Republic of Algeria

Ministry of Higher Education and Scientific Research

École Supérieure en Informatique - Sidi Bel Abbès

Department: Higher Cycle
Laboratory: LabRI-SBA

Complexity and Problem Solving

Course Support

Academic Year: 2025–2026

Level: 2nd Year – Higher Cycle

Specialization: Artificial Intelligence and Data Science (IASD)

Author:
Laboratory: LabRI-SBA
Email:

Contents

Abstract

This course support document is intended for second-year graduate students in Artificial Intelligence and Data Science (IASD) at the École Supérieure en Informatique de Sidi Bel Abbès. It introduces fundamental and advanced concepts in algorithmic complexity and problem-solving, providing a structured learning path through three chapters.

The first chapter covers algorithmic complexity, including time and space analysis, recurrence relations, asymptotic notations, and the analysis of recursive algorithms. The second chapter focuses on the classification of computational problems, discussing complexity classes (P, NP, NP-complete), undecidability, and the practical implications of the *P vs NP* question. The final chapter applies these concepts to artificial intelligence, covering problem types, search strategies, decomposition methods, and algorithm evaluation, linking theory to practical applications.

This document combines theoretical insights with practical problem-solving strategies and serves as both a reference and a learning tool to strengthen students' understanding of computational complexity and its applications in intelligent systems.

Introduction

This course material is intended for second-year graduate students specializing in **Artificial Intelligence and Data Science (IASD)** at the École Supérieure en Informatique de Sidi Bel Abbès. It serves as a structured pedagogical support for the module *Complexity and Problem Solving*, taught during the first semester.

The polycopie is designed to deepen students' understanding of key theoretical and practical concepts related to algorithmic complexity and problem-solving strategies, with a particular focus on applications in artificial intelligence.

To make the most of this material, students should already be familiar with:

- Fundamental concepts in algorithmics and data structures (both static and dynamic),
- Basics of graph theory and formal languages.

While this document does not primarily focus on solved exercises, it provides numerous examples, explanations, and practice problems to support the comprehension of core ideas. The content aims to:

- Reinforce the theoretical foundations of algorithmic complexity,
- Explore the classification of problems based on computational hardness,
- Introduce practical methods for solving complex or intractable problems,
- Apply abstract notions to real-world AI scenarios,
- Develop systematic approaches for designing and analyzing algorithms.

Objectives

The main objectives of this polycopie are as follows:

- Understand algorithmic complexity and its implications,
- Estimate execution time to compare different algorithmic solutions,
- Identify intractable or undecidable problems,
- Apply complexity theory to real-world AI use cases,
- Develop skills in systematic problem-solving and strategy design.

Structure of the Document

This course material is structured into three main chapters:

- **Chapter 1: Algorithm Complexity**
  This chapter explores the concept of time complexity, its role in evaluating algorithm efficiency, and methods for analyzing and comparing algorithms.
- **Chapter 2: Problem Complexity**
  This chapter addresses the classification of computational problems, introducing complexity classes such as P, NP, NP-complete, and others. Emphasis is placed on understanding how the inherent difficulty of a problem impacts the feasibility of finding a solution.
- **Chapter 3: Problem Solving in Artificial Intelligence**
  This last chapter focuses on search strategies in state spaces, problem decomposition techniques, and intelligent heuristics commonly used in AI.

In addition to theoretical insights, this polycopie provides a valuable learning framework for enhancing programming and analytical skills. It also includes bibliographic references at the end, which supported the development of the material and provide additional reading for interested students.

Chapter 1

Algorithmic Complexity

Introduction

An algorithm is said to be correct if, for each input data, it must produce the correct output. In this case, it is said that the (correct) algorithm solves the given problem. To computationally solve a given problem, an algorithm is implemented on a computer. That is, the algorithm is rewritten using a programming language. However, for a given problem, there are usually several algorithms that can solve it. These algorithms differ from each other in terms of efficiency. These differences can be much more significant than those due to hardware and software. Is there any benefit in choosing algorithms? And if so, how do we choose them?

In computer science, the notion of complexity refers to two concepts:

- Algorithmic Complexity: This is the study of efficiency when comparing algorithms. Indeed, complexity introduces the concept of cost and shows that it is not sufficient to find a correct method to solve a problem; it must also be efficient. Efficiency is measured by the time and space required by an algorithm to solve a problem.
- Problem Complexity: This is the study that allows the classification of problems based on the performance of the best known algorithms that solve them.

This chapter focuses on algorithmic complexity.

1.1 Generalities

1.1.1 Algorithm

The word "algorithm" comes from the name of the 9th-century Arab mathematician, Mohammed Ibn-Moussa Al-Khuwarizmi. Al Khwarizmi proposed fundamental methods for adding, multiplying, and dividing integers, as well as calculating the roots of an equation and the decimals of the number π. These procedures were precise, unambiguous, mechanical, efficient, and correct: they were simply algorithms. It is worth noting that algorithms were known and used long before the advent of computing. The first algorithm is the Euclidean Algorithm (300 BC) for computing the greatest common divisor of two integers. The modern notion of algorithm was developed by logicians in the 1930s (Herbrand, Gödel, Church, and Turing) (Lebbah, 2021). In 1968, Knuth published the first volume of "The Art of Computer Programming," which focused on the modern study of computer algorithms (Knuth, 1997).

There are various ways to define an algorithm. Here, we present a definition that can provide an idea of the concept of an algorithm.

Definition An algorithm is a well-defined computational procedure that takes an input value or a set of values and produces an output value or a set of values. An algorithm is, therefore, a sequence of computational steps that transforms the input value into the output value.

1.1.2 Elementary Operations

An algorithm is a set of elementary computational operations, organized according to specific rules to solve a given problem. For each input of the problem, the algorithm produces an output after a finite number of operations. Elementary operations include usual arithmetic operations, data transfers, comparisons between data, etc. It is useful to consider as truly elementary only those operations with constant computation time, i.e., independent of the size of operands (Aho & Hopcroft, 1974). For example:

- Addition of integers with a bounded size (e.g., 'int' in C) is an elementary operation.
- Addition of integers with arbitrary size is not an elementary operation.
- Similarly, testing membership of an element in a set is not an elementary operation in this sense because its execution time depends on the size of the set, even if some programming languages provide basic instructions to perform this operation.

1.1.3 Algorithm Complexity

Performance is a central concern in computer science. If a problem is solved within a reasonable time, the user is satisfied. Generally, precise time measurement is not essential; only an approximation of the time matters. If the obtained time is perceived as relatively long, the question of a more efficient solution arises. With the machine being unmodified, improvements will focus on how algorithms are designed based on the data volume n to be processed. For example, if by doubling the volume of data, an algorithm returns a result by multiplying the execution time by 2 and another returns it by multiplying the execution time by $n > 2$, we will obviously opt for the first: we says that its time complexity is better. Additionally, space complexity can be defined to account for the memory space used during algorithm execution. The larger the complexity, the more memory zones the program implementing the algorithm will require to store data. In this course, we focus on the study of time complexity.

Why do we mainly address time complexity over space complexity? Modern machines are highly robust with extraordinary hardware capabilities. Algorithms with poor time complexity may become infeasible to run on large datasets, even if they use less memory. We mainly concentrate on time complexity as space complexity is generally less critical. This doesn't mean space complexity is ignored, especially in cases where memory is constrained (e.g., embedded systems).

1.2 Computing Fibonacci Numbers

1.2.1 Recursive Definition

Let (F_n) be the Fibonacci sequence:

$$F_n = \begin{cases} F_{n-1} + F_{n-2} & \text{if } n > 1 \\ 1 & \text{if } n = 1 \\ 0 & \text{if } n = 0 \end{cases}$$

A sequence defined by recurrence is a sequence defined by its first term (or first terms) and a recurrence relation, which defines each term based on the previous one or ones, if they exist.

The Fibonacci sequence is a sequence of integers where each term is the sum of the two preceding terms. The sequence generates the following numbers: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, ...

1.2.2 A First Version of the Algorithm

In programming, pseudo-code is an Algorithm Description Language. It is a way of describing an algorithm in almost natural language, without reference to a particular programming language. Below the pseudo-code for a first naive version to compute Fibonacci Numbers.

```
1 if n = 0 then
2 |  return 0;
3 else
4 |  if n = 1 then
5 |  |  return 1;
6 |  else
7 |  |  return FIB1(n-1)+FIB1(n-2);
8 |  end
9 end
```

Algorithm 1: FIB1(n)

Three essential questions are raised (Lebbah, 2021):

1. Is the algorithm correct[1]?
2. How much time does it take to terminate, depending on n?
3. Can it be improved?

1. **Correctness:** The answer to the first question is quite evident. The algorithm is correct since it precisely implements the recursive definition of the Fibonacci sequence.
2. **Execution Time:** The second question is intricate. Let T(n) be the number of steps needed for the algorithm to compute F_n.

$$T(n) \leq 2 \text{ for } n \leq 1$$

$$T(n) = T(n-1) + T(n-2) + 3 \text{ for } n > 1$$

[1] An algorithm is correct if it performs as expected.

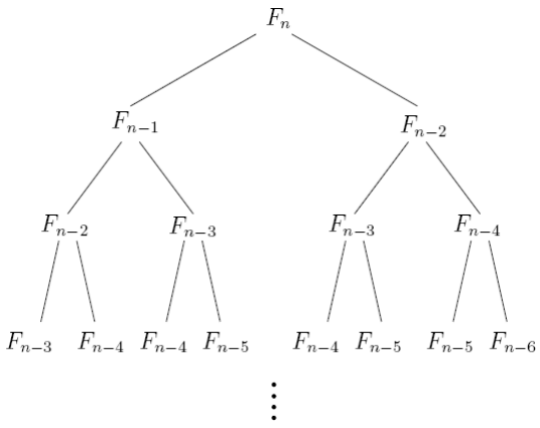


Figure 1.1: The trace obtained when executing FIB1(n) algorithm.

- An approximate solution: $T(n) \approx 2^{0.694n}$
- T(n) exhibits exponential growth
  $\Longrightarrow$ What would be the required time for this calculation on a computer?

For example, let's consider the calculation of F_{200}. The computation of FIB1(n) executes approximately $T(200) \approx 2^{138}$ elementary steps. Let's take a powerful computer today, executing 40 trillion 40×10^{12} instructions per second. On this computer, we would need 2^{92} seconds! This means, waiting until the theoretical extinction of the sun! According to Moore's hypothesis which assumes that computers double their power every 18 months, then, we could arrive at the fact that: if we could reasonably calculate F_{100}, then with Moore's hypothesis , we could probably calculate with the same power F_{101}, ... Meaning that we will gain a number each year. Thus, we are completely discouraged by the harsh reality of exponential growth, which cannot be overcome by any current machine! Can we do better in terms of computation cost? And without relying on machine improvements, which we have just seen are inadequate!

1.2.3 A Polynomial Version

The behavior of FIB1(n) (the trace) is shown in the Figure 1.1.

By analyzing this trace, we can observe that many calls are repeated. We can adopt the trick of storing the already computed values. This leads to the use dynamic programming strategy. This technique consists of storing the values of subproblems to avoid redundant computations. There are two methods to effectively compute a solution: bottom-up method and top-down method.

1. In the top-down method, we write a recursive algorithm, but we use memoization to avoid solving the same problem multiple times. In other words, we store the results of recursive calls in an array F[.]:

```
1 if F[n] ≠ null then return F[n] ;
2 if n = 0 or n == 1 then F[n] = n ;
3 else F[n] = FIB3(n − 1) + FIB3(n − 2) ;
4 return F[n] ;
```

Algorithm 2: FIB3(n)

2. In the bottom-up method, we start by calculating solutions for the smallest subproblems, then, step by step, we compute solutions for larger problems and store the results in an array F[.] (see Fib2).

```
1  if n = 0 or n == 1 then
2  |  return n;
3  end
4  Create an array F[0..n];
5  F[0] ← 0;
6  F[1] ← 1;
7  for i ← 2 to n do
8  |  F[i] ← F[i − 1] + F[i − 2] ;
9  end
10 return F[n];
```

Algorithm 3: FIB2(n)

- The algorithm is obviously correct since, once again, it follows the recurrent definition of Fibonacci.
- How much does this algorithm require to compute F_n as a function of n? The loop in the algorithm requires n - 1 calculation steps. Therefore, the execution time of this algorithm is linear in n.
- It is now possible to compute F_{200000} with little effort[2]

1.3 Algorithm: Correctness and Complexity

Algorithm Correctness An algorithm is correct if, for every input instance, it terminates with the correct desired output. The loop invariant in an algorithm is a property that holds true at:

1. Before starting the loop
2. After each iteration of the loop
3. At the end of the loop

To prove the invariant, we need to demonstrate (Cormen, Leiserson, Rivest, & Stein, 2022) :

INITIALIZATION Show that the loop invariant is true before the first loop iteration.

MAINTENANCE Show that if the loop invariant is true before a loop iteration, it remains true before the next iteration. Maintenance is also referred to as Conservation.

TERMINATION Once the loop is finished, the invariant provides a useful property that helps demonstrate the algorithm's correctness.

Establishing this proof systematically provides the proof of the algorithm's correctness.

[2]Designing a good algorithm can lead to exponential savings!

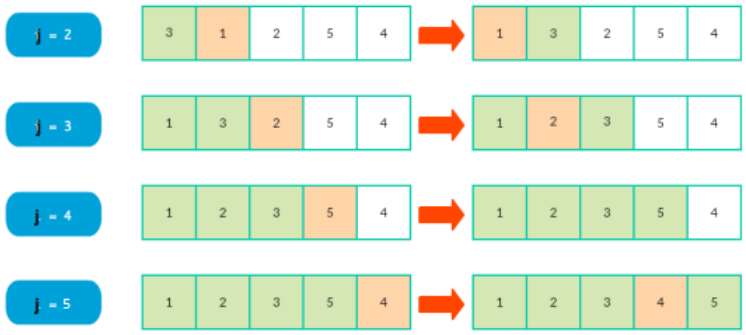


Figure 1.2: Illustration of how the Insertion Sort algorithm works.

1.3.1 Insertion Sort

Sorting Problem Definition To illustrate our approach, we address a frequently encountered practical problem: sorting a sequence of numbers in ascending order. Below, we present a formal definition of this problem.

Input A sequence of n numbers $\langle a_1, a_2, ..., a_n \rangle$

Output A permutation $\langle a'_1, a'_2, ..., a'_n \rangle$ of the input sequence such that $a'_1 \leq a'_2 \leq ... \leq a'_n$

Given an input sequence such as $\langle 5, 90, 45, 89 \rangle$, a sorting algorithm will produce an output of $\langle 5, 45, 89, 90 \rangle$. Such an input sequence is referred to as an instance of the problem.

Insertion Sort: Algorithm A first solution to the sorting problem: Insertion Sort (see Algorithm 4).

Data: A sequence of n numbers $A = \langle a_1, a_2, ..., a_n \rangle$
Result: A permutation $\langle a'_1, a'_2, ..., a'_n \rangle$ of the input sequence such that $a'_1 \leq a'_2 \leq ... \leq a'_n$

```
1  for j ← 2 to length(A) do
2      pivot ← A[j] ;
3      {Insert A[j] into the sorted sequence A[1..j − 1]};
4      i ← j − 1 ;
5      while (i > 0) ∧ (A[i] > pivot) do
6          A[i + 1] ← A[i] ;
7          i ← i − 1 ;
8      end
9      A[i + 1] ← pivot ;
10 end
```

Algorithm 4: INSERTION-SORT(A)

Insertion sort considers each element of the array and inserts it at the correct position among the already sorted elements. Therefore, when considering an element, the elements preceding it are already sorted, while the elements following it are not yet sorted. Figure 1.2 illustrates how the algorithm works.

Exercise 1 Prove the correctness of the Insertion Sort algorithm defined in Algorithm 4.

Solution **Loop Invariant of Insertion Sort:** The subarray A[1..j-1] is sorted. We will utilize this property (the invariant) to demonstrate the correctness of the Insertion Sort algorithm.
Proof of correctness of Insertion Sort:

- **INITIALIZATION:** Before the first iteration of loop 1, the subarray A[1..j-1] contains only one element (j=2). An array with a single element is inherently sorted (trivial), hence our invariant "the subarray A[1..j-1] is sorted" holds true.

- **MAINTENANCE:** Overall, what does loop 1 do? It moves the elements A[j-1], A[j-2], A[j-3], etc. one position to the right until the correct position for A[j] is found. After these right shifts are performed, the value A[j] is inserted at the right place. During an iteration of loop 1, we transition from an unsorted array A[1..j] (subarray A[1..j-1] is sorted, but element A[j] is not yet in the correct place, so A[1..j] is not sorted yet) to a sorted array (shifting elements A[j-1], A[j-2], A[j-3], etc. to the right and placing A[j] at the correct position). At the end of an iteration (just before "j $\leftarrow$ j+1"), A[1..j] is sorted, and right after "j $\leftarrow$ j+1", the previously mentioned array A[1..j] becomes the array A[1..j-1]. Thus, we can affirm that if the invariant "the subarray A[1..j-1] is sorted" is true before an iteration of loop 1, it remains true before the next iteration.

- **TERMINATION:** When the loop terminates, we must have j = n + 1 (for an array of n elements). If we substitute n+1 with j in the loop invariant, we obtain the subarray A[1..n]. When the loop ends, the statement "the subarray A[1..j-1] is sorted" becomes "the subarray A[1..n] is sorted." Now, the subarray A[1..n] corresponds to the array composed of n elements. Hence, the original array is now sorted, demonstrating the correctness of the algorithm.

1.3.2 Concept of input size

The time taken by the Insertion Sort procedure depends on the size of its input: sorting a thousand numbers takes more time than sorting three numbers. The concept of input size depends on the problem under study; sometimes it is more appropriate to describe the input size with two numbers. For example, in a problem related to graphs, the input size could be the number of vertices and the number of edges. For each problem studied, we will specify the utilized measure of size. In the context of computational complexity, the size of a problem refers to a quantitative mesure that indicates the amount of the input required to define the problem.

1.3.3 Insertion Sort: Complexity

The execution time of an algorithm on a particular input is the number of elementary operations executed. The execution time of the algorithm is the sum of the execution times of each instruction executed (Lebbah, 2021). An instruction i that executes in time c_i and is executed n times contributes $c_i n$ to the total.

Exercise 2 Compute the complexity of the Insertion Sort algorithm defined in Algorithm 4.

Solution To calculate $T(n)$, the execution time of Insertion Sort, we sum the products of costs by the number of occurrences. If the sequence is already ordered, we get:
$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5(n-1) + c_9(n-1) = (c_1 + c_2 + c_4 + c_5 + c_9)n + (-c_2 - c_4 - c_5 - c_9)$

We can express this final formula in the form $an + b$, where a and b are constants. We say the algorithm is in $O(n)$. We have calculated the cost of the Insertion Sort algorithm in the best-case scenario.

If the sequence is in reverse order, we are in the worst-case scenario:
$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5 \sum_{j=2}^{n} t_j + c_6 \sum_{j=2}^{n}(t_j - 1) + c_7 \sum_{j=2}^{n}(t_j - 1) + c_9(n-1)$
where t_j is the number of iterations of the second loop at iteration j of the first loop.

We need to compare each element $A[j]$ with each element of the sorted subarray $A[1..j-1]$, so $t_j = j$ for $j = 2, 3, ..., n$.

A sequence (U_n) is called arithmetic if, for every natural number n, we have: $U_{n+1} = U_n + r$, where r is the common difference of this sequence. Given that: $\sum_{j=2}^{n} j = \frac{n(n+1)}{2} - 1$ and $\sum_{j=2}^{n}(j-1) = \frac{n(n-1)}{2}$

We get:
$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5(\frac{n(n+1)}{2} - 1) + c_6(\frac{n(n-1)}{2}) + c_7(\frac{n(n-1)}{2}) + c_9(n-1)$
$T(n) = (\frac{c_5}{2} + \frac{c_6}{2} + \frac{c_7}{2})n^2 + (c_1 + c_2 + c_4 + \frac{c_5}{2} - \frac{c_6}{2} - \frac{c_7}{2} + c_9)n + (-c_2 - c_4 - c_5 - c_9)$

This final cost is of the form $an^2 + bn + c$ where a, b, and c are constants. We say the algorithm is in $O(n^2)$. We have calculated the cost of the Insertion Sort algorithm in the worst-case scenario, which is the longest execution time for any input of size n. This is the most commonly used cost measure in practice. In the following, we will explore several methods for calculating this cost.

1.4 Notion of Algorithm Complexity

Studying the complexity of an algorithm[3] involves predicting the necessary resources (memory, computation time) for the program that implements that algorithm. Sometimes, the relevant resources are:

- used memory (spatial complexity);
- but in most cases, we want to measure computation time (time complexity).

Time Complexity quantifies the time required for an algorithm to execute based on the length of the input.

Space Complexity quantifies the amount of memory space taken by an algorithm to execute based on the length of the input.

In this course, we emphasize time complexity, and it's this notion that we will delve into. We will use a model of computation called a Random Access Machine (RAM) with a single processor (Cormen et al., 2022). In this model, instructions are executed sequentially without simultaneous operations.

Consider a given problem and an algorithm to solve it. For input data x of size n, the algorithm requires a certain amount of time, measured in the number of elementary operations, denoted as $c(x)$.

Elementary operations include common arithmetic operations, data transfers, data comparisons, etc. It is useful to consider as truly elementary only those operations with constant computation time, i.e., independent of operand size. **For example:**

[3] Algorithm Analysis $\Longleftrightarrow$ Study of Algorithm Complexity

- Addition of bounded-size integers (e.g., `int` in C) is an elementary operation.
- Addition of arbitrary-size integers is not elementary.
- Similarly, testing membership of an element in a set is not an elementary operation in this sense, as its execution time depends on the set size, even if some programming languages have basic instructions to perform this operation.

The time cost obviously varies with the data size, but it can also vary among different data of the same size n.

For example: Let's consider the bubble sort algorithm, which takes a sequence $\langle a_1, ..., a_n \rangle$ of distinct real numbers to be sorted in ascending order. It searches for the first descent, i.e., the smallest index i such that $a_i > a_{i+1}$, swaps these two elements, and repeats the process on the resulting sequence.

Bubble sort is a sorting algorithm that repeatedly steps through the list, compares adjacent elements, and swaps them if they are in the wrong order.

The number of inversions performed:

- 0 for a sorted sequence.
- $\frac{n(n-1)}{2}$ for a decreasing sequence[4].

Our goal is to evaluate the cost of an algorithm according to certain criteria, and based on the data size n.

1.4.1 Fundamental Operations

For certain problems, we can identify one or more operations that are fundamental (the most important elementary operations) in the sense that the execution time of an algorithm solving this problem is always proportional to the number of these operations (Cormen et al., 2022). It is then possible to compare algorithms dealing with this problem using this simplified measure.

Examples:

Problem	**Fundamental Operations**
Searching for an element in a list	Comparison between this element and entries in the list
Sorting a list of elements	Comparison between elements and element rearrangement
Multiplying two matrices of numbers	Multiplication and addition of two numbers
Primality testing of a given integer n	Division

If a very precise microanalysis of program execution time is desired, one can decide that all elementary operations of the program are fundamental.

[4] $\sum_{i=1}^{n}(i-1) = \frac{n(n-1)}{2}$

1.4.2 Rules for Counting the Number of Fundamental Operations

After identifying the fundamental operations, the next step is to count the number of operations of each type. There is no complete system of rules for counting the number of operations based on the syntax of algorithms, but some observations can be made.

Sequence When operations are in a sequence of instructions, their numbers are added together.

Example for the Complexity of a sequence of instructions: Let's take the following sequence of instructions:

$S =$ Begin
$i_1; i_2; ...; i_n$
End

Therefore, $Nb(S) = \sum_{p=1}^{n} Nb(i_p)$
$Nb(i_k)$ refers to the number of fundamental operations contained in algorithmic instruction i_k.

If-Then-Else For conditional branches, it is generally difficult to determine which branch of the condition is executed, and therefore which operations to count. However, the number of operations can be upper-bounded.

Example for the Complexity of a conditional statement: Let's take the following program portion:
Cond = if Expr then E1 else E2
Therefore, $Nb(Cond) \leq Nb(Expr) + \max(Nb(E_1), Nb(E_2))$

Loops For loops, the number of operations in the loop is $\sum P(i)$, where i is the loop control variable, and $P(i)$ is the number of fundamental operations during the execution of the ith iteration. If the number of iterations is difficult to calculate, one can use a good upper bound.

Example for the Complexity of Loops: Let's consider the program segment below:

Iter = Iteration Expr(i)
S
endIteration

Therefore: Nb(Iter) = [Nb(S) + Nb(Expr(i))] × Nb_Iterations
So, for the case of a For loop:

Iter = For I := a to b do
Begin
i_1 ; i_2 ; ... ; i_n
End;

$Nb(Iter) = (|b - a| + 1) \times (\sum_{p=1}^{n} Nb(i_p) + 1)$
The complexity of the continuation condition $I \leq b$ (Expr(i)) is: 1.

Procedure Calls For non-recursive procedure and function calls, we can calculate the complexity of these calls and take them into account based on their nesting in the algorithm.

Recursive Calls For recursive procedure and function calls, counting the number of fundamental operations generally leads to solving recurrence relations. Indeed, the

number $T(n)$ of operations in a procedure call with an argument of size n is expressed, based on recursion, in terms of various $T(k)$ for $k < n$. The example of Fibonacci given in the introductory chapter illustrates this case. **Example for the Complexity of Subroutine Calls:**

1. Without recursion:
   Example:
   Subroutine A:
   $\boxed{I_1 \text{ ; } I_2 \text{ ; Call of B; } I_3 \text{ ;}}$
   $Nb(A) = Nb(I_1) + Nb(I_2) + Nb(B) + Nb(I_3)$

2. In case of recursion, calculating Nb(A) leads to the resolution of a recurrence relation.
   Example:
   Function fact(n:integer):integer
   Begin
   If (n=0) then fact ← 1 else fact ← n × fact(n-1) ;
   End;
   The number $T(n)$ of fundamental operations ($\times$) satisfies:
   $T(0) = 0$ and $T(n) = T(n-1) + 1$ for $n \geq 1$
   By direct resolution, we obtain: $T(n) = n$

1.4.3 Data Size

It is evident that the calculation of an algorithm's cost depends on the data it operates on.

First, it is necessary to define a measure of size for the data that reflects the amount of contained information. **Example** When adding or multiplying integers, a meaningful measure is the number of digits in the numbers.

For certain algorithms, the execution time depends not only on the size of the data, but most of the time, the complexity also varies for fixed data size based on the specific data. Thus, the number of operations will vary according to the nature of the data, leading to the use of complexities in the worst-case, best-case, and average scenarios.

Example Let's consider a problem $P(n)$ with data size n. Let x be an instance of problem P. For example, let $P(n)$ be the problem of arranging a list of length n in ascending order. An instance x could be sorting the array $\langle 5, 7, 12, 6, 8\rangle$ where $n = 5$. Another instance could be $\langle 154, 45, 89, 12, 1547, 4\rangle$, ...

The problem is formally defined for a unique understanding, while its number of instances is generally infinite.

The challenge in computing complexity is to reason about this infinity of instances by establishing a function $c(n)$ that characterizes the number of operations based on the variable n, representing the data size.

1.4.4 Types of Analysis (Best and Worst Cases)

Best-Case Complexity
The cost $Min_A(n)$ of an algorithm A in the best-case scenario for a problem $P(n)$ is

defined as:

$$Min_A(n) = \min_{|x|=n} c(x)^5$$

Worst-Case Complexity
The cost $Max_A(n)$ of an algorithm A solving problem $P(n)$ in the most unfavorable or worst-case scenario is defined as the maximum cost over all data of size n:

$$Max_A(n) = \max_{|x|=n} c(x)$$

Average-Case Complexity In situations where the worst-case scenario is expected to be rare, we are more interested in the average cost of the algorithm. A correct formulation of this average cost[6] assumes that a probability distribution over data of size n is known. If $p(x)$ is the probability of data x, the average cost $Avg_A(n)$ of an algorithm A solving problem $P(n)$ on data of size n is defined as (Gaudel, Soria, & Froidevaux, 1987; Zampieri, Riviere, & Amerein-Soltner, 2018):

$$Avg_A(n) = \sum_{|x|=n} p(x)c(x)$$

In most cases, a uniform distribution is assumed, meaning $p(x) = \frac{1}{|D_n|}$[7]. In this case, the expression for average cost becomes:

$$Avg_A(n) = \frac{1}{|D_n|} \sum_{|x|=n} c(x)$$

Uniform distribution means that all configurations of data with a fixed size n are equally probable.

Average-case complexity and extreme complexities are related by the inequality:

$$Min_A(n) \leq Avg_A(n) \leq Max_A(n)$$

If the algorithm's behavior depends solely on the data size (as in the example of matrix multiplication), then these three quantities coincide. However, in general, this is not the case, and we don't even know if the average cost is closer to the minimal or maximal cost.

1.5 Recursion and Divide and Conquer Approach

1.5.1 Divide and Conquer approach description

Many useful algorithms follow a recursive structure: to solve a given problem, they call themselves recursively one or more times on smaller, very similar subproblems. These algorithms embrace the "divide and conquer" approach: they break the problem into multiple subproblems similar to the original problem but smaller in size, solve the subproblems recursively, and then combine these solutions to recover a solution to the original problem. This problem-solving paradigm involves three steps at each level of recursion (see Algorithm 6) (Gaudel et al., 1987; Lebbah, 2021):

[5] The execution cost of the algorithm on data x

[6] Average-case complexity is much harder to determine than worst-case complexity, partly because the analysis becomes mathematically challenging and also because it is not always easy to establish a suitable probability model for the problem.

[7] D_n is the set of data of size n

Divide the problem into a number of subproblems

Conquer the subproblems by solving them recursively. Moreover, if the size of a subproblem is small enough, it can be solved directly.

Combine the solutions to the subproblems into a complete solution for the original problem.

```
1 if Small(P) then Solution(P) ;
2 else
3 |  Divide P into P_1, P_2, ..., P_k ;
4 |  D&C_Recursive(P_1) ; D&C_Recursive(P_2) ; ... ; D&C_Recursive(P_k) ;
5 |  Combine(D&C_Recursive(P_1), D&C_Recursive(P_2), ..., D&C_Recursive(P_k)) ;
6 end
```

Algorithm 5: D&C_Recursive(P)

Exercise 3 The merge sort algorithm closely follows the "divide and conquer" rule:

Divide Divide the array $T[1, ..n]$ to be sorted into two subarrays $T[1, ..n/2]$ and $T[n/2 + 1, ..n]$

Conquer Sort the two subarrays $T[1, ..n/2]$ and $T[n/2+1, ..n]$ (recursively, or do nothing if they have size 1)

Combine Merge the two sorted subarrays $T[1, ..n/2]$ and $T[n/2 + 1, ..n]$

Write the Merge Sort algorithm in pseudocode and provide the execution trace when applying the algorithm to the vector $[70, 50, 30, 10, 20, 40]$.

Solution The algorithm is :

Data: A sequence of n numbers $A = \langle a_1, a_2, ..., a_n \rangle$
Result: A permutation $\langle a'_1, a'_2, ..., a'_n \rangle$ of the input sequence such that $a'_1 \leq a'_2 \leq ... \leq a'_n$

```
1 if p < r then
2 |  q ← ⌊(p + r)/2⌋;
3 |  MergeSort(A, p, q);
4 |  MergeSort(A, q + 1, r);
5 |  Merge(A, p, q, r);
6 end
```

Algorithm 6: MergeSort(A, p, r)

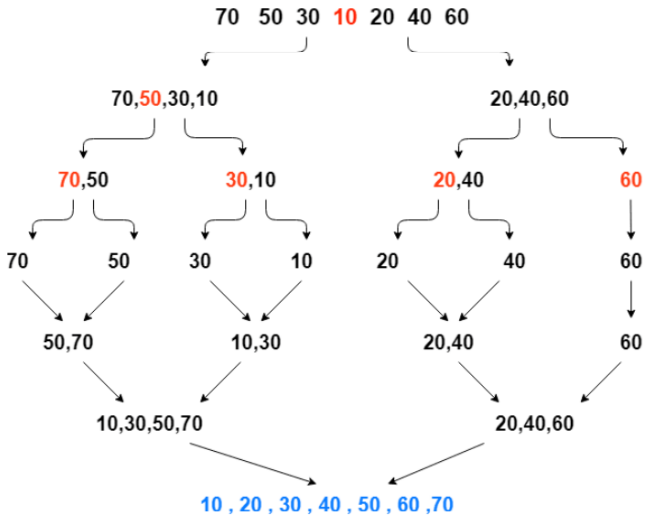


Figure 1.3: Execution of the Merge Sort algorithm – illustration.

1 Tmp: temporary array of size $end - start + 1$;
2 $i \leftarrow 1$; $i_1 \leftarrow start$; $i_2 \leftarrow middle + 1$;
3 **while** $(i_1 \leq middle) \wedge (i_2 \leq end)$ **do**
4 | **if** $(Tab[i_1] \leq Tab[i_2])$ **then**
5 | | $Tmp[i] \leftarrow Tab[i_1]$; $i_1 \leftarrow i_1 + 1$;
6 | **else**
7 | | $Tmp[i] \leftarrow Tab[i_2]$; $i_2 \leftarrow i_2 + 1$;
8 | **end**
9 | $i \leftarrow i + 1$;
10 **end**
11 **while** $(i_1 \leq middle)$ **do** $Tmp[i] \leftarrow Tab[i_1]$; $i_1 \leftarrow i_1 + 1$; $i \leftarrow i + 1$;
12 **while** $(i_2 \leq end)$ **do** $Tmp[i] \leftarrow Tab[i_2]$; $i_2 \leftarrow i_2 + 1$; $i \leftarrow i + 1$;
13 **for** $i \leftarrow start$ **to** end **do** $Tab[i] \leftarrow Tmp[i - start + 1]$;

Algorithm 7: Merge$(Tab, start, middle, end)$

Illustration of Merge Sort (see Figure 1.3) (Lebbah, 2021):

1.5.2 Complexity of Recursive Algorithms

When an algorithm contains a recursive call to itself, its execution time can often be described by a recurrence equation. This equation describes the overall execution time for a problem of size n in terms of the execution time for smaller-sized inputs. We can use mathematical tools to solve the recurrence and find bounds for the algorithm's performance.

1.5.3 Computing the Complexity of a Divide and Conquer Algorithm

- Let $T(n)$ be the execution time of a problem of size n.
- If the problem size is sufficiently small, say $n \leq c$ for some constant c, the solution takes constant time, denoted as $\Theta(1)$.
- Suppose we divide the problem into a subproblems, each of size $1/b$ of the initial problem size.

- If it takes time $D(n)$ to divide the problem into subproblems and time $C(n)$ to combine the solutions to the subproblems into the final solution, we obtain the recurrence:

$$T(n) = \begin{cases} \Theta(1) & \text{if } n \leq c \\ aT(n/b) + D(n) + C(n) & \text{otherwise} \end{cases}$$

Exercise 4 Compute the complexity of the Merge Sort algorithm, assuming that the size of the input vector is a power of 2.

Solution

- We assume that the size of the initial problem is a power of two[8].
- Each "divide" step then generates two sub-sequences of exactly size $n/2$. We will see later that this assumption does not affect the order of magnitude of the recurrence solution.
- Merge sort on a single element takes constant time.
- With $n > 1$ elements, we segment the execution time as follows:

  Divide It only involves computing the middle of the subsequence, which takes constant time, $D(n) = \Theta(1)$.

  Conquer We recursively solve two subproblems, resulting in $2T(n/2)$.

  Combine It can be shown that the merging procedure on a subproblem of size n is of the form $C(n) = an + b$, where a and b are constants. We will denote this as $C(n) = \Theta(n)$.

Therefore, we have:

$$T(n) = \begin{cases} \Theta(1) & \text{if } n \leq 1 \\ 2T(n/2) + \Theta(n) & \text{if } n > 1 \end{cases}$$

We will prove later that $T(n)$ is of the form $an\log(n) + bn + c$, where a, b, and c are constants. We denote this form as $\Theta(n\log(n))$.

Note: This cost is significantly lower than that of insertion sort, as $\exists n_0, \forall n, n > n_0, n^2 > n\log(n)$.

1.5.4 Fast Exponentiation

The goal in fast exponentiation problem is to compute x^n for given values of x and n, by calculating the complexity with respect to n. The naive method using the definition below results in linear complexity (Lebbah, 2021) :

$$x^n = \begin{cases} 1 & \text{if } n = 0 \\ x^{n-1} \times x & n > 0 \end{cases}$$

[8]Our recurrence-based analysis is simplified if we assume that the size of the initial problem is a power of two.

By using the following observations, we can improve further the complexity:

$$x^n = \begin{cases} 1 & \text{if } n = 0 \\ (x^2)^{n/2} & \text{if n is even} \\ x(x^2)^{(n-1)/2} & \text{if n is odd} \end{cases}$$

The first recursive definition of the power function $x^3 = x^2 \times x = (x^1 \times x) \times x = ([x^0 \times x] \times x) \times x$ (this definition requires 3 multiplications to compute x^3). The recursive definition of multiplication count for power function:

$$T(n) = \begin{cases} 0 & \text{if } n = 0 \\ T(n-1) + 1 & n > 0 \end{cases}$$

Second definition

- If n is even, x^n is equivalent to computing the power of x^2 to $n/2$.
- If n is odd, x^n is equivalent to multiplying x by the power of x^2 to $(n-1)/2$

$$T'(n) = \begin{cases} 0 & \text{if } n = 0 \\ T'(\frac{n}{2}) + 1 & \text{if n is even} \\ T'(n-1) + 1 & \text{if n is odd} \end{cases}$$

This definition speeds up the process towards the base case.

Exercise 5 Provide the algorithm in pseudocode for fast exponentiation.
Determine its complexity in terms of the number of multiplications required to compute x^{100}.
Then, deduce its general complexity as a function of an arbitrary exponent n.

Solution The algorithm for fast exponentiation is written below (see Algorithm 8):

```
1 if (n = 0) then
2 |   return 1;
3 else
4 |   if n is even then
5 |   |   return FastExp(x × x, n/2);
6 |   else
7 |   |   return x × FastExp(x × x, (n − 1)/2);
8 |   end
9 end
```

Algorithm 8: FastExp(x,n)

T'(100) = 1 + T'(50)
1 + T'(25)
1 + T'(24)
1 + T'(12)
1 + T'(6)
1 + T'(3)
1 + T'(2)
1 + T'(1)
1 + T'(0)

The complexity of x^{100} is equal to 9. There is a significant improvement from the linear complexity (100 multiplications) to the complexity of this second definition, which requires only 9 multiplications to compute x^{100}.

Hence, the complexity of fast exponentiation is $O(\log n)$.

1.6 Asymptotic Analysis and Function Magnitudes

We have determined the complexity of an algorithm as a function of the size of the data. It is crucial to understand now the rate of growth of this function as the data size increases (Cormen et al., 2022; Lebbah, 2021). For solving a small-sized problem, the chosen method matters less. However, for a large-sized problem, performance differences between algorithms can be substantial. In most cases, a simple approximation of the complexity function is sufficient to determine whether an algorithm is usable or not, or to compare different algorithms.

Example For a large n, it is generally of secondary importance whether an algorithm performs $n + 1$ or $n + 2$ operations.

Sometimes, multiplicative constants are also of little importance: Let's consider comparing algorithm A_1 with complexity $M_1(n) = n^2$ and algorithm A_2 with complexity $M_2(n) = 2n$. A_2 is better than A_1 for almost all n ($n > 2$). Similarly, if $M_1(n) = 3n^2$ and $M_2(n) = 25n$, A_2 is better than A_1 for $n > 8$.

Neglecting a term in an addition: Regardless of the multiplicative constants k_1 and k_2 such that $M_1(n) = k_1 n^2$ and $M_2(n) = k_2 n$, algorithm A_2 is always better than A_1 for some value of n, as the function $f(n) = n^2$ grows much faster than the function $g(n) = n$. Indeed:

$$\lim_{n\to\infty} \frac{g(n)}{f(n)} = 0$$

We say that the asymptotic order of magnitude of $f(n)$ is strictly larger than that of $g(n)$. In this case, we can write that $f(n)$ is of the same order as $f(n) + g(n)$.

Neglecting a Constant: On the other hand, regardless of the multiplicative constants k_1 and k_2 such that $M_1(n) = k_1 f(n)$ and $M_2(n) = k_2 f(n)$, algorithm A_2 is always of the same order as A_1. In fact:

$$\lim_{n\to\infty} \frac{k_1 f(n)}{k_2 f(n)} = \frac{k_1}{k_2}$$

In this case, we can state that $M_1(n)$ is of the same order as $M_2(n)$.

The asymptotic orders of magnitude of the functions 1, $\log_2(n)$, $n \log_2(n)$, n^2, n^3, 2^n strictly increase; these functions form a scale of comparison[9] (see Figure 1.4) (Cormen et al., 2022; Aho & Hopcroft, 1974).

To analyze the complexity $M_A(n)$ of an algorithm A, we aim to determine the asymptotic order of magnitude of $M_A(n)$. We search within a comparison scale, which may be more comprehensive than that formed by the functions in the Figure 1.4, for a function that has a similar growth rate to $M_A(n)$.

Suppose we need to compare two algorithms A_1 and A_2 with complexities $M_{A_1}(n)$ and $M_{A_2}(n)$. If the order of magnitude of $M_{A_1}(n)$ is strictly greater than the order of magnitude of $M_{A_2}(n)$, then we can immediately conclude that A_2 is better than A_1 for large n. However, if $M_{A_1}(n)$ and $M_{A_2}(n)$ have the same asymptotic order of magnitude, a more detailed analysis is required to compare A_1 and A_2.

[9] A scale of comparison is a set of reference functions ordered by asymptotic order of magnitude.

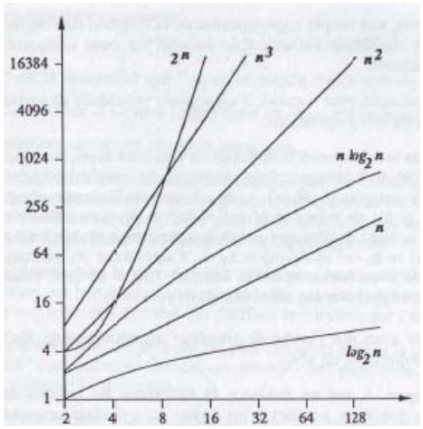


Figure 1.4: Growth of Certain Common Functions

1.7 Landau Notations

To compare the asymptotic order of magnitude of complexity functions, we commonly use the following notion (Knuth, 1997):

Définition. Given two functions f and g from $\mathbb{N}$ to $\mathbb{R}^+$,

$$f = O(g) \Leftrightarrow \exists c \in \mathbb{R}^{+*}, \exists n_0 \in \mathbb{N} \text{ such that } \forall n > n_0, f(n) \leq c.g(n)$$

It is also said that:

$$f = \Omega(g) \Leftrightarrow g = O(f)$$

Interpretation of the mathematical definition: computing the exact number of operations $f(n)$ is in general out of reach for various reasons. This difficulty is overcome by finding a function g that bounds and dominates the number of operations.

$f = O(g)$ means that $f(n)$ is bounded above by $g(n)$ with a positive constant factor for sufficiently large n.

$f = \Omega(g)$ means that $f(n)$ is bounded below by $g(n)$ with a positive constant factor for sufficiently large n.

Thus, $f = O(g)$ means that the asymptotic order of magnitude of f is less than or equal to that of g. We say that f is asymptotically dominated by g. **For example:** $2n = O(n^2)$, and also $2n = O(n)$.

This notion provides an upper bound on the asymptotic order of magnitude of f.

Bounding the complexity's magnitude is not always sufficient to compare the performance of different algorithms with the same magnitude, as one needs to know the exact orders of magnitudes, not just upper bounds. When we say that the complexity $M_A(n)$ of an algorithm A is in $h(n)$, we mean that its asymptotic order of magnitude is precisely $h(n)$[10].

This notion is formalized as follows (Knuth, 1997):

Définition. Given two functions f and g from $\mathbb{N}$ to $\mathbb{R}^+$,

$$f = \Theta(g) \Leftrightarrow \exists c, d \in \mathbb{R}^{+*}, \exists n_0 \in \mathbb{N} \text{ such that } \forall n > n_0, d.g(n) \leq f(n) \leq c.g(n)$$

Or equivalently,

$$f = \Theta(g) \Leftrightarrow f = O(g) \text{ and } g = O(f)$$

[10] $h(n)$ is the smallest upper bound

$f = \Theta(g)$ means that the ratio $\frac{f(n)}{g(n)}$ is bounded between two positive constants for sufficiently large n.

If $f = \Theta(g)$, we say that f and g have the same asymptotic order of magnitude. The Θ notation is more precise than the O notation. **For example:** $2n = \Theta(n)$, but $2n$ is not in $\Theta(n^2)$.

However, in most algorithmic works, analysis results are presented in the form of $O(f(n))$, even though a precise count of fundamental operations often leads to the conclusion that the complexity is exactly $\Theta(f(n))$. **For example:** It is commonly stated that heapsort is in $O(n \log(n))$, but in fact, it is in $\Theta(n \log(n))$.

To sum up:
$f = O(g)$, if there exists a strictly positive constant c and an integer n_0 such that: $n > n_0$, $f(n) \leq c.g(n)$, see the first plot in the Figure 1.5.

$f = \Omega(g)$, if there exists a strictly positive constant c and an integer n_0 such that: $n > n_0$, $f(n) \geq c.g(n)$, see the third plot in the Figure 1.5.

$f = \Theta(g)$, if there exist two strictly positive constants c and d and an integer n_0 such that: $n > n_0$, $d.g(n) \leq f(n) \leq c.g(n)$, see the second plot in the Figure 1.5.

Figure 1.5 provides an illustration of these Landau notations.

1.8 Execution Time and Data Size

The notion of order of magnitude of the complexity of algorithms has significant practical importance. Let's consider a scenario where we have seven algorithms to solve a given problem, and their worst-case complexities have the following order of magnitude: 1 (constant function, independent of data size), $log_2(n)$ (logarithmic function base 2), n (linear function), $nlog_2(n)$ (quasi-linear function), n^2 (polynomial function of degree 2), n^3 (polynomial function of degree 3), and 2^n (exponential function).

The table in Figure 1.6 (Cohen, n.d.; Lebbah, 2021) provides an estimate of the execution time for each algorithm for different data sizes n of the problem on a computer capable of performing 10^6 operations per second.

It clearly demonstrates that as the size of the data increases, the differences between the various execution times become more pronounced.

This table in Figure 1.7 (Lebbah, 2021) provides an estimate of the maximum data size that can be processed by each of the algorithms within a fixed execution time (and still on a computer performing 10^6 operations per second).

According to these two tables, it is evident that certain algorithms are usable for solving problems on computers, while others are not or are less practical.

Algorithms that are usable for large data sizes are those that execute in the following time complexities:

Constant Example: Accessing the k-th element of an array

Logarithmic Example: Binary search, or in binary search trees

Linear Example: Sequential search

Quasi-Linear (n.log(n)) Example: Efficient sorting algorithms

- Algorithms that take polynomial time, i.e., $\Theta(n^k)$ with $k > 0$, are truly usable only for $k < 2$.

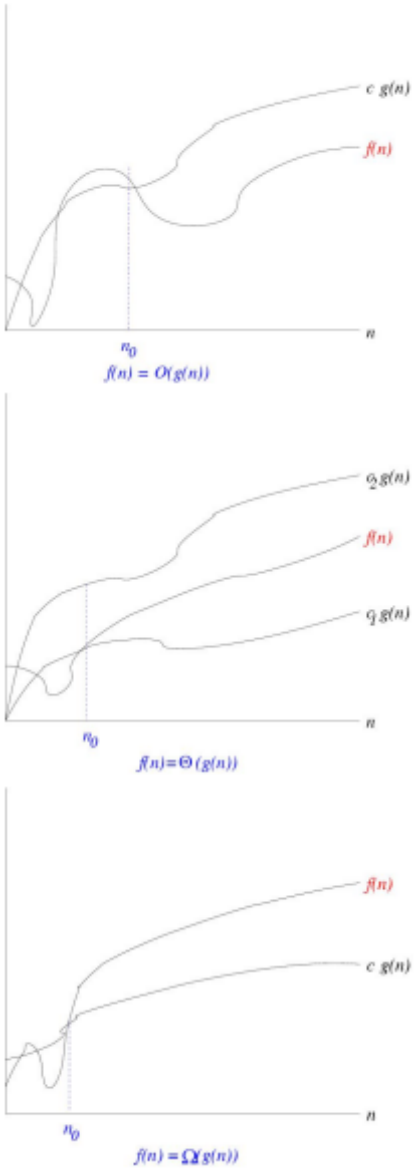


Figure 1.5: Plots illustrating Landau notations.

Complexity / size	1	$\log_2 n$	n	$n\log_2 n$	n^2	n^3	2^n
$n = 10^2$	$\simeq 1\mu s$	6,6 μs	0,1 ms	0,6 ms	10 ms	1 s	4×10^{16} a
$n = 10^3$	$\simeq 1\mu s$	9,9 μs	1 ms	9,9 ms	1 s	16,6 mn	∞
$n = 10^4$	$\simeq 1\mu s$	13,3 μs	10 ms	0,1 s	100 s	11,5 j	∞
$n = 10^5$	$\simeq 1\mu s$	16,6 μs	0,1 s	1,6 s	2,7 h	31,7 a	∞
$n = 10^6$	$\simeq 1\mu s$	19,9 μs	1 s	19,9 s	11,5 j	$31,7 \times 10^3$ a	∞

1μs (microseconds) $= 10^{-6}s$
1ms (milliseconds) $= 10^{-3}s$

Figure 1.6: Estimated execution time of various algorithms for different input sizes n, assuming a machine capable of 10^6 operations per second.

Complexity / Calculation time	1	$\log_2 n$	n	$n\log_2 n$	n^2	n^3	2^n
1s	∞	∞	10^6	63×10^3	10^3	100	19
1 mn	∞	∞	6×10^7	28×10^5	77×10^2	390	25
1 h	∞	∞	36×10^8	13×10^7	60×10^3	15×10^2	31
1 jour	∞	∞	86×10^9	27×10^8	29×10^4	44×10^2	36

Figure 1.7: Estimated maximum input size n that each algorithm can handle within a fixed execution time, on a machine capable of 10^6 operations per second.

Complexity	1	$\log_2 n$	n	$n\log_2 n$	n^2	n^3	2^n
Evolution of time when the size is multiplied by 10	t	t+3,32	$10\times t$	$(10+\varepsilon)\times t$	$100\times t$	$1000\times t$	t^{10}
Evolution of the size when the time is multiplied by 10	∞	n^{10}	$10\times n$	$(10-\varepsilon)\times n$	$3{,}16\times n$	$2{,}15\times n$	n+3,32

Figure 1.8: Relationship between input size and execution time across different algorithms.

- When $2 \leq k \leq 3$, they are only suitable for medium-sized problems, and when k exceeds 3, they can handle only small problems.
- Algorithms with exponential time complexity, like $\Theta(2^n)$ for instance, are almost unusable, except for very small-sized problems. Such algorithms have been labeled as inefficient.

The table in Figure 1.8 (Lebbah, 2021) below shows how data size and execution time vary in relation to each other.

If the computer's processing speed is increased by a factor of 10. The maximum data size that can be processed by an exponential algorithm is hardly affected. While the data size that can be processed by a linear algorithm is clearly multiplied by 10.

1.9 Optimality of an Algorithm

Suppose we have an algorithm A to solve a given problem. It is natural to wonder if we can find an even better algorithm than A. Hence, it is interesting to know the complexity of the best possible algorithm for solving a problem. Consider a problem P, and let's look at the class C of all algorithms solving P. We also have a complexity measure, which is the number of fundamental operations (evaluated either in the worst case or on average).

We define the optimal complexity of class C as the lower bound of complexities among the algorithms in the class (Aho & Hopcroft, 1974; Cormen et al., 2022). An algorithm A in class C is called optimal if its complexity is equal to the optimal complexity of C, denoted as $M_{opt}(n)$. Therefore, an algorithm A in class C is optimal if there is no algorithm B in C that solves problem P with fewer operations than A.

Sometimes, we cannot precisely determine $M_{opt}(n)$, but we can know the order of magnitude of the optimal complexity of the class. In this case, an algorithm A in class C is considered optimal if its complexity is of lower or equal order of magnitude compared to the complexity of any algorithm in class C:

$$\forall B \in C, M_A = O(M_B)$$

The order of magnitude of $M_{opt}(n)$ is then $\Theta(M_A)$

It is clear that in this situation, multiple algorithms with the same order of complexity can be optimal.

Example. Consider the problem of finding the largest element in a list. An algorithm for solving this problem is formalized in Algorithm 6.

Data: L: a non-empty list with n integer elements
Result: Finds the largest element M in the non-empty list L

```
1 M ← L[1] ;
2 for i ← 2 to n do
3 |   if L[j] > M then
4 |   |   M ← L[j] ;
5 |   end
6 end
```

Algorithm 9: FindMax(L)

It can be observed that the number of comparisons is equal to the number of iterations in the for loop (minus one). Regardless of the evaluation measure considered ($Min_A(n)$, $Avg_A(n)$, or $Max_A(n)$), the complexity remains the same (A refers to FindMax algorithm): $Min_A(n) = Avg_A(n) = Max_A(n) = n - 1$.

Now, we might wonder if this number $n - 1$ can be improved? (Gaudel et al., 1987) In fact, it can be proven that this algorithm is optimal.

1.10 Recurrence Equations and Solving Techniques

When analyzing a recursive algorithm, we typically obtain a complexity function that is itself recursive. It frequently happens that, in order to analyze the complexity of an algorithm, we need to solve a recurrence. The goal of this final part of the chapter is to study methods for solving recurrence equations.

1.10.1 Recurrence Relations for Quick Sort and Merge Sort Algorithms Complexity

QuickSort Algorithm To sort a table using the QuickSort algorithm (see Algorithm 10), we need to (Cormen et al., 2022):

- Perform quick sort on the elements of array A from index lo to index hi.
- Choose a pivot, places it at position hi, and calls the partition function.
- Recursively call itself on the left and right partitions.

```
1 function QuickSort(A,lo,hi):
2 |   if lo < hi then
3 |   |   p ← choose the pivot element ;
4 |   |   swap A[hi] and A[p] ;
5 |   |   i ← Partition(A,lo,hi) ;
6 |   |   QuickSort(A,lo,i − 1) ;
7 |   |   QuickSort(A,i + 1,hi) ;
8 |   end
```

Algorithm 10: The QuickSort Algorithm.

The function PARTITION(A,lo,hi) partitions the elements of array A from index lo to index hi using $A[hi]$ as the pivot. It returns the position of the pivot in the partitioned array.

Average number of comparisons in QuickSort The recurrence equation C_N for the average number of comparisons in QuickSort, where N is the table size, is:

$$\begin{cases} C_0 & = 0 \\ C_1 & = 0 \\ C_N & = N + 1 + \sum_{k=1}^{N} \frac{1}{N}(C_{k-1} + C_{N-k}) \text{ for } N > 1 \end{cases}$$

This recurrence equation is derived from the following detailed form:

$$\begin{cases} C_0 & = 0 \\ C_1 & = 0 \\ C_N & = N + 1 + \frac{C_0+C_{N-1}}{N} + \frac{C_1+C_{N-2}}{N} + ... + \frac{C_{N-1}+C_0}{N} \text{ for } N > 1 \end{cases}$$

Where: The partition requires $N + 1$ comparisons.

- If the pivot is in the first position, the left call will receive an array of 0 elements, and the right call an array of $N - 1$ elements.
- If it's in the second position, the left call will receive an array of 1 element, and the right call an array of $N - 2$ elements.
- ... etc.

It is necessary to consider the probability of each pivot position occurring (the probability of all pivot positions). We assume a uniform distribution ($p(x) = \frac{1}{N}$).

QuickSort Complexity (Cormen et al., 2022)
Best case: $O(n \log n)$ (pivot chosen partitions the array into two equal parts).
Worst case: $O(n^2)$ (pivot partitions the array with one empty side and the other side with $n-1$ elements). Occurs when the smallest or largest element is always chosen as the pivot (e.g., sorted arrays).
Average Complexity: The average number of comparisons C_N to quickly sort an array of N distinct elements is in $O(2n \ln n)$

Number of comparisons in the worst case for the Merge Sort algorithm The recurrence equation $T(N)$ for the worst-case time complexity of Merge Sort, where N is the size of the input array, is:

$$\begin{cases} C_1 & = 0 \\ C_N & = C_{\lceil \frac{N}{2} \rceil} + C_{\lfloor \frac{N}{2} \rfloor} + N \text{ for } N > 1 \end{cases}$$

1.10.2 Recurrence Relation for Computing Fibonacci Numbers

The naive version (first version) of the algorithm that computes Fibonacci numbers is shown in Algorithm 11.

```
1 if n = 0 then
2 |  return 0;
3 else
4 |  if i = 1 then
5 |  |  return 1;
6 |  else
7 |  |  return FIB1(n-1)+FIB1(n-2);
8 |  end
9 end
```

Algorithm 11: FIB1(n)

The number of additions performed by the FIB1 algorithm for an input of size N is given by the following recurrence equation:

$$\begin{cases} F_0 & = 0 \\ F_1 & = 0 \\ F_N & = F_{N-1} + F_{N-2} + 1 \text{ for } N > 1 \end{cases}$$

1.10.3 Recurrence Relation for Computing the Number of Moves in the Tower of Hanoi

Goal: Move the entire tower from the first peg to one of the other two pegs.
Constraints:

- Only one disk can be moved at a time.
- A disk can only be placed on a disk with a larger diameter.

The number of disk movements for a tower of N disks from one peg to another satisfies the following recurrence relation (Knuth, 1997):

$$\begin{cases} T_1 & = 1 \\ T_N & = 2T_{N-1} + 1 \text{ for } N > 1 \end{cases}$$

1.10.4 Solving Techniques for Recurrence Equations

Once the recurrence equation is established, the next step is to derive its analytical or asymptotic solution in order to determine the algorithm's complexity. Several approaches can be used for this purpose:

- Guess-and-verify.
- Plug-and-chug (telescoping).
- Recursion trees.
- Characteristic equations (linear).
- Master theorem (general theorem).
- Change of variable.

Guess-and-Verify Method

The principle applied in this method is to (Cormen et al., 2022; Knuth, 1997):

1. Calculate the initial values of T_n;
2. Guess an analytical solution;
3. Prove its correctness, for example, by induction

Example application
$T_n = 2T_{n-1} + 1$, which represents the recurrence relation for the Tower of Hanoi problem in terms of the number of moves. To solve T_n, First, we calculate the initial values of T_n:

n	1	2	3	4	5	6	...
T_n	1	3	7	15	31	63	...

Second, we guess $T_n = 2^n - 1$.
Third, We must prove (by induction) that the solution is correct:

Théorème. *$T_n = 2^n - 1$ satisfies the recurrence:*

$$\begin{cases} T_1 & = 1 \\ T_N & = 2T_{N-1} + 1 \text{ for } N > 1 \end{cases}$$

Proof. By induction on n. Let $P(n) =$ "$T_n = 2^n - 1$"
Base case: $P(1)$ is true since $T_1 = 1 = 2^1 - 1$
Inductive case: We will show that $T_n = 2^n - 1$ (for $n \geq 2$) is true whenever $T_{n-1} = 2^{n-1} - 1$ is true:

$$\begin{aligned} T_n =& 2T_{n-1} + 1 \\ =& 2(2^{n-1} - 1) + 1 \\ =& 2^n - 1 \end{aligned}$$

□

"Plug-and-Chug" Method (Brute Force)

The plug-and-chug method (also known as telescoping or method of summation factors) is a simple technique for solving recurrence equations by repeatedly expanding the recurrence to find a pattern (Cormen et al., 2022; Knuth, 1997).

We explain the application of this method for solving the recurrence equation for the Tower of Hanoi complexity:

1. "Plug" (apply the recurrence equation) and "Chug" (simplify).

$$\begin{aligned} T_n =& 1 + 2T_{n-1} \\ =& 1 + 2(1 + 2T_{n-2}) \\ =& 1 + 2 + 4T_{n-2} \\ =& 1 + 2 + 4(1 + 2T_{n-3}) \\ =& 1 + 2 + 4 + 8T_{n-3} \\ =& ... \end{aligned}$$

Note: Simplify moderately.

2. Identify and Verify a "Pattern".

   - Identification:
   $$T_n = 1 + 2 + 4 + ... + 2^{i-1} + 2^i T_{n-i}$$
   - Verification by expanding one more step:
   $$\begin{aligned} T_n =& 1 + 2 + 4 + ... + 2^{i-1} + 2^i(1 + 2T_{n-(i+1)}) \\ =& 1 + 2 + 4 + ... + 2^{i-1} + 2^i + 2^{i+1}T_{n-(i+1)} \end{aligned}$$

3. Express the n-th term in terms of previous terms.
   Note that the base case, $T_1 = 1$, occurs when $n - i = 1$. That is, $i = n - 1$, which gives:
   $$\begin{aligned} T_n =& 1 + 2 + 4 + ... + 2^{n-2} + 2^{n-1}T_1 \\ =& 1 + 2 + 4 + ... + 2^{n-2} + 2^{n-1} \end{aligned}$$

4. Find an analytical solution for the n-th term.
   $$\begin{aligned} T_n =& 1 + 2 + 4 + ... + 2^{n-2} + 2^{n-1} \\ =& \sum_{i=0}^{n-1} 2^i \\ =& \frac{1 - 2^n}{1 - 2} \\ =& 2^n - 1 \end{aligned}$$

"Plug-and-Chug" Method Applied to a Special Case of First-Order Linear Recurrence

Recurrences are classified based on the combination of terms, the nature of coefficients, as well as the number and nature of the preceding terms involved.
Examples
First-order linear: $a_n = na_{n-1} - 1$
First-order nonlinear: $a_n = 1/(1 + a_{n-1})$

Second-order linear: $a_n = a_{n-1} + 2a_{n-2}$
Second-order nonlinear: $a_n = a_{n-1}2a_{n-2} + \sqrt{a_n - 2}$
Second-order with variable coefficients: $a_n = na_{n-1} + (n - 1)a_{n-2} + 1$

Non-linear recurrences involve non-linear operations like multiplication, division, or other functions that are not simple sums or differences of previous terms.

The plug-and-chug applied to a recurrence of the form (Geurts, 2013/2014):

$$T_n = T_{n-1} + f(n)$$

directly yields:

$$T_n = T_0 + \sum_{k=1}^{n} f(k) \text{ or } T_n = T_1 + \sum_{k=2}^{n} f(k)$$

If the coefficient of T_{n-1} is not 1:

$$T_n = g(n)T_{n-1} + f(n)$$

dividing both sides by $h(n) = g(n).g(n-1).g(n-2)...g(1)$ reduces it to a recurrence of the form:

$$\frac{T_n}{h(n)} = \frac{T_{n-1}}{h(n-1)} + \frac{f(n)}{h(n)} \Rightarrow T_n = h(n)\left(\frac{T_0}{h(0)} + \sum_{k=1}^{n} \frac{f(k)}{h(k)}\right)$$

Useful if the product $h(n)$ has a simple form.

Exercise 6 Solve the recurrence relation $T_0 = 0$, and $T_n = 2T_{n-1} + 2^n$ for $n > 0$, using the "Plug-and-Chug" method.

Solution By dividing both sides by $h(n) = 2 \cdot 2 \cdot ... \cdot 2 = 2^n$, we obtain:

$$\frac{T_n}{2^n} = \frac{T_{n-1}}{2^{n-1}} + 1 \Rightarrow T_n = 2^n(0 + \sum_{k=1}^{n} 1) = n \times 2^n$$

Exercise 7 Solve the recurrence relation $T_0 = 0$ and $T_n = \left(1 + \frac{1}{n}\right) T_{n-1} + 2$ for $n > 0$, using the "Plug-and-Chug" method. This recurrence relation represents the average-case complexity of the Quicksort algorithm.

The computations below illustrate the essence of this formula.
Average Complexity C_N of QuickSort for sorting a table with size N:
$C_N = N + 1 + \sum_{k=1}^{N} \frac{1}{N}(C_{k-1} + C_{N-k})$ for $N > 1$
We bring this formula into the form: $C_N = g(n)C_{N-1} + f(n)$

$$\begin{aligned} C_N &= N + 1 + \tfrac{2}{N}(C_0 + C_1 + ... + C_{N-1}) \\ N \times C_N &= N^2 + N + 2(C_0 + C_1 + ... + C_{N-1}) \\ (N-1) \times C_{N-1} &= (N-1)^2 + (N-1) + 2(C_0 + C_1 + ... + C_{N-2}) \end{aligned}$$

$$N \times C_N - (N-1) \times C_{N-1} = 2N + 2C_{N-1}$$

$$\begin{aligned} N \times C_N &= 2N + 2C_{N-1} + (N-1) \times C_{N-1} \\ &= 2N + C_{N-1} + N \times C_{N-1} \\ C_N &= 2 + \tfrac{1}{N}C_{N-1} + C_{N-1} \\ &= (1 + \tfrac{1}{N})C_{N-1} + 2 \end{aligned}$$

Solution By dividing both sides by:

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

we obtain:

$$\frac{T_n}{n+1} = \frac{T_{n-1}}{n} + \frac{2}{n+1} = \sum_{k=1}^{n} \frac{2}{k+1} \Rightarrow T_n \approx (n+1)2[\ln(n+1) + \gamma - 1]$$

$$\Rightarrow T_n = O(n \log n)$$

Where $\gamma \approx 0.577$ is the Euler's constant. The Euler-Mascheroni constant γ is defined as follows:
$\gamma = \lim_{n \to \infty} \left(1 + \frac{1}{2} + \frac{1}{3} + \frac{1}{4} + ... + \frac{1}{n} - \ln n\right)$

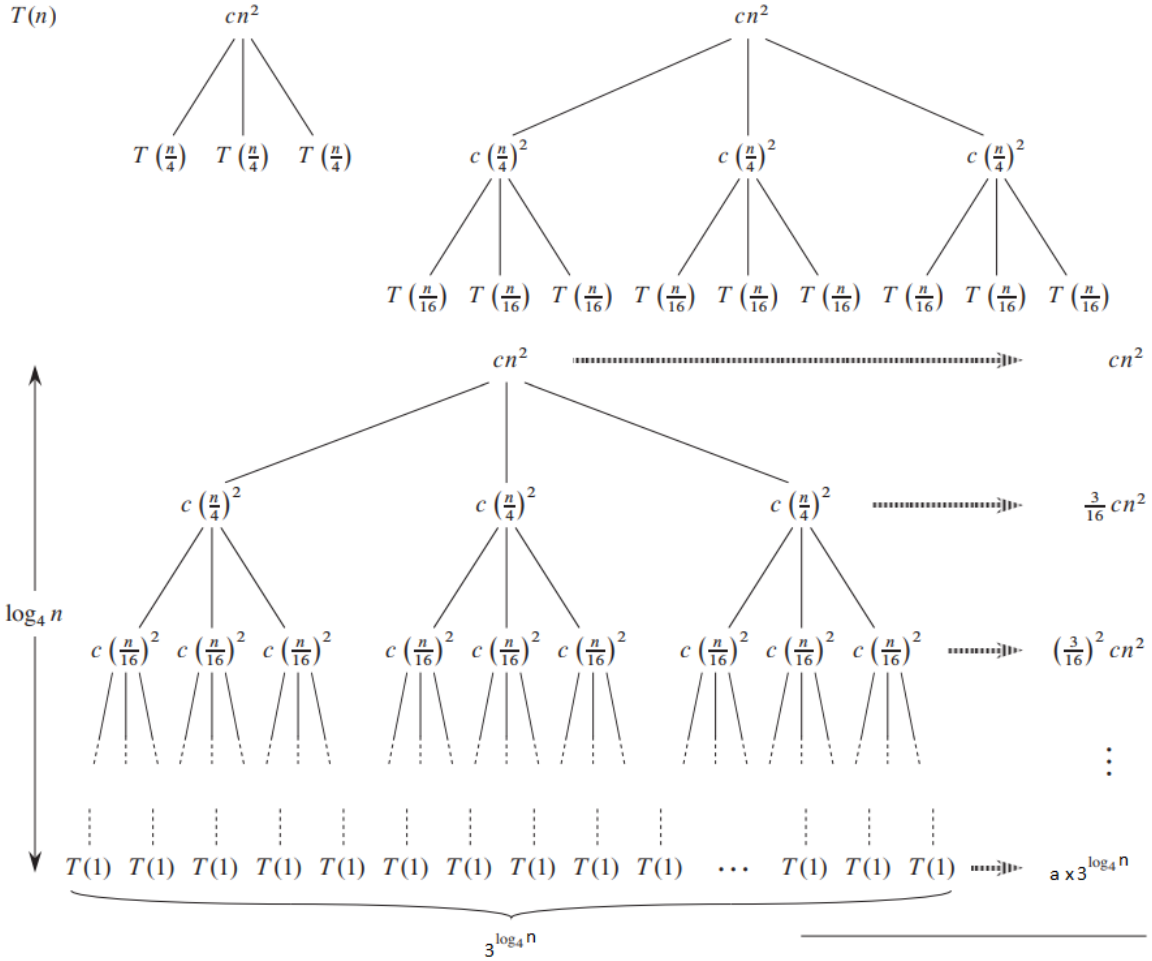
"Recursion Trees" Method

It is a graphical approach to guess an analytical solution (or an asymptotic bound) for a recurrence. The solution must always be subsequently proven (typically by induction) (Cormen et al., 2022).

We illustrate the application of this method with the following recurrence:

$$\begin{cases} T(1) = & a \\ T(n) = & 3T(\frac{n}{4}) + cn^2 \text{ for } n > 1 \end{cases}$$

The recurusion trees obtained for $T(n)$ are:



The total cost is the sum of the cost at each level of the tree:

$$\begin{aligned} T(n) =& cn^2 + \frac{3}{16}cn^2 + ... + \left(\frac{3}{16}\right)^{\log_4 n-1} cn^2 + a3^{\log_4 n} \\ =& \sum_{i=0}^{\log_4 n-1} \left(\frac{3}{16}\right)^i cn^2 + a3^{\log_4 n} \\ <& \sum_{i=0}^{\infty} \left(\frac{3}{16}\right)^i cn^2 + a3^{\log_4 n} \\ \approx& \frac{1}{1-\left(\frac{3}{16}\right)} cn^2 + a3^{\log_4 n} \\ =& O(n^2) \end{aligned}$$

Summary about the three previous methods

These methods ("Guess-and-verify", "Plug-and-Chug", "Recursion Trees") are empirical approaches to find an analytical solution. Empirical means that they are based solely on experience, observation, not on theory or reasoning. The interest in these approaches is general, but:

- It's not always easy to find the pattern
- The obtained sum needs to be solved
- The solution must be verified by induction

We will explore more systematic methods to solve specific recurrences:

- Linear recurrences of order $k \geq 1$ with constant coefficients (exact analytical solution)
- "Divide-and-conquer" recurrences (asymptotic bounds only)

Method to solve linear recurrences of order $k \geq 1$ with constant coefficients

Definition A uniform linear recurrence of degree k with constant coefficients is a recurrence of the form:

$$f(n) = a_1 f(n-1) + a_2 f(n-2) + ... + a_k f(n-k)$$

where $a_1, a_2, ..., a_k \in \mathbb{R}$ are constants, $a_k \neq 0$, and with k initial conditions:

$$f(0) = v_0, f(1) = v_1, ..., f(k-1) = v_{k-1}$$

Example The following recurrence $f(n) = 8f(n-2) - 16f(n-4)$ is a uniform linear recurrence with constant coefficients.

Definition A (general) non-uniform linear recurrence of degree k with constant coefficients is a recurrence of the form:

$$f(n) = a_1 f(n-1) + a_2 f(n-2) + ... + a_k f(n-k) + g(n)$$

with k initial conditions:

$$f(0) = v_0, f(1) = v_1, ..., f(k-1) = v_{k-1}$$

where $a_1, a_2, ..., a_k \in \mathbb{R}$ are constants, $a_k \neq 0$, and $g(n) \neq 0$ is a function depending only on n. The associated uniform recurrence relation (AURR) is:

$$f(n) = a_1 f(n-1) + a_2 f(n-2) + ... + a_k f(n-k)$$

Example The non-uniform linear recurrence relation $f(n) = 3f(n-1) + 2n$ has an associated AURR $f(n) = 3f(n-1)$.

Solving method steps

Consider a recurrence of the form:

$$f(n) = a_1 f(n-1) + a_2 f(n-2) + ... + a_k f(n-k) + g(n)$$

with k initial conditions: $f(0) = v_0, f(1) = v_1, ..., f(k-1) = v_{k-1}$

Step 1: Find the roots of the characteristic equation of the associated uniform recurrence relation (AURR) (Geurts, 2013/2014). AURR refers to Associated uniform Recurrence Relation.

$$r^k - a_1 r^{k-1} - a_2 r^{k-2} - ... - a_{k-1} r - a_k = 0$$

Note: The term $g(n)$ is not included in the characteristic equation

- Characteristic root: Solution r_0 of the characteristic equation.
- Multiplicity m of a root r_0:

  $$r^k - a_1 r^{k-1} - a_2 r^{k-2} - ... - a_{k-1} r - a_k = (r - r_0)^m Q(r)$$

  where r_0 is not a root of $Q(r)$.

Step 2: Find a uniform solution $u(n)$, disregarding initial conditions (Geurts, 2013/2014).

- If the characteristic equation has t distinct roots $r_1, r_2, ..., r_t$ with multiplicities $m_1, ..., m_t$ respectively ($m_i \geq 1$ and $m_1 + m_2 + \ + m_t = k$), then the solution $\{u(n)\}$ of the recurrence relation is of the form:

  $$u(n) = p_1(n) r_1^n + p_2(n) r_2^n + ... + p_t(n) r_t^n$$

  where $p_i(n)$ is a polynomial of degree $m_i - 1$ in the variable n

Example: If the characteristic equation of a uniform linear recurrence has roots:

r_1 with multiplicity 1, r_2 with multiplicity 3, r_3 with multiplicity 2

then the solution is of the form:

$$u(n) = \alpha_1 r_1^n + (\alpha_2 + \alpha_3 n + \alpha_4 n^2) r_2^n + (\alpha_5 + \alpha_6 n) r_3^n$$

Step 3: Find a particular solution $p(n)$, disregarding initial conditions (Geurts, 2013/2014). In the case where $g(n) = Q(n)s^n$, with $Q(n)$ being a polynomial of degree t and $s \in \mathbb{R}$, then

1. if s is not a root of the characteristic equation of the AURR, then
   $$p(n) = (\beta_0 + \beta_1 n + ... + \beta_{t-1} n^{t-1} + \beta_t n^t) s^n$$
2. otherwise, if s is a root of the characteristic equation with multiplicity m of the AURR, then
   $$p(n) = n^m (\beta_0 + \beta_1 n + ... + \beta_{t-1} n^{t-1} + \beta_t n^t) s^n$$

Example: What is the form of the particular solution of the recurrence relation $f(n) = 5f(n-1) - 6f(n-2) + g(n)$ if:

- $g(n) = 7^n$?

- $g(n) = 2^n$?
- $g(n) = n^2(-1)^n$?
- $g(n) = (n-1)3^n$?

Solution: The AURR is $f(n) = 5f(n-1) - 6f(n-2)$, the characteristic equation is $r^2 - 5r + 6 = 0$. Therefore, the characteristic roots are $r_1 = 2$ and $r_2 = 3$, both with a multiplicity of 1.

1. $p(n) = \beta_0 7^n$ ($Q(n) = 1$; $t = 0$; $s = 7$ is not a root)
2. $p(n) = n\beta_0 2^n$ ($Q(n) = 1$; $t = 0$; $s = 2$ is a root with multiplicity 1)
3. $p(n) = (\beta_0 + \beta_1 n + \beta_2 n^2)(-1)^n$ ($Q(n) = n^2$; $t = 2$; $s = -1$ is not a root)
4. $p(n) = n(\beta_0 + \beta_1 n)3^n$ ($Q(n) = n - 1$; $t = 1$; $s = 3$ is a root with multiplicity 1)

When solving a non-uniform linear recurrence relation, the values of the coefficients in the particular solution (the β_i in the example above) do not depend on the initial conditions. Their values can be directly determined from the recurrence relation. We then use the fact that the particular solution is a solution of the initial recurrence relation.
Example: In example 1), the particular solution is of the form $p(n) = \beta_0 7^n$. Thus, we know that $\beta_0 7^n$ is a solution of the recurrence relation $f(n) = 5f(n-1) - 6f(n-2) + 7^n$. By assigning a value to n, we obtain an equation in terms of β_0. For instance, we can choose $n = 2$:

$$\begin{aligned} p(n) =& 5p(n-1) - 6p(n-2) + 7^n \\ p(2) =& 5p(1) - 6p(0) + 7^2 \\ 49\beta_0 =& 5(7\beta_0) - 6(\beta_0) + 49 \end{aligned}$$

Thus, we find $\beta_0 = \frac{49}{20}$. Consequently, we have $p(n) = \frac{7^{n+2}}{20}$.
In this example, the particular solution is expressed in terms of a single unknown. Thus, only one equation was needed to determine its value. In general, the particular solution is expressed in terms of multiple unknowns; in that case, as many equations as unknowns are required. To generate more equations, it's enough to substitute different integer values for n in the initial recurrence relation.
Step 4: Form a general solution, disregarding initial conditions (Geurts, 2013/2014).
To do that, just add the uniform solution and the particular solution: Any solution $\{s(n)\}$ of the recurrence relation:

$$f(n) = a_1 f(n-1) + a_2 f(n-2) + ... + a_k f(n-k) + g(n)$$

is of the form:

$$s(n) = p(n) + u(n)$$

where $\{p(n)\}$ is a particular solution and $\{u(n)\}$ is a solution of the AURR.
Step 5: Determine the values of the constants introduced in Step 2 (Geurts, 2013/2014).

- For each initial condition, apply the general solution to that condition. This yields an equation in terms of the constants to be determined.
- Solve the system of equations formed by these equations

Exercise 8 Find the solution of $f(n) = 4f(n-1)+3n$, with the initial condition $f(1) = 2$.

Solution

Step 1: Find the roots of the characteristic equation of the associated uniform recurrence relation (AURR)
We identify the **AURR**: $f(n) = 4f(n-1)$.
We determine the **characteristic equation of the AURR**: $r - 4 = 0$.
We calculate **the roots of the characteristic polynomial of the AURR**: $r_1 = 4$.
Step 2: Find a uniform solution $u(n)$, without considering initial conditions
We find **the form of the solution of the AURR**: $u(n) = \alpha_1 4^n$.
Step 3: Find a particular solution $p(n)$, without considering initial conditions
Since $g(n) = 3n = 3n(1)^n$, we have $s = 1$ (which is not a root) and $t = 1$ (the degree of $3n$). The solution $p(n)$ is therefore of the form:

$$p(n) = \beta_0 + \beta_1 n$$

We determine **the values of the coefficients** β_0 and β_1: since $p(n)$ is a particular solution, it satisfies the recurrence relation, i.e., $p(n) = 4p(n-1) + 3n$. Thus,

$$\begin{aligned} \beta_0 + \beta_1 n =& 4(\beta_0 + \beta_1(n-1)) + 3n \\ =& (4\beta_0 - 4\beta_1) + (4\beta_1 + 3)n \end{aligned}$$

$$\Rightarrow \begin{cases} \beta_0 = 4\beta_0 - 4\beta_1 \\ \beta_1 = 4\beta_1 + 3 \end{cases} \quad \Rightarrow \quad \begin{aligned} \beta_0 &= -\frac{4}{3} \\ \text{et } \beta_1 &= -1 \end{aligned}$$

Step 4: Form a general solution $s(n)$, without considering initial conditions.

$$s(n) = p(n) + u(n) = \alpha_1 4^n - \frac{4}{3} - n$$

Step 5: Determine the values of the constants introduced in step 2.
We determine the value of the coefficient α_1. Since there is only one coefficient to find, we need one equation:

$$s(1) = \alpha_1 \cdot 4 - \frac{4}{3} - 1 = 2 = f(1) \Longrightarrow \alpha_1 = \frac{13}{12}$$

The solution is:

$$\boxed{s(n) = \left(\tfrac{13}{12}\right) 4^n - \tfrac{4}{3} - n}$$

Master Theorem (General Theorem)

This theorem can be used to determine the time complexity of divide and conquer algorithms if the recurrence is in the form specified in the theorem (Knuth, 1997; Cormen et al., 2022).

Théorème. *If the recurrence is of the following form:*

$$T(n) = aT\left(\frac{n}{b}\right) + f(n)$$

where n: the size of the problem, $a \geq 1$, $b > 1$, and $f(n) = \Theta(n^k \log^p n)$ is a function with $k \geq 0$ and p is a real number.
$T(n)$ can then be asymptotically bounded as follows:

1. *If* $a > b^k$, *then* $T(n) = \Theta(n^{\log_b a})$.
2. *If* $a = b^k$, *then:*
   (a) $p > -1 \longrightarrow T(n) = \Theta(n^{\log_b a} \log^{p+1} n)$
   (b) $p = -1 \longrightarrow T(n) = \Theta(n^{\log_b a} \log\log n)$
   (c) $p < -1 \longrightarrow T(n) = \Theta(n^{\log_b a})$
3. *If* $a < b^k$, *then:*
   (a) $p \geq 0 \longrightarrow T(n) = \Theta(n^k \log^p n)$
   (b) $p < 0 \longrightarrow T(n) = \Theta(n^k)$

Exercise 9 Give the solution, using the Master Theorem, for the recurrence relations provided below (Bari, 2018; "Complexité d'un algorithme", 2021/2022):

1. $T(n) = 2T\left(\frac{n}{2}\right) + 1$
2. $T(n) = 4T\left(\frac{n}{2}\right) + n$
3. $T(n) = 8T\left(\frac{n}{2}\right) + n\log n$
4. $T(n) = 2T\left(\frac{n}{2}\right) + n$
5. $T(n) = 4T\left(\frac{n}{2}\right) + n^2\log n$
6. $T(n) = 2T\left(\frac{n}{2}\right) + \frac{n}{\log n}$
7. $T(n) = 2T\left(\frac{n}{2}\right) + \frac{n}{\log^2 n}$
8. $T(n) = T\left(\frac{n}{2}\right) + n^2$
9. $T(n) = 2T\left(\frac{n}{2}\right) + n^2\log n$
10. $T(n) = 4T\left(\frac{n}{2}\right) + \frac{n^3}{\log n}$

Solution

$$T(n) = 2T\left(\frac{n}{2}\right) + 1$$

For this recurrence, we have $a = 2$, $b = 2$ and $f(n) = 1 = \Theta(n^0 \log^0 n)$ ($k = 0$ and $p = 0$)

- $a > b^k (2 > 2^0) \longrightarrow T(n) = \Theta(n^{\log_2 2}) = \Theta(n)$

$$T(n) = 4T\left(\frac{n}{2}\right) + n$$

For this recurrence, we have $a = 4$, $b = 2$ and $f(n) = n = \Theta(n^1 \log^0 n)$ ($k = 1$ and $p = 0$)

- $a > b^k (4 > 2^1) \longrightarrow T(n) = \Theta(n^{\log_2 4}) = \Theta(n^2)$

$$T(n) = 8T\left(\frac{n}{2}\right) + n\log n$$

For this recurrence, we have $a = 8$, $b = 2$ and $f(n) = n\log n = \Theta(n^1 \log^1 n)$ ($k = 1$ and $p = 1$)

- $a > b^k (8 = 2^1) \longrightarrow T(n) = \Theta(n^{\log_2 8}) = \Theta(n^3)$

$$T(n) = 2T\left(\frac{n}{2}\right) + n$$

For this recurrence, we have $a = 2$, $b = 2$ and $f(n) = n = \Theta(n^1 \log^0 n)$ ($k = 1$ and $p = 0$)

- $a = b^k (2 = 2^1) \longrightarrow T(n) = \Theta(n^{\log_2 2} \log n) = \Theta(n \log n)$ ($p > -1$)

$$T(n) = 4T\left(\frac{n}{2}\right) + n^2 \log n$$

For this recurrence, we have $a = 4$, $b = 2$ and $f(n) = n^2 \log n = \Theta(n^2 \log^1 n)$ ($k = 2$ and $p = 1$)

- $a = b^k (4 = 2^2) \longrightarrow T(n) = \Theta(n^{\log_2 4} \log^2 n) = \Theta(n^2 \log^2 n)$ ($p > -1$)

$$T(n) = 2T\left(\frac{n}{2}\right) + \frac{n}{\log n}$$

For this recurrence, we have $a = 2$, $b = 2$ and $f(n) = \frac{n}{\log n} = \Theta(n^1 \log^{-1} n)$ ($k = 1$ and $p = -1$)

- $a = b^k (2 = 2^2) \longrightarrow T(n) = \Theta(n^{\log_2 2} \log \log n) = \Theta(n \log \log n)$ ($p = -1$)

$$T(n) = 2T\left(\frac{n}{2}\right) + \frac{n}{\log^2 n}$$

For this recurrence, we have $a = 2$, $b = 2$ and $f(n) = \frac{n}{\log^2 n} = \Theta(n^1 \log^{-2} n)$ ($k = 1$ and $p = -2$)

- $a = b^k (2 = 2^2) \longrightarrow T(n) = \Theta(n^{\log_2 2}) = \Theta(n)$ ($p < -1$)

$$T(n) = T\left(\frac{n}{2}\right) + n^2$$

For this recurrence, we have $a = 1$, $b = 2$ and $f(n) = n^2 = \Theta(n^2 \log^0 n)$ ($k = 2$ and $p = 0$)

- $a < b^k (1 < 2^2) \longrightarrow T(n) = \Theta(n^2 \log^0 n) = \Theta(n^2)$ ($p \geq 0$)

$$T(n) = 2T\left(\frac{n}{2}\right) + n^2 \log n$$

For this recurrence, we have $a = 2$, $b = 2$ and $f(n) = n^2 \log n = \Theta(n^2 \log^1 n)$ ($k = 2$ and $p = 1$)

- $a < b^k (2 < 2^2) \longrightarrow T(n) = \Theta(n^2 \log^1 n) = \Theta(n^2 \log n)$ ($p \geq 0$)

$$T(n) = 4T\left(\frac{n}{2}\right) + \frac{n^3}{\log n}$$

For this recurrence, we have $a = 4$, $b = 2$ and $f(n) = \frac{n^3}{\log n} = \Theta(n^3 \log^{-1} n)$ ($k = 3$ and $p = -1$)

- $a < b^k (4 < 2^3) \longrightarrow T(n) = \Theta(n^3)$ ($p < 0$)

Conclusion

This chapter introduced the core concepts of algorithmic complexity, starting with the definition of an algorithm and progressing through time and space analysis using examples like Fibonacci computation and Insertion Sort.

We covered key tools such as asymptotic analysis, Landau notations, and recurrence relations, along with techniques for analyzing recursive and divide-and-conquer algorithms.

The chapter also discussed how data size affects execution time and the importance of identifying optimal solutions. These foundations prepare students to evaluate algorithm performance and address more advanced topics in problem complexity and AI, which follow in the next chapters.

Chapter 2

Problem Classification

Introduction

Some combinatorial optimization problems have polynomial algorithms, while others still do not have any[1].

But, is there truly a class of combinatorial problems for which polynomial algorithms will never be found, or do difficult problems indeed have such algorithms that have not yet been discovered?

For a long time, there has been a conjecture about the existence of a class of inherently difficult problems. Because several difficult problems (such as the traveling salesman problem) have resisted research efforts for over 50 years: despite these efforts, no polynomial algorithm has been discovered.

In this chapter, we will delve into the classification of problems based on the complexity of the best algorithm that solves them.

2.1 Computational Problems and Combinatorial Optimization

A **combinatorial problem** (Papadimitriou & Steiglitz, 1998) consists of finding, from a finite collection of objects and under a set of constraints, an object that satisfies all the constraints—and possibly optimizes some objective function. These problems typically involve a vast number of combinations to explore, making them computationally challenging.

A **combinatorial optimization problem** (Papadimitriou & Steiglitz, 1998) is a special type of combinatorial problem where the goal is to find the optimal solution—usually the *maximum* or *minimum* of a given objective function—under certain constraints. An example is the *Traveling Salesman Problem*: given a list of cities and distances between each pair of cities, the task is to find the shortest route that visits each city exactly once.

Optimization problems can be further categorized based on the nature of their variables:

- If the variables are **discrete**, the problem is a **combinatorial optimization problem**.
- If the variables are **continuous**, the problem is known as a **continuous optimization problem**.

[1]The complexity of a problem represents the complexity of the best program to solve it.

For every combinatorial optimization problem, there is an associated **decision problem** that asks whether a solution exists within certain bounds or constraints. For instance, instead of finding the shortest route, one might ask: *"Is there a route shorter than 1000 km?"*

A **decision problem** is a mathematical question with a "yes" or "no" answer. Many theoretical questions in computer science are framed this way.

A **computing solution problem** goes a step further. Instead of just asking whether a solution exists, it seeks to *construct* one. For example:

- If a Boolean formula is satisfiable, compute a satisfying assignment.
- If a graph is 3-colorable, compute a valid 3-coloring.

Notably, for each **computing solution problem**, there is a corresponding **decision problem** asking whether a solution exists.

2.2 From Decision Problems to Solution and Optimization

Complexity theory was developed in the 1970s to address the question indicated in the introduction. The main result is that all difficult problems are interconnected. The discovery of a polynomial algorithm for a single difficult problem would lead to deducing polynomial algorithms for all others. Complexity theory deals only with existence problems[2], with yes/no answers. A decision problem is a mathematical question to which the answer is either "yes" or "no". This is not problematic for problems involving computing solutions or optimization (Cormen et al., 2022).

- An efficient algorithm for an existence problem can be used to efficiently solve its computing solution problem.
- Similarly, an efficient algorithm for an existence problem can be used to efficiently solve its associated optimization problem through a simple dichotomy on the objective function values.

For example, to transform "finding the smallest non-trivial divisor of N", we would add an input k to get the question "is there a non-trivial divisor of N that is less than k?". If we can answer this question, then we can quickly find the smallest non-trivial divisor of N: we just need to perform a binary search by testing if there is a divisor $\leq N/2$, then if yes, if there is a divisor $\leq N/4$, or if not, if there is a divisor $\leq 3N/4$, and so on. Solving the problem of finding the smallest non-trivial divisor of 25. The decision corresponding problem instance is : Is there a non-trivial divisor of 25 that is less than k ?

Let's consider the SAT problem and assume it has a polynomial algorithm A to solve it. Then, we can construct the following algorithm (see Algorithm 12), which calculates a satisfying valuation σ (if one exists) for a given formula φ:

[2]These are also called decision problems

Data: A formula $\varphi(x_1, ..., x_n)$
Result: A satisfying valuation for φ (if one exists)
if $A(\varphi)$ returns "No" **then**
  return "Unsatisfiable"
end
for $i \leftarrow 1$ **to** n **do**
  if $A(\varphi(b_1, ..., b_{i-1}, 1, x_{i+1}, ..., x_n))$ returns "Yes" **then**
    $b_i \leftarrow 1$;
  else
    $b_i \leftarrow 0$;
  end
end
return $(b_1, ..., b_n)$;

Algorithm 12: SolSAT(φ)

2.3 Easy Problems and Hard Problems

A problem consists of two elements: an input (or instance) and a question or task to perform. Here are two examples that can be reformulated as follows:
Primality:

Data: An integer N.

Question: Is N prime?

Sorting:

Data: A list of integers l.

Task: Sort l in ascending order.

2.3.1 Easy Problems

Easy means that polynomial algorithms are known to solve these problems, i.e., the complexity of these algorithms is of the form $O(n^k)$ with $k \in \mathbb{N}$. An algorithm with $O(n^{100})$ complexity is not efficient (Lebbah, 2021). However, the discovered polynomial algorithms rarely go beyond degree 6.

Below, we list a set of easy problems:
Graph Exploration:

Data: A directed graph $G = (V, E)$, two vertices s and t in V.

Question: Is there a path from s to t?

Polynomial algorithms: Breadth-First Search (BFS) and Depth-First Search (DFS) algorithms with $O(|V| + |E|)$ complexity
Shortest Path:

Data: $G = (V, E, \nu)$ a valued directed graph with an application[3] $\nu : E \to \mathbb{R}$, two vertices s and t in V.

[3] A function $f : E \to F$ is an application if $Dom(f) = E$.

Task: Find a shortest path from s to t.

Polynomial algorithms: Bellman's algorithm with $O(|V||E|)$ complexity. Dijkstra's algorithm with $O(|E| + |V| \log |V|)$ complexity
Max Flow:

Data: $G = (V, E, \nu)$ a valued directed graph with a capacity function $\nu : V \times V \to \mathbb{N}$ such that $\nu(x, y) = 0$ if $(x, y) \notin E$, two vertices s and t in V.

Task: Maximize the flow rate that can go through the network between s and t.

Polynomial algorithms: Ford-Fulkerson algorithm with $O(|V||E|^2)$ complexity.
[4] [5] [6]

Minimum Weight Spanning Tree:

Data: $G = (V, E, \nu)$ a valued simple[7] undirected graph with an application $\nu : E \to \mathbb{R}$

Task: Find a minimum weight spanning tree.

Polynomial algorithms: Prim's algorithm with $O(|V|^2)$ complexity. Kruskal's algorithm with $O(|E| \log |E|)$ complexity. If G is directed, an arborescence could be computed in $O(|V||E|)$ using Edmonds' algorithm
Matching:

Data: A simple undirected graph $G = (V, E)$.

Task: Find a maximum cardinality matching[8].

Polynomial algorithm: Edmonds' algorithm with $O(|V|^4)$ complexity.
Eulerian Cycles:

Data: An undirected graph $G = (V, E)$.

Question: Does an Eulerian cycle[9] exist in G?

The existence problem for Eulerian Cycles is solvable in $O(|V|^2)$.
Planarity Testing:

Data: A simple graph $G = (V, E)$.

Question: Is G planar, i.e., can it be drawn in the plane without edge crossings?

Algorithm with $O(|E|)$ complexity by Hopcroft and Tarjan
Bipartiteness Testing:

Data: A simple graph $G = (V, E)$.

[4] A "chain" in a graph is a succession of edges that make it possible to link two vertices in the graph.
[5] A chain that touches each vertex one time at most is called "elementary."
[6] A chain that uses each edge one time at most is called "simple."
[7] A simple graph is a graph without loops or multiple edges. There are only edges between distinct vertices, and between two vertices there is at most one edge.
[8] A matching of G is a subset of edges such that no two of them have a common vertex.
[9] An Eulerian cycle is a path that visits each edge of the graph exactly once. A Chinese postman tour visits each edge at least once.

Question: Is G bipartite[10]?

It can be proven that a graph is bipartite if and only if it contains no odd-length cycle[11]. This can be verified using an algorithm with $O(|E|)$ time complexity.
Search in n elements:
This is the search for an element in an array of n elements.
Sorting N numbers:
This problem is solvable using heap sort algorithm in $O(n \log n)$.

2.3.2 Hard Problems

Below, we list a set of hard problems (Lebbah, 2021):
Maximum Stable Set[12]:

Data: A simple graph $G = (V, E)$.

Task: Find a maximum cardinality stable set, i.e., a largest possible subset of graph vertices where no two elements are adjacent.

Minimum Vertex Cover[13]:

Data: A simple graph $G = (V, E)$.

Task: Find a minimum set of vertices to cover all edges.

Maximum Clique[14]:

Data: A simple graph $G = (V, E)$.

Task: Find a clique (a subset of vertices of graph G that are all pairwise adjacent, also known as a complete subgraph) with the maximum number of vertices.

Minimum Coloring:

Data: A simple graph $G = (V, E)$.

Task: Determine the chromatic number[15] of G.

Hamiltonian Problem:

Data: A simple graph $G = (V, E)$.
A Hamiltonian cycle is a Hamiltonian path that forms a cycle[16]. A Hamiltonian path in an undirected or directed graph is a path that passes through all vertices exactly once.

Question: Does G have a Hamiltonian cycle?

[10] A graph is bipartite if its vertex set can be partitioned into two disjoint subsets U and U' such that each edge has one endpoint in U and the other in U'.

[11] A cycle containing an odd (respectively even) number of edges is called an odd (respectively even) cycle.

[15] G is k-colorable if its vertices can be colored with k distinct colors, such that no two adjacent vertices have the same color. The smallest value of k for which G is k-colorable is the chromatic number of G.

[16] A cycle is a sequence of consecutive edges, a simple path (not containing the same edge multiple times), with identical endpoints.

Traveling Salesman Problem (TSP):

Data: $G = (V, E, \nu)$ a complete directed weighted graph with an application $\nu : E \to \mathbb{R}$.

Task: Find a Hamiltonian cycle, with minimal cost.

Boolean Satisfiability Problem (SAT):

Data: A propositional formula.

Question: Determine whether there exists an assignment of propositional variables that makes all clauses true.

Integer Knapsack Problem[17]:

Data: A set of items, each with a weight and a value.

Task: Determine the quantity of each item to include in a collection so that the total weight is less than or equal to a given limit and the total value is maximized.

Knapsack Problem Variants:

0-1 Knapsack Problem:

$$\begin{cases} \text{Maximize } \sum_{i=1}^{n} v_i x_i \\ \sum_{i=1}^{n} w_i x_i \leq W \\ x_i \in \{0, 1\} \end{cases}$$

Bounded Knapsack Problem:

$$\begin{cases} \text{Maximize } \sum_{i=1}^{n} v_i x_i \\ \sum_{i=1}^{n} w_i x_i \leq W \\ x_i \in \{0, 1, 2, \ldots, c\} \end{cases}$$

Unbounded Knapsack Problem:

$$\begin{cases} \text{Maximize } \sum_{i=1}^{n} v_i x_i \\ \sum_{i=1}^{n} w_i x_i \leq W \\ x_i \in \mathbb{N} \end{cases}$$

x_i represents the number of instances of item i to include in the knapsack.

Bin Packing:

Data: n items with weights a_i and an unlimited number of bins with capacity b

Task: Distribute the items into a minimal number of bins.

Examples (Bin Packing Problem):

Input: Weights $= \{4, 8, 1, 4, 2, 1\}$
Bin capacity $b = 10$.

Output: 2.
Solution: We need at least 2 bins to accommodate all the items. The first bin contains $\{4, 4, 2\}$ and the second bin $\{8, 1, 1\}$.

Input: Weights $= \{9, 8, 2, 2, 5, 4\}$
Bin capacity $b = 10$.

Output: 4.
Solution: We need at least 4 bins to accommodate all the items.

2.4 Undecidable Problems

An undecidable problem is a decision problem for which it has been proven impossible to construct an algorithm that always leads to a correct answer of yes or no (Kozen, 2006). Below, we give two undecidable problems:
Halting Problem:

Data: A computer program and input data for that program.

Question: Determine whether the program will eventually stop or continue running forever.

Post's Correspondence Problem (1946):

Data: n pairs of words $(u_1, v_1), ..., (u_n, v_n)$ over an alphabet Σ.

Question: Is there a finite sequence $i_1, ..., i_k$ $(k > 0)$ such that $u_{i_1}...u_{i_k} = v_{i_1}...v_{i_k}$.
Note: The index sequences are the same.

The pairs (u_i, v_i) can be seen as dominos.
Example:

a	aa	ba	bab
ab	a	aa	abba

A solution: a.bab.ba.aa.aa = ab.abba.aa.a.a
Index sequence: 1, 4, 3, 2, 2.

2.5 Classes P et NP

Given a function $f : \mathbb{N} \longrightarrow \mathbb{N}$, a problem is said to belong to the class $DTIME(f)$ if there exists a deterministic machine[18] that, on any input of length n, solves the problem in $O(f(n))$ computational steps.

The set of decision problems that have polynomial-time algorithms forms the class P (Cormen et al., 2022). We can define $P = \bigcup_{k \geq 0} DTIME(n \longrightarrow n^k)$.

The class NP[19] consists of decision problems for which a proposed solution of "yes" can be verified in polynomial time (Cormen et al., 2022). We also define the class $NP = \bigcup_{k \geq 0} NTIME(n \longrightarrow n^k)$, where N stands for non-deterministic[20].

Problems not in NP exist, but for the most part, they are of theoretical interest only (Lebbah, 2021). NP includes P. Problems in the class NP are quickly verifiable, whereas problems in the class P are quickly solvable.

[18] A deterministic machine is the formal model of a machine as we know them (our computers are deterministic machines).

[19] NP does NOT mean Non-Polynomial; it stands for Non-Deterministic Polynomial!

[20] The non-deterministic computation model is enriched by an "choice" instruction, where there exists an immediate way to lead to the solution. A non-deterministic machine is a purely theoretical variant of deterministic machines: we can't build one. At each step of its computation, this machine can make a non-deterministic choice: it has a choice between several actions, and it takes one. If one of the choices leads to accepting the input, it's considered that it made the right choice. In some sense, it always guesses correctly.

2.5.1 Complexity Class NP

Let A be a problem. An verifier for A is an algorithm V that takes pairs $\langle x, y \rangle$ as input, where x is an instance[21] of A, and verifies that y is indeed a proof (or certificate) showing that x is a positive instance.

Problem A has a polynomial verifier V if:

1. There exists a polynomial $p(.)$ such that for every instance x of A: x is a positive instance[22] if and only if there exists a certificate y of size at most $p(\text{size}(x))$[23] and such that V accepts $\langle x, y \rangle$

2. The algorithm V is polynomial in size(x).

Examples

- A verifier for SAT takes as input a CNF formula and a valuation of its variables, and decides if the valuation makes the formula true.

- A verifier for 3-coloring takes as input a graph and a vertex coloring with $1, 2, 3$, and verifies that adjacent vertices have different colors.

- A verifier for Hamiltonian path takes as input a graph $G = (V, E)$ and a permutation $v_{i_1}, v_{i_2}, ..., v_{i_n}$ of vertices in V, and verifies that it forms a path in G: $(v_{i_j}, v_{i_{j+1}}) \in E$ for all j

How to Prove Membership in NP?

The class NP is the class of problems that have polynomial verifiers (Cormen et al., 2022).

Exercise 10 Prove that the following problem: given a set S of n integers and an integer b, does there exist a subset T of S whose sum of elements equals b? belongs to the class NP.

Solution This problem is in NP because verifying that the sum of a subset T (certificate), which is a subset of a set S (problem instance) of size n, is equal to b can be done in $O(n^2)$. In this verification, it is necessary to ensure that all elements of T are in S, which will require at most n^2 tests.

[21] An instance of problem A is the question posed on a particular input/data of A. Example:

- Problem: Determine if a non-directed graph is connected.
- Instance: $V = 1, 2, 3, 4, 5$, $E = (1, 2), (2, 3), (3, 1), (4, 5)$
- Algorithm: Depth-first search

[22] A positive instance of a decision problem is an instance (data) for which the problem's output is "True" (Yes).

[23] We can always restrict ourselves to certificates of polynomial length in the length of x, since in polynomial time in the length of x, the algorithm V cannot read more than a polynomial number of symbols from the certificate.

2.5.2 NP-Complete Problems

These are the most difficult problems in NP, often referred to as the "hard core". The concept of NP-complete problems is based on that of polynomial transformation (reduction) of one problem to another (Arora & Barak, 2009).

Polynomial Reduction

A problem of existence P_1 is polynomially reducible to another problem P_2 if there exists a polynomial algorithm A that transforms any instance of P_1 into an instance of P_2, while preserving the Yes and No answers.

- We denote this as $P_1 \leq_p P_2$.

Example A stable set in a simple graph G is a clique in the complement graph G_c. [24]

An NP-complete problem[25] is an NP problem to which any other NP problem can be polynomially reduced (Arora & Barak, 2009). The class of NP-complete problems is denoted as NPC.

Implications of NP-Completeness and the SAT Problem

The strong practical consequence of the NPC class:

- If we were to find a polynomial algorithm A for a single NP-complete problem X. We could derive one for any other difficult problem Y in NP.
- It's sufficient to polynomially transform instances of Y into instances for X, and then execute the algorithm for X.

SAT Problem:

Data Given a set of variables $x_1, ..., x_n$ and a logical formula φ in conjunctive normal form $\varphi = c_1 \wedge c_2 \wedge ... \wedge c_l$, where each clause $c_i = (y_{i,1} \vee y_{i,2} \vee ... \vee y_{i,k_i})$, and each $y_{i,j}$ is a literal.

Question Is there a truth value assignment to the variables that makes φ true?

2.5.3 How to Prove a Problem is NP-Complete?

Given a problem P, we consider its associated existence problem:

- If P is an optimization problem[26], we consider the associated existence problem by replacing the search for an optimal solution with a search for a solution with cost at most k, where k is an integer added to the input.

[24] The existence problem of a stable set in a simple graph G corresponds to the existence problem of a clique in the complement graph G_c. The polynomial transformation here involves computing the complement graph G_c.

[25] In the literature, you may encounter the term "NP-hard" for optimization problems: a combinatorial optimization problem is NP-hard if its associated existence problem is NP-complete.

[26] The goal of an optimization problem is to find a solution that maximizes (or minimizes) a given objective function. For each optimization problem, we can associate a decision problem that determines whether there exists a solution for which the objective function is greater (or less) than or equal to a given value. The complexity of an optimization problem is related to that of the associated decision problem. In particular, if the decision problem is NP-complete, then the optimization problem is called NP-hard.

The technique used to prove that an existence problem X in NP is NP-complete involves demonstrating that a known NP-complete problem Y can be polynomially reduced to X (and not the other way around, a common mistake!) (Arora & Barak, 2009).

Reduction Technique To prove the NP-completeness of a problem P, it is sufficient to prove:

1. It has a polynomial verifier (ensuring that $P \in NP$).

2. $P' \leq_p P$ with a known NP-complete problem P' (ensuring $C \leq_p P$ for any problem $C \in NP$).

2.5.4 Establishing NP-Completeness for 3-SAT

Définition. Data: A set of variables $x_1, ..., x_n$ and a formula $\varphi = c_1 \wedge c_2 \wedge ... \wedge c_p$ where $c_i = y_{i,1} \vee y_{i,2} \vee y_{i,3}$ and for all i, j, $y_{i,j}$ is a literal, i.e., $y_{i,j}$ is either x_k or $\neg x_k$ for one of the x_k.

Question: Is there an assignment of truth values to the variables that makes φ true?

The 3-SAT problem is a restriction of the SAT problem to formulas that are in conjunctive normal form with 3 literals.

Théorème. *The 3-SAT problem is NP-complete.*

To prove this theorem, it's necessary to demonstrate:

1. 3-SAT $\in NP$

2. $P' \leq_p$ 3-SAT with a known NP-complete problem P' (we'll take $P' = SAT$)

3-SAT Proven NP-Complete

1. Is 3-SAT in NP?
   A verifier for 3-SAT takes as input a 3-CNF formula and a variable valuation (certificate), and checks if the valuation makes the formula true. This verification can be done in $O(p)$ time (p is the number of clauses). Therefore, the verifier is polynomial. Consequently, 3-SAT is in NP.

2. Is SAT $\leq_p$ 3-SAT?[27]
   For any instance φ of SAT, we will associate an instance $\widetilde{\varphi}$ of 3-SAT such that:

   $$\varphi \text{ is satisfiable } \iff \widetilde{\varphi} \text{ is satisfiable}$$

   Remarque. We will construct $\widetilde{\varphi}$ from φ in polynomial time (polynomial reduction) with respect to the size $|\varphi|$ of φ.

[27] Polynomial reduction ensures that we can use an algorithm P for 3-SAT to solve SAT: for any CNF formula φ, we first construct $\widetilde{\varphi}$ and then run P on $\widetilde{\varphi}$. Let P' be the algorithm constructed in this way. **Important:** If P is a polynomial algorithm for 3-SAT, then P' is also polynomial.

Reduction from SAT to 3-SAT If $\varphi = c_1 \wedge ... \wedge c_k$ where each c_i is a clause, we construct $\widetilde{\varphi} = \varphi_1 \wedge ... \wedge \varphi_k$, where:

- Each φ_i is a conjunction of 3-clauses
- φ_i uses the variables from c_i, + possibly new variables
- If an assignment of variables makes c_i true, we can complete it to make φ_i true
- Conversely, if an assignment of variables makes φ_i true, its restriction to the variables in c_i makes c_i true

Reduction from SAT to 3-SAT: Construction of φ_i

- If $c_i = \ell_1$ (a literal), we add 2 new variables y_i, z_i, and
$$\varphi_i = (\ell_1 \vee y_i \vee z_i) \wedge (\ell_1 \vee \neg y_i \vee z_i) \wedge (\ell_1 \vee y_i \vee \neg z_i) \wedge (\ell_1 \vee \neg y_i \vee \neg z_i)$$
- If $c_i = \ell_1 \vee \ell_2$ (2 literals), we add 1 new variable y_i, and
$$\varphi_i = (y_i \vee \ell_1 \vee \ell_2) \wedge (\neg y_i \vee \ell_1 \vee \ell_2)$$
- If c_i is a 3-clause: $\varphi_i = c_i$
- If $c_i = \ell_1 \vee ... \vee \ell_k$ with $k \geq 4$, we add $k-3$ new variables $t_{i,1}, ..., t_{i,k-3}$ and
$$\varphi_i = (t_{i,1} \vee \ell_1 \vee \ell_2) \wedge (\neg t_{i,1} \vee \ell_3 \vee t_{i,2}) \wedge (\neg t_{i,2} \vee \ell_4 \vee t_{i,3}) \wedge ... \wedge (\neg t_{i,k-3} \vee \ell_{k-1} \vee \ell_k)$$

Exercise 11 Convert the following SAT formula into an equivalent 3-SAT formula: $\varphi = (x_1 \vee \neg x_2 \vee x_3 \vee \neg x_4 \vee x_5) \wedge (x_1 \vee \neg x_2) \wedge (\neg x_1 \vee x_2 \vee x_4)$

Solution

$$\varphi = (x_1 \vee \neg x_2 \vee x_3 \vee \neg x_4 \vee x_5) \wedge (x_1 \vee \neg x_2) \wedge (\neg x_1 \vee x_2 \vee x_4)$$

Then the construction yields:

$$\begin{aligned} \widetilde{\varphi} = \quad & (t_{1,1} \vee x_1 \vee \neg x_2) \wedge (\neg t_{1,1} \vee x_3 \vee t_{1,2}) \wedge (\neg t_{1,2} \vee \neg x_4 \vee x_5) \\ & \wedge (y_2 \vee x_1 \vee \neg x_2) \wedge ((\neg y_2 \vee x_1 \vee \neg x_2)) \\ & \wedge (\neg x_1 \vee x_2 \vee x_4) \end{aligned}$$

Reduction from SAT to 3-SAT: Demonstration We verify that with the previous construction:

- If an assignment of the variables makes each c_i true, it can be easily extended to make each φ_i true.
- Conversely, if an assignment of the variables makes each φ_i true, restricting this assignment to the variables of c_i makes c_i true.
- Therefore,
$$c_i \text{ is satisfiable}$$
$$\Updownarrow$$
$$\varphi_i \text{ is satisfiable with the same values for the variables of } c_i$$
- Since the variables added in φ_i only appear in φ_i:
$$\varphi \text{ is satisfiable} \iff \widetilde{\varphi} \text{ is satisfiable}$$

Last Step in SAT to 3-SAT Reduction

- The computational time is reduced to writing the clauses of φ_i, whose length is polynomial in relation to the size of φ.
- The entire reduction process is therefore carried out in polynomial time.

We have thus established:

$$\text{SAT} \leq_p \text{3-SAT}$$

Conclusion:

$$\begin{cases} \text{3-SAT} \in NP \\ \text{SAT} \leq_p \text{3-SAT} \end{cases} \implies \text{3-SAT is NP-complete}$$

2.5.5 Establishing NP-Completeness for STABLE Problem (Independent Set)

Définition. Data: An undirected graph $G = (V, E)$ and an integer k

Question: Does there exist a subset $V' \subset V$ with $|V'| = k$, such that $u, v \in V' \Rightarrow (u, v) \notin E$?

The STABLE problem is the decision problem which, given a graph G and an integer k, aims to determine whether there exists in G a set of k vertices pairwise non-adjacent. Such a set is called a stable set of k vertices.

Théorème. *The STABLE problem is NP-complete*

To prove this theorem, we establish that:

1. STABLE $\in NP$
2. 3-SAT $\leq_p$ STABLE (It is already known that 3-SAT is an NP-complete problem)

STABLE Proven NP-Complete

1. STABLE $\in NP$?
   A verifier for STABLE takes as input a graph and a set of vertices V' (certificate), and checks whether V' is a set of pairwise non-adjacent vertices of size k. This algorithm runs in $O(|V|^2)$ in the worst case, making the verifier polynomial. Consequently, STABLE $\in NP$.

2. 3-SAT $\leq_p$ STABLE ?
   For any instance φ of 3-SAT, we associate an instance G_φ, k_φ of STABLE such that

   $$\varphi \text{ is satisfiable } \iff G_\varphi \text{ has an independent set of size } k_\varphi$$

   Remarque. We construct G_φ, k_φ from φ in polynomial time (polynomial reduction) with respect to the size $|\varphi|$ of φ

Reduction 3-SAT to STABLE Let $\varphi = \bigwedge_{1 \leq j \leq k} (x_{1j} \vee x_{2j} \vee x_{3j})$. We construct a graph G_φ with $3k$ vertices, one for each occurrence of a literal in a clause.

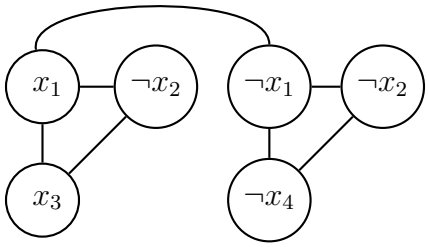
- Constraints related to literals: G_φ has an edge between each vertex associated with a literal x_i and each vertex associated with a literal $\neg x_i$ (thus, an independent set of G_φ corresponds to a truth value assignment to a subset of the variables)
- Constraints associated with clauses: for example, for a clause in φ of the form $c = (x_1 \vee \neg x_2 \vee x_3)$, G_φ has edges $(x_1, \neg x_2)$, $(\neg x_2, x_3)$, (x_3, x_1) forming a triangle (thus, an independent set of G_φ contains at most one of the three vertices associated with the clause c)

We choose the integer k_φ to be equal to k (the number of clauses in φ)

Reduction from 3-SAT to STABLE: Example for Construction of G_φ

$$\varphi = (x_1 \vee \neg x_2 \vee x_3) \wedge (\neg x_1 \vee \neg x_2 \vee \neg x_4)$$

Then the construction yields the following graph G_φ:



Reduction from 3-SAT to STABLE: Demonstration We now show that φ is satisfiable if and only if G_φ has a stable of size k_φ:

- If φ is satisfiable, we consider an assignment of variables satisfying φ. For each clause c in φ, we choose y_c as a literal in c made true by the assignment. This defines k vertices forming a stable in G_φ.
- Conversely, if G_φ has a stable of size k_φ, then it must have a vertex in each triangle. This vertex corresponds to a literal that makes the associated clause true, forming a consistent variable assignment due to the construction of edges.
- Therefore,

$$\varphi \text{ is satisfiable } \iff G_\varphi \text{ has a stable of size } k_\varphi$$

Last Step in 3-SAT to STABLE Reduction

- The computation time reduces to constructing the graph G_φ from the formula φ
- It is polynomial with respect to the size of φ

Therefore, we have:

$$\text{3-SAT} \leq_p \text{STABLE}$$

Conclusion:

$$\begin{cases} \text{STABLE} \in NP \\ \text{3-SAT} \leq_p \text{STABLE} \end{cases} \implies \text{STABLE is NP-complete}$$

2.5.6 Establishing NP-Completeness for CLIQUE Problem

Définition. Data: An undirected graph $G = (V, E)$ and an integer k

Question: Does there exist a set $V' \subset V$, with $|V'| = k$, such that $u, v \in V' \Rightarrow (u, v) \in E$?

The CLIQUE problem is to determine if a graph contains a clique (a subset of graph vertices all adjacent to each other, also known as a complete subgraph) of size k.

Théorème. *The CLIQUE problem is NP-complete*

To prove this theorem, we demonstrate that:

1. CLIQUE $\in NP$
2. STABLE $\leq_p$ CLIQUE (STABLE is an NP-complete problem)

CLIQUE Proven NP-Complete

1. Is CLIQUE $\in NP$?
   (a) **Certificate:** Let the certificate be a set S consisting of nodes in the clique, and S is a subgraph of G.
   (b) **Verification:** Verifying if the number of nodes in S equals k takes $O(1)$ time. Verifying if each vertex has an external degree of $(k-1)$ takes $O(k^2) = O(|V|^2)$ time (since $k \leq |V|$). Thus, the decision problem for clique has a polynomial verifier.
   $\Longrightarrow$ Therefore, CLIQUE $\in NP$
2. STABLE $\leq_p$ CLIQUE?
   For any instance G, k of STABLE, we associate an instance G', k' of CLIQUE such that
   $$G \text{ has a stable of size } k \Longleftrightarrow G' \text{ has a clique of size } k'$$
   Remarque. We construct G', k' from G, k in polynomial time relative to the size of G.

Reduction STABLE to CLIQUE

- We will construct the graph $G' = (V', E')$ from the graph $G = (V, E)$ by the following modifications:
  - $V' = V$, meaning that all vertices of the graph G are part of the graph G'
  - E' = complement of the edges E, that is, the edges not present in the original graph G
- The graph G' is the complementary graph of G

Reduction from STABLE to CLIQUE: Example of Constructing the Complement Graph

Reduction from STABLE to CLIQUE: Demonstration We now demonstrate that G has a stable of size k if and only if G' has a clique of size k':

- Suppose the graph G contains a clique of size k. This implies that there are k vertices in G, where each of these vertices is connected by an edge to the remaining vertices. Furthermore, since these edges are contained in G, they cannot be present in G'. Therefore, these k vertices are not adjacent to each other in G' and thus form a stable of size k.

- Conversely, assume the complementary graph G' has a stable of vertices of size k'. None of these vertices share an edge with other vertices. When we complete the graph to obtain G, these k' vertices will share an edge and thus become adjacent to each other. Consequently, the graph G will have a clique of size k'.

- Hence,
$$G \text{ has a stable of size } k \iff G' \text{ has a clique of size } k$$

Last Step in STABLE to CLIQUE Reduction

- The computation time reduces to calculating the complementary graph G', which can be done in the worst case in $O(|V|^2)$ time.

- The reduction process is thus carried out in polynomial time.

We therefore have:
$$\text{STABLE} \leq_p \text{CLIQUE}$$

Conclusion:
$$\begin{cases} \text{CLIQUE} \in NP \\ \text{STABLE} \leq_p \text{CLIQUE} \end{cases} \implies \text{CLIQUE is NP-complete}$$

2.5.7 Establishing NP-Completeness for 3-COLORABILITY Problem

Définition. Data: An undirected graph $G = (V, E)$.

Question: Does there exist a coloring of the graph using at most 3 colors, such that no two adjacent vertices receive the same color?

A k-COLORABILITY problem for undirected graphs aims to determine if there is a vertex-coloring of the graph such that no two adjacent vertices share the same color, and at most k colors are used to complete the graph coloring.

Théorème. *The 3-COLORABILITY problem is NP-complete.*

To prove this theorem, we show that:

1. 3-COLORABILITY $\in NP$

2. 3-SAT $\leq_p$ 3-COLORABILITY

3-COLORABILITY Proven NP-Complete

1. Is 3-COLORABILITY in NP?

   (a) **Certificate:** A coloring assignment $\{c_1, c_2, c_3\}$ where each vertex is assigned one of these three colors.

   (b) **Verification:** Verify that for each edge (u, v), the color of vertex u is different from that of vertex v. The number of edges in a graph in the worst case is $\frac{|V|(|V|-1)}{2}$ ($|V|$ is the number of vertices in the graph). Thus, the verification takes time in $O(|V|^2)$
   Therefore, the 3-COLORABILITY problem has a polynomial verifier
   $\Longrightarrow$ Hence, 3-COLORABILITY $\in NP$

2. Is 3-SAT $\leq_p$ 3-COLORABILITY?
   For every instance φ of 3-SAT, we associate an instance G_φ of the 3-COLORABILITY problem such that

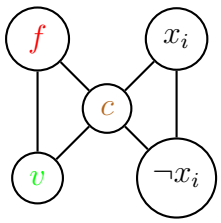
$$\varphi \text{ is satisfiable} \Longleftrightarrow G_\varphi \text{ has a 3-coloring}$$

Remarque 2.5.1. We will construct G_φ from φ in polynomial time relative to the size of φ

Reduction 3-SAT to 3-COLORABILITY Let $\varphi = c_1 \wedge ... \wedge c_m$. We construct a graph G_φ as follows:

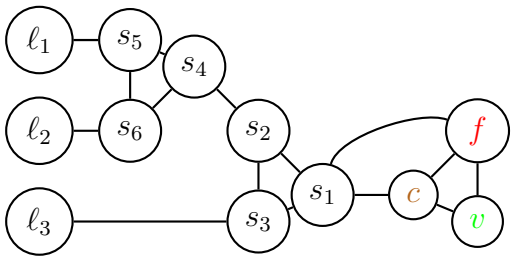
1. We create three vertices v, f, c (for true, false, control). These three vertices are connected pairwise in a triangle, so they must all be of different colors. We associate a vertex with each variable and its complement. To ensure that a variable takes either the true or false value, for each variable x_i we construct a triangle with vertices x_i, $\neg x_i$, and c
   $\Longrightarrow$ This results in a total of $2n + 3$ vertices and $3n + 3$ edges

   Illustration:



   This enforces that either color(x_i) = true and color($\neg x_i$) = false, or color(x_i) = false and color($\neg x_i$) = true

2. We now need to encode the evaluation rules of a clause. To do this, we introduce the following subgraph, which corresponds to a clause $\ell_1 \vee \ell_2 \vee \ell_3$



   Property:

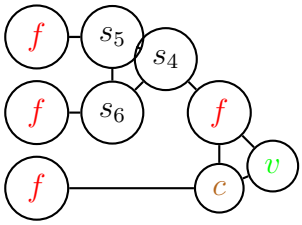
   - If ℓ_1, ℓ_2, ℓ_3 are colored false in a 3-coloring, then the output node s_1 will be colored false (which is impossible as s_1 must be true)
   - If any of ℓ_1, ℓ_2, ℓ_3 is colored true, then the constructed subgraph can be 3-colorable in a way that the output node s_1 is colored true

Similarly, we create subgraphs associated with the other clauses of φ
$\Longrightarrow$ This results in $6m$ new vertices and $12m$ new edges.

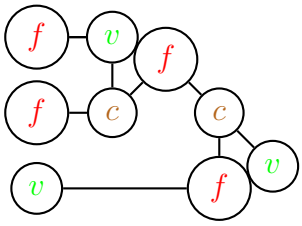
In total, the graph has $2n + 6m + 3$ vertices and $3n + 12m + 3$ edges

Proof of the Property:

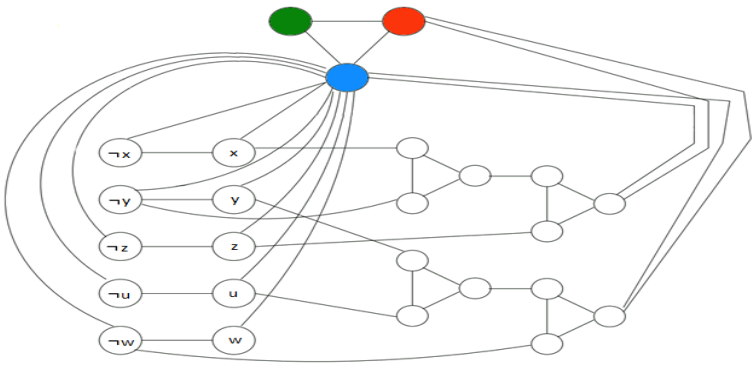
(a) If ℓ_1, ℓ_2, ℓ_3 are all colored false, then we are forced to use the coloring shown in the figure. But then s_4, s_5, s_6 must all be colored with different colors, and none of them can be colored false, so there is no legal coloring.



(a) The rest of the proof considers the 7 possible color combinations of ℓ_1, ℓ_2, ℓ_3 such that at least one is colored true and the others are colored false, and shows that a 3-coloring exists in each case.
For example, if ℓ_3 is colored true but ℓ_1 and ℓ_2 are colored false, we have the legal 3-coloring illustrated in the figure. The other cases are similar.



Reduction from 3-SAT to 3-COLORABILITY: Example $\varphi = (x \vee \neg y \vee z) \wedge (y \vee u \vee \neg w)$, the graph G_φ constructed from φ is as follows:



Reduction from 3-SAT to 3-COLORABILITY: Demonstration It remains to show that φ is satisfiable if and only if the constructed graph G_φ is 3-colorable:

- **φ is satisfiable implies G_φ is 3-colorable**
  - If φ is satisfiable, then in each clause, at least one of the literals x_i must be true. As a result, the vertex corresponding to x_i in G_φ can be assigned a true color and $\neg x_i$ can be assigned a false color. Now, by extending this, for each clause, the corresponding subgraph in G_φ can be 3-colorable. Therefore, the graph can be 3-colorable (by the property above).
- **G_φ is 3-colorable implies φ is satisfiable**
  - If φ is not satisfiable, there exists at least one clause $c_j = (\ell_1 \vee \ell_2 \vee \ell_3)$ where all three literals ℓ_1, ℓ_2, ℓ_3 are false. In this case, the subgraph corresponding to c_j cannot be 3-colorable. The output node of the subgraph is colored false, which

is impossible because it must be true (by the property above). Therefore, the graph G_φ is not 3-colorable [28].

Last Step in 3-SAT to 3-COLORABILITY Reduction

- The computation time is reduced to creating the vertices and edges of G_φ, the number of which ($2n + 6m + 3$ vertices and $3n + 12m + 3$ edges) is polynomial in relation to the size of φ.
- The reduction is thus polynomial.

Therefore, we have:

$$\text{3-SAT} \leq_p \text{3-COLORABILITY}$$

Conclusion:

$$\begin{cases} \text{3-COLORABILITY} \in NP \\ \text{3-SAT} \leq_p \text{3-COLORABILITY} \end{cases} \implies \text{3-COLORABILITY is NP-complete}$$

2.6 The $P \stackrel{?}{=} NP$ Conjecture

The crucial question in complexity theory is to determine:

- Whether NP-complete problems can be solved polynomially, in which case $P = NP$.
- Or if polynomial algorithms for them will never be found, in which case $P \neq NP$.

This question is still unsettled. As hundreds of NP-complete problems resist the assault of research efforts, most theoretical computer scientists today conjecture that $P \neq NP$. Problems with undetermined status, which are in NP, are among the hard problems to solve (the best available algorithms have exponential complexity), but a proof of their membership in NPC is not yet available (Lebbah, 2021).

2.7 Impact on the Practical Approach to a Real Problem

There are combinatorial problems[29] that you shouldn't insist on solving. If you have to solve a combinatorial problem, it's a good idea to consult a directory of NP-complete problems to see if the associated existence problem is listed there[30]. If your problem is NP-complete, it's better to give up hope of finding a polynomial algorithm. The problem of solving arises for any problem where polynomial algorithms are not available (primarily NP-complete problems). In practice, problems are solved by leveraging (Lebbah, 2021):

- **Data size:** Complete enumeration can be feasible for small cases.
- **Average complexity:** An exponential algorithm in its worst-case scenario can be quite fast on average (like the simplex algorithm[31]).

[28] We are using a proof by contrapositive. A proof by contrapositive shows an implication $P \implies Q$ by proving its contrapositive $\neg Q \implies \neg P$.

[29] A problem is classified as combinatorial when its solution involves exploring a huge number of combinations.

[30] Over 3000 problems listed in 2019!

[31] The simplex algorithm is an algorithm for solving linear programming problems. It's probably the first algorithm that allows minimizing a function over a set defined by inequalities.

- **Based on the Nature of Data:** NP-completeness proofs consider a problem in its most general form, without assumptions about the data. The problem might become easier for specific cases or for specific data of a company, never for arbitrary cases.
- **Using Methods to Reduce Combinatorial Complexity:** These include tree-based methods, dynamic programming, and constraint programming – intelligent enumeration methods.
- **Applying Approximation Methods:** If accepting solutions of good quality without optimality guarantees[32], fast algorithms for local search[33] can be found. In a business context, a solution is valuable if it enables cost savings compared to existing solutions or manual methods, even if it's not optimal.

Conclusion

This chapter introduced important distinctions between computational problems based on their complexity and solvability. We explored decision, optimization, and search problems, and classified them as tractable, intractable, or undecidable.

We focused on complexity classes such as P, NP, and NP-complete, and examined common NP-complete problems like 3-SAT and CLIQUE. Techniques for proving NP-completeness through reductions were also presented.

The *P vs NP* question was highlighted for its theoretical and practical importance. Lastly, we discussed how problem classification determines whether exact or approximate solution methods are used.

These concepts form a foundation for addressing complex AI problems in the next chapter.

[32] An optimal solution (optimization problem) is a feasible solution where the objective function achieves its maximum (or minimum) value, i.e., the highest profit or the least cost.

[33] Local search involves moving from one solution to a neighboring solution in the space of candidate solutions (the search space) until a solution considered optimal is found or the allotted time is exceeded.

Chapter 3

Problem Solving and Artificial Intelligence

Introduction

Problem solving is the process of selecting and sequencing actions in order to achieve specific goals while respecting the constraints of a given environment. This process consists of three main components:

- an **initial state** (representing the problem to be solved),
- a **goal state** (representing a satisfactory solution),
- a set of **actions** or **operators** that transition the system from one state to another.

The objective of the problem-solving process is to find a sequence of actions that leads from the initial state to a goal state.

The complexity of this process arises from the fact that, from a given state, multiple actions may be possible, each leading to different resulting states. The naive approach of exhaustively trying all possible sequences to find a solution leads to what is known as a **combinatorial explosion**. In many cases, one must choose between several possible options or seek the best solution among many—this is the domain of **combinatorial optimization problems**.

Artificial Intelligence (AI) addresses this challenge by using **heuristics**—strategies or techniques that steer the search by prioritizing actions most likely to lead to a solution.

Many problem-solving tasks can be framed as follows:

- A **sequence of operators** applied to transform an initial state into a goal state.
- The **initial state** describes the state of the "world" before any action is taken.
- The **goal state** represents the conditions under which the problem is considered solved.
- There may be **multiple acceptable goal states** that satisfy the solution criteria.

To solve such problems, the general idea is to:

- Consider the initial state,

- Identify the operators that can change the current state,
- Apply these operators systematically (or heuristically),
- Test whether the resulting state satisfies the goal conditions.

3.1 Formal Definition of a Problem

A problem can be formally defined by the following five components (Russell, Russell, Norvig, & Davis, 2010):

1. **An initial state**: the starting point from which the problem-solving process begins. It represents the initial configuration or description of the world.
2. **A set of actions**: these are the operations or transitions that can be applied to move from one state to another.
3. **A successor function**: this function specifies the resulting state when a particular action is applied to a given state. It defines the dynamics of the environment.
4. **A set of goal states**: these are the states that represent valid solutions. There must be a test (goal test) to determine whether a given state satisfies the problem's objective.
5. **A cost function**: this associates a non-negative numerical cost with each action. It allows the evaluation and comparison of different possible paths.

A problem can be visualized as a **directed graph**:

- The **nodes** of the graph represent the states that are reachable from the initial state.
- The **edges** represent the actions that transition from one state to another.

This graph is called the **state space**.

- A **solution** to the problem is a path in this graph from the initial state to a goal state.
- A solution is said to be **optimal** if the total cost (sum of the costs of actions along the path) is minimal among all possible solutions.

3.2 Examples of Problems

3.2.1 The 8-puzzle Problem

State: The states are configurations of the eight tiles in the nine squares of a 3×3 grid. See Figure 3.1 for two example states.
Initial State: Any configuration can be selected as the initial state.
Actions: There are four possible actions corresponding to the directions in which the empty square (blank) can be moved: *up*, *down*, *left*, and *right*. In some configurations, only 2 or 3 of these moves may be allowed.

Successor Function: This function defines the resulting state of applying an action to a given state. For example, applying the action *right* to the first state in Figure 3.1 produces the second state.
Goal Test: The goal is to reach a specified configuration (typically the tiles in numerical order). Any configuration can be chosen as the goal state.
Action Cost: Each tile movement costs 1 unit. This allows us to search for a solution with the minimal number of moves.

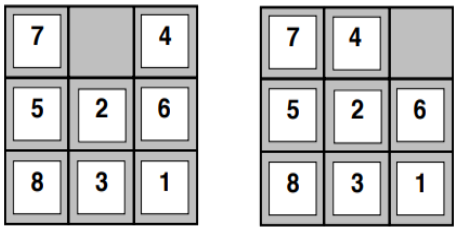


Figure 3.1: Two configurations of the 8-puzzle problem (Russell et al., 2010).

The 8-puzzle is often used to test search algorithms. By increasing the size of the grid, we can create more complex problems:

- The 3×3 and 4×4 puzzles have state spaces of size 181,440 and approximately 1.3 billion, respectively.
- Modern algorithms are able to solve 3×3 and 4×4 puzzles efficiently.
- The 5×5 puzzle, which has a state space of size 10^{25}, remains very challenging.

3.2.2 Eight Queens Problem

The objective is to place eight queens on a chessboard (8×8 grid) such that no queen attacks another—that is, no two queens share the same row, column, or diagonal.

Formal definition of the problem:

- **States:** Any configuration with 0 to 8 queens on the board.
- **Initial state:** An empty board.
- **Actions:** Add a queen to an empty square of the board.
- **Successor function:** The resulting configuration after placing a queen on a specific square.
- **Goal test:** A configuration with eight queens where no two attack each other.
- **Action cost:** Can be zero or a constant per action, since the interest lies in reaching a goal state rather than optimizing a path.

State space reduction: The total number of possible configurations is very large (1.8×10^{14}), but this can be drastically reduced (to 2057) by exploring only conflict-free configurations.

Optimized formulation:

- **States:** Configurations with n queens ($0 \leq n \leq 8$), placed one per column in the leftmost n columns, with no mutual attacks.
- **Actions:** Add a queen to an empty square in the leftmost unfilled column, avoiding any conflict.

Alternative formulation (complete initial configuration):

- **States:** Configurations with 8 queens, one per column.
- **Initial state:** Any randomly chosen configuration.
- **Actions:** Move a queen within its column.

Note: In the incremental formulation, states are never revisited. This is not the case in the complete configuration formulation or in puzzles like the sliding puzzle, where a tile or queen may return to a previous position. Care must be taken to avoid revisiting states or entering infinite loops.

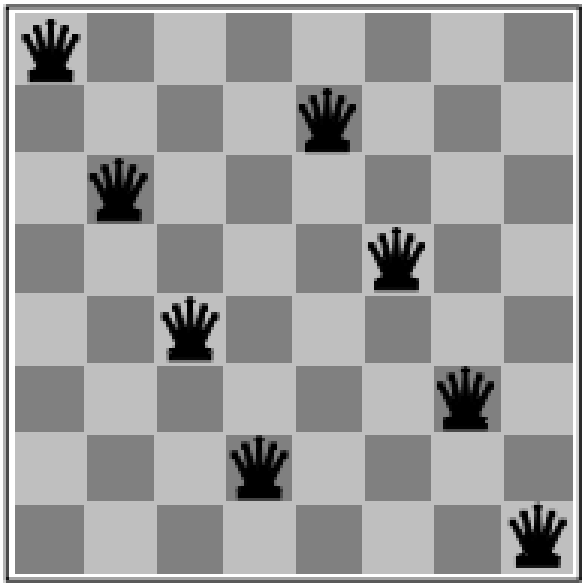


Figure 3.2: An almost correct solution to the Eight Queens problem (Russell et al., 2010).

3.3 Types of Problems

We observe that the two problems previously presented are quite different in nature:

- In the sliding puzzle, the goal state is known from the beginning, and the challenge lies in finding a sequence of actions to reach it.
- In contrast, for the Eight Queens problem, the path to the solution is irrelevant—we are only interested in finding a final state that satisfies the constraints.

These two games are examples of two major classes of problems studied in Artificial Intelligence: **planning problems** and **constraint satisfaction problems** (Russell et al., 2010).

Planning Problems

- The initial state is known.
- The goal state is known.
- The objective is to **find a path** from the initial to the goal state, possibly optimizing for certain criteria.

Constraint Satisfaction Problems

- The initial state is known.
- The path to the solution does not matter.
- The objective is to **find a final state** that satisfies a given set of constraints.

3.3.1 Planning Problems

In a planning problem, the objective is to find a sequence of actions that connects an initial state to a goal state.

Planning problems are common in everyday life:

- Finding the shortest path to reach a destination.
- Planning a trip using public transportation, trains, or flights.

Example: Flight Trip Planning

States: A state is defined by the current airport, date, and time.

Initial state: The airport from which the client departs, along with the desired departure date and time.

Actions: Flights from one airport to another.

Successor function: The possible flights that depart from the current airport at a time later than the current one. (We must also consider a minimum connection time to avoid missed connections.)

Goal test: States where the client is at the destination airport.

Action costs: This depends on client preferences:

- Cost of 1 per flight to minimize the number of flights.
- Duration of the flights to minimize total travel time.
- Price of the tickets to find the cheapest itinerary.

A similar problem is network routing, where the goal is to find efficient paths to send packets through a network.

3.3.2 Constraint Satisfaction Problems (CSP)

A constraint satisfaction problem involves a set of variables, domains of allowed values for each variable, and a set of constraints on combinations of variable values. The goal is to find an assignment of values such that all constraints are satisfied.

Examples of CSPs:

- Managing runway assignments at an airport.
- Creating a university timetable.

Example: University Timetabling

States: A partial timetable, i.e., a selection of (day, time, room) triples for a subset of courses.

Initial state: An empty timetable.

Actions: Choose a (day, time, room) triple for a course not yet scheduled.

Successor function: The resulting timetable with the new assignment.

Goal test: A timetable including all courses and satisfying all constraints (e.g., no overlaps, required time slots).

Action cost: A constant cost per action.

3.3.3 Multi-Agent Problems

So far, all the problems involved a single agent[1] making decisions independently. However, in many real-world scenarios, our decisions are influenced by the actions of others—especially in multi-player games.

Example: Multi-Player Games

- These are more complex because the player (human or AI) cannot control the actions of opponents.
- Rather than planning a single path from an initial to a goal state (which may involve unknown opponent moves), we look for a **strategy**: a mapping from each possible state where it's the player's turn to an appropriate action.

Example: Chess Game

States: A configuration of the chessboard, plus an indicator of whose turn it is.

Initial state: The standard initial configuration of a chess game and the starting player.

Actions: Legal moves available to the player whose turn it is (including resigning).

Successor function: The new configuration resulting from a move, with the turn passed to the opponent.

Goal test: Many possible endings exist; the most well-known is checkmate.

Action cost: Can be set to 1 per move to prefer faster paths to winning.

[1] An agent in AI is an entity capable of perceiving and acting in an environment.

3.4 Problem Solving Methods

Two main families of methods are distinguished according to how the solution is built from an initial state (Nilsson, 2014; Russell et al., 2010):

1. **Search in the State Space**
   The process always starts from the initial state, then the following steps are repeated in a loop until termination:

   - If there are no more states to explore, return failure.
   - Otherwise, select one of the remaining states to explore (*):
     - If the selected state is a goal state, return the corresponding solution.
     - Otherwise, remove this state from the set of states to explore, and replace it with its successor states.

   What differentiates the various algorithms is the strategy used in step (*).

2. **Problem Decomposition**

   - The problem is broken down into sub-problems until reaching "trivial" sub-problems.
   - These sub-problems are solved using the previous method.
   - The decomposition tree is then traversed back up to obtain the solution to the original problem.

3.5 Algorithm Evaluation

To compare different algorithms, the following criteria are typically used (Nilsson, 2014; Russell et al., 2010):

Time Complexity How much time does the algorithm take to find the solution?

Space Complexity How much memory is used during the search for a solution?

Completeness Does the algorithm always find a solution if one exists?

Optimality Does the algorithm always return the best solution?

Complexity Analysis

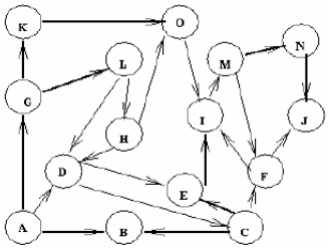
Time and space complexities depend on the following parameters:

- b — the maximum branching factor of the search tree
- d — the depth at which the (best) solution node is found
- m — the maximum depth of the search space (possibly $m = \infty$)

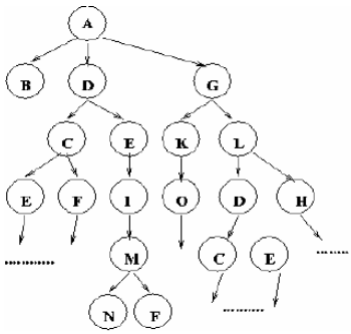
3.6 Search in the State Space

3.6.1 Construction of the Search Tree

Problem: expressed by a state graph



Transformed into a search tree representing reasoning paths



Reasoning about a problem means constructing this tree by identifying **the most relevant reasoning path** that corresponds to the solution.

Additional Notes on the construction of the search tree:

- Simulated exploration of the state space (state graph) by generating successor states from already explored states.
- Nodes are examined and then developed or extended through a process called *state expansion.*
- Each path represents a reasoning path that either leads to a solution or to failure.

3.6.2 Generic Search Algorithm

```
1  Function Search(initial_state, SUCCESSORS, GOAL_TEST, COST):
2      loop
3          nodes_to_process ← CREATE_LIST(CREATE_NODE(initial_state, [],
             0));
4          if IS_EMPTY(nodes_to_process) then
5              return failure
6          end
7          node ← REMOVE_FIRST_NODE(nodes_to_process);
8          if GOAL_TEST(STATE(node)) = true then
9              return PATH(node), STATE(node)
10         end
11         foreach (action, state) ∈ SUCCESSORS(STATE(node)) do
12             path ← [action, PATH(node)];
13             path_cost ← PATH_COST(node) + COST(action);
14             s ← CREATE_NODE(state, path, path_cost);
15             INSERT(s, nodes_to_process);
16         end
17     end
```

Algorithm 13: Generic Search Algorithm(Nilsson, 2014; Russell et al., 2010).

The algorithm (Algorithm 13) takes as input the description of a problem: an initial state, a successor function, a goal test, and a cost function.

- The first step is to initialize a list[2] of nodes to process. A node consists of:
  - a state,
  - a path (a sequence of actions from the initial state to the current state),
  - and the cost of that path.
- The search starts with a single node corresponding to the initial state.
- At each iteration of the loop:
  - If the list of nodes to process is empty, this means all possible paths have been explored without success. The algorithm returns **"failure"**.
  - Otherwise, the first node is removed from the list for processing.
  - If the state of this node satisfies the goal test, the algorithm returns the goal state and the path that leads to it.
  - If not, the algorithm generates all successor states of the current node and inserts them into the list of nodes to process.

3.6.3 Classes of Search Algorithms

Given that state spaces are generally very large, search algorithms are used to navigate the state space efficiently and effectively. A search algorithm defines how to find the path (solution) from the initial state to the final state. Different algorithms define different

[2]The method elements are inserted depends on the specific search strategy used.

methods to transition from the current state (node) to the next state (node). Some algorithms only provide systematic ways to explore the state space, while others aim for efficient and effective exploration.

Two classes of search algorithms are distinguished:

1. **Uninformed or "blind" algorithms (brute force)**

   - Breadth-First Search
   - Depth-First Search
   - Depth-Limited Search
   - Iterative Deepening Search

   These algorithms perform an exhaustive search, without using any information about the structure of the state space to optimize the search.

2. **Informed or heuristic algorithms**

   - Greedy Algorithm
   - A^* Algorithm

   These algorithms use additional sources of information. They thus achieve better performance.

3.6.4 Uninformed or "Blind" Search Algorithms (Brute Force)

Breadth-First Search (BFS)

- **Principle:** All nodes at a certain depth are examined before nodes at a greater depth (see Figure 3.3).
  - Start with the initial state.
  - Then all its direct successors.
  - Then the successors of the successors.
  - And so on.
- **Implementation:** Use a queue.
  - Always place newly generated nodes at the end of the list of nodes to process.

Note: The functioning of BFS on the example: we start with the initial state, then examine nodes at depth 1, then at depth 2, and so on.

Time Complexity: $O(b^d)$, where b is the maximum branching factor of the search tree and d is the minimal number of actions to reach a goal state from the initial state.

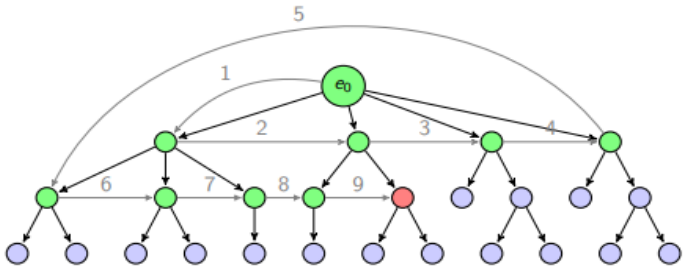


Figure 3.3: Illustration of the BFS algorithm. The algorithm explores all nodes at the present depth level before moving on to the nodes at the next depth level.

Explanation: In the worst case, we examine all nodes up to depth d to find the goal state at that depth. The total number of nodes generated is:

$$1 + b + b^2 + \ldots + b^d + (b^{d+1} - b)$$

The quantity $b^{d+1} - b$ corresponds to the number of depth-$(d+1)$ nodes produced during the expansion of nodes at depth d, minus the successors of the last expanded node (since it's a goal node, we don't expand its successors). Therefore, the time complexity is $O(b^d)$.

Space Complexity: $O(b^d)$

Explanation: Space complexity is $O(b^{d+1}) = O(b^d)$ since at most $1 + b^{d+1} - b$ nodes are kept in memory during the processing of the last node.

Completeness: The algorithm is complete provided the number of successors of each state is finite (which is very often the case in practical problems).

Explanation: Let d be the minimal number of actions to reach a goal state. Since BFS examines nodes level by level and each level contains a finite number of nodes (assuming no state has infinitely many successors), we are guaranteed to reach level d.

Optimality:

- Always finds a solution at the smallest possible depth.
- Therefore, if we seek any solution or the one with the least number of actions, BFS is optimal.
- BFS is optimal when path cost depends only on the number of actions, but it is not optimal in the general case where actions have different costs.

Figure 3.4 illustrates the time and memory requirements of BFS for a problem with 10 successors per state (assuming 10,000 nodes per second and 1,000 bytes per node).

Even for a depth of $d = 6$, BFS consumes 10 gigabytes of memory, and for $d = 8$, it requires 1 terabyte. The algorithm takes 35 years to find a solution of depth $d = 12$.

The high time and space complexity restricts the use of breadth-first search to small-scale problems.

Depth	Nodes Generated	Time	Memory
2	1,100	0.11 sec	1 megabyte
4	111,100	11 sec	106 megabytes
6	10^7	19 min	10 gigabytes
8	10^9	31 h	1 terabyte
10	10^{11}	129 days	101 terabytes
12	10^{13}	35 years	10 petabytes

Figure 3.4: Time and memory required by breadth-first search for a problem with 10 successors per state (assuming 10,000 nodes per second and 1,000 bytes per node).

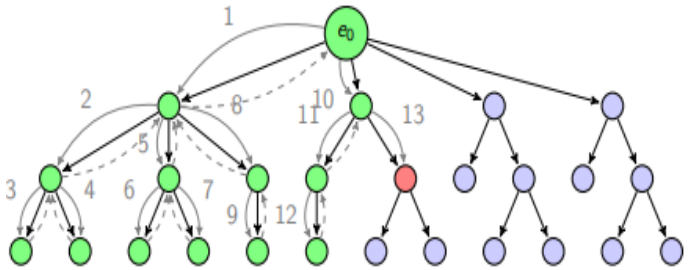


Figure 3.5: Depth-First Search traverses the tree as deep as possible before backtracking.

Depth-First Search (DFS)

Depth-First Search (DFS) is an uninformed search algorithm that explores as deeply as possible along each branch before backtracking.

- **Principle:** DFS follows a path as deep as possible before backtracking (see Figure 3.5).
- **Implementation:** uses a stack.
  - Successors of the current node are added to the top of the list of nodes to explore.

Time complexity: $1+b+b^2+\cdots+b^m = O(b^m)$, which can be problematic if the maximum depth m is much greater than the solution depth d.

Space complexity: $O(b \times m)$

Explanation: For a problem with b successors and a maximum depth of m actions, we need to keep at most $1 + (m \times (b-1))$ nodes in memory (corresponding to the current path under development, plus the depth-1 successors not yet processed, the depth-2 successors not yet processed, and so on).

Completeness: DFS is not complete — it may follow an infinite path and miss existing solutions reachable through shorter paths.

Explanation: Depth-first search is not complete because the algorithm may follow an infinite path. However, if there are only a finite number of possible paths (which is rarely the case), then depth-first search will be complete.

Optimality: DFS is not optimal — it does not guarantee that the first found solution is the best one.

DFS is not complete, not optimal, and not better than BFS in time complexity. However, it has the major advantage of requiring less memory, even when b and m are large. Thus, this lower space complexity makes DFS suitable for problems that are impractical with BFS.

Limited Depth-First Search

The DFS algorithm can get lost in an infinite-depth search! (infinite state space). It may blindly follow an infinite path that does not lead to a goal state. A naive way to avoid this problem is to set a maximum depth and not consider paths deeper than this limit.

Time Complexity: $1 + b + b^2 + \ldots + b^l = O(b^l)$, where l is the maximum depth limit.

Space Complexity: $O(b \times l)$.

Completeness: Complete if the maximum depth is greater than or equal to the minimum depth of the solutions ($l \geq d$).
Note: Since we generally don't know what depth is sufficient, we can't guarantee that the algorithm is complete for a given depth.

Optimality: Not optimal (like classical DFS).
Note: As with standard depth-first search, limited depth-first search is generally not optimal.

Iterative Deepening Depth-First Search

- Since we don't know what depth limit to choose, we test them one after the other.
- That is, we perform a depth-limited search with a limit of 1, then with a limit of 2, and continue increasing the depth limit until a solution is found.

Time Complexity: $(d+1) + (d)b + (d-1)b^2 + \ldots + (1)b^d = O(b^d)$
Note: We examine the single node at depth 0 exactly $d+1$ times, nodes at depth 1 exactly d times, depth 2 nodes $d-1$ times, ..., and nodes at depth d only once. While this may seem inefficient due to repeated visits, iterative deepening is actually less time-consuming than breadth-first search (which has a complexity of $O(b^{d+1})$ due to generating nodes at depth $d+1$ during the exploration of nodes at depth d), and better than standard depth-first search (which may go beyond depth d unnecessarily).
Comparison between time complexity of Iterative Deepening (ID) and Breadth-First Search (BFS) for $b = 10$, $d = 5$:
$N(BFS) = 1 + 10 + 100 + 1{,}000 + 10{,}000 + 100{,}000 + (b^{d+1} - b) = 1{,}111{,}101$ nodes
$N(ID) = 6 + 50 + 400 + 3{,}000 + 20{,}000 + 100{,}000 = 132{,}456$ nodes

Space Complexity: $O(b \times d)$
Note: At most $1 + d(b-1)$ nodes are stored in memory when examining nodes at depth d.

Completeness: Complete, provided that the branching factor is finite (as in BFS).

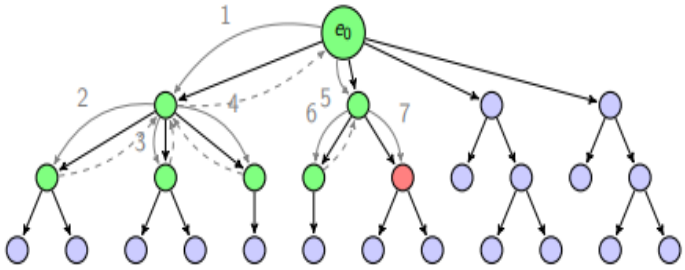


Figure 3.6: Illustration of Iterative Deepening Search: combining the space efficiency of DFS with the completeness and optimality of BFS.

Optimality: Optimal when path cost depends only on the number of actions.

Iterative deepening search therefore combines the advantages of:

- Breadth-First Search $\Longrightarrow$ **completeness** and **optimality**
- Depth-First Search $\Longrightarrow$ **low space complexity**

  This is why iterative deepening is considered the best uninformed search algorithm for large problems where the solution depth is unknown.

3.6.5 Informed or Heuristic Search Algorithms

Motivation for Heuristic Search

Uninformed (or "blind") search algorithms perform an exhaustive search of all possible paths. This makes them inefficient or even unusable for large-scale problems. The solution is to use use heuristic search algorithms. These algorithms incorporate additional information to better guide the search process.

Every heuristic search algorithm uses an evaluation function f that determines the order in which nodes are explored. The list of nodes to be processed is sorted based on their f-values. The evaluation function typically includes an heuristic function $h(n)$: $h(n)$ = Estimated cost of the cheapest path from node n to a goal state. $h(n) = 0$ if n is a goal node.

Example: Romanian Map Problem

The map in Figure 3.7 shows the major cities and road distances in Romania, which we use to illustrate heuristic search.
Consider the following problem: we are located in Arad, Romania and must reach Bucharest by traveling the shortest possible distance.

Using a Heuristic Function

- Let $d(A, B)$ denote the road distance between cities A and B, i.e., the length of the shortest path between them. Example: $d(\text{Zerind}, \text{Sibiu}) = 215$
- To evaluate how close we are to the goal when at city X, it would be useful to know $d(X, \text{Bucharest})$. Then we could define the heuristic as: $h(X) = d(X, \text{Bucharest})$

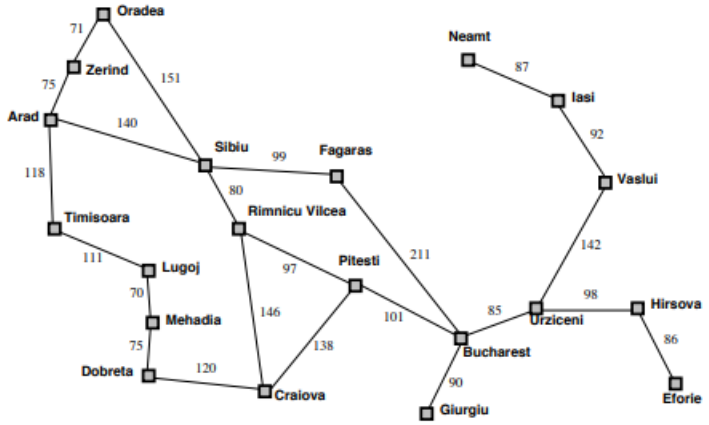


Figure 3.7: Map of Romania showing cities and road distances. Used to illustrate heuristic search from Arad to Bucharest(Russell et al., 2010).

Arad	366	Mehadia	241
Bucarest	0	Neamt	234
Craiova	160	Oradea	380
Dobreta	242	Pitesi	100
Eforie	161	Rimnicu Vilcea	193
Fagaras	176	Sibiu	253
Giurgiu	77	Timisoara	329
Hirsova	151	Urziceni	80
Iasi	226	Vaslui	199
Lugoj	244	Zerind	374

Figure 3.8: Heuristic values: straight-line (as-the-crow-flies) distances to Bucharest(Russell et al., 2010).

- However, the exact function d is often inaccessible, and computing it is non-trivial.[3]
- This is where the heuristic comes in — it serves as an approximation. We approximate $d(X, \text{Bucharest})$ with $h(X)$ = straight-line ("as-the-crow-flies") distance between city X and Bucharest. This value is easily obtained using a road map and a ruler.

As shown in Figure 3.8, the heuristic values represent the estimated distances from each city to Bucharest using straight-line distances.

- These values are used as our heuristic $h(X)$.
- Since road distances are always greater than or equal to straight-line distances, $h(X)$ is only an approximation of the real cost. If our heuristic were perfect (i.e., exactly equal to the real cost to the goal), we wouldn't need to search at all — we could go straight to the solution!

Greedy Best-First Search

The idea of this algorithm is to explore the nodes that appear to be closest to a goal state, hoping to reach a solution more quickly. To do this, it evaluates nodes using only the heuristic function: $f(n) = h(n)$.

[3] In this small-scale example, d can be computed manually, but in large graphs this becomes computationally expensive.

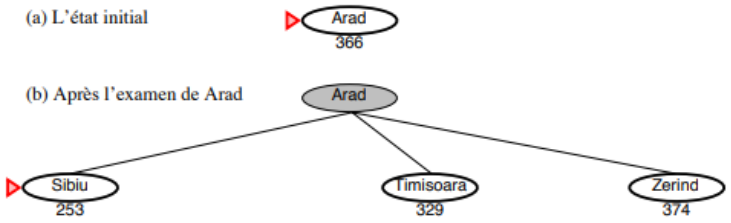


Figure 3.9: Illustration of node selection in Greedy Best-First Search based on $h(n)$(Russell et al., 2010).

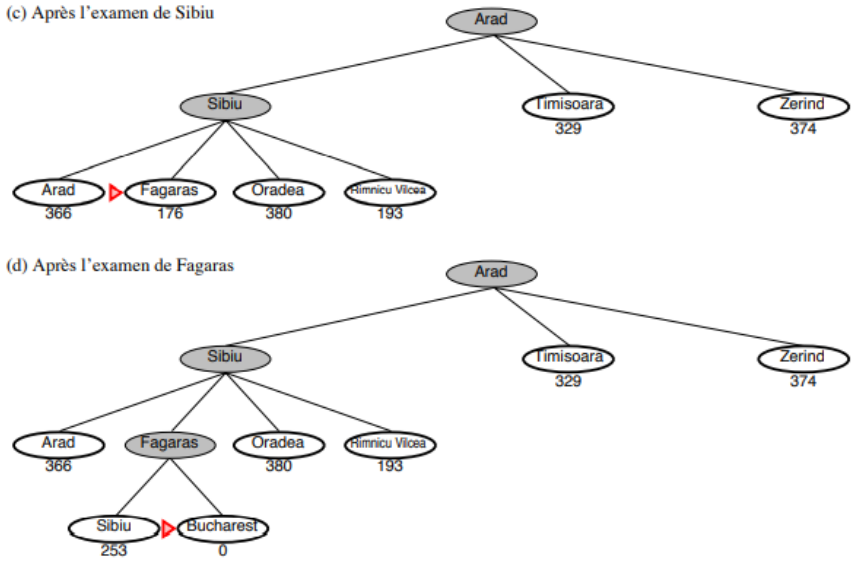


Figure 3.10: Greedy Best-First Search applied to the Romania map problem(Russell et al., 2010).

Execution example: from Arad to Bucharest
We begin at the initial state Arad. From there, we consider the three successors:

- Sibiu ($h = 253$)
- Timisoara ($h = 329$)
- Zerind ($h = 374$)

Since Sibiu has the lowest heuristic value, it is chosen for expansion. Sibiu is not the goal state, so we add its successors to the frontier:

- Arad ($h = 366$)
- Fagaras ($h = 176$)
- Oradea ($h = 380$)
- Rimnicu Vilcea ($h = 193$)

Among all nodes in the frontier, Fagaras has the lowest heuristic value, so it is expanded next. Fagaras is still not a goal state, so its successors are added:

- Sibiu ($h = 253$)
- Bucharest ($h = 0$)

Since Bucharest has the lowest heuristic value (0), it is selected next. As it corresponds to a goal state, the algorithm stops and returns the found path:

$$\text{Path: Arad} \rightarrow \text{Sibiu} \rightarrow \text{Fagaras} \rightarrow \text{Bucharest}$$

Time Complexity: $O(b^m)$, where b is the branching factor and m is the maximum depth of the search tree.

Space Complexity: $O(b^m)$ — the algorithm retains all nodes in memory, which can result in high space usage.

Completeness: Not complete — the algorithm may fall into infinite loops even if a solution exists.
Example: Suppose we want to go from Iasi to Fagaras. The heuristic suggests expanding Neamt first (it appears closer to Fagaras), but Neamt is a dead end. The algorithm reconsiders Iasi, which still appears better than Vaslui, leading to an infinite loop. Thus, the search never reaches the goal, even in a finite state space.

Optimality: Not optimal.
Example: In the previous case (Arad to Bucharest), the algorithm returned the path:
Arad → Sibiu → Fagaras → Bucharest.
However, the shorter and optimal path is:
Arad → Sibiu → Rimnicu Vilcea → Pitesti → Bucharest.

Important Observations:

- The time and space complexity, as well as the optimality of the algorithm, strongly depend on the quality of the heuristic function.
- Greedy Best-First Search prefers nodes whose states appear closer to the goal, but it **ignores the cost of the path taken so far** to reach these nodes. This can lead to suboptimal solutions.

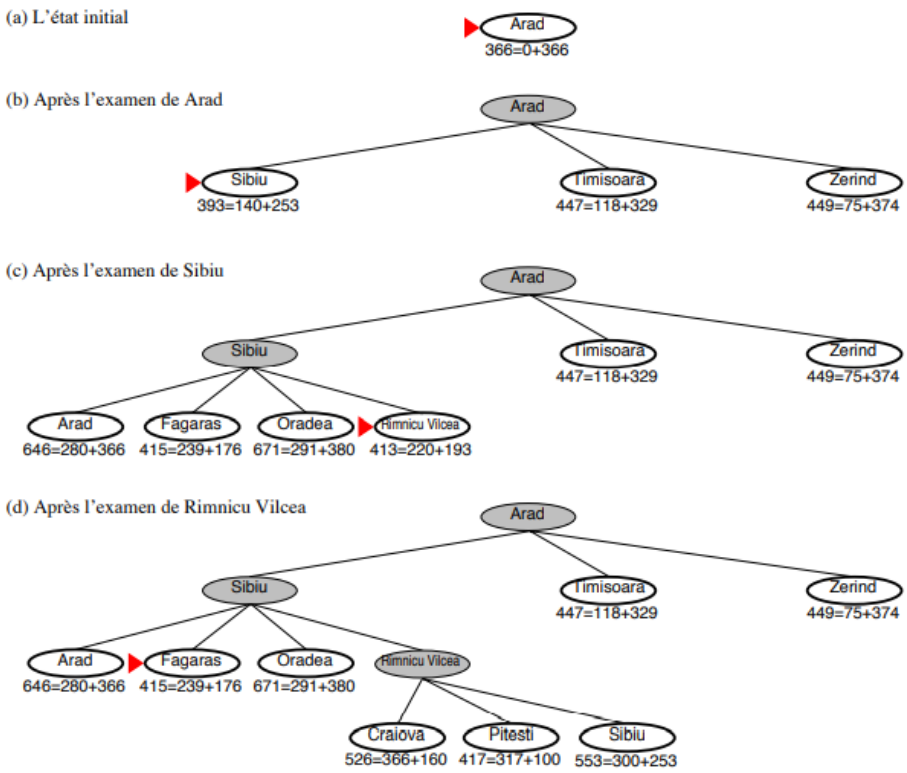


Figure 3.11: Step-by-step execution of A* search (Part 1): from Arad to Bucharest(Russell et al., 2010).

A* Search

Greedy best-first search prefers nodes whose states seem closest to a goal state but ignores the cost of the path from the initial state to those nodes. However, the cost accumulated so far is highly relevant: the total cost of a path through a node n is the sum of:

- the cost to reach n from the initial state, and
- the estimated cost from n to a goal state.

This is the central idea behind the A* search algorithm.

Let $g(n)$ be the cost from the initial state to node n. The evaluation function used by A* is:

$$\boxed{f(n) = g(n) + h(n)}$$

> The evaluation function $f(n)$ provides an estimate of the cost of the best solution passing through node n.

Note: Since $g(n)$ represents the actual cost from the start to n, and $h(n)$ is a heuristic estimate from n to a goal state, their sum $f(n)$ gives an estimate of the total cost of a solution path that goes through n.

Example: A* Search on the Romania Map

We illustrate the execution of A* search in the context of finding a path from Arad to Bucharest.

The algorithm starts at the initial state **Arad**. Since Arad is not a goal state, we examine its three successors and compute their evaluation function $f(n) = g(n) + h(n)$:

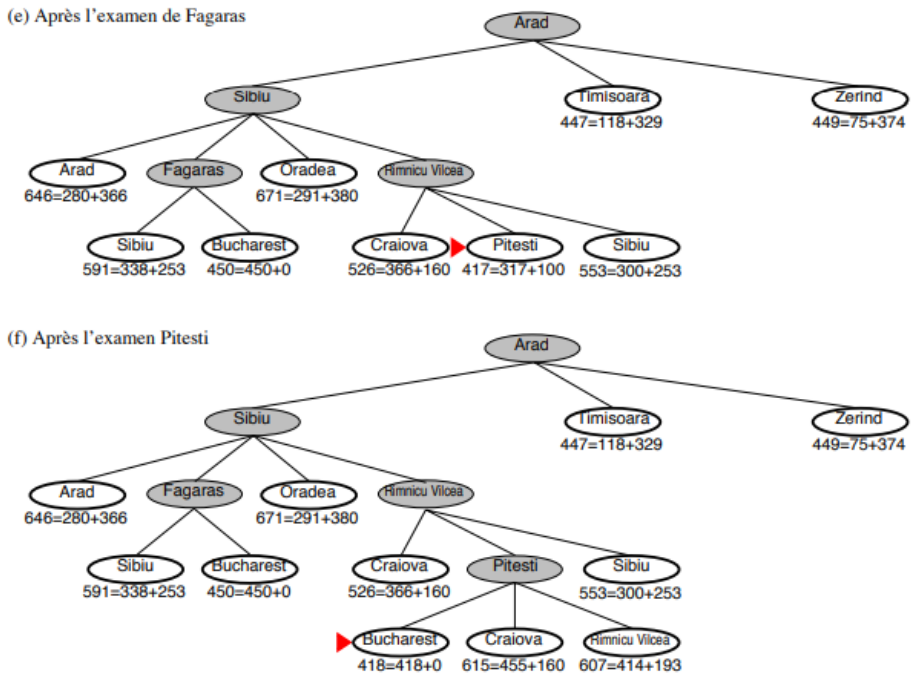


Figure 3.12: Step-by-step execution of A* search (Part 2): continuing from Rimnicu Vilcea(Russell et al., 2010).

- Sibiu: $f = 140 + 253 = 393$
- Timisoara: $f = 118 + 329 = 447$
- Zerind: $f = 75 + 374 = 449$

Sibiu has the lowest f-value, so it is selected next. We expand Sibiu and add its successors to the list of nodes to be considered:

- Arad: $f = 280 + 366 = 646$
- Fagaras: $f = 239 + 176 = 415$
- Oradea: $f = 291 + 380 = 671$
- Rimnicu Vilcea: $f = 220 + 193 = 413$

Rimnicu Vilcea is now the most promising node (lowest f). We expand it and add its successors:

- Craiova: $f = 366 + 160 = 526$
- Pitesti: $f = 317 + 100 = 417$
- Sibiu: $f = 300 + 253 = 553$

Next, among all nodes in the frontier, Fagaras has the lowest f-value, so it is expanded. We add its successors:

- Sibiu: $f = 338 + 253 = 591$

- Bucharest: $f = 450 + 0 = 450$

Although Bucharest is now in the frontier, the algorithm does not stop immediately since a lower-cost solution might still exist. Next, Pitesti is chosen because its $f = 417$ is lower than the 450 for Bucharest. Expanding Pitesti yields:

- Bucharest: $f = 418 + 0 = 418$
- Craiova: $f = 455 + 160 = 615$
- Rimnicu Vilcea: $f = 414 + 193 = 607$

Now, the new node **Bucharest** with $f = 418$ has the best evaluation among all remaining nodes, and since it is a goal state, the algorithm terminates.

Resulting path:

$$\text{Arad} \rightarrow \text{Rimnicu Vilcea} \rightarrow \text{Pitesti} \rightarrow \text{Bucharest}$$

Properties of A* Search

The A* search algorithm is both **complete** and **optimal** under certain conditions:

- The number of successors for each node is finite.
- The heuristic function $h(n)$ is **admissible**.

What is an admissible heuristic?

- A heuristic $h(n)$ is admissible if it never overestimates the true cost of the optimal path from node n to a goal state.
- Formally, for every node n, we must have:

$$h(n) \leq \text{true cost from } n \text{ to goal}$$

In our example, the straight-line (Euclidean) distance to Bucharest is an admissible heuristic. The straight-line distance (heuristic value) is never strictly greater than the actual road distance (true cost), making it a valid underestimate.

Complexity considerations:

- The time and space complexity of A* depends heavily on the heuristic used.
- In the worst case, A* has **exponential time complexity**.
- It also has **high space complexity**, as it keeps all generated nodes in memory.

To overcome these limitations, especially in large-scale problems, several memory-efficient variants of A* have been developed (Edelkamp & Schrödl, 2011):

- **Iterative Deepening A* (IDA*)**: combines the space efficiency of depth-first search with the optimality of A*, performing repeated depth-limited searches with increasing limits.
- **Recursive Best-First Search (RBFS)**: a recursive algorithm that uses linear space by keeping only the current path in memory while backtracking when necessary.
- **Memory-Bounded A* (MA*)**: limits the amount of memory used by discarding less promising nodes when memory is full.
- **Simplified Memory-Bounded A* (SMA*)**: an improved version of MA* that keeps the best forgotten nodes' information to ensure better search efficiency.

Local Search

In **constraint satisfaction problems (CSPs)**, the goal is not to find the path to a solution, but simply to find a **goal state**. Therefore, the search strategy can be adapted:

- **Main idea:** generate states successively without considering paths, until a goal state is found.
- This approach forms the basis of **local search algorithms** (Russell et al., 2010; Stützle, 1999).

Unlike systematic search algorithms, which store paths and keep track of explored and unexplored alternatives at each step, local search algorithms are **not systematic**. However, they offer two main advantages:

1. **Memory efficiency:** they use very little memory, often a constant amount.
2. **Adaptability to large state spaces:** they can often find reasonable solutions in very large or even infinite state spaces, where systematic algorithms are not practical.

   Note: Systematic algorithms explore the search space by storing paths and recording explored alternatives. Local search sacrifices completeness to gain speed and save memory.

In summary, local search prioritizes **speed and memory efficiency** over completeness, making it particularly suitable for complex or large-scale problems.

Greedy local search (hill climbing)

The simplest local search algorithm is called **greedy local search** (Russell et al., 2010; Talbi, 2009), also known as **hill-climbing** (see Algorithm 14).

The algorithm works as follows:

- Start with an initial state chosen randomly. If this state is a goal, the search stops. Otherwise, generate its successors.
- Evaluate the successors using a heuristic function.
- If no successor has a better heuristic value than the current state, the search cannot improve further and stops.
- Otherwise, move to the successor with the highest heuristic value and repeat the process.

Algorithm 14: Greedy Local Search (Hill-Climbing) (Russell et al., 2010)

```
1 function HillClimbing(problem):
2     current ← MAKE-NODE(problem.Initial-State) ;
3     loop
4         neighbor ← the best-valued successor of current ;
5         if neighbor.Value ≤ current.Value then
6             return current.State // No improvement possible, stop search
7         end
8         current ← neighbor ;
9     end
```

Local search algorithms typically use a **complete state formulation**. For the 8-Queens problem, each state contains 8 queens on the board, one per column.

- The **successors** of a state are all possible states generated by moving a single queen to another position in its column. Since each queen has 7 other positions, each state has $8 \times 7 = 56$ successors.
- The **heuristic cost function** h is defined as the number of pairs of queens that attack each other.
- The **global minimum** of this function is 0, which corresponds to perfect solutions where no queens attack each other.

For example, consider a state with $h = 17$. Figure 3.13 illustrates this state and the heuristic values of all its successors. The successors with the lowest heuristic value ($h = 12$) are considered the best moves. Hill-climbing algorithms usually select randomly among these best successors when multiple exist.

Hill-climbing often makes **rapid progress toward a solution**, because it is usually easy to improve a bad state. Starting from the state shown in Figure 3.13(a), only five moves are needed to reach the state in Figure 3.13(b), where $h = 1$, which is **very close to a solution**.

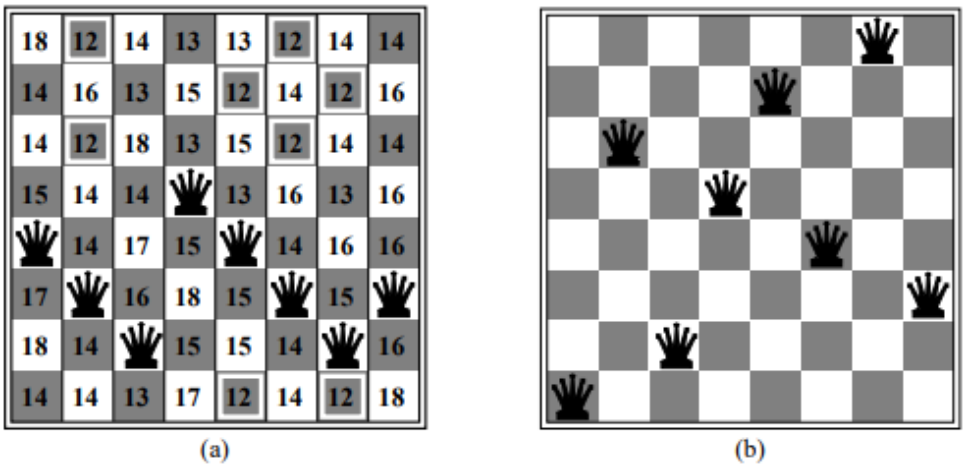


Figure 3.13: (a) An 8-Queens state with heuristic cost $h = 17$, showing the h values of all successors obtained by moving a queen in its column. The best moves are highlighted. (b) A **local minimum** in the 8-Queens state space; the state has $h = 1$ but all successors have higher costs.

Note: A local minimum occurs when a state has a lower heuristic value than all its successors, but the state is not a global solution ($h > 0$). Hill-climbing can get stuck in such local minima.

Advantages:

- **Very low memory usage:** Only the current state is stored in memory.
- **Fast execution time:** For example, in the 8-Queens problem, the algorithm typically completes in 4 steps when it finds a solution and in 3 steps when it gets stuck.

Main drawback:

- Hill-climbing has a **low success rate** because the search stops whenever the heuristic value cannot be immediately improved.

- For the 8-Queens problem, the basic hill-climbing algorithm succeeds only about 14% of the time.

- *Note:* This basic hill-climbing algorithm is incomplete—it often fails to find a goal even if one exists, as it can get trapped in local maxima.

Possible improvements:

1. **Allow moves to successors with the same heuristic value as the current state:**

   - A limit must be imposed on the number of consecutive non-improving moves to avoid infinite loops.
   - For the 8-Queens problem, allowing up to 100 consecutive moves that do not improve the heuristic increases the success rate from 14% to 94%.

2. **Introduce randomness (stochastic hill-climbing):**

   - Instead of always selecting the best neighbor, choose a neighbor probabilistically based on its heuristic value.

3. **Restart upon getting stuck:**

   - If the search becomes blocked, restart hill-climbing from a randomly chosen state and continue until a goal state is found.
   - This simple yet effective technique can solve extremely large problems, such as the 3-million-queens problem, in less than one minute on a standard desktop computer.

Remark:

Thanks to their **very low space complexity**, local search algorithms can be applied to very large problems that cannot be solved efficiently with classical systematic algorithms.

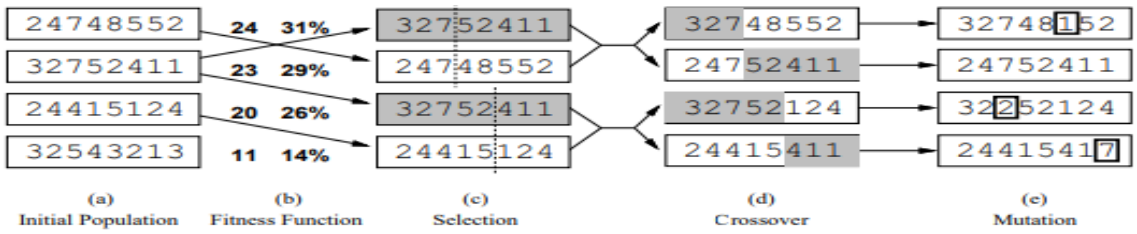


Figure 3.14: The genetic algorithm, illustrated for digit strings representing the states of the 8-queens problem.

Genetic Algorithms

Algorithm 15: Genetic Algorithms (Russell et al., 2010)

```
1  function GENETIC-ALGORITHM(population, FITNESS-FN) returns an individual:
2      repeat
3          new_population ← empty set ;
4          for i ← 1 to SIZE(population) do
5              x ← RANDOM-SELECTION(population,FITNESS-FN) ;
6              y ← RANDOM-SELECTION(population,FITNESS-FN) ;
7              child ← REPRODUCE(x,y) ;
8              if small random probability then child ← MUTATE(child) ;
9              add child to new_population ;
10         end
11         population ← new_population ;
12     until some individual is fit enough, or enough time has elapsed;
13     return the best individual in population, according to FITNESS-FN
14 function REPRODUCE(x , y) returns an individual:
15     n ← LENGTH(x) ; c ← random number from 1 to n ;
16     return APPEND(SUBSTRING(x,1,c),SUBSTRING(y,c + 1,n))
```

Genetic algorithms (GAs) (Russell et al., 2010; Chambers, 2000), see Algorithm 15, are stochastic greedy local search techniques that maintain a large population of states. Unlike classical local search, successor states are generated by **combining two parent states** rather than modifying a single state. A GA begins with a population of k randomly generated states. Each state, or individual, is represented as a string over a finite alphabet, most commonly a binary string.

For example, a state of the 8-Queens problem must specify the positions of 8 queens placed on an 8×8 chessboard. This representation requires $8 \times \log_2 8 = 24$ bits if encoded in binary. Alternatively, an individual can be represented directly as a sequence of eight digits, each between 1 and 8. A sample population of four such individuals is illustrated in Figure 3.14(a).

The generation of the next population is illustrated in Figures 3.14(b)–(e). In (b), each individual is evaluated using a **fitness function**. The fitness should assign higher values to better states. In the 8-Queens problem, a natural fitness measure is the number of pairs of queens that do not attack each other. A correct solution has a value of 28. In the example, the four individuals have fitness values 24, 23, 20 and 11. In this variant of the genetic algorithm, each individual is selected for reproduction with probability proportional to its fitness score.

In (c), two pairs of parents are randomly selected according to these probabilities. An

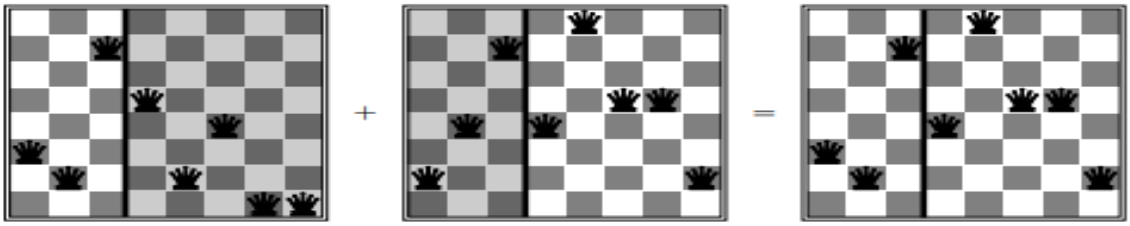


Figure 3.15: The 8-queens states corresponding to the first two parents in Figure 3.14(c) and to the first offspring in Figure 3.14(d). The shaded columns are discarded during the crossover step, while the unshaded columns are preserved.

individual may be selected more than once, while another may not be selected at all. For each selected pair, a **crossover point** is chosen at random among the positions in the representation. The two parent strings are then crossed at this point to produce offspring, as shown in Figure 3.15. In the example, the crossover points are after the third digit for the first pair and after the fifth digit for the second pair.

Finally, in (e), each position in each offspring is subjected to a **random mutation** with a small independent probability. In the 8-Queens problem, a mutation corresponds to selecting a queen at random and moving it to a random row in its column. In the illustration, one digit is mutated in the first, third, and fourth offspring.

Genetic algorithms therefore combine selection, crossover, and mutation to iteratively construct new populations, progressively improving the quality of the solutions.

3.7 Problem-solving by decomposition

Sometimes, problems only appear difficult to solve. In fact, a difficult problem can often be reduced to a set of simpler problems. Once each of these simpler problems is solved, the original complex problem is considered solved. This is the basic intuition behind the **problem decomposition** (or reduction) method.

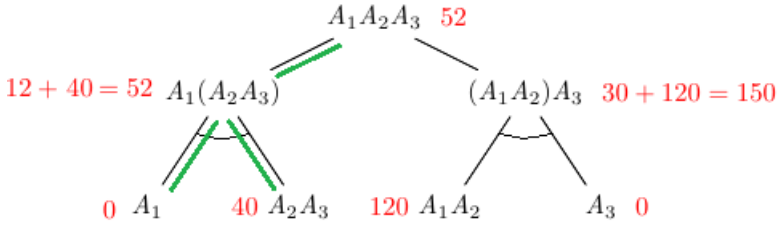
For instance, if one is searching for a sequence of actions to reach a goal, one approach is to use **state-space search**, where each node in the search space represents a state of the world, and the task is to find a sequence of actions that leads from the initial state to the goal state.

Another approach is to consider the different ways in which the goal state can be **decomposed** into simpler subgoals (Yang, 2012). By solving each subgoal individually, the overall problem can be addressed in a structured and manageable way.

A **problem decomposition (or reduction) search** consists of planning the best method to solve a problem that can be recursively decomposed into subproblems in multiple ways. This approach is applicable to a variety of problems, such as:

- Matrix multiplication problems
- The Towers of Hanoi
- The Blocks World problem
- Theorem proving
- And many others

Example (matrix multiplication problem)



$A_1 = 3 \times 4$
$A_2 = 4 \times 10$
$A_3 = 10 \times 1$

3.7.1 OR and AND-OR Graphs in Problem Solving

The state-space search techniques described above can all be represented using a tree where each successor node represents an alternative action to take. The search structure is called an **OR-graph**. In an OR-graph, we aim to find a unique path to a goal. This is because we will know how to move from a node to a target state if we can discover how to reach the target state along one of the branches leaving that node. To represent problem reduction techniques, we must use an **AND-OR graph/tree**. The problem reduction technique using the AND-OR graph is useful for representing a solution to a problem that can be solved by breaking it down into a set of smaller problems, all of which must be solved.

3.7.2 AND-OR Graphs

An **AND-OR graph** is a graphical structure used to represent problem-solving strategies where a problem can be decomposed into subproblems in different ways. In such a graph, there are two types of nodes:

- **OR nodes**, where achieving any one of the successor nodes is sufficient. These represent alternative ways to solve a subproblem.
- **AND nodes**, where all successor nodes must be achieved for the node to be considered resolved. These represent subproblems that must all be solved to reach a solution.

Decomposition or reduction of a problem generates **AND arcs**, which may point to multiple successor nodes. All these successors must be solved for the arc to lead to a solution. Similar to OR graphs, multiple arcs can emerge from a single node, indicating the variety of ways the original problem can be addressed.

This combination of AND and OR nodes is why the structure is called an **AND-OR graph** (sometimes also referred to as an **AND-OR tree**).

The AND-OR graph provides a clear way to represent two scenarios:

1. Where all subgoals must be satisfied to achieve the main goal (AND nodes).
2. Where there exist alternative subgoals, any of which could lead to the main goal (OR nodes).

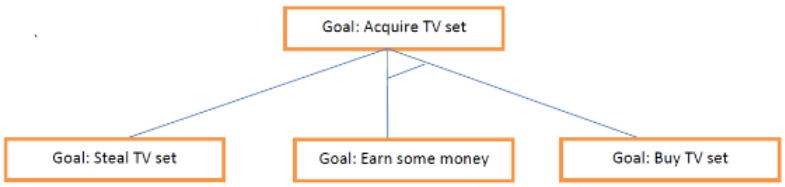


Figure 3.16: A simple AND-OR graph.

Such graphs are particularly useful in problem-solving techniques that rely on **problem decomposition**, as they explicitly model both necessary sequences of actions and alternative strategies.

The **search problem in an AND-OR graph** can be formally defined as follows:

Definition

Given the tuple $[G, s, T, h]$, where:

- G[4] : the specified AND-OR graph.
- s : the initial node of the AND-OR graph.
- T : the set of terminal nodes.
- $h(n)$: a heuristic function estimating the cost of solving the subproblem at node n.

The goal is to find a **solution tree with minimum cost**.

3.7.3 Algorithms for AND-OR Graph Search

Searching for a solution in an AND-OR graph requires careful handling of the AND nodes. A standard A* algorithm cannot efficiently search AND-OR graphs because it does not account for the combined costs of AND subgoals.

Consider the example in Figure 3.17. The top node A is expanded, producing two branches: one leading to B, and the other leading to the subgraph C-D. The numbers associated with each node represent the heuristic estimate h of solving the subproblem at that node.

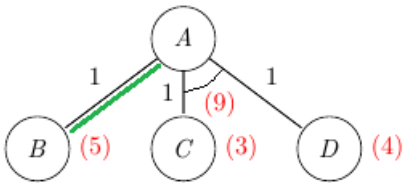


Figure 3.17: Initial expansion in an AND-OR graph.

At first glance, node C seems to be the most promising to expand next since its estimated cost $f = h + g = 3 + 1$ is the lowest. However, using C requires also using D, leading to a total cost of $f(C) = 9$ $(3 + 4 + 1 + 1)$. Expanding B instead yields a lower cost of 6 $(5 + 1)$.

Thus, the selection of the next node to expand depends on whether the node lies along the current best path from the initial node. Figure 3.18 illustrates the updated situation after considering combined costs:

[4] G is a weighted AND-OR graph, where each arc is associated with a cost.

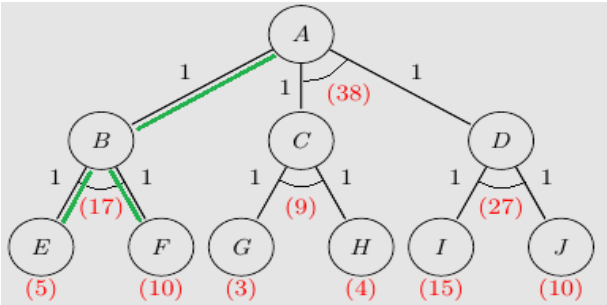


Figure 3.18: Evaluating the best path considering AND node costs.

This example demonstrates why naive A* fails in AND-OR graphs: it cannot correctly evaluate nodes that are part of AND subgoals. Specialized algorithms, such as **AO***, are required to properly handle AND nodes while still using heuristics.

Conclusion

This chapter has presented key concepts in problem solving for artificial intelligence. We defined problems, illustrated examples such as the 8-puzzle and Eight Queens, and classified problems into planning, constraint satisfaction, and multi-agent types.

We examined search methods, from uninformed algorithms like BFS and DFS to heuristic approaches such as Greedy Best-First and A* Search, as well as local search and genetic algorithms.

Problem decomposition using OR and AND-OR graphs was highlighted, showing how complex problems can be structured and solved efficiently with specialized algorithms.

Effective AI problem solving relies on understanding the problem structure and choosing suitable methods, combining search strategies, heuristics, and decomposition to handle diverse challenges.

General Conclusion

This polycopie has presented a comprehensive overview of important concepts in algorithmic complexity and problem-solving, with a progressive structure across three chapters.

We began by laying the theoretical foundation of algorithm complexity, introducing essential tools such as asymptotic analysis, recurrence relations, and algorithm evaluation methods. This gave students the ability to analyze and compare algorithm performance.

In the second chapter, we moved from analyzing algorithms to classifying problems themselves. We studied the distinction between tractable and intractable problems, the notion of undecidability, and the complexity classes P and NP. The framework of NP-completeness was explored through classic problems, deepening our understanding of computational limitations.

The final chapter highlighted problem solving as a core aspect of artificial intelligence. It covered formal problem definitions, search techniques from uninformed to heuristic-based methods, and alternative approaches like local search and genetic algorithms. Special attention was given to problem decomposition using OR and AND-OR graphs showing how complex problems can be structured and solved efficiently.

Taken together, these chapters equip students with the theoretical and practical skills needed to analyze problems, design efficient solutions, and understand when and why certain problems are inherently difficult to solve.

Future Directions. While this polycopie provides a solid foundation, several important topics remain to be explored, including:

- Approximation algorithms for NP-hard problems,
- Probabilistic and randomized algorithms,
- Advanced AI search methods,
- Parameterized complexity and fixed-parameter tractability (FPT),
- Complexity in distributed and parallel computing environments.

These topics represent the next steps for students who wish to deepen their expertise in computational complexity and advanced problem-solving, particularly in domains where classical algorithmic approaches are insufficient.

References

References

Adam, E. (2008). Intelligence Artificielle - Résolution de Problèmes. Retrieved from http://emmanuel.adam.free.fr/site/IMG/pdf/resolution.pdf

Aho, A. V., & Hopcroft, J. E. (1974). The design and analysis of computer algorithms. Pearson Education India.

Alliot, J.-M., Schiex, T., Brisset, P., & Garcia, F. (1994). Intelligence artificielle et informatique théorique. Cépaduès-éd.

Ao* algorithm. (2020). Vidéo en ligne. Retrieved from https://www.youtube.com/watch?v=bA2LJ2MyxTY

Arora, S., & Barak, B. (2009). Computational complexity: a modern approach. Cambridge University Press.

Atallah, M. J., & Blanton, M. (2009). Algorithms and theory of computation handbook, volume 2: special topics and techniques. CRC press.

Baase, S. (2009). Computer algorithms: introduction to design and analysis. Pearson Education India.

Bari, A. (2018). Masters theorem in algorithms for dividing function. Vidéo en ligne. Retrieved from https://www.youtube.com/watch?v=OynWkEjOS-s

Belaïd, A. (n.d.-a). Algorithmes de recherche aveugle. Université de Nancy 2. Retrieved from http://jul.andre.free.fr/Cavaliers/cours2_RechAveugle.pdf

Belaïd, A. (n.d.-b). Algorithmes de recherche heuristique. Université de Nancy 2. Retrieved from http://jul.andre.free.fr/Cavaliers/cours3_RechHeuristique.pdf

Bienvenu, M. (2006). Introduction to artificial intelligence. Université de Provence, Marseille, France. Retrieved from https://www.labri.fr/perso/meghyn/teaching.html

Bourahla, M. (2020). Chapitre 2 : L'IA et la résolution des problèmes. Université de M'Sila. Retrieved from https://elearning.univ-msila.dz/moodle/pluginfile.php/371253/mod_resource/content/1/_IAPA_papier_chapitre_2.pdf

Boyer, M. (n.d.). RELATIONS DE RÉCURRENCE. Retrieved from http://www.iro.umontreal.ca/~dift1063/cours16-17.pdf

Castaneda, H., Chaput, R., Guin, N., & Lefevre, M. (2021/2022). TD1 – Résolution de problèmes à l'aide de graphes d'états.

Chambers, L. D. (2000). The practical handbook of genetic algorithms: applications. Chapman and Hall/CRC.

Cohen, J. (n.d.). GRAPHES ET COMPLEXITE. LRI-CNRS. Retrieved from https://www.lri.fr/~jcohen/documents/enseignement/Cours-glouton.pdf

Cohen, J. (2018). Exercices corrigés sur problèmes np-complets. Retrieved from https://www.lri.fr/~jcohen/documents/enseignement/exercices-NP.pdf

Complexité d'un algorithme. (2021/2022). Lycée Michelet - classes de PCSI. Retrieved from https://cahier-de-prepa.fr/psi-michelet/download?id=239

Cook, S. A. (1971). The complexity of theorem-proving procedures. In Proceedings of the third annual acm symposium on theory of computing (pp. 151–158).

Cori, H. G. K. C., R., & Steyaert, J.-M. (2010). Conception et analyse d'algorithmes. Cours de l'Ecole Polytechnique.

Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2022). Introduction to algorithms. MIT press.

Doty, D. (n.d.). Who is the hardest. one of all? np-completeness. Retrieved from https://canvas.ucdavis.edu/courses/299448/files/5227263/download?wrap=1

Edelkamp, S., & Schrödl, S. (2011). Heuristic search: theory and applications. Elsevier.

Espinasse, B. (2004). Résolution de problemes et Intelligence Artificielle. Université d'Aix-Marseille.

Farzan, A., & Kincaid, Z. (2015). Compositional recurrence analysis. In 2015 formal methods in computer-aided design (fmcad) (pp. 57–64).

Fumera, G. (2019/2020a). Artificial Intelligence - MSc Programs in: Computer Engineering, Cybersecurity and Artificial Intelligence Electronic Engineering Academic. Department of Electrical and Electronic Engineering, University of Cagliari.

Fumera, G. (2019/2020b). Exercises on search algorithms. University of Cagliari. Retrieved from https://unica.it/unica/protected/398005/0/def/ref/MAT238987/

Garey, M. R., & Johnson, D. S. (1979). Computers and intractability. W. H. Freeman and Co.

Gaudel, M.-C., Soria, M., & Froidevaux, C. (1987). Types de données et algorithmes: Recherche, tri, algorithmes sur les graphes. INRIA.

Geurts, P. (2013/2014). Introduction à la théorie de l'informatique. Université de Liège, Institut Montefiore. Retrieved from https://people.montefiore.uliege.be/geurts/Cours/iti/2013/cours-complet-2013-2014.pdf

Ghallab, M., Nau, D., & Traverso, P. (2004). Automated planning: theory and practice. Elsevier.

Goldreich, O. (2008). Computational complexity: a conceptual perspective. ACM Sigact News, 39(3), 35–39.

Goldreich, O. (2010). P, np, and np-completeness: The basics of computational complexity. Cambridge University Press.

Goodrich, M. (2004). Solving Recurrences.

Greef, A., & Reinecke, R. (1988). Problem solving using artificial intelligence techniques. ORiON, 4(1).

Habermehl, P. (2005/2006). Algorithmes de recherche informés. Retrieved from https://www.irif.fr/~haberm/cours/ia05/cours3.pdf

Hadi, A. S. (n.d.). Problem Reduction. Retrieved from https://www.uobabylon.edu.iq/eprints/publication_12_357_213.pdf

Hartmanis, J. (1982). Computers and intractability: a guide to the theory of np-completeness (michael r. garey and david s. johnson). Siam Review, 24(1), 90.

Hazarika, S. (n.d.). Searching and-or graphs. Department of Mechanical Engineering, Indian Institute of Technology - Guwahati.

Hermawanto, D. (2013). Genetic algorithm for solving simple mathematical equality problem. arXiv preprint arXiv:1308.4675. Retrieved from https://arxiv.org/ftp/arxiv/papers/1308/1308.4675.pdf

Karp, R. M. (1972). Reducibility among combinatorial problems. In Complexity of computer computations (pp. 85–103). Springer.

Knuth, D. E. (1997). The art of computer programming (Vol. 3). Pearson Education.

Kozen, D. C. (2006). Theory of computation. Springer.

Lacomme, P. P., Prins, C., & Sevaux, M. (2003). Algorithmes de Graphes. Eyrolles. Retrieved from https://hal.archives-ouvertes.fr/hal-00009111 (ISBN 2-212-11385-4)

Lebbah, Y. (2021). Polycopié : Algorithmique avancée et complexité. Département Informatique, Faculté des Sciences Exactes et Appliquées, Université d'Oran 1. Retrieved from https://sites.google.com/site/ylebbah/teach#h.jsfrxtszqi7d

Lefevre, M. (2021). CM2 : Résolution de problèmes. Université Claude Bernard Lyon 1. Retrieved from https://perso.liris.cnrs.fr/marie.lefevre/ens/BIA/BIA2022-CM2-RP.pdf

Luger, G. F. (2005). Artificial intelligence: structures and strategies for complex problem solving. Pearson education.

Muscholl, A. (2021). Complexité et calculabilité. Département ST, Université Bordeaux. Retrieved from https://www.labri.fr/perso/anca/MC/poly.pdf

Nilsson, N. J. (2014). Principles of artificial intelligence. Morgan Kaufmann.

Ortiz, L. E. (2005/2006a). Practice problems on search in games. MIT EECS. Retrieved from http://www-personal.umd.umich.edu/~leortiz/teaching/6.034f/Fall06/games/games_probs.pdf

Ortiz, L. E. (2005/2006b). Solutions to practice problems on search in games. MIT EECS. Retrieved from http://www-personal.umd.umich.edu/~leortiz/teaching/6.034f/Fall06/games/games_probs_sol.pdf

Oussous, N.-E., & Wegrzynowski, E. (2008). Complexité des algorithmes (1).

Panneerselvam, R. (2015). Design and analysis of algorithms. PHI Learning Pvt. Ltd.

Papadimitriou, C. H. (1994). Computational complexity. Addison-Wesley.

Papadimitriou, C. H., & Steiglitz, K. (1998). Combinatorial optimization: algorithms and complexity. Courier Corporation.

Pearl, J. (1984). Heuristics: intelligent search strategies for computer problem solving. Addison-Wesley Longman Publishing Co., Inc.

Perifel, S. (n.d.). COMPLEXITÉ ALGORITHMIQUE. Retrieved from https://www.irif.fr/~sperifel/complexite.pdf

Russell, S. (1992). Efficient memory-bounded search methods. In Ecai (Vol. 92, pp. 1–5).

Russell, S., & Norvig, P. (2003). Instructor's manual: Exercise solutions for artificial intelligence a modern approach second edition. Alan R. Apt.

Russell, S., Russell, S., Norvig, P., & Davis, E. (2010). Artificial intelligence: A modern approach. Prentice Hall. Retrieved from https://books.google.dz/books?id=8jZBksh-bUMC

Réductions, problèmes p, np-complétude. (Mars 2017). University of Lyon 1.

Schwarzentruber, F. (n.d.). Algorithmes sur les nombres.

Stützle, T. (1999). Local search algorithms for combinatorial problems: analysis, improvements, and new applications.

Talbi, E.-G. (2009). Metaheuristics: from design to implementation. John Wiley & Sons.

Tygert, M. (2010). Recurrence relations and fast algorithms. Applied and Computational Harmonic Analysis, 28(1), 121–128.

Vivien, F. (2002). Algorithmique avancée. IUP 2. Retrieved from http://perso.ens-lyon.fr/frederic.vivien/Enseignement/Algo-2001-2002/Cours.pdf

Wegener, I. (2005). Complexity theory: exploring the limits of efficient algorithms. Springer Science & Business Media.

Wilf, H. S. (2002). Algorithms and complexity. AK Peters/CRC Press.

Yang, Q. (2012). Intelligent planning: a decomposition and abstraction based approach. Springer Science & Business Media.

Zampieri, K., Riviere, S., & Amerein-Soltner, B. (2018). Complexité des algorithmes - Algorithmique. Retrieved from https://ressources.unisciel.fr/algoprog/s34plexite/emodules/cx00macours1/co/cx00macours1.html