The Feasibility of Applying a Backtracking Algorithm for Finding Every Winning Condition in Pokémon's Double Battle Format

Benedict Darrel Setiawan - 13524057
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jalan Ganesha 10 Bandung
E-mail: benedictdarrel572006@gmail.com , 13524057@std.stei.itb.ac.id

***Abstract*— Being the highest-grossing media franchise of all time, Pokémon's battling mechanics are deeper and more complex than they seem on the surface. This is more prevalent in the Double Battles Format, where each turn can have hundreds of possible outcomes. This paper aims to answer the question of whether it is feasible to find a winning condition through exploring all known choices and possibilities using a Backtracking algorithm.**
.

Keywords* — *Pokémon, Backtracking, Algorithms, Pokémon VGC, NP Problem

I. Introduction

Fig 1. Pokémon Champions banner (Source: https://bulbapedia.bulbagarden.net/wiki/Pok%C3%A9mon_Champions)

Pokémon Champions is an online-only side series Pokémon game for the Nintendo Switch, iOS, iPadOS, and Android. It allows players to battle using the same mechanics as the core series games, including types, Abilities, and moves. The game was released worldwide on April 8, 2026, on the Nintendo Switch, and on June 17, 2026, for mobile devices. The game's conception was to be the main hub for competitive Pokémon battling, featuring two main formats, Single Battle 3v3 and Double Battle 4v4, also mostly known as VGC.

At first glance, Pokémon might just be about clicking the strongest moves against your opponent. However, behind every move comes a set of unique effects that might be impactful to the battle, even if the move isn't very strong. And behind every Pokémon comes a set of unique abilities that may help in the face of a disadvantage. Not to mention the type chart that defines the interactions, be it offensive or defensive, between each of Pokémon's 18 types.

This is more apparent in the Double Battle format or also known more popularly as VGC (Video Game Championships), as it is the format that is played in official Pokémon tournaments. In this format, it is expected that the games will last a shorter number of turns compared to Single Battles. However, the complexity of a single turn is beyond that of the average Single Battle. It is akin to chess with fewer pawns and a smaller board size, but with a higher variance in the decisions that can be made each turn.

This already high variance of the Double Battles format is exemplified by the randomness that is guaranteed to be crucial to every battle. Factors such as critical hits, move secondary effects, misses, and speed ties can sometimes be the make-or-break to winning a match.

This paper will answer the question of whether winning a game of Pokémon's VGC format is feasible by exploring all known choices and possibilities. This will be done by using a backtracking algorithm to create a tree whose nodes define each state of a turn in a battle.

II. Theoretical Framework

A. Backtracking Algorithm

Backtracking is a problem-solving algorithmic technique that aims to find a solution recursively and return to a previous state if a dead end is found. It starts in a certain state and explores every possible state until a certain solution is found or every state has been explored already. The steps to backtracking are as follows:

1. Choose an initial state
2. Explore the next possible state recursively in a depth-first order
3. If a dead end is found, return to the previous state and explore the other states

4. Repeat 2-3 until a solution is found or all states are explored.

Backtracking algorithms are usually visualized as a decision tree, as they can easily depict the backtracking action.

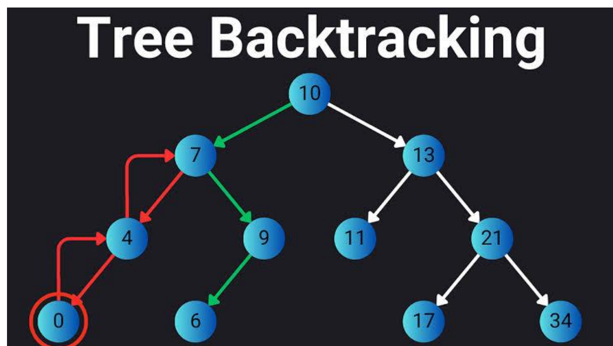

Fig 2. Backtracking Decision Tree (Source: https://levelup.gitconnected.com/backtracking-in-binary-trees-solving-pathfinding-problems-339d9dcc2592?gi=855a7fc2e583)

There are three main components in a backtracking algorithm, which are as follows

1. Solution set

$$X = (x_1, x_2, ..., x_n)$$

A solution set is an n-tuple that defines the solution that the algorithm is searching for.

2. Candidate Generator

$$T(x_1, x_2, ..., x_k)$$

A function that generates solution candidates for a certain state in the algorithm.

3. Bounding function

$$B(x_1, x_2, ..., x_k)$$

A function that decides if a state is considered a dead end. This component is the main driver of backtracking in the algorithm

B. Complexity Classes

In computer science, every known problem is assigned to classes based on complexity, known as Complexity Classes. These classes differentiate the level of time complexity is needed to solve a problem. The 4 main Complexity Classes are as follows

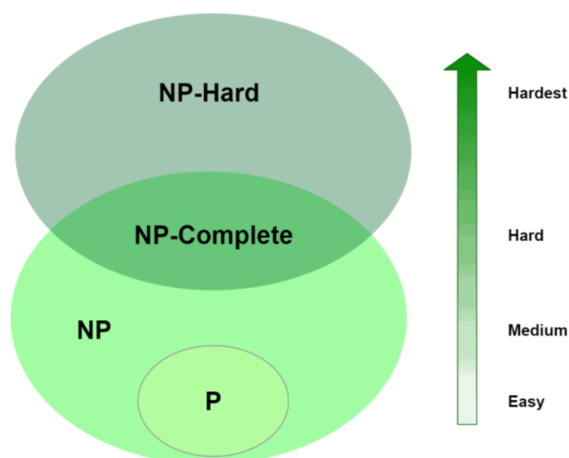

Fig 3. Diagram of the 4 Main Complexity Classes (Source: https://www.baeldung.com/cs/p-np-np-complete-np-hard)

1. P Class
   P problems are defined as decision problems that can be solved by deterministic machines in polynomial time. P is often a class of computational problems that are solvable and tractable, which means that the problems can be solved in theory as well as in practice.

2. NP Class
   NP problems are defined as decision problems that can be solved by a nondeterministic machine in polynomial time. This means that a solution to a problem in the NP class can be verified in polynomial time. However, finding said solution cannot be done in polynomial time, as it is solved by a nondeterministic machine.

3. NP-Hard Class
   NP-Hard problems are defined as problems that are at least as hard as the hardest problems in NP. Solutions to problems in the NP-Hard class tend not to be able to be found or verified in polynomial time.

4. NP-Complete Class
   NP-Complete problems are defined as the intersection between NP and NP-Hard, as they are the hardest problems in NP. If there exists an algorithm that can solve an NP-Complete problem in polynomial time, then an NP problem that can be reduced to said NP-Complete problem can also be solved in polynomial time.

C. Pokémon Battling

1. Using moves

Every Pokémon in a battle can have up to 4 moves that it can choose from. Each move has its own type, damage, accuracy, priority, and effects. Each turn, every Pokémon in the field gets to move in an order that is determined by the Pokémon's speed stats as well as the used moves' priority. If a Pokémon faints before it gets its turn, the move will not be done.

There are a few mechanics that can stop a Pokémon from using a move or stop it from moving in general. Some of which are as follows

- Taunt: Stops the afflicted user from using status moves;
- Flinch: Prevents the user from moving during the turn it flinched
- Paralysis: A chance to stop a Pokémon afflicted with it from moving
- Sleep: A status that keeps the afflicted Pokémon from moving until it wakes up
- Recharging moves: Stops a Pokémon from moving the next turn the move is used;
- Encore: Prevents the user from using a move other than the one it got encored into
- Choice Items: Prevents the user from using a move other than the first move it used after entering the field

2. Randomness

In Pokémon battling, randomness is a constant, as almost every attack has some kind of variance to it, making each turn in a battle rarely be the same as the last. Some causes of variance in a turn of a battle are as follows

- Damage roll: Each damaging move roll between a two floating-point decimal of 0.85 and 1
- Critical hit: Each damaging move has a chance to deal a critical hit, which will deal 1.5 times more damage and ignore all stat changes that reduce the move's damage
- Accuracy: If a move has an accuracy of below 100% then it has a chance to miss and do nothing for that turn.
- Consecutive Protects: If a Pokémon uses the move Protect or any moves similar to it consecutively, it will have a chance to fail, with it increasing by each time it succeeds
- Status conditions: Some status conditions, like Sleep, Paralysis, and Freeze, rely on randomness to activate their effects
- Move effects: Some moves have secondary effects that have a chance to do something other than damaging the opponent

III. Method

A. Defining the Decision Tree

To create the backtracking algorithm, the definition of the decision tree needs to be determined first. It could be defined that each node represents a single turn in the match. However, this will lead to complicating the visualization. Hence, to simplify, each node will be defined as a single decision that one player makes or a variance that happens in the middle of a turn.

However, the occurrence of a random effect is only decided after a turn is started. Meaning that all 4 Pokémon have to decide their moves first before the variances are decided in the turn order. So, the tree will decidedly have two different types of nodes, Action Decision Nodes, which define the decision of an action that a Pokémon makes, and Turn Order Action Nodes, which define the occurrences that will happen in said turn order action (status activation, critical hit, move effect, etc).

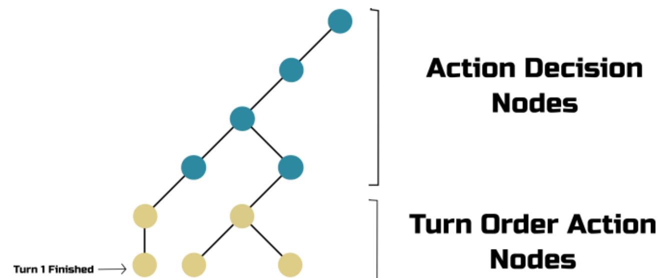

Fig 4. Defined Decision Tree

B. Determining Turn Order Variance

To determine every single variance that can happen in a certain turn order, every single cause of variance would need to be determined. While every random effect may have different probabilities, the result will always be binary, whether it occurred or it didn't. Meaning that for every turn order, the number of possible child nodes is as follows

$$p(n) = 2^n$$

With n being the number of variance causes. However, there is a possibility that a Pokémon can be knocked out before doing its turn order. This will cause the following child nodes to be redundant, as they will have the same result since no action is happening in that turn order. So there needs to be an extra condition to prevent this possibility from affecting the algorithm's efficiency

$$p(n) = \begin{cases} 2^n & \text{if } P \text{ is still alive,} \\ 1 & \text{else} \end{cases}$$

With n being the number of variance causes and P being the would-be acting Pokemon. With this, the random seed generator can be made as in the following code snippet

```
public class RNGSeedGenerator {
  public static List<RNGSeed> generateSeeds
    (BattleState state,
    int userSlot,
    int userTeamIdx,
    int moveSlot,
    boolean targetOpposingTeam,
    int targetSlot) {
    PokemonBattleState user = state.getTeam(userTeamIdx)
                .getPokemonOnSlot(userSlot);
    PokemonBattleState target = targetOpposingTeam ?
              state.getOpposingTeam(userTeamIdx)
              .getPokemonOnSlot(targetSlot) :
              state.getTeam(userTeamIdx)
              .getPokemonOnSlot(targetSlot);
    MoveState move = user.getMoves().get(moveSlot);
    int seedLength = 0;
    List<String> log = new ArrayList<>();

    for (Object chance :
      user.getStatusesWithProperty
      (RNGDependent.class)) {
      if (chance instanceof Immobilizing) {
        if (((Immobilizing)chance).immobilizedChance() < 1) {
          seedLength++;
          log.add(user.getPokemon().getName() +
          " immobilized by " +
           chance.getClass().getSimpleName());
        } else {
          seedLength++;
          log.add(user.getPokemon().getName() +
          ((RNGDependent)chance).effectLog());
        }
      }
    }
    MoveAction action = new MoveAction(userSlot,
            moveSlot,
            targetOpposingTeam,
            targetSlot);
    if (move.getAccuracy(user, target) < 100) {
      for (PokemonBattleState targetCandidates : action
        .getMoveActionTargets(state.getTeam(userTeamIdx),
        state.getOpposingTeam(userTeamIdx))) {
        seedLength++;
        log.add(user.getPokemon().getName() +
        "'s " + move.getMove().name() +
        " missed on " +
        targetCandidates.getPokemon().getName());
      }
    }

    if (move.getMove().getEffect() != null) {
      for (PokemonBattleState targetCandidates :
        action.getMoveActionTargets(state.getTeam(userTeamIdx),
        state.getOpposingTeam(userTeamIdx))) {
        seedLength += move.getMove().getEffect()
              .rngDependentEvents();
        for (String effectLog : move.getMove().getEffect()
          .rngDependentEventsLog()) {
          log.add(targetCandidates.getPokemon().getName() +
          effectLog);
        }
      }

    }

    List<RNGSeed> seedStrings = new ArrayList<>();
    getAllRNGSeeds(seedStrings, "", seedLength, log);
    return seedStrings;
  }

  private static void getAllRNGSeeds(List<RNGSeed> seeds,
              String seed,
              int seedLength,
              List<String> seedLog) {
    if (seed.length() == seedLength) {
      seeds.add(new RNGSeed(seed, seedLog));
    } else {
      getAllRNGSeeds(seeds, seed + "0", seedLength, seedLog);
      getAllRNGSeeds(seeds, seed + "1", seedLength, seedLog);
    }
  }
}
```

C. Determining Action Candidates

In a Pokémon battle, there are two main actions that a player can choose from, which are using a move and switching out to another Pokémon in their party. From here, it can be inferred that the action candidates consist of moves that can be done by the player in the current turn.

In the case of choosing a move, a Pokemon can choose said move if :

1. The amount of PP the move has is not 0
2. The Pokémon that has said move is not restricted to choose it (Taunt, Encore, Torment, etc)

In the case of switching into another Pokémon, a Pokémon can do such a switch if:

1. The Pokémon being switched into is still alive
2. The Pokémon being switched into is not already on the field
3. The Pokémon being switched from is not affected by a trapping status (Shadow Tag, Bind, Mean Look, etc)

A Pokémon can choose an action in general if :

1. The Pokémon isn't in a state that automatically chooses its own moves (Outrage, Uproar, Recharging moves, etc)
2. The Pokémon is still alive
3. If a Pokémon no longer has a move with any PP left and cannot switch to any other Pokémon. (Forces into Struggle)

```
public class MoveAction extends Action {
    private int userSlot;
    private int moveSlot;
    private boolean targetOpposingTeam;
    private int targetSlot;

    public MoveAction(int userSlot,
                    int moveSlot,
                    boolean targetOpposingTeam,
                    int targetSlot) {
```

```
        this.userSlot = userSlot;
        this.moveSlot = moveSlot;
        this.targetOpposingTeam = targetOpposingTeam;
        this.targetSlot = targetSlot;
    }

    @Override
    public boolean canDoAction(
        TeamState team,
        TeamState opposingTeam) {
        Move move = team.getPokemonOnSlot(userSlot)
                        .getMoves()
                        .get(moveSlot)
                        .getMove();
        for (Object restriction :
            team.getPokemonOnSlot(userSlot)
                .getStatusesWithProperty
                (RestrictsMoveChoice.class)) {
            if (!((RestrictsMoveChoice) restriction)
                .canUseMove(
                    move,
                    team.getPokemonOnSlot(userSlot))) {
                return false;
            }
        }
        if (!team.getPokemonOnSlot(userSlot).isAlive()) {
            return false;
        }

        if (team.getPokemonOnSlot(userSlot)
                .getMoves()
                .get(moveSlot)
                .getPP() <= 0) {
            return false;
        }

        if (team.getPokemonOnSlot(userSlot)
            .getMoves().get(moveSlot)
            .getMove() == Move.FAKE_OUT &&
            team.getPokemonOnSlot(userSlot)
            .getLastAction() != null) {
            return false;
        }

        if (team.getPokemonOnSlot(userSlot)
            .getMoves()
            .get(moveSlot)
            .getMove() == Move.PROTECT &&
            team.getPokemonOnSlot(userSlot)
            .getConsecutiveProtects() > 0) {
            return false;
        }

        switch(move.getTarget()) {
            case SINGLE:
                return targetOpposingTeam;
            case ADJACENT:
            case ALL:
                return targetOpposingTeam && targetSlot == userSlot;
            case FIELD:
            case TEAM:
            case SELF:
                return !targetOpposingTeam && targetSlot == userSlot;
            case ALLY:
                return !targetOpposingTeam && targetSlot != userSlot;
            default:
                return false;
        }


    }
}

public class SwitchAction extends Action {
    private int switchFromSlot;
    private int switchToSlot;

    public SwitchAction(int fromSlot, int toSlot) {
        switchFromSlot = fromSlot;
        switchToSlot = toSlot;
    }

    @Override
    public boolean canDoAction(
                    TeamState team,
                    TeamState opposingTeam) {
        return !team.getPokemonOnSlot(switchFromSlot)
                    .hasStatusWithProperty(CannotSwitch.class) &&
               !team.getPokemonOnSlot(switchFromSlot)
                    .hasStatusWithProperty(SkipsTurnChoice.class) &&
                team.getPokemonOnSlot(switchToSlot).isAlive() &&
                switchFromSlot != switchToSlot &&
                (switchToSlot != 1 || switchToSlot != 0);
    }
}
```

```
public class ActionManager {
    public static List<Action> getPossibleActions(BattleState state,
                                                  int userSlot,
                                                  int userTeamID) {
        PokemonBattleState pokemon = state
                                    .getTeam(userTeamID)
                                    .getPokemonOnSlot(userSlot);
        List<Action> actions = new ArrayList<>();
        if (!pokemon.isAlive()) return actions;
        for (int moveSlot = 0;
            moveSlot < pokemon.getMoves().size();
            moveSlot++) {
            for (int teamID = 0; teamID < 2; teamID++) {
                for (int targetSlot = 0;
                    targetSlot < 2;
                    targetSlot++) {
                    MoveAction moveAction =
                        new MoveAction(
                        userSlot,
                        moveSlot,
                        teamID != userTeamID,
                        targetSlot);
                    if (moveAction.canDoAction
                        (state.getTeam(userTeamID),
                        state.getOpposingTeam(userTeamID))) {
                        actions.add(moveAction);
                    }
                }
            }
        }
        for (int pokemonSlot = 0;
            pokemonSlot < pokemon.getTeam().getAllPokemon().size();
            pokemonSlot++) {
            SwitchAction switchAction =
                new SwitchAction(userSlot, pokemonSlot);
            if (switchAction.canDoAction(state.getTeam(userTeamID),
                state.getOpposingTeam(userTeamID))) {
                actions.add(switchAction);
            }
        }
        if (actions.size() == 0) {
            for (int targetSlot = 0; targetSlot < 2; targetSlot++) {
                    StruggleAction moveAction =
                        new StruggleAction(userSlot, targetSlot);
                    if (moveAction.canDoAction
                        (state.getTeam(userTeamID),
                        state.getOpposingTeam(userTeamID))) {
                        actions.add(moveAction);
                    }
            }
        }

        return actions;
    }
}
```

D. *Defining the Backtracking Algorithm*

To fully define a backtracking algorithm, the solution set, candidate generator, and the bounding function need to be defined. For this problem, it can be defined that the ordered string of actions and occurrences that happen in the battle is the solution set, or can be seen as follows

$$X = (x_{1,1}, x_{1,2}, \dots, y_{1,1} \dots, x_{mn}, \dots, y_{mn})$$

$$x = \{move \mid move \in M(P)\} \cup \{switch \mid switch \in SW(P)\}$$

$$y = \text{variance seed}$$

$$M(P) = \text{Set of moves that Pokemon } P \text{ has}$$

$$SW(P) = \text{Set of switches that Pokemon } P \text{ can do}$$

Next, the candidate generator can be defined by combining the previously made turn order variance generator function and the action candidate generator function. Hence, the following function definition

$$T(x_{1,1}, x_{mn}, ..., y_{mn}) = \begin{cases} A(x_{1,1}, x_{mn}, ..., y_{mn}) & \text{if in Action Decision phase,} \\ V(x_{1,1}, x_{mn}, ..., y_{mn}) & \text{if in Turn Order phase} \end{cases}$$

Finally, the bounding function of the algorithm is simply defined as a function that checks if the selected team of Pokémon has lost over the other, which is defined as having zero Pokémon alive on the team. This means that the algorithm will stop the search for the solution if said condition is true.

$$B(x_{1,1}, x_{mn}, ..., y_{mn}, Team) = \begin{cases} False & \text{if } Team \text{ has no Pokemon alive,} \\ True & \text{else} \end{cases}$$

IV. TESTING

A. Testing Limitations & Specification

Due to the exponentially high variance of the game, some limitations need to be made to visualize a successful result in testing. Firstly, the test will only go so far as one turn, which means it will find the possibilities of the selected team winning in one turn. To make sure this is possible, the test will also only use 2 Pokémon per team, as it is impossible for either side to win in one turn if both had more than 2 Pokémon due to the nature of the game.

For the specific Pokémon that will be used in this test, they are as follows.

Incineroar @ Sitrus Berry
Ability: Intimidate
Level: 50
EVs: 32 HP / 32 Atk / 2 Def
Adamant Nature
- Flare Blitz
- Fake Out
- Throat Chop
- Close Combat

Garchomp @ Choice Scarf
Ability: Rough Skin
Level: 50
EVs: 2 HP / 32 Atk / 32 Spe
Jolly Nature
- Dragon Claw
- Earthquake
- Rock Slide
- Iron Head

Sneasler @ Mental Herb
Ability: Unburden
Level: 50
EVs: 2 HP / 32 Atk / 32 Spe
Adamant Nature
- Dire Claw
- Fake Out
- Close Combat
- Protect

Kingambit @ Focus Sash
Ability: Defiant
Level: 50
EVs: 32 HP / 32 Atk / 2 Def
Adamant Nature
- Iron Head
- Kowtow Cleave
- Sucker Punch
- Protect

B. Testing Results

After running the algorithm on the battle between Incineroar and Garchomp vs Kingambit and Sneasler, it resulted in 80 outcomes where Incineroar and Garchomp wins in the first turn of the battle.

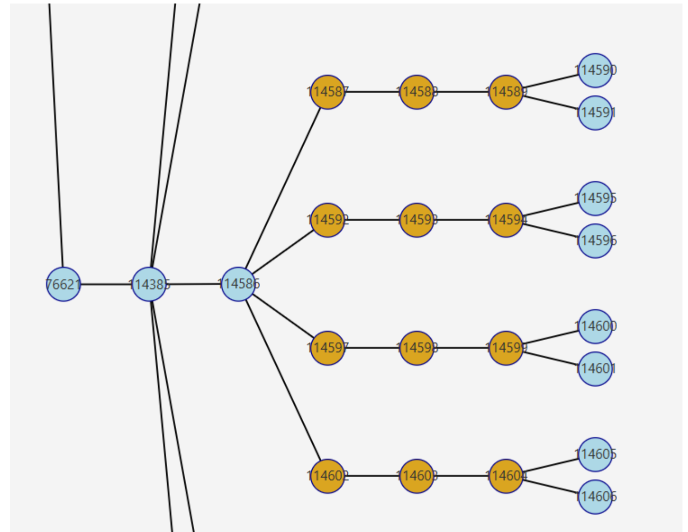

Fig 5. Snippet of the visualized Pruned Decision Tree

This result confirms that the algorithm is working as intended, as the solutions have both Sneasler and Kingambit on 0 HP left. To prove this even further, the path to one of the solutions is as follows.

1. Incineroar chooses Heat Wave on Kingambit and Sneasler
2. Garchomp chooses Earthquake on Kingambit, Sneasler, and Incineroar
3. Sneasler chooses Close Combat on Garchomp
4. Kingambit chooses Sucker Punch on Garchomp
5. Kingambit hits Garchomp with Sucker Punch
6. Garchomp hits Kingambit, Sneasler, and Incineroar with Earthquake and KOs Sneasler
7. Since Sneasler is KOed, Incineroar moves next
8. Incineroar uses Heat Wave on Kingambit and KOs it

From the scenario above, it is proven that the sequence of events lines up with the result, which is that Incineroar and Garchomp win.

C. Discussion

While the test results provably validates the algorithm, it can still only calculate up to the first turn. When a maximum depth of two is inserted as the parameter, the program is still trying to find solutions even when tens of thousands have already been found. This most likely means that the algorithm needs an exponential amount of time to find every solution.

This would mean that Pokémon Double Battles are, by nature, an NP problem because the algorithm has no trouble verifying the solution in polynomial time. Hence, to solve Pokémon Double Battles in polynomial time, there needs to be an NP-Complete problem that it can be reduced to. The easiest reduction would be to reduce it to a k-SAT problem, as it will only require further extending the decision tree into a binary tree. However, to this day, there is no algorithm to solve k-SAT in polynomial time, which in turn means that Pokémon Double Battles is most likely also currently unsolvable in polynomial time.

V. CONCLUSION

By using a backtracking algorithm designed to find every possible winning condition in Pokémon Double Battle, the author has surmised that the approach is unfeasible for calculating turns beyond the first, as the game has too high a variance. However, this result has led to the conclusion that finding every winning condition in a game of Pokémon Double Battles is an NP problem that is most likely could only be solved by an algorithm that can solve the k-SAT problem, which, at the time of writing, doesn't currently exist.

This conclusion only further proves the underlying depth and complexity of Pokémon, as this research did not account for the possibility of different Pokémon builds in a closed team-sheet environment or even further gimmicks like Mega Evolution and Terastalization.

VI. APPENDIX

GitHub repository for the source code made for this research paper:
https://github.com/BenedictD-RH/Total-Pathing-Application-of-Backtracking-for-Pokemon-VGC-Turn-Planning

Explanation of the research paper in video form: https://youtu.be/_mXurL1mjI4

VII. ACKNOWLEDGEMENT

First of all, the author would like to give his deepest gratitude to the Almighty God for giving him the health and capacity to finish this paper and this semester. The author would also like to thank Mr. Rinaldi Munir for his guidance and teachings on Algorithm Strategies, which provided the foundations for the concepts used in this paper.

VIII. REFERENCE

[1] R. Munir, "Algoritma Runut-balik (backtracking) - Bagian 1" Informatika STEI ITB, Bandung, Indonesia. [Online]. Available: https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2025-2026/15-Algoritma-backtracking-(2026)-Bagian1.pdf. [Accessed: Jun. 14, 2026].

[2] GeeksForGeeks, "Backtracking Algorithm" GeeksForGeeks, Sanchaya Education Private Limited. [Online]. Available: https://www.geeksforgeeks.org/dsa/backtracking-algorithms/. [Accessed: Jun. 14, 2026].

[3] GeeksForGeeks, "P, NP, CoNP, NP hard and NP complete | Complexity Classes" GeeksForGeeks, Sanchaya Education Private Limited. [Online]. Available: https://www.geeksforgeeks.org/dsa/types-of-complexity-classes-p-np-conp-np-hard-and-np-complete/. [Accessed: Jun 15, 2026].

[4] Baeldung, "P, NP, NP-Complete and NP-Hard Problems in Computer Science", Baeldung. [Online]. Available: https://www.baeldung.com/cs/p-np-np-complete-np-hard. [Accessed: Jun 17, 2026]

STATEMENT

Hereby, I declare that this paper I have written is my own work, not a reproduction or translation of someone else's paper, and not plagiarized.

Bandung, 19 June 2026

Benedict Darrel Setiwan
13524057