Dynamic Programming and Applications

Luca Gonzalez Gauss and Anthony Zhao

May 2020

Abstract

In this paper, we discover the concept of dynamic programming. Dynamic programming can be used in a multitude of fields, ranging from board games like chess and checkers, to predicting how RNA is structured. In order to explain aspects of dynamic programming, we include background information covering: induction, counting and combinatorics, probability theory, and time and space complexity.

Contents

1 Introduction

In this paper, we provide concepts important to the understanding of dynamic programming. These topics are either utilized later in the paper, or allow for a deeper and more contextual understanding of subjects which we do not cover. We include inductive reasoning, time complexity, formulas of permutations, expected values, and polynomial versus nondeterministic polynomial time. We then explore dynamic programming, examining the Longest Common Subsequence (LCS) problem, the dominant strategy of checkers, and the dominant strategy of Go. This information should also demonstrate the effectiveness of dynamic programming, as well as show the numerous applications that dynamic programming has through various mathematical fields. Conclusively, we recognize what can be learned from this paper, and identify future exploration.

Dynamic programming is applicable in graph theory; game theory; AI and machine learning; economics and finance problems; bioinformatics; as well as calculating the shortest path, which is used in GPS. Dynamic programming is also used in laptops and phones to store data through caches so that it does not need to retrieve data from the servers when data is needed. In fact, you may be using dynamic programming without realizing that you are.

2 Background Information

2.1 Mathematical Induction

Induction is a way of proving a statement which depends on a natural number. There are 2 main methods of induction - weak and strong induction. Weak induction works like this:

1. Initial Step (base case) - This is the lowest case we look at (typically $n = 1$ or 0). The case must be solved using an alternative method.

2. Induction hypothesis - This step assumes that the claim holds true when $n = k$.

3. Induction step - This step tries to demonstrate that the claim is true when $n = k + 1$ using the assumption from step 2.

Induction works because of its recursive nature. If we establish that the base case is true, let us assume the base case is it when $n = 1$, we can use the base case to prove the next case: $n = 2$ is true, and then use that to prove $n = 3$ is true and so on.

The difference between weak and strong induction is during the induction hypothesis, weak induction assumes that only the kth step is true, while strong induction assumes all values up to the kth step are true.

Example 1. A problem that can be solved through induction is calculating the sum of the first n natural numbers. The formula for the sum of the first n natural numbers is this

$$1 + 2 + 3 + ... + (n - 1) + n = \frac{(n)(n + 1)}{2}$$

To prove it through induction, we first test the base case - when $n = 1$. The base case is true because $1 = \frac{(1)(2)}{2}$. Next, we assume that the formula works when $n = k$ and prove

$$1 + 2 + 3 + ... + k + (k + 1) = \frac{(k + 1)(k + 2)}{2}$$

which means it still works at $n = k + 1$. The left side of the equation can be changed from

$$1 + 2 + 3 + ... + k + (k + 1) \text{ to } \frac{(k)(k + 1)}{2} + (k + 1)$$

because we assumed when $n = k$ the equation was true. Next, using simple algebra we see that $\frac{(k)(k+1)}{2} + (k+1) = \frac{(k+2)(k+1)}{2}$ and this completes the proof.

2.2 Time Complexity and Runtime

Runtime is used to estimate the time it takes to run an algorithm. *Time complexity* measures the asymptotic behavior of runtime as the input size is increased indefinitely (i.e. runtime is a function demonstrable using big-O notation, while time complexity is graphable). Time complexity is one way of classifying the runtime of an algorithm.

2.2.1 Big-O Notation

Big-O notation represents growth in a function. This paper utilizes big-O notation to quantify the runtime of algorithms, because it is an efficient way of displaying time complexity.

Definition 2.1 (Big-O Notation)**.** If $g(x)$ is a real or complex function, and $f(x)$ is a real function, $g(x)$ is $\mathrm{O}(f(x))$ if and only if the value $|g(x)|$ is at most a positive constant multiple of $f(x)$ for all sufficiently large values of x.

Example 2. QuickSort, the on-average most efficient common sorting algorithm, has a runtime of $O(n \times \log(n))$ when rearranging n numbers in a predetermined way (such as from least to greatest).

2.2.2 Brute-force versus Dynamic Programming Methods

Brute-force methods are the more common, non-efficient algorithms easily created to perform a certain task. They function by systematically verifying all possible cases, and checking whether these cases satisfy the algorithm's conditions. This is commonly not ideal. Brute-force methods often compute duplicate scenarios and other scenarios which would not be computed if using a well-constructed algorithm. Alternatively, one can utilize dynamic programming methods to limit the runtime of our algorithm and therefore optimize its efficiency.

2.3 Counting and Useful Formulas

A permutation is a rearrangement of a finite set. Two formulas involving permutations become useful for applications in dynamic programming. Firstly, that the number of permutations for a set of n elements is $n!$; and secondly, that n^k sequences of length k can be formed from an n-element alphabet.

2.4 Expected Values

Definition 2.2. The *expected value* is the weighted average, calculated by multiplying each independent outcome of a random variable x and its associated probability of occurrence. Expected values are linear, meaning that they are additive.

When x_i is the ith outcome and p_i is the probability that the ith outcome occurs, the formula for the expected value $E(x)$ is

$$E(x) = \sum_{i=1}^{n} x_i p_i \tag{1}$$

In **Section 3.5.5**, we use this to calculate the average number of pieces on a Go gameboard.

2.5 Polynomial Time (P), Nondeterministic Polynomial Time (NP) and Space Complexity

The concepts of *Time and Space complexity* are useful in describing the efficiency of

We introduce the topic of the *Turing machine* to help describe Time and Space complexity.

2.5.1 Turing Machine

Definition 2.3. A *Turing machine* is a theoretical, idealized computer that is meant to simulate an algorithm step by step, much like how a human would execute an algorithm. The Turing machine has a tape, a head, a set of states, and a transition function (Figure 1).

The *tape* is considered the "input" of the machine. It is a one dimensional array of cells, that reads a symbol from a finite alphabet (A). Two symbols in the alphabet must be a "start" symbol and a "blank" symbol. The blank symbol just means the cell does not have anything written on it. The start symbol represents where the head starts at.

The *head* can move along the tape in both ways, and read and edit the symbols it is looking at.

The *set* of states (S) corresponds to how the head reacts to the symbols on the tape. As the head moves along the tape, the state changes. The way the machine reacts to a symbol depends on the state the machine is in already. For example, if Turing machine T is in state z, and reads the symbol g, the machine will react differently from when its reading symbol h. The set of states usually has "yes" and a "no" states.

The *transition function* describes how the Turing machine works. We will write the function in an input ($f : x$) and output (y) format - $f : x \to y$. Assume S is a set of states and A is a finite alphabet. The transition function is written out as

$$f : S \times A \to (S \cup \text{“}Yes\text{”} \cup \text{“}No\text{”}) \times A \times (\leftarrow, \rightarrow, \text{stay})$$

The $(S \cup \text{“}Yes\text{”} \cup \text{“}No\text{”})$ describes, in its current state, whether it accepts ("Yes") or rejects ("No") the symbol on the cell. Finally, the $A \times (\leftarrow, \rightarrow, \text{stay})$ part describes replacing the current symbol using A and moving the head.

Definition 2.4. Turing machines can be deterministic or nondeterministic. A *deterministic Turing machine* differs from a *nondeterminstic Turing machine* in that every state can be changed to one and only one other state, or the next state is "determined". In a nondeterminstic Turing machine, the state can change into any one state of a set of states.

2.5.2 Polynomial and Nondeterministic Polynomial Time

Polynomial Time (P) and *Nondeterministic Polynomial Time (NP)* are categories related to time complexity. In the following definition, n is the size of the input and k is any natural number.

Definition 2.5. An algorithm that runs in polynomial time is an algorithm that when put through a deterministic Turing machine, the Turing machine can complete the algorithm in $\mathrm{O}(n^k)$.

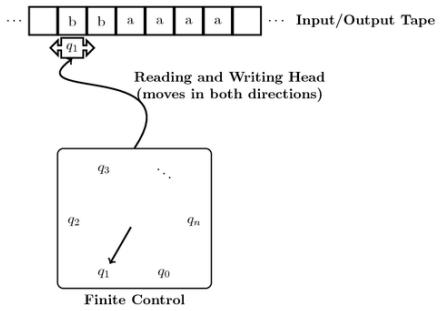

Figure 1: A Turing Machine.

Definition 2.6. An algorithm that runs in nondeterministic polynomial time is an algorithm that when put through a nondeterministic Turing machine, the nondeterministic Turing machine accepts it in polynomial time.

Example 3. If an algorithm's runtime were $O(n^2)$, the algorithm would run in polynomial time. If an algorithm's runtime were $O(3^n)$ it would not run in polynomial time.

2.5.3 Space Complexity

Space complexity is the amount of working storage a computational algorithm requires to be solved.

Definition 2.7. PSPACE contains all computational algorithms such that when solving the algorithm the amount of storage used is polynomial.

Definition 2.8. LSPACE contains all algorithms such that the algorithm is solved by a deterministic Turing machine using a logarithmic amount of storage. The time complexity correspondent to LSPACE is L.

Definition 2.9. Nondeterministic Logarithmic Space (NL) holds all computational algorithms such that, when solved by a nondeterministic Turing machine the algorithms takes up a logarithmic amount of storage. Since any deterministic Turing machine is also nondeterministic, LSPACE is a subset of NL.

Definition 2.10. Encompassing all of these complexity classes, EXPSPACE is the set of all algorithms solved by a deterministic Turing machine using an exponential amount of space. EXPTIME is the time complexity class of algorithms which take an exponential amount of time to solve.

3 Explorations in Dynamic Programming

This section is dedicated to introducing dynamic programming, and explaining common dynamic programming algorithms through three examples.

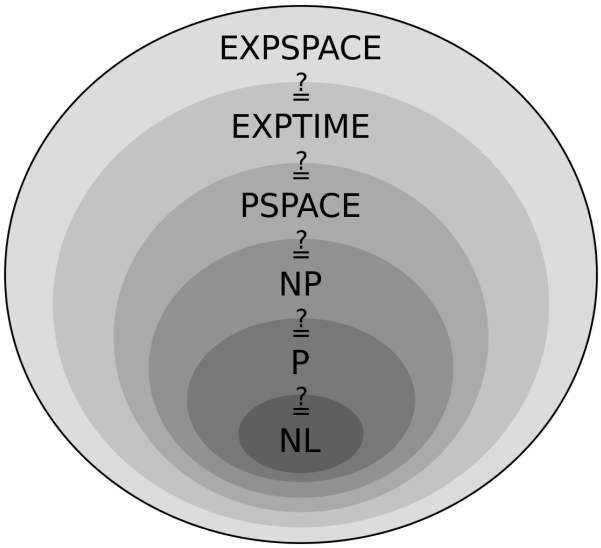

Figure 2: A Visual Representation of Common Computational Complexity Sets.

3.1 Dynamic Programming

Dynamic programming is a method of simplifying complicated problems efficiently; it achieves this by breaking the problem into subproblems recursively. The method should preferably optimize the runtime of the algorithm being solved. Dynamic programming has applications in mathematical optimization, computational complexity theory and computer programming. When an algorithm is broken down such that the runtime is most efficient, it is in its optimal substructure.

In order to quantify the reorganization of an algorithm into its optimal substructure, the sequence of subproblems r_1, r_2, ... r_n is defined. Taking into account an argument which determines the state of each subproblem and utilizing the Bellman equation we can, once determining the state of r_n recursively decide the state of r_{n-1}, r_{n-2}, ... , r_2, r_1.

The value function returns the value given by correlating functions which minimize the amount of calculations needed at each subproblem. For example, in the case of dynamic programming, these value functions signify the maximum amount of calculations we do not perform by using our dynamic programming methods versus brute-force methods. Value functions can be used to demonstrate the possible payoffs given by structure of subproblems. Dynamic decision problems such as ours can be rephrased as a value function at time 0, corresponding to the state x_0. By this standard, the state at time t is x_t. For the most efficient usage of dynamic programming, we want to have an optimal value function at time t.

3.1.1 Bellman Equation

The value function $V(x_0)$ of an infinite-horizon dynamic decision problem can be rephrased as the maximum of the summation of every action's payoff within the problem.

The Bellman equation rephrases value functions to quantify and identify the recursive means by which dynamic programming is applied. It defines the value function $V(x_0)$ as being the maximum of the payoff in state x_0 plus the value function $V(x_1)$.

The Bellman equation has widespread applications in economics, particularly among idealized problems surrounding economic growth, resource extraction, and public finance. The problem can also be reworked to adapt to stochastic optimal control problems.

An example of a state-determining argument is common among problems based around finding the dominant strategy of games. If, based on the board state of the game, it is possible that one player utilizing perfect play must win the game, then that board can be given a state of "win". This state is utilized later when analyzing the dominant strategies of checkers and Go.

The following sections describe examples of dynamic programming.

3.2 Length of the Longest Common Subsequence of 2 Sequences

We find the length of the longest common subsequence of 2 sequences by using a dynamic programming algorithm.

3.2.1 Problem Statement

The problem is to describe an algorithm and explain its run time for finding the length of the longest common subsequence of two sequences each of length n.

For example, the length of the longest common subsequence of [1, 3, 5, 7, 2, 10, 11] and [1, 4, 3, 5, 10, 2, 9] would be 4 because they share [1, 3, 5, 10] in that order.

3.2.2 The Brute-Force Algorithm

One approach to this problem is to brute-force it by testing every possible subsequence from one sequence to the other. Since there are 2^n subsequences each in the two n-length sequences this is highly inefficient. Another approach to this problem is to use dynamic programming.

3.2.3 Dynamic Programming Algorithm

Allow the two sequences A and B to be of length n. We can use a function L to compare A and B, through the first a and b elements respectively. Function

L is formatted like this, $L[a, b]$, which is considered a subproblem in dynamic programming.

Definition 3.1. $L[a, b]$ refers to the length of the longest common subsequence between the first a characters of A and the first b characters of B. The inequalities $1 \leq a \leq n$ and $1 \leq b \leq n$ must be true.

The algorithm compares the last elements of the two subsequences. This comparison results in two cases - where the last elements are same, and where the last elements are not.

In the case where the last digits match, add one to the length of the previous largest common subsequence or $L(a-1, b-1)$. This is because the matching last characters of A and B are part of the longest common subsequence, so the length of the longest common subsequence must be one more than the length of the previous common subsequence.

In the case where the last digits are different, The longest subsequence would be the length of $L(a-1, b)$ or the length of $L(a, b-1)$ depending on which is longer. This is because the last digit does not add to the longest common subsequence so the length of the longest common subsequence does not change.

Example 1. Take into account two example sequences A and B. Let us assume that A : [1,2,3,4,5] and that B : [1,6,7,8,5]. Following the algorithm above, we can see that $L[1, 1]$ equals 1. We can, using this action, find $L[2, 1]$ and $L[1, 2]$, from which we can then find $L[2, 2]$, $L[1, 3]$ and $L[3, 1]$. By repeating this process we can use previous stored data to find $L[n, n]$. If we follow this method, we will only calculate each L-value once.

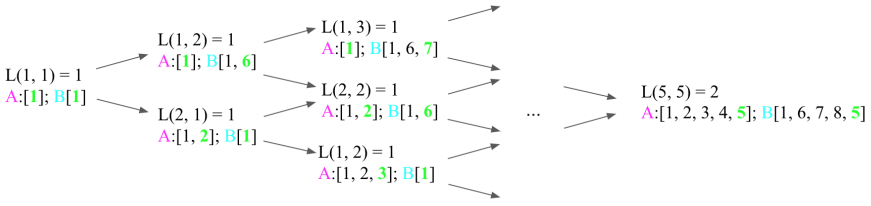

Figure 3: A visual of an example of the Longest Common Subsequence problem

3.2.4 Runtime

By repeating this process for each element of the sequences, we can find the length of the longest common subsequence. The algorithm takes into account each value of L-function only once, which is why it is efficient. The L-function calculation simply compares the last element of each sequence, which makes each comparison run in O(1) time. Ultimately, to calculate $L(n, n)$, the algorithm must go through every element of one sequence and compare it to every element of the other sequence; this amounts to n^2 L-function calculations. Finally,

multiplying the time it takes to complete one L-function calculation, to the total amount of L-function calculations, results in a runtime of O(n^2). This is of the time complexity set P.

3.3 Checkers Dominant Strategy

3.3.1 Problem Statement

The goal of this problem is to solve or find the optimal strategy for the board game checkers by using Dynamic Programming. The dominant strategy is not necessarily the winning strategy, but rather the strategy that, when another player is also using the dominant strategy, results in an optimal outcome for you; so, if there is no possible way to win the game, the next best thing is to tie the game.

3.3.2 International Rules of Checkers

Checkers is board game with 8 rows and 8 columns, amassing to 64 squares. Every square is either black or white and no 2 adjacent squares are the same color. Each player has 12 pieces which are placed on the black squares nearest to them. These pieces can only move forward: diagonally; or over opposing pieces, which "captures" that piece. Once a piece reaches the end row of the opponent's side, that piece is *crowned* and becomes a "king/queen", gaining the ability to go backwards. The goal of the game is to capture all opposing pieces.

3.3.3 Tracing Backwards

A scenario where you can force a win can be used as leverage for disregarding other scenarios. Using the scenario, we can then test setups which could lead to a scenario in which we must win. When finding the possibility of forcing a win, we can then disregard every other calculation that would be made in this earlier scenario. By storing previous moves, if we find that we can force a win in certain setups midway into the game, we can disregard any future possible scenarios (except for that in which we can force a win).

In order to trace backwards we must use the top-down method. The top-down method uses the final states as its base case. Next, The top-down method looks at the cases that could have preceded it. As with other dominant strategy methods, the top-down method gives board configurations a state based off of whether you can win, tie, or lose if both players are using perfect play. The top-down method repeats this process until it reaches the beginning board configuration.

3.3.4 Algorithm

Let us look at an 8 by 8 board and also disregard crowned pieces. We will assign either win, loss, or draw state for each individual scenario. These labels mean

from that point on, if both players were to play perfectly, that would be the only result that could happen.

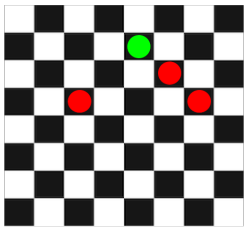

Figure 4: An Inescapable Win.

We first look at scenarios in which the opposing player has only one piece left, which we can automatically consider a win.

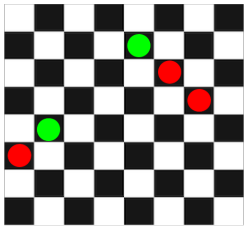

Figure 5: A Possible Precursor to the Previous Board Configuration.

After, understanding that any scenario with one piece left can force a win, we can then observe scenarios with two pieces that could have preceded the one piece. We should then verify that this scenario, under optimal play, it still of the value Win or Draw. If not, disregard that case. After limiting the amount of two piece scenarios, we can then move to three piece scenarios; again, finding the optimal value possible for preceding scenarios, and disregarding the rest. By repeating this process we can trace a chain of optimal moves back to the beginning of the game, thus creating a means of optimal play.

The algorithm does not necessarily need to go through every sequence of connected cases. An optimal move is a move that leads into another optimal move. This means we have to look at our move and the opponents move to guarantee an optimal move. Essentially, we will only have to look at 2 moves

total to determine whether a move is optimal or not, which will have an effect on our runtime.

3.3.5 Runtime

What is the runtime of our algorithm? First, let us generalize the board to be k by k. This means that each player has $\frac{k}{2}(\frac{k}{2}-1)$ pieces. The number written in Big O notation is $O(k^2)$ because k is squared when simplified. Next, we find how many moves each player can make. There are 2 types of pieces, regular pieces and crowned pieces. These pieces can move in 2 different ways, capturing a piece or moving regularly.

Definition 3.2. Regular Piece: A regular non-capturing piece can move in 2 one tiled ways and a regular capturing piece can move in 2^k ways. Assume the regular capturing piece can jump in both ways it can move. Then assume after the jump, the piece can capture in both ways again. This can be repeated until the end of the board during a worst case scenario, making it 2^k. Written in Big O notation, a regular non-capturing piece is $O(1)$ because the number is constant, and the regular capturing piece is $O(2^k)$.

Definition 3.3. Crowned Piece: A crowned non-capturing piece can move in 4 unlimited-tiled ways - this ends up being at max k tiled because worst case scenario is if they are in a corner and the move to the opposite corner which is k tiles one direction. A crowned capturing piece can move 2×2^k ways because the crowned piece is like the regular piece but it can move backwards too. Written in Big O notation, a crowned non-capturing piece is $O(k)$ because the paths are 2k, and the crowned capturing piece is $O(2^k)$.

The amount of subproblems can be determined by combining our factors together.

$$\text{(possible pieces)(possible moves)} = O(k^2) \times O(2^k) = O(k^2 \times 2^k) \qquad (2)$$

To determine the payoff state of this subproblem, we can check the state of the $O(k^2 \times 2^k)$ possible board configurations derived from this board. Figure 5 illustrates a system of subproblems, where each subproblem is connected to the subproblem that preceded it. Determining state takes $O(1)$ because of our usage of recursion, so the total runtime of solving a subproblem is $O(k^2 \times 2^k)$. There are $O(k^2)$ spaces where pieces can be placed on a checkers board, each with three possible states of a white piece, a black piece, or no piece. This means that there $O(3^{k^2})$ possible board configurations. Therefore, the total runtime of the applied dynamic programming algorithm is $O(k^2 \times 2^k \times 3^{k^2})$. This is in the set of EXPTIME.

3.3.6 Simplifications

We can simplify checkers to significantly reduce the runtime of the algorithm. One massive simplification to the algorithm is if we do not include the capturing

pieces. The capturing part of checkers takes a lot of time and space to process because $O(2^k)$ is a extremely significant factor in the runtime and increases very fast. Another simplification that can be made is by removing the crowned piece aspect of the game. This limits each piece to at most 2 possible moves. This allows each subproblem to have $O(k^2)$ runtime. The total algorithm then has a runtime of $O(k^2 \times 3^{k^2})$.

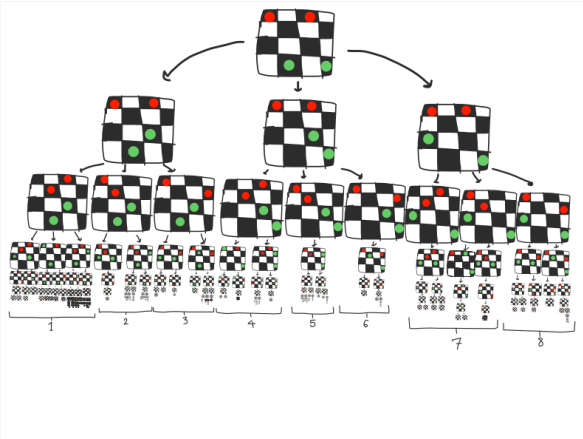

Figure 6: A System of Subproblems.

3.4 Go Dominant Strategy

3.4.1 Problem Statement

Our goal is to find the dominant strategy of the game of Go in the least possible amount of calculations.

3.4.2 International Rules of Go

Go is a board game where the goal is to surround more territory than the opponent using "houses". Go is played on a nineteen-by-nineteen grid of horizontal and vertical lines, and stones may only be placed on the 361 intersections of these lines. It is a two-player game that is played with "stones"; one player uses white stones, and the other black stones. Black always moves first. If a stone or group of stones is surrounded by stones of the opposing player, they are considered "captured" and are removed from the board.

A player, seeing no possibilities for profitable play, may pass. When both players pass, the game is ended. Points are tallied based on the number of

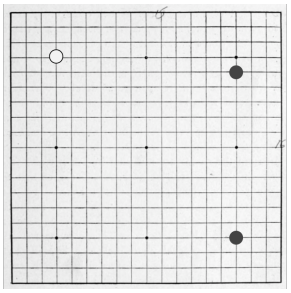

Figure 7: The Beginnings of a Go Game.

intersection points within houses subtracted by the number of captured stones. Whichever player has the greater score at the end of this tally wins the game.

3.4.3 Case Examples

Our first attempts of solving Go focuses on a smaller example. In order to limit calculations, we will constrain the board to a smaller size. This constraint should be made in an efficient and realistic manner; therefore, we must decide on our smaller board based on certain criteria.

1. The board should remain square. Any other shape would obviously distort the game-play. In some scenarios it would be impossible to capture stones or create eyes due to how compromising the shape of the board is. Players would be able to constrain stones more easily, and opponents would struggle to defend their stones.

2. Go is a game of significant complexity. As such, this complexity should be reflected in the size of the board. Pieces interact with one another in ways that would be hard to replicate in smaller board sizes. Yet we also wish to limit the amount of calculations.

3.4.4 Impossible Scenarios

Many board positions are impossible to create. For instance, when randomly generating board positions, you will inevitably come into a scenario where a group of stones would have, in the actual game, been captured by the opposing stones. These scenarios are impossible and should be disregarded. Consequently, our runtime calculation is a bit of an overestimate because it considers these impossible scenarios.

3.4.5 Runtime

Bottom-Up Method Previously, when solving the checkers dominant strategy, the tracing backwards, or top-down method is used. The bottom-up method is the alternative to the top-down method. The bottom-up method

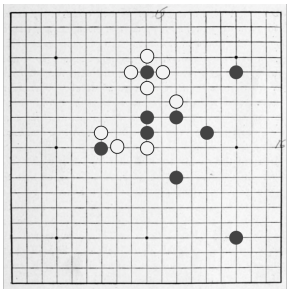

Figure 8: An Impossible Scenario.

is a method which recognizes the first possible boards, checks which configurations are possible from the sequential board states, until the algorithm reaches a stopping point where the win/loss state is palpable. The algorithm then looks back on preceding configurations, to validate that, with perfect play, the board configuration leads to a win.

The total runtime of calculating the bottom-up method is equal to the number of board positions multiplied by the number of possible moves in these board positions.

Runtime of Simplified Board In our simplified board, we have a seven by seven grid where each intersection can either be occupied by the player's stone, the opposing player's stone, or unoccupied. There are forty-nine intersections a stone can be on. Therefore, the amount of board positions possible is 3^{49}, because there are three possible cases for each intersection. This is approximately 2.3×10^{23} board positions.

The amount of moves the player can take is dependent on the amount of pieces already on the board. The possible amount of pieces varies greatly in each board position. But the sum total of pieces out of every scenario stays constant. Specifically, this total is equal to 2^{49}, meaning that there will be an average of $32.\overline{6}$ stones on each board. This means there is an average of $16.\overline{3}$ intersections where a piece can be placed. Furthermore, a player can pass their turn, providing another move and bringing the average to $17.\overline{3}$.

Combining the two criteria, for a seven by seven grid and utilizing the bottom-up method, we find the total runtime is equal to 17.34×3^{49}, or roughly 3.9×10^{24} computations.

We find that the runtime of any board of length $\sqrt{I}$ and total intersections I is equal to $(I/3 + 1) \times 3^I$. $I/3$ is derived from the fact that there are three possible cases per intersection: a white stone occupying; a black stone occupying; and empty space (we do not consider houses to be separate from empty space). Because we are taking into consideration every board configuration, $I/3$ is the expected value of the average amount of positions a player can place a stone. The added 1 corresponds to the fact that a player may always pass. 3^I is

given because, as previously stated, there are three possible cases per intersection. Board configurations are derived of different arrangements of these cases. Every intersection provides three cases, surmounting to 3^I total possible board configurations. This runtime is within the set EXPTIME.

Therefore, for the traditional 19×19 board with 361 intersections, the runtime of calculating the dominant strategy via the bottom-up method is 121.34×3^{361}, or approximately 2.1124×10^{174}.

3.4.6 Brute-Force Method

To demonstrate the efficiency of dynamic programming and our algorithm, we take into account an alternate algorithm which does not utilize such optimization methods. Rather than storing information, this algorithm goes through every possible scenario by methodically playing out every possible game of Go. Because this method does not store information, calculations are repeated for board states we have encountered multiple times.

This algorithm must first consider I possible moves from the empty board, then $I-1$ for all possible moves (excluding their first), until there are no possible further moves. This means that the runtime of the brute-force method is $O(I!)$. This is also within the set EXPTIME, though less efficient than its brute-force counterpart. For context, when $I = 49$, the runtime of the brute-force method is $6.08 * 10^{62}$, while the runtime of our algorithm is only $4.14 * 10^{24}$.

We supplement these case examples with information from other branches of mathematics, such as induction, computational complexity theory, and counting. This information should demonstrate the effectiveness of dynamic programming, as well as show the numerous applications that dynamic programming has through various mathematical fields.

Dynamic programming is useful in graph theory; AI and machine learning; economics and finance problems; as well as calculating the shortest path, which is used in GPS. Laptops and phones uses dynamic programming by storing data through caches so that it does not need to retrieve data from the servers when data is needed. You may be using dynamic programming without realizing that you are.