A Comparative Analysis of Greedy Algorithms and Dynamic Programming for AI Strategy in Turn-Based RPG Combat

Muhammad Kinan Arkansyaddad - 13523152
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jalan Ganesha 10 Bandung
E-mail: mkinanarkansyaddad@gmail.com , 13523152@std.stei.itb.ac.id

***Abstract*— Turn-based Role-Playing Games (RPGs) rely on the quality of their Artificial Intelligence (AI) to create engaging strategic challenges. This paper presents a quantitative comparative analysis of two fundamental algorithms for designing combat AI: the Greedy algorithm and Dynamic Programming (DP). A custom text-based combat simulator was developed in Java to provide a controlled experimental environment. Within this simulator, a fast, heuristic-driven Greedy AI was implemented and tested against a methodically optimal DP AI across a suite of eight varied combat scenarios designed to probe strategic decision-making. Performance was measured based on win rates, battle duration, and the computational time required for each AI to make a decision. The results demonstrate a clear trade-off: the Dynamic Programming agent achieved a perfect win rate by finding the globally optimal solution in every case, but at a significantly higher computational cost. Conversely, the Greedy agent was extremely fast but proved strategically brittle, failing completely in scenarios that required any degree of foresight. This study provides a practical, data-driven benchmark of this classic "speed versus smarts" dilemma, offering tangible insights for game developers on selecting appropriate AI strategies for different in-game situations.**

Keywords*— *Game AI, Greedy Algorithm, Dynamic Programming, Turn-Based Combat, Role-Playing Games (RPGs)

I. Introduction

Fig. 1. Baldur's Gate III (Source: https://baldursgate3.game/wallpapers/thumbnails/wallpaper-01-thumb.jpg)

Turn-based Role-Playing Games (RPGs) are a foundational genre in the world of video games, with roots stretching back to tabletop classics like Dungeons & Dragons. Unlike action games that test a player's reflexes, turn-based games are a test of the mind. They give players the space to think, analyze the battlefield, and make calculated decisions about how to best use their characters' unique abilities. A significant portion of the gameplay in these RPGs is dedicated to this strategic combat, making it a vital part of the overall experience [1], [3]. The recent success of titles like Baldur's Gate 3, which won Game of the Year, proves that this thoughtful, strategic style of combat is more popular than ever.

Because players have the luxury of time to plan their moves, the quality of the opponent's Artificial Intelligence (AI) becomes incredibly important. The AI acts as the "brain" for the enemies and is responsible for providing a meaningful challenge. A well-designed AI can turn a simple fight into a complex puzzle, forcing the player to adapt and think critically about their strategy. A poorly designed AI, on the other hand, can make battles feel like a repetitive chore, with enemies that are predictable and easy to defeat. Therefore, the choice of the algorithm that drives this AI has a direct and significant impact on the game's quality.

This paper explores two classics, yet fundamentally different, algorithmic approaches for designing this AI brain: the Greedy algorithm and Dynamic Programming (DP). The Greedy algorithm operates on a simple "live in the moment" philosophy, so at every turn, it looks at its available moves and picks the one that offers the best immediate reward, without any consideration for future turns. It is fast, simple to implement, and often effective. In contrast, Dynamic Programming is the ultimate planner. It methodically explores the entire tree of possible future moves to calculate the single, provably optimal path to victory. It represents a gold standard for strategic perfection but comes at a much higher computational cost. Both techniques are well-established paradigms in the broader field of game AI [2].

The purpose of this paper is to provide a quantitative comparative analysis of these two algorithms for AI strategy within a controlled, simulated turn-based RPG battle. While previous research has explored this field from several angles, this study carves out a specific niche. This paper performs an experimental deep dive into the behavioral logic itself, placing two fundamental algorithms in a direct, head-to-head

comparison. The primary contribution of this work is to generate clear, measurable data that illustrates the classic "speed vs. smarts" dilemma. This paper will present hard numbers on key metrics, such as win rates, combat duration, and the raw computation time for a decision, to create a practical benchmark for game developers. This analysis aims to provide a reference for choosing an AI strategy, weighing the benefits of a perfectly optimal opponent against the significant performance costs it may incur. The remainder of this paper is structured as follows. Section II will cover the theoretical foundations of the Greedy and Dynamic Programming algorithms. Section III details the implementation of our combat simulator and the AI agents. Section IV presents the results of our experiments and provides a detailed analysis. Finally, Section V concludes the paper, summarizing our findings and suggesting potential directions for future research.

II. Theoretical Basis

A. Greedy Algorithm

The Greedy algorithm is a popular and intuitive method for solving optimization problems. Its core principle is to make a locally optimal choice at each step with the hope of reaching a globally optimal solution, a strategy aptly summarized as "take what you can get now!" [4]. The algorithm builds its solution step-by-step, and at each stage, it selects the best available option according to some predefined heuristic, without considering the long-term consequences of that choice. Crucially, once a choice is made, it is irrevocable; the algorithm never backtracks to reconsider a past decision.

The general structure of a Greedy algorithm can be formalized through several key elements [4]:

- Candidate Set (C): The set of all possible elements from which a solution can be built (e.g., coins, jobs, graph edges).
- Solution Set (S): The set containing the candidates that have been chosen to be part of the solution.
- Selection Function: A heuristic function that determines which candidate from C is the "best" choice to consider next.
- Feasibility Function: A function that checks if a selected candidate can be feasibly added to the current solution set S.
- Solution Function: A function that determines if the current solution set S constitutes a complete and valid solution to the problem.
- Objective Function: The function that the algorithm seeks to maximize or minimize.

The primary appeal of the Greedy method is its simplicity and speed. However, its major drawback is that it does not guarantee an optimal solution for all problems [4]. The coin exchange problem is a classic example: for a currency system, a greedy strategy of always picking the largest denomination coin will produce the minimum number of coins. Yet, for a different system of denominations (e.g., coins of value {1, 7, 10} to make 15), the same greedy strategy fails, yielding a suboptimal result [4].

Despite this, for a specific class of problems, a Greedy strategy is provably optimal. Examples include:

- The Activity Selection Problem: By sorting activities by their finish times and always choosing the next compatible activity, the algorithm guarantees the maximum number of activities can be scheduled [4].
- Minimum Spanning Tree (MST): Both Prim's algorithm (which greedily grows a tree from a single vertex by adding the cheapest connecting edge) and Kruskal's algorithm (which greedily adds the cheapest edge in the entire graph that doesn't form a cycle) are guaranteed to find the MST of a graph [5].
- Fractional Knapsack Problem: Unlike the 0/1 Knapsack problem where a greedy approach fails, the Fractional Knapsack problem is optimally solved by greedily choosing items with the highest profit-to-weight ratio [4].
- Huffman Coding: This data compression algorithm builds an optimal prefix-free code tree by greedily merging the two nodes with the lowest frequencies at each step [6].

For problems where optimality is not guaranteed, such as the 0/1 Knapsack Problem or the Traveling Salesperson Problem (TSP), a Greedy algorithm can still serve as a fast heuristic to find an approximate or sub-optimal solution [4], [6].

B. Dynamic Programming

Dynamic Programming (DP) is a more powerful and exhaustive method for solving optimization problems. In contrast to the Greedy method, which commits to a single path of decisions, DP methodically explores multiple decision paths by breaking a problem down into a sequence of stages and solving simpler, overlapping subproblems [4]. The term "programming" here refers not to coding but to the use of tables to construct a solution.

The power of Dynamic Programming is rooted in Richard Bellman's Principle of Optimality. This principle states that an optimal solution to a problem is composed of optimal solutions to its subproblems [4]. This means that the optimal path from a starting state to an ending state must contain optimal sub-paths between any two intermediate states along that path. This property allows DP to build a solution from the ground up, storing the results of subproblems in a table (a technique known as memoization) so they never have to be recomputed.

Problems well-suited for Dynamic Programming typically have the following characteristics [7]:

- The problem can be divided into multiple stages, where a decision is made at each stage.
- Each stage has a set of associated states.

- A decision at one stage transforms the current state into a state in the next stage.
- The cost or value of the solution accumulates stage by stage.
- There exists a recursive relationship that connects the optimal solution of one stage to the optimal solutions of previous stages.

The process of developing a DP algorithm involves characterizing the structure of the optimal solution, recursively defining the value of that solution, and then computing this value in a bottom-up (or top-down) fashion, typically by filling out a table. This approach guarantees a globally optimal solution because it considers all feasible decision sequences.

Classic problems that can be solved optimally using Dynamic Programming include:

- The 0/1 Knapsack Problem: DP can solve this problem, for which the Greedy algorithm fails. The recursive formula decides whether to include an item by comparing the optimal profit without the item versus the optimal profit for the remaining capacity plus the current item's profit [7].
- Shortest Path in a Multistage Graph: DP is used to find the shortest path by calculating the minimum cost from the start to every node at each stage. [7]
- The Traveling Salesperson Problem (TSP): While extremely computationally intensive, DP can find the guaranteed shortest tour for the TSP. It does so by computing the shortest paths for all possible subsets of vertices, building up from smaller subsets to larger ones. [8]

III. Implementation

A. The Combat Simulation Environment

To ensure a controlled and reproducible experiment, a custom text-based combat simulator was developed using Java programming language. This environment serves as the testbed where both the Greedy and Dynamic Programming AI agents operate under an identical set of rules, allowing for a direct comparison of their performance. The simulator manages the game state, enforces combat rules, and logs the outcome of each battle.

The environment is defined by the following core components:

- Character Definitions: The simulation involves two combatants:
    - The Player Character (PC): This is the entity controlled by the AI agents. It begins each battle with 100 HP (Health Points) and 50 Mana.
    - The Enemy: This character acts as a consistent benchmark. It starts with 150 HP and follows a fixed, predictable pattern: on its turn, it always uses "Claw Attack," which deals a constant 12 damage to the PC.
- PC Action System: The AI has three distinct actions it can choose from on its turn, each with specific costs and effects designed to create strategic trade-offs:
    - Basic Attack: Deals 10 damage. Costs 0 Mana.
    - Fireball: Deals 35 damage. Costs 20 Mana and has a 1-turn cooldown (it cannot be used on the turn immediately after it was used).
    - Meditate: Deals 0 damage but restores 15 Mana.
- Combat Flow: The battle unfolds in discrete turns managed by the simulator. The PC always takes the first turn. A turn consists of the active character choosing and executing an action. The simulation concludes when either the PC's or the Enemy's HP drops to 0 or below. The entire battle sequence, including actions taken, damage dealt, and state changes, is printed to the console as a text-based log.

B. Mapping Turn-Based RPG Combat Elements to Greedy Algorithm Elements

To translate the abstract theory of a Greedy algorithm into a functional AI, we must first map its formal components to the concrete elements of our turn-based combat scenario. The design of our Greedy AI is directly based on the elemental structure of a Greedy algorithm as outlined in the lecture notes. The following list details this mapping:

- Candidate Set (C): For any given turn, the candidate set consists of the three possible actions the PC can take: Basic Attack, Fireball, and Meditate.
- Solution Set (S): The solution set is the final sequence of moves chosen by the AI agent from the start of the battle to its conclusion.
- Selection Function: This is the core heuristic of the Greedy AI. The function scores each candidate action based on its immediate, local benefit. For this implementation, the primary heuristic is maximizing damage output on the current turn.
- Feasibility Function: Before an action can be chosen, its feasibility is checked. For the Fireball action, this function verifies that the PC's current mana is sufficient (more or equal than 20) and that the skill is not on cooldown. For Basic Attack and Meditate, the moves are always considered feasible.
- Solution Function: This function determines if a complete solution to the problem has been reached. In the context of the simulation, this function checks for the end-of-battle conditions after every turn. A solution is considered complete when either the PC's HP or the Enemy's HP drops to 0 or below.

- Objective Function: The implicit objective of the Greedy AI is to win the battle by applying its local optimization strategy at each step, thereby maximizing its chances of victory based on turn-by-turn decisions.

C. *Mapping Turn-Based RPG Combat Elements to Dynamic Programming Elements*

The Dynamic Programming AI is implemented by modeling the entire combat scenario as a multi-stage optimization problem, directly aligning with DP theory. The abstract DP concepts are mapped to our simulation as follows:

- Stages: Each turn in the battle represents a single stage in the DP model. The problem progresses from stage k (the current turn) to stage k+1 (the next turn).
- States: A *state* is a unique snapshot of the battle, containing all information needed to make a perfect decision. For our simulation, a state is defined by the tuple: (pc_hp, pc_mana, enemy_hp, fireball_cooldown). This state representation is the key used for memoization.
- Decisions: At each stage, the decision to be made is which of the valid actions (Basic Attack, Fireball, or Meditate) to execute.
- Principle of Optimality: The implementation relies on the Principle of Optimality, which posits that the best sequence of moves from any state is composed of the best first move plus the best sequence of moves from the resulting state. This allows us to build the optimal solution from the end of the battle backward.
- Recursive Relationship: The core of the DP solution is the recurrence relation used to calculate the value of each state. The value is defined as the minimum number of turns required to win from that state. This is analogous to the shortest path problem in a multistage graph. The relation can be expressed as:

$$\text{TurnsToWin}(\text{State}) = 1 + \min(\text{TurnsToWin}(\text{NextState})) \quad (1)$$

D. *Greedy AI Agent Implementation*

The Greedy AI was implemented as a lightweight and straightforward agent. Its logic follows a simple two-step process each turn: score all possible moves, then pick the best one. The scoring is based entirely on the immediate heuristic value of each action.

The *chooseMove* method iterates through the available actions (Basic Attack, Fireball, Meditate), checks their feasibility based on the current mana and cooldowns, and assigns a score. The scoring heuristic was designed to aggressively prioritize damage: Fireball is assigned a score of 35, Basic Attack a score of 10, and Meditate a score of 1. The low score for Meditate ensures it is only ever chosen if no damaging actions are possible. The agent then simply executes the action with the highest score.

```java
public Action chooseMove(GameState state) {
    Action bestMove = null;
    int maxScore = -1;

    // Evaluate Fireball
    if (state.canUseFireball()) {
        if (35 > maxScore) {
            maxScore = 35;
            bestMove = Action.FIREBALL;
        }
    }

    // Evaluate Basic Attack
    if (10 > maxScore) {
        maxScore = 10;
        bestMove = Action.BASIC_ATTACK;
    }

    // Evaluate Meditate
    if (1 > maxScore) {
        maxScore = 1;
        bestMove = Action.MEDITATE;
    }

    return bestMove;
}
```

Fig. 2. Greedy AI Algorithm

E. *Dynamic Programming AI Implementation*

```java
private Outcome findOptimalOutcome(GameState state) {
    if (state.getEnemyHp() <= 0) return new Outcome(null, 0);
    if (state.getPcHp() <= 0) return new Outcome(null, 999);
    if (memoTable.containsKey(state)) return memoTable.get(state);

    Outcome bestOutcome = new Outcome(null, 999);

    // Recursive Exploration for each valid move
    if (state.canUseFireball()) {
        evaluateMove(state, Action.FIREBALL, bestOutcome);
    }
    evaluateMove(state, Action.BASIC_ATTACK, bestOutcome);
    evaluateMove(state, Action.MEDITATE, bestOutcome);

    memoTable.put(state, bestOutcome);
    return bestOutcome;
}
```

Fig. 3. Dynamic Programming AI Implementation

The Dynamic Programming AI was implemented as a far more complex agent that seeks a provably optimal solution. It uses a recursive function with memoization to find the shortest path to a win state.

The core of the implementation is a *Map<GameState, Outcome>* which serves as the memoization table. The GameState object, which implements *equals()* and *hashCode()* methods, acts as the key. The Outcome is a simple class storing the best move from that state and the minimum number of turns required to win.

The main recursive function, *findOptimalOutcome(state)*, first checks if the current state has already been solved by looking it up in the memoization table. If not, it explores every valid move from the current state, recursively calling itself on the resulting nextState. It compares the outcomes of these recursive calls (adding 1 to the turn count for the current move) and identifies the action that leads to victory in the fewest

turns. Before returning, it stores this optimal result in the memoization table to avoid re-computation in the future.

F. Experimental Design and Data Collection

To generate comprehensive data for our analysis, a rigorous experimental protocol was established. The goal was to test each AI's performance, efficiency, and strategic decision-making across a wide variety of combat scenarios specifically designed to highlight the differences between the Greedy and Dynamic Programming approaches.

A suite of eight distinct test cases was designed. These scenarios move beyond a simple baseline to create complex situations involving resource management, strategic planning, and efficiency under pressure. The initial conditions for each test case are detailed below:

1. Baseline: A standard encounter with no immediate constraints (PC: 100 HP, 50 Mana; Enemy: 150 HP, 12 Dmg).
2. Resource Scarcity: Tests the AI's ability to plan when powerful moves are not immediately available (PC: 100 HP, 15 Mana).
3. Race Against Time: Puts the AI under significant pressure from a standard enemy (PC: 70 HP, 50 Mana).
4. Durable Enemy: A longer battle against a less threatening foe, testing mana efficiency (Enemy: 120 HP, 8 Dmg).
5. Glass Cannon Enemy: A short, high-damage race where burst damage is optimal (Enemy: 70 HP, 25 Dmg).
6. Strategic Cooldown: Tests planning by making the best move unavailable on the first turn (PC: Fireball Cooldown = 1).
7. Mana Trap: A subtle resource puzzle where the AI is just one mana point short of a key ability (PC: 100 HP, 19 Mana).
8. Perfect Lethal: A scenario where the optimal path is simple and requires exact resource usage (PC: 100 HP, 40 Mana; Enemy: 70 HP).

The Java simulation was executed via a master script, the *ExperimentRunner*. The simulator was instrumented to automatically log the following key data points for every individual battle:

- Result: The outcome of the battle, recorded as a "Win" or a "Loss".
- Total Turns: The number of turns the battle lasted from start to finish.
- Decision Time: The computation time required for the AI to choose a move. This was measured in nanoseconds using Java's System.nanoTime() for high precision and later converted to milliseconds (ms) for analysis.

The results from all simulations were aggregated and written to a results.csv file, forming the basis for the quantitative analysis presented in the following chapter.

IV. Result and Analysis

A. Overall Performance Summary

The complete aggregated results from all simulations are presented in Table I. This table provides a high-level overview of each AI's performance across the test suite, serving as the foundation for the subsequent analysis.

TABLE I. AI PERFORMANCE ACROSS ALL TEST CASES

Test Case	AI Type	Result	Turns to End	Decision Time (ms)
Baseline	Greedy	Win	9	0.0002
	DP	Win	9	0.0075
Resource Scarcity	Greedy	Win	13	0.0002
	DP	Win	10	0.0049
Race Against Time	Greedy	Loss	6	0.0002
	DP	Win	6	0.0031
Durable Enemy	Greedy	Win	12	0.0002
	DP	Win	12	0.0053
Glass Cannon Enemy	Greedy	Win	3	0.0002
	DP	Win	3	0.0016
Strategic Cooldown	Greedy	Win	9	0.0001
	DP	Win	9	0.0039
Mana Trap	Greedy	Lose	13	0.0002
	DP	Win	10	0.0041
Perfect Lethal	Greedy	Win	4	0.0002
	DP	Win	4	0.0022

[a.] Decision time may be different for different devices

Several clear trends emerge from this data. First, the Dynamic Programming AI achieves a 100% win rate across all scenarios where victory is possible. The Greedy AI, in contrast, fails completely in two specific scenarios ("Race Against Time" and "Mana Trap"). Second, in cases where both AIs win, the DP AI consistently finds the solution in an equal or fewer number of turns. Finally, there is a stark and consistent difference in computational cost: the DP AI's average decision time is consistently one to two orders of magnitude greater than that of the nearly instantaneous Greedy AI.

B. Analysis of Strategic Capabilities

To understand why these performance differences occurred, a deeper analysis of the AI's decision-making in key scenarios is required. The test cases were specifically designed to expose the logical limitations of the Greedy approach and highlight the value of the DP algorithm's foresight.

The most telling results come from the "Resource Scarcity" and "Mana Trap" scenarios. In the "Mana Trap" case (PC starts with 19 Mana), the PC is just one mana point short of being able to cast its most powerful spell, Fireball. The Greedy AI's heuristic, which scores Basic Attack (10) higher than Meditate (1), forces it into a suboptimal loop. It repeatedly uses Basic Attack, never choosing to sacrifice a turn of minor damage to

gain the mana needed for a Fireball. This flawed logic leads to a 0% win rate.

The DP AI, however, demonstrates the power of foresight. By evaluating future states, its algorithm correctly identifies that the path to victory, although non-obvious, begins with the Meditate action. It understands that sacrificing one turn of damage to gain 15 mana unlocks a much faster and more powerful sequence of Fireball attacks later on. This single, strategic setup move is something the Greedy algorithm is fundamentally incapable of reasoning about, leading to the DP AI's win in 10 turns compared to the Greedy AI's guaranteed loss in 13 turns. A similar, though less catastrophic, outcome is observed in the "Resource Scarcity" scenario, where the DP AI finds the win three turns faster than the Greedy AI by using a similar lookahead strategy.

Not all scenarios require complex planning. In the "Glass Cannon Enemy" and "Perfect Lethal" test cases, the performance of the Greedy AI, in terms of turns-to-win, was identical to that of the DP AI. In the "Glass Cannon" scenario, the enemy is a high-damage threat that must be defeated as quickly as possible. Here, the Greedy AI's simple heuristic of "use the highest immediate damage move" aligns perfectly with the globally optimal strategy found by the DP AI. There is no need for a setup move or long-term planning; maximum immediate damage is the correct choice at every step.

This is a critical finding, as it demonstrates that a more complex algorithm is not universally superior. The effectiveness of a simple heuristic is highly dependent on the problem's context. For straightforward combat encounters where the optimal path is aggressive and direct, a Greedy algorithm can perform just as well as a more computationally expensive DP algorithm, achieving the same outcome with a fraction of the processing power.

C. Analysis of Computational Cost

While the DP AI's strategic superiority is clear, it comes at a significant and measurable cost. Decision Time metric in Table I quantifies the computational overhead required for the DP algorithm to find its optimal solution.

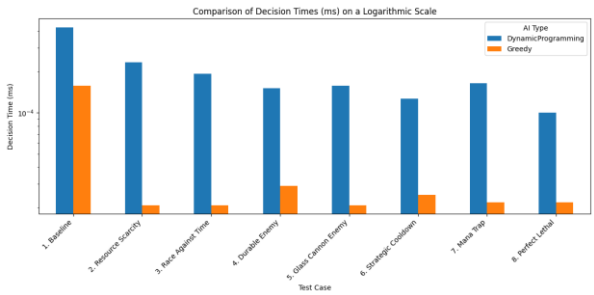

Fig. 4. Comparison of Decision Times (ms) on A Logarithmic Scale

As illustrated in Fig. 4, the computational resources required by the DP AI are consistently greater than those required by the Greedy AI. The Greedy AI's decision is nearly instantaneous (averaging ~0.0002 ms), as it involves only a simple loop and comparison. The DP AI, in contrast, must recursively explore a tree of potential game states. While memoization prevents this from becoming a full brute-force search, the process of generating states, checking the memoization table, and performing recursive calls still carries a substantial overhead (averaging ~0.005 ms). Although this time is trivial in our simple simulator, in a full-scale game with a vastly larger state space (more actions, status effects, etc.), this cost could scale to become a source of noticeable lag or performance degradation.

D. Discussion of Implications for Game Development

The results from this experiment confirm the well-understood theoretical trade-off between Greedy and Dynamic Programming algorithms and place it in a practical game development context. The DP AI is strategically perfect but computationally "expensive," while the Greedy AI is computationally "cheap" but strategically "brittle."

This data provides a clear framework for a game developer's design choices. A one-size-fits-all approach to AI is not optimal. For the hundreds of common, low-stakes enemies a player might encounter, a fast and efficient Greedy AI is more than sufficient. Its flaws are unlikely to be noticed in quick battles, and its low performance overhead allows for many such enemies to be active at once without impacting the game's frame rate.

For critical, memorable encounters, such as a major boss battle, a developer could justify the higher computational cost of a DP-style agent. In these fights, players expect a deep, strategic challenge. An AI that can make non-obvious, intelligent setup moves, the very behavior the DP AI demonstrated in the "Mana Trap" scenario, provides exactly this kind of engaging experience. Therefore, a hybrid approach, using cheap, simple AIs for the rank-and-file and reserving complex, optimal AIs for key moments, represents the most effective application of these findings.

V. Conclusion

This paper presented a quantitative comparative analysis of a Greedy algorithm and a Dynamic Programming approach for designing AI strategy in a controlled, turn-based RPG combat simulator. Through a series of eight distinct test cases, each AI is being evaluated on its strategic capability, combat effectiveness, and computational performance. The experimental results confirmed the initial hypothesis: Dynamic Programming AI, by exhaustively exploring the state space, consistently achieved a strategically optimal outcome, winning battles in the minimum possible number of turns. However, this optimality came at the cost of a significantly higher computational overhead. Conversely, the Greedy AI proved to be extremely fast and efficient but demonstrated critical strategic flaws in scenarios that required any degree of foresight, leading to suboptimal solutions or outright failure.

The primary contribution of this research is the generation of a clear, data-driven benchmark that illustrates the classic "speed vs. smarts" trade-off within the practical context of game AI development. By providing hard numbers on metrics like win rates, turn counts, and decision times across varied

scenarios, this paper offers a tangible reference for developers. However, the study has its limitations. The combat model was intentionally simplified, involving a single enemy and a limited set of actions. A full-scale RPG with multiple party members, a wide array of skills, and numerous status effects would create a state space with many orders of magnitude larger, which would likely make the pure Dynamic Programming approach used here computationally infeasible. Furthermore, this analysis was limited to two classic, deterministic algorithms and did not explore other modern AI paradigms.

Looking forward, this research opens several interesting avenues for future work. The most promising direction would be the development of a hybrid AI model that combines the strengths of both algorithms. One could design an agent that uses the fast Greedy logic for the majority of a battle but invokes a limited-depth DP search when the situation becomes critical (e.g., when health is low or a powerful enemy ability is about to be used). Another area for future research would be to expand the experiment to a more complex combat environment to analyze how these performance trade-offs scale. Finally, comparing these foundational algorithms against other techniques, such as Monte Carlo Tree Search or a simple Reinforcement Learning agent, would provide an even broader understanding of the AI strategy landscape.

VIDEO LINK AT YOUTUBE

Video Link: https://youtu.be/HqALZ6o6EkY

Program Source Code: https://github.com/kin-ark/Simple-Turn-Based-RPG-Combat-Simulator

ACKNOWLEDGMENT

The author expresses gratitude to all parties who have assisted in the making of this paper, especially to:

1. Allah Swt.

2. Both parents, for providing moral and material support.

3. Friends who have encouraged and aided in the completion of this paper.

4. Monterico Adrian, S.T., M.T. as the lecturers for the IF2211 Algorithm Strategy course, for his invaluable guidance and support throughout the semester.

The author deeply appreciates all the assistance, encouragement, and kindness received from these individuals and groups, without which the completion of this paper would not have been possible.

REFERENCES

[1] C. D. Stenström and S. Björk, "Understanding computer role-playing games - a genre analysis based on gameplay features in combat systems," in Proc. International Conference on Foundations of Digital Games, 2013. Available online: https://www.diva-portal.org/smash/get/diva2:1043334/FULLTEXT01.pdf (accessed on 22 June 2025).

[2] Y. Lu and W. Li, "Techniques and paradigms in modern game AI systems," Algorithms, vol. 15, no. 282, Aug. 2022. Available online: https://pdfs.semanticscholar.org/b7f9/89be140ac00c1fbd0a9cf60a62964e0d452a.pdf (accessed on 24 June 2025).

[3] F. Amereh, "A study and implementation of turn-based combat systems in role-playing games," M.S. thesis, Dept. Computer Science, Aalto University, Espoo, Finland, 2024. Available online: https://aaltodoc.aalto.fi/server/api/core/bitstreams/9eb57411-982e-4e64-ad57-377a5eb78f19/content (accessed on 22 June 2025)

[4] R. Munir, Algoritma Greedy (Bagian 1): Bahan Kuliah IF2211 Strategi Algoritma. Bandung, Indonesia: Program Studi Teknik Informatika, Institut Teknologi Bandung, 2025. Available online: https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2024-2025/04-Algoritma-Greedy-(2025)-Bag1.pdf (accessed on 24 June 2025)

[5] R. Munir, Algoritma Greedy (Bagian 2): Bahan Kuliah IF2211 Strategi Algoritma. Bandung, Indonesia: Program Studi Teknik Informatika, Institut Teknologi Bandung, 2025. Available online: https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2024-2025/05-Algoritma-Greedy-(2025)-Bag2.pdf (accessed on 24 June 2025)

[6] R. Munir, Algoritma Greedy (Bagian 3): Bahan Kuliah IF2211 Strategi Algoritma. Bandung, Indonesia: Program Studi Teknik Informatika, Institut Teknologi Bandung, 2025. Available online: https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2024-2025/06-Algoritma-Greedy-(2025)-Bag3.pdf (accessed on 24 June 2025)

[7] R. Munir, Program Dinamis (Bagian 1): Bahan Kuliah IF2211 Strategi Algoritma. Bandung, Indonesia: Program Studi Teknik Informatika, Institut Teknologi Bandung, 2025. Available online: https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2024-2025/25-Program-Dinamis-(2025)-Bagian1.pdf (accessed on 24 June 2025)

[8] R. Munir, Program Dinamis (Bagian 2): Bahan Kuliah IF2211 Strategi Algoritma. Bandung, Indonesia: Program Studi Teknik Informatika, Institut Teknologi Bandung, 2025. Available online: https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2024-2025/26-Program-Dinamis-(2025)-Bagian2.pdf (accessed on 24 June 2025)

STATEMENT

I hereby declare that this paper I have written is my own work, not an adaptation or translation of someone else's paper, and not plagiarism.

Bandung, 24 June 2025

Muhammad Kinan Arkansyaddad
13523152