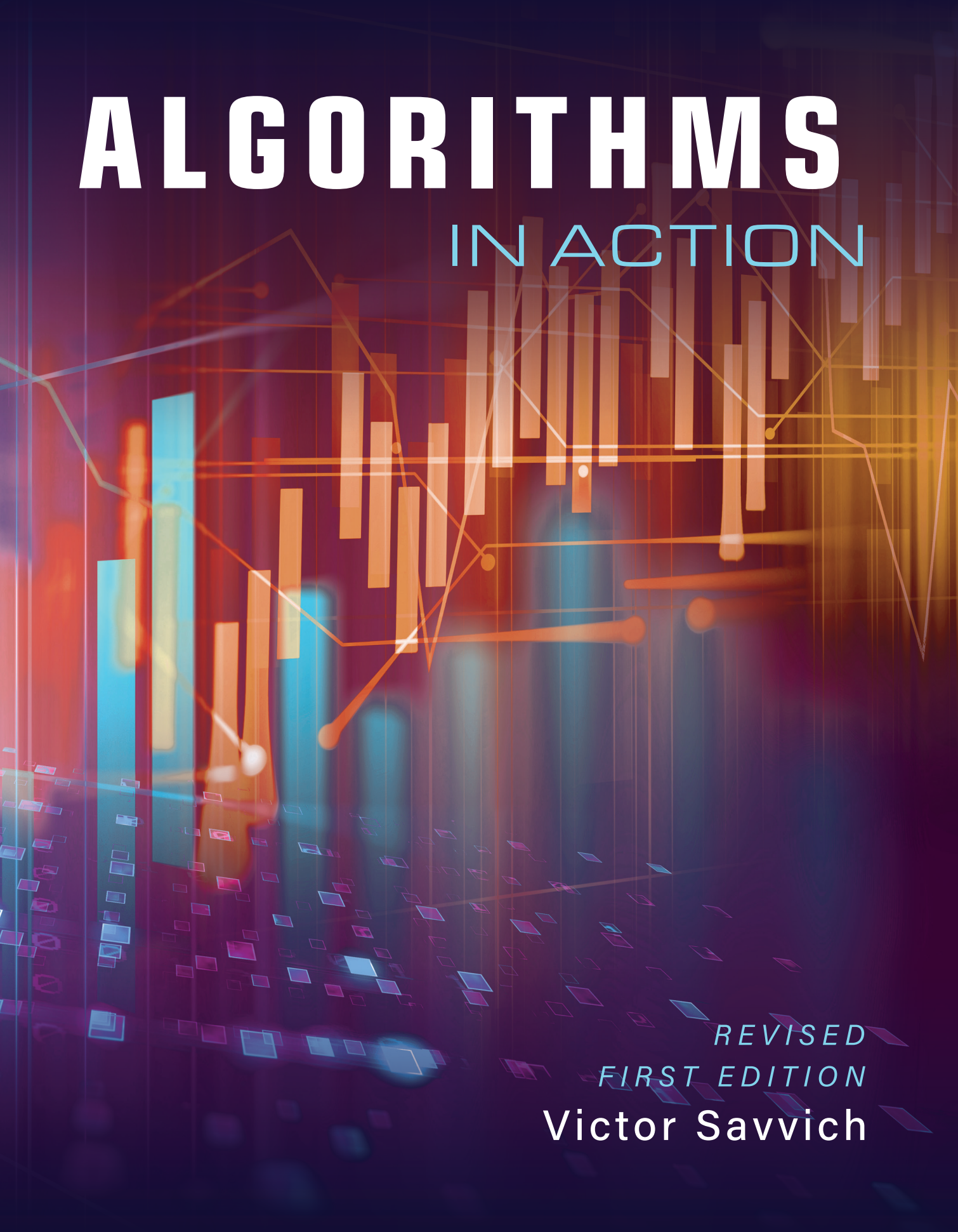


The background is a dark blue gradient with abstract data visualizations. It features a 3D bar chart with orange and blue bars, a line graph with orange lines and dots, and a grid of small, tilted squares in blue and purple at the bottom. The overall aesthetic is high-tech and digital.

ALGORITHMS IN ACTION

REVISED
FIRST EDITION
Victor Savvich

ALGORITHMS IN ACTION

Revised First Edition

Victor Savvich

Univeristy of Southern California

Bassim Hamadeh, CEO and Publisher
Mieka Portier, Senior Acquisitions Editor
Carrie Baarns, Manager, Revisions and Author Care
Kaela Martin, Project Editor
Rachel Kahn, Production Editor
Jess Estrella, Senior Graphic Designer
Natalie Piccotti, Director of Marketing
Kassie Graves, Senior Vice President, Editorial
Jamie Giganti, Director of Academic Publishing

Copyright © 2023 by Cognella, Inc. All rights reserved. No part of this publication may be reprinted, reproduced, transmitted, or utilized in any form or by any electronic, mechanical, or other means, now known or hereafter invented, including photocopying, microfilming, and recording, or in any information retrieval system without the written permission of Cognella, Inc. For inquiries regarding permissions, translations, foreign rights, audio rights, and any other forms of reproduction, please contact the Cognella Licensing Department at rights@cognella.com.

Trademark Notice: Product or corporate names may be trademarks or registered trademarks and are used only for identification and explanation without intent to infringe.

Cover image copyright © 2018 iStockphoto LP/monsitj.

Printed in the United States of America.

Contents

Preface vii

Chapter 1 Review 1

- 1.1 Runtime Complexity 1
- 1.2 Lower Bound for Sorting 3
- 1.3 Trees and Graphs 4

Chapter 2 Amortized Analysis 15

- 2.1 Unbounded Array 15
- 2.2 Binary Counter 18
- 2.3 Amortized Dictionary 19
- 2.4 Amortized Trees 20

Chapter 3 Heaps 25

- 3.1 Binary Heaps 25
- 3.2 Binomial Heaps 33
- 3.3 Fibonacci Heaps 40

Chapter 4 Greedy Algorithms 45

- 4.1 The Money Changing Problem 46
- 4.2 Scheduling Problem 47
- 4.3 Huffman Code 49
- 4.4 Minimum Spanning Trees 56
- 4.5 Shortest Path Problem 62

Chapter 5 Divide-and-Conquer Algorithms 69

- 5.1 Solving Divide-and-Conquer Recurrences 71
- 5.2 Integer Multiplication 76
- 5.3 Matrix Multiplication 78
- 5.4 The Maximum Subsequence Sum Problem 80
- 5.5 Computing Fibonacci Numbers 81

Chapter 6 Dynamic Programming 87

- 6.1 Introduction 87
- 6.2 Knapsack Problem 89
- 6.3 Static Optimal Binary Search Tree 94
- 6.4 The Bellman-Ford Algorithm 98
- 6.5 The Shortest Path in DAGs 101

Chapter 7 Network Flow 107

- 7.1 Introduction 107
- 7.2 The Ford–Fulkerson Algorithm 110
- 7.3 Reduction to Network Flow 116
- 7.4 Augmenting Path Heuristics 120
- 7.5 The Circulation Problem 123
- 7.6 Reduction to Circulation 130

Chapter 8 Linear Programming 137

- 8.1 Introduction: A Production Problem 137
- 8.2 The Standard Maximum Problem 141
- 8.3 A Few Applications 144
- 8.4 The Dual Linear Program 148

Chapter 9 NP Completeness 157

- 9.1 A Brief Introduction to the Turing Machines 157
- 9.2 Computational Intractability 159
- 9.3 NP-Complete Problems 163

Preface

THIS BOOK PROVIDES AN INTRODUCTION TO algorithm design, intended to teach a variety of techniques for designing algorithms, with a focus on an intuitive understanding of algorithms. The book trains you on many basic algorithms, so after reading this book you will have a foundational knowledge of algorithms that will enable you to take more advance courses (like machine learning, and artificial intelligence) and eventually apply them in your professional work. The book does not intend to be comprehensive nor complete.

The genesis of this book came about through the lecture notes I developed while teaching undergraduate and graduate computer science courses at Carnegie Mellon University and the University of Southern California. The book covers roughly a semester's worth of coursework, though some chapters go far beyond the standard lecture material, letting students dive deeper into the concepts and thus providing material to stimulate further thought and discussion. The book is intended for advanced undergraduate or master-level students in computer science and/or related technical disciplines. A foundation of undergraduate coursework in discrete mathematics, data structures, and calculus is highly recommended as a prerequisite. The book does not emphasize or require programming, just pseudocode to encourage readers on conceptual understanding.

Analysis of algorithms builds upon the mathematics knowledge to express concepts that seem intuitively obvious but are surprisingly difficult to comprehend. Analyzing algorithms is a challenging topic for the most students, as they have not yet developed an experience in algorithmic problem solving. Students often come up with an erroneous intuitive solution, demonstrating their overconfidence in understanding the material. My approach is to explain how to design algorithms, focusing on providing fundamental mathematical concepts, with a detailed, visual step-by-step algorithm execution.

The book contains over 160 figures that help the reader to visualize the process. While I provide proofs of algorithm correctness, my goal is not to overwhelm the reader with rigorous mathematical proofs.

The purpose of the book is to give you a collection of analytical tools that you can use when approaching a problem. In each chapter, I present a different algorithm technique that is successfully used in practice, and some common scenarios that could be used to classify those problems. Once you master the most common algorithms, you will develop a systematic way of thinking through problems and develop algorithmic thinking. The latter is increasingly prevalent not only in computer science but in disciplines like physics, biology, and economics.

My hope with this book is to offer a reader-friendly approach to algorithms and algorithmic thinking, with the review questions and exercises at the end of each chapter (122 short review questions and 129 exercises in total) allowing readers to practice and apply the concepts taught. Reading this book may help you to land a job or an internship, since large technology companies want their software developers to apply algorithmic thinking. Very often the interviewers do not ask for an exact answer, but rather want to see how you approach and problem-solve an unfamiliar problem.

Victor Savvich
Playa Vista, California

Chapter 1

Review

IN THIS CHAPTER WE REVIEW BASIC concepts, from asymptotic complexity, graph theory, and mathematical proof techniques, as they are required for better understanding for the chapters that follow. If the reader has some previous acquaintance with these topics, the chapter should be enough to get started. If the reader has no previous background in these, we suggest a more thorough introduction such as *Mathematics for Computer Science* by Eric Lehman, Thomson Leighton and Albert Meyer.¹

1.1 Runtime Complexity

The term *analysis of algorithms* is used to describe approaches to study the performance of algorithms. With each algorithm we associate a sequence of steps comprising this algorithm. We measure the run time of an algorithm by counting the number of steps and therefore define an algorithmic complexity as a numerical function $T(n)$, where n is the input size. Consider a problem of addition of two n -bit binary numbers. Let $T(n)$ represent an amount of time addition used to add two n -bit numbers. We want to define “time” $T(n)$ taken by the method of addition without having to worry about implementation details. The process of addition consists of the following two steps:

- Adding 3 bits (one bit is a carry bit)
- Writing down 2 bits (again, one bit is a carry bit)

¹ Eric Lehman, Thomson Leighton, and Albert Meyer, *Mathematics for Computer Science*, (Great Britain: Samurai Media Limited, 2017).

On any computer, adding and writing two bits can be done in constant time. By constant time we mean that the time is independent of the input size. Therefore, the total time of addition of two n -bit binary numbers is $T(n) = n \cdot c$, where the constant c can be different on different computers. We say that bit addition is a linear time algorithm. The process of abstracting away details and determining the rate of resource usage in terms of the input size is one of the fundamental ideas in computer science. In this course we will perform the following types of analysis:

1. The worst-case complexity (the maximum number of steps taken on any input)
2. The best-case complexity (the minimum number of steps taken on any input)
3. The average case complexity (the average number of steps taken on a random input)
4. The amortized time complexity (the average complexity over a sequence of operations)

We measure the runtime of an algorithm using following asymptotic notations: O , Ω , Θ .

1.1.1 Upper Bound (Big-O)

For any monotonic functions, f, g from the positive integers to the positive integers, we say $f(n) = O(g(n))$ (or $f(n) \in O(g(n))$) if $g(n)$ eventually dominates $f(n)$. Figure 1.1 helps you to visualize this relationship.

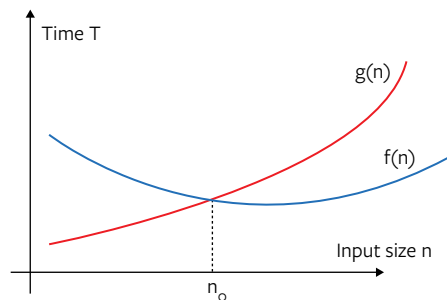


FIGURE 1.1 $f(n) = O(g(n))$.

Formally, there exists a positive real number, $c > 0$, and a real number, n_0 , such that $f(n) \leq c \cdot g(n)$ for all $n \geq n_0$.

Example: $n^2 + 2n + 1 = O(n^2)$. Since $n^2 + 2n + 1 \leq n^2 + 2n^2 + n^2 = 4n^2$ for $n \geq 1$, we choose $c = 4$ and $n_0 = 1$.

1.1.2 Lower Bound (Big-Omega)

For any monotonic functions, f, g from the positive integers to the positive integers, we say $f(n) = \Omega(g(n))$ (or $f(n) \in \Omega(g(n))$) if $f(n)$ eventually dominates $g(n)$.

Formally, there exists a positive real number, $c > 0$, and a real number, n_0 , such that $f(n) \geq c \cdot g(n)$ for all $n \geq n_0$.

Example: $n^2 + 2n + 1 = \Omega(n^2)$. Since $n^2 + 2n + 1 \geq n^2$ for $n \geq 1$, we choose $c = 1$ and $n_0 = 1$.

1.1.3 Exact Bound (Big-Theta)

For any monotonic functions, f, g from the positive integers to the positive integers, we say $f(n) = \Theta(g(n))$ (or $f(n) \in \Theta(g(n))$) if $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$. Formally, there exists positive real numbers, c_1 and c_2 , and a real number, n_0 , such that $c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n)$ for all $n \geq n_0$.

Example: $n^2 + 2n + 1 = \Theta(n^2)$.

1.2 Lower Bound for Sorting

We will show here that any deterministic comparison-based sorting algorithm must take $\Omega(n \log n)$ time to sort an array of n elements in the worst case. Comparison-based sorting algorithms operate on the input by comparing pairs of elements. For example, Mergesort and insertion sort are comparison-based sorting algorithms. But bucket sort and radix sort are not. In order to show $\Omega(n \log n)$ bound we will play the Guess-a-Number game. The computer will select a number, x , between 1 and 10, and you need to determine x by asking questions. You'll keep guessing numbers until you find x . The guessing game can be viewed abstractly as a binary search tree (also called a decision tree). Figure 1.2 shows that we can guess any number between 1 and 10 by asking at most four questions.

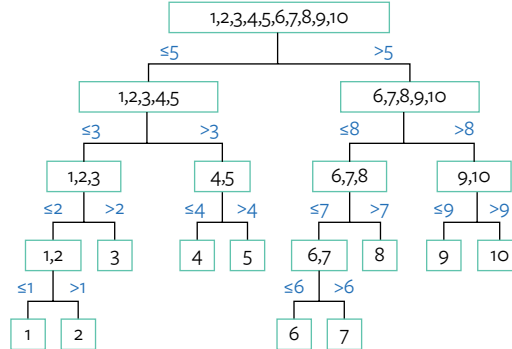


FIGURE 1.2 A binary search tree to guess a number between 1 and 10.

We will be using a decision tree to model the execution of any comparison-based sort. The execution of the sorting algorithm corresponds to tracing a path from the root of the decision tree to a leaf. At each internal node we make a comparison; based on that comparison we proceed further down to either the left or right subtree. Figure 1.3 depicts sorting an array of three elements $[a, b, c]$. In that tree each leaf represents a permutation of $[a, b, c]$. Generally, for sorting an array of n elements, each leaf in the

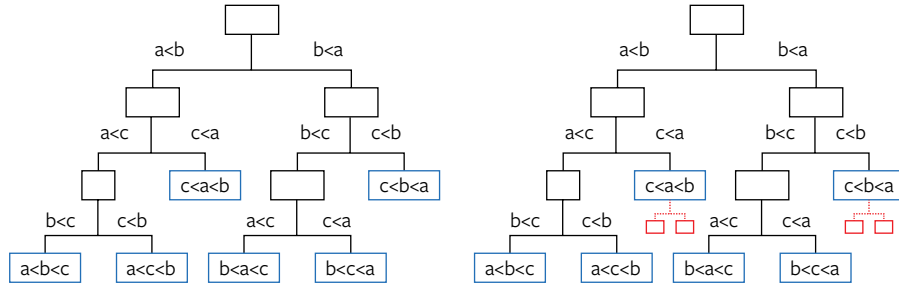


FIGURE 1.3 A binary search tree to guess a number between 1 and 10.

decision tree represents a permutation of the n elements. Hence, there are $n!$ leaves in the tree. Let us denote the tree by B . Next, we make the tree complete by adding extra nodes, as seen in figure 1.3. We will denote the new tree by B^* and its height by h . Since tree B^* is a complete binary tree, it has 2^h leaves. By construction, B^* has more leaves than B . It follows, $2^h \geq n!$, or, after taking the log of both parts, $h \geq \log(n!)$.

The height of the decision tree is the number of comparisons in a sorting algorithm; in other words, h is the runtime complexity $T(n)$ of sorting. Thus, $T(n) \geq \log(n!)$. Lastly, we simplify $\log(n!)$ as follows:

$$\begin{aligned} \log(n!) &= \log(n(n-1)(n-2) \dots 1) \\ &\geq \log(n(n-1) \dots (n-n/2)) \\ &\geq \log((n/2)^{n/2}) = \Omega(n \log n) \end{aligned}$$

We have proved that any comparison-based sorting algorithm needs $\Omega(n \log n)$ comparisons. This holds even for a quantum computer!

1.3 Trees and Graphs

A graph G is a pair (V, E) , where V is a set of vertices (or nodes) and E is a set of edges connecting the vertices. A self-loop is an edge that connects to the same vertex twice. A multi-edge is a set of two or more edges that have the same two vertices. A graph is simple if it has no multi-edges or self-loops. We always assume simple graphs unless otherwise noted. Graphs could be directed and undirected and weighted and unweighted (weights will usually be edge weights).

Theorem. *Let G be a graph with V vertices and E edges. The following statements are equivalent:*

1. G is a tree.
2. Every two vertices of G are connected by a unique path.
3. G is connected and $V = E + 1$.

4. G is acyclic and $V = E + 1$.
5. G is acyclic and if any two non-adjacent vertices are joined by an edge, the resulting graph has exactly one cycle.

To prove this, it suffices to show $1 \Rightarrow 2 \Rightarrow 3 \Rightarrow 4 \Rightarrow 5 \Rightarrow 1$. We'll just show $1 \Rightarrow 2 \Rightarrow 3 \Rightarrow 4$ and leave the rest to the reader.

Proof of $1 \Rightarrow 2$. We prove it by contradiction. Assume G is a tree that has two vertices connected by two different paths like in figure 1.4.

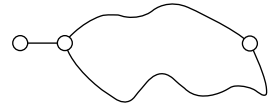


FIGURE 1.4 Two vertices connected by two different paths.

Then there exists a cycle! It follows that G cannot be a tree: a contradiction. ■

Proof of $2 \Rightarrow 3$. Since every two vertices in G are connected by a path, G is a connected graph. We prove that in G the number of nodes and edges are related by $V = E + 1$. The proof is by strong induction on the number of nodes.

Base case: $V = 2$. Since a graph is simple, $E = 1$. Thus, $V = E + 1$.

Inductive hypothesis: Assume $V = E + 1$ for every graph with $V < n$ vertices.

Inductive step: Prove $V = E + 1$ for every graph with $V = n$ vertices.

Graph G has n vertices. We will use notation $V(G) = n$. We choose two adjacent vertices, x and y . We know that every two vertices in G are connected by a unique path. It follows that x and y are joined by an edge, like in figure 1.5.

Note in both subgraphs G_1 and G_2 the number of vertices is less than n . Indeed, G_1 does not contain vertex y , and G_2 does not contain vertex x . We can apply the inductive hypothesis to G_1 and G_2 . It follows,

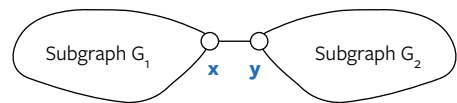


FIGURE 1.5 Graph G consists of two subgraphs G_1 and G_2 .

$$V(G) = V(G_1) + V(G_2) = E(G_1) + 1 + E(G_2) + 1 = E(G) + 1.$$

This concludes the proof. ■

Proof of $3 \Rightarrow 4$. We prove that G is an acyclic graph by contradiction. Assume that G has a cycle with k vertices in it. This cycle also contains k edges. Now let us count edges in

the whole graph. We claim the number of edges in the graph will be at least V . Indeed, there are k edges in the cycle and at least $V - k$ outside the cycle. ■

Theorem. *In an undirected simple graph $G = (V, E)$, there are at most $V(V - 1)/2$ edges. In short, by using the asymptotic notation, $E = O(V^2)$.*

Proof. Choose any vertex in G . The possible number of edges leaving this vertex is $V - 1$. Take another vertex (different from the previous one). The possible number of edges leaving that vertex is $V - 2$ (don't count the edge between two vertices twice!), and so on. We have that the total number of edges is at most

$$(V - 1) + (V - 2) + \dots + 2 + 1 = V(V - 1)/2$$

This concludes the proof. ■

We define a *dense* graph $G = (V, E)$ as a graph in which the number of edges is $E = \Omega(V^2)$. We say that a graph is *sparse* if $E = O(V)$.

1.3.1 Graph Traversals

Graphs traversal means visiting all vertices in a systematic order. We can choose any vertex as a starting point. Then we will systematically enumerate all vertices accessible from it. Because a graph might contain cycles, we need some way for marking a vertex as having been visited. To do so we will keep a Boolean array, with all elements initially set to false. We will set a correspondent element to true as soon as we visit a particular vertex. Also, we need to keep in mind that the graph might be disconnected. There are two most common traversals:

- Depth-first search (DFS)
- Breadth-first search (BFS)

DFS uses a stack data structure for backtracking. BFS uses a FIFO queue for book-keeping. Here is a pseudocode:

```
for all v in V do visited[v] = false
for all v in V do if !visited[v] traversal(v)
traversal(v) {
    visited[v] = true
    for all w in adj(v)
        do if !visited[w] traversal(w)
}
```

The runtime complexity of traversal is $O(V + E)$. There are two important properties of traversal: (1) It visits all the vertices in the connected component; (2) edges labeled by traversal form a spanning tree of the connected component.

1.3.2 Topological Sort

Suppose each vertex represents a task that must be completed and a directed edge (u, v) indicates that task v depends on task u . That is, u must be completed before v . If G is a directed acyclic graph (DAG), then there exists a valid order in which you can complete the tasks. This is called topological order or topological sort. If the graph is cyclic, no topological order exists. Consider the graph in figure 1.6.

The following sequence $\{a, b, c, d, e, f, g, h, i\}$ is a valid topological sort. In other words, a topological order means arranging the vertices along a line so that all edges go from left to right. It should be evident from figure 1.6 that a topological sort is not unique. The following list $\{a, c, b, f, e, d, h, g, i\}$ is another topological order.

The algorithm of finding a topological sort is based on traversal: run DFS (from any vertex) and return a vertex that has no undiscovered leaving edges. In figure 1.6, the possible DFS run may be $a \rightarrow e \rightarrow i$, making i the first vertex with no undiscovered leaving edges. From i , backtrack to e and then go to g . This makes g another vertex with no undiscovered leaving edges. From g , backtrack to e and then go to h . This makes h the third vertex with no undiscovered leaving edges. And so on. The algorithm will produce a topological order in reverse. Note, if we start DFS at any other vertex but a , we will need another run of DFS. The runtime complexity of the algorithm is linear $O(V + E)$.

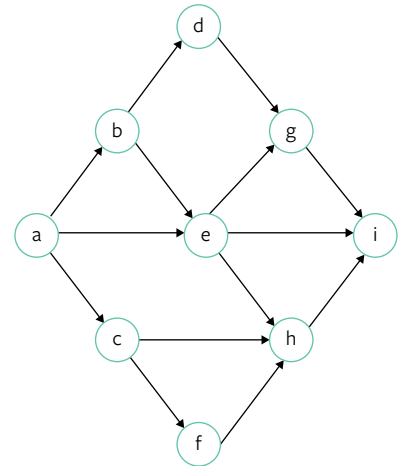


FIGURE 1.6 A directed acyclic graph.

1.3.3 Planar Graphs

A connected graph is planar if it can be drawn in the plane with each edge drawn as a continuous curve such that no two edges cross. There are many examples of planar graphs: any tree is planar, every cycle is planar, a complete graph K_4 is planar.

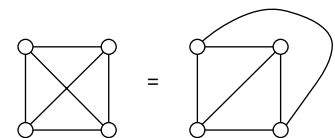


FIGURE 1.7 An example of a planar graph.

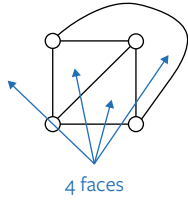


FIGURE 1.8 A planar graph with four faces.

A planar graph in addition to vertices and edges also has disjoint faces.

Theorem. (Euler's formula) *If G is a connected planar graph with V vertices, E edges, and F faces, then*

$$V - E + F = 2.$$

Proof. The proof by induction on the number of edges.

Base case: $E = 1$. The identity holds, since $V = 2$ and $F = 1$.

Inductive hypothesis: Assume it's true for any graph with no more than E edges.

Inductive step: Prove it for graphs with E edges.

Start with a graph G that has E edges and remove one edge. There are two cases to consider:

1. The edge to remove lies on a cycle. See figure 1.9.
2. The edge to remove does not lie on a cycle. See figure 1.10.

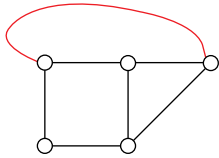


FIGURE 1.9 A graph in case 1.

In case (1) by removing an edge (in red) on a cycle, we obtain a new graph G' with $E - 1$ edges and $F - 1$ faces. Since G' has $E - 1$ edges, the relation works by the induction hypothesis. That is $V - (E - 1) + (F - 1) = 2$. This simplifies to $V - E + F = 2$.

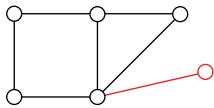


FIGURE 1.10 A graph in case 2.

In case (2) by removing an edge (in red) that does not lie on a cycle, we obtain a new graph G' with $E - 1$ edges and $V - 1$ vertices. Since G' has $E - 1$ edges, the relation works by the induction hypothesis. That is $(V - 1) - (E - 1) + F = 2$. This simplifies to $V - E + F = 2$.

This completes the proof. ■

Theorem. *In any simple connected planar graph with at least 3 vertices, $E \leq 3V - 6$.*

Proof. If a graph has no cycles, then

$$E = V - 1 \leq V \leq V + (2V - 6) = 3V - 6,$$

since $V \geq 3$, and therefore $2V - 6 \geq 0$.

Assume a graph with cycles. We will count the number of pairs (edge, face) (i.e., $\Sigma(\text{edge, face})$). Since each face is bounded by at least 3 edges, then $\Sigma(\text{edge, face}) \geq 3F$. Since each edge is associated with at most 2 faces, then $\Sigma(\text{edge, face}) \leq 2E$. Combining these two inequalities, we find $3F \leq 2E$. But we know from the previous theorem that $V - E + F = 2$. It follows,

$$6 = 3V - 3E + 3F \leq 3V - 3E + 2E = 3V - E.$$

Thus, we conclude $E \leq 3V - 6$. ■

Corollary. *A simple connected planar graph with at least 3 vertices has a vertex of degree 5 or less.*

Proof. We know that in any graph, the sum of the degrees of all vertices is equal to twice the number of edges, $\Sigma \text{degree}(v) = 2E$. From the previous theorem, $2E \leq 6V - 12$. Thus, the average degree is at most 6:

$$\frac{1}{V} \sum_{v \in V} \text{degree}(v) \leq \frac{6V - 12}{V} = 6 - \frac{12}{V}$$

It follows there exists a vertex of degree 5 or less. ■

1.3.4 Coloring Planar Graphs

Given a planar graph, how many colors do you need in order to color the vertices so that no two adjacent vertices get the same color? Back in the 1880s, Francis Guthrie conjectured that four colors suffice. In 1976 K. Appel and W. Haken, using a special-purpose computer program, have proved that conjecture.

We won't prove the conjecture but let us prove the six-color theorem.

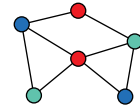


FIGURE 1.11 A graph coloring problem.

Theorem. (6-color theorem) *Every planar graph can be colored with at most six colors.*

Proof. By induction on the number of vertices.

Base case: If a graph has six or less vertices, then the result is obvious.

Inductive hypothesis: Assume that all graphs with $V - 1$ vertices are six-colorable.

Inductive step: Prove it for any graph with V vertices.

By previous corollary any graph G with V vertices has at least one vertex of degree 5 or less. Remove it from G . The remaining graph is planar and by induction can be colored with at most 6 colors. Now insert that vertex back. Since this vertex has at most 5 neighbors then at least one of 6 colors is not used. We color the vertex with one of the unused colors. ■

1.3.5 Bipartite Graphs

A graph is bipartite if the vertices can be partitioned into two disjoint sets, X and Y , such that all edges go only between X and Y (no edges go from X to X or from Y to Y).

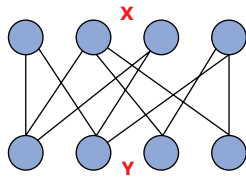


FIGURE 1.12 A bipartite graph.

A complete bipartite graph (denoted by $K_{n,m}$, where n and m are sizes of two sets) is a special kind of *bipartite graph* where every vertex of the first partition is connected to every vertex of the second partition.

Theorem. A graph is bipartite if and only if it does not contain an odd length cycle.

Proof. Note, the length of a cycle is the number of vertices in the cycle.

$\Rightarrow$) In a bipartite graph every cycle has vertices that must alternate between two partitions. Since the number of vertices in such a cycle is even, it is an even cycle.

$\Leftarrow$) Assume connected graph G has no odd cycle. Pick any vertex v . Define two sets of vertices based on parity of distance (even or odd) from v :

$$X = \{u \in V \mid d(v, u) \text{ is even}\}$$

$$Y = \{u \in V \mid d(v, u) \text{ is odd}\}$$

These sets provide a bipartition. If G had an odd cycle, then there will be a vertex present on both sets. Finish the proof for disconnected graph G . ■

One of the famous problems on bipartite graphs is a matching problem. A subset of edges is a *matching* if no two edges have a common vertex. A *maximum matching* is a matching with the largest possible number of edges. Our goal is to find the maximum matching in a graph. We will show in chapter 7 that the problem of finding the maximum matching can be reduced to the maximum flow problem.

1.3.6 Other Famous Graph Problems

A Euler path is a path that uses each edge of a graph exactly once. A Euler cycle is a cycle that uses every edge of a graph exactly once. A graph that contains a Euler cycle is called a *Eulerian graph*.

Theorem. *A connected graph G is a Eulerian graph if and only if all vertices of G are of even degree.*

Proof. $\Rightarrow$) Let $G = (V, E)$ be a Euler graph. Thus, G contains a Euler path. Let us walk that path. Visiting an intermediate vertex in the path contributes two to the degree of that vertex. It follows that every intermediate vertex has an even degree.

$\Leftarrow$) Assume that all vertices of G are of even degree. We construct a cycle starting at an arbitrary vertex v , going through the edges of G only once. Since every vertex is of even degree, we eventually come back to v . If this cycle includes all the edges of G , then G is a Eulerian graph. If not, we remove from G the cycle we have constructed. We will get a connected subgraph G_0 in which all vertices are of even degree. We again construct a new cycle in G_0 . This process is repeated until we obtain a cycle that traces all the edges of G . We showed that G is a collection of cycles, hence G is a Eulerian graph. ■

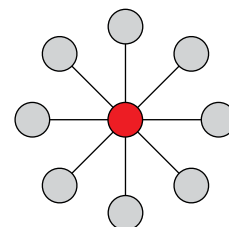


FIGURE 1.13 The vertex cover problem.

A *Hamiltonian path* is a path that visits each vertex of a graph exactly once. A *Hamiltonian cycle* is a cycle that visits every vertex in a graph exactly once (except for the start and end vertices). A graph that contains a Hamiltonian cycle is called a *Hamiltonian graph*. A problem closely related to Hamiltonian cycle is the famous *traveling salesman problem*. Given a weighted graph, find the shortest weighted Hamiltonian cycle. We will prove in chapter 9 that both problems are *unlikely to be solved in polynomial time*. The *traveling salesman problem* is one of the most intensively studied problems in computer science.

A *vertex cover* of an undirected graph is a subset of *vertices* such that for every edge (u, v) either u or v is in a *vertex cover*. The *vertex cover problem* is to find the minimum size *vertex cover*.

Given a graph, we say that a subset of vertices is independent if no two of them are joined by an edge. Given an undirected graph, the *independent set* problem is to find the largest independent set. Vertex cover and independent set are very closely related graph problems; see Exercise 15.

Two graphs, $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$, are *isomorphic* if there is a bijective function $f: V_1 \rightarrow V_2$ such that an edge $(u, v) \in E_1$, if and only if an edge $(f(u), f(v)) \in E_2$. Two isomorphic graphs look differently but are structurally the same, up to the renaming of the vertices.

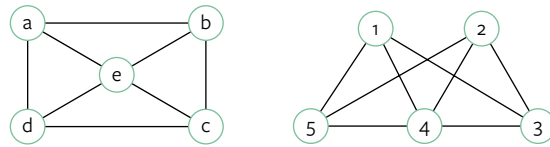


FIGURE 1.14 Two isomorphic graphs.

In figure 1.14 we can match vertices as follows: $a - 1$, $b - 2$, $c - 3$, $d - 5$, $e - 4$. Although we matched vertices in one particular way, there could be several ways to do it. Determining whether two graphs are isomorphic is not an easy task; however, computer scientists believe the problem can be solved in polynomial time.

REVIEW QUESTIONS

1. Mark the following assertions as TRUE or FALSE. No need to provide any justification.
 - a. $n = O(n^2)$
 - b. $n = O(\sqrt{n})$
 - c. $\log n = \Omega(n)$
 - d. $n^2 = \Omega(n \log n)$
 - e. $n^2 \log n = \Theta(n^2)$
 - f. $7 \log^2 n + 2n \log n = \Omega(\log n)$
 - g. $5n \log n + 1024 n \log(\log n) = \Theta(n \log n)$
 - h. $2^n + 100n^2 + n^{100} = O(n^{101})$
 - i. $(1/3)^n + 100 = O(1)$
2. (T/F) Any function which is $\Omega(\log n)$ is also $\Omega(\log(\log n))$.
3. (T/F) If $f(n) = \Theta(g(n))$ then $g(n) = \Theta(f(n))$.

4. (T/F) If $f(n) = \Theta(g(n))$ then $f(n) = \Omega(g(n))$.
5. (T/F) If $f(n) = \Omega(g(n))$ then $2^{f(n)} = \Omega(2^{g(n)})$.
6. (T/F) BFS can be used to find the shortest path between any two nodes in a non-weighted graph.
7. (T/F) A DFS tree is never the same as a BFS tree.
8. (T/F) Algorithm A has a running time of $O(n^2)$ and algorithm B has a running time of $O(n \log n)$. From this we conclude that A can never run faster than B on the same input.
9. (T/F) Planar graph is a sparse graph.
10. (T/F) Every DAG contains a vertex with no incoming edges.

EXERCISES

1. Prove $g(n) = \Omega(f(n))$ if and only if $f(n) = O(g(n))$.
2. Prove or disprove $f(n) = O(g(n))$ implies $2^{f(n)} = O(2^{g(n)})$.
3. Arrange the following functions

$$\log n^n, n^2, n^{\log n}, n \log \log n, 2^{\log n}, \log^2 n, n^{\sqrt{2}}$$

in increasing order of growth rate, with $g(n)$ following $f(n)$ in your list if and only if $f(n) = O(g(n))$.

4. Arrange the following functions

$$4^{\log n}, \sqrt{\log n}, n^{\log \log n}, (\sqrt{2})^{\log n}, 2^{\sqrt{2 \log n}}, n^{1/\log n}, (\log n)!$$

in increasing order of growth rate with $g(n)$ following $f(n)$ in your list if and only if $f(n) = O(g(n))$.

5. What is the Big-O runtime complexity of the following function?

```
void bigOh1 (int n):
    for i=1 to n
        j=1;
        while j < n
            j = j*2;
```

6. What is the Big-O runtime complexity of the following function?

```

void bigOh2 (int n):
    if(n == 0) return "a";
    string str = bigOh2(n-1);
    return str + str;

```

7. What is the Big-Theta runtime complexity of the following function? Here `find_max` finds the maximum element in the array `L[0], L[1], ..., L[n - 1]`.

```

void bigTheta (int[] L, int n):
    while (n > 0)
        find_max(L, n);
        n = n/4

```

8. The complete graph on n vertices, denoted K_n , is a simple graph in which there is an edge between every pair of distinct vertices. What is the height of the DFS tree for the complete graph K_n ? What is the height of the BFS tree for the complete graph K_n ?
9. We are interested in finding a simple path in a directed acyclic graph that visits all vertices once and only once. Design a linear time algorithm to determine if there is such a path in a given DAG.
10. Prove that a complete graph K_5 is not a planar graph.
11. Prove that a complete bipartite graph $K_{3,3}$ is not a planar graph.
12. In a connected bipartite graph, is the bipartition unique? Justify your answer.
13. Given a directed graph $G = (V, E)$ and a particular node $v \in V$, design a linear time algorithm to determine whether v is in a triangle of edges (a cycle of length 3).
14. Design a linear time algorithm which, given an undirected graph $G = (V, E)$ and a particular edge $e \in E$, determines whether G has a cycle containing e .
15. Given an undirected graph $G = (V, E)$, prove that S is an independent set if and only if $V - S$ is a vertex cover.