An Optimized Algorithm for Analyzing Counter-Strike 2 Match-Winning Probabilities Using Dynamic Programming Approach to Recurrence Relations

Muhammad Zaky Amani - 13525040
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jalan Ganesha 10 Bandung
E-mail: mzakyrice@gmail.com , 13525040@std.stei.itb.ac.id

***Abstract*— Counter-Strike 2 is a First-Person Shooter game that features a complex tactical economic system where the outcome of each round dynamically influences future financial capacities, making match-winning probabilities highly interdependent. This paper models these match dynamics into a conditional recursive function utilizing discrete state spaces and recurrence relations. While a naive recursive approach successfully maps the state transitions, it suffers from an intractable exponential time complexity due to massive overlapping subproblems within the match state hierarchy. To overcome this limitation, the algorithm is restructured using a top-down Dynamic Programming paradigm with memoization. By encapsulating match parameters into a unique composite tuple key and utilizing an associative lookup map, the program effectively prunes redundant execution branches. Empirical runtime evaluations demonstrate that the optimized approach successfully restricts the computational upper bound to the cardinality of the valid state space, reducing the execution time from thousands of milliseconds to tens of milliseconds. This drastic efficiency improvement confirms the practicality of discrete mathematical modeling for analyzing tactical game probabilities.**

Keywords—Counter-Strike 2; Match-Winning Probability; Recurrence Relations; Time Complexity; Dynamic Programming; Memoization

I. Introduction

Counter-Strike 2 (CS2) is a 5 vs 5 tactical first-person shooter game that does not only rely on aiming skills and strategy, but also economic. The economic system in CS2 restrics players to buy some weapons and utilities with their limited money. Every round, each team gets additional money based on their previous round (win or lose). If a team has enough money to buy some worthy weapons, then they can buy it on that round (Full Buy) but if they don't have then they must save their money on that round (Full Eco). It means, except for the last round, every round affects the next round. Therefore, we can use recurrence relations to calculate the match-winning probability of an ongoing match based on the economic and round differentials.

Calculating the probability manually will take such a long time. To calculate the probability faster, we will build an algorithm to calculate it. We can build the algorithm to brute force all the possibilities to calculate probability. But brute force algorithm is such an inefficient due to its time complexity.

In this paper, we will be modelling match probabilities into a recursive function and analyzing its computational efficiency using a C++ program that we will build. At the end, we will rebuild the program using dynamic programming approaches to optimize the algorithm. Therefore, the intention of this paper is to build an optimized algorithm for analyzing the match-winning probabilities in CS2.

II. Theoretical Background

Predictive probability modeling in a tactical game environment requires a strong integration between discrete mathematics theory and algorithm design analysis. A solid theoretical foundation is established by mapping the match dynamics into a state space structure, where each transition is calculated sequentially using conditional recurrence relation functions. To evaluate the feasibility of this recurrence relation, the asymptotic time complexity is analyzed to identify the efficiency bounds of a naive recursive computation. This theoretical background concludes by introducing Dynamic Programming principles and memoization techniques as a formal solution to reduce the exponential computational load into an efficient algorithm.

A. Basic Concepts of Probability in State Space

A CS 2 Match can be modeled as a series of discrete state. In the context of discrete mathematical modelling, a match isn't a random event but rather a journey in a finite state space. Every completion of a round represents the state change from the previous round to the next round until reaching the final round.

In this model, state is defined as a tuple that contains all the parameters that needed to calculate match-winning probability. A state is represented as S:

$$S = (S_A, S_B, M_A, M_B)$$

with S_A and S_B is the total score of team A and team B on that moment ($0 \leq S_A, S_B \leq 16$), M_A and M_B is the total money of team A and team B ($0 \leq M_A, M_B \leq 80000$).

State space is a universal set that contains all the possibility of state S that valid. In this context, valid means legal based on the economic and scoring system in CS2. The transition of a state to another state will be executed until it hits the terminal state, the base case when the match end. The terminal state reached if it satisfies one of these conditions:

- There is a team that win by having total score of 13 before the overtime round (the other team have total score of less than 12)
- There is a team that win by having total score of 16 while the other team have less than 15.
- Both team have total score of 15 and the match ends tie.

From a state S_i, we can transition into state S_{i+1} through two mutually exclusive events, that is the scenario that team A win the round and the scenario that team B win the round. The transition probability in this case isn't uniform, but rather based on the economic condition of each team on that state.

B. *Sequences, Summations, and Recurrence Relations*

Analytical approach to calculate the probability of match-winning in CS2 requires dividing a big problem into small subproblems. It can be mathematically modelled using sequences principle, summation, and recurrence relations.

A CS2 match operates as a sequence of rounds status. Each completion of a round n will be resulting in a transition state that forms the next term in the sequence of rounds status, represented as S_n. Because every transition S_n to S_{n+1} is counted step by step until reach the win, lose, or tie condition, all of these transitions form a sequential structure.

Match-winning probability (cumulative) of a state isn't counted using permutation, but rather using a recursive function, a function that call itself. Let P(S) is a Probability function of a team winning the match from a state $S = (S_A, S_B, M_A, M_B)$ then the value of P(S) can be formulized as recurrence relation based on win probability of that round:

$$P(S) = (p_w \cdot P(S_{win})) + (p_l \cdot P(S_{lose}))$$

which p_w is the probability of winning that round (based on the M_A and M_B comparation, p_l is the probability of losing that round which is 1-p_w, S_{win} and S_{lose} are state in the future after win or losing that round with additional score (S_A+1 or S_B+1) and economic changes (M_A and M_B).

In order to make the calculation didn't trapped in an infinite loop, those formulation should be bounded by the base case which is equal to 1 when the team is already win the match and equal to 0 when the team is already lose the match:

$$P(S) = \begin{cases} 1, & \text{Condition 1} \\ 0, & \text{Condition 2} \\ P(S) = (p_w \cdot P(S_{win})) + (p_l \cdot P(S_{lose})), & \text{Other Condition} \end{cases}$$

where Condition 1: $(S_A = 13 \wedge S_B < 12) \vee S_A = 16$
Condition 2: $(S_B = 13 \wedge S_A < 12) \vee S_B = 16 \vee (S_A = 15 \wedge S_B = 15)$

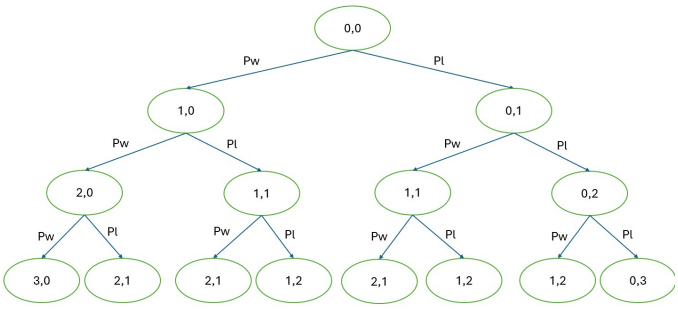

Fig. 1. Decision Tree Representation of Round Possibilities

Conceptually, the execution of this recursive function will form a decision tree. The total of match-winning probability of a team from score (0,0) is actually a summation of all probability route that ends on a leaf node that valued 1 (winning route). Here is the summation notation:

$$P_{total} = \sum_{k=1}^{N} \left(\prod_{i=1}^{m_k} p_{k,i} \right)$$

which N is total of possible winning route, m_k is total of round that played in route k, and $p_{k,i}$ is probability of winning or losing on round i in route k.

C. *Algorithmic Complexity*

To evaluate the efficiency of an algorithm, we have to analyze the time complexity of algorithm. Time Complexity is a notion that quantifies how time execution needs grow as input size (N) increases. This growth rate standardly defined as Big-O Notation that gives the upper bound number of basic operations which is executed by an algortihm on its worst case scenario.

In a recursive function algorithm, the main problem is divided into smaller subproblems. If a recursive function call itself c times without memoization, this algorithm will form a tree with exponential time complexity, notated as $O(c^N)$.

Specifically, binary recursion (c = 2) have $O(2^N)$ time complexity. This time complexity is intractable due to its high execution time growth. In this case, input size (N) represents the maximum possible rounds left.

D. *Dynamic Programming and Memoization*

To overcome the exponential asymptotic limitations inherent in pure recursive functions, the algorithm is restructured using the Dynamic Programming (DP) paradigm. Dynamic Programming is an algorithm optimization technique that solves a problem by dividing them down into smaller

subproblems, storing their solutions, and reusing these results when the exact same subproblems are encountered in subsequent computational processes.

A computational problem can be optimized using the DP paradigm if it satisfies two main prerequisites:

- The problem has an optimal substructure
- The problem has overlapping subproblems

Overlapping subproblems indicate a condition where a set of identical subproblems is repeatedly called across various branches of the execution hierarchy.

In recursive modeling, DP optimization is implemented using a top-down approach through a memory anchoring technique known as memoization. Memoization operates by integrating a lookup table (such as an array) into the function's flow. Before a function executes its computation, it checks the memory table with a constant time complexity of O(1). If a solution for that specific state is already available, the previous value will be used without re-executing the recurrence chain. This mechanism effectively avoids the program to process the same subproblems.

In the context of this CS2 match probability modeling, the phenomenon of overlapping subproblems occurs massively. A specific state is highly likely to be reached through various permutations of different round-win sequences. Without DP optimization, the program would recalculate the transition subproblems of that exact same state point multiple times across different branches of the recursive tree.

Through the application of memoization, the five parameters defining the match state are encapsulated into a unique key structured as a tuple. Every time the transition probability calculation is fully executed for a newly discovered state tuple, its value is instantly registered into the global memory table. This memory intervention successfully transforms the traversal architecture structure, which originally takes the form of a binary search tree, into a Directed Acyclic Graph (DAG). This reduction prunes the growth of basic operations from an exponential complexity of $O(2^N)$ into a polynomially bounded limit, ensuring that the computation can be executed with optimal space and time efficiency.

III. Methodology

The transformation of the theoretical recurrence relations into a functional computational system is conducted through a structured programming approach using C++. The design phase begins by initializing the financial parameters and match economic assumptions, which are calibrated based on real analytical data. The mathematical formulation is then translated into a recursive function to map all the state transition decision branches. As a final step to overcome the execution time growth limitations inherent in pure recursion, the code architecture is thoroughly optimized using a tuple-based associative memory mapping structure, known as memoization.

A. Parameter Initialization and Assumptions

This computation experiment is modelled using C++. Before evaluating recurrence relations, program will initialize constant parameter that represent CS2 economic state. Financial boundaries defined as constant variable maxMoney=80000 (each player maximum money is $16000, so each team total maximum money is $16000 * 5 = $80000), the total money of the first round and 13th round (first round of the second half) defined as constant variable initMoney = 4000 (each player starts with $800 so each team total money starts with $800 * 5 = $4000), and the total money of the first overtime round (round 25th and round 28th) defined as a constant variable initOTMoney = 50000 (each player have $10000 so each team have $50000). The Full Buy is assumed to be constant, which is $24000 (each player spend $4800 for M4A4/M4A1, kevlar + helmet, defusal kit, flashbang, and smoke grenade) for Counter-Terrorist team and $21000 (each player spends $4200 for AK47, kevlar + helmet, flashbang, and smoke grenade) for Terrorist team.

To avoid state space explosion because of the huge total permutation of player that survived until the end of the round, this model applies a simplification using static expected value. The win bonus and loss bonus are assumed to be constant values, winBonus = 16250 and lossBonus = 12000. This model also assumes that there are 2 players in a round-winner that survived and no one in a round-loser that survived. This model use the probability of round-winning and round-losing based on HLTV analytics result, which the ratio of CT team win and T team win is 96% : 4% for an unbalanced economic round (Full Buy vs Full Eco), the ratio of T team win and CT team win is 96% : 4% for an unbalanced economic round (Full Buy vs Full Eco), and the ratio of CT team win and T team win is 48.8% : 51.2% for a balanced economy round (Full Buy vs Full Buy or Full Eco vs Full Eco) [7].

B. Recurrence Model Formulation into Algorithm

The state space is implemented into a recursive function calculate_win_probability that receive 5 parameters: (bool team, int moneyA, int moneyB, int scoreA, int scoreB), boolean variable team is a variable that contain which team that we are calculating the match-winning probability (1 for team A, 0 for team B), moneyA and moneyB are the total money of each team on that round, and scoreA and scoreB are the total score of each team on that round. Execution of the function will always start by evaluating the base case. If the function detects a base case, which is ((scoreA == 15) && (scoreB == 15)), (((scoreA == 13) && (scoreB < 12)) || (scoreA == 16)), or (((scoreB == 13) && (scoreA < 12)) || (scoreB == 16)), the algorithm will instantly stop the recursive and return the absolute probability value (1 or 0).

If the base case hasn't reached yet, the function will evaluate the economic state of each team (moneyA and moneyB) that already adjusted to the side switch and overtime rules. This economic comparation will be used to decide the transition probability, then recursively call the function itself to simulate every win and lose scenario in the future.

Therefore, the algorithm of recursive function calculate_win_probability is defined as follows:

```
double calculate_win_probability(bool team, int moneyA, int
moneyB, int scoreA, int scoreB){
     // Recurrence Base (if the game ends tie)
    if((scoreA == 15) && (scoreB == 15)){
        return 0;
    }
     // Recurrence Base (if the game ends on A win the game)
    if(((scoreA == 13) && (scoreB < 12)) || (scoreA == 16)){
        return team;
    }
// Recurrence Base (if the game ends on B win the game)
    if(((scoreB == 13) && (scoreA < 12)) || (scoreB == 16)){
        return !team;
    }

    if(scoreA + scoreB == 12){
        moneyA = initMoney;
        moneyB = initMoney;
    }

    if((scoreA + scoreB == 24) || (scoreA + scoreB == 27)){
        moneyA = initOTMoney;
        moneyB = initOTMoney;
    }

    if(moneyA > maxMoney){
        moneyA = maxMoney;
    }
    if(moneyB > maxMoney){
        moneyB = maxMoney;
    }

    if((scoreA + scoreB < 12) || ((scoreA + scoreB >= 24) &&
(scoreA + scoreB < 27))){
        if((moneyA >= CTFullBuy) && (moneyB >= TFullBuy)){
            return 0.488 * calculate_win_probability(team,
moneyA - (CTFullBuy/5)*3 + winBonus, moneyB - TFullBuy +
lossBonus, scoreA + 1, scoreB) + 0.512 *
calculate_win_probability(team, moneyA - CTFullBuy +
lossBonus, moneyB - (TFullBuy/5)*3 + winBonus, scoreA, scoreB
+ 1);
        }else if((moneyA >= CTFullBuy) && (moneyB <
TFullBuy)){
            return 0.96 * calculate_win_probability(team,
moneyA - (CTFullBuy/5)*3 + winBonus, moneyB + lossBonus,
scoreA + 1, scoreB) + 0.04 * calculate_win_probability(team,
moneyA - CTFullBuy + lossBonus, moneyB + winBonus, scoreA,
scoreB + 1);
        }else if((moneyA < CTFullBuy) && (moneyB >=
TFullBuy)){
            return 0.04 * calculate_win_probability(team,
moneyA + winBonus, moneyB - TFullBuy + lossBonus, scoreA + 1,
scoreB) + 0.96 * calculate_win_probability(team, moneyA +
lossBonus, moneyB - (TFullBuy/5)*3 + winBonus, scoreA, scoreB
+ 1);
        }else{
            return 0.488 * calculate_win_probability(team,
moneyA + winBonus, moneyB + lossBonus, scoreA + 1, scoreB) +
0.512 * calculate_win_probability(team, moneyA + lossBonus,
moneyB  + winBonus, scoreA, scoreB + 1);
        }
    }else{
        if((moneyA >= TFullBuy) && (moneyB >= CTFullBuy)){
            return 0.512 * calculate_win_probability(team,
moneyA - (TFullBuy/5)*3 + winBonus, moneyB - CTFullBuy +
lossBonus, scoreA + 1, scoreB) + 0.488 *
calculate_win_probability(team, moneyA - TFullBuy + lossBonus,
moneyB - (CTFullBuy/5)*3 + winBonus, scoreA, scoreB + 1);
        }else if((moneyA >= TFullBuy) && (moneyB <
CTFullBuy)){
            return 0.96 * calculate_win_probability(team,
moneyA - (TFullBuy/5)*3 + winBonus, moneyB + lossBonus, scoreA
+ 1, scoreB) + 0.04 * calculate_win_probability(team, moneyA -
TFullBuy + lossBonus, moneyB + winBonus, scoreA, scoreB + 1);
        }else if((moneyA < TFullBuy) && (moneyB >=
CTFullBuy)){
            return 0.04 * calculate_win_probability(team,
moneyA + winBonus, moneyB - CTFullBuy + lossBonus, scoreA + 1,
scoreB) + 0.96 * calculate_win_probability(team, moneyA +
lossBonus, moneyB - (CTFullBuy/5)*3 + winBonus, scoreA, scoreB
+ 1);
        }else{
            return 0.512 * calculate_win_probability(team,
moneyA + winBonus, moneyB + lossBonus, scoreA + 1, scoreB) +
0.488 * calculate_win_probability(team, moneyA + lossBonus,
moneyB  + winBonus, scoreA, scoreB + 1);
        }
    }
}
```

This algorithm is evaluating recurrence relation naively to predict the state transition will be result in a binary tree that represents the probability of win and lose. The consecuent, its complexity time is $O(2^N)$ makes the program runs slowly, especially when the initial score state is (0,0) so that we need the optimization of this algorithm.

C. Algorithm Optimization Using Memoization

To avoid the recurrence relation over and over again on a tree, the algorithm can be optimized using memoization techniques. This technique's intention is to keep the value of probability of every state that has been calculated, so then the program doesn't need to retrace the same branch of recursive tree.

A state that contains some element can be represented as a tuple. In this optimization, five parameters that define the state of a round are represented as a tuple:

```
auto currentState = make_tuple(team,moneyA,moneyB,scoreA,scoreB);
```

This tuple act as an identity key in the map data structures. Program use map to map each state tuple key to a double type value that contain a win probability:

```
map<tuple<bool, int, int, int, int>, double> memo;
```

The probability evaluation flow in calculate_win_probability function will be modified so it will not always form a new recursion branch. The flow divided into 3 simple phases:

- Lookup
  Before continuing the recursion, program checks if the current tuple state has already existed in `memo` through condition `if(memo.count(currentState))`. If it has already existed, branch exploration pruned and the value immediately returned.
- Accumulation
  If that tuple state is discovered for the first time, program will compute the probability based on local variable `prob`.
- Registration
  After the calculation done, the result must be saved to the memory by `memo[currentState] = prob;` so it can be recalled if the algorithm found the same state in the future.

Therefore, the complete optimized program will be like this:

```
#include <bits/stdc++.h>

using namespace std;

const int initMoney = 4000;
const int initOTMoney = 50000;
const int maxMoney = 80000;
```

```
const int winBonus = 16250;
const int lossBonus = 12000;
const int CTFullBuy = 24000;
const int TFullBuy = 21000;

map<tuple<bool, int, int, int, int>, double> memo;

void teamInisiation(string *teamA, string *teamB, int *moneyA,
int *moneyB, int *scoreA, int *scoreB){
    cout << "Insert the first team (First half playing as CT):
";
    cin >> *teamA;
    cout << "Insert the second team (First half playing as T):
";
    cin >> *teamB;
    cout << "Insert the current score of " << *teamA << ": ";
    cin >> *scoreA;
    cout << "Insert the current score of " << *teamB << ": ";
    cin >> *scoreB;
    if((*scoreA + *scoreB == 0) || (*scoreA + *scoreB == 12)){
        *moneyA = initMoney;
        *moneyB = initMoney;
    }else{
        cout << "Insert the current total money of " << *teamA
<< ": ";
        cin >> *moneyA;
        cout << "Insert the current total money of " << *teamB
<< ": ";
        cin >> *moneyB;
    }
}

double calculate_win_probability(bool team, int moneyA, int
moneyB, int scoreA, int scoreB){
    // Recurrence Base (if the game ends tie)
    if((scoreA == 15) && (scoreB == 15)){
        return 0;
    }
    // Recurrence Base (if the game ends on A win the game)
    if(((scoreA == 13) && (scoreB < 12)) || (scoreA == 16)){
        return team;
    }
    // Recurrence Base (if the game ends on B win the game)
    if(((scoreB == 13) && (scoreA < 12)) || (scoreB == 16)){
        return !team;
    }


    if(scoreA + scoreB == 12){
        moneyA = initMoney;
        moneyB = initMoney;
    }
    if((scoreA + scoreB == 24) || (scoreA + scoreB == 27)){
        moneyA = initOTMoney;
        moneyB = initOTMoney;
    }
    if(moneyA > maxMoney){
        moneyA = maxMoney;
    }
    if(moneyB > maxMoney){
        moneyB = maxMoney;
    }

    auto currentState = make_tuple(team, moneyA, moneyB,
scoreA, scoreB);
    if(memo.count(currentState)){
        return memo[currentState];
    }

    double prob = 0;

    if((scoreA + scoreB < 12) || ((scoreA + scoreB >= 24) &&
(scoreA + scoreB < 27))){
        if((moneyA >= CTFullBuy) && (moneyB >= TFullBuy)){
            prob = 0.488 * calculate_win_probability(team,
moneyA - (CTFullBuy/5)*3 + winBonus, moneyB - TFullBuy +
lossBonus, scoreA + 1, scoreB) + 0.512 *
calculate_win_probability(team, moneyA - CTFullBuy +
lossBonus, moneyB - (TFullBuy/5)*3 + winBonus, scoreA, scoreB
+ 1);
        }else if((moneyA >= CTFullBuy) && (moneyB <
TFullBuy)){
            prob = 0.96 * calculate_win_probability(team,
moneyA - (CTFullBuy/5)*3 + winBonus, moneyB + lossBonus,
scoreA + 1, scoreB) + 0.04 * calculate_win_probability(team,
moneyA - CTFullBuy + lossBonus, moneyB + winBonus, scoreA,
scoreB + 1);
        }else if((moneyA < CTFullBuy) && (moneyB >=
TFullBuy)){
            prob = 0.04 * calculate_win_probability(team,
moneyA + winBonus, moneyB - TFullBuy + lossBonus, scoreA + 1,
scoreB) + 0.96 * calculate_win_probability(team, moneyA +
lossBonus, moneyB - (TFullBuy/5)*3 + winBonus, scoreA, scoreB
+ 1);
        }else{
            prob = 0.488 * calculate_win_probability(team,
moneyA + winBonus, moneyB + lossBonus, scoreA + 1, scoreB) +
0.512 * calculate_win_probability(team, moneyA + lossBonus,
moneyB  + winBonus, scoreA, scoreB + 1);
        }
    }else{
        if((moneyA >= TFullBuy) && (moneyB >= CTFullBuy)){
            prob = 0.512 * calculate_win_probability(team,
moneyA - (TFullBuy/5)*3 + winBonus, moneyB - CTFullBuy +
lossBonus, scoreA + 1, scoreB) + 0.488 *
calculate_win_probability(team, moneyA - TFullBuy + lossBonus,
moneyB - (CTFullBuy/5)*3 + winBonus, scoreA, scoreB + 1);
        }else if((moneyA >= TFullBuy) && (moneyB <
CTFullBuy)){
            prob = 0.96 * calculate_win_probability(team,
moneyA - (TFullBuy/5)*3 + winBonus, moneyB + lossBonus, scoreA
+ 1, scoreB) + 0.04 * calculate_win_probability(team, moneyA -
TFullBuy + lossBonus, moneyB + winBonus, scoreA, scoreB + 1);
        }else if((moneyA < TFullBuy) && (moneyB >=
CTFullBuy)){
            prob = 0.04 * calculate_win_probability(team,
moneyA + winBonus, moneyB - CTFullBuy + lossBonus, scoreA + 1,
scoreB) + 0.96 * calculate_win_probability(team, moneyA +
lossBonus, moneyB - (CTFullBuy/5)*3 + winBonus, scoreA, scoreB
+ 1);
        }else{
            prob = 0.512 * calculate_win_probability(team,
moneyA + winBonus, moneyB + lossBonus, scoreA + 1, scoreB) +
0.488 * calculate_win_probability(team, moneyA + lossBonus,
moneyB  + winBonus, scoreA, scoreB + 1);
        }
    }
    memo[currentState] = prob;
    return prob;
}

void printCalculation(string teamA, string teamB, double AWin,
double BWin){
    cout << "The probability of " << teamA << " win the game
is " << AWin << "%" << endl;
    cout << "The probability of " << teamB << " win the game
is " << BWin << "%" << endl;
    cout << "The probability of the game ends tie is " << 100
- AWin - BWin << "%";
}

int main(){
    string teamA, teamB;
    int moneyA, moneyB, scoreA, scoreB;

teamInisiation(&teamA,&teamB,&moneyA,&moneyB,&scoreA,&scoreB);
    memo.clear();
    double AWin = calculate_win_probability(1, moneyA, moneyB,
scoreA, scoreB) * 100;
    memo.clear();
    double BWin = calculate_win_probability(0, moneyA, moneyB,
scoreA, scoreB) * 100;
    printCalculation(teamA, teamB, AWin, BWin);
}
```

By applying memoization, function will only evaluate a unique tuple state each time. Mathematically, the upper bound of this algorithm operation doesn't rely on total route in the tree anymore, but rather bounded by cardinality of state space set

itself, can be notated as O(|S|) which |S| is total of all possibility valid combination of score and economy tuple in that match.

IV. Results

Evaluation of the recurrence relation computational model is conducted to assess the mathematical validity and execution efficiency of the designed algorithm. The testing is implemented utilizing two primary metrics, the result of match-winning probability by specific state calculation and the comparative analysis of execution time (runtime).

A. Probability Analysis Result Using the Algorithm

After building the algorithm for analyzing match-winning probabilities in CS2, we will use the algorithm to analyse a real game match. For example, we want to analyse an ongoing FaZe vs NaVi match with the state of 6 - 4 score, FaZe's total money is $28000, NaVi's total money is $18000, FaZe start playing as Counter-Terrorist team, and NaVi start playing as Terrorist team. Therefore, here is the probabilities of FaZe win, NaVi win, and tie game:

```
Insert the first team (First half playing as CT): FaZe
Insert the second team (First half playing as T): NaVi
Insert the current score of FaZe: 6
Insert the current score of NaVi: 4
Insert the current total money of FaZe: 28000
Insert the current total money of NaVi: 18000
The probability of FaZe win the game is 78.0309%
The probability of NaVi win the game is 18.917%
The probability of the game ends tie is 3.05207%
```

Fig. 2. Console output predicting the match-winning probabilities between FaZe and NaVi.

Both the unoptimized and optimized algorithm have the same output whatever the input is. Here is the various input that will be tested in both algorithm with the first team is FaZe and the second team is NaVi:

TABLE I. FAZE VERSUS NAVI MATCH PARAMETER INPUT SCENARIOS

No	Input			
	FaZe Score	NaVi Score	FaZe Money	NaVi Money
1	14	14	38000	39500
2	8	9	20000	18000
3	3	5	34000	18500
4	0	4	8000	36000
5	0	0	No Input needed for the first round	No Input needed for the first round

Those five various inputs give outputs:

TABLE II. MATCH-WINNING PROBABILITY ANALYSIS RESULTS ACROSS VARIOUS STATE SCENARIOS

No	Unoptimized Algorithm			Optimized Algorithm		
	FaZe Win%	NaVi Win%	Tie%	Faze Win%	NaVi Win%	Tie%
1	26.2144	23.8144	49.9712	26.2144	23.8144	49.9712
2	51.2945	44.8001	3.90533	51.2945	44.8001	3.90533
3	50.8907	45.0270	4.0823	50.8907	45.0270	4.0823
4	15.9912	81.191	2.81788	15.9912	81.191	2.81788
5	48.2272	48.2272	3.5456	48.2272	48.2272	3.5456

B. Time Execution Result

Previously, we knew that the unoptimized algorithm have time complexity of $O(2^N)$ which N is the maximum possible rounds left and the optimized algorithm have time complexity of O(|S|) which |S| is total of all possibility valid combination of score and economy tuple in that match. Now, we will see the actual time execution result of the input on table in milliseconds:

TABLE III. EXECUTION TIME COMPARISON BETWEEN NAIVE RECURSIVE AND OPTIMIZED ALGORITHMS (IN MILLISECONDS)

No	Time Execution (ms)	
	Unoptimized Algorithm	Optimized Algorithm
1	0.0026	0.0263
2	0.0715	0.7534
3	19.6138	14.2379
4	198.2316	22.3049
5	3154.7035	31.1589

From table, we can see that on the first and second inputs, the unoptimized algorithm executed a little bit faster than the optimized algorithm. But, on the third, fourth, and fifth inputs, the optimized algorithm executed much faster than the unoptimized algorithm and the time execution gap keep wider following the increment of the maximum possible rounds left. It implies that the algorithm with Dynamic Programming approach (Optimized Algorithm) is way more efficient than the algorithm with Brute Force approach (Unoptimized Algorithm).

V. Conclusion

This paper has successfully demonstrated the application of discrete mathematics and algorithm optimization in modelling match-winning probability in Counter-Strike 2. By formalizing the match progress as a route inside a finite state space, the interdependencies of scoring and economic shifts were accurately captured through conditional recurrence relations. The implementation of naïve recursive on the beginning shows a critical computational problem, with exponential time complexity $O(2^N)$ that makes the execution slow for the early match state.

Through strategic integration of Dynamic Programming top-down approach using tuple-based memoization, redundant evaluation of overlapping subproblems was successfully eliminated. This optimization successfully transformed the recursive search tree into a Directed Acyclic Graph (DAG), effectively reducing the asymptotic time complexity bound to the cardinality of the valid state space, O(|S|). Runtime testing validated this optimization, showing a massive decrease in execution time for complex states.

VIDEO LINK AT YOUTUBE

Explanatory video of this paper: https://youtu.be/-Fwgg8dqtg4

GITHUB LINK

Here is the github link consist in two C++ Program that I made for Analyzing Counter-Strike 2 Match-Winning Probabilities with additional feature to measure the execution time, one without optimization and the other one with optimization
https://github.com/zakyamani/Makalah-Matematika-Diskrit

ACKNOWLEDGMENT

Praise be to Allah SWT for his guidance and mercy. Through his grace, I am able to complete this paper for IF1220 Matematika Diskrit. I also give my gratitude to Prof. Dr. Ir. Rinaldi, M.T, the lecturer of IF1220 Matematika Diskrit, for his dedication to teach us, his clear explanation, and his comprehensive teaching materials. I am grateful to my family for supporting me financially and mentally, my friends for helping in learning together, and all the authors of the references I have cited for the knowledge and help to let me complete this paper.

REFERENCES

[1] Munir, R. (2026). Rekursi dan Relasi Rekurensi - Bagian 1. Retrieved from https://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2025-2026/13-Barisan,%20rekursi-dan-relasi-rekurens-(Bagian1)-2026.pdf

[2] Munir, R. (2026). Rekursi dan Relasi Rekurensi - Bagian 2. Retrieved from https://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2025-2026/14-Barisan,%20rekursi-dan-relasi-rekurens-(Bagian2)-2026.pdf

[3] Munir, R. (2026). Kompleksitas Algoritma - Bagian 1. Retrieved from https://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2025-2026/25-Kompleksitas-Algoritma-Bagian1-2026.pdf

[4] Munir, R. (2026). Kompleksitas Algoritma - Bagian 2. Retrieved from https://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2025-2026/26-Kompleksitas-Algoritma-Bagian2-2026.pdf

[5] MathWorks. (n.d.). *What Is State-Space Control?*. Retrieved from https://www.mathworks.com/discovery/state-space.html

[6] Aji, A. F., & Gozali, W. (2021). *Pemrograman Kompetitif Dasar*. Retrieved from https://osn.toki.id/data/pemrograman-kompetitif-dasar.pdf

[7] NER0cs. (2025). *Introducing Rating 3.0*. Retrieved from https://www.hltv.org/news/42485/introducing-rating-30

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, 19 Juni 2026

Muhammad Zaky Amani 13525040