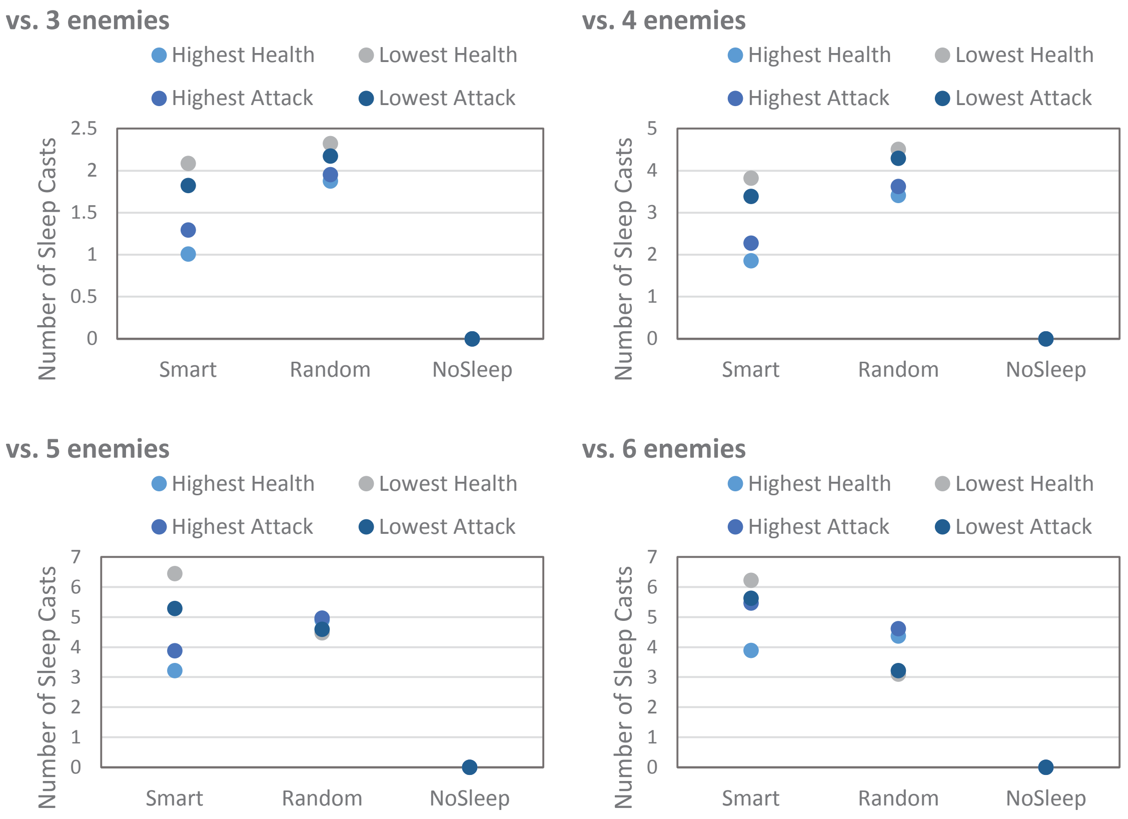


Figure 4.6: Averaged total number of sleep casts per combat of companions

thus have less risk of being eliminated within one round. *Smart Sleep* is able to detect this real-time situation (as Case B or Case D in Section 4.1) and input more *sleep* casts (as shown in Figure 4.6), reducing enemy damage, and bringing higher remaining health to companions. On the other hand, *sleep* targeting strategies also begin to have an impact on the result in these tougher scenarios for companions. *Highest Health Targeting* and *Highest Attack Targeting* appear to be the better targeting tactics for *sleep* as the number of enemies increases, confirming our deduced conclusion in Section 4.1 and 4.2 that an enemy with either high health or high attack power should be a better target to sleep.

In all the four charts, the *No Sleep* strategy performs the worst. Although *sleep* is designed to be useful in most games, the dominant pattern of using *sleep* against using *attack* in this experiment implies that the *Pokémon* game's data and move settings specifically strengthen the power of *sleep*. We note that this is not necessarily the case in every attrition game, especially in games that have more complex moves or synergy between the moves of different characters. In the game *World of Warcraft* for example, the attack of a player can be temporarily magnified by teammates [wow], potentially making *sleep* less beneficial than the enhanced *attack* move (although with a higher cost).

Next, we take a look at the **P_CWIN** score for all strategies, shown in Figure 4.7. The trend remains similar to the **SUM_H** chart as the *Smart Sleep* has increasingly larger advantages over the other two strategies. Surprisingly, however, in the "vs. 5 enemies" and "vs. 6 enemies" scenarios, we notice that *Random Sleep* has very low chance of winning, even if its remaining health score (in winning situations) in Figure 4.5 does not differ that much from *Smart Sleep*. This suggests further that *Smart Sleep* implicitly focuses more on helping companions win the game rather than achieving high health, especially when enemies are stronger, which is preferred by many modern games in general.

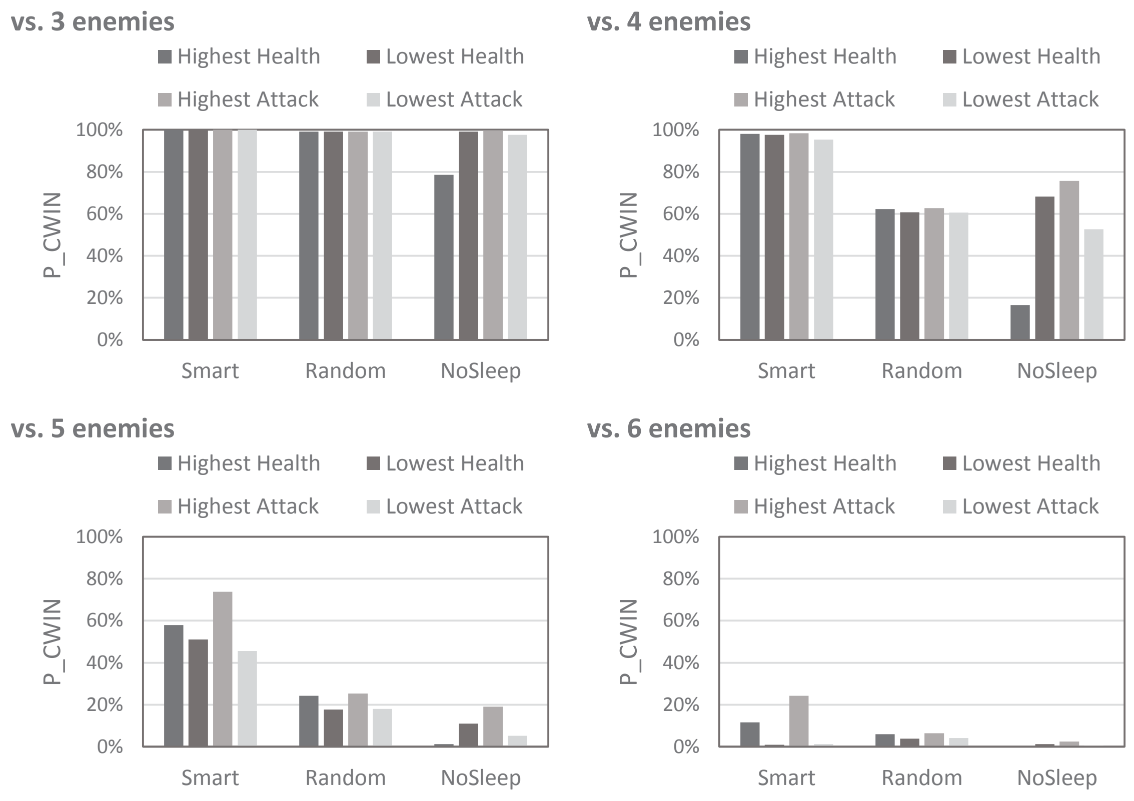


Figure 4.7: Percentage win-rate for the companion team over all 1000 test cases by each sleep strategy

To conclude, the derived *Smart Sleep* strategy shows improvements to a sleeping strategy in practice by considering and categorizing the current combat information each turn. In comparison with *Random Sleep* we do not see any degradation between equivalent targeting strategies, and we always exceed the *No Sleep* strategy. Experimental results also show that the different targeting strategies match our analytical inference done in previous sections.

Chapter 5

Heuristic Approach for Heal Moves

Healing moves appear frequently in combat systems of many modern games. Different from moves like sleep that restrict or harm the enemy, healing moves raise health for the move caster itself or its allies, improving the outcome by giving opportunities for staying alive longer. In this chapter, we follow the same analytical pattern as we did for sleep to investigate heuristic strategies of using healing moves specifically.

In Section 5.1, we briefly review the *heal* move setting in the game model. Then in Section 5.2, based on the evaluation of how companions can benefit from the extra health gained by healing, we analyze and approximate the trade-offs between using *heal* and *attack* moves, given that enemy strategies are not fully visible to companions. In Section 5.3, we provide suggestions on the move type selection and move targeting problem according to the *heal* analysis. Finally in Section 5.4, we perform experiments to determine the performance of our decision heuristics in practice.

5.1 Heal Move Setting

As introduced in Section 2.2.2, a *heal* move heals the target for *HEAL_AMOUNT* health and can only be used on the move caster itself or its allies. The maximum times a companion can use *heal* in one combat, PP_{max}, is set to 15 by default for experiments in the *Pokémon* game. The value of *HEAL_AMOUNT* varies in different games. For our experimental model it has been set to 50% of the target's maximum health. Note that in the case of over-healing where a target's maximum health $\mathit{target}.h_{\mathit{max}}$ may be exceeded, the health value is capped at $\mathit{target}.h_{\mathit{max}}$. Let c_H be the healed companion and h' be the companion's health after healed, then

$$c_H.h' = \min(c_H.h_{\mathit{max}}, c_H.h + \mathit{HEAL\_AMOUNT})$$

5.2 Heal Analysis

Recall that in our trade-off analysis of *sleep*, the benefit and the cost are represented by a damage decrease to enemies and to the *sleep* move user respectively, and so both benefit and cost can be both described in terms of a time-damage chart. For *heal*, the cost remains the same—by foregoing an *attack* move the companion team loses an amount of damage equal to the *heal* user's attack:

$$\mathit{cost} = c_k.a \tag{5.1}$$

where c_k is the *heal* user.

However, the benefit part has no direct relation to the damage dealt by any player in combat. In a trivial sense, the benefit is simply the health increase provided to the healing target—the *HEAL_AMOUNT* itself. Intuitively, however, healing is most useful when applied to character who would otherwise be killed by the enemy, and this then relates to the amount of damage done to the enemies. The recovered portion of health potentially extends the healed companion c_H's lifetime by a certain number of rounds, Δt, and the damage dealt by c_H during this extra time is the gain for the companion team. This gives

$$\mathit{benefit} = \Delta t * c_H.a \tag{5.2}$$

where Δt is given by

$$\Delta t = T_H - T_B \tag{5.3}$$

T_H is the number of rounds enemies take to kill c_H after *heal* is used on c_H, expressed by

$$T_H = \left\lceil \frac{c_H.h'}{E(c_H)_H.a} \right\rceil \tag{5.4}$$

where $E(c_H)_H.a$ is the total attack power of the alive enemies targeting companion c_H after healing. T_B is the number of rounds used to eliminate c_H by enemies without *heal* and it is expressed by

$$T_B = \left\lceil \frac{c_H.h}{E(c_H)_B.a} \right\rceil \tag{5.5}$$

where $E(c_H)_B.a$ is the attack sum of enemies targeting c_H for the non-healing case.

Calculating T_H and T_B is hard because both $E(c_H)_H.a$ and $E(c_H)_B.a$ are unknown and highly dependent on how enemies make their actual attacking target assignments. To ensure that using *heal* has advantage over using *attack*, we find the minimum benefit and see if it is greater than the constant cost. According to Equation (5.2) and (5.3), the minimum benefit is obtained by a smallest Δt, which is then produced by the smallest possible T_H and largest possible T_B. By Equations (5.4) and (5.5) this means all the enemies switch their targets from other companions to c_H right after *heal* is used on c_H, so that $E(c_H)_H.a$ is maximized and $E(c_H)_B.a$ is minimized. Note that $E(c_H)_B.a$ could be zero in theory making T_B undefined (divide by zero). In this case we treat $E(c_H)_B.a$ as infinitesimally small, and simply set T_B to be a very large number.

In real games, however, this target switching rarely happens. Enemies usually follow a scripted attacking routine and their priority ranking of targets remains relatively constant over time (also shown in the paper [TDV14]). Here we make a reasonable hypothesis that enemies do not switch their target once decided. This allows us to approximate these unknowns at least as far as assuming $E(c_H)_H.a = E(c_H)_B.a$. Given this constraint, we can find the lower bound of Δt so that the benefit is minimized by setting $E(c_H)_{H,B}$ to be the set of all living enemies, and so maximizing the attack sum:

$$E(c_H)_H.a = E(c_H)_B.a = \sum_{e \in E} e.a \tag{5.6}$$

Then by expanding Equation (5.3) we get

$$\Delta t_{min} \geq \left\lceil \frac{c_H.h'}{\sum\limits_{e \in E} e.a} \right\rceil - \left\lceil \frac{c_H.h}{\sum\limits_{e \in E} e.a} \right\rceil \tag{5.7}$$

and thus the benefit of using *heal* in terms of increased damage done to the enemy is approximated as

$$\textit{benefit} \approx \left(\left\lceil \frac{c_H.h'}{\sum\limits_{e \in E} e.a} \right\rceil - \left\lceil \frac{c_H.h}{\sum\limits_{e \in E} e.a} \right\rceil\right) * c_H.a \tag{5.8}$$

On the other hand, the timing of using *heal* also matters. Think of a situation where companions are about to win the game with a huge advantage against the enemies. In such a situation using *heal* may increase the length of game by losing damage to enemies but provide no benefit as it does not change the combat result. In this case players would probably prefer to finish the game as quickly as possible and use **SUM_D** evaluation instead of **SUM_H** (defined in Section 2.7). To avoid such unnecessary healing, we make another hypothesis that *heal* is used only when there is an ally in danger. If one of the companions could potentially get killed within the next round, then saving it would most likely achieve a benefit, assuming healing can prevent the untimely death. The minimum health that could surely prevent a kill is expressed by

$$h_{\textit{safe}} > \sum_{e \in E} e.a \tag{5.9}$$

This ensures the companion's current health is larger than the maximum damage enemies could possibly deal in one round. Therefore, the condition of using *heal* is

$$c_H.h \leq \sum_{e \in E} e.a \text{ and } c_H.h' > \sum_{e \in E} e.a \tag{5.10}$$

Note that here we overestimate the opponents' damage on one companion slightly by assuming the attack set is the entire set of enemies. This is conservatively necessary because losing a player means the entire companion team loses a portion of damage for all future combat rounds.

5.3 Heal Decisions

Based on the analysis in the previous section, we now summarize the heuristic for deciding the move type between *heal* and *attack*, as well as selecting the healing target.

For the move type decision, if the current companion c_k is eligible to use *heal* (enough remaining PP) and there exists a companion c_H such that the condition in Equation (5.10) is satisfied, we estimate the benefit of using *heal* on c_H by Equation (5.8) and compare it to the cost by Equation (5.1). If the benefit is higher, then c_k uses *heal*, otherwise it uses *attack*.

For the move targeting decision, if there are multiple qualified healing candidates, we compare the value of approximated benefit for all and choose the one with the highest value.

5.4 Experiment

The theoretical result provides an insight into the trade-off of using *heal* moves from a general perspective. Actual game combat, however, contains many variables such as size of the team, the value of health and attack, and so forth. In this section, we examine the performance of the derived smart healing heuristic in practice by comparing it with other common strategies, based on the data and combat framework from the *Pokémon* game. The test includes various team setups to simulate different scenarios with specific combat evaluations.

Setup

The companion team consists of **3** pokemons as usual (refer to the default ones in Section 2.8.2). To simulate combats of different difficulty, and thus where healing may be useful, we alter the size of the enemy team, giving us the following test cases:

- *Against 2 Enemies* - To simulate the combat where enemies are weak
- *Against 3 Enemies* - To simulate the combat where enemies are equal to companions
- *Against 4 Enemies* - To simulate the combat where enemies are slightly stronger
- *Against 5 Enemies* - To simulate the combat where enemies are much stronger
- *Against 8 Enemies* - To simulate the combat where companions are very unlikely to win

Enemy pokemons are selected randomly from the entire *Pokémon* database and are individually competitive with companions (same as experiment in Chapter 4). Note that here we test against both a lower and a higher number of enemies than we did for our sleep tests, in order to consider situations where we expect healing to likely be ineffective, and where it is likely to be useful. For the enemy teams of size 2, 3, 4 and 5, we evaluate the combat using **SUM_H** health score, under the expectation that the companions are able to at least sometimes able to survive. For the enemy teams of size 3, 4, 5 and 8, we test by using **SUM_D** evaluation to investigate the situation where just maximizing the damage made by companions is important (such as in a boss fight where enemies may be unbeatable). For the **SUM_D** evaluation, we assume that the enemies do not heal themselves to minimize the noise in result. Lastly, we evaluate the percentage win-rate **P_CWIN** for all five test scenarios to find out whether a good healing strategy is truly effective in raising the chance of winning for companions.

Strategies

We have implemented 4 strategies in deciding the move type, and 2 strategies in deciding the target of *heal* for companions. The targeting strategy for *attack* is the *highest threat strategy* discussed in the paper [TDV14].

- *Move Type - Smart Heal*: the smart strategy derived in Section 5.3.
- *Move Type - Greedy Heal*: uses *heal* if there is at least one companion whose health is below a certain threshold (here we set it as 50%). This strategy is commonly used in modern games.
- *Move Type - Random Heal*: uses *heal* or *attack* randomly, regardless of each companion's health.
- *Move Type - No Heal*: no healing is used (attack only). Comparison with this baseline strategy will show whether healing is at all useful.

The healing targeting strategies are as follows:

- *Smart Targeting*: the targeting strategy based on comparing the healing benefit as described in Section 5.3
- *Random Targeting*: randomly picks the target among qualified candidates

Result

Simulations of each strategy combination in each scenario are run 1000 times, sufficient for the data result to show trends and patterns. In each run, the choices of enemies are randomized (from Index 001 to Index 721 in database) and fixed for all strategy tests within that run. We visualize the results in three figures with **SUM_H** evaluation in Figure 5.1, **SUM_D** evaluation in Figure 5.4 and **P_CWIN** evaluation in Figure 5.6. The error bars in Figure 5.1 and 5.4 represent ± 1 standard deviation of the data.

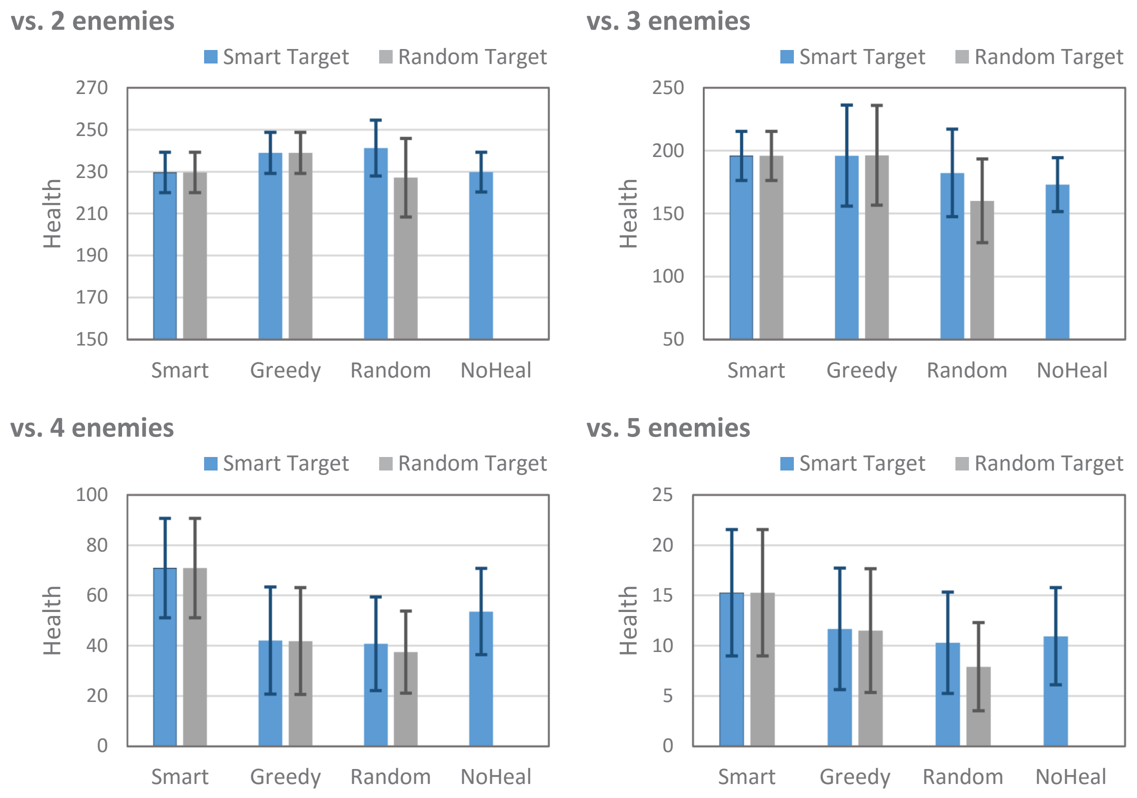


Figure 5.1: Total remaining health of companions by each healing strategy

In Figure 5.1, when using **SUM_H** evaluation, *Smart Heal* does show advantages over other strategies in most cases. An exception is seen in the left-top chart where companions play against only 2 enemies. Unsurprisingly, when enemies are relatively weak companions are very likely to survive without healing themselves. This explains the result that *Smart Heal* behaves similar to *No Heal*. The *Greedy Heal* strategy, however, focuses only on the companions' health and heals even if healing is not necessarily needed. This is also true of *Random Heal*, and thus in combat against few enemies we can see that *Greedy* and *Random Heal* substantially increases the final health of companions, deriving benefit from the simple increase in final health a heal action provides. We note that this improvement comes at the price of a longer combat length, as shown in Figure 5.2.

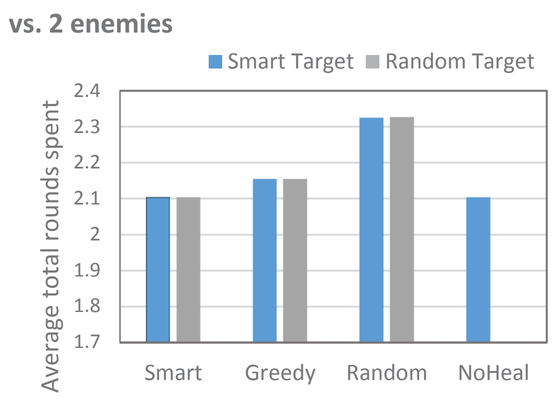


Figure 5.2: Average total number of rounds spent in combat of "vs. 2 enemies"

As the combat becomes more difficult, and especially in the bottom two charts "vs. 4 enemies" and "vs. 5 enemies" of Figure 5.1, we can see not only that *Smart Heal* becomes a much better strategy, but also that *Greedy* and *Random Heal* start to perform even worse than *No Heal*. With more enemies, it is possible for many enemies to target the same companion and one *heal* might not be enough to allow c_{healed} to survive. Evaluating the current combat situation and each companion's status becomes more crucial in deciding whether or not *heal* is used. We can see the impact of selective healing by plotting the average number of *heal* casts, shown in Figure 5.3. In this we notice that for 3 or 4 enemies, *Random Heal* has made many more *heal*'s than *Smart Heal*, and these are very likely to be unnecessary—the wasted healing action merely causes a loss of *attack* opportunity, allowing enemies to live longer, leading to more damage to companions. This chain reaction also reflects the importance of applying the healing condition in Equation (5.10). At 5 enemies *Smart Heal* and *Random Heal* are healing at approximately the same rate. Here again, however, intelligently choosing heal moves where benefit outweights cost has a large positive impact.

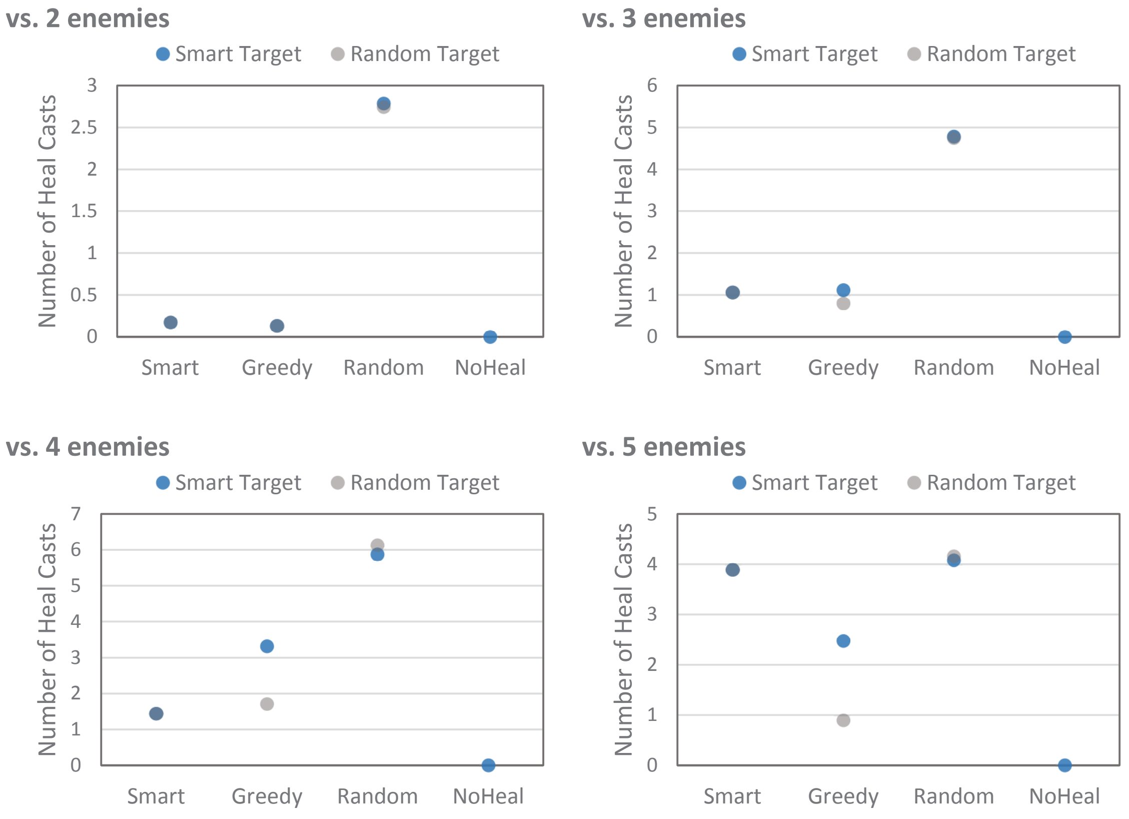


Figure 5.3: Average number of healing casts

Interestingly, the healing targeting strategy seems to have relatively little influence on the result. This is possibly caused by the enemy strategy of trying to focus on the same target as often implemented in modern games, so that most of the time we would have only one healing candidate with very low health, distinguished from other companions. In *Random Heal*, because of the imprecise use of *heal* that leads to a set of potentially unwanted candidates, *Smart Targeting* turns out to be better than the *Random Targeting* by further filtering companions with low healing benefit.

In general, Figure 5.1 implies that with more enemies involved, healing moves should be used more carefully if the remaining health in particular is concerned.

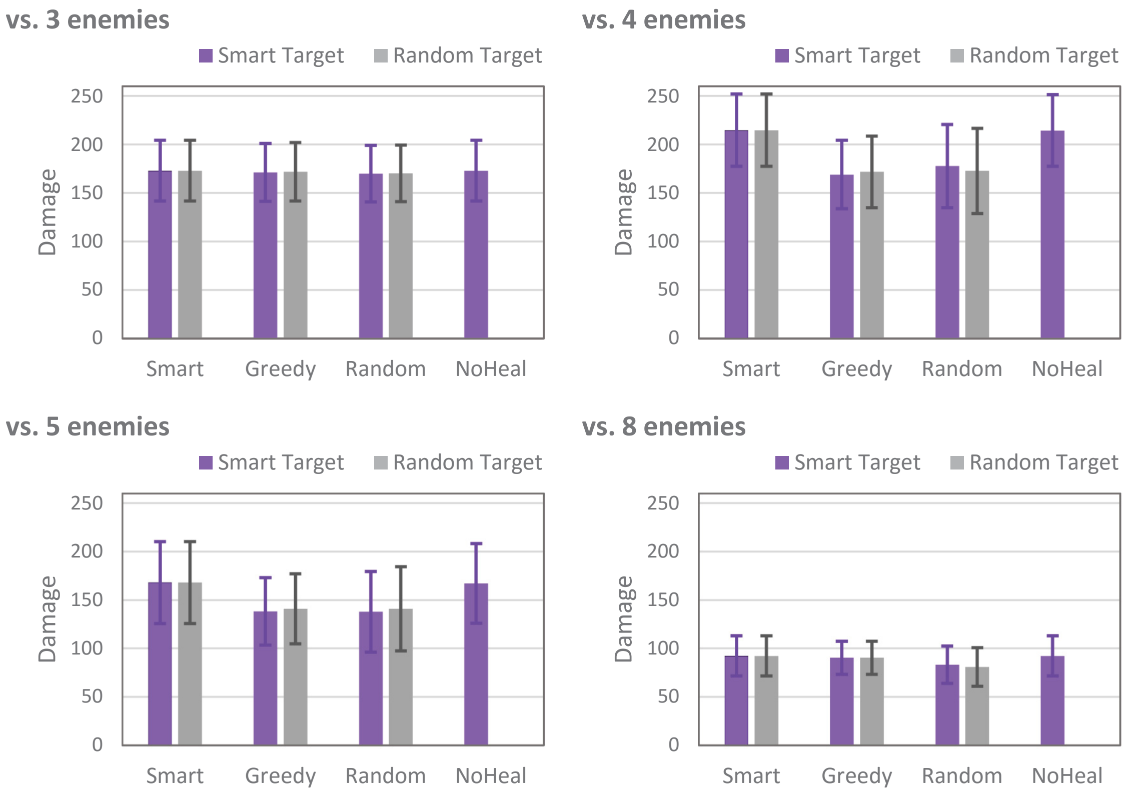


Figure 5.4: Total damage dealt by companions by each healing strategy

On the other hand, Figure 5.4 with the **SUM_D** evaluation illustrates a slightly different pattern of behavior. In the “vs. 3 enemies” case against equal number of opponents, companions deal roughly the same amount of damage regardless of the healing strategies, even though the remaining health given by each strategy may differ. This implies that enemies are all killed in each game, where **SUM_D** is exactly the total sum of their maximum health.

In the games where enemies may survive and win, *Smart Heal* begins to show advantages much like in the **SUM_H** evaluation. Notice that in these cases the *No Heal* strategy tends to do quite well. By not including any *heal* moves, *attack*s are maximized, and thus so is the total damage score. This comes, of course, at the cost of having a much lower remaining health (referring to Figure 5.1).

When we simulate more unbalanced combat with 3 companions playing against 8 enemies, all the healing strategies performs more or less equally again. Here healing is no longer effective at all, as enemies can easily eliminate a companion with full health in one round, leaving no candidates for *Smart Heal* or even *Greedy Heal*. *Random Heal,* which performs some casts of *heal* regardless, only makes the situation worse in this kind of scenario. The number of *heal* casts for this specific scenario is shown in Figure 5.5, further verifying that healing is barely used by *Smart Heal*, only slightly more by *Greedy Heal*, and although much by *Random Heal* it remains ineffective.

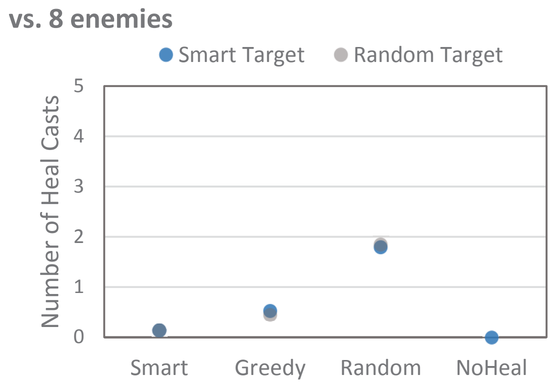


Figure 5.5: Average number of healing casts for vs. 8 enemies case

Finally, the **PC_WIN** percentage win-rate score is displayed in Figure 5.6. The first four charts ("vs. 2" – "vs. 5") follow a similar distribution to the health scores in Figure 5.1. However, by comparing the last four charts ("vs. 3" – "vs. 8") with Figure 5.4 we notice that the *Random Heal*'s winning probability drops quickly despite its damage score being close to others. Meanwhile, the *Smart Heal* strategy makes best use of each portion of damage when it is not using *heal*, toward the ultimate victory. Combining the previous figures we can see that the advantage of *Smart Heal* over other strategies gets larger and larger when combat becomes more challenging for companions, as long as winning is feasible, under all types of evaluations.

Overall, the tests have shown that the performance of healing strategy is stably improved in practice by using *Smart Heal* compared with some other common heuristics, at least outside of situations where healing is moot because the enemy team is either too weak or too strong. This is consistent with the theoretical deduction.

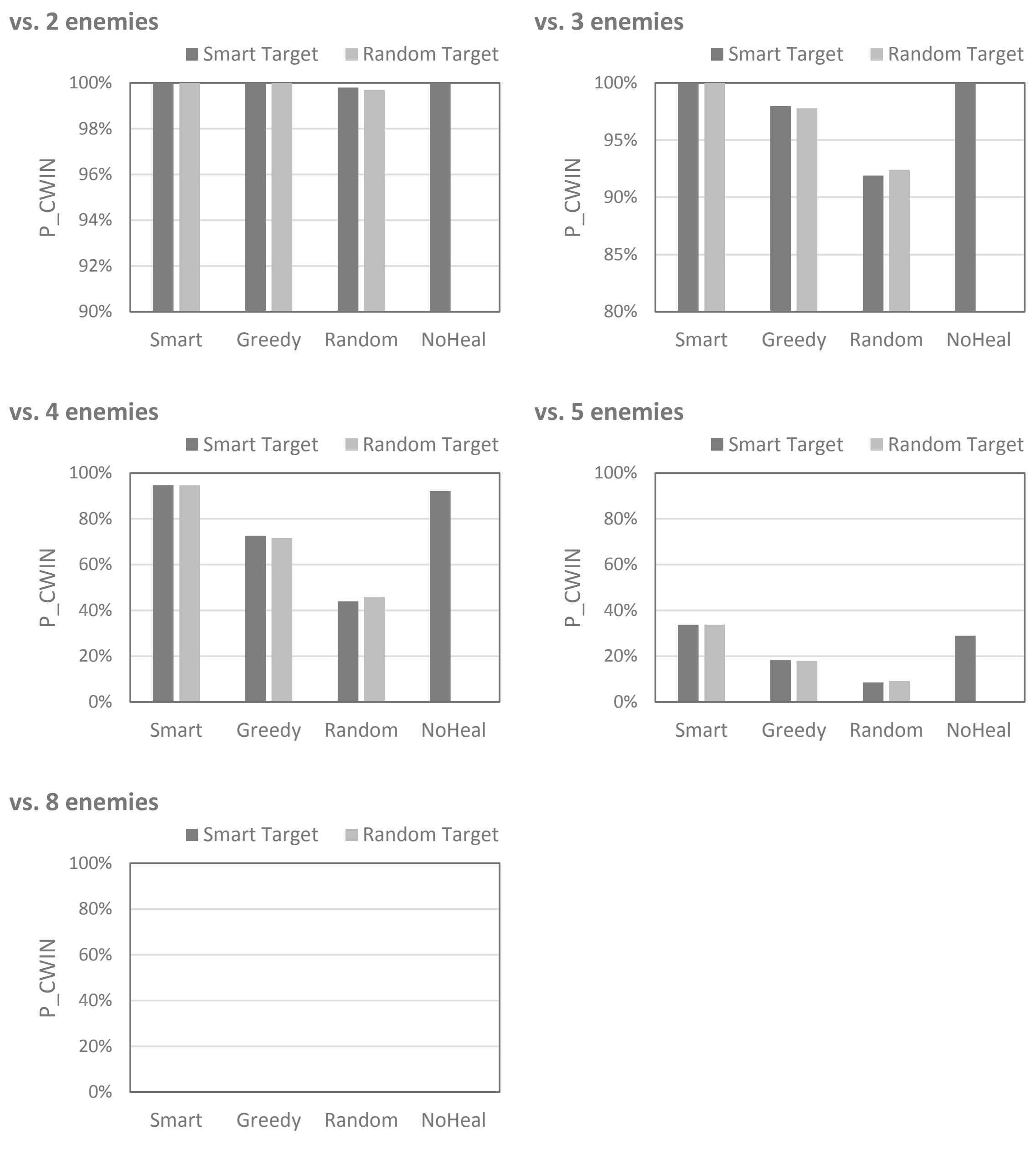


Figure 5.6: Percentage win-rate for companions over all 1000 test cases by each healing strategy

Chapter 6

An Experiment with All Move Types

In this chapter, we combine the smart heuristics of both *sleep* and *heal* in the previous chapters, developing and testing a straightforward strategy that decides on the best move type among all three actions (*sleep*, *heal*, *attack*). We undertake an experiment specifically to test the combined smart heuristic against greedy and random strategies, which are commonly implemented in the game industry.

In Chapter 4 and Chapter 5, we derived conditions and benefits of using *sleep* and *heal* based on damage as well as health of players. By checking the conditions against using *attack* separately, we could know whether *sleep* or *heal* is individually valid to be used or not. If it turns out that both of them are superior to *attack*, then by comparing the benefit values of *sleep* and *heal* directly, we can obtain a final decision out of all three move types.

Section 6.1 describes details of the combined smart heuristic for choosing move types. Section 6.2 gives the experimental setup for comparing the smart heuristic with greedy and random strategies. Lastly, results and discussions are included in Section 6.3.

6.1 Combined Smart Heuristic

Recall that in the analysis in Section 4.1 we decided on *sleep* over *attack* if any of the following condition is satisfied:

- $e_{slept}.a * \mathit{SLEEP\_DURATION} > c_k.a$, $r_{min} > 1$, $e_{slept}.a > c_k.a$

- $e_{slept}.a * \mathit{SLEEP\_DURATION} > c_k.a$, $1 < r_{min} < \mathit{SLEEP\_DURATION}$, $e_{slept}.a < c_k.a$
- $e_{slept}.a * \mathit{SLEEP\_DURATION} > c_k.a$, $r_{min} \geq \mathit{SLEEP\_DURATION}$

Additionally, in Section 5.2 we chose *heal* over *attack* based on the following condition:

- $c_H.h \leq \sum\limits_{e \in E} e.a$, $c_H.h' > \sum\limits_{e \in E} e.a$

Combining these conditions is trivial if at most one of our tests is true: if only *sleep* or only *heal* is considered viable, we perform the action, and if the conditions for neither are satisfied then we *attack*. When conditions for both *sleep* and *heal* are satisfied we select the action with the highest benefit, determining the sleep benefit as in Section 4.1, and the heal benefit as in Section 5.2. If both benefits are equal, we (arbitrarily) default to *sleep*.

For targeting strategies, we choose the *Highest Attack Targeting* if *sleep*, *Smart Targeting* if *heal*, and *Highest Threat Targeting* if *attack*, as these targeting strategies were individually shown best in previous experiments. Other targeting approaches are of course possible, but the purpose of this chapter is to investigate move type selection in scenarios allowing all three move kinds, rather than to test the targeting strategies themselves.

6.2 Experimental Setup

The experiment is again based on the *Pokémon* data and combat framework specified in Section 2.8.2, with fixed move targeting strategies as explained above. To reduce potential noise due to individual pokemon's having distinct move-sets, we make each player capable of using all three move types (*sleep, heal, attack*) and set the PP$_{max}$ for each move type to a very large number (40).

In terms of team composition, we categorize the test data into two groups. The first one represents combat of small team sizes. The companion team contains exactly 3 companions and the enemy team's size varies from 2 to 8, similar to what we have done in *sleep* and *heal* chapters. The second group simulates relatively larger combat scenarios, with the companion team's size fixed to 10 and enemy team's size varying from 8 to 18.

This time, players of both teams are randomized from Index 001 to Index 721 (with equal competitiveness, as described in Section 4.3) in the *Pokémon* database.

Apart from the *Combined Smart* strategy we have derived in Section 6.1, we also include the *Combined Greedy* strategy and *Combined Random* strategy in test, defined as follows,

- *Combined Greedy:*

  ```
  if there exists a living companion whose health is below 25% then
      moveType ← heal
  else if the living enemy with highest attack is not InSleep then
      moveType ← sleep
  else
      moveType ← attack
  ```

  This baseline strategy is similar to the *Pokémon*'s scripted NPC behavior and is simple enough to be adapted to many modern games.

- *Combined Random:*

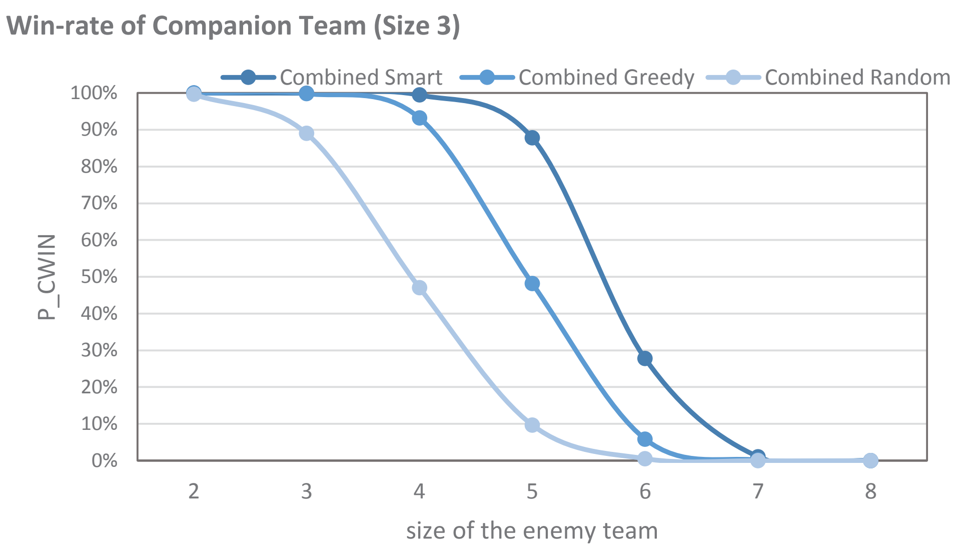
  Randomly assign sleep, heal, or attack to moveType with 1/3 chance each.

  This baseline strategy represents a very naïve approach, but by comparing with it we can know whether a heuristic strategy (even the greedy one) is truly necessary.

6.3 Result

For each group of data, we run the test for each combined strategy 1000 times. The averaged percentage win-rate of companion team **P_CWIN** is taken over all the 1000 runs for each strategy and is shown in Figure 6.1, while the total remaining health score **SUM_H** of two data groups is shown in Figure 6.2. Last but not least, we have recorded the number of each moves casts in Figure 6.3 to reproduce and investigate companion behaviors during combat under each strategy.

From the **P_CWIN** result we can see that, surprisingly, just by straightforward comparison of the potential benefit of each move type produced by formulas in previous chapters, the *Combined Smart* strategy achieves a much better performance than *Combined Greedy*



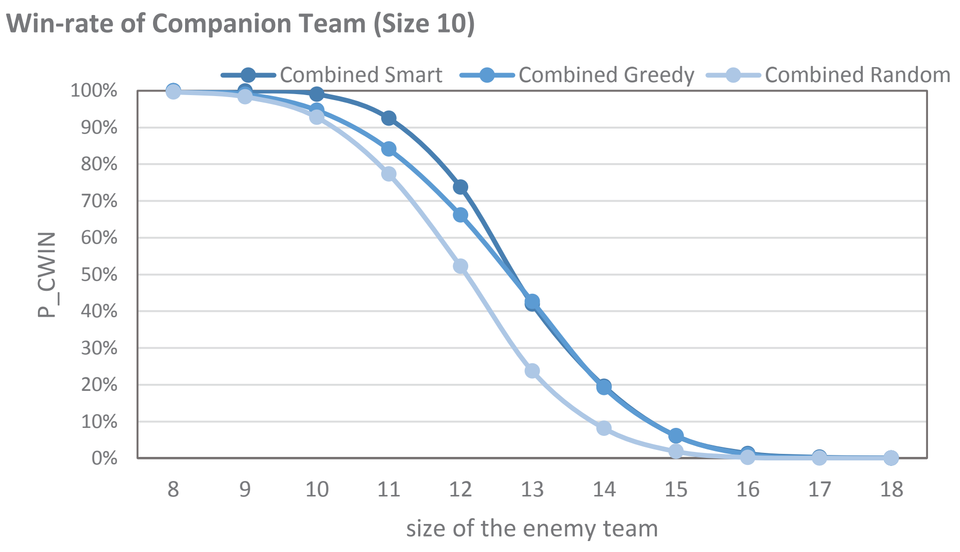


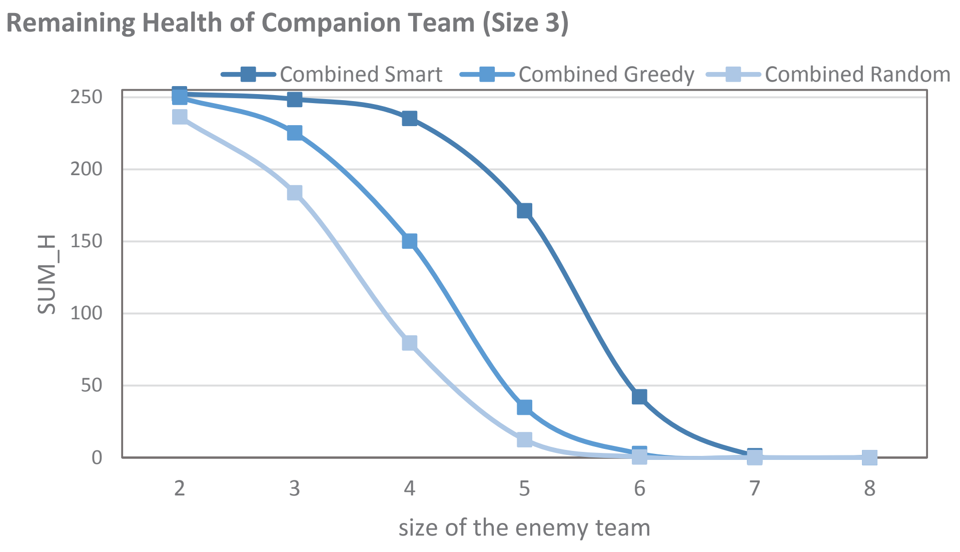
Figure 6.1: Percentage win-rate for companions by each combined strategy

and *Combined Random*. In the “3 companions vs. 5 enemies” case shown in the top chart for example, *Combined Smart* could still maintain a high win-rate (close to 90%) while both of the other two strategies have fallen below 50%.

A further comparison of the two charts with different combat sizes shows that as the number of players increases, the *Combined Smart* strategy becomes less effective. The curves are much closer in “Companion Team with Size 10” as opposed to the “Size 3” case, implying that our derived strategy works best in small-scale attrition games. We can attribute this change to the increasing complexity of enemy behaviors when more players are added, which leads to a greater challenge in accurately approximating the benefit of each move type. With more players involved the range of potential benefit between different approaches becomes quite large—consider, for example, the difference between 10 enemies focusing attack on 1 companion and distributing their attack equally on 10 companions. Our heuristics, while demonstrably effective with limited options, are not able to find best moves as consistently at large scales as they can within the small scale attribution games we have focused on.

The **SUM_H** results in Figure 6.2 show a more or less similar trend to the **P_CWIN** data. Some slight differences exist, however, and we can see that in terms of remaining health, the *Combined Smart* seems to have a larger advantage over *Combined Greedy* in the small-scale case, and also that the difference between the three strategies is even less in the large-scale combat case.

More interesting results are found in the average number of move casts, shown in Figure 6.3. Here we see that overall the use of *sleep* and *heal* tends to reach its peak at the largest combat sizes for *Combined Smart*, smaller for *Combined Greedy*, and smaller still for *Combined Random*. This essentially matches the way win rates decay for the different strategies, reflecting the fact that the better a strategy is the larger a combat situation it can handle and still be effective. Interestingly, the use of sleep and heal by *Combined Greedy* and *Combined Smart* in the Size 10 results is much less than in the Size 3, in proportion term. In fact, in the top-right chart (Sleep Casts, Size 10) we notice that the *Combined Smart* strategy does not cast *sleep* at all. According to Case C of calculating the sleeping benefit in Section 4.1, this is not unexpected: it is possible to kill any individual enemy in one round given a large number of living companions attacking it, and so casting *sleep*



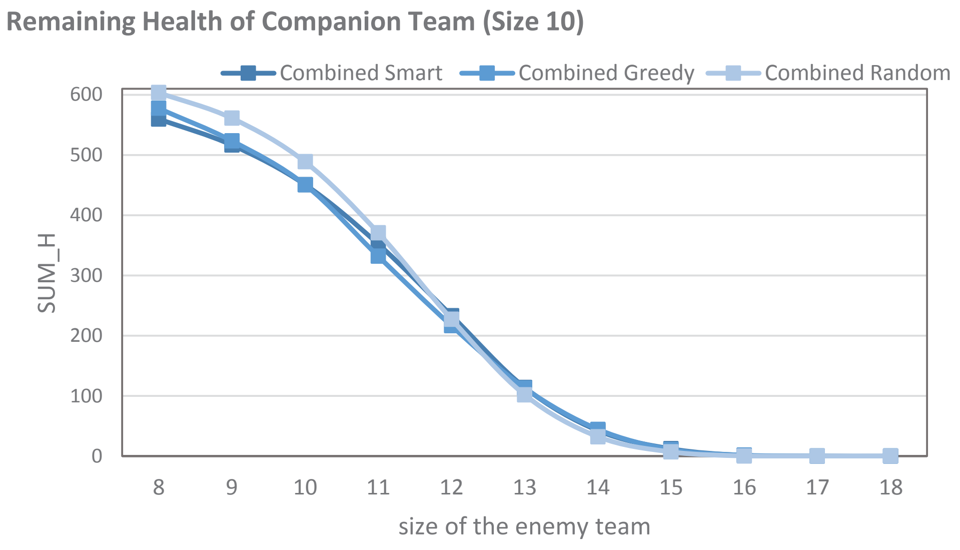


Figure 6.2: Total remaining health of companions by each combined strategy

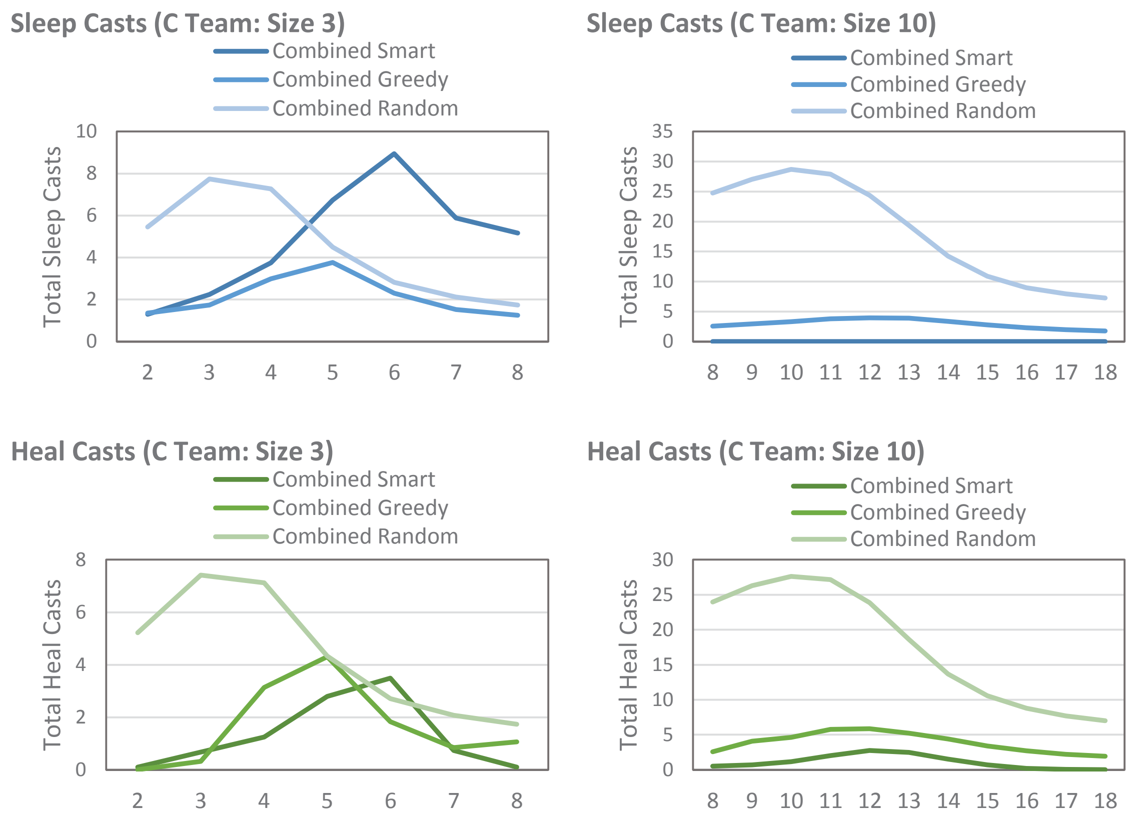


Figure 6.3: Average number of each move casts

is not considered necessary. This, however, further reveals the limitation of the derived combined heuristic for handling large-scale combat, where a more complex distribution of targets and moves may have significant advantages. Nevertheless, combining heuristics of individual moves still shows good potential in improving existing scripted strategies for more general small-scale situations.

Chapter 7

Related Work

Our work in the thesis analyzes several AI decision problems for NPC companions in small-scale turn-based attrition games. We initially addressed this problem by investigating search approaches, such as the brute-force tree search and the more efficient RRT algorithm. The model for our game simulation is extensible so that other approaches may also be applied. In the bulk part of thesis, however, we focused on the properties and trade-offs of two popular, individual moves and came up with heuristic strategies for each move. The heuristic approaches have improved the performance of making companion AI decisions under most scenarios compared to other commonly used strategies, without the cost of doing expensive searches.

7.1 Analyzing Attrition Games

The main focus of our work in on small-scale attrition games. Attrition games can be very complicated in general. The paper by Furtak, T. et al. [FB10] presents proofs on the complexity of two-player attrition games to show that the problem is computationally hard for most game cases: to decide the existence of deterministic winning strategies for basic attrition games is PSPACE-hard and in EXPTIME. The research by Ontanón et al. [OSU$^+$13] provides a survey of existing works on solving AI problems in the commercial game *StarCraft*—a much larger scale example of an attrition game compared to the ones we consider. They discuss topics such as current tactics, strategies, and state-of-art AI agents for StarCraft, shedding light on challenges in general attrition games.

In analyzing attrition games, some people tend to use behaviour based methods. The work by Swen Gaudl et al. [GDB13] describes the Behaviour Oriented Design (BOD) approach, a technique based on Finite State Machines, and Behaviour Trees. Behaviour Trees are a general decision-making framework [Ogr12], heavily used in real games and have great potential to be combined with our move-specific heuristic models in *Chapter 4* and *Chapter 5*. When we start to merge heuristics for different types of moves, Behaviour Trees could provide a framework to further test and improve these AI decisions, and integrate them into more general decision-making.

7.2 Tree Search Strategies

We have analyzed several discrete space search strategies in this thesis (mainly in *Chapter 3*) for solving AI players' decision problems. In particular we have discussed brute-force search and Rapidly Exploring Random Tree (RRT) search. The brute-force method is often considered an initial approach in many searching problems. Faiz Ilham Muhammad [Muh12] describes the implementation of brute-force search using BFS in small-scale games such as turn-based games, maze games, etc. RRT was first introduced by Steven M. LaValle's work [Lav98] in 1998 to solve path-finding problems, while Morgan Stuart Bruce then adapted it to the sampling-based planning algorithm for discrete space problems in his thesis in 2004 [MB04].

Besides these two approaches, there has been research on other methods as well. Monte Carlo Tree Search (MCTS) [MTC] is a heuristic search algorithm that also relies on random sampling. Bruce Abramson first experimented with the idea in turn-based two-player games including *Tic-tac-toe*, Chess, in his work [Abr91] in 1991. Years later the MCTS algorithm had been extended to solve AI problems in more genres of modern computer games including real-time games such as *Total War: Rome II* [Cha14], card games such as *Magic: The Gathering* [WC09], etc. Recently a paper by Browne et al. [BPW$^+$12] provided a snapshot that summarized recent work on the MCTS algorithm itself.

Work on Fast Alpha-Beta Search [CSB12] suggests another possibility for improving game AI in combat scenarios. The Alpha-Beta Search is based on durative moves in Real-Time Strategy (RTS) games, which could be potentially reduced to a turn-based game model to fit our thesis. Experiments have shown that this search algorithm performs better than many scripted strategies in RTS games. It runs efficiently and is scalable for large combat scenarios involving over 20 player units. The Alpha-Beta Search is also mentioned in the paper [KB05] by Alexander Kovarsky et al., which briefly discusses a list of different feasible strategies.

7.3 Other General AI Approaches

For improving general game AI, there are a variety of other approaches. Neural Networks are one such technique that can be used as a tool to decide actions for AI players. Darryl Charles and Stephen McGlinchey reviewed the history of using neural networks in games and identified its advantages and disadvantages [CM04]. Kenneth O. Stanley et al. explained in the paper [SBM05] how neural networks could be used to train AI players in video games based on the NERO project [Col03]. Ross Graham et al. addressed in their paper [RGS04] an approach to enhance the AI path-finding ability by using neural networks. In our turn-based game models, neural networks can be considered as an alternative method to help NPC companions make move decisions through training so that the more games they play the better action they choose, although this is in general an expensive strategy in terms of time required.

Adaptive Spatial Reasoning [BS08] is another approach in designing a Turn-Based Strategy (TBS) AI. Based on the ADAPTA architecture (Allocation and Decomposition Architecture for Performing Tactical AI), this approach yields more satisfactory performance over multiple scripted AI tactics. It is possible that using this ADAPTA architecture could further optimize our enhanced heuristic strategies in this thesis.

Chapter 8

Conclusions and Future Work

In this chapter, we summarize and conclude our work in the thesis and give an insight into possible future work.

8.1 Conclusions

Artificial Intelligence (AI) in modern video games has obtained increasing attention from both academics and game designers. Not only do we expect smart behaviors generated for both NPC companions and enemies to make games fun, but also we require fast AI response to provide better play experience. While the AI of NPC enemies has been analyzed for decades, the AI of NPC companions still has a large amount of room for optimization.

In this thesis we built an analytical game model for attrition games, explored different existing algorithms for solving AI decisions for NPC companions and in the end improved the decision strategies for various moves and scenarios. We found out that although state-based search algorithms such as brute-force or RRT can be applied to help find best moves for companions, their time efficiency is generally not acceptable for even turn-based interactive games, at least not without a great deal of further optimization. On the other hand, some commonly used greedy algorithms run fast and are simpler to implement, with the drawback that we have to resolve complexity concerns for games allowing a large set of various actions. A straightforward greedy strategy can easily fail to find good solutions under certain game settings.

By looking at the specific actions of sleeping and healing, we made an approach in the rest of the thesis to improve the performance of heuristic algorithms, especially with respect to the possibility of finding a better game result compared to commonly implemented strategies in modern games. Based on the move cost/benefit evaluations, we have shown for each move, how, when and on which target it should be used so that the outcome is generally better than with simple greedy algorithms. Meanwhile, we argue that for some decision problems, an optimal solution may not be feasible at the point the companion AI agent is required to respond; still, we can find good (if not necessarily optimal) solutions under certain assumptions and restrictions on the game. Our experiments applying data from the small-scale battles in the *Pokémon* game have further verified that our enhancement for companion AI is potentially effective in real game contexts.

8.2 Future Work

There are many extensions to this work. We have investigated how companions should use each individual action wisely depending on costs and benefits, and made an simple experiment of directly combining the individual heuristics together by comparing damage-related benefits. The improvement of game result for larger combats, however, is not so obvious. It could be worthwhile to theoretically analyze multiple moves together if they are given at the same time in combat. Figuring out the relations among all types of moves and trying to design the corresponding decision trees can help us further increase the performance of existing greedy approaches.

In our experiments for heuristic approaches, the move cost parameter PP_{max} for each move type was set to the default value from the game *Pokémon*, and it was relatively large (10 for *sleep* and 15 for *heal*) with respect to the number of combat rounds, so that the actual move count rarely exceeded this value. Nevertheless, PP_{max} could impact the decision strategies if it is set small enough. Each single use of a move type with limited casts could become more crucial. Think of the situation where $\text{PP}_{max} = 1$ for *heal*. Intuitively, the only *heal* should be used to save the most important ally player, sacrificing other less critical allies (even when healing has a positive trade-off value at that point). A possible approach for handling this is to assign weights to the benefit calculation of each move type, additionally considering the remaining PP amount as well as the value of each target as factors in move decisions. The less PP the move type has, the more tendency it has to be used for the most valuable player.

Using a more advanced move cost model is another big challenge. Our experiments and moves implemented the cast count model that assigned a PP_{max} limitation to each individual move type. When all move types share a mutual resource, such as in the game *World of Warcraft* where all moves cost certain amount of "mana"—a player-specific attribute, considering the mutual resource consumption would be another task for trade-off calculation. This is especially necessary for lengthy combat in which players may run out of resources.

Apart from possible move oriented analyses, we may also apply other AI techniques to potentially improve companion decisions. As mentioned in related work in *Chapter 7*, we can adapt other search algorithms such as Monte Carlo search, or Alpha-Beta search to our game model. Monte Carlo search is famous for its excellent performance in solving Go, whose complexity also relies on the large number of move choices. It is possible that Monte Carlo search would work well for our small-scale attrition games, although we would have to carefully choose parameters in its node evaluation function so that the search tree is well-balanced [BPW$^+$12].

Bibliography

[Abr91] Bruce Abramson. Expected-Outcome Model of Two-Player Games. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1991.

[BPW$^+$12] Cameron B Browne, Edward Powley, Daniel Whitehouse, Simon M Lucas, Peter I Cowling, Philipp Rohlfshagen, Stephen Tavener, Diego Perez, Spyridon Samothrakis, and Simon Colton. A survey of monte carlo tree search methods. Computational Intelligence and AI in Games, IEEE Transactions on, 4(1):1–43, 2012.

[BS08] Maurice H. J. Bergsma and Pieter Spronck. Adaptive Spatial Reasoning for Turn-based Strategy Games. In Artificial Intelligence and Interactive Digital Entertainment Conference. The AAAI Press, 2008.

[bul] Bulbagarden Forums. `http://bmgf.bulbagarden.net/`.

[BW84a] Alan Bundy and Lincoln Wallen. Alpha/beta pruning. In Alan Bundy and Lincoln Wallen, editors, Catalogue of Artificial Intelligence Tools, Symbolic Computation, pages 4–4. Springer Berlin Heidelberg, 1984.

[BW84b] Alan Bundy and Lincoln Wallen. Minimax. In Alan Bundy and Lincoln Wallen, editors, Catalogue of Artificial Intelligence Tools, Symbolic Computation, pages 75–75. Springer Berlin Heidelberg, 1984.

[Cha14] Alex J. Champandard. Monte-Carlo Tree Search in TOTAL WAR: ROME II's Campaign AI, 2014. `http://aigamedev.com/open/coverage/mcts-rome-ii/`.

[CM04] Darryl Charles and Stephen McGlinchey. The past, present and future of artificial neural networks in digital games. In Proceedings of the 5th international conference on computer games: artificial intelligence, design and education. The University of Wolverhampton, pages 163–169, 2004.

[Col03] Digital Media Collaboratory. Neuro Evolving Robotic Operatives (NERO), 2003. `http://dev.ic2.org/nero`.

[CSB12] David Churchill, Abdallah Saffidine, and Michael Buro. Fast Heuristic Search for RTS Game Combat Scenarios. In Eighth Artificial Intelligence and Interactive Digital Entertainment Conference, 2012.

[ess] Pokemon Essentials. `http://pokemonessentials.wikia.com`.

[FB10] Timothy Furtak and Michael Buro. On the complexity of two-player attrition games played on graphs. In Proceedings of the Sixth AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment Conference, 2010.

[GDB13] Swen Gaudl, Simon Davies, and Joanna J. Bryson. Behaviour oriented design for real-time-strategy games: An approach on iterative development for STARCRAFT AI. In Foundations of Digital Games Conference 2013 (FDG 2013), pages 198–205, May 2013.

[KB05] Alexander Kovarsky and Michael Buro. Heuristic search applied to abstract combat games. In Advances in Artificial Intelligence, pages 66–78. Springer, 2005.

[Lav98] Steven M Lavalle. Rapidly-Exploring Random Trees: A New Tool for Path Planning. Technical report, Iowa State University, 1998.

[MB04] Stuart Morgan and Michael S Branicky. Sampling-based planning for discrete spaces. In IEEE/RSJ International Conference on Intelligent Robots and Systems, 2004.

[MTC] Monte Carlo Tree Search. `http://mcts.ai`.

[Muh12] Faiz Ilham Muhammad. Graph searching implementation in game programming cases using BFS and DFS algorithms. Master's thesis, Sekolah Teknik Elektro dan Informatika, 2012.

[Ogr12] Petter Ogren. Increasing Modularity of UAV Control Systems using Computer Game Behavior Trees. In AIAA Guidance, Navigation and Control Conference, Minneapolis, MN, 2012.

[OSU$^+$13] Santiago Ontanón, Gabriel Synnaeve, Alberto Uriarte, Florian Richoux, David Churchill, and Mike Preuss. A survey of real-time strategy game AI research and competition in starcraft. Computational Intelligence and AI in Games, IEEE Transactions on, 5(4):293–311, 2013.

[pok] Bulbapedia, the community driven Pokémon encyclopedia. `http://bulbapedia.bulbagarden.net/wiki/Main_Page`.

[RGS04] Hugo McCabe Ross Graham and Stephen Sheridan. Neural Networks for Real-Time Pathfinding in Computer Games. In Proceedings of ITB Research Conference, 2004.

[SBM05] Kenneth O. Stanley, Bobby D. Bryant, and Risto Miikkulainen. Evolving Neural Network Agents in the NERO Video Game. In Proceedings of the IEEE 2005 Symposium on Computational Intelligence and Games (CIG'05), Piscataway, NJ, 2005. IEEE.

[Sha13] Jon Shafer. Turn-Based VS Real-Time. `http://jonshaferondesign.com/2013/01/03/turn-based-vs-real-time/`, 2013.

[TDV14] Jonathan Tremblay, Christopher Dragert, and Clark Verbrugge. Target selection for AI companions in FPS games. In FDG'14: Proceedings of the 9th International Conference on Foundations of Digital Games, April 2014.

[WC09] Colin D Ward and Peter I Cowling. Monte Carlo search applied to card selection in Magic: The Gathering. In Computational Intelligence and Games, 2009. CIG 2009. IEEE Symposium on, pages 9–16. IEEE, 2009.

[wow] World of Warcraft Official Website. `http://us.battle.net/wow/en/`.