MATH 409: DISCRETE OPTIMIZATION

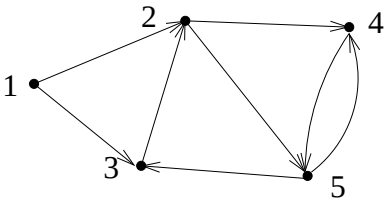
Course Notes (Version: March 2009)

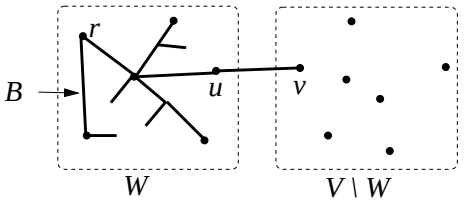
by

Paul Tseng

Department of Mathematics

University of Washington





FOREWORD

As the chief engineer for the fabled kingdom Kopán, you are charged to build roads to link its five principal cities Al, Bo, Ci, Da, Ee. Your engineers estimate that building a road between Al & Bo would take 15000 man-hrs, between Al & Ci would take 11000 man-hrs, etc. Which roads should be built to link all cities in fewest man-hours? [There are clearly more than one way to link the cities: Al–Bo, Bo–Ci, Ci–Da, Da–Ee or Al–Bo, Bo–Ci, Ci–Ee, Ee–Da or ...] Instead of building roads, we can similarly consider laying down power lines or communication cables to link cities/towns. This is called the "minimum spanning tree" (MST) problem, and it is an example of a discrete optimization (also called combinatorial optimization) problem, namely, find from among a *finite* number of choices, one (an "optimal solution") that minimizes/maximizes a certain quantity.

For another example, suppose that the five cities are already joined by roads, and the travel time on each road is known. What is the fastest way to go from, say, Al to Ee? If there is a road between Al and Ee, then that's certainly one path. But it could be that there is also a road between Al and Ci, between Ci and Da, and between Da and Ee, which would be another path. Among all such paths, we wish to find one whose travel time is minimum. [There is a finite number of such paths, but the number can be large.] This is called the "shortest path" (SP) problem. A variant of this problem is to find a shortest path that visits each city exactly once, called the "Traveling Salesman problem" (TSP). Another example, called "integer program" (IP), is a linear program (i.e., the problem of minimizing/maximizing a linear function subject to the variables satisfying some linear equations and inequalities) with the variables further required to be integers. This is common since quantities often must be integers (e.g., we can buy 1 or 2 or 3 tickets, but not $\frac{1}{2}$ ticket).

Discrete optimization traces its roots to a problem studied by the famous 18th century mathematician Leonhard Euler (1707-1783). In the Hanseatic town of Königsberg (now named Kalinigrad), there were 7 bridges going across the river Pregel to join four land masses. It was asked whether a trip can be made to cross each bridge exactly once and return to the start. Euler showed (using the mathematics of what we now call Eulerian path in graph theory) that this is impossible. This problem is closely related to a popular childhood game of tracing a certain line drawing without lifting the pen/pencil, as well as to street sweeping! Over the years, discrete optimization has developed to play major roles in real-world decision making, from scheduling to vehicle routing to VLSI layout, etc. This is helped by advances in computer softwares for solving large IP (e.g., LINDO, CPLEX); also see recent issues of the bulletin *OR/MS Today*.

For discrete optimization problems such as the ones described above (MST, SP, TSP, IP), the number of choices is often large–too large to explicitly search through them all even on a fast computer. For example, in the above 5 city example, the number of possible paths visiting each city exactly once can be as much as $4! = 24$. For n cities, this number can be as much as $n!$, which is huge even for $n = 100$. Thus, clever algorithms are needed to exploit the problem structure to find an optimal solution faster. We will study these structures and algorithms. Our study will involve graph theory, networks, linear programming, and some

combinatorics, matrices, and linear algebra. We will see that MST and SP are much "easier" problems than TSP and IP in the sense of (worst case) computational complexity. Computational complexity concerns how easily a problem can be solved by a digital computer (or, more formally, by a Turing machine). A famous open question concerns whether TSP and IP are also "easy" (so-called "P=NP" question).

For a good discussion of the TSP and discrete optimization, see the book [LLRS90]. For a history of IP and other optimization topics, see the collected personal reminiscences [LKS91]. For computational complexity, the classic [GJ79] is still worth a look–at least for the cartoons on pages 2-3!

These course notes are regularly updated, so suggestions/comments are welcome!

References

[GJ79] M.R. Garey and D.S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness,* W.H. Freeman and Co., San Francisco, 1979.

[LLRS90] E.L. Lawler, J.K. Lenstra, A.H.G. Rinnooy Kan and D.B. Shmoys *The Traveling Salesman Problem: a Guided Tour of Combinatorial Optimization,* Wiley-Interscience, Chichester, 1985 and 1990.

[LRS91] J.K. Lenstra, A.H.G. Rinnooy Kan, and A. Schrijver, Editors, *History of Mathematical Programming,* Elsevier North-Holland, Amsterdam, 1991.

Table of Contents

Foreword

Chapter 1. What is a Discrete Optimization? 1

Chapter 2. Graphs and Digraphs 3

2.1 Sets 3

2.2 Digraphs 3

2.3 Graphs 7

2.4 Vertex Degrees and Data Structures for Graphs/Digraphs 10

2.5 Subgraphs, Trees, and Spanning Trees 13

2.6 Eulerian Paths 18

2.7 Additional Topics 21

Chapter 3. Minimum Spanning Tree in a Weighted Graph 23

3.1 Definitions and Properties 23

3.2 Prim and Kruskal Algorithms 25

3.3 Optimality Condition and Sensitivity Analysis 29

3.4 Additional Topics 31

Chapter 4. Shortest Path in a Weighted Digraph 33

4.1 Definitions and Properties 33

4.2 Dijkstra and Bellman-Ford Algorithms 35

4.3 Additional Topics 40

Chapter 5. Flows, Cuts, and Matchings 42

5.1 Maximum Flow: Definitions and Properties 42

5.2 Maximum Flow: Flow Augmentation Algorithm 45

5.3 Matching and Covering in Bipartite Graphs 48

5.4 Minimum Cost Flow 52

5.5 Additional Topics 55

Chapter 6. Integer Program 56

6.1 Definitions and Properties 56

6.2 Branch and Bound Algorithm 58

6.3 Cutting Plane Algorithm 63

6.4 Additional Topics 66

Chapter 7. Computational Complexity 68

7.1 Problem Size and Method Efficiency 68

7.2 $\mathcal{P}$, $\mathcal{NP}$, and NP-Hard Problems 69

7.3 Approximate Solutions: Traveling Salesman Problem 72

Index 76

1 What is Discrete Optimization?

In the children's tale of Ali Baba and the Forty Thieves, Ali Baba carried away treasures from a cave in a bag. Suppose his bag can carry at most 10 kg in weight and there are five treasures, weighing 2, 3, 4, 5, 6 kg and worth 3,4,6,7,10 Dinars respectively. Which treasures should he carry to maximize their total worth? He can carry three treasures weighing 2, 3, 5 kg, with total worth of 14 Dinars. Or he can carry two treasures weighing 4, 6 kg, with total worth of 16 Dinars. Can he do better? After checking all possible choices, we can convince ourselves that 16 Dinars is the maximum. But, if instead of 5 treasures, there are 100 or even 1000 treasures, checking all possible choices would take too long. Is there a faster way to find an answer?

The above problem, known as the "knapsack problem" ("bag problem" just doesn't sound the same), is an example of a discrete optimization problem, i.e., among a finite number of choices, choose one whose worth/cost is maximum/minimum. Typically, the number of choices, though finite, is too large to check exhaustively so we must seek a faster way to find an answer.

For a second example of a discrete optimization problem, suppose there are 10 cities, numbered from 1 to 10, and the distance between each pair of cities is known. Starting at city 1, we wish to visit all other cities exactly once and to return to city 1. What order should we visit the other cities so that the total distance traveled is minimum? We can first visit city 2, then 3, then 4, ... Or we can first visit city 3, then 2, then 4, ... Or we can first visit city 10, then 9, ... This is called the Traveling Salesman Problem (TSP). A counting argument shows that there are $9 \cdot 8 \cdot 7 \cdots 2 \cdot 1 = 51840$ possible choices. In general, for n cities, the number of choices is $(n-1) \cdot (n-2) \cdots 2 \cdot 1 = (n-1)!$ which grows very rapidly with n.

Both the knapsack problem and the TSP belong to a class of notoriously difficult problems, called NP-hard problems. This means that no "fast" way of finding an answer exactly is known for these problems (and likely none exists). In practice, people settle for finding an inexact answer fast. But not all problems are so difficult. Suppose, in our 10 city example, we wish to travel from city 1 to city 10 in the shortest total distance, but we can visit any number of cities in between. Thus, we can go $1 \to 10$ directly or $1 \to 2 \to 10$ or $1 \to 3 \to 10$ or ... or $1 \to 2 \to 3 \to 10$ or ... This is called the Shortest Path (SP) problem. A counting argument shows that there are $1+8(1+7(1+6(1+\cdots(1+1)\cdots))) = 8!(1/8!+1/7!+\cdots+1/1!+1) \geq 8! \cdot 2$ possible choices. In general, for n cities, the number of choices is $(n-2)!(1/(n-2)!+1/(n-3)!+\cdots+1/1!+1) \geq (n-2)! \cdot 2$ which also grows very rapidly with n. Yet, this problem turns out to be much easier in that fast ways of finding an answer are known (see Chapter 4). A linear program (LP), which is the problem of maximimizing/minimizing a linear function over the feasible solutions to a system of linear equations/inequalities, may be viewed as a discrete optimization problem since at least one optimal solution occurs at a "corner point" of the feasible solution set (which is a polyhedron) and the number of such points is finite. This number can be huge, but fast ways of finding an answer are known. On the other hand, if we further restrict the variables to be

integer-valued, then the problem, called integer program (IP), is NP-hard (see Chapter 7).

Thus, the knapsack problem, TSP, and IP are "hard" while SP and LP are "easy". In general, the difficulty of a discrete optimization problem can be subtle. If the problem is changed somewhere, then it can go from difficult to easy or vice versa. Much experience and knowledge may be needed to detect which is which (like an art), and the boundary between hard and easy problems is one of the mysteries in discrete optimization. Knowing the difference can also save fruitless effort in trying to solve an impossibly difficult problem!

These days, discrete optimization problems arise all around us. When we fly on a major airline, there is a good chance that the airfare we pay, the seat availability, the crew on the plane, and the plane itself are determined by solving very large (millions of variables) discrete optimization problems. When we send packages by UPS, the routes taken by the packages may well be determined by solving large shortest path type problems. Computer models of protein use discrete optimization. Schedules of college sports teams may be determined by discrete optimization. Routing of Meals-on-Wheels vans has been done by solving inexactly a TSP (using, of all things, a space-filling curve).

Unlike LP, there is no one or two "dominant" methods for solving discrete optimization problems. Rather, there are many different methods, each with strengths and weaknesses. A good understanding of these methods will enable us to choose a suitable method from our tookit and adapt it to the application we face. (In real life, a problem rarely is exactly like ones in a book, so we must be able to adapt accordingly.) We will study these methods in the next chapters. One can hardly wait! ☺

2 Graphs and Digraphs

As we saw from the examples of Chapter 1, discrete optimization may involve optimally traveling through cities or connecting cities in some way. Mathematically, cities and the connections between them can be represented by "graphs" and "digraphs", depending on whether each connection can be traveled in both directions or only in one direction. To describe them, we first review some basic set theory.

2.1 SETS

We will use capital letters like S, X, Y, V to denote a set (of elements) and use braces "{" and "}" to delimit the elements of the set. The emptyset is always denoted by $\emptyset$. For example, let

$$U = \{2, 3, 5, 7, 11, ...\}$$

denote the set of prime numbers, and let $V = \{a, b, c, ..., z\}$ denote the set of English alphabets. We can take intersection $\cap$ and union $\cup$ of sets: If $W = \{0, 2, 4\}$, then

$$U \cap W = \{2\}, \qquad U \cup W = \{0, 2, 3, 4, 5, 7, 11, ...\},$$

and $U \cap \emptyset = \emptyset$, $U \cup \emptyset = U$, etc. Also, $U \cap W \subseteq U, W$ ("$\subseteq$" means "is a subset of") while $U, W \subseteq U \cup W$. We use $|S|$ to denote the **cardinality** (number of elements) of a set S, and use $x \in S$ (respectively, $x \notin S$) to mean x is (respectively, is not) an element of S. Thus

$$|U| = \infty, \qquad |V| = 26, \qquad |U \cap W| = 1, \qquad |\emptyset| = 0,$$

$3 \in U$, $1 \notin U$, etc. Is $\left\{ \frown, \frown \right\}$ a set? Most sets we consider will have finitely many elements.

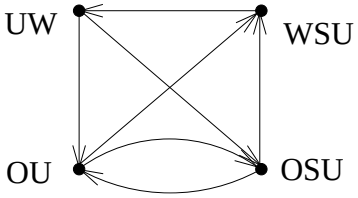
More than Oprah, mathematics is all about relationships. We will be interested in pairwise relationships. [No ménage à trois!] A nonempty set V and a (possibly empty) set A of *pairs* of elements of V form a **graph**, written as (V, A) and typically denoted by the captial letter G. Each $u \in V$ is called a **vertex** (or node). Each pair $(u, v) \in A$ is called an **arc** (or edge). We will consider only *finite* graphs, i.e., $|V| < \infty$. [There is some study of infinite graphs, but relatively few.] In the next two sections, we study two kinds of graphs, directed and undirected.

2.2 DIGRAPHS

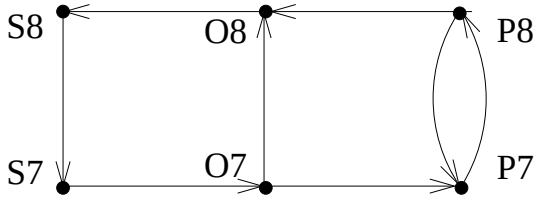
A graph is **directed** (called **digraph**) if the pairs in A are ordered, i.e., $(u, v) \in A$ is distinct from $(v, u) \in A$. This is represented pictorially by a dot for each vertex u and a line joining two dots for each arc (u, v), with an arrow pointing from u to v. Think of each arc as a one-way street.

Example 2.2.1. Let $V = \{UW, WSU, OU, OSU\}$ denote the northwest men's basketball teams in the PAC-10, and an arc (u, v) means team u has at least one win over team v in the 2005-2006 season. Then $A =$

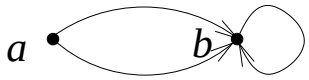
$\{(UW, OU), (UW, OSU), (WSU, UW), (OU, WSU), (OU, OSU), (OSU, WSU), (OSU, OU)\}$. Here $|V| = 4$, $|A| = 7$. The digraph (V, A) has the pictorial respresentation:



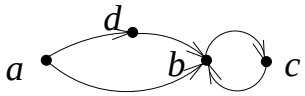
Example 2.2.2. Let $V = \{S7, O7, P7, S8, O8, P8\}$ denote six street intersections in downtown Seattle (cerca 2000), and an arc (u, v) means a car is allowed to travel in the direction from intersection u to intersection v. Then $A = \{(S7, O7), (O7, P7), (O7, O8), (P7, P8), (P8, P7), (P8, O8), (O8, S8), (S8, S7)\}$. Here $|V| = 6$, $|A| = 8$. The digraph (V, A) has the pictorial respresentation:



For convenience, we will assume that digraphs have *no self-loops* and *no parallel arcs in the same direction.* Thus, a digraph like



will be excluded. This avoids ambiguity in its algebraic representation $V = \{a, b\}$, $A = \{(a, b), (a, b), (b, b)\}$. [This ambiguity can be resolved by introducing additional notation to distinguish between parallel arcs, but it's not worth the extra complexity.] In fact, there is no loss of generality by making this assumption since each self-loop and parallel arc can be "broken up" by introducing a dummy vertex:



Here $V = \{a, b, c, d\}$, $A = \{(a, b), (a, d), (b, c), (c, b), (d, b)\}$. The topology of the graph is unchanged by this transformation. The only drawback is the increase in the number of vertices and arcs.

What does the digraph with vertex set $V = \{1, ..., m, m + 1, ..., m + n\}$ and arc set $A = \{(i, j) \mid i = 1, ..., m,\ j = m + 1, ..., m + n\}$ look like? What is $|A|$?

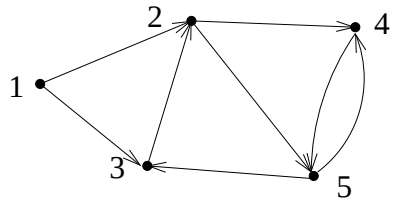
In a digraph $G = (V, A)$, we often like to know if we can reach a vertex from another vertex. A **path** in G (which we typically denote by the letter P) is a *sequence* of vertices and arcs in G:

$$P: \quad u_1, u_2, ..., u_{p+1},$$

where $p \geq 0$, $u_i \in V$ for $i = 1, ..., p + 1$, and $(u_i, u_{i+1}) \in A$ for $i = 1, ..., p$. [So u_i is the ith vertex in the path and p is the number of arcs in the path. For simplicity, we do not explicitly write out the arcs.]

Thus, in a digraph we can traverse a path only in the direction of the arcs in the path. For the digraph in Example 2.2.2, one path is $S7, O7, O8, S8$. [CAUTION: Do not confuse "sequence" with "set". They are very different.]

Example 2.2.3. For the digraph shown below



we have $V = \{1, 2, 3, 4, 5\}$, $A = \{(1,2), (1,3), (2,4), (2,5), (3,2), (4,5), (5,4), (5,3)\}$. Which of the following sequences is a path?

$$P_1: \ 1$$

$$P_2: \ 1, 2, 5, 3$$

$$P_3: \ 1, 2, 5, 3, 2$$

$$P_4: \ 4, 5, 4$$

$$P_5: \ 4, 5, 3, 2, 5, 4$$

$$P_6: \ 1, 2, 3$$

$$P_7: \ 1, 2, 4, 3$$

[Answer: All except P_6, P_7 are paths.] For P_1, $p = 0$ and $u_1 = 1$. For P_2, $p = 3$ and $u_1 = 1, u_2 = 2, u_3 = 3, u_4 = 5$, and so on.

A path is **simple** if no vertex repeats in it, i.e.,

$$u_i \neq u_j \quad \text{whenever} \quad i < j.$$

In Example 2.2.3, P_1 and P_2 are simple. A path is **closed** if it starts and ends at the same vertex, i.e.,

$$u_1 = u_{p+1}.$$

In Example 2.2.3, P_1, P_4, and P_5 are closed. A closed path is a **cycle** if (i) no vertex repeats in it except at the start and end and (ii) it has at least 2 arcs, i.e.,

$$u_i \neq u_j \quad \text{whenever} \quad i < j, \text{ except } i = 1, j = p+1 \qquad \text{and} \qquad p \geq 2.$$

In Example 2.2.3, only P_4 is a cycle. Notice that a simple path can never be a cycle, but can be a closed path (i.e., when it is a single vertex).

Fact 2.2.1. *In a digraph $G = (V, A)$, if there is a path from vertex $u \in V$ to vertex $v \in V$, then there is a simple path from u to v.*

Proof. This is left as an exercise.

A digraph G is **strongly connected** if every vertex can reach every other vertex by a path in G. It can be checked that the digraphs in Examples 2.2.1 and 2.2.2 are strongly connected, but not the one in Example 2.2.3 (no vertex can reach 1). Now, verifying a digraph is strongly connected using the definition is a lot of work since, for each pair of vertices we must check there is a path from one to the other. The following equivalent characterization provides an easier way to show that a digraph is strongly connected. [This characterization does not seem to make it easier to show that a graph is *not* strongly connected, however.]

Fact 2.2.2. *A digraph $G = (V, A)$ is strongly connected if and only if G has a closed path P going through all vertices (i.e., each vertex in V appears at least once in P).*

Proof. "If": Assume G has a closed path P going through all vertices, i.e.,

$$P: \quad u_1, u_2, ..., u_{p+1} = u_1,$$

for some $p \geq 0$. For any $u, v \in V$, since P goes through all vertices, u and v must appear in the P somewhere (possibly more than once), i.e.,

$$u = u_i \quad \text{and} \quad v = u_j \qquad \text{for some } 1 \leq i, j \leq p.$$

If $i = j$, then a path from u to v is $\quad u_i$.
If $i < j$, then a path from u to v is $\quad u_i, u_{i+1}, ..., u_j$.
If $i > j$, then a path from u to v is $\quad u_i, u_{i+1}, ..., u_{p+1} = u_1, u_2, ..., u_j$.

"Only if": Assume G is strongly connected. Let $n = |V|$ and label the vertices from 1 to n. Since G is strongly connected, there is a path P_1 from vertex 1 to 2, a path P_2 from 2 to 3, ..., a path P_{n-1} from $n-1$ to n, and a path P_n from n to 1, i.e.,

$$\begin{aligned}
P_1: &\ 1 = u_1^1, u_2^1, ..., u_{p^1+1}^1 = 2 && (p^1 \geq 0)\\
P_2: &\ 2 = u_1^2, u_2^2, ..., u_{p^2+1}^2 = 3 && (p^2 \geq 0)\\
&\ \vdots\\
P_{n-1}: &\ n-1 = u_1^{n-1}, u_2^{n-1}, ..., u_{p^{n-1}+1}^{n-1} = n && (p^{n-1} \geq 0)\\
P_n: &\ n = u_1^n, u_2^n, ..., u_{p^n+1}^n = 1 && (p^n \geq 0)
\end{aligned}$$

Connecting these paths in sequence yields a closed path P through all vertices:

$$P: \quad 1 = u_1^1, u_2^1, ..., u_{p^1+1}^1 = 2 = u_1^2, u_2^2, ..., u_{p^2+1}^2 = 3 = u_1^3, \ldots, u_{p^{n-1}+1}^{n-1} = n = u_1^n, u_2^n, ..., u_{p^n+1}^n = 1.$$

■

For Example 2.2.1, a closed path through all vertices is UW, OU, OSU, WSU, UW. For Example 2.2.2, one such path is $S7, O7, P7, P8, O8, S8, S7$. Can we find other such paths?

Exercises.

2.2.1. Let $P: u_1, u_2, ..., u_{p+1}$ be a path in some digraph (V, A). Suppose that $u_i = u_{i+4}$ for some $1 \leq i \leq p-3$ (so $p \geq 4$).

a) Write down a path from u_1 to u_{p+1} with 4 fewer arcs than P.

b) Write down a path with 8 more arcs than P.

c) Can the digraph have a path with 1 fewer arc than P? If yes, give an example. If no, explain why not.

d) Can $|V|$ be less than 4? If yes, give an example. If no, explain why not.

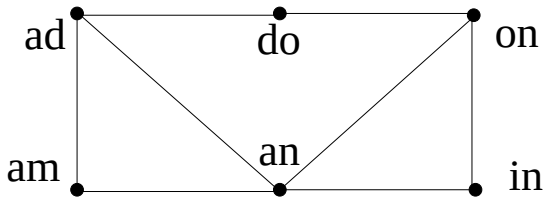
2.2.2. Prove Fact 2.2.1. [Hint: Start with the definition of a path: $u = u_1, u_2, ..., u_{p+1} = v$ with $p \geq 0$ and $u_i \in V$, $(u_i, u_{i+1}) \in A$, $i = 1, ..., p$. Then write down what it means mathematically if this path is not simple, etc.]

2.2.3. Bo owns a vacation home on Vachon Island that she wishes to rent for the period June 1 to Aug 31. She has solicited a number of bids, each having the following form: the day the rental starts (a rental day starts at 3 p.m.), the day the rental ends (checkout time is noon), and the total amount of the bid (in dollars). Bo wants to choose a selection of bids that would maximize her total revenue. Formulate this as a longest path problem, i.e., finding a path in a digraph from a given starting vertex to a given ending vertex whose total weight (sum of weights on each arc) is maximum. Specify clearly the digraph, the weight of each arc, etc. Is this digraph acyclic (i.e., contains no cycle)?

2.3 GRAPHS

A graph is **undirected** if the pairs in A are unordered, i.e., $(u, v) \in A$ is the same as $(v, u) \in A$. This is represented pictorially like a directed graph, but *without the arrows.* Since each pair is unordered, an arc can be traversed in both directions, so think of each arc as a two-way street.

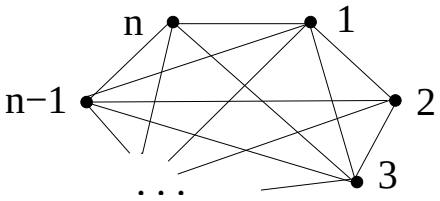
Example 2.3.1. Let $V = \{ad, am, do, me, on, in, an\}$ denote six 2-letter English words, and an arc (u, v) means word u has a letter in common with v. Here each pair is unordered since (u, v) means the same as (v, u). Then $A = \{(ad, am), (ad, an), (ad, do), (am, me), (am, an), (do, on), (on, in), (in, an)\}$. Here $|V| = 6$, $|A| = 8$. The undirected graph (V, A) has the pictorial respresentation:



We will follow a common convention and use "graph" to always mean an undirected graph. We will use "digraph" to mean a directed graph. [We can think of a graph as a digraph whereby each undirected arc is replaced by two directed arcs in opposite direction.] We will assume that graphs have *no self-loops* and *no parallel arcs*. Thus, a graph like

will be excluded. Like digraphs, there is no loss of generality by making this assumption since self-loops and parallel arcs can be "broken up" by introducing dummy vertices.

A graph is called **complete** if every pair of vertices is an arc. As a counting exercise, how many arcs does a complete graph with n vertices have?



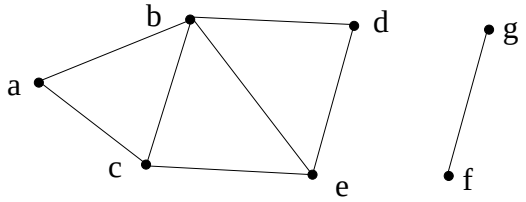
[Answer: $\frac{n(n-1)}{2}$ or, equivalently, $\binom{n}{2}$. Each vertex has $n-1$ arcs joined to it, and there are n vertices, for a total of $n(n-1)$ arcs, but we double count so we divide this by 2.]

For a graph, a **path** P is defined exactly as for a digraph, i.e.,

$$P: \quad u_1, u_2, ..., u_{p+1},$$

where $p \geq 0$, $u_i \in V$ for $i = 1, ..., p+1$, and $(u_i, u_{i+1}) \in A$ for $i = 1, ..., p$. (A path in an undirected graph is sometimes called a "chain".) However, since the pair is unordered so that $(u_i, u_{i+1}) = (u_{i+1}, u_i)$, each arc in the path can be traversed in either direction.

Example 2.3.2. For the graph shown below



we have $V = \{a, b, c, d, e\}$, $A = \{(a, b), (a, c), (b, c), (b, d), (b, e), (c, e), (d, e), (f, g)\}$. Some examples of (undi-

rected) paths are

$$P_1: \ a$$

$$P_2: \ f, g$$

$$P_3: \ a, b, c, a$$

$$P_4: \ a, b, d, e, b, c, a$$

$$P_5: \ a, c, b, a$$

$$P_6: \ a, b, a$$

Notice that P_3 and P_5 are the same path. A path in a graph can be traversed in either direction. For P_3, $p = 3$ and $u_1 = a, u_2 = b, u_3 = c, u_4 = a$.

For a graph, **simple** path and **closed** path are defined exactly as for a digraph. **Cycle** is defined as for a digraph *except* that it must have at least 3 arcs. This is to exclude closed paths with 2 arcs like P_6 in Example 2.2.2 (which has no repeating vertex except at the start and end). In Example 2.2.2, P_1, P_2 are simple paths, P_1, $P_3 = P_5$, P_4, P_6 are closed paths, and only P_3 is a cycle.

Fact 2.3.1. *In a graph $G = (V, A)$, if there is a path between vertex $u \in V$ and vertex $v \in V$, then there is a simple path between u and v.*

Proof. Similar to the proof of Fact 2.2.1.

A graph is **connected** if every vertex is joined to every other vertex by a path. It can be checked that the graph in Example 2.3.1 is connected, as is the complete graph, but not the one in Example 2.3.2. Similar to Fact 2.2.2 for digraph, the following equivalent characterization provides an easier way to show that a graph is connected.

Fact 2.3.2. *A graph $G = (V, A)$ is connected if and only if G has a closed path P going through all vertices.*

Proof. This is similar to the proof of Fact 2.2.2.

Exercises.

2.3.1.

a) Give an example of a digraph and a path in that digraph which is not a simple path but has no repeated arcs.

b) Give an example of a graph in which the shortest circuit has 5 arcs and the longest cycle has 8 arcs.

c) For a digraph (V, A) that is strongly connected and $|V| = n$, what is the least number of arcs? What is the most?

2.3.2.

a) In a connected graph with n ($n \geq 1$) vertices, what is the maximum possible number of arcs? What is the minimum possible number of arcs?

b) In a connected graph with 15 arcs, what is the the maximum possible number of vertices? What is the minimum possible number of vertices? [Hint: Use your answer to (a).]

2.3.3. A given set of pins on a circuit board, numbered from 1 to n, has to be connected by wires. [Think of the pins as points on the Euclidean plane.] In view of possible future changes or corrections, each pin can have *at most* two wires attached to it. Let c_{ij} denote the length of wire needed to connect pin i to pin j. The total wire length is to be minimized. Formulate this as a shortest hamiltonian cycle problem, i.e., find a cycle in a graph that visits each vertex exactly once and whose total weight is minimum. Specify clearly the graph, the weight of each arc, etc. [Hint: Add a dummy pin 0, with $c_{0i} = c_{i0} = 0$ for $i = 1, ..., n$.]

2.4 VERTEX DEGREES AND DATA STRUCTURES FOR GRAPHS/DIGRAPHS

The **degree** of a vertex $u \in V$ in a graph/digraph $G = (V, A)$ is the number of arcs joined to u, denoted by $\deg(u)$. Thus, the degree counts the number of "neighbors" of a vertex.

For the digraph in Example 2.2.1 with 6 arcs, we have $\deg(UW) = 3$, $\deg(WSU) = 3$, $\deg(OU) = 4$, $\deg(OSU) = 4$, whose sum $3 + 3 + 4 + 4 = 12$ equals $2 \cdot 6$. For the graph in Example 2.3.2 with 8 arcs, we have $\deg(a) = 2$, $\deg(b) = 4$, $\deg(c) = 3$, $\deg(d) = 2$, $\deg(e) = 3$, $\deg(f) = 1$, $\deg(g) = 1$, whose sum $2 + 4 + 3 + 2 + 3 + 1 + 1 = 16$ equals $2 \cdot 8$. Hmm..., the sum of the degrees seems to always equal twice the number of arcs!

Fact 2.4.1. *For any graph or digraph* $G = (V, A)$, $\sum_{u \in V} \deg(u) \ = \ 2|A|$.

Proof. Each arc is joined to exactly 2 vertices, so summing the degree over all vertices double counts each arc.

A consequence of the above fact is the following basic property of graphs/digraphs, whose proof we leave as an exercise.

Fact 2.4.2. *For any graph or digraph, the number of vertices of odd degree is even.*

For the digraphs in Examples 2.2.1, 2.2.2, 2.2.3 and the graphs in Examples 2.3.1, 2.3.2, the number of vertices of odd degree are, respectively, 2, 4, 2 and 2, 4.

The **outdegree** $\deg_{\text{out}}(u)$ of a vertex $u \in V$ is the number of arcs pointing out of u, i.e., of the form (u, v) for some $v \in V$. The **indegree** $\deg_{\text{in}}(u)$ is defined analogously. Notice that $\deg_{\text{out}}(u) + \deg_{\text{in}}(u) = \deg(u)$ for all $u \in V$. For the digraph in Example 2.2.2, $\deg_{\text{out}}(O7) = 2$, $\deg_{\text{in}}(O7) = 1$, $\deg_{\text{out}}(P7) = 1$, $\deg_{\text{in}}(P7) = 2$,..., etc. For the digraph in Example 2.2.3, $\deg_{\text{out}}(1) = 2$, $\deg_{\text{in}}(1) = 0$, $\deg_{\text{out}}(2) = 2$, $\deg_{\text{in}}(2) = 2$,..., etc.

While small graphs/digraphs (say, with fewer than 10 vertices) can be studied by their pictorial representations, large graphs/digraphs (with thousands or millions of vertices) can be studied only by using digital computers. How should graphs/digraphs be stored on digital computers? We consider two common data structures below:

1. Adjacency matrix. For a graph or digraph $G = (V, A)$, label the vertices from 1 to $n = |V|$, and form an $n \times n$ matrix Adj whose (i, j)th entry is

$$(\text{Adj})_{ij} = \begin{cases} 1 & \text{if } (i,j) \in A \\ 0 & \text{else} \end{cases} .$$

Adj is called the (vertex-vertex) **adjacency matrix** associated with G.

For the digraph in Example 2.2.3, the adjacency matrix is

$$\text{Adj} = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{array} \begin{array}{c} \begin{array}{ccccc} 1 & 2 & 3 & 4 & 5 \end{array} \\ \begin{pmatrix} & 1 & 1 & & \\ & & & 1 & 1 \\ & 1 & & & \\ & & & & 1 \\ & & 1 & 1 & \end{pmatrix} \end{array} .$$

[Empty entry means a 0.] For the graph in Example 2.3.1, the adjacency matrix is

$$\text{Adj} = \begin{array}{c} \\ ad \\ am \\ do \\ an \\ on \\ in \end{array} \begin{array}{c} \begin{array}{cccccc} ad & am & do & an & on & in \end{array} \\ \begin{pmatrix} & 1 & 1 & 1 & & \\ 1 & & & & & \\ 1 & & & & 1 & \\ 1 & & & & 1 & 1 \\ & & 1 & 1 & & 1 \\ & & & 1 & 1 & \end{pmatrix} \end{array} .$$

Notice that the adjacency matrix for a graph is always symmetric and has $2|A|$ nonzero entries. In contrast, the adjacency matrix for a digraph may not be symmetric and has $|A|$ nonzero entries.

Adjacency matrices can say a great deal about the topology of graphs/digraphs. One such property is the following:

Fact 2.4.3. *For any graph or digraph $G = (V, A)$ with adjacency matrix* Adj,

$$((i,j)\textit{th entry of } (\text{Adj})^k) = (\#\textit{ paths with } k \textit{ arcs from } i \textit{ to } j) \quad 1 \le i, j \le |V|.$$

Proof. This is left as an exercise.

For example, for the adjacency matrix associated with the digraph in Example 2.2.3, raising it to the 8th power gives

$$(\text{Adj})^8 = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{array} \begin{array}{c} \begin{array}{ccccc} 1 & 2 & 3 & 4 & 5 \end{array} \\ \begin{pmatrix} 0 & 5 & 7 & 10 & 10 \\ 0 & 4 & 6 & 9 & 9 \\ 0 & 3 & 4 & 6 & 6 \\ 0 & 2 & 4 & 6 & 5 \\ 0 & 4 & 5 & 7 & 8 \end{pmatrix} \end{array}$$

By Fact 2.4.3, there are 10 paths using exactly 8 arcs from vertex 1 to 4, and there are 8 paths using exactly 8 arcs from vertex 5 to itself. Notice that $(\text{Adj})^k$ can be computed efficiently in $O(\log_2(k))$ matrix-matrix multiplications. For example, we can compute $(\text{Adj})^{13}$ by first computing $(\text{Adj})^2$, $(\text{Adj})^4 = ((\text{Adj})^2)^2$, $(\text{Adj})^8 = ((\text{Adj})^4)^2$ and then

$$(\text{Adj})^{13} = (\text{Adj})^8(\text{Adj})^4(\text{Adj}).$$

This requires only 5 matrix-matrix multiplications.

2. Arc lists. Here, we order the arcs from 1 to $|A|$ and use two arrays (lists) to store the start and end vertex for the jth arc in their jth location.

For the digraph in Example 2.2.3, one such arc lists are:

Arc #	1	2	3	4	5	6	7	8
start vtx	1	1	2	2	3	4	5	5
end vtx	2	3	4	5	2	5	3	4

For the graph in Example 2.3.1, one such arc lists are:

Arc #	1	2	3	4	5	6	7	8
start vtx	*ad*	*ad*	*ad*	*am*	*an*	*an*	*do*	*on*
end vtx	*am*	*an*	*do*	*an*	*on*	*in*	*on*	*in*

An advantage of arc lists is the low storage when the graph/digraph is sparse, say, $|A| \leq \alpha|V|$ with $\alpha > 0$ a constant (though additional data structure may be needed to do certain graph operations efficiently). In contrast, the adjacency matrix would need to store $|V|^2$ entries or use more complicated data structure to store only the nonzero entries.

Exercises.

2.4.1. Prove that in any graph (V, A), the number of vertices of odd degree is even. [Hint: Use Fact 2.4.1 and break $\sum_{u \in V} \deg(u)$ into two sums: one over those $x \in V$ of odd degree and the other over those $x \in V$ of even degree. Alternatively, do an induction argument on the number of arcs.]

2.4.2. Show that in a graph (V, A), there is a vertex of degree at least $2|A|/|V|$. [Hint: Argue by contradiction.]

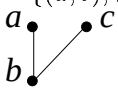
2.5 SUBGRAPHS, TREES, AND SPANNING TREES

In this section, we study special kinds of graphs that are useful later. For a graph $G = (V, A)$, a **subgraph** is a graph $G' = (V', A')$ satisfying

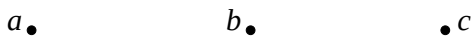
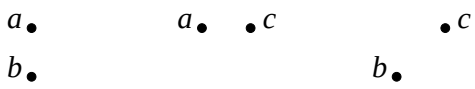
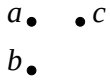
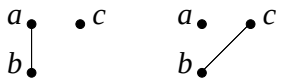
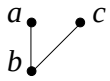
$$V' \subseteq V, \qquad A' \subseteq A.$$

In particular, (V, A) and $(V, \emptyset)$ are subgraphs of (V, A).

Example 2.5.1. For the graph $V = \{a, b, c\}$, $A = \{(a, b), (b, c)\}$ shown below



its subgraphs are:

#vtxs	#edges	subgraphs
1	0	
2	0	
2	1	
3	0	
3	1	
3	2	

Notice that the number of sugraphs can be large even for a small graph.

A **connected component** of a graph G is a connected subgraph that is not a subgraph of any other connected subgraph of G. In Example 2.5.1, the subgraphs in rows 1, 3, and 6 are connected. Of these, only the subgraph in row 6 (namely, the graph itself) is a connected component. The graph in Example 2.3.2 has two connected components–one comprising the two vertices f, g and the arc joining them, and the other comprising the rest of the graph.

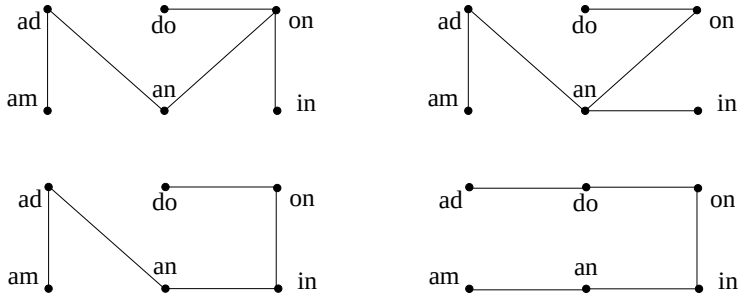
The following property of connected components, which is intuitive and not hard to prove, will be useful. By the way, can vertices in different connected components be joined by a path?

Fact 2.5.1. *If $G_1 = (V_1, A_1), ..., G_t = (V_t, A_t)$ $(t \geq 1)$ are the connected components of a graph $G = (V, A)$, then*

$$|V_1| + \cdots + |V_t| = |V|, \qquad |A_1| + \cdots + |A_t| = |A|.$$

A graph that is connected and has no cycle is called a **tree**. The graphs in Examples 2.3.1 and 2.3.2 are not trees. In Example 2.5.1, the subgraphs in rows 1, 3, and 6 are trees, while the other subgraphs are not.

A **spanning tree** of a graph G is a subgraph T that (i) has all vertices of G and (ii) is a tree. [More precisely, T is a spanning tree of $G = (V, A)$ if (i) $T = (V, B)$ with $B \subseteq A$ and (ii) T is connected and has no cycle.] The graph in Example 2.5.1 has one spanning tree (namely, itself). The graph in Example 2.3.1 has many spanning trees. We show four of them below:



The graph in Example 2.3.2 has no spanning tree. Since a spanning tree of G is connected and contains all vertices of G, G being connected is necessary for it to have a spanning tree. [When we write "have a XXX", we mean "have at least one XXX".] The following fact shows that this is also sufficient.

Fact 2.5.2. *A graph $G = (V, A)$ is connected if and only if G has a spanning tree.*

Proof. "If": Assume $G = (V, A)$ has a spanning tree $T = (V, B)$ with $B \subseteq A$. For any vertices $u, v \in V$, since T is connected, there is a path in T joining u and v (i.e., $u = u_1, u_2, ..., u_{p+1} = v$, where $p \geq 0$, $(u_i, u_{i+1}) \in B$ for $i = 1, 2, ..., p$). Since $B \subseteq A$, this path is also in G. This shows G to be connected.

"Only if": Assume G is connected. We will construct a spanning tree of G using the following TREE algorithm which searches along arcs of G to "color" new vertices until all vertices are colored.

TREE

Input: A graph $G = (V, A)$.

Output: A spanning tree of G if G is connected.

0. Choose any $r \in V$.

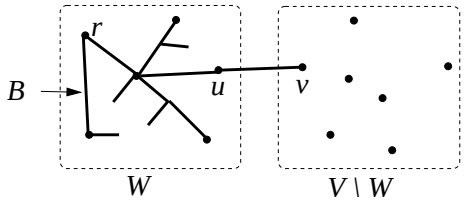
Initialize $W \leftarrow \{r\}$, $B \leftarrow \emptyset$.

1. If $W = V$, then output $T = (W, B)$; stop.

Else choose any $(u, v) \in A$ with $u \in W$, $v \notin W$. [If no such arc exists, G is not connected; stop.]

Update $W \leftarrow W \cup \{v\}$, $B \leftarrow B \cup \{(u, v)\}$.

Return to Step 1.



Clearly $T = (W, B)$ always remains a subgraph of G (since only vertices in V and arcs in A are added to W and B, respectively). We now prove by induction that $T = (W, B)$ always remains a tree. Initially,

$$T = (\{r\}, \emptyset),$$

which is a tree. Suppose $T = (W, B)$ is a tree on entering Step 1. We show below that

$$T^{\text{new}} = (W \cup \{v\}, B \cup \{(u, v)\})$$

is also a tree, where (u, v) is the chosen arc in Step 1. This would complete the induction.

- Since T is connected, by Fact 2.3.2, it has a closed path going through all vertices of W, which we can without loss of generality assume to start and end at u:

  $$u = u_1, u_2, ..., u_{p+1} = u.$$

  Then $v, u_1, u_2, ..., u_{p+1}, v$ is a closed path in T^{new} that goes through all vertices of $W \cup \{v\}$. By Fact 2.3.2, T^{new} is connected.
- Since T has no cycle, if T^{new} has a cycle, the cycle must use (u, v). But v has degree 1 in T^{new}, so v cannot be in a cycle of T^{new} (since each vertex in a cycle has degree at least 2). Thus T^{new} has no cycle.

Thus, when we stop in Step 1 with $W = V$, $T = (W, B)$ is a spanning tree of G (since T is a tree and its vertices spans V). ■

Notice that, initially $|W| = 1$, $|B| = 0$ and both are increased by 1 in Step 1 of TREE. So when $W = V$, we must have

$$|W| = |V|, \qquad |B| = |V| - 1.$$

Thus TREE has the following additional property.

Fact 2.5.3. *If a graph $G = (V, A)$ is connected, then TREE outputs a spanning tree with $|V| - 1$ arcs.*

If G is not connected, then TREE would find a spanning tree for the connected component containing the vertex r chosen in Step 0. If a graph has more than one spanning tree, then which spanning tree will be found by TREE depends on the choice of r in Step 0 and of the arc (u, v) in Step 1. **Depth-first search**

(DFS) corresponds to choosing (u, v) such that u was *most* recently added to W. **Breadth-first search** (BFS) corresponds choosing (u, v) such that u was *least* recently added to W. DFS tends to find "long and skinny" spanning trees while BFS tends to find "short and fat" spanning trees.

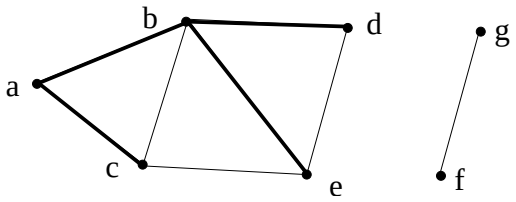
Example 2.5.2. Let us apply DFS and BFS to the graph in Example 2.3.2, starting at a.

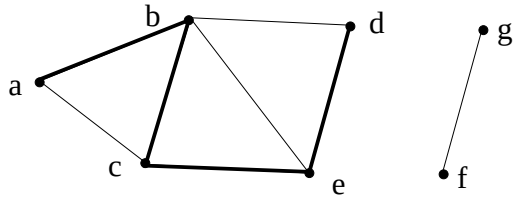
(a) BFS

W	B
$\{a\}$	$\emptyset$
$\{a, b\}$	$\{(a, b)\}$
$\{a, b, c\}$	$\{(a, b), (a, c)\}$
$\{a, b, c, d\}$	$\{(a, b), (a, c), (b, d)\}$
$\{a, b, c, d, e\}$	$\{(a, b), (a, c), (b, d), (b, e)\}$

(b) DFS

W	B
$\{a\}$	$\emptyset$
$\{a, b\}$	$\{(a, b)\}$
$\{a, b, c\}$	$\{(a, b), (b, c)\}$
$\{a, b, c, e\}$	$\{(a, b), (b, c)), (c, e)\}$
$\{a, b, c, e, d\}$	$\{(a, b), (b, c)), (c, e), (e, d)\}$





The trees found by DFS and BFS (which span the connected component containing a) are shown above. During the first iteration, instead of adding b to W and (a, b) to B, we could have added c to W and (a, c) to B, and so on. So other spanning trees could have been found.

There are many equivalent characterizations of a tree, each of which may be easier to use, depending on the situation. Below we give four well-known characterizations, including the definition. The proof makes use of the earlier facts.

Theorem 2.5.1. *For any graph $G = (V, A)$, the following four properties are equivalent.*

(a) *G is connected and has no cycle (i.e., a tree).*

(b) *G is connected and $|A| = |V| - 1$.*

(c) *G has no cycle and $|A| = |V| - 1$.*

(d) *For any $u, v \in V$, there is exactly one simple path in G joining u and v.*

Proof. (a)⇒(b): Assume G is a tree. Apply TREE to G. Sine G is connected, TREE finds a spanning tree $T = (V, B)$ with $B \subseteq A$ and $|B| = |A| = 1$. We claim that $B = A$. If not, there would exist an arc $(u, v) \in A \backslash B$. Sine T is connected, there is a simple path in T joining u and v (see Fact 2.3.1):

$$u = u_1, u_2, ..., u_{p+1} = v,$$

where $p \geq 0$, $(u_i, u_{i+1}) \in B$ for $i = 1, 2, ..., p$, and $u_1, u_2, ..., u_{p+1}$ are distinct. Then $u_1, u_2, ..., u_{p+1}, u$ is a cycle in G, contradicting G being a tree.

(b)⇒(c): Assume G is connected and $|A| = |V| - 1$. Apply TREE to G. Since G is connected, TREE finds finds a spanning tree $T = (V, B)$ of G with $|B| = |V| - 1$ (see Fact 2.5.3). Since $|A| = |V| - 1$, this means $|B| = |A|$. Since $B \subseteq A$, then $B = A$. So $T = G$. Since T is a tree, it has no cycle, so neither does G.

(c)⇒(a): Assume G has no cycle and $|A| = |V| - 1$. Let $(V_1, A_1), ..., (V_t, A_t)$ $(t \geq 1)$ be the connected components of G. These subgraphs are connected and have no cycle (since G has no cycle), so (a)⇒(b) implies that $|A_1| = |V_1| - 1$, ..., $|A_t| = |V_t| - 1$. Summing this yields

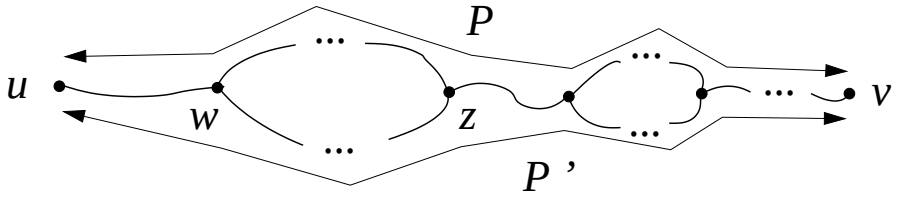
$$|A_1| + \cdots + |A_t| = |V_1| + \cdots + |V_t| - t,$$

which, by Fact 2.5.1, is equivalent to $|A| = |V| - t$. Since $|A| = |V| - 1$ by assumption, then $t = 1$, so G is connected.

(a)⇒(d): Assume G is a tree. For any $u \neq v \in V$, since G is connected, there is a simple path P from u to v (see Fact 2.3.2). Suppose there is another simple path P' in G from u to v. Then P and P' must diverge at some vertex w and then meet up again at some vertex z later, i.e.,

$$P: \ u = u_1, \ \ldots, \ w = u_i, \ \ldots, \ z = u_j, \ \ldots, \ v = u_{p+1}$$
$$P': \ u = u'_1, \ \ldots, \ w = u'_{i'}, \ \ldots, \ z = u'_{j'}, \ \ldots, \ v = u'_{p'+1}$$

for some $p \geq 2, p' \geq 2$ and $1 \leq i < j \leq p+1$, $1 \leq i' < j' \leq p+1$.



Since P is simple, $w = u_i, \ \ldots, \ z = u_j$ have no repeating vertex. Since P' is simple, $w = u'_{i'}, \ \ldots, \ z = u'_{j'}$ have no repeating vertex. Also, no vertex in $w = u_i, \ \ldots, \ z = u_j$ repeats in $w = u'_{i'}, \ \ldots, \ z = u'_{j'}$ except at w and z (or else P and P' would have met up earlier before z). Then

$$w = u_i, u_{i+1}, \ \ldots, \ z = u_j = u'_{j'}, \ \ldots, \ u'_{i'+1}, u'_{i'} = w$$

is a cycle in G, contradicting the assumption of G being a tree.

(d)⇒(a): This is left as an exercise. ■

Theorem 2.5.1 is very useful. For example, we can use it to prove that if $G = (V, A)$ is a tree with $|V| \geq 2$ (at least 2 vertices), then G has at least 2 vertices of degree 1. In particular, since G is connected, each vertex has degree at least 1. Suppose G has fewer than 2 vertices of degree 1. Then the number of vertices of degree 2 or more is at least $|V| - 1$, so

$$\sum_{u \in V} \deg(u) \geq 2(|V| - 1) + 1.$$

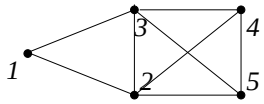
Since $\sum_{u \in V} \deg(u) = 2|A|$ by Fact 2.4.1 and $|A| = |V| - 1$ by Theorem 2.5.1, this implies that

$$2(|V| - 1) \geq 2(|V| - 1) + 1,$$

which is a clear contradiction.

Exercises.

2.5.1. Consider the graph shown:



a) Apply the DFS (Depth First Search) version of TREE to find a spanning tree for this graph, starting at vertex 1. Write down the vertex set W and arc set B at each iteration.

b) Apply the BFS (Breadth First Search) version of TREE to find a spanning tree for this graph, starting at vertex 1. Write down the vertex set W and arc set B at each iteration.

2.6 EULERIAN PATHS

In the foreword we mentioned the famous bridges of Königsberg problem solved by Leonhard Euler in the 18th century. This is the question of whether each of 7 bridges can be traversed exactly once (in either direction) and return to the start.

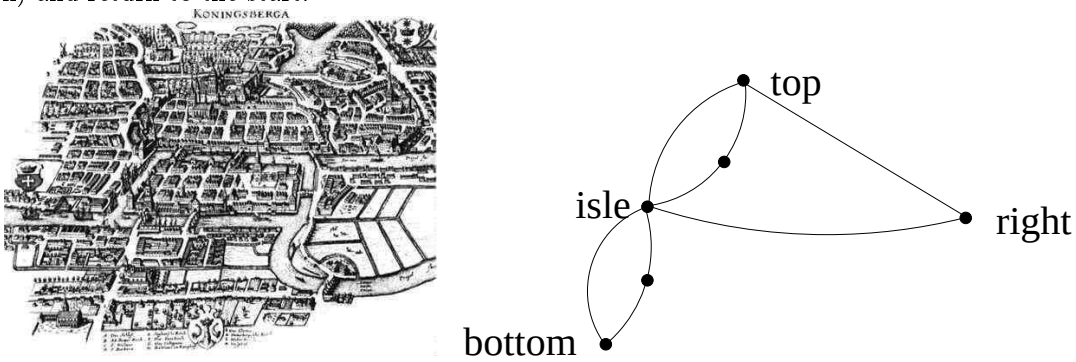


Figure 2.6.1

On the left are the bridges as they were seen in Euler's time. The corresponding graph is shown on the right, with each bridge corresponding to an arc and each land mass corresponding to a vertex. Two dummy vertices have been added to "break up" parallel arcs. Euler proved that the answer is "no". How did he do it? In general, proving that something does *not* exist is difficult!

A popular childhood game is to trace a line figure without lifting the pen, like the two figures below (whose vertices we have numbered):

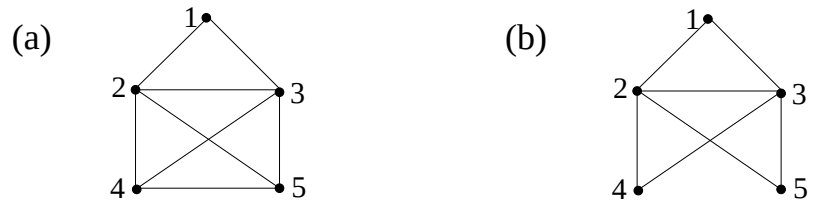


Figure 2.6.2

This is of the same nature as the bridge problem.

Let's make our problem more precise. We say that a path P in a graph/digraph $G = (V, A)$ is **eulerian** if P traverses every arc in A exactly once. Then the bridge problem reduces to asking whether the graph in Figure 2.6.1 has a closed eulerian path. Similarly, the game of tracing a line figure reduces to asking whether the figure, viewed as a graph, has an eulerian path. It is not difficult to see that both graphs in Figure 2.6.2 have eulerian paths, e.g., (a) $4, 2, 1, 3, 5, 2, 3, 4, 5$ and (b) $1, 2, 4, 3, 2, 5, 3, 1$. But if the graph has hundreds or thousands of arcs, checking by hand would be impossible. Is there an easy way to check whether a graph has a (closed) eulerian path? Is there a systematic way (i.e., a programmable algorithm) to find such a path? The following theorem, attributed to Euler, gives an answer to these questions.

Theorem 2.6.1. *A graph $G = (V, A)$ has a closed eulerian path if and only if (i) G has no vertex of odd degree and (ii) G is connected, excluding isolated vertices.*[1]

Proof. "Only if": Assume $G = (V, A)$ has a closed eulerian path

$$P: \quad u_1, u_2, ..., u_{p+1} = u_1,$$

where $p = |A|$ and $(u_i, u_{i+1}) \in A$ for $i = 1, 2, ..., p$. Start at u_1 and traverse along P. Each time we leave and return to u_1, the number of traversed arcs joined to u_1 increases by 2, so this number always remains even. For each vertex $u \neq u_1$, each time we enter and leave u, the number of traversed arcs joined to u increases by 2, so this number always remains even. When we finish traversing P (so we return to u_1), the number of traversed arcs joined to each vertex is even. Since each arc is traversed exactly once, this number is the degree of the vertex. The vertices in P belong to one connected component of G (see Fact 2.3.2). The remaining vertices must have degree 0 (since all arcs are joined to vertices in P), so they are isolated.

"If": Assume $G = (V, A)$ has no vertex of odd degree and is connected, excluding isolated vertices. Then G has one connected component with all the arcs, plus some isolated vertices. If we discard these isolated vertices, then G still has no vertex of odd degree and is connected. We will construct a closed eulerian path of G by applying the following EPATH algorithm to G with these isolated vertices discarded. The algorithm starts at any vertex, and extends the path by searching along previously untraversed arcs until it closes on itself, and then restarts the search from a vertex in this closed path that has an untraversed arc joined to it, etc, until all arcs are traversed.

[1] *An isolated vertex is a vertex of degree 0.*

EPATH

Input: A connected graph $G = (V, A)$ with no vertex of odd degree.

Output: A closed eulerian path of G.

0. Choose any $r \in V$ that has an arc in A joined to it.

$P \leftarrow r$, $B \leftarrow \emptyset$, $u \leftarrow r$.

1. Choose any $(u, v) \in A \backslash B$.

$P \leftarrow P, v$

$B \leftarrow B \cup \{(u, v)\}$

If $v \neq r$, then

$u \leftarrow v$. Return to Step 1.

Else ($v = r$, so P closed on itself)

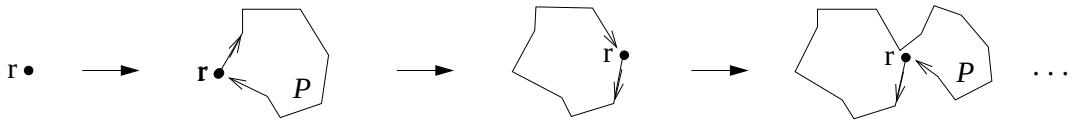
If $B = A$, then

output P; stop.

Else choose any vertex r in P with an arc in $A \backslash B$ joined to it.

Resequence P to start and end at r (i.e., $P:\ r, ..., r$).

$u \leftarrow r$. Return to Step 1.

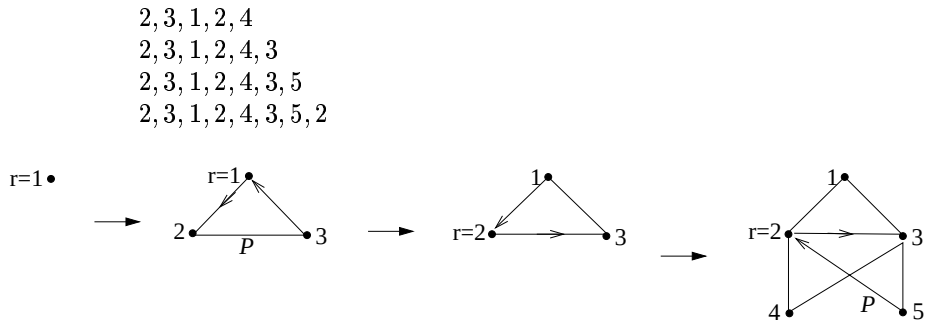


[P can be easily updated by storing the predecessor and successor of each vertex in P.] To see that EPATH works correctly, first note that P always remains a path and B is the set of arcs in P. Since we only add to B an arc in $A \backslash B$, no arc in B repeats. Thus, P always remains an eulerian path of the subgraph (V, B) and is closed whenever $v = r$ in Step 1. So when $v = r$ and $B = A$, P is a closed eulerian path of G and remains so after isolated vertices are added back to G.

It only remains to show that each time we return to Step 1, we can find an untraversed arc $(u, v) \in A \backslash B$ joined to u. Here we use the assumption that each vertex of G has an even number of arcs joined to it, so each time the path P enters and leaves a vertex $u \neq r$, the number of untraversed arcs joined to u decreases by 2 and hence remains an even number. This means that when we extend P by an arc to enter a vertex $u \neq r$, there must be another untraversed arc joined to u which allows P to leave u. ■

If we apply EPATH to the graph in Figure 2.6.2(b), with $r = 1$ in Step 0 and with v in Step 1 chosen to have the least label, then r and P are updated as follows:

root r	path P
1	1
	1, 2
	1, 2, 3
	1, 2, 3, 1
2	2, 3, 1, 2



Corollary 2.6.1. *A graph $G = (V, A)$ has an eulerian path that is not closed if and only if (i) G has exactly 2 vertices of odd degree and (ii) G is connected, excluding isolated vertices.*

Proof. See Exercise 2.6.2. ■

By Theorem 2.6.1, the graph in Figure 2.6.2(b) has a closed eulerian path. By Corollary 2.6.1, the graphs in Figures 2.6.1 and 2.6.2(a) have eulerian paths that are not closed (which must start and end at the two vertices of odd degree). What about the graphs in Examples 2.3.1 and 2.3.2?

For digraphs, analogous results can be shown by taking into account the direction of the arcs. The proofs are similar to those for graphs and are omitted.

Theorem 2.6.2. *A digraph $G = (V, A)$ has a closed eulerian path if and only if (i) all vertices in V have indegree equal outdegree and (ii) ignoring directions, G is connected, excluding isolated vertices.*

Corollary 2.6.2. *A digraph $G = (V, A)$ has an eulerian path that is not closed if and only if (i) all vertices in V have indegree equal outdegree, except for two vertices one of which has indegree equal outdegree plus 1 and the other has indegree equal outdegree minus 1, and (ii) ignoring directions, G is connected, excluding isolated vertices.*

By Theorem 2.6.2, the digraph in Example 2.2.1 has an eulerian path that is not closed (starting at UW and ending at WSU). By Corollary 2.6.2, the digraphs in Figures 2.2.2 and 2.2.3 have no eulerian path.

Can we easily find the "largest" subgraph that has a closed eulerian path?

Exercises.

2.6.1. Suppose that, after removing an arc from some graph G, the remaining graph has a closed eulerian path. Explain your answers below.

(a) Must G have a closed eulerian path?

(b) Must G have an eulerian path?

(c) Does your answer to (a) and (b) change if G is connected?

2.6.2. Prove Corollary 2.6.1. You can use Theorem 2.6.1. [Hint: Add a vertex z and join z by two arcs to two suitably chosen vertices in G. Treat the "if" and the "only if" directions separately.]

2.7 ADDITIONAL TOPICS

A path P in a graph/digraph $G = (V, A)$ is **hamiltonian** (named after Sir William Rowan Hamilton who described such path in 1857) if each vertex in V appears in P exactly once (except at the start and end). Thus, a hamiltonian path is either a simple path or a cycle, depending on whether its start and end vertices are the same. Although the definition of a hamiltonian path looks very similar to that of an eulerian path, determining whether a graph/digraph has a hamiltonian path is much harder. In fact, this problem is among the class of NP-complete problems (see Chapter 7). Some partial results are known. For example, O. Ore showed in 1960 that a sufficient condition for a graph $G = (V, A)$ with $|V| \geq 3$ to have a hamiltonian cycle is that

$$\deg(u) + \deg(v) \geq |V| \qquad \text{for all} \quad u \neq v, \ (u, v) \notin A.$$

[Roughly speaking, any two non-neighboring vertices must together have many neighbors.] However, this condition is not necessary for existence, as can be seen from examples. Similarly, D.R. Woodall showed in 1972 that a a sufficient condition for a digraph $G = (V, A)$ with $|V| \geq 3$ to have a hamiltonian cycle is that

$$\deg_{\text{out}}(u) + \deg_{\text{in}}(v) \geq |V| \qquad \text{for all} \quad u \neq v, \ (u, v) \notin A.$$

A. Ghouila-Houri proved in 1960 a different result that a digraph $G = (V, A)$ has a hamiltonian cycle if G is strongly connected and

$$\deg(u) \geq |V| \qquad \text{for all} \quad u \in V.$$

This sufficient condition is somewhat easier to check.

Exercises.

2.7.1. Prove that a digraph $G = (V, A)$ is strongly connected if

$$\deg_{\text{out}}(u) \geq \frac{|V|}{2} \quad \text{and} \quad \deg_{\text{in}}(u) \geq \frac{|V|}{2} \qquad \text{for all } u \in V.$$

3 Minimum Spanning Tree in a Weighted Graph

In Chapter 2, we studied graphs/digraphs, paths, subgraphs, and spanning trees for graphs. In this and the next two chapters, we will use these knowledges to study three kinds of discrete optimization problems.

In problems where we wish to link up cities or towns or computers at minimum cost or maximum reliability (see the foreword for such an example), it often involves finding a spanning tree on a graph whose "weight" is minimum or maximum. In this chapter we study this kind of discrete optimization problem and, in particular, properties of such spanning trees and algorithms to find them.

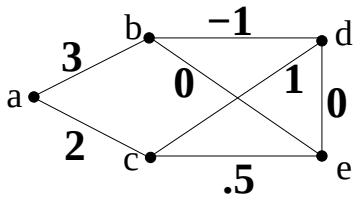
3.1 DEFINITIONS AND PROPERTIES

Recall from Fact 2.5.2 that a graph has a spanning tree if and only if it is connected. Let $G = (V, A)$ be a connected graph, with a "weight/length" $\omega_{uv} \in \Re$ for each arc $(u, v) \in A$. ["$w \in \Re$" means w is a real number.] The **weight** of a spanning tree $T = (V, B)$ of G is defined to be the *sum* of the weight of all arcs in T, i.e.,

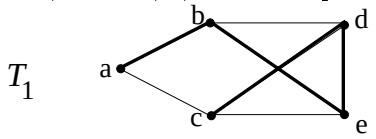
$$\omega(T) = \sum_{(u,v)\in B} \omega_{uv}.$$

[Instead of sum, other operands such as "max" can also be used; see Exercise 3.3.4.] In applications, the arc weight ω_{uv} could be the physical distance between two points u and v or the cost of linking u and v or the time to travel between u and v, etc.

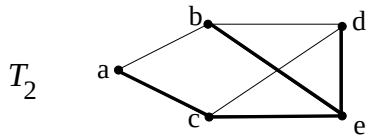
Example 3.1.1. Consider the graph with arc weights shown below



So $\omega_{ab} = 3$, $\omega_{ac} = 2$, $\omega_{be} = 0$,..., etc. Two spanning trees (shown darkened) with their weights are



T_1 $\quad w(T_1) = 3 + 0 + 1 + 0 = 4$



T_2 $\quad w(T_2) = 2 + .5 + 0 + 0 = 2.5$

Below we give some properties of $\omega(T)$ based on its definition as a sum of arc weights in T. For example, if we add 7 to all arc weights in Example 3.1.1, then the weight of the spanning trees T_1 and T_2 both change

by 28 to, respectively, 32 and 53.

Fact 3.1.1. *Let $G = (V, A)$ be a graph with weights ω_{uv} for $(u, v) \in A$.*

(a) If we add $\alpha \in \Re$ to ω_{uv} for all $(u, v) \in A$, then $\omega(T)$ changes by an additive factor of $(|V| - 1)\alpha$ for all spanning trees T of G.

(b) If we multiply $\alpha \in \Re$ to ω_{uv} for all $(u, v) \in A$, then $\omega(T)$ changes by a multiplicative factor of α for all spanning trees T of G.

Proof. (a): By Theorem 2.5.1, each spanning tree $T = (V, B)$ of G has $|B| = |V| - 1$ arcs, so if we change ω_{uv} to $\omega_{uv} + \alpha$ for all $(u, v) \in A$, then

$$w^{\text{new}}(T) \;=\; \sum_{(u,v)\in B} (\omega_{uv} + \alpha) \;=\; \sum_{(u,v)\in B} \omega_{uv} + |B|\;\alpha \;=\; w^{\text{old}}(T) + (|V| - 1)\alpha.$$

(b): If we change ω_{uv} to $\omega_{uv}\;\alpha$ for all $(u, v) \in A$, then

$$w^{\text{new}}(T) \;=\; \sum_{(u,v)\in B} \omega_{uv}\;\alpha \;=\; \left(\sum_{(u,v)\in B} \omega_{uv}\right)\alpha \;=\; w^{\text{old}}(T)\;\alpha.$$

■

We wish to find a spanning tree whose weight is *minimum*, which we call a **minimum-weight spanning tree** (**MST**). [The problem of finding a spanning tree of maximum weight can be reduced to this problem by negating all arc weights.] This is a discrete optimization problem since there are finitely many spanning trees to choose from. But explicitly search through them all would be too slow, so a more efficient algorithm is needed. Notice that simply picking the "cheapest" (i.e., least weight) arcs would not work since they might form cycles. In Example 3.1.1, the three cheapest arcs form the cycle b, c, d, b.

Fact 3.1.1(a) implies that we can change all arc weights by equal amount without changing the MST(s). Thus we can add a large positive constant to all arc weights to make them all positive. Similarly, Fact 3.1.1(b) implies that we can multiply all arc weights by a *positive* constant without changing the MST(s). We next have a key property of a spanning tree T, namely, any arc (u, v) outside of T can be swapped with any arc in the cycle created by adding (u, v) to T to yield another spanning tree.

Fact 3.1.2. *Let $T = (V, B)$ be any spanning tree of a graph $G = (V, A)$. For any $(u, v) \in A \backslash B$, let P_{uv} be the simple path in T joining u and v. For any arc (w, z) in P_{uv}, the subgraph*

$$T' = (V, B') \qquad \textit{with} \qquad B' = (B \cup \{(u, v)\}) \setminus \{(w, z)\}$$

is also a spanning tree of G.

Proof. Since $B \subseteq A$ and $(u, v) \in A \backslash B$, we have $B' \subseteq A$, so T' is a subgraph of G. Since we replaced an arc in B by an arc not in B to obtain B', $|B'| = |B|$. Also, since T is a spanning tree, Theorem 2.5.1 implies $|B| = |V| - 1$. Together this means $|B'| = |V| - 1$.

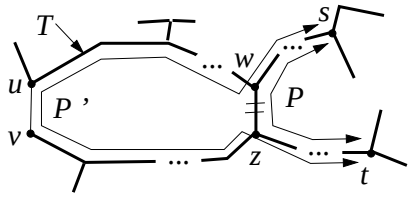
We now prove that T' is connected which, by Theorem 2.5.1, would imply that T' is a tree and hence a spanning tree of G. Since (w,z) is in P_{uv}, we have

$$P_{uv}: \quad u = u_1, ..., u_i = w, u_{i+1} = z, ..., u_{p+1} = v,$$

for some $1 \leq i \leq p$, where $p \geq 1$. Since T is connected, for any two vertices $s, t \in V$, there is a simple path P in T joining them. If P does not use (w,z), then P is still a path in T'. If P uses (w,z), then P has the form $s, ..., w, z, ..., t$, with (w,z) appearing only once in P (since P is simple). Hence

$$P': \quad s, ..., u_i = w, ..., u_1 = u, u_{p+1} = v, ..., u_{i+1} = z, ..., t$$

is a path in T' joining s, t.



Thus, for any two vertices in V, there is a path in T' joining them, so T' is connected. ■

[CAUTION: Do *not* write $T \subseteq G$ to indicate that T is a subgraph of G. "$\subseteq$" means a subset of, not a subgraph of. A graph is not a set.]

Exercises.

3.1.1. Can the following problems be transformed into an MST problem? Explain your answer.

(a) $G = (V, A)$ is a connected graph with arc weights $\{\omega_{uv}\}_{(u,v)\in A}$. Find a spanning tree $T = (V, B)$ of G that minimizes the Euclidean weight $\left[\sum_{(u,v)\in B} (\omega_{uv})^2\right]^{1/2}$.

(b) $G = (V, A)$ is a connected graph with each arc (u, v) having a probability $p_{uv} > 0$ of not failing. Assuming the arcs fail independently, the probability that no arc in a spanning tree $T = (V, B)$ fails is $\prod_{(u,v)\in B} p_{uv}$. Find a spanning tree of G that maximizes the probability of no arc failing. [Hint: Use log.]

(c) $G = (V, A)$ is a connected graph with every arc colored either red or blue. Find a spanning tree with the maximum number of red arcs.

3.2 PRIM AND KRUSKAL ALGORITHMS

A simple and popular algorithm for finding an MST is to specialize the TREE algorithm of Section 2.5 so that in Step 1 we choose an arc whose weight is *minimum* among all arcs between W and $V \setminus W$. This

is usually called Prim's algorithm since it was publicized in a 1957 paper of R.C. Prim, though it had been described by V. Jarník in 1930 and rediscovered by E.W. Dijkstra in 1959.

MST-Prim

Input: A graph $G = (V, A)$, with weights ω_{uv} for $(u, v) \in A$.

Output: An MST if G is connected.

0. Choose any $r \in V$.

Initialize $W \leftarrow \{r\}$, $B \leftarrow \emptyset$.

1. If $W = V$, then output $T = (W, B)$; stop.

Else choose any $(u, v) \in A$ with $u \in W$, $v \notin W$, and whose weight ω_{uv} is minimum among all such arcs. [If no such arc exists, G is not connected; stop.]

Update $W \leftarrow W \cup \{v\}$, $B \leftarrow B \cup \{(u, v)\}$.

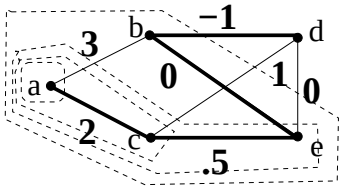
Return to Step 1.

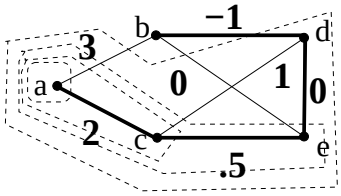
How efficient is MST-Prim? It searches at most $|A|$ arcs per Step 1, and repeats Step 1 at most $|V| - 1$ times (see Fact 2.5.3). Thus the overall effort is about $(|V| - 1)|A|$ operations, which is $O(|V||A|)$. [By **operations**, we mean $+$, $-$, $\cdot$, $/$, $>$, $<$, $=$. Also, $O(\beta)$ means a quantity that is less than or equal to $\alpha \cdot \beta$ for s some $\alpha > 0$ independent of the graph and the arc weights.] With a more sophisticated data structure called "Fibonacci heaps", this can be improved to $O(|A| + |V| \log_2 |V|)$ operations.

If we apply MST-Prim to the weighted graph in Example 3.1.1, starting at vertex a, then arcs may be added in the order:

W	B
$\{a\}$	$\emptyset$
$\{a, c\}$	$\{(a, c)\}$
$\{a, c, e\}$	$\{(a, c), (c, e)\}$
$\{a, c, e, b\}$	$\{(a, c), (c, e), (e, b)\}$
$\{a, c, e, b, d\}$	$\{(a, c), (c, e), (e, b), (b, d)\}$

and the resulting MST is shown below on the left, with each W set circled by dotted line.





In this example, the MST is not unique, and MST-Prim can alternatively add arc (e, d) instead of (e, b):

W	B
$\{a\}$	$\emptyset$
$\{a, c\}$	$\{(a, c)\}$
$\{a, c, e\}$	$\{(a, c), (c, e)\}$
$\{a, c, e, d\}$	$\{(a, c), (c, e), (d, e)\}$
$\{a, c, e, d, b\}$	$\{(a, c), (c, e), (d, e), (b, d)\}$

which results in the MST shown on the right. Which MST is found depends on the mechanism used to break tie when the minimum in Step 1 is achieved by more than one arc. as a computer code, Such a tie-breaking

mechanism is inherent when writing MST-Prim as a computer code.

We show below that MST-Prim indeed works correctly. [This is not obvious!] The proof uses Fact 3.1.3. If G is not connected, then MST-Prim will find an MST for the connected component that contains the starting vertex r.

Fact 3.2.1. *MST-Prim outputs an MST whenever the graph $G = (V, A)$ is connected.*

Proof. We will prove by induction that each (W, B) in Step 1 of MST-Prim is a subgraph of some MST. Then, since MST-Prim is a special case of the TREE algorithm, when $W = V$, the output $T = (W, B)$ is a spanning tree, so it must be equal to this MST (since spanning trees have the same number of arcs).

Initially $(W, B) = (\{r\}, \emptyset)$, which is a subgraph of any MST.

Suppose, on entering Step 1, (W, B) is a subgraph of some MST $T^* = (V, B^*)$ (so $B \subseteq B^*$). We now show that

$$(W^{\text{new}}, B^{\text{new}}) \qquad \text{with} \qquad W^{\text{new}} = W \cup \{v\}, \quad B^{\text{new}} = B \cup \{(u, v)\},$$

where (u, v) is the chosen arc in Step 1, is also a subgraph of some MST (possibly different from T^*), which would complete the induction.

If $(u, v) \in T^*$, then $B^{\text{new}} \subseteq B^*$ (as well as $W^{\text{new}} \subseteq V$), so $(W^{\text{new}}, B^{\text{new}})$ is a subgraph of T^*.

Else $(u, v) \notin T^*$, and let P_{uv} be the simple path in T^* joining u and v. Since $u \in W$ and $v \notin W$, there exists some arc (w, z) in P_{uv} with $w \in W$ and $z \notin W$. Then our choice of (u, v) in Step 1 implies

$$\omega_{uv} \leq \omega_{wz}.$$

By Fact 3.1.2,

$$T^{*'} = (V, B^{*'}) \qquad \text{with} \qquad B^{*'} = (B^* \cup \{(u, v)\}) \setminus \{(w, z)\}$$

is also a spanning tree of G. Moreover,

$$\omega(T^{*'}) \;=\; \omega(T^*) + \omega_{uv} - \omega_{wz} \;\leq\; \omega(T^*).$$

Since T^* is an MST, this shows that $T^{*'}$ is also an MST. Since $B \subseteq B^*$ and $(w, z) \notin B$ (due to $z \notin W$), we see that $B^{\text{new}} \subseteq B^{*'}$ and hence $(W^{\text{new}}, B^{\text{new}})$ is a subgraph of $T^{*'}$. ∎

A second popular algorithm for finding an MST, attributed to J.B. Kruskal in 1956, uses the simple idea of adding arcs in order of increasing weight, skipping any that forms a cycle. However, some care is needed to easily detect the formation of a cycle. This is achieved by giving the same label to all vertices in the same connected component, so two vertices in the same connected component are easily detected.

MST-Kruskal

Input: A graph $G = (V, A)$, with weights ω_{uv} for $(u, v) \in A$.

Output: An MST if G is connected.

0. Sort the arcs in order of increasing weight.

Initialize $B \leftarrow \emptyset$, $\ell(u) \leftarrow u$ for all $u \in V$.

1. Choose the next arc (u, v) in the ordering.

If $\ell(u) = \ell(v)$, then return to Step 1.

Else ($\ell(u) \neq \ell(v)$)

Update $B \leftarrow B \cup \{(u, v)\}$, $\ell(w) \leftarrow \ell(u)$ for all $w \in V$ with $\ell(w) = \ell(v)$.

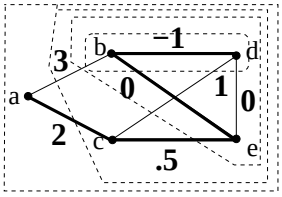
Return to Step 1.

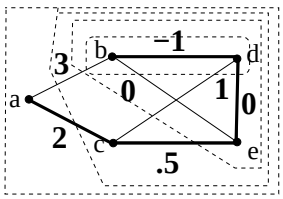
How efficient is MST-Kruskal? It is well-known that the sorting $|A|$ numbers can be done in $O(|A| \log_2 |A|)$ operations. Since $|A| \leq |V|(|V| - 1)/2$ so that $\log_2 |A| \leq 2 \log_2 |V|$, this means $O(|A| \log_2 |V|)$ operations. The first case in Step 1 uses $O(1)$ operations and can repeat at most $|A|$ times, for a total of $O(|A|)$ operations. The second case in Step 1 uses $O(|V|)$ operations and can repeat at most $|V| - 1$ times, for a total of $O(|V|^2)$ operations. So the total effort is $O(|A| \log_2 |V| + |V|^2)$ operations. This can be improved to $O(|A| \log_2 |V|)$ by adopting the convention that the connected component of (V, B) containing v is smaller (in number of vertices) than the one containing u. [This necessitates keeping track of the number of vertices in each connected component of (V, B), which can be done efficiently.] Then, each time when a vertex's label is updated, the connected component of (V, B) to which it belongs at least doubles in size, so its label can change at most $\log_2 |V|$ times. Summing over all vertices yields at most $|V| \log_2 |V|$ label changes in total.

If we apply MST-Kruskal to the weighted graph in Example 3.1.1, starting at vertex a, then arcs (after sorting by weights -1, 0, 0, .5, 1, 2, 3) may be added in the order:

B	$\ell(a), \ell(b), \ldots \ell(e)$	
$\emptyset$	a, b, c, d, e	
$\{(b, d)\}$	a, b, c, b, e	
$\{(b, d), (b, e)\}$	a, b, c, b, b	(skip arc (d, e))
$\{(b, d), (b, e), (c, e)\}$	a, b, b, b, b	(skip arc (c, d))
$\{(b, d), (b, e), (c, e), (a, c)\}$	b, b, b, b, b	

and the resulting MST is shown below on the left, with each new connected component circled by dotted line.





In this example, the MST is not unique, and MST-Kruskal can alternatively add arc (d, e) instead of (b, e), which results in the MST shown on the right.

We can similarly show that MST-Kruskal works correctly.

Fact 3.2.2. *MST-Kruskal outputs an MST whenever the graph $G = (V, A)$ is connected.*

Proof. Similar to the proof of Fact 3.2.1, this involves showing by induction that each B in Step 1 of MST-Kruskal belongs to some MST. We leave this as an exercise. ■

3.3 OPTIMALITY CONDITION AND SENSITIVITY ANALYSIS

Suppose after we have found an MST, some of the arc weights change, as can happen in practice. Can we update the MST in some simple way (instead of finding it from scratch using one of the two algorithms of the last section)? We will see in this section that the answer is "yes", by appealing to an optimality condition for MST.

Imagine that a stranger approaches you with a spanning tree and claims it to be an MST. How would you check his/her claim? Let's revisit one of the MST we found for the weighted graph in Example 3.1.1 for ideas:

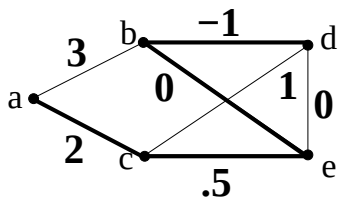


Figure 3.3.1

Notice that each arc not in the MST has the property that it's heavier (i.e., higher weight) than all arcs in the cycle formed by adding this arc to the MST. In particular, arc (a,b) forms a cycle with arcs $(b,e),(e,c),(c,a)$ whose weights $0,.5,2$ are all below $\omega_{ab}=3$; arc (c,d) forms a cycle with arcs $(d,b),(b,e),(e,c)$ whose weights $-1,0,.5$ are all below $\omega_{cd}=1$; arc (d,e) forms a cycle with arcs $(e,b),(b,d)$ whose weights $-1,0$ are all below $\omega_{de}=0$. In other words,

$$\begin{aligned} \omega_{ab} &\geq \max\{\omega_{ac},\omega_{ce},\omega_{be}\}, \\ \omega_{cd} &\geq \max\{\omega_{ce},\omega_{be},\omega_{bd}\}, \\ \omega_{de} &\geq \max\{\omega_{bd},\omega_{be}\}. \end{aligned} \tag{3.3.1}$$

Remarkably, the above condition is necessary and sufficient for the above spanning tree (shown darkened) to be an MST. We state and prove the general result below.

Fact 3.3.1 *(Optimality Condition for MST). Let $G=(V,A)$ be a graph with weights ω_{uv} for $(u,v)\in A$. A spanning tree $T=(V,B)$ of G is an MST if and only if*

$$\textit{for every } (u,v)\in A\backslash B, \quad \omega_{uv}\geq\omega_{wz} \quad \textit{for all arcs } (w,z) \textit{ in the simple path of } T \textit{ joining } u,v. \tag{3.3.2}$$

Proof. "If": This is left as an exercise.

"Only if": Let $T = (V, B)$ be a spanning tree that does not satisfy Eq. (3.3.2). Then there is some $(u, v) \in A \backslash B$ such that

$$\omega_{uv} < \omega_{wz} \quad \text{for some arc } (w, z) \text{ in the simple path of } T \text{ joining } u, v.$$

By Fact 3.1.2,

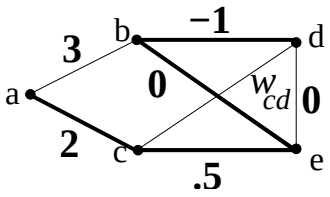
$$T' = (V, B') \qquad \text{with} \qquad B' = (B \cup \{(u, v)\}) \setminus \{(w, z)\}$$

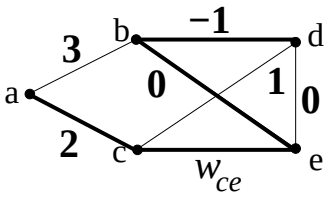
is also a spanning tree of G. Moreover,

$$\omega(T') \;=\; \omega(T) + \omega_{uv} - \omega_{wz} \;<\; \omega(T).$$

Thus T is not an MST. ■

Fact 3.3.1 is useful for sensitivity analysis of MST. For example, what is the range of ω_{cd} for which the spanning tree shown in Figure 3.3.1 (in dark) remains an MST? Since adding (c, d) forms a cycle with arcs $(c, e), (b, e), (b, d)$ in the spanning tree, Fact 3.3.1 implies we need $\omega_{cd} \geq \max\{.5, 0, -1\} = .5$ (also see Eq. (3.3.1)). If $\omega_{cd} < .5$, then (c, d) would replace (c, e) in an MST.





What is the range of ω_{ce} for which the spanning tree shown in Figure 3.3.1 remains an MST? Since (c, e) belongs to cycles formed by adding arc (a, b) or (c, d), Fact 3.3.1 implies we need $3 \geq \omega_{ce}, 1 \geq \omega_{ce}$ or, equivalently, $\omega_{ce} \leq \min\{3, 1\} = 1$ (also see Eq. (3.3.1)). If $\omega_{ce} > 1$, then (c, e) would be replaced by (c, d) in an MST.

Fact 3.3.1 also suggests an arc swapping algorithm for finding an MST, starting at any spanning tree:

MST-arcswap

Input: A connected graph $G = (V, A)$, with weights ω_{uv} for $(u, v) \in A$, and a spanning tree $T = (V, B)$.

Output: An MST.

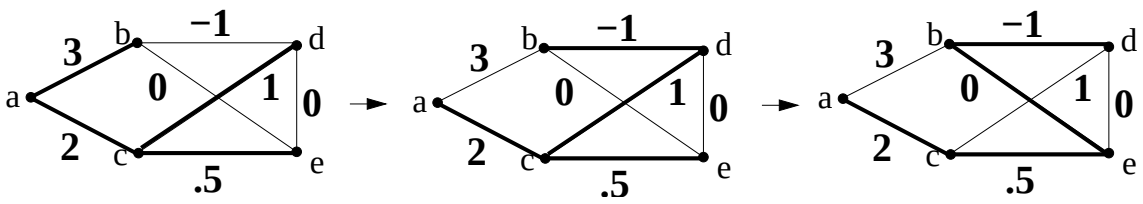
1. If there is some $(u, v) \in A \backslash B$ and some arc (w, z) in the simple path of T joining u, v such that $\omega_{uv} < \omega_{wz}$, then

$B \leftarrow (B \cup \{(u, v)\}) \setminus \{(w, z)\}$; return to Step 1.

Else

output $T = (V, B)$; stop.

For example, if we start with the spanning tree shown below on the left, then after swapping (b, d) with (a, b), and then (b, e) with (c, d), we find an MST. [Here, we choose the heaviest arc to swap out.]



What is the efficiency of MST-arcswap? This seems not well studied.

There is yet a fourth algorithm for finding an MST: Starting with $B = A$, successively remove the next heaviest arc from B if it does not disconnect (V, B), until no more arc can be removed. However, there is no fast way to check whether removing an arc will disconnect a graph, so this algorithm is little used.

Exercises.

3.3.1. This is an exercise on logic. Recall that the negation of the qualifiers "for all" is "for some (at least one)" and vice versa. The negation of "and" is "or" and vice versa. For example, the negation of "every person is happy every day" is "some person is unhappy some day" Give the negation of each of the following statements.

a) Every cycle has at least one arc with weight zero.

b) At least one cycle has an odd number of arcs.

c) At least one spanning tree has two vertices of degree 1 and no vertex of even degree.

3.3.2. Consider a graph $G = (V, A)$ with arc weights ω_{uv} for $(u, v) \in A$. Prove that a spanning tree $T = (V, B)$ is an MST if, for every $(u, v) \in A \setminus B$, we have $\omega_{uv} \geq \omega_{wz}$ for all arcs (w, z) in the simple path of T joining u, v. [Hint: Let $T' = (V, B')$ be any MST. Show that if $B' \neq B$, then we can replace some arc in T' with some arc in T to get another MST.]

3.3.3. $G = (V, A)$ is a connected graph with arc weights ω_{uv} for $(u, v) \in A$. Consider the problem of finding a spanning tree $T = (V, B)$ whose *max-weight* $\omega_{\max}(T) = \max_{(u,v)\in B} \omega_{uv}$ is minimum.

a) Prove that every MST T solves this problem, i.e., $\omega_{\max}(T)$ is minimum. [The argument is similar to Exercise 3.3.2.]

b) Is the converse true? In other words, if a spanning tree T has minimum max-weight, is T an MST? Explain.

3.3.4. $G = (V, A)$ is a connected graph with arc weights ω_{uv} for $(u, v) \in A$. Let $\phi : \Re \to \Re$ be any strictly increasing function, i.e., $a < b$ if and only if $\phi(a) < \phi(b)$. Can the MST change if we change the arc weights from ω_{uv} to $\phi(\omega_{uv})$ for all $(u, v) \in A$? Explain your answer.

3.4 ADDITIONAL TOPICS

There are many variants of the MST problem, some of which are much more difficult. For example, there may be an additional requirement that the spanning tree has at most k arcs joined to each vertex

(so the tree is not too "bushy"), where $k \geq 2$ is a fixed integer. However, in general, even finding such a spanning tree is a difficult problem, belonging to the so-called NP-hard problems.

Another famous variant of the MST problem is the so-called Steiner tree problem (named after a mathematician Steiner who had studied this kind of problem). In this problem, a subset of the vertices is designated as Steiner vertices, and the tree is required to connect only the non-Steiner vertices (possibly using some of the Steiner vertices) rather than all vertices. Such a tree is called a Steiner tree. The problem is to find a Steiner tree whose weight (sum of arc weights in the tree) is minimum. In the special case where all vertices are non-Steiner, this reduces to the MST problem. In general, the Steiner tree problem is much more difficult, belonging to the NP-hard problems.

4. Shortest Path in a Weighted Digraph

Whether it's driving to your grandma's house or routing traffic on the internet, finding a shortest path is one of the most often solved discrete optimization problems in practice. In this chapter we study its structures as well as popular algorithms for finding shortest paths.

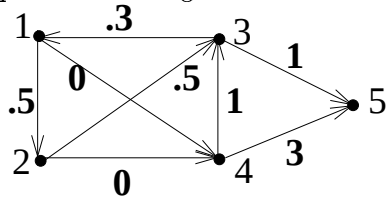
4.1 DEFINITIONS AND PROPERTIES

Let $G = (V, A)$ be a digraph, with a weight/length $\omega_{uv} \in \Re$ for each arc $(u, v) \in A$. The **weight** of a path $P:\ u_1, u_2, ..., u_{p+1}$ in G is defined to be the *sum* of the weight of all arcs in P, i.e.,

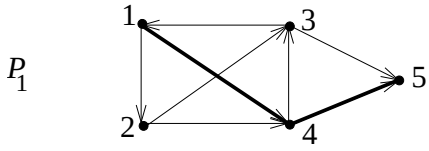
$$\omega(P) \ = \ \sum_{i=1}^{p} \omega_{u_i u_{i+1}} \ = \ \omega_{u_1 u_2} + \omega_{u_2 u_3} + \cdots + \omega_{u_p u_{p+1}}.$$

[Instead of sum, other operands such as "max" or $\cdot$ can also be used; see Exercise ??.] In applications, the arc weight ω_{uv} could be the cost/time/distance to travel from u to v.

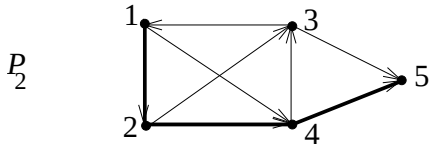
Example 4.1.1. Consider the digraph with arc weights shown below



So $\omega_{12} = .5$, $\omega_{14} = 0$, $\omega_{23} = .5$,..., etc. Two paths from vertex 1 to 5 (shown darkened) with their weights are



$w(P_1) = 0 + 3 = 3$



$w(P_2) = .5 + 0 + 3 = 3.5$

Other paths and their weights can be similarly computed.

Below we give some properties of $\omega(P)$ based on its definition as a sum of arc weights in P. For example, if we add 7 to all arc weights in Example 4.1.1, then the weight of the paths P_1 and P_2 change by 14 and 21 to, respectively, 17 and 24.5. Thus, unlike MST, the weight of the paths do not change by equal amount.

Fact 4.1.1. *Let $G = (V, A)$ be a digraph with weights ω_{uv} for $(u, v) \in A$.*

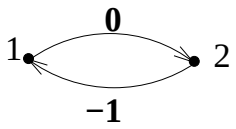
(a) If we add $\alpha \in \Re$ to ω_{uv} for all $(u,v) \in A$, then $\omega(P)$ changes by an additive factor of $(\#\text{arcs in } P) \cdot \alpha$ for all paths P in G.

(b) If we multiply $\alpha \in \Re$ to ω_{uv} for all $(u,v) \in A$, then $\omega(P)$ changes by a multiplicative factor of α for all paths P in G.

Proof. The proof is very similar to that of Fact 3.1.1 and is omitted.

In general there is more than one path from one vertex to another. We wish to find a **shortest path** (**SP**), i.e., a path of minimum weight, from a given vertex $s \in V$ to all vertices $v \in V$ reachable from s. This is sometimes called the one-to-all SP problem. [The problem of finding paths of maximum weight can be reduced to this problem by negating all arc weights.] For the weighted digraph in Example 4.1.1, it is not hard to convince ourselves that a SP from 1 to 5 is $1, 4, 3, 5$ with weight $\omega_{14} + \omega_{4,3} + \omega_{3,5} = 0 + 1 + 1 = 2$. This SP is not unique, however, as the path $1, 2, 3, 5$ also has weight 2. Fact 4.1.1(b) implies that we can multiply all arc weights by a *positive* constant without changing the SP(s). This is not true if we add a positive constant to all arc weights.

Unlike MST, negative arc weight can pose difficulties for SP. For example, the following digraph with arc weights shown



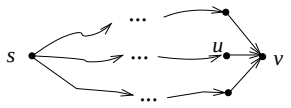
has a path $1, 2$ with weight 0. But $1, 2, 1, 2$ is also a path from vertex 1 to vertex 2, with weight -1. And $1, 2, 1, 2, 1, 2$ is another path from 1 to 2, with weight -2. In general, by going k times around the cycle $1, 2, 1$, the path weight would be $-k$, which tends to $-\infty$ as $k \to \infty$. In other words, the above weighted graph *has no SP* from 1 to 2, due to the presence of a cycle of negative weight. If there is no such cycle, then not only is there a SP (maybe more than one) from any vertex to any other vertex reachable from it, but there is at least one such SP that is simple (compare with Fact 2.2.1).

Fact 4.1.2. *Let $G = (V, A)$ be a digraph with arc weights ω_{uv} for $(u,v) \in A$, and $s \in V$. If there is no cycle of negative weight, then, for any $v \in V$ reachable from s by a path, there exists a SP from s to v that is simple.*

Proof. The proof is similar to that of Fact 2.2.1 and is left as an exercise.

Thus, under the assumption that there is no cycle of negative weight (equivalently, every cycle has positive or zero weight), our SP problem is a discrete optimization problem since there are finitely many simple paths to choose from. But explicitly search through them all would be slow since there can be many such paths (possibly exponential in $|V|$; see Exercise 3.1.2), so a more efficient algorithm is needed. We will study two such algorithms in the next section. These algorithm are based on a simple (but very useful) observation about SP, namely, if the shortest path from s to v goes through a predecessor vertex u, then the

portion of this path from s to u must be shortest, and u minimizes the weight of this path plus the weight of the arc to v.



More precisely, suppose

$$d_v \;=\; (\text{weight of SP from } s \text{ to } v) \quad \text{for all} \quad v \in V \tag{4.1.1}$$

(with $d_v = \infty$ if v is not reachable from s). If $s, ..., u, v$ is a SP from s to v, then $s, ..., u$ is a SP from s to u and $d_v = d_u + \omega_{uv}$. Moreover,

$$d_s = 0 \qquad \text{and} \qquad d_v \;=\; \min_{u:(u,v)\in A} (d_u + \omega_{uv}) \quad \text{for all} \quad v \in V \setminus \{s\}. \tag{4.1.2}$$

In Example 4.1.1, starting at $s = 1$, we have $d_1 = 0$ and it can be checked that $d_2 = .5$, $d_3 = 1$, $d_4 = 0$, $d_5 = 2$. Thus $d_2 = \min(d_1 + \omega_{12})$, $d_3 = \min(d_2 + \omega_{23}, d_4 + \omega_{43})$, $d_4 = \min(d_1 + \omega_{14})$, $d_5 = \min(d_3 + \omega_{35}, d_4 + \omega_{45}\}$. This provides a very useful characterization of SP weights, which we state below without proof.

Fact 4.1.3 *(Optimality Condition for SP). Let $G = (V, A)$ be a digraph with weights ω_{uv} for $(u, v) \in A$, and $s \in V$. Any nonnegative d_u (possibly $d_u = \infty$), $u \in V$, satisfy (4.1.1) if and only if they satisfy (4.1.2).*

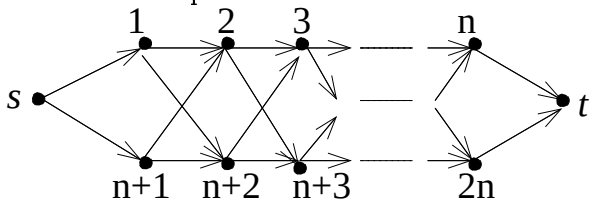
Thus, if a stranger claims to have found the SP weights, we can easily check his/her claim by checking that the weights satisfy the equations (4.1.2).

Exercises.

4.1.1.

a) If all arcs of a digraph have different nonnegative weights, is the SP from a vertex s to another vertex t necessarily unique? Explain.

b) It is not efficient to consider all possible paths from s to t in search of a SP from s to t. For the digraph shown below, find the number of distinct paths from s to t.



4.1.2. We wish to find the shortest route from point A to point B in the 2-dimensional plane, but there are obstacles (in the shape of polygons) in our way. Formulate this as a SP problem on a suitable weighted digraph. [Hint: The corner points of the polygons correspond to the vertices of the digraph.]

4.2 DIJKSTRA AND BELLMAN-FORD ALGORITHMS

A popular algorithm for the one-to-all SP problem was suggested by E.W. Dijkstra in a 1959 paper. It builds the paths one arc at a time, like Prim's algorithm for MST. Its main restriction is that all arc weights should be nonnegative.

SP-Dijkstra

Input: A digraph $G = (V, A)$, with *nonnegative* weights ω_{uv} for $(u, v) \in A$, and $s \in V$.

Output: For every $v \in V$ reachable from s by a path, a SP from s to v and its weight d_v.

0. Initialize $W \leftarrow \{s\}$, $B \leftarrow \emptyset$, $d_s \leftarrow 0$.

1. Choose any $(u, v) \in A$ with $u \in W$, $v \notin W$, and whose $d_u + \omega_{uv}$ is minimum among all such arcs. [If no such arc exists, the vertices in $V \setminus W$ are not reachable from s; stop.]

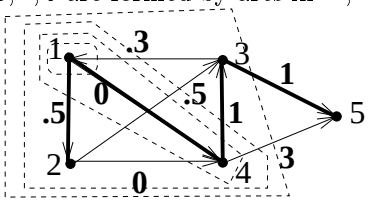
Update $W \leftarrow W \cup \{v\}$, $B \leftarrow B \cup \{(u, v)\}$, $d_v \leftarrow d_u + \omega_{uv}$.

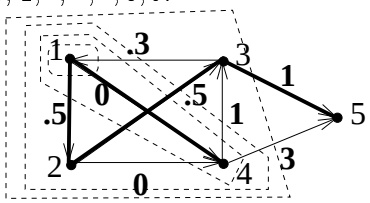
Return to Step 1.

If we apply SP-Dijkstra to the weighted digraph in Example 4.1.1, starting at vertex 1, then arcs may be added in the following order:

$\min\{d_u + \omega_{uv} \mid u \in W, v \notin W\}$	W	B	d_v
	$\{1\}$	$\emptyset$	$d_1 \leftarrow 0$
$\min\{\underbrace{0+.5}_{(1,2)}, \underbrace{0+0}_{(1,4)}\} = 0$	$\{1,4\}$	$\{(1,4)\}$	$d_4 \leftarrow 0$
$\min\{\underbrace{0+.5}_{(1,2)}, \underbrace{0+1}_{(4,3)}, \underbrace{0+3}_{(4,5)}\} = .5$	$\{1,4,2\}$	$\{(1,4),(1,2)\}$	$d_2 \leftarrow .5$
$\min\{\underbrace{0+1}_{(4,3))}, \underbrace{0+3}_{(4,5)}, \underbrace{.5+.5}_{(2,3)}\} = 1$	$\{1,4,2,3\}$	$\{(1,4),(1,2),(4,3)\}$	$d_3 \leftarrow 1$
$\min\{\underbrace{0+3}_{(4,5)}, \underbrace{1+1}_{(3,5)}\} = 2$	$\{1,4,2,3,5\}$	$\{(1,4),(1,2),(4,3),(3,5)\}$	$d_5 \leftarrow 2$

and the resulting B is shown below on the left, with each W set circled by dotted line. The SPs from 1 to vertices $2, 3, 4, 5$ are formed by arcs in B, respectively, $1, 2$; $1, 4, 3$; $1, 4$; $1, 4, 3, 5$.





In this example, the SPs from vertex 1 to 3 is not unique, and SP-Dijkstra can alternatively use arc $(2, 3)$ instead of $(4, 3)$, resulting in the SPs shown on the right. Which SPs are found depends on the mechanism used to break tie when the minimum in Step 1 is achieved by more than one arc. Notice that the arcs of the SPs form a spanning tree when arc directions are ignored. Also, SPs are found in order of their weights, with the shortest paths found first. If we want only an SP from s to a particular vertex t, then we can stop SP-Dijkstra early whenever t is reached.

How efficient is SP-Dijkstra? It searches at most $|A|$ arcs per Step 1, and repeats Step 1 at most $|V| - 1$ times. Thus the overall effort is about $(|V| - 1)|A|$ operations, which is $O(|V||A|)$. By storing and updating the quantity $d_v = \min_{u \in W:(u,v) \in A} \{d_u + \omega_{uv}\}$ for all $v \notin W$, the work can be reduced to $O(|V|^2)$

operations. Specifically, in Step 1, choose $v \notin W$ whose d_v is minimum, and update $W \leftarrow W \cup \{v\}$, $d_w \leftarrow \min\{d_w, d_v + \omega_{vw}\}$ for all $w \notin W$ with $(v,w) \in A$.

We show below that SP-Dijkstra indeed works correctly. The proof uses Fact 3.1.3. If G is not connected, then MST-Prim will find an MST for the connected component that contains the starting vertex r.

Fact 4.2.1. *SP-Dijkstra outputs d_v as the weight of SP from s to every vertex $v \in V$ reachable from s by a path.*

Proof. We will prove by induction that in Step 1 we always have

$$d_u \;=\; (\text{weight of SP from } s \text{ to } u) \quad \text{for all} \quad u \in W. \tag{4.2.1}$$

Initially $d_s = 0$, $W = \{s\}$, so (4.2.1) is true.

Suppose, on entering Step 1, (4.2.1) is true. We now show that (4.2.1) is true when v is added to W, i.e.,

$$d_v \;=\; (\text{weight of SP from } s \text{ to } v), \tag{4.2.2}$$

where (u,v) is the chosen arc in Step 1. Since $u \in W$, (4.2.1) implies d_u is the weight of a SP from s to u:

$$s = u_1^*, u_2^*, ..., u_{p^*+1}^* = u$$

where $p^* \geq 0$, $(u_i^*, u_{i+1}^*) \in A$ for $i = 1, ..., p^*$. Then $d_v = d_u + \omega_{uv}$ is the weight of the path from s to v:

$$P^* : \quad u_1^*, u_2^*, ..., u_{p^*+1}^*, v.$$

Consider any path P from s to v:

$$P : \quad s = u_1, u_2, ..., u_{p+1} = v,$$

where $p \geq 0$, $(u_i, u_{i+1}) \in A$ for $i = 1, ..., p$. Since $s \in W$ and $v \notin W$, the path P must contain an arc from W to $V \setminus W$, i.e., there exists $1 \leq i \leq p$ such that

$$u_i \in W, \qquad u_{i+1} \notin W. \tag{4.2.3}$$

Let

$$P_1 : \; u_1, ..., u_i$$

$$P_2 : \; u_{i+1}, ..., u_{p+1}.$$

Then

$$\omega(P) \;=\; \omega(P_1) + \omega_{u_i u_{i+1}} + \omega(P_2).$$

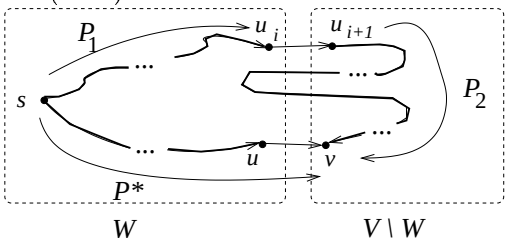
Since P_1 is just a path from s to u_i, its weight is at least that of a SP from s to u_i, which by (4.2.1) is equal to d_{u_i} (since $u_i \in W$). Thus $\omega(P_1) \geq d_{u_i}$. Also, since all arc weights are nonnegative, $\omega(P_2) \geq 0$. Thus we conclude that

$$\omega(P) \;\geq\; d_{u_i} + \omega_{u_i u_{i+1}}.$$

Since (4.2.3) holds, our choice of the arc (u, v) in Step 1 implies the right-hand side is not less than $d_u + \omega_{uv}$, which we showed earlier to equal $\omega(P^*)$. Thus

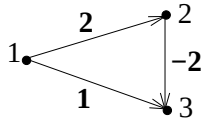
$$\omega(P) \geq \omega(P^*).$$

Since this is true for any path P from s to v, and P^* is itself a path from s to v, P^* must be a SP from s to v. Since $d_v = \omega(P^*)$, this proves (4.2.2).



■

If there are arcs with negative weight, even if no cycle of negative weight, SP-Dijkstra can fail to find the correct SP. Take the digraph with arc weights shown:



If we apply SP-Dijkstra to this example with $s = 1$, it would output $d_3 = 1$, which is not the correct SP weight to vertex 3. The problem here is that Dijkstra's algorithm is in some sense myopic and cannot look far ahead enough to take into account negative arc weights. It is sometimes called "label-fixing", as it fixes the path weights in increasing order. We will next look at a second popular algorithm, attributed to R.E. Bellman (1958) and L.R. Ford (1956), that uses "label-correction" via Fact 4.1.3(b) to avoid this pitfall.

SP-Bellman-Ford

Input: A digraph $G = (V, A)$, with weights ω_{uv} for $(u, v) \in A$, having no cycle of negative weight, and $s \in V$.

Output: For every $v \in V$ reachable from s by a path, the weight d_v of SP from s to v.

0. $d_s \leftarrow 0$. $d_v \leftarrow \infty$ for all $v \in V \setminus \{s\}$.

1. For each $v \in V \setminus \{s\}$, chosen in any order, update

$$d_v \leftarrow \min_{u:(u,v)\in A} (d_u + \omega_{uv}).$$

If no distance label changes, stop. Else return to Step 1.

If we apply SP-Bellman-Ford to the weighted digraph in Example 4.1.1, the following distance labels may be generated:

0. $d_1 \leftarrow 0$, $d_2 \leftarrow \infty, d_3 \leftarrow \infty, ..., d_5 \leftarrow \infty$.
1. $d_2 \leftarrow \min\{d_1 + .5\} = \min\{.5\} = .5$
   $d_3 \leftarrow \min\{d_2 + .5, d_4 + 1\} = \min\{1, \infty\} = 1$

$d_4 \leftarrow \min\{d_1 + 0, d_2 + 0\} = \min\{0, .5\} = 0$
$d_5 \leftarrow \min\{d_3 + 1, d_4 + 3\} = \min\{2, 3\} = 2$

1. $d_2 \leftarrow \min\{d_1 + .5\} = \min\{.5\} = .5$
   $d_3 \leftarrow \min\{d_2 + .5, d_4 + 1\} = \min\{1, 1\} = 1$
   $d_4 \leftarrow \min\{d_1 + 0, d_2 + 0\} = \min\{0, .5\} = 0$
   $d_5 \leftarrow \min\{d_3 + 1, d_4 + 3\} = \min\{2, 3\} = 2$

1. No further change in distance labels, stop.

The shortest paths can be found among the arcs that achieve the minimum. In the above example, the minimum in computing d_2 is achieved by the arc $(1, 2)$, so this arc is on the SP from 1 to 2. The minimum in computing d_3 is achieved by the arc $(2, 3)$ or $(4, 3)$, so either of these arcs is on a SP from 1 to 3, and so on.

Since SP-Bellman-Ford stops only when no distance label changes, the distance labels must satisfy (4.1.2) on stopping and hence, by Fact 4.1.3, are SP weights. Thus, it suffices to show that SP-Bellman-Ford eventually stops, which we do below.

Fact 4.2.2. *SP-Bellman-Ford stops after at most* $|V| - 1$ *iterations.*

Proof (idea). The key step in the proof is to show by induction that, for all vertices v that has a SP from s with at most k arcs, their distance labels would stop changing after k iterations (and possibly earlier). [In the above example, vertices 2 and 4 have SP from 1 with one arc, so their distance labels would stop changing after the 1st iteration. Vertex 3 has SP from 1 with two arcs, so its distance label would stop changing after the 2nd iteration, and so on.] ■

Since each iteration of SP-Bellman-Ford involves $O(|A|)$ operations, Fact 4.2.2 shows that the overall effort for SP-Bellman-Ford is $O(|V||A|)$ operations. In fact, we can run SP-Bellman-Ford without knowing a priori whether the digraph has a cycle of negative weight. We simply run the algorithm for $|V|$ iterations. If the distance labels stop changing after $|V| - 1$ iterations, then these are the SP weights. Otherwise Fact 4.2.2 shows that the digraph has a cycle of negative weight. With some refinement, the algorithm can even find one such cycle. The ability to detect (and possibly find) negative-weight cycle is a key advantage of the Bellman-Ford algorithm. And in contrast to the Dijkstra algorithm, Bellman-Ford does not require all arc weights to be nonnegative. For example, the home rental problem of Exercise 2.2.3 can be formulated as a SP problem by negating the arc weights. The digraph has no cycle, so it has no cycle of negative weight, even though the arc weights are all negative. Thus we can apply SP-Bellman-Ford to solve this SP problem.

Similar to Section 3.3, we can use the optimaly conditions of Fact 4.1.3 for sensitivity analysis. For example, the arc $(2, 4)$ in Example 4.1.1 does not belong to any SP starting from vertex 1. What if we decrease ω_{24}? From the optimality condition

$$d_4 = \min\{d_1 + \omega_{14}, d_2 + \omega_{24}\} = \min\{0, .5 + \omega_{24}\},$$

we see that $(2,4)$ remains outside of any SP as long as $0 < .5 + \omega_{24}$ or, equivalently, $\omega_{24} > -.5$. Similarly, the arc $(1,4)$ belongs to an SP. What if we increase ω_{14}? This will affect the weight of all SPs that use the arc $(1,4)$. From the optimality conditions for these SP weights:

$$d_4 = \min\{d_1 + \omega_{14}, d_2 + \omega_{24}\} = \min\{\omega_{14}, .5\},$$
$$d_3 = \min\{d_2 + \omega_{23}, d_4 + \omega_{43}\} = \min\{1, d_4 + 1\},$$
$$d_5 = \min\{d_3 + \omega_{35}, d_4 + \omega_{45}\} = \min\{d_3 + 1, d_4 + 3\},$$

we see that $(1,4)$ would not be on any SP from 1 to 3 or 5 whenever $\omega_{14} > 0$ (since d_3 remains at 1). It also would not be on any SP from 1 to 4 whenever $\omega_{14} > .5$.

Exercises.

4.2.1. Consider a digraph $G = (V, A)$ with nonnegative arc weights ω_{uv} for $(u, v) \in A$, and $s \in V$. For any path from s to $v \in V$, define its *max-weight* to be the maximum of the weight of all arcs in the path. We wish to find, for each $v \in V$ reachable from s, a path from s to v whose max-weight is minimum.

a) Show by example that a SP need not have minimum max-weight nor vice versa.

b) How would you modify SP-Dijkstra to find a path of minimum max-weight?

4.3 ADDITIONAL TOPICS

Both SP-Dijkstra and SP-Bellman-Ford find SP from one vertex s to all other vertices. In some situations, we may wish to find a SP from every vertex to every other vertex. Of course, we can simply run SP-Dijkstra or SP-Bellman-Ford $|V|$ times, each time with a different start vertex s. However, there is an elegant alternative algorithm, attributed to R.W. Floyd and S. Warshall in 1962, which we describe now.

Let us label the vertices from 1 to n, i.e., $V = \{1, ..., n\}$. Denote

$$d_{ij}^k \;=\; \text{weight of SP from } i \text{ to } j \text{ using only } 1, ..., k \text{ as intermediate vertices,} \quad \text{for } i, j \in V.$$

Then

$$d_{ij}^0 = \begin{cases} \omega_{ij} & \text{if } (i,j) \in A \\ 0 & \text{if } i = j \\ \infty & \text{else.} \end{cases}$$

Moreover, any SP from i to j using only $1, ..., k+1$ as intermediate vertices either (i) does not use $k+1$ or (ii) is made up of a SP from i to $k+1$ and a SP from $k+1$ to j. This means that

$$d_{ij}^{k+1} = \min\{d_{ij}^k, d_{i\ k+1}^k + d_{k+1\ j}^k\}.$$

The Floyd-Warshall algorithm simply applies the above recursive formula for $k = 0, 1, ..., n-1$ to compute d_{ij}^n for $i, j \in V$.

Let D^k be the $n \times n$ matrix with d_{ij}^k as its (i,j)th entry. Then for the weighted digraph of Example 4.1.1, we have that

$$D^0 = \begin{pmatrix} 0 & .5 & \infty & 0 & \infty \\ \infty & 0 & .5 & 0 & \infty \\ .3 & \infty & 0 & \infty & 1 \\ \infty & \infty & 1 & 0 & 3 \\ \infty & \infty & \infty & \infty & 0 \end{pmatrix}, \quad D^1 = \begin{pmatrix} 0 & .5 & \infty & 0 & \infty \\ \infty & 0 & .5 & 0 & \infty \\ .3 & .8 & 0 & .3 & 1 \\ \infty & \infty & 1 & 0 & 3 \\ \infty & \infty & \infty & \infty & 0 \end{pmatrix}, \quad \cdots \quad D^5 = \begin{pmatrix} 0 & .5 & 1 & 0 & 2 \\ .8 & 0 & .5 & 0 & 1.5 \\ .3 & .8 & 0 & .3 & 1 \\ 1.3 & 1.8 & 1 & 0 & 2 \\ \infty & \infty & \infty & \infty & 0 \end{pmatrix}$$

[When programming this on a computer, "∞" would be replaced by a very large constant.] The overall effort for the Floyd-Warshall algorithm is $O(|V|^3)$ operations. With some refinement, this algorithm can also detect cycles with negative weight.

We saw in Section 4.1 that if there is a cycle of negative weight, then there may not exist a path of minimum weight. What if we instead wish to find a *simple* path of minimum weight? Since there is only a finite number of simple paths between two vertices, such a shortest simple path always exists. However, it turns out that this problem is much more difficult to solve. In fact, like the traveling salesman problem, it belongs to the class of NP-hard problems!

In general, there is often a fine line between "easy" problems (such as MST and SP) and more difficult problems.

5. Flows, Cuts, and Matchings

In this chapter, we study optimization problems that can be interpreted as optimizing flows on a capacitated digraph. For example, we may wish to send as much flow as possible from a source to a sink or send flow at minimum cost. Such problems have applications to routing of data or packages or vehicles on networks, as well as to matching people to tasks, for examples.

5.1 MAXIMUM FLOW: DEFINITIONS AND PROPERTIES

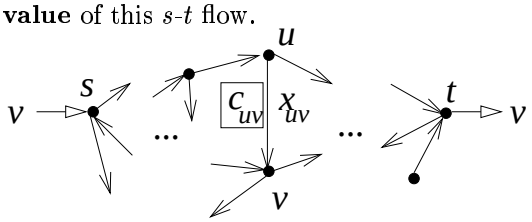
Suppose we wish to send goods over a road network or a rail network from a source to a sink. Or we wish to transmit data over a data network. Each link in the network has a capacity/limit on the rate of transmission (amount of goods/data that can be sent per time unit). What is the maximum rate of transmission? This can be modeled (in simplest form) as a maximum flow problem, which we describe below.

Let $G = (V, A)$ be a digraph, with a nonnegative number c_{uv} ("capacity") for each arc $(u, v) \in A$ (possibly $c_{uv} = \infty$). Let s and t be two distinct vertices in V. An **s-t flow** on G is a vector of real numbers $(x_{uv})_{(u,v)\in A}$ ("arc flows") satisfying

$$0 \le x_{uv} \le c_{uv} \qquad \text{for all } (u,v) \in A, \qquad \text{"capacity"}$$

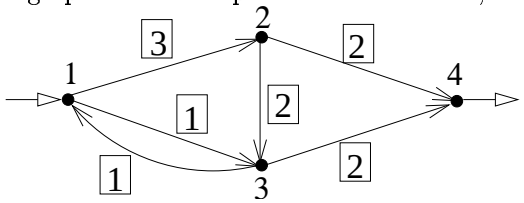
$$\sum_{v:(u,v)\in A} x_{uv} - \sum_{v:(v,u)\in A} x_{vu} = \begin{cases} \vartheta & \text{if } u = s \\ -\vartheta & \text{if } u = t \\ 0 & \text{else} \end{cases} \qquad \text{for all } u \in V, \qquad \text{"flow conservation"}$$

for some $\vartheta \in \Re$. We call ϑ the **value** of this s-t flow.



The capacity constraints require the flow on each arc to be nonnegative and below the arc capacity. The flow conservation constraints require the total flow out of each vertex to equal the total flow into that vertex, with an exogenous inflow of ϑ at the source s and an exogenous outflow of ϑ at the sink t. We wish to find an s-t flow whose value is maximum. This is the **maximum flow problem**, which was first studied by L.R. Ford and D.R. Fulkerson.

Example 5.1.1. Consider the digraph with arc capacities shown boxed, with $s = 1$ and $t = 4$:



So $c_{12} = 3$, $c_{13} = 1$, $c_{23} = 2$,..., etc. [The arc capacities need not be integer in general.] The capacity and flow conservation constraints are:

$$0 \le x_{12} \le 3, \quad 0 \le x_{13} \le 1, \quad 0 \le x_{23} \le 2, \quad 0 \le x_{24} \le 2, \quad 0 \le x_{31} \le 1, \quad 0 \le x_{34} \le 2,$$

$$\begin{array}{rrrrrrr} x_{12} & +x_{13} & & & -x_{31} & & = \vartheta \\ -x_{12} & & +x_{23} & +x_{24} & & & = 0 \\ & -x_{13} & -x_{23} & & +x_{31} & +x_{34} & = 0 \\ & & & -x_{24} & & -x_{34} & = -\vartheta \end{array}$$

Some examples of 1-4 flows and their values are listed below:

$(x_{12}, x_{13}, x_{23}, x_{24}, x_{31}, x_{34})$	ϑ
$(0, 0, 0, 0, 0, 0)$	0
$(2, 0, 0, 2, 0, 0)$	2
$(2, 1, 0, 2, 0, 1)$	3
$(2.5, 1, .5, 2, 0, 1.5)$	3.5
$(3, 1, 1, 2, 0, 2)$	4

Is 4 the maximum value of 1-4 flows? How can we check this systematically?

Notice in Example 5.1.1 that the capacity constraints are linear inequalities and the the flow conservation constraints are linear equations in the variables $x_{12}, x_{13}, ..., x_{34}, \vartheta$. Also ϑ is a linear function of these variables. Thus, the maximum flow problem is a special case of a linear program (LP), whereby we wish to maximize a linear function subject to the variables satisfying a (finite) system of linear inequalities and equations. [Thus the maximum flow problem is a discrete optimization problem in the sense that if an LP has an optimal solution, then it has an optimal solution that is a "corner point" of the feasible region, and the number of such "corner points" is finite.] We might recall from duality theory for LP that every LP has a "dual" which has the same optimal cost/objective value. Moreover, solving the LP also solves its dual. The dual of a maximum flow problem has very interesting graphical properties, which we will now study and then use to solve the maximum flow problem.

For any $S \subseteq V$ and $T \subseteq V$, we denote

$$[S, T] = \{(u, v) \in A \ : \ u \in S, \ v \in T\},$$

i.e., $[S, T]$ comprises those arcs in A directed from vertices in S to vertices in T. For any $S \subseteq V$ that contains s but not t (i.e., $s \in S$, $t \notin S$), we denote

$$c[S] = \sum_{(u,v) \in [S, V \setminus S]} c_{uv},$$

i.e., $c[S]$ is the sum of capacities of arcs directed from S to its complement $V \setminus S$. $[S, V \setminus S]$ is sometimes called an **s-t cut** since deleting these arcs would cut off all connections between s and t, and $c[S]$ is called the "capacity" of this cut (which depends on S). We wish to find an s-t cut whose capacity is minimum. This is the **minimum cut problem**. Notice that this is a discrete optimization problem since the number of s-t cuts is finite. For Example 5.1.1, the 1-4 cuts and their capacities are listed below.

S	$[S, V \setminus S]$	$c[S]$

$\{1\}$	$\{(1,2),(1,3)\}$	$3+1=4$
$\{1,2\}$	$\{(1,3),(2,3),(2,4)\}$	$1+2+2=5$
$\{1,3\}$	$\{(1,2),(3,4)\}$	$3+2=5$
$\{1,2,3\}$	$\{(2,4),(3,4)\}$	$2+2=4$

Is 4 the maximum value of 1-4 cuts? How can we check this systematically?

Notice that the value of each 1-4 flow is below the capacity of each 1-4 cut in Example 5.1.1. Is this a coincidence? [Recall weak duality for LP and its dual.] Observe that the flow conservation constraints have exactly two nonzero coefficients, a 1 and a -1, in each left-hand column. Linear equations of this type have special properties. In particular, for any $S \subseteq V$, if we sum the flow conservation constraints over all $u \in S$, then each term of the form x_{uv} with $u, v \in S$ will appear twice in the sum, once with a plus sign and once with a minus sign, and hence they cancel. This leaves only terms of the form x_{uv} with $u \in S, v \notin S$ (with a plus sign) and x_{vu} with $u \in S, v \notin S$ (with a minus sign). For example, if we sum the flow conservation constraints in Example 5.1.1 over vertices in $S = \{1,2\}$, we obtain

$$x_{13} + x_{23} + x_{24} - x_{31} = \vartheta.$$

Since any 1-4 flow satisfies $x_{13} \leq c_{13}$, $x_{23} \leq c_{23}$, $x_{24} \leq c_{24}$, and $x_{31} \geq 0$, the left-hand side is below $c_{13} + c_{23} + c_{24} - 0$, which exactly equals $c[\{1,2\}]$. Thus $c[\{1,2\}] \geq \vartheta$. If we instead sum over vertices in $S = \{1,2,3\}$, we obtain

$$x_{24} + x_{34} = \vartheta.$$

By a similar reasoning, the left-hand side is below $c_{24} + c_{34}$, which exactly equals $c[\{1,2,3\}]$. Thus $c[\{1,2,3\}] \geq \vartheta$. This can be generalized to the following key fact.

Fact 5.1.1 *(Weak Duality). For any s-t flow $(x_{uv})_{(u,v)\in A}$ with value ϑ, and any $S \subseteq V$ with $s \in S, t \notin S$, we have $c[S] \geq \vartheta$.*

Proof. Since $(x_{uv})_{(u,v)\in A}$ satisfies flow conservation at each vertex $u \in V$, summing over all $u \in S$ yields

$$\sum_{u \in S} \left(\sum_{v:(u,v)\in A} x_{uv} - \sum_{v:(v,u)\in A} x_{vu} \right) = \vartheta.$$

We get ϑ on the right-hand side because S contains s but not t, so the right-hand side is the sum of ϑ and a bunch of zeros. Thus

$$\begin{aligned}
\vartheta &= \sum_{u \in S} \sum_{v:(u,v)\in A} x_{uv} - \sum_{u \in S} \sum_{v:(v,u)\in A} x_{vu} \\
&= \sum_{(u,v)\in[S,V]} x_{uv} - \sum_{(v,u)\in[V,S]} x_{vu} \\
&= \left(\sum_{(u,v)\in[S,S]} x_{uv} + \sum_{(u,v)\in[S,V\setminus S]} x_{uv} \right) - \left(\sum_{(v,u)\in[S,S]} x_{vu} + \sum_{(v,u)\in[V\setminus S,S]} x_{vu} \right) \\
&= \sum_{(u,v)\in[S,V\setminus S]} x_{uv} - \sum_{(v,u)\in[V\setminus S,S]} x_{vu}.
\end{aligned}$$

Since $x_{uv} \leq c_{uv}$ for all $(u,v) \in [S, V \setminus S]$ and $x_{vu} \geq 0$ for all $(v,u) \in [V \setminus S, S]$, the right-hand side is below $\sum_{(u,v) \in [S, V \setminus S]} c_{uv} - 0 = c[S]$. Thus $\vartheta \leq c[S]$. ■

Fact 5.1.1 implies that if an s-t flow has a value *equal* to the capacity of an s-t cut, then this flow value must be maximum (since no flow value can exceed the cut capacity) and this cut capacity must be minimum (since no cut capacity can be less than the flow value). In the next section, we will describe an algorithm for finding such flow and cut.

Exercises.

5.1.1.

a) Give an example of a digraph $G = (V, A)$ with *integer* arc capacities and two vertices s, t such that $|V| \leq 4$ and the s-t flow of maximum value is not unique.

b) For your answer to (a). Find an s-t flow of maximum value whose arc flows are *not* all integer.

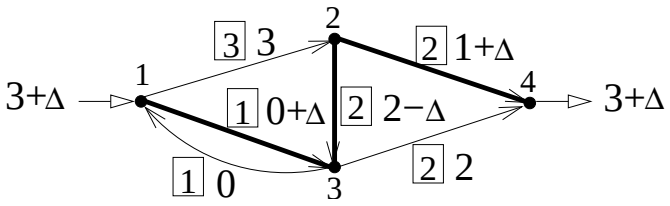
c) Give an example of a digraph $G = (V, A)$ with arc capacities of 1 and two vertices s, t such that $|V| = 3$ and the s-t cut of minimum capacity is not unique.

5.1.2. Prove that if $\{x_{uv}\}_{(u,v) \in A}$, is an s-t flow of value ϑ and $[S, V \setminus S]$ is an s-t cut of capacity c, then $v = c$ if and only if $x_{uv} = c_{uv}$ for every $(u,v) \in [S, V \setminus S]$ and $x_{vu} = 0$ for every $(v,u) \in [V \setminus S, S]$. [You can use the fact that $\vartheta = \sum_{(u,v) \in [S, V \setminus S]} x_{uv} - \sum_{(v,u) \in [V \setminus S, S]} x_{vu}$ for any s-t flow $\{x_{uv}\}_{(u,v) \in A}$ of value ϑ.]

5.1.3. Consider a digraph $G = (V, A)$ with arc capacities c_{uv}, $(u,v) \in A$, and $s \neq t \in V$. Suppose that $[S_1, T_1]$ and $[S_2, T_2]$ are two s-t cuts of minimum capacity, where $T_1 = V \setminus S_1$, $T_2 = V \setminus S_2$. Prove that $[S_1 \cap S_2, T_1 \cup T_2]$ and $[S_1 \cup S_2, T_1 \cap T_2]$ are also s-t cuts of minimum capacity. [Hint: First show that they are s-t cuts. Then show that their capacities equal that of $[S_1, T_1]$ and $[S_2, T_2]$ by considering arcs out of $S_1 \cap S_2$, $S_1 \setminus S_2$, $S_2 \setminus S_1$.]

5.2 MAXIMUM FLOW: FLOW AUGMENTATION ALGORITHM

An intuitively reasonable way to solve the max flow problem is to start with any s-t flow (e.g., zero flow on all arcs), find a "path" along which more flow can be sent from s to t to obtain a new s-t flow, and so on, until no more flow can be sent. However, care is needed in finding such a "path". To illustrate, consider Example 5.1.1 again. An 1-4 flow for its capacitated digraph is $x_{12} = 3$, $x_{23} = x_{34} = 2$, $x_{24} = 1$, $x_{13} = x_{31} = 0$, with value 3. The only way we can send more flow from 1 to 4 (and still satisfy flow conservation at all vertices) is by increasing flow on the arcs $(1,3), (2,4)$ and decreasing flow on the arc $(2,3)$. This is shown in the figure below, where Δ is the increase in flow value.



To satisfy the capacity constraints on these arcs, we need $0 \le 0+\Delta \le 1$, $0 \le 2-\Delta \le 2$, $0 \le 1+\Delta \le 2$, so the largest Δ is

$$\Delta = \min\{1, 2, 1\} = 1.$$

After the flow change, the new 1-4 flow is $x_{12} = 3$, $x_{13} = x_{23} = 1$, $x_{24} = x_{34} = 2$, $x_{31} = 0$, with value 4.

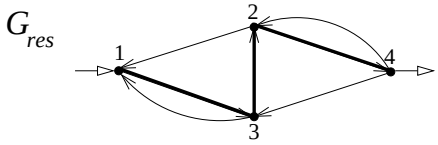
The above example shows that, in order to send more flow, we need to consider path from s to t that has not only forward arcs along which we can increase flow, but also backward arcs along which we can decrease flow! Formally, we say that a sequence of vertices $s = u_1, u_2, ..., u_{p+1} = t$ $(p \ge 1)$ is an s-t **augmenting path** relative to an s-t flow $(x_{uv})_{(u,v)\in A}$ if

$$\text{either} \quad (u_i, u_{i+1}) \in A,\ x_{u_i u_{i+1}} < c_{u_i u_{i+1}} \quad \text{or} \quad (u_{i+1}, u_i) \in A,\ x_{u_{i+1} u_i} > 0, \quad i = 1, 2, ..., p.$$

Such an augmenting path corresponds to a path (in the usual sense) in the **residual digraph** relative to $(x_{uv})_{(u,v)\in A}$, defined as

$$G_{\text{res}} = (V, A_+ \cup A_-) \quad \text{with} \quad \begin{array}{l} A_+ = \{(u,v) : (u,v) \in A,\ x_{uv} < c_{uv}\}, \\ A_- = \{(u,v) : (v,u) \in A,\ x_{vu} > 0\}. \end{array}$$

[In words, G_{res} is obtained from G by reversing the direction of any arc whose flow is at capacity and adding an opposite direction arc for any arc whose flow is positive and not at capacity.] In the above example, the 1-4 augmenting path is $1, 3, 2, 4$, which corresponds to a path from 1 to 4 in the residual digraph G_{res} below (in fact, the only such path):



Thus we can find an s-t augmenting path on G by finding a path from s to t in G_{res}. Such a path can be found using, say, a SP algorithm (Section 4.2) with the arc weights of G_{res} set to zero (or one). We formally describe the algorithm below, attributed to Ford and Fulkerson in 1956.

MAXFLOW

Input: A digraph $G = (V, A)$, with nonnegative capacities c_{uv} for $(u, v) \in A$, and distinct $s, t \in V$.

Output: An s-t flow of maximum value and an s-t cut of minimum capacity.

0. Initialize $x_{uv} \leftarrow 0$ for $(u, v) \in A$. $\vartheta \leftarrow 0$.

1. Find an s-t augmenting path P relative to $(x_{uv})_{(u,v)\in A}$ and augment flow:

$$\begin{array}{ll} x_{uv} \leftarrow x_{uv} + \Delta & \text{for } (u,v) \in A_+ \text{ in } P \\ x_{vu} \leftarrow x_{vu} - \Delta & \text{for } (u,v) \in A_- \text{ in } P \end{array} \qquad \vartheta \leftarrow \vartheta + \Delta,$$

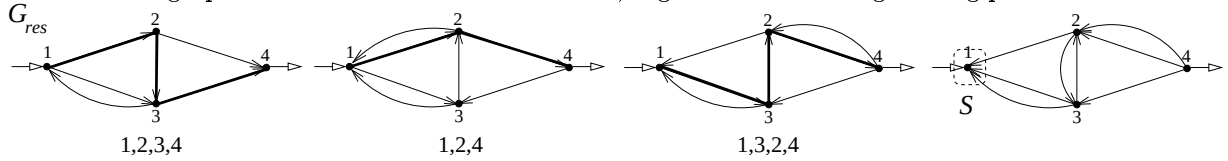
where $\Delta = \min\left\{ \min_{(u,v)\in A_+ \text{ in } P} c_{uv} - x_{uv} \ , \ \min_{(u,v)\in A_- \text{ in } P} x_{vu} \right\}$; return to Step 1.

If no such augmenting path can be found, then t is not reachable from s in $G_{\text{res}} = (V, A_+ \cup A_-)$; stop. $(x_{uv})_{(u,v)\in A}$ is an s-t flow of maximum value. Let $S = \{v \in V : v \text{ is reachable from } s \text{ in } G_{\text{res}}\}$. $[S, V\backslash S]$ is an s-t cut of minimum capacity.

Applying MAXFLOW to Example 5.1.1 yields the following sequence of 1-4 flows and 1-4 augmenting paths:

$(x_{12}, x_{13}, x_{23}, x_{24}, x_{31}, x_{34})$	ϑ	1-4 augm. path	Δ
$(0,0,0,0,0,0)$	0	$1,2,3,4$	$\min\{3-0, 2-0, 2-0\} = 2$
$(2,0,2,0,0,2)$	2	$1,2,4$	$\min\{3-2, 2-0\} = 1$
$(3,0,2,1,0,2)$	3	$1,3,2,4$	$\min\{1-0, 2, 2-1\} = 1$
$(3,1,1,2,0,2)$	4	none, so stop	

The residual digraph for each 1-4 flow is shown below, together with the augmenting path used.



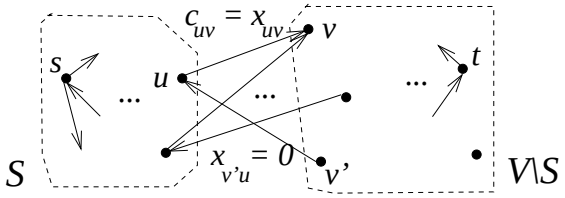
The final 1-4 flow has value of 4. Only 1 is reachable from 1 in G_{res}, so $S = \{1\}$ and the 1-4 cut found by MAXFLOW is $[\{1\}, \{2,3,4\}] = \{(1,2), (1,3)\}$ with capacity of $1 + 3 = 4$. By weak duality (Fact 5.1.1), 4 is the maximum flow value and the minimum cut capacity. Notice that $[\{1,2,3\}, \{4\}] = \{(2,4), (3,4)\}$ is also an 1-4 cut of minimum capacity, though MAXFLOW would not find it.

If the arc capacities c_{uv} are all integers, as in the case of Example 5.1.1, then it is not difficult to see that the arc flows x_{uv} and value ϑ would remain integer at all iterations of MAXFLOW. [Each pass through Step 1 is an iteration.] Then Δ would be a positive integer at all iterations, so the flow value ϑ increases by at least 1 at each iteration. Thus MAXFLOW must stop after a finite number of iterations whenever a flow of maximum value exists (i.e., the maximum value is finite). The number of iterations in general would depend on the size of the arc capacities. MAXFLOW would still terminate finitely if the arc capacities are rational numbers. This is because we can scale the capacities by a common denominator to make them integer, which does not affect finite termination. However, if the arc capacities are irrational, then there exist examples on which MAXFLOW may perform an infinite number of flow updates. But if we choose the augmenting paths more carefully and, in particular, always choose an augmenting path with fewest number of arcs (which can be found by finding a SP in the residual digraph with arc weights set to 1), then it can be shown that the number of iterations is at most $|V||A|$, regardless of whether arc capacities are rational.

When MAXFLOW stops, we have that $t \notin S = \{v \in V : v \text{ is reachable from } s \text{ in } G_{\text{res}}\}$. Moreover, there is no arc in G_{res} from S to $V\backslash S$. This can happen only if

$$x_{uv} = c_{uv} \quad \text{for } (u,v) \in [S, V\backslash S],$$

$$x_{vu} = 0 \quad \text{for } (v,u) \in [V\backslash S, S],$$



and it is not difficult to see that this implies $\vartheta = c[S]$ (see Exercise 5.1.2), so that, by weak duality (Fact 5.1.1), ϑ is maximum s-t flow value and $c[S]$ is minimum s-t cut capacity.

Exercises.

5.2.1.

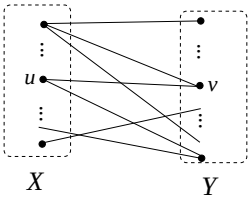
5.2.2. Consider a digraph $G = (V, A)$ with nonnegative arc capacities c_{uv}, $(u, v) \in A$, and a vertex $s \in V$. For each path P in G, define the "capacity" of P to be the *minimum* of the capacity of all the arcs in P. [Thus, if $P : u_1, u_2, ..., u_{p+1}$, then the capacity of P is $\min\{c_{u_1u_2}, ..., c_{u_pu_{p+1}}\}$.] Indicate how you would modify SP-Dijkstra so to find a path from s to each vertex v reachable from s that is of *maximum* capacity. [Hint: Replace "minimum" and "+" in SP-Dijkstra by suitable arithmetic operations.]

5.3 MATCHING AND COVERING IN BIPARTITE GRAPHS

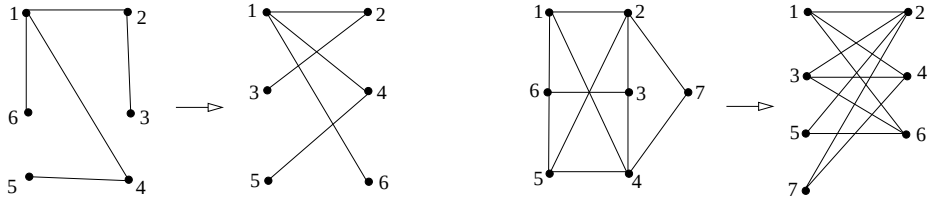
How best to pair up boys with girls (or students with tutors or people with jobs or..) if not all pairings are compatible? This is a classical problem of "matching" on bipartite graphs which, as we shall see, is closely related to the maximum flow problem.

More precisely, a graph $G = (V, A)$ is **bipartite** if we can partition V into two nonempty subsets X, Y such that all arcs in A go across X and Y only, i.e.,

$$X \cup Y = V, \quad X \cap Y = \emptyset, \qquad (u, v) \in A \implies u \in X, v \in Y \text{ or } u \in Y, v \in X.$$

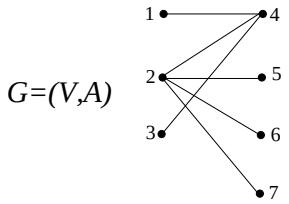


Notice that a tree is bipartite. More generally, it can be shown that a graph is bipartite if and only if all its cycles have *even* number of arcs.



A **matching** of G is any subset of arcs $M \subseteq A$ whose end vertices are all distinct (i.e., no two arcs in M share an end vertex). We wish to find a matching of maximum cardinality (most number of arcs). [If X represent girls and Y represent boys and an arc (u, v) means u and v are compatible, then we wish to match up as many compatible pairs of boys and girls as possible.]

Example 5.2.1. For the bipartite graph



some examples of matchings and their cardinalities are listed below:

M	$\lvert M\rvert$
$\emptyset$	0
$\{(1,4)\}$	1
$\{(2,7)\}$	1
$\{(3,4),(2,7)\}$	2
$\{(1,4),(2,6)\}$	2

Is 2 the maximum cardinality of matchings? How can we check this?

A **covering** of G is any subset of vertices $W \subseteq V$ such that every arc in A is joined to at least one vertex in W. We wish to find a covering of minimum cardinality (fewest number of vertices).

Example 5.2.1 continued. Some examples of coverings and their cardinalities for the bipartite graph of Example 5.2.1 are listed below:

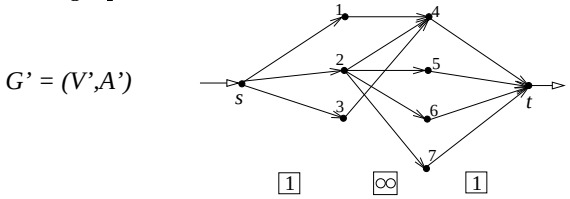
W	$\lvert W\rvert$
$\emptyset$	0
$\{1,2,3,4,5,6,7\}$	7
$\{1,2,3\}$	3
$\{4,5,6,7\}$	4
$\{2,4\}$	2

Is 2 the minimum cardinality of matchings? How can we check this?

The above maximum matching and minimum covering problems are in some sense dual to each other. In particular, weak duality holds in that $|M| \leq |W|$ for any matching M and covering W. Moreover, equality holds if and only if M has maximum cardinality and W has minimum cardinality. This can be proven from scratch, but it is more easily shown by transforming the maximum matching problem into a maximum flow problem and then applying the results from previous sections. This we do below, using the bipartite graph of Example 5.2.1 for illustration.

First we direct the arcs in A from X to Y and assign capacity of ∞ to them. Then we add to two vertices s and t, with arcs from s to X and from Y to t, each with capacity of 1. This results in a capacitated digraph, which we denote by $G' = (V', A')$. [Thus $V' = V \cup \{s, t\}$.] For the bipartite graph of Example

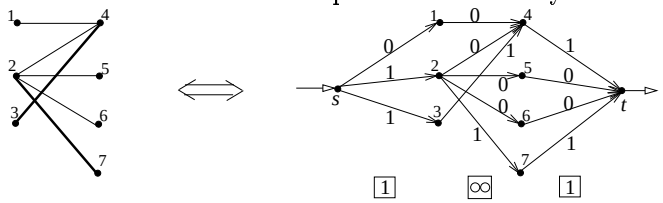
5.2.1, the associated capacitated digraph G' is shown below



Each arc (u, v) in a matching M of G corresponds to one unit of flow sent on the arcs $(s, u), (u, v), (v, t)$ in G', i.e.,

$$(u, v) \in M \qquad \Leftrightarrow \qquad x_{su} = x_{uv} = x_{vt} = 1. \tag{5.3.1}$$

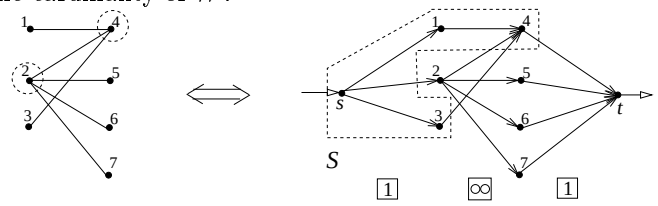
For example, the matching $M = \{(3, 4), (2, 7)\}$ corresponds to the s-t flow $x_{s3} = x_{34} = x_{4t} = 1$, $x_{s2} = x_{27} = x_{7t} = 1$, with all other arc flows zero. Its value of 2 equals the cardinality of M.



[G' also has s-t flows that do not correspond to matchings of G. This occurs when some arc flow is fractional. An example is $x_{s1} = x_{s3} = x_{14} = x_{34} = .5$, $x_{4t} = 1$, with all other arc flows zero.] Each vertex $u \in X$ (respectively, $v \in Y$) in a covering W of G corresponds to arc (s, u) (respectively, (v, t)) in an s-t cut in G'. In particular, the s-t cut in G' corresponding to W is $[S, V' \backslash S]$, where

$$S = \{s\} \cup (X \backslash W) \cup (Y \cap W) \qquad \text{or, equivalently,} \qquad W = (X \backslash S) \cup (Y \cap S). \tag{5.3.2}$$

For example, the covering $W = \{2, 4\}$ corresponds to the s-t cut $[\{s, 1, 3, 4\}, \{2, 5, 6, 7, t\}] = \{(s, 2), (4, t)\}]$. Its capacity of 2 equals the cardinality of W.



[G' also has s-t cuts that do not correspond to coverings of G. This occurs when some cut arc has ∞ capacity. An example is $[\{s, 1, 2\}, \{3, 4, 5, 6, 7, t\}] = \{(s, 3), (1, 2), (1, 7), (2, 4), (2, 5), (2, 6), (2, 7)\}$.]

Since each matching M of G corresponds to an s-t flow in G' with value equal to $|M|$ and each covering W of G corresponds to an s-t cut in G' with capacity equal to $|W|$, the following weak duality result follows from that for max flow-min cut, namely, Fact 5.1.1.

Fact 5.3.1 *(Weak Duality). For any matching M and any covering W of a bipartite graph, we have $|W| \geq |M|$.*

For $M = \{(3, 4), (2, 7)\}$ and $W = \{2, 4\}$ above, we have $|M| = |W|$ so that, by Fact 5.3.1, M is a maximum matching and W is a minimum covering. In general, we can find a maximum matching M

and minimum covering W by applying the MAXFLOW algorithm of Section 5.3 to find a maximum s-t flow $(x_{uv})_{(u,v)\in A}$ and a minimum s-t cut $[S, V'\backslash S]$ in G', and then use (5.3.1) and (5.3.2) to find the corresponding matching M and covering W in G with $|M| = |W|$.[2] A consequence of this is the following classical result of P. Hall in 1935, characterizing when all vertices on one side can be matched.

Fact 5.3.2 *(Philip Hall's Theorem). A bipartite graph $(X \cup Y, A)$ has a matching M with $|M| = |X|$ if and only if $|U| \leq |N_A(U)|$ for all $U \subseteq X$.*[3]

Proof. "If": Assume $|U| \leq |N_A(U)|$ for all $U \subseteq X$. We know there exist matching $M \subseteq A$ and covering $W \subseteq V$ such that $|M| = |W|$. Since W is a covering, each arc in A that has one end not in W must have the other end in W. Thus $N_A(X \setminus W) \subseteq Y \cap W$ and hence $|N_A(X \setminus W)| \leq |Y \cap W|$. Also, by assumption, $|X \setminus W| \leq |N_A(X \setminus W)|$. Thus $|X \setminus W| \leq |Y \cap W|$. This yields

$$\begin{aligned} |X| &= |X \setminus W| + |X \cap W| \\ &\leq |Y \cap W| + |X \cap W| \\ &= |W| \\ &= |M|. \end{aligned}$$

Since M is a matching so that $|M| \leq |X|$ also, we must have $|M| = |X|$.

"Only if": Assume $(X \cup Y, A)$ has a matching M with $|M| = |X|$. For any $U \subseteq X$, since $M \subseteq A$, we have $N_M(U) \subseteq N_A(U)$ and hence $|N_M(U)| \leq |N_A(U)|$. Since M is a matching, $|N_M(U)| = |U|$. Thus $|U| \leq |N_A(U)|$.

Fact 5.3.2 equivalently says that there does not exist a matching M with $|M| = |X|$ if and only if there exists a $U \subseteq X$ with $|U| > |N_A(U)|$. This allows for easy verification of the non-existence of such a matching. For the graph in Example 5.1.1, we have for $U = \{1, 3\}$ that $N_A(U) = \{4\}$, so $|U| = 2 > 1 = |N_A(U)|$. Thus, by Hall's theorem, vertices in X cannot all be matched.

If the graph is not bipartite, then finding a matching of maximum cardinality becomes more involved (though still not so difficult) and must take into account the presence of cycles with odd number of arcs.

---

[2] Why does this work? Since the capacities of arcs in G' are either 1 or ∞ and its maximum flow value is at most $\min\{|X|, |Y|\}$ (it cannot exceed the total capacity of arcs out of s, which is $|X|$, or the total capacity of arcs into t, which is $|Y|$), the MAXFLOW algorithm applied to G' will stop after a finite number of iterations and output an integer s-t flow and an s-t cut whose respective value and capacity equal. Since the arcs from s to X and from Y to t have capacity 1, the flow on each arc must be either 0 or 1, and this s-t flow corresponds to a matching M of G (via (5.3.1)). Also, the s-t cut has finite capacity, so it cannot contain any arc with ∞ capacity. Therefore it corresponds to a covering W of G (via (5.3.2)). Moreover, equal flow value and cut capacity means $|M| = |W|$, so Fact 5.3.1 implies $|M|$ is maximum and $|W|$ is minimum.

[3] $N_A(U) = \cup_{u\in U}\{v \in Y : (u, v) \in A\}$ *denotes the set of Y vertices joined by arcs in A to U.*

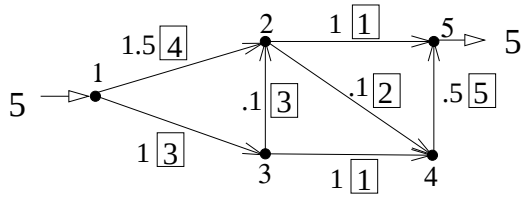
Exercises.

5.3.1. For each positive integers k and l, let $G_{k,l}$ denote the bipartite graph with k vertices on one side, l vertices on the other side, and an edge joining every vertex on one side to every vertex on the other side. Give all values of k and l for which $G_{k,l}$ has a closed eulerian path. Explain your answer.

5.4 MINIMUM COST FLOW

Both the shortest path problem and the maximum flow problem are themselves special cases of a discrete optimization problem, called the **minimum cost flow** (MCF) problem. This problem has many real-world applications. For example, when I visited the National University of Colombia in Medellín in the late 1980's, they had a computer lab of PCs. As each PC can be used by only one student, each student had to fill out a form indicating his/her preferred times for using the lab per week, and the students were assigned to lab times based on their preferences. The assignment was done by hand and hence was slow and inefficient. We formulated this problem as an MCF, which we then solved using a computer algorithm that I co-developed with my former thesis advisor. Let us motivate the problem with an example.

Example 5.4.1. Consider a situation whereby 5 units of a certain product (say, wheat) is to be shipped from city 1 to city 5 via cities 2, 3, 4. The cities are joined by 1-way roads as shown and, on each road, there is a capacity (limit) on the number of units that can be shipped and a per unit cost for shipping the product. We wish to find a way to ship the 5 units at minimum cost.



If we introduce, for each road from city u to city v, the variable

$$x_{uv} = \text{ number of units shipped from } u \text{ to } v,$$

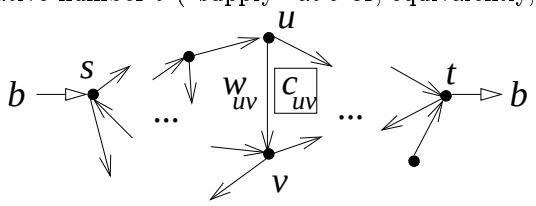
then this problem may be written as the following linear program (LP) with 8 variables, 5 equations, and 16 inequalities:

$$\begin{array}{lrcrcrcrcrcrcrcl}
\min & 1.5x_{12} & + & x_{13} & + & .1x_{24} & + & x_{25} & + & .1x_{32} & + & x_{34} & + & .5x_{45} & & \\
\text{subject to} & x_{12} & + & x_{13} & & & & & & & & & & & = & 5 \\
& -x_{12} & & & + & x_{24} & + & x_{25} & - & x_{32} & & & & & = & 0 \\
& & - & x_{13} & & & & & & x_{32} & + & x_{34} & & & = & 0 \\
& & & & - & x_{24} & & & & & - & x_{34} & + & x_{45} & = & 0 \\
& & & & & & - & x_{25} & & & & & - & x_{45} & = & -5
\end{array}$$

$$0 \le x_{12} \le 4, 0 \le x_{13} \le 3, 0 \le x_{24} \le 2, 0 \le x_{25} \le 1, 0 \le x_{32} \le 3, 0 \le x_{34} \le 1, 0 \le x_{45} \le 5.$$

["s.t." is an abbreviation of "subject to".]

In the general MCF problem, we are given a digraph $G = (V, A)$ and, for every arc $(u, v) \in A$, two nonnegative numbers c_{uv} ("capacity") and ω_{uv} ("cost" per unit of flow). We are also given two distinct vertices $s, t \in V$ and a nonnegative number b ("supply" at s or, equivalently, "demand" at t).



An s-t flow $\{x_{uv}\}_{(u,v)\in A}$ on G is **feasible** if its value equals b and we define its **cost** to be $\sum_{(u,v)\in A} \omega_{uv} x_{uv}$. [An s-t flow is defined as in Section 5.1] We wish to find a feasible s-t flow whose cost is minimum, that is, less than or equal to the cost of all other feasible s-t flows. This problem may be written as

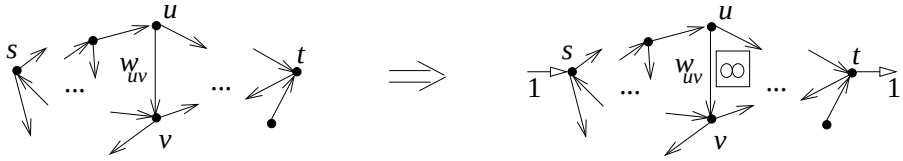
$$\begin{array}{lll} \min & \sum_{(u,v)\in A} \omega_{uv} x_{uv} & \\ \text{s.t.} & \sum_{v:(u,v)\in A} x_{uv} - \sum_{v:(v,u)\in A} x_{vu} & = \begin{cases} b & \text{if } u = s, \\ -b & \text{if } u = t, \\ 0 & \text{else,} \end{cases} \end{array}$$

$$0 \le x_{uv} \le c_{uv}, \quad \text{for all } (u, v) \in A$$

which is a special kind of LP with $|A|$ variables, $|V|$ equations, and $2|A|$ inequalities. [Recall that an LP is the problem of minimizing/maximizing a linear function subject to the variables satisfying linear equations and linear inequalities.]

As we mentioned earlier, both the shortest path problem and the maximum flow problem are special cases of the MCF problem, which we show below.

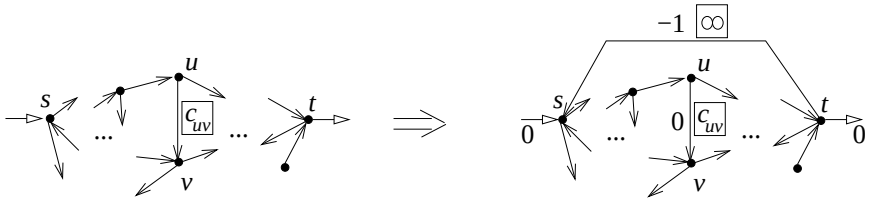
Shortest Path. Consider the shortest path problem (see Chapter 4) of finding the minimum weight path in a digraph (V, A) with arc weights ω_{uv}, $(u, v) \in A$, from a vertex s to another vertex t. This problem is equivalent to the MCF problem on (V, A) with capacities $c_{uv} = \infty$, $(u, v) \in A$, weights ω_{uv}, $(u, v) \in A$, and supply $b = 1$ at s and demand $b = 1$ at t. [Intuitively, the cheapest way to send 1 unit of flow from s to t is via the minimum weight path from s to t.]



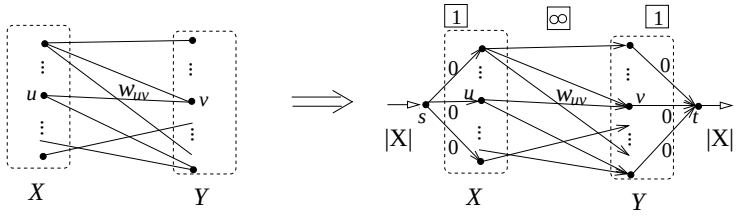
SP problem equivalent MCF problem

Maximum Flow. Consider the maximum flow problem of finding an s-t flow in a digraph (V, A) with nonnegative capacities c_{uv}, $(u, v) \in A$, whose value is maximum. We can transform this as a MCF problem as follows: Let $\omega_{uv} = 0$ for all $(u, v) \in A$. Let $A' = A \cup \{(t, s)\}$, $c_{ts} = \infty$, and $\omega_{ts} = -1$. Then, this problem is equivalent to the MCF problem on (V, A') with capacities c_{uv}, $(u, v) \in A'$, arc weights ω_{uv}, $(u, v) \in A'$, and supply $b = 0$ at s and demand $b = 0$ at t. [Intuitively, since $\omega_{(t,s)} = -1$ and all other arc weights are zero, minimizing the cost of a feasible s-t flow on (V, A') is equivalent to maximizing the flow on (t, s) which

in turn is equivalent to maximizing the external flow from s to t in (V, A).]



Maximum Weight Matching. Consider a bipartite graph $G = (X \cup Y, A)$ with $|X| = |Y|$, and nonnegative arc weights ω_{uv}, $(u, v) \in A$. We wish to find a matching M of cardinality $|X|$ whose weight $\omega(M) = \sum_{(u,v)} \omega_{uv}$ is maximum. In the special case where $\omega_{uv} = 1$ for all $(u, v) \in A$, we have $\omega(M) = |M|$, and the problem reduces to that of finding a matching of maximum cardinality. [If X represents girls and Y represents boys, then ω_{uv} can be interpreted as the level of compatibility between girl u and boy v, and the problem is to maximize the average compatibility of the matched pairs.] Unlike maximum-cardinality matching, it cannot be transformed into a maximum flow problem. However, it can be transformed into a MCF problem as follows: We construct the associated capacitated digraph $G' = (V', A')$ as in Section 5.2, and we assign a weight of ω_{uv} for each arc (u, v) from X to Y, and assign zero weight to all arcs from s to X and from Y to t.



It can be shown that a matching of G of maximum weight corresponds to an integer feasible s-t flow on G' of minimum cost.

Since the MCF problem is an LP, we can solve it using any algorithm for solving LP, such as the simplex method or dual simplex method. Many algorithms have been proposed. One such algorithm extends the MAXFLOW algorithm in Section 5.2, which ties in nicely with what we have seen so far. We sketch its idea: We start with the zero s-t flow (i.e., $x_{uv} = 0$ for all $(u, v) \in A$). Then, as in the MAXFLOW algorithm, we successively augment flow from s to t along s-t augmenting paths until its value equals b. The only difference is that each s-t augmenting path is chosen specially, namely, it corresponds to not just any path from s to t in the residual digraph $G_{\text{res}} = (V, A_+ \cup A_-)$ from but a *shortest path*, where the weight of each arc $(u, v) \in A_+$ is ω_{uv} and the weight of each arc $(u, v) \in A_-$ is $-\omega_{vu}$. Notice that arc weights can be negative, so G_{res} can have a cycle of negative weight, in which case we instead augment flow around this cycle, and repeat this until no cycle in the residual digraph has negative weight. [It can be shown that augmenting flow along a cycle of negative weight corresponds to a simplex pivot.] The SP paths can be found using either Bellman-Ford algorithm (Section 4.2) or Floyd-Warshall algorithm (Section 4.3). These algorithms can also

find the negative-weight cycles when suitably modified.

The MCF problem has the nice property that if it has a minimum cost flow and the supply/demand b and the arc capacities c_{uv} are all integer, then it has an *integer* minimum cost flow.

5.5 ADDITIONAL TOPICS

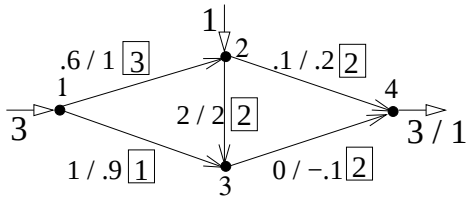
A great deal of research have been devoted to understanding the structures of and designing fast algorithms for solving shortest path, maximum flow, maximum weight matching (also known as the optimal assigment problem), and MCF problems. These problems can also arise in unexpected places.

In some applications, a fraction of the flow on an arc may get lost. For example, in Example 5.4.1, suppose that a fraction .1 of the flow on arc $(1,2)$ is lost. Then the flow conservation equations at vertices 1 and 2 would change to

$$\begin{array}{rcrcrcrcrcl} x_{12} & + & x_{13} & & & & & & & = & 5 \\ -.9x_{12} & & & + & x_{24} & + & x_{25} & - & x_{32} & = & 0 \end{array}$$

A digraph whose arc flows are attenuated in this manner is called a "gain network", where the fraction .9 is called the "gain" of arc $(1,2)$. The resulting MCF problem has a more complicated structure, though it is still a special case of an LP.

In the MCF problem, the incoming flows into a vertex are mixed and sent out on outgoing arcs. Thus the flows are in a sense of the same kind, so they can be mixed. In many applications, flows of different kinds may share the same arcs, but they cannot be mixed. For example, apples sent from Washington to Georgia might share some routes with potatoes sent from Idaho to Florida, but they cannot be mixed (they are not interchangeable to consumers). Such problem is sometimes called a "minimum cost multicommodity flow" (MCMF) problem. Different commodities may have different costs, but they must satisfy the same flow conservation constraints, and they share the capacity constraints. As an illustration, consider the 2-commodity flow problem shown below, with the commodity unit costs and the capacity shown for each arc. Here, 3 units of commodity 1 is to be sent from vertex 1 to 4 and 1 unit of commodity 2 is to be sent from vertex 2 to 4.



Letting

$$x_{uv}^k = \text{ number of units of commodity } k \text{ shipped from } u \text{ to } v,$$

the problem can be written in the form:

$$
\begin{array}{lccccccccccc}
\min & .6x_{12}^1 & +x_{13}^1 & +2x_{23}^1 & +x_{24}^1 & +x_{12}^2 & +.9x_{13}^2 & +2x_{23}^2 & +.2x_{24}^2 & -.1x_{34}^2 & \\
\text{s.t.} & x_{12}^1 & +x_{13}^1 & & & & & & & & = 3 \\
& -x_{12}^1 & & +x_{23}^1 & +x_{24}^1 & & & & & & = 0 \\
& -x_{13}^1 & -x_{23}^1 & & +x_{34}^1 & & & & & & = 0 \\
& & & -x_{24}^1 & -x_{34}^1 & & & & & & = -3 \\
& & & & & x_{12}^2 & +x_{13}^2 & & & & = 0 \\
& & & & & -x_{12}^2 & & +x_{23}^2 & +x_{24}^2 & & = 1 \\
& & & & & & -x_{13}^2 & -x_{23}^2 & & +x_{34}^2 & = 0 \\
& & & & & & & & -x_{24}^2 & -x_{34}^2 & = -1
\end{array}
$$

$$0 \le x_{12}^1 + x_{12}^2 \le 3, \quad 0 \le x_{13}^1 + x_{12}^2 \le 1, \quad 0 \le x_{23}^1 + x_{23}^2 \le 2, \quad 0 \le x_{24}^1 + x_{24}^2 \le 2, \quad 0 \le x_{34}^1 + x_{34}^2 \le 2.$$

The MCMF problem is still a special case of LP. However, due to its large size (real-world problems can have thousands of vertices and tens of commodities), specialized algorithms are needed to solve it fast. If the flows are further required to be integers, then the MCMF problem becomes much harder, belonging to the class of NP-hard problems.

6. Integer Program

Oftentimes in practice we encounter a linear program (LP) whose variables are further required to be *integer-valued.* For example, when a manufacturing company decides on how many factories to build or when an airline decides on how many planes of each type to purchase, the answer must be integers. Such a problem is called an **integer program** ("IP" for short). More precisely, a $\{\begin{smallmatrix}\max\\ \min\end{smallmatrix}\}$ IP is the problem of $\{\begin{smallmatrix}\text{maximizing}\\ \text{minimizing}\end{smallmatrix}\}$ a linear function (the **cost** function) subject to the variables being integer-valued and satisfying a system of linear equations/inequalities. In general, an IP is much harder to solve than an LP, but, as we shall see, we can in some sense solve an IP by solving a sequence of LP's.

6.1 DEFINITIONS AND PROPERTIES

Let's see some examples to get a sense of how IP can arise in practice.

Example 6.1.1. Suppose that we have three objects 1, 2, 3 with weight of 3 lb, 2 lb, 4 lb and value of \$1, \$2, \$3 respectively. We also have a boat that can carry at most 5 lbs. of cargo (small boat!) What combination of these objects should we put on the boat so to maximize the total value of the cargo?

By introducing the variables

$$x_i = \begin{cases} 1 & \text{if we take object } i; \\ 0 & \text{else,} \end{cases} \qquad i = 1, 2, 3,$$

we can formulate this problem as the max IP:

$$\begin{array}{lrcrcrcl} \max & x_1 & + & 2x_2 & + & 3x_3 & & \\ \text{s.t.} & 3x_1 & + & 2x_2 & + & 4x_3 & \leq & 5, \\ & & & 0 \leq x_i \leq 1, \text{ integer}, & & & & i = 1, 2, 3. \end{array}$$

[Notice that we have expressed the constraint $x_i \in \{0, 1\}$ as $0 \leq x_i \leq 1$ and x_i is integer.]

Suppose that, in addition, we are told that objects 1 and 2 cannot go together. How can we model this? Answer: We add to the above IP the linear inequality

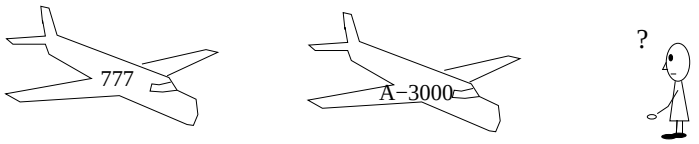
$$x_1 + x_2 \leq 1.$$

[So if $x_1 = 1$, then x_2 must be 0 and if $x_2 = 1$, then x_1 must be 0.] Suppose that instead we are told that exactly one of objects 1, 2 and 3 must be on the boat. How can we model this? Answer: We add to the above IP the linear equation

$$x_1 + x_2 + x_3 = 1.$$

[So either $x_1 = 1, x_2 = x_3 = 0$ or $x_2 = 1, x_1 = x_3 = 0$ or $x_3 = 1, x_1 = x_2 = 0$.] Thus, we see that an IP is very good for modeling logical constraints (e.g., if event A occurs, then event B cannot occur...)

Example 6.1.2. Yankee airline is deciding how many Airbus A-3000 and how many Boeing 777 to buy. Each Airbus A-3000 costs \$5 million and has a service life of 5 years; each Boeing 777 costs \$9 million and has a service life of 8 years (not the most realistic figures!) Yankee estimates they need to buy only 6 jets in total and their purchasing budget is \$45 million. What should Yankee do so to maximize the total service life of its new fleet of jets?



By introducing the variables

x_1 = number of Airbus A-3000 bought,

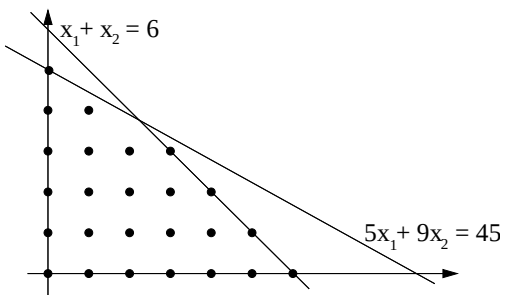
x_2 = number of Boeing 777 bought,

we can formulate this problem as the max IP:

$$\begin{array}{lrcrcl} \max & 5x_1 & + & 8x_2 & & \\ \text{s.t.} & x_1 & + & x_2 & \leq & 6, \\ & 5x_1 & + & 9x_2 & \leq & 45, \\ & & & x_1,\ x_2 \geq 0, & & \text{integer.} \end{array}$$

For an IP, we will call any integer value of the variables that satisfy the system of linear equations/inequalities in the IP a **feasible solution** of the IP. For the IP in Example 6.1.1, $x_1 = x_2 = x_3 = 0$ is a feasible solution as are $x_1 = x_2 = 1, x_3 = 0$ and $x_1 = x_2 = 0, x_3 = 1$, but $x_1 = x_2 = x_3 = 1$ is not. For the IP in Example 6.1.2, $x_1 = x_2 = 0$ is a feasible solution, as is $x_1 = 1, x_2 = 4$, but $x_1 = 1, x_2 = 5$ is not. We call the set of all feasible solutions the **feasible region** of the IP. Thus, a max (respectively, min) IP is the problem of finding a feasible solution whose cost is greater (respectively, less) than or equal to the cost of all other feasible solutions. Such a feasible solution, if it exists, will be called an **optimal solution** of the IP and its cost will be called the **optimal cost** of the IP. Finding an optimal solution will be our main goal.

Example 6.1.2 continued. For the IP in Example 6.1.2, the feasible region comprise the integer points (indicated by the black dots) inside the shaded region.



And, as we shall see shortly, this IP has an optimal solution at $x_1 = 0, x_2 = 5$ and the optimal cost is $5x_1 + 8x_2 = 40$. [So Yankee should buy 5 Boeing 777s and no Airbus A-3000!]

6.2 BRANCH AND BOUND ALGORITHM

How do we solve an IP? A naive way is to check all possible integer-values for the variables, discarding those that do not satisfy the equations/inequalities, and, amongst those remaining, pick one that yields the maximum/minimum cost if we are maximizing/minimizing the cost function. [For Example 6.1.2, say, we would first fix $x_1 = 0$ and try $x_2 = 0, 1, 2, ...,$, etc.; then we would fix $x_1 = 1$ and try $x_2 = 0, 1, 2, ...,$, etc.; then we would fix $x_1 = 2$ and try $x_2 = 0, 1, 2, ...,$, etc. , and so on.] This, of course, is very slow. It turns out that IP belongs to a famous class of hard problems (called NP-Hard problems) for which no "fast" solution method is known. [In contrast, an LP is considered to be fairly easy.] Nonetheless, there are methods for solving IP's that, at least on small to medium-sized problems (say, fewer than 300 variables and fewer than 500 equations/inequalities), work reasonably well. The best known of them is **branch and bound**, invented by Ralph Gomory around 1960 and used by many computer packages (such as LINDO), which we describe below.

For any IP, if we ignore the requirement that the variables be integer-valued, we obtain an LP, which we will call the **LP relaxation** of the IP. The feasible region for this LP clearly contains that for IP. The idea of branch and bound is to start by solving the LP relaxation of IP and, whenever one of the variables is not integer in the optimal solution, we add either an $\geq$ inequality or an $\leq$ inequality to the LP so to rule out this noninteger value for the variable. Thus, we will solve a sequence of LP's. For simplicity, we will describe the procedure for solving a max IP only. [A min IP can be reduced to this case by negating its cost function.]

Branch & Bound (B&B)

Input: A max IP with variables $x_1, ..., x_n$ and whose LP relaxation has an optimal solution.

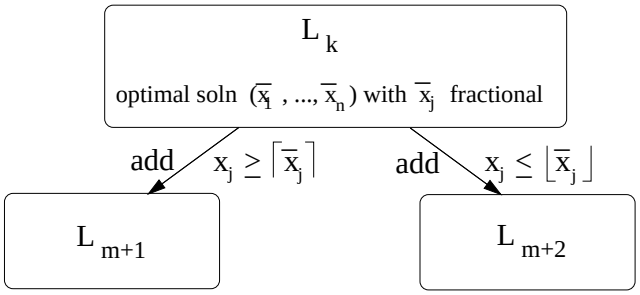
Output: An optimal solution of the max IP, if there is one.

0. Let L_0 be the LP relaxation of max IP. Initialize $m = 0$. Go to Step 1.

1. If L_0, L_1, ..., L_m have all being solved, stop. [Claim: If one or more of these LPs has an integer-valued optimal solution, then the integer-valued optimal solution having the maximum cost is an optimal solution of the max IP. Else the max IP has no optimal solution.] Else pick some L_k ($k \in \{0, 1, ..., m\}$) that has not yet been solved. We solve L_k (using any suitable method such as the simplex method or dual simplex method) and either (i) L_k has no optimal solution or (ii) L_k has an optimal solution $(\bar{x}_1, ..., \bar{x}_n)$ and $\bar{x}_1, ..., \bar{x}_n$ are all integers or (iii) L_k has an optimal solution $(\bar{x}_1, ..., \bar{x}_n)$ and at least one of $\bar{x}_1, ..., \bar{x}_n$ is not an integer. In cases (i) and (ii), go to Step 1; in case (iii), go to Step 2.

2. (Branching on a variable). Choose any index $j \in \{1, ..., n\}$ such that $\bar{x}_j$ is not integer. Let L_{m+1} be L_k with the inequality $x_j \geq \lceil \bar{x}_j \rceil$ added and let L_{m+2} be L_k with the inequality $x_j \leq \lfloor \bar{x}_j \rfloor$ added. Increment

m by 2 and return to Step 1.



Example 6.1.2 continued. Let us apply the branch and bound procedure to solve the max IP of Example 6.1.2.

Step 0. L_0 is:

$$\begin{array}{lrcrcl} \max & 5x_1 & + & 8x_2 & & \\ \text{s.t.} & x_1 & + & x_2 & \leq & 6, \\ & 5x_1 & + & 9x_2 & \leq & 45, \\ & & & x_1,\ x_2 \geq 0. & & \end{array}$$

Initialize $m = 0$. Go to Step 1.

Step 1. Upon solving L_0, we obtain the optimal solution $x_1 = 2\frac{1}{4}$, $x_2 = 3\frac{3}{4}$, neither of which is integer. Thus we go to Step 2.

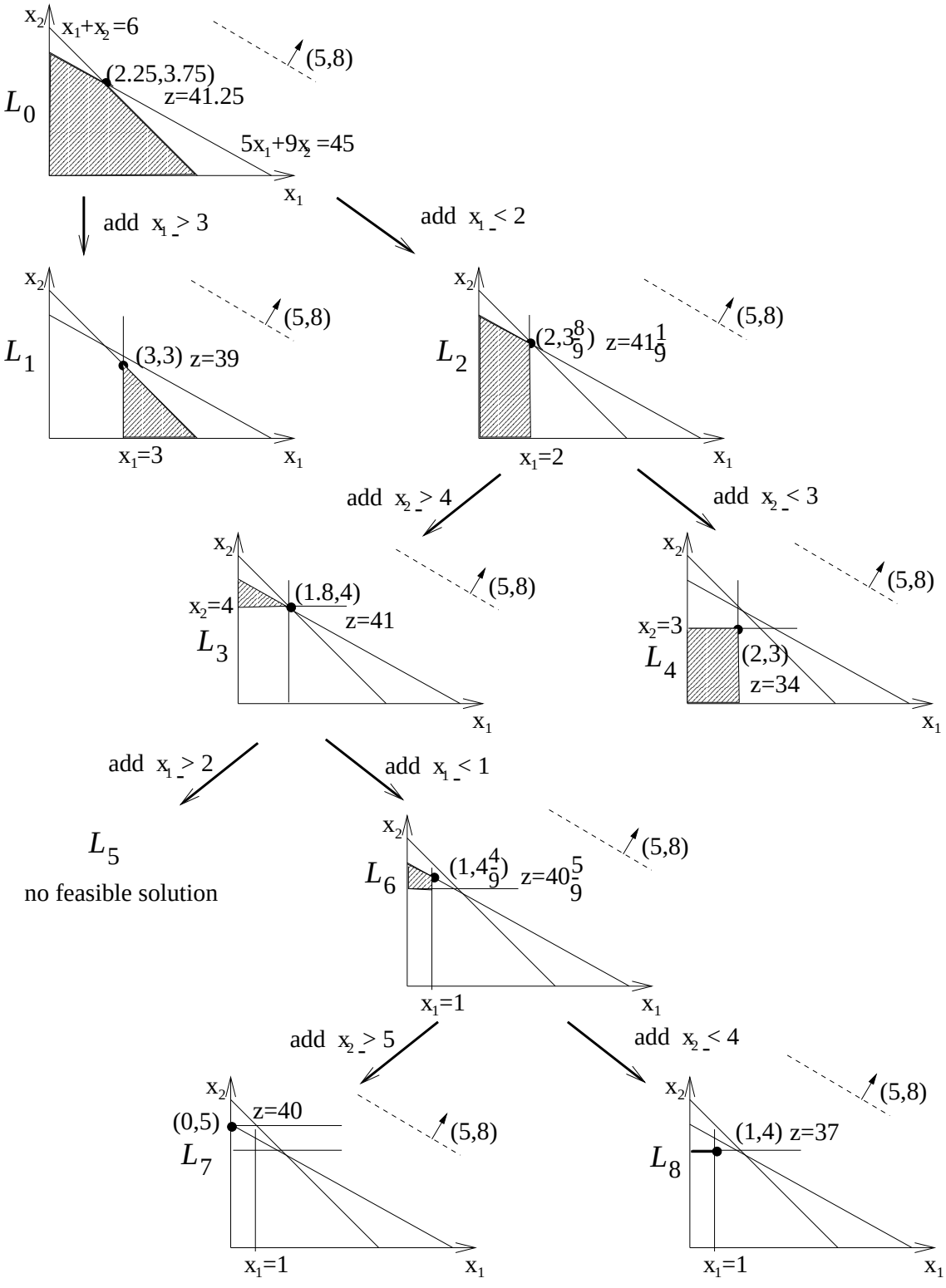
Step 2. We can choose $j = 1$ or 2, say $j = 1$. [Either choice works, though the work involved may differ.] We let L_1 be L_0 with the inequality $x_1 \geq 3$ added and let L_2 be L_0 with the inequality $x_1 \leq 2$ added. Increment m by 2 to $m = 2$ and go to Step 1.

Step 1. We can pick either L_1 or L_2 to solve, say L_1. We obtain the optimal solution $x_1 = 3$, $x_2 = 3$ both of which are integer, so we go to Step 1.

Step 1. Only L_2 has not been solved. Upon solving L_2, we obtain the optimal solution $x_1 = 2$, $x_2 = 3\frac{8}{9}$, the second of which is not integer. Thus we go to Step 2.

Step 2. $j = 2$ and we let L_3 be L_2 with the inequality $x_2 \geq 4$ added, and let L_4 be L_2 with the inequality $x_2 \leq 3$ added. Increment m by 2 to $m = 4$ and go to Step 1.

We continue in this manner until we reach $m = 8$ at which time we stop. The calculations are shown in the last two pages. The branching on variables can be summarized graphically in the form of an "B&B tree" shown below. [The boxes are called the "nodes" of the tree.] Upon comparing all the integer-valued optimal solutions we found, we see that $x_1 = 0, x_2 = 5$ has the maximum cost of 40, so this is an optimal solution of the IP. Pheww...



It is not hard to show that the B&B procedure works and that the number of repetitions of Steps 1 and 2 is finite. A precise proof is a bit involved, so we will not give it here. Notice that the amount of effort in the B&B procedure depends on m, the number of LPs we need to solve (which can be 2^n in the worst case).

We can reduce this number in a number of ways (see Section 6.4 for further discussions):

1 (Bounding) When we are in case (iii) of Step 1, if the optimal cost of L_k is less than the cost of some feasible solution of the max IP, then go to Step 1, rather than Step 2. [All feasible solutions of L_k are worse than this feasible solution of the IP, so none of them can be an optimal solution of the IP and hence L_k can be discarded.]

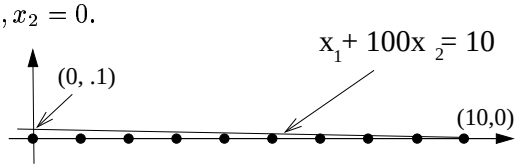
2 In Step 1, pick L_k to be an LP whose optimal cost is the highest possible. [Since the optimal solution of the IP is more likely to be in the feasible region of such an LP.]

We can further reduce the effort by using the dual simplex method to resolve the LPs created in Step 2. [Each new LP is created by adding an inequality to an LP we just solved. The dual simplex algorithm is very efficient at resolving an LP after an equation/inequality is added.] If an LP has more than one optimal solution, then we should seek a "corner point" optimal solution since such an optimal solution tends to have fewer fractional components.

Integer programming is a fascinating subject with many unexpected results. For example, can we approximate the optimal solution of an IP by "rounding off" the optimal solution of its LP relaxation? The answer is, alas, "no." Consider the IP:

$$\begin{array}{lrcrcl} \max & x_1 & + & 200x_2 & & \\ \text{s.t.} & x_1 & + & 100x_2 & \leq & 10, \\ & & & x_1,\ x_2 \geq 0, & \text{integer.} & \end{array}$$

The LP relaxation has the optimal solution $x_1 = 0, x_2 = 0.1$ which rounds off to $x_1 = x_2 = 0$. The IP has the optimal solution of $x_1 = 10, x_2 = 0$.



For another example, if IP has an optimal solution, must its LP relaxation have an optimal solution? The answer again is "no". Consider the IP

$$\begin{array}{lrcrcl} \max & x_1 & & & & \\ \text{s.t.} & \sqrt{2}x_1 & - & x_2 & = & 0, \\ & & & x_1,\ x_2 \geq 0, & \text{integer.} & \end{array}$$

Because $\sqrt{2}$ is irrational, there can be only one integer point on the line $\sqrt{2}x_1 - x_2 = 0$, namely, $x_1 = x_2 = 0$. Thus, IP has this as its optimal solution. However, its LP relaxation clearly has no optimal solution. On the other hand, if all coefficients are all integer or, more generally, rational, then this cannot happen. In that case, if IP has an optimal solution, then so does its LP relaxation. Or, put it in another way, if its LP relaxation has no optimal solution, then neither does IP (which is obviously very useful to know!)

Exercises.

6.2.1. Consider the following IP in 2 variables:

$$\begin{array}{lrcrcll} \max & x_1 & + & 5x_2 & & & \\ \text{s.t.} & 3x_1 & + & 2x_2 & \leq & 18, & \\ & -4x_1 & + & 3x_2 & \leq & 6, & \\ & x_1 & & & \geq & 0, & \\ & & & x_2 & \geq & 0, & x_1,\ x_2 \text{ integer.} \end{array}$$

Apply the branch & bound algorithm to this IP to determine if it has an optimal solution and, if so, to find an optimal solution.

6.2.2. Consider the following IP in 2 variables:

$$\begin{array}{lrcrcll} \max & x_1 & + & x_2 & & & \\ \text{s.t.} & -x_1 & + & 2x_2 & \leq & 0, & \\ & x_1 & & & \leq & 1, & \\ & & & 4x_2 & \geq & 1, & x_1,\ x_2 \text{ integer.} \end{array}$$

Apply the branch & bound algorithm to this IP to determine if it has an optimal solution and, if so, to find an optimal solution.

6.2.3.

a) Give an example of an IP that has exactly two feasible solutions. Give an example of an IP that has an infinite number of optimal solutions.

b) Consider a max IP whose cost function is of the form $c_1x_1 + \cdots + c_nx_n$, where $c_1, ..., c_n$ are integers. Suppose we solve the LP relaxation of this IP and obtain an optimal solution $(x_1^*, ..., x_n^*)$ with non-integer cost, i.e., $c_1x_1^* + \cdots + c_nx_n^*$ is not an integer. Write down a linear inequality that, when added to the LP relaxation, will (i) exclude $(x_1^*, ..., x_n^*)$ from its feasible region and (ii) not exclude any optimal solution of the IP.

6.3 CUTTING PLANE ALGORITHM

The branch & bound algorithm creates two new LPs from each LP whose optimal solution is non-integer. An alternative algorithm, proposed by Ralph Gomory and others in the late 1950s and 1960s, creates only one new LP. Thus it has the advantage that we only need to keep track of a single LP at all times. This algorithm, like branch & bound, repeatedly adds new linear constraints to the LP to cut away the current non-integer optimal solution.

A **valid cut** of an LP is a linear inequality that is satisfied by *all* integer feasible solutions of the LP. To simplify the discussion, we assume that the IP has the "standard" form:

$$\begin{array}{ll} \min & c'x \\ \text{s.t.} & Ax = b, \\ & x \geq 0, \text{ integer,} \end{array} \tag{6.3.1}$$

where $A \in Z^{m\times n}$, $b \in Z^{m\times 1}$, $c \in Z^{n\times 1}$, and we assume that A has linearly independent rows. Its LP relaxation is

$$\begin{array}{ll} \min & c'x \\ \text{s.t.} & Ax = b, \\ & x \geq 0. \end{array} \tag{6.3.2}$$

Cutting Plane

Input: An IP in standard form (6.3.1).

Output: An optimal solution of the IP, if there is one.

0. Solve the LP relaxation.

If LP is infeasible, stop; IP is infeasible.

If LP is unbounded, stop; IP has no optimal solution (see Exercise 6.3.1)).

Else LP has an optimal BFS x^*, go to Step 1.

1. If x^* is integer, stop; x^* is an optimal solution of IP (since feasible solutions of IP are feasible for LP).

Else x^* is non-integer. Add to LP a valid cut that is violated by x^*; return to Step 1.

How can we construct the desired valid cut? Let x^* be any non-integer optimal BFS of (6.3.2). Then

$$x^*_{\mathcal{B}} = A_{\mathcal{B}}^{-1} b, \qquad x^*_{\mathcal{N}} = 0,$$

for some $\mathcal{B} \subseteq \{1, ..., n\}$ and $\mathcal{N} = \{1, ..., n\} \backslash \mathcal{B}$. We describe two well-known choices of valid cuts.

1. Any feasible solution x of (6.3.1) satisfies

$$A_{\mathcal{B}} x_{\mathcal{B}} + \sum_{j\in\mathcal{N}} A_j x_j = b,$$

where A_j denotes the jth column of A. If $x_{\mathcal{N}} = 0$, then $x_{\mathcal{B}} = A_{\mathcal{B}}^{-1} b = x^*_{\mathcal{B}}$ would be fractional. So it must be that $x_{\mathcal{N}} \neq 0$. Since x_j is integer and nonnegative for all $j \in \mathcal{N}$, this implies

$$\sum_{j\in\mathcal{N}} x_j \geq 1, \tag{6.3.3}$$

so (6.3.3) is a valid cut. Moreover, x^* violates (6.3.3).

2. (Gomory cut) Any feasible solution x of (6.3.1) satisfies

$$\begin{aligned} & A_{\mathcal{B}} x_{\mathcal{B}} + \sum_{j\in\mathcal{N}} A_j x_j = b \\ \Leftrightarrow \quad & x_{\mathcal{B}} + \sum_{j\in\mathcal{N}} A_{\mathcal{B}}^{-1} A_j x_j = A_{\mathcal{B}}^{-1} b \\ \Leftrightarrow \quad & x_{\mathcal{B}(i)} + \sum_{j\in\mathcal{N}} \overline{a}_{ij} x_j = x^*_{\mathcal{B}(i)}, \quad i = 1...m, \end{aligned}$$

where $\overline{a}_{ij} = \left(A_{\mathcal{B}}^{-1} A_j\right)_i$ and $\mathcal{B}(i)$ is the ith element of $\mathcal{B}$. Since x^* is non-integer, $x^*_{\mathcal{B}(i)}$ is fractional for some i. Since $x_j \geq 0$ for all $j \in \mathcal{N}$,

$$x_{\mathcal{B}(i)} + \sum_{j \in \mathcal{N}} \lfloor \overline{a}_{ij} \rfloor x_j \leq x_{\mathcal{B}(i)} + \sum_{j \in \mathcal{N}} \overline{a}_{ij} x_j = x^*_{\mathcal{B}(i)}.$$

Since $x_{\mathcal{B}(i)}$ and $x_{\mathcal{N}}$ are integer, this implies

$$x_{\mathcal{B}(i)} + \sum_{j \in \mathcal{N}} \lfloor \overline{a}_{ij} \rfloor x_j \leq \lfloor x^*_{\mathcal{B}(i)} \rfloor, \tag{6.3.4}$$

so (6.3.4) is a valid cut. Moreover, for any $i \in \{1, ..., m\}$ such that $x^*_{\mathcal{B}(i)}$ is fractional, we have $\lfloor x^*_{\mathcal{B}(i)} \rfloor < x^*_{\mathcal{B}(i)}$, implying that x^* violates (6.3.4).

Example 6.3.1. We illustrate the Gomory cuts. Suppose the original IP is

$$\begin{array}{lrcrcl} \min & x_1 & - & 2x_2 & & \\ \text{s.t.} & -4x_1 & + & 6x_2 & \leq & 9 \\ & x_1 & + & x_2 & \leq & 4 \\ & & & x_1 \geq 0,\ x_2 \geq 0,\ \text{integer.} & & \end{array}$$

Transform this IP into standard form:

$$\begin{array}{lrcrcrcrcl} \min & x_1 & - & 2x_2 & & & & & & \\ \text{s.t.} & -4x_1 & + & 6x_2 & + & x_3 & & & = & 9 \\ & x_1 & + & x_2 & & & + & x_4 & = & 4 \\ & & & x_1 \geq 0,\ ...,\ x_4 \geq 0,\ \text{integer.} & & & & & & \end{array}$$

Its LP relaxation has optimal BFS: $x^* = (1.5, 2.5, 0, 0)$ corresponding to basis $\mathcal{B} = \{1, 2\}$. Then

$$A_{\mathcal{B}} = \begin{bmatrix} -4 & 6 \\ 1 & 1 \end{bmatrix}, \qquad A_{\mathcal{B}}^{-1} A = \begin{bmatrix} 1 & 0 & -.1 & .6 \\ 0 & 1 & .1 & .4 \end{bmatrix}, \qquad A_{\mathcal{B}}^{-1} b = \begin{bmatrix} 1.5 \\ 2.5 \end{bmatrix}.$$

The corresponding "optimal tableau" is

$$\begin{bmatrix} 1 & 0 & -.1 & .6 \\ 0 & 1 & .1 & .4 \end{bmatrix} \begin{bmatrix} x_1 \\ \vdots \\ x_4 \end{bmatrix} = \begin{bmatrix} 1.5 \\ 2.5 \end{bmatrix}.$$

Both 1.5 and 2.5 are fractional, so we can use either the 1st or 2nd equation, say the 2nd. The 2nd equation is

$$x_2 + .1x_3 + .4x_4 = 2.5,$$

so the Gomory cut is:

$$x_2 \leq 2$$

or, equivalently,

$$x_2 + x_5 = 2, \qquad x_5 \geq 0,$$

which we add to LP. The new LP is in standard form. The new optimal BFS is: $x^* = (.75, 2, 0, 1.25, 0)$. The 1st equation in the new "optimal tableau" is:

$$x_1 - .25x_3 + 1.5x_5 = .75$$

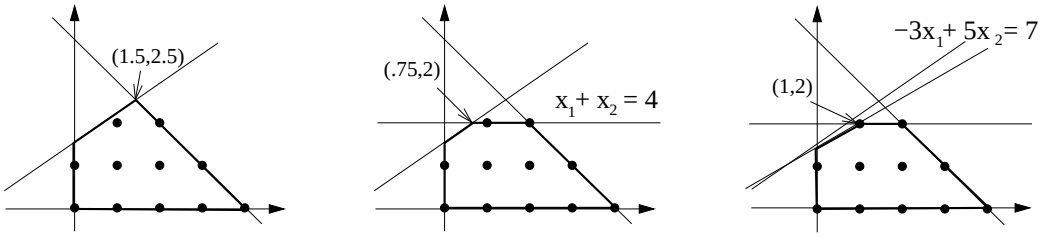
The Gomory cut is

$$x_1 - x_3 + x_5 \leq 0$$

or, equivalently,

$$x_1 - x_3 + x_5 + x_6 = 0, \quad x_6 \geq 0,$$

which we add to LP. The new optimal BFS is: $x^* = (1, 2, 1, 1, 0, 0)$. Since x^* is integer, stop; x^* is optimal for the transformed IP. $(1, 2)$ is an optimal solution of the original IP.



Gomory cuts: $x_2 \leq 2$ and $x_1 - x_3 + x_5 \leq 0 \Leftrightarrow x_1 - (9 + 4x_1 - 6x_2) + (2 - x_2) \leq 0 \Leftrightarrow -3x_1 + 5x_2 \leq 7$

When a new Gomory cut is added, we can resolve LP using the dual simplex method, starting with the optimal tableau of the previous LP. It can be shown that the cutting plane algorithm terminates after a finite number of Gomory cuts with an optimal solution of the IP. However, its practical performance is not good.

Exercises.

6.3.1. Consider the IP (6.3.1), with integer coefficients A, b, c. Suppose that its LP relaxation is unbounded. Prove that the IP has no optimal solution. [Hint: Show that there exists an $z \in Z^n$ satisfying $cz < 0$, $Az = 0,\ z \geq 0$.]

6.4 ADDITIONAL TOPICS

In practice, the best algorithm for solving IP is obtained by combining branch & bound with cutting plane, namely, add valid cuts to each LP generated by the branch & bound algorithm to cut off more fractional solutions. This yields the **branch & cut** algorithm.

There are many ways to generate valid cuts, not necessarily always from an LP optimal solution. For example, the IP

$$\begin{array}{lrcrcrcl} \max & x_1 & + & x_2 & + & x_3 & & \\ \text{s.t.} & x_1 & + & x_2 & & & \leq & 1 \\ & x_1 & & & + & x_3 & \leq & 1 \\ & & & x_2 & + & x_3 & \leq & 1 \end{array}$$

$$x_1 \geq 0,\ x_2 \geq 0,\ x_3 \geq 0 \text{ integer}$$

has three optimal solutions $(1,0,0)$, $(0,1,0)$, $(0,0,1)$, with optimal cost 1. Its LP relaxation has optimal solution $(.5,.5,.5)$, with optimal cost 1.5. If we sum the three inequalities and divide by 2, we obtain

$$x_1 + x_2 + x_3 \leq 1.5.$$

If x_1, x_2, x_3 are integer, then the left-hand side is an integer below 1.5, so it must be below $\lfloor 1.5 \rfloor = 1$. Thus, a valid cut for the IP is

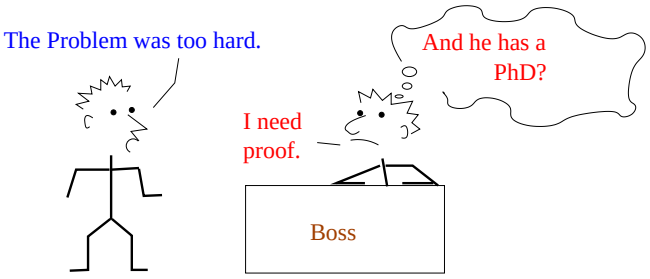
$$x_1 + x_2 + x_3 \leq 1.$$

If we add this constraint to the LP relaxation, its optimal cost becomes 1, the same as that of IP!

In general, we can generate valid cuts by taking positive weighted sums of inequality constraints and suitably rounding up or down the left-hand and right-hand coefficients. There are other ways to generate valid cuts by exploiting the problem structure. For example, when a TSP is formulated as an IP, we can generate valid cuts based on properties of the traveling salesman tours.

7. Computational Complexity

Problems are not created equal. Some are harder than others, maybe much harder. For example, it is fairly easy to check whether a graph has an eulerian path (see Section 2.6), but no one has found an easy way to check whether a graph has a hamiltonian path (see Section 2.7). Imagine that you have been slaving away for days at some discrete optimization problem that your boss assigned you, and have made no progress in solving it. Perhaps it is not you but the problem is just too difficult. But how can you persuade your boss that the problem is difficult?



We need a way to quantify the difficulty of a problem. If the problem is difficult, then instead of trying to solve it exactly, we might settle for finding approximate solutions. We study this topic in this chapter.

7.1 Problem Size and Method Efficiency

For the following three LPs, which looks to be the hardest?

$$\begin{array}{lllll} \min & 3x_1 & +2x_2 & & \\ \text{s.t.} & x_1 & -x_2 & = 4 & \\ & x_1 \geq 0, & x_2 \geq 0. & & \end{array} \tag{LP$_1$}$$

$$\begin{array}{lllll} \min & 3x_1 & +4x_2 & +\cdots & +3x_{100} \\ \text{s.t.} & 2x_1 & -5x_2 & +\cdots & +x_{100} \quad = 5 \\ & -x_1 & +4x_2 & +\cdots & -6x_{100} \quad = -2 \\ & x_1 \geq 0, & x_2 \geq 0, & \cdots, & x_{100} \geq 0. \end{array} \tag{LP$_2$}$$

$$\begin{array}{lllll} \min & 3215x_1 & +2451x_2 & +\cdots & +7134x_{100} \\ \text{s.t.} & 124x_1 & -345x_2 & +\cdots & +764x_{100} \quad = 523 \\ & -997x_1 & +436x_2 & +\cdots & -158x_{100} \quad = -213 \\ & x_1 \geq 0, & x_2 \geq 0, & \cdots, & x_{100} \geq 0. \end{array} \tag{LP$_3$}$$

If you think that LP$_3$ looks hardest, it is probably because LP$_3$ has more variables and constraints than LP$_1$ and its coefficients have a wider range than LP$_2$. Let's quantify this intuition below.

For an LP in standard form:

$$\begin{array}{lll} \min & c'x & \\ \text{s.t.} & Ax = b, & \\ & x \geq 0, & \end{array} \tag{7.1.1}$$

with A, b, c having *integer* coefficients, the **size** (also called length and denoted by L) is the number of binary (i.e., 0/1) bits/digits needed to represent the data A, b, c. The size estimates the length of the data file when stored on a digital computer. If A, b, c have rational coefficients, we can always multiply them by a common denominator to make them integer. We do not consider irrational coefficients since an infinite number of binary digits is required to represent them.

For LP_1, data are $3, 2, 1, -1, 4$.

$3 = 1 \cdot 2^0 + 1 \cdot 2^1 = +(1,1)_{\mathrm{bin}}$

$2 = 0 \cdot 2^0 + 1 \cdot 2^1 = +(0,1)_{\mathrm{bin}}$

$1 = 1 \cdot 2^0 = +(1)_{\mathrm{bin}}$

$-1 = -1 \cdot 2^0 = -(1)_{\mathrm{bin}}$

$4 = 0 \cdot 2^0 + 0 \cdot 2^1 + 1 \cdot 2^2 = +(0,0,1)_{\mathrm{bin}}$

So size of LP_1 is

$$L = 3 + 3 + 2 + 2 + 4 = 14.$$

In general, we need

$$\langle a \rangle = \lfloor \log_2 |a| \rfloor + 2$$

binary bits to store an integer a. Then the size of LP (7.1.1) is:

$$L = \sum_{i=1}^{m} \sum_{j=1}^{n} \langle a_{ij} \rangle + \sum_{i=1}^{m} \langle b_i \rangle + \sum_{j=1}^{n} \langle c_j \rangle \ .$$

It is readily seen that LP_3 has larger size than LP_1 and LP_2.

Intuitively, larger the problem size L, more difficult is the problem. But size alone does not determine the difficulty of a problem. What really matters is the total time taken by a method/algorithm to solve the problem on a digital computer (Turing machine) and, in particular, how fast this time grows with L. Does it grow linearly with L? Or polynomially? Or exponentially? Examples of each type of growth is shown in the table below.

L	10	100	1000
$1000L$ (linear)	10^4	10^5	10^6
$10L^3$ (poly)	10^4	10^7	10^{10}
2^L (expon)	$\approx 10^3$	$\approx 10^{30}$	$\approx 10^{300}$

Linear and polynomial growth are OK. Exponential growth is clearly bad (it would take forever to solve a problem even if its size is only 1000)!

7.2 $\mathcal{P}$, $\mathcal{NP}$, and NP-Hard Problems

A problem is "easy" if we know of a method/algorithm that can find a solution (or determine no solution exists) in time that is polynomial in L. The class of such polynomial-time solvable problems is denoted $\mathcal{P}$. Assuming that each operation $(+, -, \cdot, /, >, <, =)$ takes time that is polynomial in L (which is true for the algorithms studied in earlier chapters and is typically true in general), the total time would be polynomial in L whenever the total number of operations is polynomial in L. Below are some examples.

- Consider the MST problem of Chapter 3. We are given a graph with vertices $V = \{1, ..., n\}$, arcs $A \subset V \times V$, and integer weights ω_{uv}, $(u, v) \in A$. [Rational weights can be scaled up to be integer.] We wish to find an MST. The data are ω_{uv}, u, v for $(u, v) \in A$. The problem size is

$$L = \sum_{(u,v)\in A} (\langle u \rangle + \langle v \rangle + \langle \omega_{uv} \rangle) .$$

  We saw that the MST can be solved by, say, Prim's algorithm in about $(n-1)|A| \leq L^2$ operations. Since L^2 is a polynomial function of L, MST is in $\mathcal{P}$.

- Consider the SP problem of Chapter 4. We are given a digraph with vertices $V = \{1, ..., n\}$, arcs $A \subset V \times V$, nonnegative integer weights ω_{uv}, $(u, v) \in A$, and $s \in V$. [Rational weights can be scaled up to be integer.] We wish to find a SP from s to all $v \in V$ reachable by a path. The data are ω_{uv}, u, v for $(u, v) \in \mathcal{A}$, s. The problem size is

$$L = \sum_{(u,v)\in A} (\langle u \rangle + \langle v \rangle + \langle \omega_{uv} \rangle) + \langle s \rangle.$$

  We saw that the SP problem can be solved by, say, Dijkstra's algorithm in about $(n-1)|A| \leq L^2$ operations. Since L^2 is a polynomial function of L, SP is in $\mathcal{P}$.

- For the LP in standard form (7.1.1) with integer coefficients, the simplex method can, in the *worst case*, take time exponential in L to solve LP. However, there exist other methods, called ellipsoid method and interior point method, that can solve LP in time polynomial in L. Thus LP is in $\mathcal{P}$. The maximum flow problem and minimum cost flow problems of Chapter 5 are special cases of LP, so these problems are also in $\mathcal{P}$.

On the other hand, no one has found polynomial-time algorithms for solving IP or TSP or determining the existence of a hamiltonian cycle. So these problems seem hard. Moreover, these problems are often related in that if one of them is in $\mathcal{P}$, then so is the other! For example, the problem of determing whether a graph $G = (V, A)$ has a hamiltonian cycle is equivalent to the TSP (on complete graph) with arc weight $\omega_{uv} = 0$ if $(u, v) \in A$ and $\omega_{uv} = 1$ if $(u, v) \notin A$. G has a hamiltonian cycle if and only if there is a TSP with zero total weight. Notice that the size L' of this TSP is quadratic in the size L of the hamiltonian

cycle problem, i.e., $L' \leq L^2$,[4] Thus, if the TSP is in $\mathcal{P}$ so that there exists an algorithm to solve it in at most $p(L')$ time, where $p(\cdot)$ is some increasing polynomial function, then by applying this algorithm to the aforementioned TSP, we obtain an algorithm that solves the hamiltonian cycle problem in at most $p(L^2)$ time ($L' \leq L^2$ and $p(\cdot)$ is increasing, so $p(L') \leq p(L^2)$), which is a polynomial in L. Intuitively, the hamiltonian cycle problem is a special case of TSP, so it is no harder than the TSP. [Thus, if TSP is easy to solve, then so is the hamiltonian cycle problem. Equivalently, if the hamiltonian cycle problem is hard, then so is the TSP.]

In general, if a problem can be reduced in polynomial time to another problem with at most a polynomial increase in the size, then we say that the first problem is "polynomial-time reducible" to the second problem. The hamiltonian cycle problem has an important feature, namely, if the problem has a "yes" answer (i.e., the graph has a hamiltonian cycle), then there is a "short" certificate to confirm this. In particular, the cycle itself (in the form of a sequence of vertices and arcs) would be the certificate. Given such a sequence, we can check in polynomial time that indeed every vertex appears in it exactly once (except at the start and end) and that every arc is in the graph. The size of the sequence is also polynomial in the size of the problem. The class of problems for which a positive answer can be verified in polynomial time using a polynomial-sized certificate is denoted $\mathcal{NP}$ (short for "nondeterministic polynomial" since such a problem can be solved in polynomial time by something called a non-deterministic Turing machine). The class of problems for which a *negative* answer can be verified in polynomial time using a polynomial-sized certificate is denoted **co**-$\mathcal{NP}$. Thus the hamiltonian cycle problem belongs to $\mathcal{NP}$ (but is not known to belong to co-$\mathcal{NP}$ since we know of no short certificate for non-existence of hamiltonian cycle). LP with integer coefficients belongs to $\mathcal{NP}$,[5] as well as co-$\mathcal{NP}$. In general, any problem in $\mathcal{P}$ belongs to $\mathcal{NP}$. Another well-known problem belonging to $\mathcal{NP}$ is the 3-SAT problem: Given a logical expression of n variables, with 3 variables per clause, e.g.,

$$(x_1 \vee x_3 \vee x_7) \wedge (\bar{x}_2 \vee x_3 \vee x_5) \wedge \cdots,$$

does there exist $x_1, ..., x_n \in \{\mathrm{T}, \mathrm{F}\}$ to make this expression T?

In 1970, Steve Cook proved a remarkable result that *every problem in $\mathcal{NP}$ is polynomial-time reducible to the 3-SAT problem*! Thus, 3-SAT is in some sense the hardest among all problems in $\mathcal{NP}$. Moreover, people notice that many other problems are at least as hard as 3-SAT in the sense that 3-SAT is polynomial-time reducible to these problems. Among these are IP, TSP, hamiltonian cycle problem, shortest simple

[4] The data of the hamiltonian cycle problem is the graph $G = (V, A)$, so its size is $L = \sum_{(u,v)\in A} (\langle u \rangle + \langle v \rangle)$. We need $|A| \geq |V|$ in order for G to have a hamiltonian cycle, so we can assume without loss of generality that $|A| \geq |V|$ and hence $L \geq |A| \geq |V|$. The data of the TSP are the $|V|^2$ arc weights, each of which requires 1 binary bit to represent, so its size is $L' = |V|^2$. Thus $L' = |V|^2 \leq L^2$.

[5] By strong duality, any optimal solution of the LP has associated with it a feasible solution of the dual LP with equal cost. This feasible solution of the dual LP is the certificate. With a bit of work (using Cramer's rule), we can show that the size of this certificate is polynomial in the size of the LP.

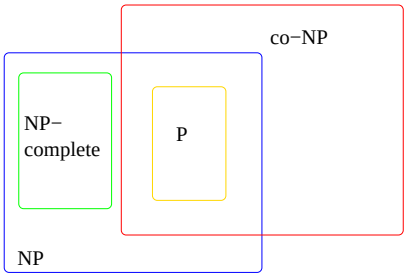
path problem (allowing negative arc weights), knapsack problem, ... These problems are called **NP-hard**. So, a problem is NP-hard if the 3-SAT (or any NP-hard problem) is polynomial-time reducible to it. And a problem is **NP-complete** if it is NP-hard and in $\mathcal{NP}$. 3-SAT and hamiltonican cycle problem are NP-complete. IP and TSP are NP-hard, but not known to be in NP. [If a stranger claims to have found an optimal solution of an IP, how can we verify this claim without solving the IP?] On the other hand, closely related "yes/no" questions (e.g., does the IP have a feasible solution with cost below ϑ?) are NP-complete.

A famous open questions is whether

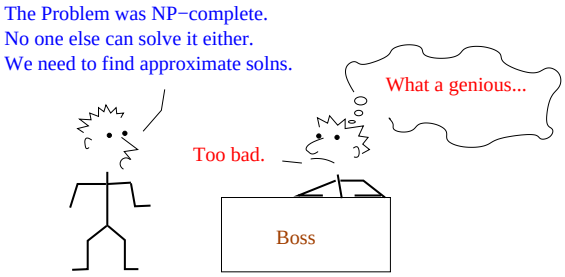
$$\mathcal{P} = \mathcal{NP}?$$

[Many false proofs have been claimed. The bet is on a negative answer.] Another is whether

$$\mathcal{P} = \mathcal{NP} \cap co - \mathcal{NP}?$$



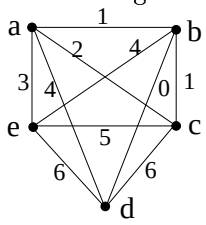
Thus if you can show that the problem your boss assigned you belongs to the class of NP-hard problems (by showing that a known NP-hard problem is polynomial-time reducible to it), then he/she will know that no one else can solve the problem faster either!



7.3 Approximate Solutions: Traveling Salesman Problem

If a problem is NP-hard, then we are unlikely to design a fast algorithm that is guaranteed to find an exact solution always. But if the operation of your company depends on this problem getting solved, then we must settle for designing a fast algorithm that can find exact solution "most of the time" or find inexact/suboptimal solutions. This topic has been extensively studied. In this section, we look at some of these strategies. For concreteness, we illustrate them on the Traveling Salesman Problem (TSP).

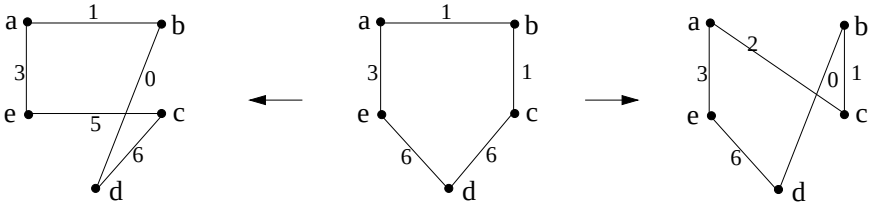
Consider the following complete graph with arc weights shown.



We wish to find a hamiltonian cycle ("tour") of minimum weight. Our aim is to find a suboptimal tour quickly, rather than find an optimal tour (since the problem is NP-hard). One way is the so-called **greedy** algorithm: Starting with the cheapest arc, we continue to add the next cheapest arc to our path, as long as it maintains a simple path, until all vertices are in the path. [This is in the spirit of Kruskal's algorithm for MST.] For the above weighted graph, we would add, say, the arcs

$$(a,b),(b,c),(a,e),(c,d),(d,e).$$

This creates the initial tour a,b,c,d,e,a of weight 17. We next try to improve on this tour through a **local exchange** algorithm. In a 2-exchange algorithm, we try to swap 2 arcs in the current tour with two arcs outside the tour to see if this would decrease the weight. For example, if we swap (b,c), (d,e) with (b,c), (c,e), the resulting tour has weight 15. If instead we swap (a,b), (c,d) with (a,c), (b,d), the resulting tour has weight 12.



(a) (b)

This 2-arc swapping technique can be repeatedly applied to generate a sequence of improving tours. In general, one can use k-exchange algorithm, in which k arcs are swapped ($k = 2,3,4,...$). Practical experience suggests that $k = 2,3$ work best. There are various refinements of this arc-swapping idea using look-ahead to predict which swaps are most promising. However, the tour found by these algorithms can, in the worst case, be far from optimal.

Triangle-Inequality Case.

If the arc weights satisfy the **triangle-inequality**:

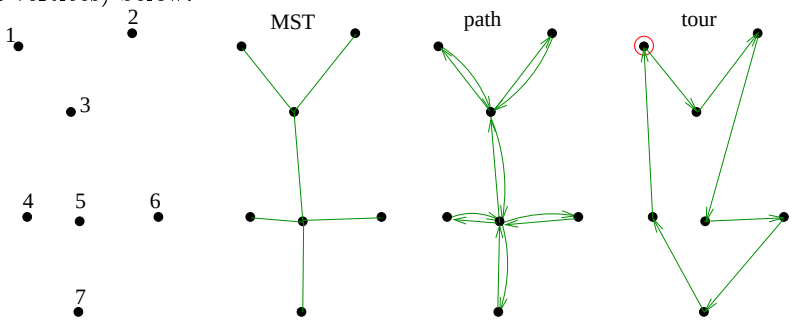
$$\omega_{uv} \leq \omega_{uk} + \omega_{kv} \qquad \text{for all } u,v,k,$$

then it is possible to quickly find a suboptimal tour that is never too far from optimal. To illustrate, suppose that the vertices are points in the plane and ω_{uv} is the Euclidean distance between points u and v. [Euclidean distances satisfies the triangle-inequality.] We construct a tour as follows:

TSP-MST

1. Find an MST.
2. Construct a closed path that starts at some vertex, visits all vertices using only arcs in the MST (with each arc used twice), and returns to the starting vertex.
3. Convert the path to a tour by shortcutting intermediate vertices already visited.

We illustrate this on an example of 7 vertices (with the weight of each arc being the Euclidean distance between its two end vertices) below:



Let W_{MST}, W_{path}, W_{tour} denote the weight of the MST, the path, and the tour constructed above. Then

$$W_{\text{path}} = 2W_{\text{MST}}$$

and, by triangle-inequality, each time we shortcut an intermediate vertex, the weight of the path is reduced. Thus

$$W_{\text{tour}} \leq W_{\text{path}}.$$

Also, removing any arc from an optimal tour yields a spanning tree. Since the removed arc has positive weight, the spanning tree has lesser weight than the optimal tour (whose weight we denote by W_{TSP}). Being a spanning tree, it has greater or equal weight than the MST. This implies

$$W_{\text{TSP}} \geq W_{\text{MST}}.$$

Putting the above three relations together yields

$$W_{\text{tour}} \leq 2W_{\text{TSP}}.$$

Thus, the tour we constructed is guaranteed to be within a factor of 2 in weight from the optimal tour! Since an MST can be found quickly (Chapter 3) and the shortcutting can also be done quickly, this is a pretty fast algorithm. We can further improve on the tour using, say, an exchange algorithm.

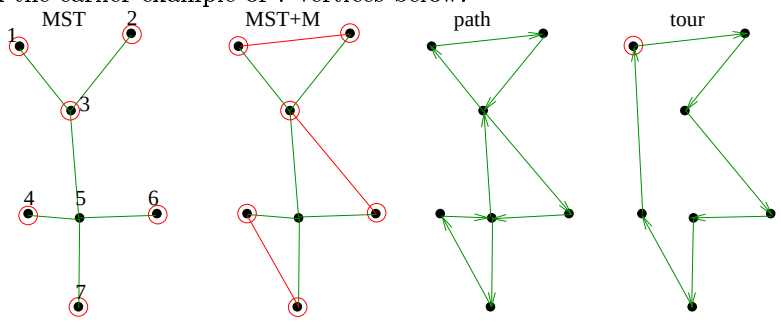
N. Christofides showed in 1970s that, by also using minimum weight matching (optimal assignment), the factor 2 can be further improved to $\frac{3}{2}$. His algorithm constructs a tour as follows:

TSP-Christofides

1. Find an MST.

2. Let $V_{\text{odd}} = \{\text{vertices of odd degree in MST}\}$. There is an even number of these vertices.
3. Find a matching $M \subseteq V_{\text{odd}} \times V_{\text{odd}}$ (i.e., each vertex in V_{odd} has exactly 1 arc of M joined to it) of minimum weight.
4. Adding M to MST gives each vertex even degree. Construct a closed path.
5. Convert the path to a tour by shortcutting intermediate vertices already visited.

We illustrate this on the earlier example of 7 vertices below:



Let W_{MST}, W_M, W_{path}, W_{tour} denote the weight of the MST, the matching M, the path, and the tour constructed above. Then

$$W_{\text{path}} = W_{\text{MST}} + W_M$$

and, by triangle-inequality, we have

$$W_{\text{tour}} \leq W_{\text{path}}.$$

Also, as we argued earlier,

$$W_{\text{TSP}} \geq W_{\text{MST}}.$$

Finally, by triangle-inequality, an optimal tour on all vertices has greater weight than an optimal tour on V_{odd}. Moreover, the arcs of an optimal tour on V_{odd} form two matchings M_1 and M_2 on V_{odd}. [As we move along this tour, the 1st, 3rd, 5th arcs are in M_1 and the 2nd, 4th, 6th arcs are in M_2, etc.] Thus

$$W_{\text{TSP}} \;\geq\; W_{\text{TSP on } V_{\text{odd}}} \;=\; W_{M_1} + W_{M_2} \;\geq\; 2W_M.$$

Putting the above four relations together yields

$$W_{\text{tour}} \;\leq\; \frac{3}{2} W_{\text{TSP}}.$$

Clever, no? For a long time, $\frac{3}{2}$ was the best approximation factor achievable in polynomial time for the Euclidean TSP. The drawback with Christofides' algorithm is that it requires solving an additional matching problem.

Index

Adjacency matrix, 11
Arc, 3
  list, 12
Augmenting path, 46
Breadth-first search, 15
Branch and bound, 59
Branch and cut, 66
Cardinality, 3
co-NP, 71
Connected component, 13
Covering, 49
Cutting plane, 64
Cycle, 5 and 9
Degree, 10
Depth-first search, 15
Digraph, 3
  residual, 46
  strongly connected, 6
Dual simplex method, 62
EPATH, 20
Exchange algorithm, 73
Flow, 42
  feasible, 53
  value, 42
Graph, 3
  bipartite, 48
  complete, 8
Greedy algorithm, 73
Indegree, 10

Integer program, 57
  feasible solution, 58
  feasible region, 58
  optimal solution, 58
  optimal cost, 58
Linear program relaxation, 59
Matching, 49
  maximum weight 54
MAXFLOW, 46
Maximum flow problem, 42
Minimum cost flow, 52
Minimum cost multicommodity flow, 55
Minimum cut problem, 43
MST, 24
MST-arcswap, 30
MST-Kruskal, 27
MST-Prim, 26
NP, 71
NP-complete, 72
NP-hard, 72
Outdegree, 10
P, 70
Path, 4 and 9
  augmenting, 46
  closed, 5 and 9
  eulerian, 18
  hamiltonian, 21
  shortest, 33
  simple, 5 and 9
Problem size, 69
SP, 34

SP-Dijkstra, 36
SP-Bellman-Ford, 38
Subgraph, 13
Traveling salesman problem, 73
Triangle inequality, 73
TREE, 14
Tree, 14
  spanning, 14
Vertex, 3
Weight
  of a path, 33
  of a spanning tree, 23