Dynamic Programming and Applications

Diego Luca Gonzalez Gauss and Anthony Zhao

Mentored by Jung Soo (Victor) Chu

Table of Contents

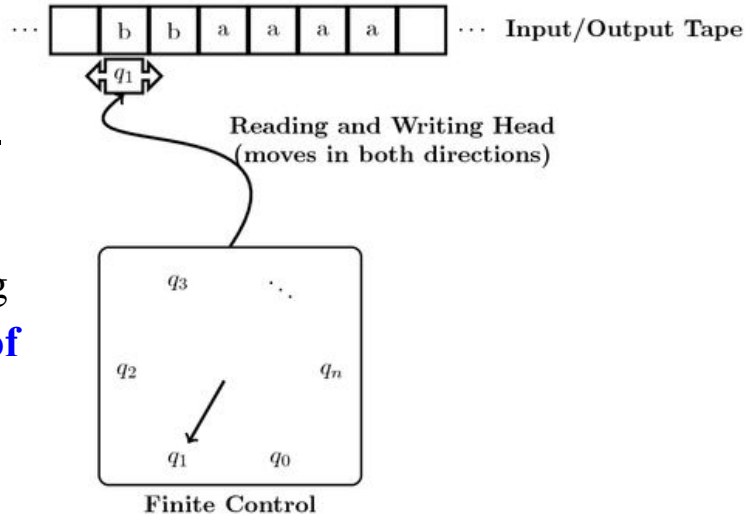
List of Background Information

- Polynomial Time and Nondeterministic Polynomial Time
- Space Complexity and Common Complexity Sets

Background Information - Turing machines

Turing machine

A theoretical, idealized computer that reads a tape as an input and changes its state. The Turing machine can be used to determine time and space complexity.



A Turing machine

Deterministic versus Nondeterministic Turing machines

Every state of a **deterministic** Turing machine can only be changed to **one** other state, while in a **nondeterministic** Turing machine, the state can be changed randomly to **any one state of a set** of states.

Background Information - P and NP

n: the size of the input into the algorithm

k: any natural number.

Polynomial time (P)

Polynomial time algorithm: deterministic Turing machine completes in O(n^k) time.

Nondeterministic Polynomial time (NP)

Nondeterministic Polynomial time algorithm: nondeterministic Turing machine completes in Polynomial time.

Background Information - Space complexity

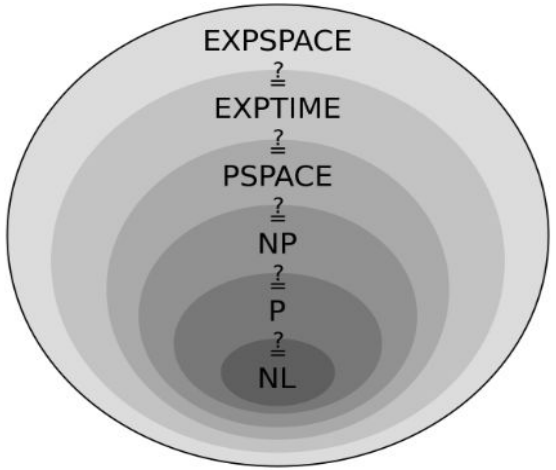
Space complexity

The amount of **working storage** an algorithm requires to be solved.

L: solved by a deterministic Turing machine in a logarithmic amount of time

LSPACE: the algorithm is solved by a deterministic Turing machine using a logarithmic amount of storage

Nondeterministic Logarithmic Space (NL): the algorithm, when solved by a nondeterministic Turing machine, takes up a logarithmic amount of storage



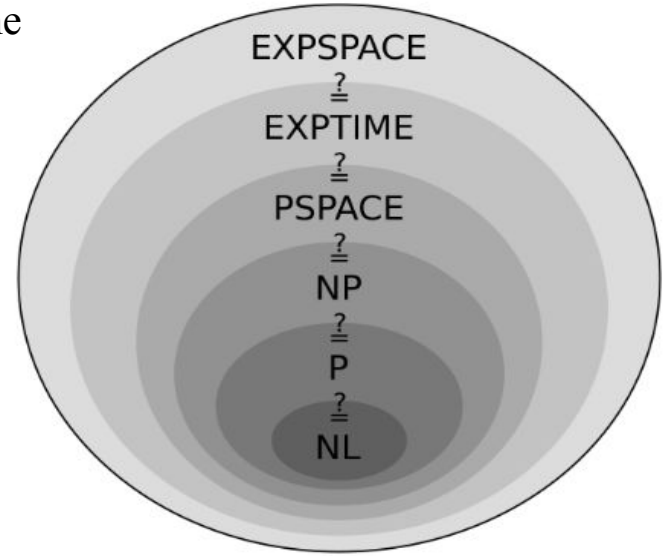
A visual representation of common complexity classes

Background Information - Common complexity sets

PSPACE: the algorithm's storage usage is polynomial in terms of the input size

EXPTIME: algorithm takes an exponential amount of time to solve.

EXPSPACE: solved by a deterministic Turing machine using an exponential amount of space.



A visual representation of common complexity classes

List of Explorations in Dynamic Programming

- Definition of Dynamic Programming
- Longest Common Subsequence (LCS) Problem
- Dominant Strategy of Go

Definition of Dynamic Programming

- Simplifies through **recursively** creating subproblems
- Sequence of subproblems $r_1, r_2, \ldots r$□
- State-defining arguments for each subproblem
- An example - “win” versus “tie” versus “loss”.
- Each board configuration can be labeled as “win,” “tie,” or “loss.”

Bellman Equation

- The value function
- Definition of infinite-horizon decision problem in terms as $V(x_0)$
- **Recursively** defines infinite-horizon decision problems

Payoff at time i

$$V(x_i) = \max\{P(x_i) + V(x_{i+1})\}$$

Value function at time i

Value function at time i + 1

2 Examples of Dynamic Programming

- 2 examples of dynamic programming
  - Longest Common Subsequence (LCS) Problem
  - Dominant Strategy of Go
- Brute Force algorithm and runtime
- Dynamic Programming algorithm and runtime

Longest Common Subsequence (LCS) Problem Statement

- The problem: find the length of the longest common subsequence of two sequences, each of length *n*.
- An example - the LCS of [1, 3, 5, 7, 2, 10, 11] and [1, 4, 3, 5, 10, 2, 9] is [1, 3, 5, 2] (length 4).

Longest Common Subsequence (LCS) Brute-Force

- Brute-force algorithm - tests all possible subsequences
- Runtime of brute-force algorithm is $O(n \times 2^n)$

Longest Common Subsequence Dynamic Programming

L-function

***L*(a,b):** length of LCS of the first *a* characters of the first sequence and first *b* characters of the second sequence.

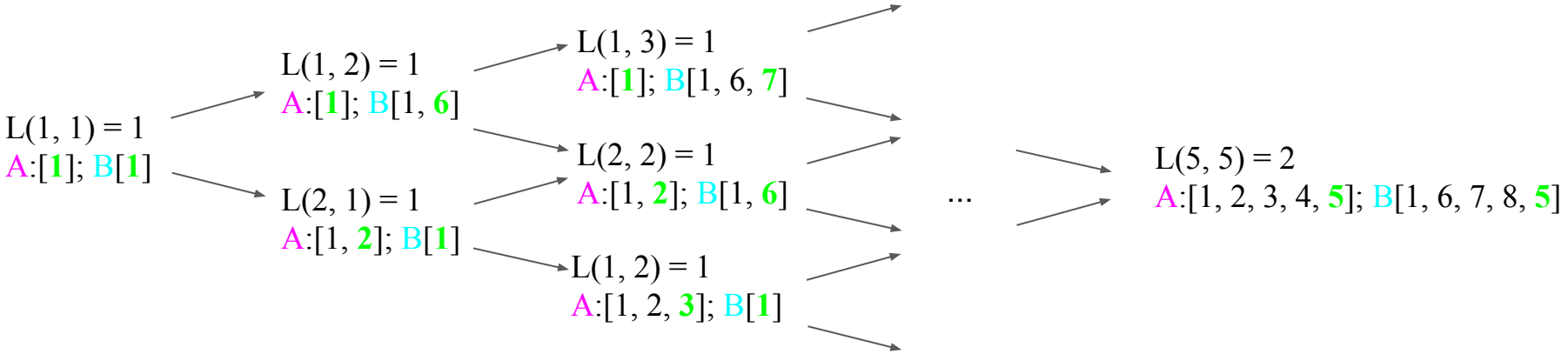
- Dynamic programming algorithm: defining new *L*-function recursively
- When last digits match, add one to the tally of the LCS of *L*(a - 1, b - 1)
- When last digits do not match, previous LCS is now the LCS of *L*(a, b - 1) or *L*(a - 1, b); take the LCS of whichever *L*-function is larger

Longest Common Subsequence (LCS) Example

An example using the *L*-function

A = [1, 2, 3, 4, 5]

B = [1, 6, 7, 8, 5]



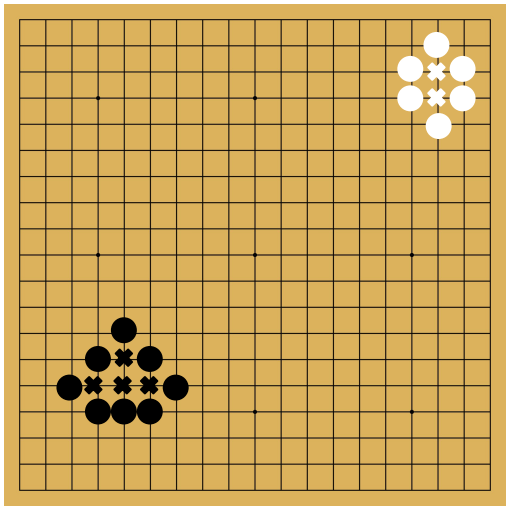
Longest Common Subsequence (LCS) Runtime

Runtime

- n^2 subproblems.
- O(1) runtime for each subproblem calculation.
- Runtime of dynamic programming algorithm is O(n^2).

Dominant Strategy of Go Problem Statement

- International rules of Go
- Use dynamic programming to find the optimal strategy of Go.

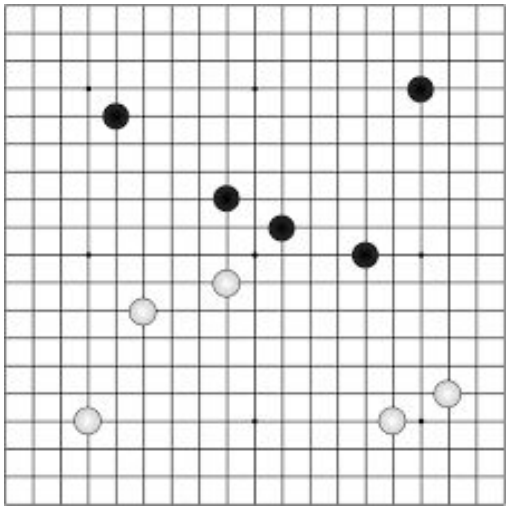


A Go Board Showing Houses.

✖ : houses surrounded by black stones.

✖ : houses surrounded by white stones.

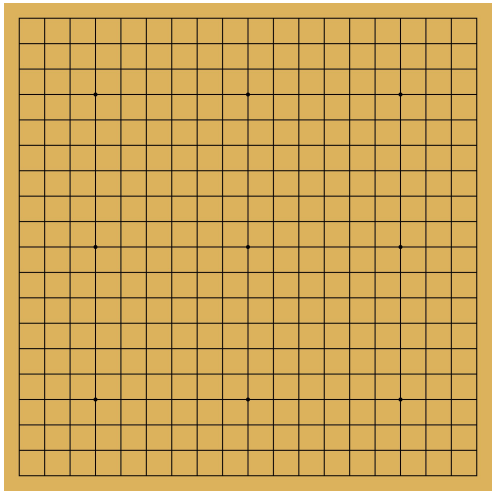
Black: 4 houses
White: 2 houses



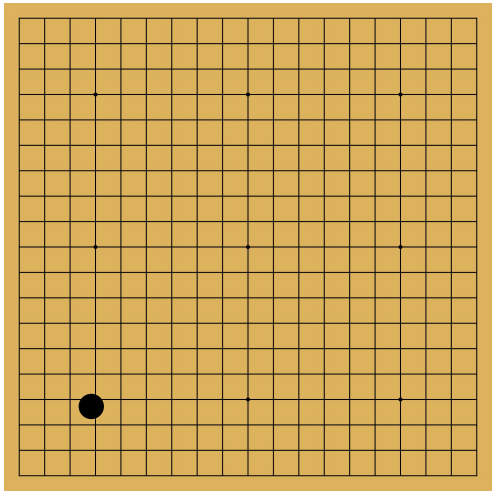
A Go Board.

Dominant Strategy of Go Brute-Force

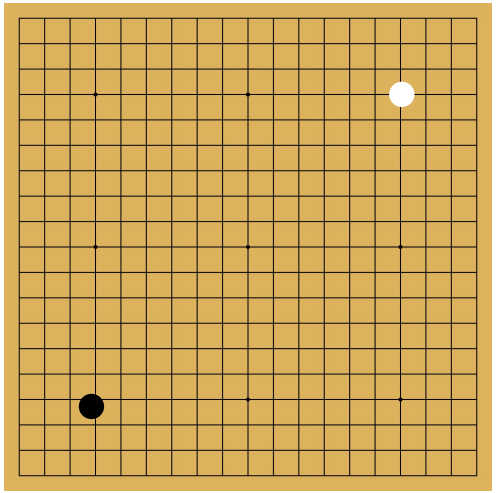
- Brute-force algorithm - methodically plays out every possible game of Go.
- Runtime of brute-force algorithm is **at least** O(I!) for a board of total intersections I.



A Go Board, with 361 possible moves.



After placing a piece, there are now 360 possible moves.



After placing 2 pieces, there are only 359 possible moves.

Dominant Strategy of Go Dynamic Programming

- Dynamic programming algorithm: bottom-up method
- Runtime of dynamic programming algorithm is

$$O((I/3 + 1) \times 3^I)$$

Avg. # of possible moves

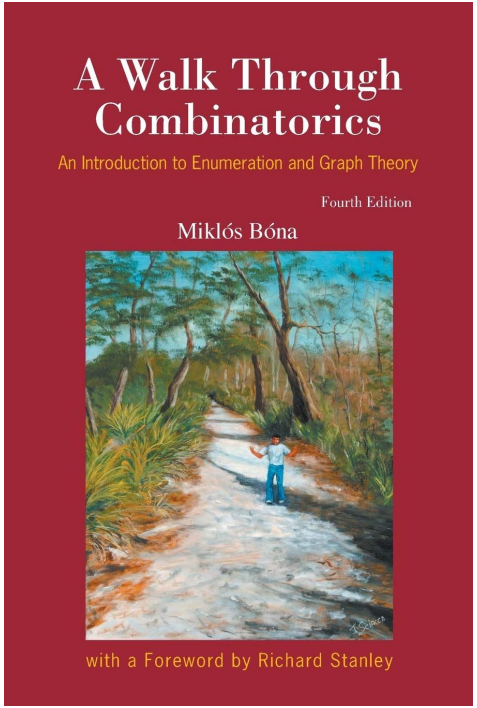
# of possible board configurations

- When I equals 49 (on a 7 × 7 board) the total number of calculations for brute-force versus dynamic programming methods is 6.08×10^{62} versus 4.14×10^{24}.

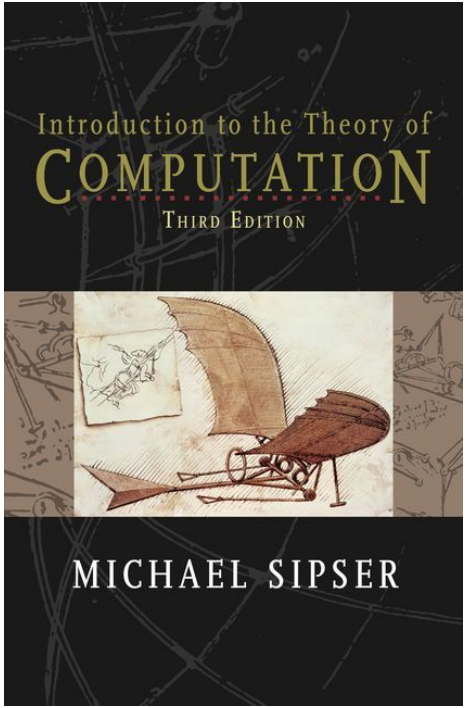
Conclusion

- Summary
- Worldly applications of dynamic programming

References



A Walk Through Combinatorics, Fourth Edition; Miklós Bóna



Introduction to the Theory of Computation, Third Edition; Michael Sipser

Background Information - Time complexity and Runtime

Time Complexity versus 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.

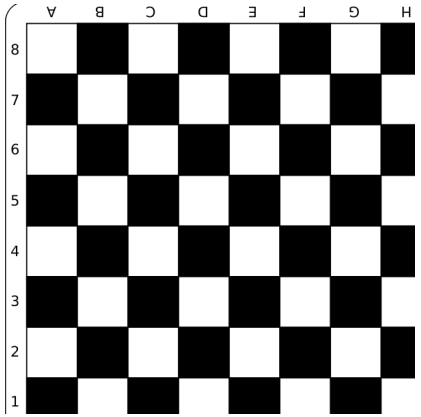
Big-O Notation

If $g(x)$ is a real or complex function, and $f(x)$ is a real function, $g(x)$ is $O(f(x))$ 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.

- An example: QuickSort
- Brute-force versus dynamic programming methods
- Repeated scenarios, redundant calculations

Dominant Strategy of Checkers

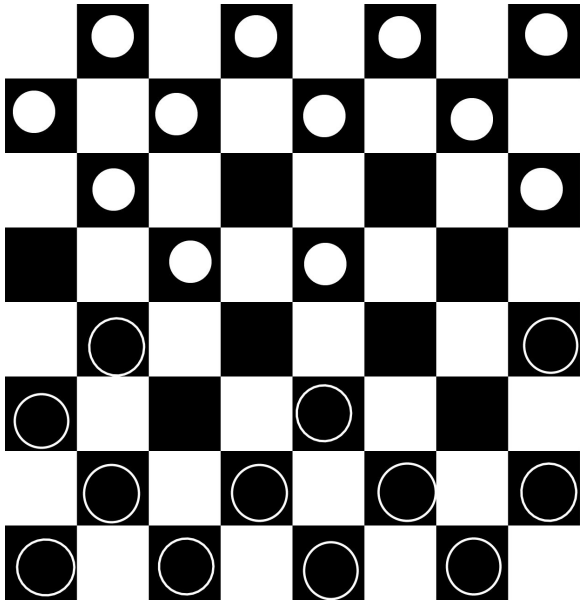
- International rules of checkers
- Dominant strategy does not certify a win
- Use dynamic programming to find the optimal strategy for checkers



A checkers board.

Dominant Strategy of Checkers (Cont.)

- Brute-force algorithm - plays every possible game of checkers



Dominant Strategy of Checkers (Cont.)

- Dynamic programming algorithm: top-down method
- Possible moves of regular capturing pieces and crowned capturing pieces: $O(2^k)$
- Runtime of dynamic programming algorithm is:

Possible moves per piece

$$O(k^2 \times 2^k \times 3^{k\text{^}2})$$

Possible pieces

Possible board configurations