Optimal Decision Strategies in a Roguelike Game using Dynamic Programming

David Bakti Lodianto - 13523083
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jalan Ganesha 10 Bandung
E-mail: david.lodianto@gmail.com , 13523083@std.stei.itb.ac.id

***Abstract*—This paper presents a analysis of optimization in a simplified roguelike-style game where a player progresses through a sequence of floors, making choices between combat, resource gathering, and stat upgrades. The goal is to maximize the gold accumulated by the end of the final floor while managing health and attack stats. We model this problem using a dynamic programming approach, defining a multi-dimensional state space to represent player status. The DP explores all possible decision paths efficiently and identifies the optimal strategy. Experiment is done on a 5 to 15-floor configuration, which results reveal trade-offs between aggression, saving, and survival.**

***Keywords*—dynamic programming; optimization; roguelike**

I. INTRODUCTION (*HEADING 1*)

As of late, a certain genre of single-player games has exploded in popularity, known as roguelikes or rogue-lites. This is due to its simplistic design, and high replayability. After thorough observation, these games consistently highlight a few key elements: strategic planning, resource management, and survival under uncertainty. The player will often journey through a series of procedurally generated levels, and each step forces the player with choices, either fight, save resources, or spend them to get stronger.

This constant push and pull creates the core fun of roguelikes. Games like Slay the Spire, FTL, and the recent hit Balatro are perfect examples. The player will be forced to constantly balance what helps right now, short-term survival or with what benefits in the long run. Spending it early on an upgrade might make a current fight much easier. But that same gold could have been saved to unlock something more powerful later, or it might count towards the final score. On the other hand, hoarding everything might seem smart for future challenges, but it could also leave the player too weak to handle what's directly ahead, leading to an early game over.

From its tricky blend of choices, this gives birth to a fascinating optimization puzzle. A big reason these games are so popular is the sheer number of ways you can play to try and get the highest score or the best outcome. This deep strategic element, combined with levels that are always changing, makes roguelikes a great subject for deeper study. So, this paper will try to create a simpler, more predictable version of a roguelike. This will make it suitable for formal algorithmic analysis, specifically with dynamic programming techniques to determine optimal resource management strategies.

II. THEORETICAL FOUNDATION

A. Dynamic Programming

Dynamic Programming (DP) is a problem-solving strategy designed to efficiently solve optimization problems by breaking them into smaller overlapping sub-problems. The core idea is to store intermediate results to avoid redundant computation and to build up solutions to larger problems from those sub-solutions.

The word "programming" refers to mathematical planning and optimization, not computer code, while "dynamic" refers to the step-by-step update of solutions as decisions evolve across stages. It is a fundamental technique used in operations research and algorithmic problem-solving. Unlike greedy algorithms, which make locally optimal decisions at each step, DP will evaluate multiple paths or states, comparing them to find a globally optimal solution.

This makes DP algorithms superior to greedy algorithms in solving optimization problems where the solution isn't always locally optimal. Problems such as 0/1 Knapsack or TSP can't be solved optimally with greedy, but solvable with DP. However, this does come at a cost, as it needs to store the previous computations done before and explore other possibilities, where if the problem is NP-hard, it can be highly costly in both time and memory complexity.

1. The Principle of Optimality

   The foundation of dynamic programming rests on the Principle of Optimality:

   If the total solution is optimal, then any partial solution up to stage k must also be optimal.[1]

   This principle enables us to work from stage *k* to stage *k+1* using optimal results from stage *k* without needing to recalculate from the beginning. The cost relationship can be expressed as follows:

$$C_{K+1} = C_K + C_{K \to (K+1)} \quad (1)$$

where C_K is the cost at stage k.

2. Suitable Problem Characteristics

A problem can be solved using dynamic programming if it exhibits these characteristics:

- **Multi-stage Structure**
  The problem can be divided into several stages, with one decision made at each stage
- **State Definition**
  Each stage consists of multiple states representing various possible inputs at that stage
- **State Transformation**
  Decisions at each stage transform the current state to the next state in the following stage
- **Cumulative Cost**
  Costs increase steadily as the number of stages increases
- **Cost Dependency**
  Cost at any stage depends on costs from previous stages plus the transition cost
- **Recursive Relationship**
  A recursive relationship exists that identifies the best decision for each state at stage k, leading to the best decision for each state at stage k+1
- **Optimality Principle**
  The principle of optimality applies to the problem

3. Two Approaches

There are two approaches in solving a DP problem:

- **Forward Dynamic Programming (Top-Down)**
  Build solutions progressively from the initial state to the goal. This is useful when all start states are known, and we explore all possible outcomes. This is also known as tabulation.
- **Backward Dynamic Programming (Bottom-Up)**
  Start from the goal and recursively determine which previous states could lead to the optimal result. Often paired with *memoization*, which is a technique where results are stored to avoid doing the same computations many times.

Here is a table that details their differences.

TABLE I. TOP-DOWN VS. BOTTOM-UP DP [2]

	Top-Down	**Bottom-Up**
State	State transition relation is relatively difficult to think	State Transition relation is easier to think
Speed	Fast, as we do not have recursion call overhead.	Slow due to a lot of recursive calls.
Subproblem solving	If all subproblems must be solved at least once, a bottom-up dynamic programming algorithm definitely outperforms a top-down memoized algorithm by a constant factor	If some subproblems in the subproblem space need not be solved at all, the memoized solution has the advantage of solving only those subproblems that are required
Table entries	In the Tabulated version, starting from the first entry, all entries are filled one by one	Unlike the Tabulated version, all entries of the lookup table are not necessarily filled in Memoized version. The table is filled on demand.

B. *Roguelike Game Elements*

Roguelike games, in a formal context, represent a class of stage-limited sequential decision problems. These games are composed of discrete decision points (e.g., game stages or floors) where the player must make strategic choices that affect future outcomes. While traditionally randomized or procedurally generated, they can be abstracted as deterministic models to study decision optimization under constraints.

1. Stateful Sequential Decisions

   Each game state is defined by a set of quantifiable variables (e.g., health, resources, combat ability), and transitions between states occur based on player actions. This setup resembles a finite state space, where:

   - States represent the player's current condition
   - Actions transform one state into another
   - The game progresses linearly, forming a multi-stage decision process.

   Although most roguelike games will have randomly generated levels or resources, we restrict our formulation to a deterministic setting where state transitions and rewards are known in advance.

2. Resource-Constrained Optimization

   Players manage a limited set of resources (such as health, currency, etc.), and their choices influence both short-term gain and long-term survival. This aligns with well-known algorithmic problems, such as knapsack-type problems, multi-stage scheduling, and path-dependent planning.

3. Deterministic Decision Graph

   Each player's action will lead to a new, deterministic game state, which can be represented as a node in a directed graph of possible outcomes. This formulation lends itself to graph traversal algorithms and dynamic programming over state transitions.

a. State Space Modeling

   State space modeling is the formal representation of all possible configurations or conditions the system can occupy. The player's state will be defined by a tuple of key variables, such as the player's resources and current stage level. Each unique combination of these attributes represents a distinct state in the game. This finite, discrete, and bounded representation is fundamental to enabling dynamic programming: it ensures that the number of reachable states is computationally tractable and that transitions between states can be explicitly enumerated. The entire decision-making process can thus be viewed as movement through a well-defined state space.

b. Action Space Analysis

   At each decision point or state, the player is presented with a limited set of available actions. These may include fighting an enemy, taking a safe treasure reward, or visiting a shop to trade gold for improvements. This discrete and deterministic action space defines the branching possibilities from any given state. The decision at each step determines how the current state transitions to a new state in the next stage. From a theoretical standpoint, the action space directly influences the shape of the decision tree and the size of the problem's solution space. Because the actions are limited and deterministic, the full space of outcomes can be systematically explored, making the problem suitable for exhaustive methods like dynamic programming.

c. Cumulative Reward Functions

   A key characteristic of this decision problem is that rewards accumulate over time, depending on the player's choices. The objective of optimization is to maximize the final gold after all decisions are made. This introduces a cumulative reward function, where the total utility of a decision sequence is the sum of rewards obtained at each stage. This additive structure is essential to dynamic programming, as it enables the use of recursive relationships to evaluate and compare partial solutions. The ability to associate value with each transition allows the algorithm to propagate optimality forward across stages, ensuring that local decisions contribute meaningfully to the global objective.

d. Terminal Conditions

   Terminal conditions define when the decision process ends, either because the game has reached its natural conclusion (such as defeating the boss, reaching the final floor), or because the player can no longer continue (HP $\leq$ 0). These conditions form the boundaries of the state space and help determine the feasibility of decision sequences. Any state where the player has no remaining health is considered invalid and excluded from further computation. On the other hand, reaching the final stage with any positive HP leads to an evaluation of the accumulated reward. These terminal rules are vital in constraining the optimization problem and ensuring that only viable paths through the game state graph are considered. They also support pruning during computation, improving both efficiency and correctness.

III. Implementation

In this paper's implementation, we will try to recreate a simplified deterministic rogue-like game.

A. Core Game Mechanics

As mentioned previously, the current implementation adopts a simplified roguelike model, in which all game mechanics and outcomes are deterministic. This simplification enables the use of dynamic programming (DP) techniques to exhaustively and efficiently explore the space of possible player decisions across multiple game stages (or floors). Each decision leads to a predictable outcome, allowing the algorithm to precisely evaluate the long-term consequences of each action.

The core gameplay is structured into a series of floors, each representing a discrete decision point. At every floor, the player can choose one of three actions:

1. Enter Combat
   Engaging in combat against a floor-specific enemy involves turn-based exchanges of damage. The player always attacks first, followed by the enemy if it survives. Combat is resolved deterministically, and the result affects the player's HP. If the player wins, they are rewarded with a predetermined amount of gold.
2. Collect Treasure
   The player can opt to take a safe, flat gold reward without engaging in combat. This path ensures no loss of HP but yields lower long-term potential for improvement.
3. Enter Shop
   The player may spend gold on upgrades, including restoring HP or increasing attack power (ATK). This introduces trade-offs between short-term resource reduction and long-term survival or strength gains.

Each action transforms the player's current state into a new state, defined by the tuple:

(floor, HP, ATK, Gold).

These state transitions form the basis of the DP recurrence.

B. Constraints

To maintain computational feasibility and adhere to the structure of a typical stage-limited decision problem, the following constraints are enforced:

1. Finite Floors
   The game progresses through a fixed number of stages (e.g., 5 floors), after which the process terminates.

2. Bounded State Variables
   - HP has both a minimum (HP ≥ 0) and a maximum (HP ≤ 100).
   - ATK and Gold are also capped to a certain upper bound to prevent unbounded state growth.

3. Deterministic Transitions
   All state updates are predictable and depend solely on the current state and selected action. There is no randomness in outcomes.

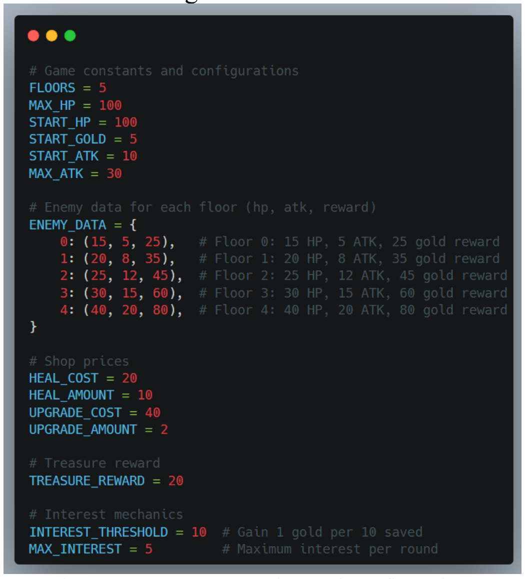
4. Terminal States
   Any state where HP ≤ 0 is considered invalid and excluded from further transitions. Only states with strictly positive HP are carried forward to the next floor.

5. Interest Mechanic
   To encourage resource-saving strategies, an interest bonus is applied at the end of each floor, where players gain +1 gold for every 10 gold saved (up to a cap of +5 per round). This mechanic adds a strategic layer encouraging gold conservation.

C. Source Code

The following program implementation is done using Python.

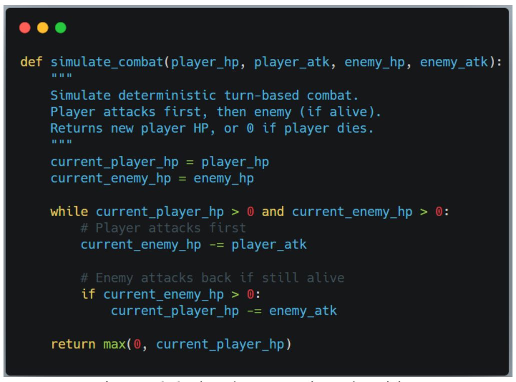
1. Constraints & Configurations



```python
# Game constants and configurations
FLOORS = 5
MAX_HP = 100
START_HP = 100
START_GOLD = 5
START_ATK = 10
MAX_ATK = 30

# Enemy data for each floor (hp, atk, reward)
ENEMY_DATA = {
    0: (15, 5, 25),    # Floor 0: 15 HP, 5 ATK, 25 gold reward
    1: (20, 8, 35),    # Floor 1: 20 HP, 8 ATK, 35 gold reward
    2: (25, 12, 45),   # Floor 2: 25 HP, 12 ATK, 45 gold reward
    3: (30, 15, 60),   # Floor 3: 30 HP, 15 ATK, 60 gold reward
    4: (40, 20, 80),   # Floor 4: 40 HP, 20 ATK, 80 gold reward
}

# Shop prices
HEAL_COST = 20
HEAL_AMOUNT = 10
UPGRADE_COST = 40
UPGRADE_AMOUNT = 2

# Treasure reward
TREASURE_REWARD = 20

# Interest mechanics
INTEREST_THRESHOLD = 10  # Gain 1 gold per 10 saved
MAX_INTEREST = 5         # Maximum interest per round
```

Picture 3.1. Game Constraints and Configurations

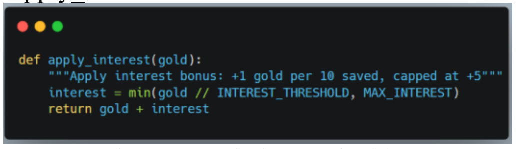
2. Algorithms
   a. simulate_combat



```python
def simulate_combat(player_hp, player_atk, enemy_hp, enemy_atk):
    """
    Simulate deterministic turn-based combat.
    Player attacks first, then enemy (if alive).
    Returns new player HP, or 0 if player dies.
    """
    current_player_hp = player_hp
    current_enemy_hp = enemy_hp

    while current_player_hp > 0 and current_enemy_hp > 0:
        # Player attacks first
        current_enemy_hp -= player_atk

        # Enemy attacks back if still alive
        if current_enemy_hp > 0:
            current_player_hp -= enemy_atk

    return max(0, current_player_hp)
```

Picture 3.2 simulate_combat algorithm

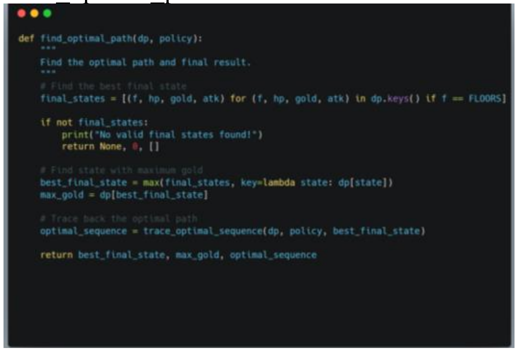
   b. apply_interest



```python
def apply_interest(gold):
    """Apply interest bonus: +1 gold per 10 saved, capped at +5"""
    interest = min(gold // INTEREST_THRESHOLD, MAX_INTEREST)
    return gold + interest
```

Picture 3.3 apply_interest algorithm

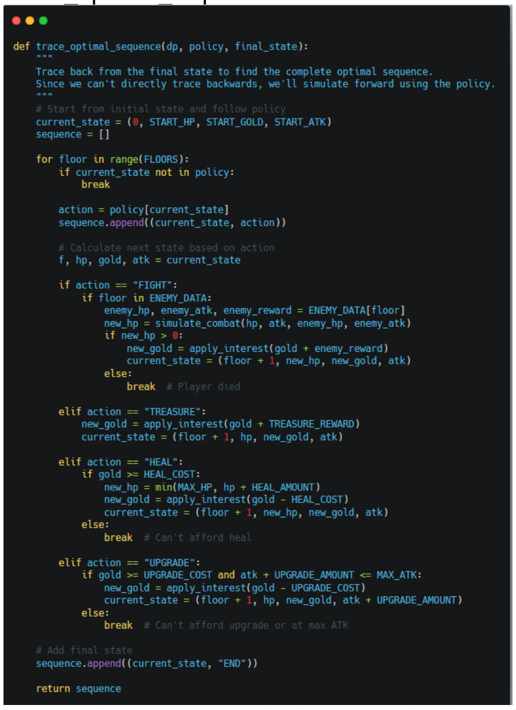
   c. find_optimal_path



```python
def find_optimal_path(dp, policy):
    """
    Find the optimal path and final result.
    """
    # Find the best final state
    final_states = [(f, hp, gold, atk) for (f, hp, gold, atk) in dp.keys() if f == FLOORS]

    if not final_states:
        print("No valid final states found!")
        return None, 0, []

    # Find state with maximum gold
    best_final_state = max(final_states, key=lambda state: dp[state])
    max_gold = dp[best_final_state]

    # Trace back the optimal path
    optimal_sequence = trace_optimal_sequence(dp, policy, best_final_state)

    return best_final_state, max_gold, optimal_sequence
```

Picture 3.4 find_optimal_path algorithm

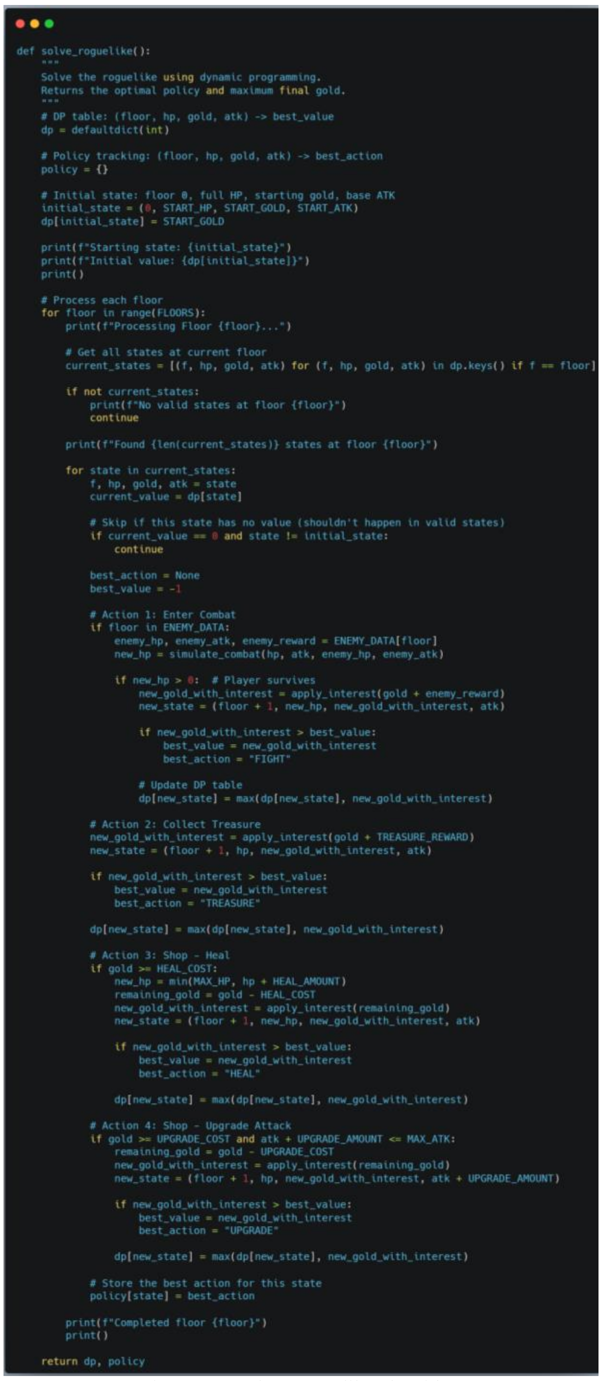
   d. trace_optimal_sequence



```python
def trace_optimal_sequence(dp, policy, final_state):
    """
    Trace back from the final state to find the complete optimal sequence.
    Since we can't directly trace backwards, we'll simulate forward using the policy.
    """
    # Start from initial state and follow policy
    current_state = (0, START_HP, START_GOLD, START_ATK)
    sequence = []

    for floor in range(FLOORS):
        if current_state not in policy:
            break

        action = policy[current_state]
        sequence.append((current_state, action))

        # Calculate next state based on action
        f, hp, gold, atk = current_state

        if action == "FIGHT":
            if floor in ENEMY_DATA:
                enemy_hp, enemy_atk, enemy_reward = ENEMY_DATA[floor]
                new_hp = simulate_combat(hp, atk, enemy_hp, enemy_atk)
                if new_hp > 0:
                    new_gold = apply_interest(gold + enemy_reward)
                    current_state = (floor + 1, new_hp, new_gold, atk)
                else:
                    break  # Player died

        elif action == "TREASURE":
            new_gold = apply_interest(gold + TREASURE_REWARD)
            current_state = (floor + 1, hp, new_gold, atk)

        elif action == "HEAL":
            if gold >= HEAL_COST:
                new_hp = min(MAX_HP, hp + HEAL_AMOUNT)
                new_gold = apply_interest(gold - HEAL_COST)
                current_state = (floor + 1, new_hp, new_gold, atk)
            else:
                break  # Can't afford heal

        elif action == "UPGRADE":
            if gold >= UPGRADE_COST and atk + UPGRADE_AMOUNT <= MAX_ATK:
                new_gold = apply_interest(gold - UPGRADE_COST)
                current_state = (floor + 1, hp, new_gold, atk + UPGRADE_AMOUNT)
            else:
                break  # Can't afford upgrade or at max ATK

    # Add final state
    sequence.append((current_state, "END"))

    return sequence
```

Picture 3.5 trace_optimal_sequence algorithm

   e. solve_roguelike



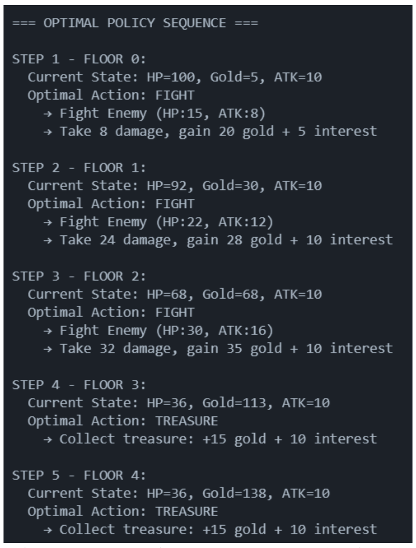
```python
def solve_roguelike():
    """
    Solve the roguelike using dynamic programming.
    Returns the optimal policy and maximum final gold.
    """
    # DP table: (floor, hp, gold, atk) -> best_value
    dp = defaultdict(int)

    # Policy tracking: (floor, hp, gold, atk) -> best_action
    policy = {}

    # Initial state: floor 0, full HP, starting gold, base ATK
    initial_state = (0, START_HP, START_GOLD, START_ATK)
    dp[initial_state] = START_GOLD

    print(f"Starting state: {initial_state}")
    print(f"Initial value: {dp[initial_state]}")
    print()

    # Process each floor
    for floor in range(FLOORS):
        print(f"Processing Floor {floor}...")

        # Get all states at current floor
        current_states = [(f, hp, gold, atk) for (f, hp, gold, atk) in dp.keys() if f == floor]

        if not current_states:
            print(f"No valid states at floor {floor}")
            continue

        print(f"Found {len(current_states)} states at floor {floor}")

        for state in current_states:
            f, hp, gold, atk = state
            current_value = dp[state]

            # Skip if this state has no value (shouldn't happen in valid states)
            if current_value == 0 and state != initial_state:
                continue

            best_action = None
            best_value = -1

            # Action 1: Enter Combat
            if floor in ENEMY_DATA:
                enemy_hp, enemy_atk, enemy_reward = ENEMY_DATA[floor]
                new_hp = simulate_combat(hp, atk, enemy_hp, enemy_atk)

                if new_hp > 0:  # Player survives
                    new_gold_with_interest = apply_interest(gold + enemy_reward)
                    new_state = (floor + 1, new_hp, new_gold_with_interest, atk)

                    if new_gold_with_interest > best_value:
                        best_value = new_gold_with_interest
                        best_action = "FIGHT"

                    # Update DP table
                    dp[new_state] = max(dp[new_state], new_gold_with_interest)

            # Action 2: Collect Treasure
            new_gold_with_interest = apply_interest(gold + TREASURE_REWARD)
            new_state = (floor + 1, hp, new_gold_with_interest, atk)

            if new_gold_with_interest > best_value:
                best_value = new_gold_with_interest
                best_action = "TREASURE"

            dp[new_state] = max(dp[new_state], new_gold_with_interest)

            # Action 3: Shop - Heal
            if gold >= HEAL_COST:
                new_hp = min(MAX_HP, hp + HEAL_AMOUNT)
                remaining_gold = gold - HEAL_COST
                new_gold_with_interest = apply_interest(remaining_gold)
                new_state = (floor + 1, new_hp, new_gold_with_interest, atk)

                if new_gold_with_interest > best_value:
                    best_value = new_gold_with_interest
                    best_action = "HEAL"

                dp[new_state] = max(dp[new_state], new_gold_with_interest)

            # Action 4: Shop - Upgrade Attack
            if gold >= UPGRADE_COST and atk + UPGRADE_AMOUNT <= MAX_ATK:
                remaining_gold = gold - UPGRADE_COST
                new_gold_with_interest = apply_interest(remaining_gold)
                new_state = (floor + 1, hp, new_gold_with_interest, atk + UPGRADE_AMOUNT)

                if new_gold_with_interest > best_value:
                    best_value = new_gold_with_interest
                    best_action = "UPGRADE"

                dp[new_state] = max(dp[new_state], new_gold_with_interest)

            # Store the best action for this state
            policy[state] = best_action

        print(f"Completed floor {floor}")
        print()

    return dp, policy
```

Picture 3.6 solve_roguelike algorithm

IV. Result and Analysis

A. Experimental Setup

We evaluated our dynamic programming algorithm on several procedurally generated dungeon layouts ranging from 5 to 15 floors. Each floor contained a deterministic event: battle, shop, or healing. The initial player state is defined by a tuple *(floor, hp, gold, atk)*. We ran the algorithm from a fixed initial state to simulate a complete dungeon run and collect all reachable states and their associated optimal values.



```
=== OPTIMAL POLICY SEQUENCE ===

STEP 1 - FLOOR 0:
  Current State: HP=100, Gold=5, ATK=10
  Optimal Action: FIGHT
    → Fight Enemy (HP:15, ATK:8)
    → Take 8 damage, gain 20 gold + 5 interest

STEP 2 - FLOOR 1:
  Current State: HP=92, Gold=30, ATK=10
  Optimal Action: FIGHT
    → Fight Enemy (HP:22, ATK:12)
    → Take 24 damage, gain 28 gold + 10 interest

STEP 3 - FLOOR 2:
  Current State: HP=68, Gold=68, ATK=10
  Optimal Action: FIGHT
    → Fight Enemy (HP:30, ATK:16)
    → Take 32 damage, gain 35 gold + 10 interest

STEP 4 - FLOOR 3:
  Current State: HP=36, Gold=113, ATK=10
  Optimal Action: TREASURE
    → Collect treasure: +15 gold + 10 interest

STEP 5 - FLOOR 4:
  Current State: HP=36, Gold=138, ATK=10
  Optimal Action: TREASURE
    → Collect treasure: +15 gold + 10 interest
```

Picture 4.1 Sample Program Output (5 Floors)

B. State Space Growth

The number of unique states grew exponentially with the number of floors due to branching choices (e.g., shop purchases, whether to use potions, fight or flee).

TABLE II. State Space Growth Table

Floors	*Unique States Explored*
0	1
1	2
2	7
3	23
4	84
5	226
6	529
7	1085
8	1914
9	3078
10	4651
11	6631
12	8813
13	10868
14	12603
15	13921

This growth is mitigated by state pruning, where dominated states, or those worse in all dimensions than others, are discarded at each step.

C. Optimal Strategy Patterns

Analysis of optimal strategies revealed several consistent patterns, such as:

1. Delayed Spending
   The algorithm often chooses to save gold rather than spending early in shops, especially when later shops offer better value (e.g., healing or rare items).

2. Aggressive Trading
   When HP is high, the strategy chooses aggressive options (e.g., battles) to maximize score rather than play conservatively.

D. Limitations and Edge Cases

In rare cases, multiple paths lead to the same final state. This implementation favors the lexicographically smaller path, which may not always align with player intuition. Furthermore, we need to take note that although determinism may simplify analysis, this makes it ignore randomness, which comes in real typical roguelikes. Introducing randomness would require probabilistic DP.

V. Conclusion

In this paper, we presented a dynamic programming approach to solve a simplified, deterministic roguelike game. By modeling the game as a stage-limited decision process, we constructed a state space that stores all relevant player attributes, which are as follows: health; gold; attack power; and computed optimal strategies through exhaustive value propagation.

Our implementation ensures that decision-making is not driven by immediate reward alone, but by the maximization of long-term outcomes, such as survivability and final resource value. By integrating components like shop actions, combat simulation, and treasure restrictions into the state, we demonstrated that strategic planning across multiple floors can yield significantly better results than greedy or myopic heuristics.

The results show that the dynamic programming algorithm naturally discovers human-like strategies such as delaying purchases, prioritizing survivability when health is low and optimizing when to collect one-time rewards.

Acknowledgment

I would like to acknowledge and express my gratitude to the Algorithm Strategies lecturers, Dr. Ir. Rinaldi Munir, M.T. and Dr. Nur Ulfa Maulidevi, S.T., M.Sc. for all of their efforts to teach me and my peers throughout the semester, which contributed a lot on making this paper possible. Their lectures and tasks given has greatly boosted my understanding of the subject matter, especially the topic I have chosen, dynamic programming.

I am also grateful to my peers and colleagues in the who provided constructive feedback and support during the research and writing process.

References

[1] R. Munir, Program Dinamis (Dynamic Programming) Bagian 1. [Online]. Available: https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2024-2025/25-Program-Dinamis-(2025)-Bagian1.pdf/

[2] GeeksforGeeks, "Tabulation vs Memoization," GeeksforGeeks. [Online]. Available: https://www.geeksforgeeks.org/dsa/tabulation-vs-memoization/

[3] Puterman, M. L. (2005). Markov Decision Processes: Discrete Stochastic Dynamic Programming. John Wiley & Sons.

[4] Martello, S., & Toth, P. (1990). Knapsack Problems: Algorithms and Computer Implementations. Wiley-Interscience.R. Nicole, "Title of paper with only first word capitalized," J. Name Stand. Abbrev.

[5] Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). Introduction to Algorithms (3rd ed.). MIT Press.

PERNYATAAN

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

Bandung, 1 Juni 2025

[illegible]

David Bakti Lodianto 13523083