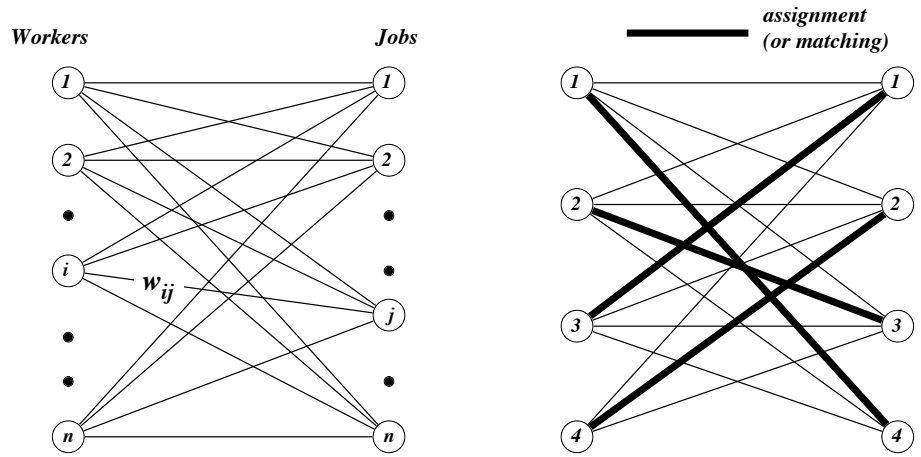
Chapter 5

Combinatorial Optimization and Complexity

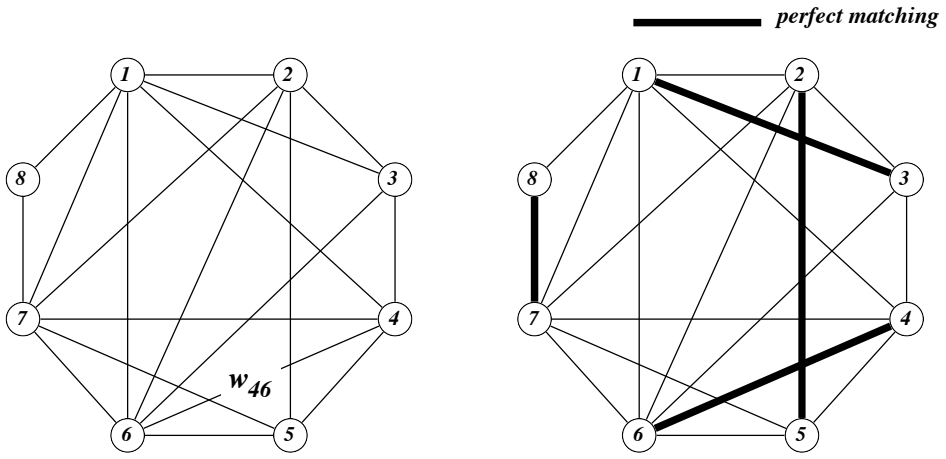
5.1 Examples

Efficiently Solvable Problems (Polynomially Solvable Problems)

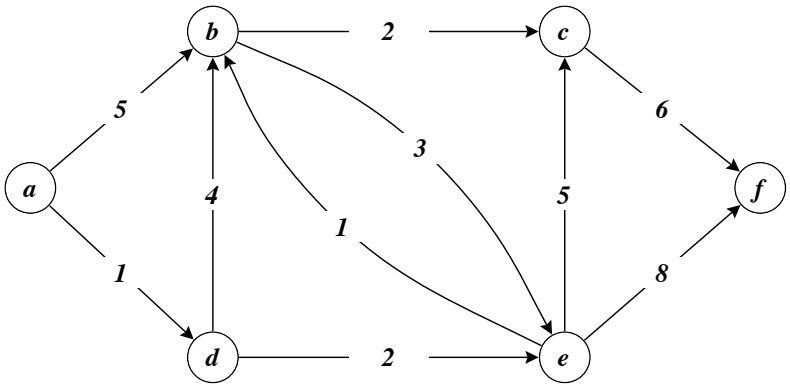
- Assignment Problem (Bipartite Perfect Matching Problem)
  Given an $n \times n$ matrix $W = [w_{ij}]$, select n entries one for each row and column so as to maximize (or minimize) the sum.



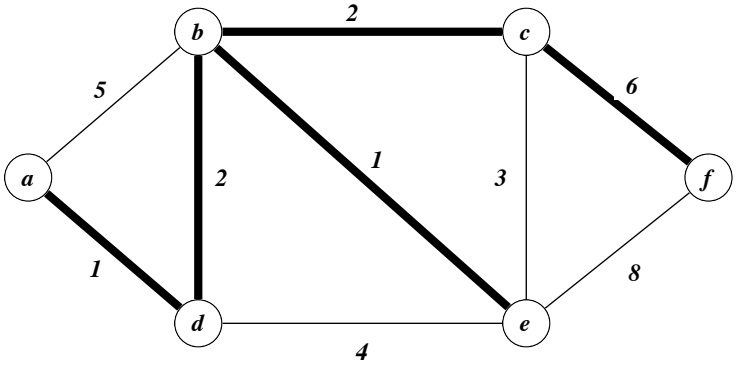
- Perfect Matching Problem
  Given a graph with edge weights, find a perfect matching (a set of edges that cover each node exactly once) of maximum total weight.



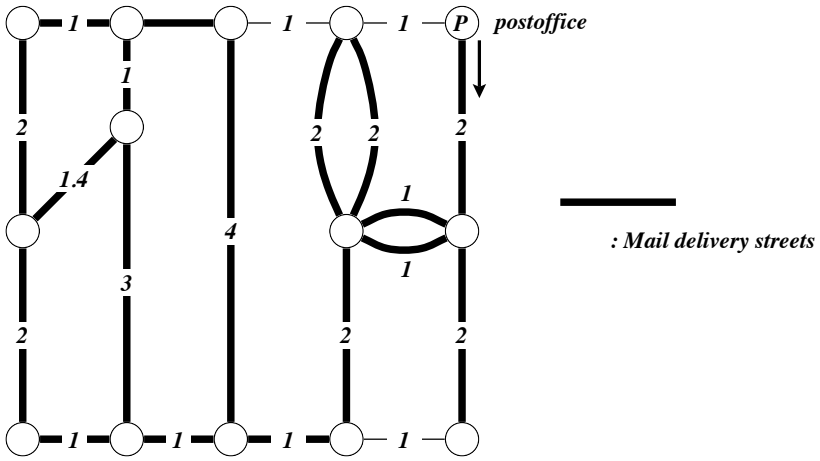
- Shortest Path Problem
  Given a directed graph with positive edge weights (e.g. distance, cost), find a path between given two nodes that minimizes the total weight (i.e. the sum of the weights of its edges).



- Minimum Spanning Tree Problem
  Given a graph with edge weights (e.g. distance, cost), find a spanning tree that minimizes the total weight (i.e. the sum of the weights of its edges).



- Chinese Postman Problem (slightly extended)
  Given a graph with edge weights (e.g. distance, cost) and a connected subgraph, find a route of visiting each edge of the subgraph at least once that minimizes the total weight.

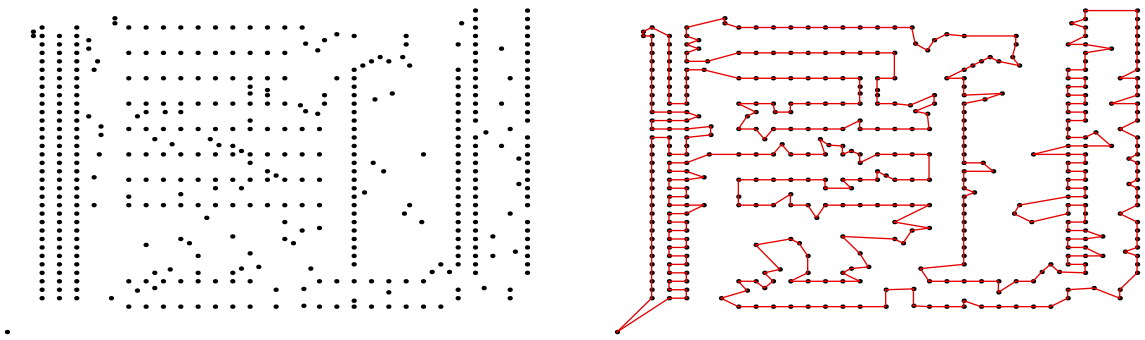


- Maximal Flow Problem
  Given a directed graph with edge capacities (e.g. volume/second), find a flow from a given source node to a given sink node that maximizes the total flow.

- Minimum Cost Flow Problem
  Given a directed graph with edge weight (e.g. cost) and demands for each nodes, find a flow satisfying all demands that minimizes the total cost.

Hard Problems (NP-Complete Problems)

- Traveling Salesman Problem (TSP)
  Given a complete graph with edge weights (e.g. distance, cost), find a tour of visiting each node exactly once that minimizes the total weight.

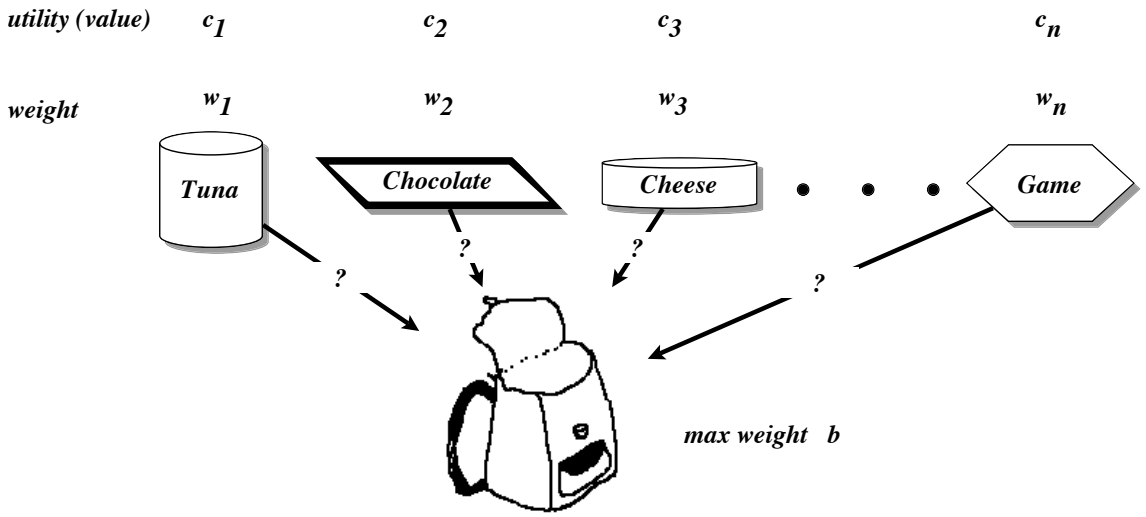


Eucledian TSP Problem and an Optimum Tour (a TSPLIB problem, pcb442.tsp)

- 3-dimensional Assignment Problem
  Given a 3-dimensional $n \times n \times n$ matrix $W = [w_{ijk}]$, select n entries one for each row, column and level so as to maximize (or minimize) the sum.

- Longest Path Problem
  Given a directed graph with positive edge weights (e.g. distance, cost), find a **simple** path between given two nodes that maximizes the total weight.

- Knapsack Problem
  From n items with given values, select items with total weight at most b (a given constant) so as to maximize the total value.



- Set Cover Problem
  Let U be a finite set and let $\mathcal{S}$ be a family of subsets of U. Find a minimum-cardinality subfamily $\mathcal{C}$ of $\mathcal{S}$ that *covers* the ground set U, i.e. the union of all sets in $\mathcal{C}$ is U.

- Satisfiability Problem
  Given a boolean expression $B(x) := \bigwedge_{i=1}^{m} C_i$ in conjunctive normal form where C_i is the disjunction of one or more literals (e.g. $(x_1 \bigvee \neg x_2 \bigvee x_5)$), decide whether there is an assignment $x \in \{0,1\}^n$ with $B(x) = 1$.

5.2 Efficiency of Computation

What do we mean by "the assignment problem is efficiently solvable" ?

Efficient Algorithms

- Intuitive Explanation

  An algorithm is *efficient* if the time necessary to solve a problem instance does not increase too fast as the problem size increases.

- More Precise Definition

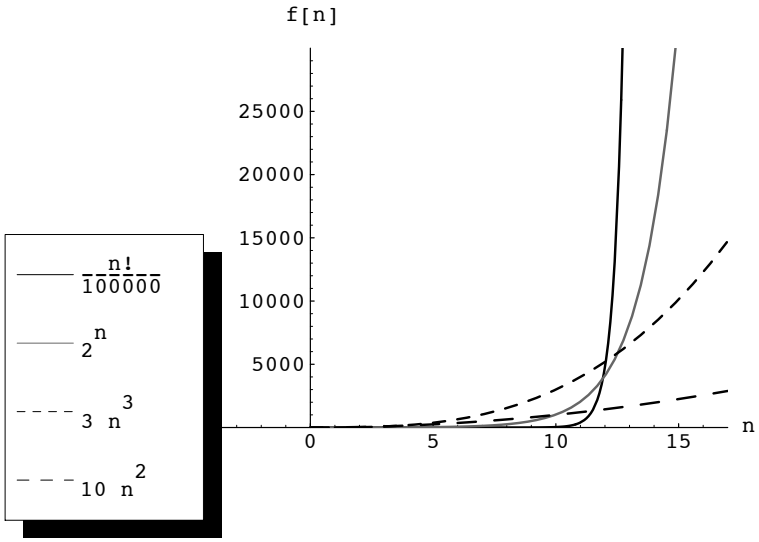
  An algorithm is *efficient* (*good*, *polynomial*) if the time necessary to solve any instance of input size L is bounded above by a polynomial function of L. (Thus it does not

increase as fast as an exponential function such as 2^L.)

The *size* of a problem instance is the number of bits necessary to represent the problem (in binary encoding).

- Even More Precise Definition
  One still needs to set up a machine model to define "time". For this, the Turing machine is standard. It is a very simple mathematical model of digital computers with an infinite tape device. For example the machine can do elementary arithmetic operations of two numbers in polynomial time in the size of the two numbers. Thus, to prove an algorithm runs in a polynomial time in the Turing machine, it is essential to show that both the number of arithmetic operations the algorithm executes and the sizes of the operands (numbers) are bounded by polynomial functions of the input size.

Polynomial Functions and Exponential Functions



- There is an $O(n^3)$ algorithm for the assignment problem.
- No polynomial algorithm is known for the traveling salesman problem.

5.2.1 Evaluating the Hardness of a Problem

How can we distinguish the hardness of problems listed in Section 5.1? More explicitly,

> Which problems can be solved in polynomial time?
> Which ones cannot be?

- Easy Problems (Class P)

  A problem is in the class P (or polynomially solvable) if it admits a polynomial algorithm.

- To classify "possibly easy problems" we define two classes NP and co-NP.

- First we consider, for each optimization problem, the associated decision problem (problem demanding only YES or NO answer):

For any fixed (rational) number K, decide whether there exists a feasible solution with objective value better than K.

  (Here "better" means "greater" for maximization, and "smaller" for minimization.)

- A Universe of Problems (Class NP)

  A problem is in the class NP (nondeterministic polynomial) if YES answer for the associated decision problem admits a certificate with which one can prove the correctness of the answer in polynomial time. (Such a cerficate is called succinct certificate.)

  Proposition 5.1 *The traveling salesman problem (and every problem in Section 5.1) is in NP.*

  Proof. Let K be any number. If there is a tour in the given graph whose total length is less than K, such a tour itself is a succinct certificate. We can check the length is less than K by simple computation with a polynomial number of elementary arithmetic operations. ∎

- Another Universe of Problems (Class co-NP)

  A problem is in the class co-NP (dually nondeterministic polynomial) if NO answer for the associated decision problem admits a certificate with which one can prove the correctness of the answer in polynomial time.

- A Key Proposition:

  Proposition 5.2 *Every problem in P belongs to both NP and co-NP.*

- Important Remark:

  Remark 5.3 *If a problem belongs to both NP and co-NP, then it often belongs to P.*

- Polynomial Reduction:

  We say a decision problem A is polynomially reducible to a decision problem B, denoted by A $\propto$ B, if there is a polynomial time transformation of every instance of A to an instance of B that preserves the answer.

  The polynomial reducibility provides a convenient way to say one problem is not harder than another. That is:

  Proposition 5.4 *If $A \propto B$, then A is not harder than B, i.e., $B \in P$ implies $A \in P$.*

  Remark 5.5 *One can easily see that (the decision problem of) the assignment problem is polynomially reducible to the 3-dimensional assignment problem.*

- The Hardest Problems in NP (and in co-NP)

  Here is the most important theorem in Computational Complexity Theory:

  Theorem 5.6 (Cook's Theorem) .
  The class of hardest problems in NP, that is the class of problems in NP to each of which every problem in NP is polynomially reducible, is nonempty. In particular, this class, defined as the class NP-complete or NPC, contains the satisfiability problem.

  It has been proved by various people that all hard problems in Section 5.1 are NP-complete.

- Important questions remain open and appear to be hard:

  (1) Is P = NP? (Probably no. If yes, P = NP = co-NP).

  (2) Is P = NP $\cap$ co-NP?

  See Figure 5.1 for the complexity map assuming the answer to (1) is no.

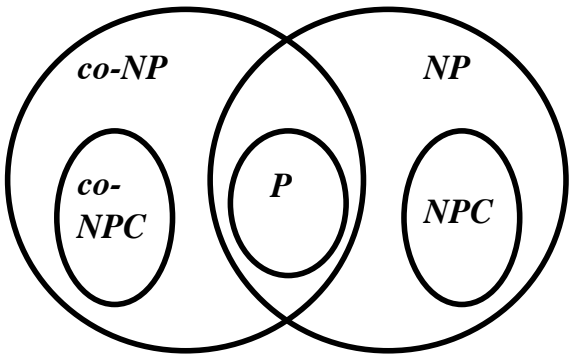


Figure 5.1: The complexity map assuming $P \neq NP$.

5.2.2 A Little History

1937	Turing Machine	Alain Turing
1953	Polynomial vs Exponential time	von Neumann
1965	Class $\mathcal{L}$ (a subclass of NP)	Alan Cobham
1965	Good Algorithms (P), Good Characterizations (NP)	Jack Edmonds
1970	Axioms for Complexity Measures	Michael Rabin
1971	Class $\mathcal{L}^+$ (equiv. to NP), Completeness	Stephen Cook
1972	Classes P, NP, NPC	Richard Karp
1979	Class #P, #P-complete	Leslie Valiant

Note: Class #P is a universe of counting problems. Surprisingly, even if a decision problem is polynomially solvable, the associated counting problem might be hard. Valiant proved, for example, the counting problem for the assignment problem is in #P-complete, implying it is not easier than any NP-complete problems.

5.3 Basic Graph Definitions

5.3.1 Graphs and Digraphs

Graphs or digraphs are a mathematical formalization of diagrams such as those illustrated in Figure 5.2.

A *graph* or *undirected graph* is a pair $G = (V, E)$ where $V = V(G)$ is a finite set called the *vertex* (or *node*) set and $E = E(G)$ is a subset of $\binom{V}{2}$, called the *edge* (or *arc*) set. Each edge $\{i, j\}$ is often denoted by ij or ji, both of which represent the same edge. When ij is an edge ($ij \in E$), the vertices i and j are called *endvertices* (or *endnodes*) of e, and they are said to be *adjacent*. We also say the edge $e = ij$ *joins* i and j, e is *incident to* i and j and conversely.

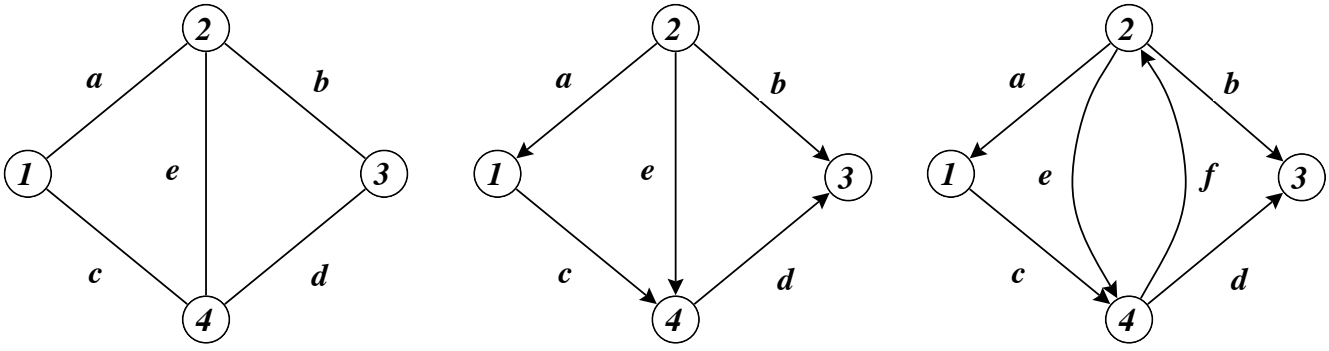


Figure 5.2: A graph, an oriented graph and a directed graph

A *directed graph* or a *digraph* is a pair $G = (V, E)$ where everything is the same as for undirected graphs except that E is a set of ordered pairs of distinct vertices. Each edge is

denoted by (i, j) or ij ($\neq ji$). Clearly $|E| \leq |V| \times (|V| - 1)$. If $e = ij \in E$, i is called the *tail* and j is the *head* of e. We say an edge e is *directed from i to j*. In order to distinguish a digraph from a graph, one might use $\overrightarrow{G}$, $\overrightarrow{E}$, $\overrightarrow{ij}$ instead of G, E, ij.

An *oriented graph* is a digraph $G = (V, E)$ with at most one of (i, j) or (j, i) is an edge, for each $i, j \in V$.

For example, the left one in Figure 5.2 illustrates a graph with $V = \{1, 2, 3, 4\}$ and $E = \{a, b, c, d, e\}$. Each edge is denoted by a line segment joining the two endvertices, and therefore $a = 12 = 21$, $b = 23 = 32$, $c = 14 = 41$, etc. The right most figure is a digraph with the vertex set $V = \{1, 2, 3, 4\}$ and the edge set $\overrightarrow{E} = \{a, b, c, d, e, f\}$, where each edge is an ordered pair of vertices, $a = \overrightarrow{21} (\neq \overrightarrow{12})$, $e = \overrightarrow{24}$, $f = \overrightarrow{42}$, etc. In this digraph, the edge a is directed from its tail 2 to its head 1. The head (tail) of e is the tail (head) of f, and thus they are directed oppositely and parallel.

Sometimes it is useful to introduce multiple *parallel edges* (i.e. edges between the same pair of vertices), or to have a *loop* (i.e. an edge connecting the same vertex). One can treat such a general structure by modifying the definitions. Some people call such structures *multigraphs*. and others call them *graphs* and define our graphs as *simple graphs*. Since our lecture does not profit from this generality, our definition is appropriate.

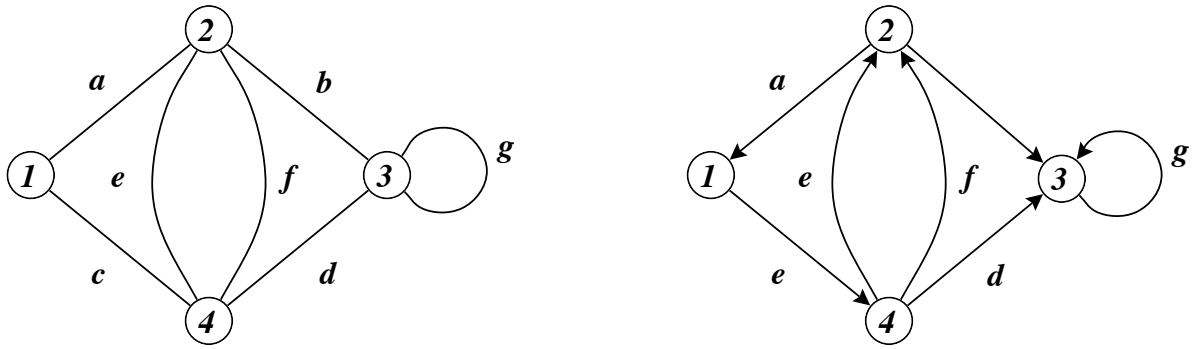


Figure 5.3: Multigraphs, that we do NOT treat in the lecture

Two graphs G and H are called *isomorphic* (denoted by $G \simeq H$) if there is a bijection $f : V(G) \to V(H)$ such that for any $i, j \in V(G)$, i and j are adjacent in G if and only if $f(i)$ and $f(j)$ are adjacent in H.

The *order* of a (di)graph $G = (V, E)$ is $|V|$ and its *size* is $|E|$. We use the letter n for $|V|$ and m for $|E|$, unless otherwise stated.

A *degree* or *valency* $\deg(v)$ of a vertex v in a (di)graph G is the number of edges incident to v. A vertex of degree 1 is called a *leaf*.

Proposition 5.7 *Every graph has an even number of vertices with odd degrees.*

Proof. Since every edge has two endvertices, we have

$$\sum_{i \in V} \deg(i) = 2|E|.$$

Since the RHS is even, we must have an even number of odd degrees in the LHS. ∎

A graph G' is said to be a *subgraph of* G (denoted by $G' \subseteq G$) if $V(G') \subseteq V(G)$ and $E(G') \subseteq E(G)$. We say G' is *contained in* G if $G' \subseteq G$. A subgraph G' is *spanning* if $V(G') = V(G)$.

For any subset V' of V, the subgraph of $G = (V, E)$ with the vertex set V' and the edge set E' consisting of all edges of G whose endvertices are in V' is called the *subgraph of* G *induced by* V', and denoted as $G[V']$. (i.e. $E' = \{ij \in E : \{i, j\} \subseteq V'\}$.) For any $i \in V$, the induced subgraph $G[V - i]$ is simply denoted by $G - i$.

Similarly, for any subset E' of E, the subgraph $G' = (V', E')$ of G whose vertex set V' is the set of vertices of G which are incident to some edge of E' is called the *subgraph of* G *induced by* E' and denoted by $G[E']$. (i.e. $V' = \{i \in V : ij \in E' \text{ for some } j\}$). For any subset E' of E, $G - E'$ denotes the subgraph of G defined by $(V, E - E')$. Similarly, for $E' \subseteq \binom{V}{2} - E$, we use $G + E'$ to denote the graph $(V, E + E')$. For simplicity, we use $G - e$ and $G + e$ for $G - \{e\}$ and $G \cup \{e\}$, respectively.

A *empty* or *null* graph is a graph G with no edges (i.e. $E(G) = \emptyset$).

A *path* graph is a graph isomorphic to the graph $G = (V, E)$ with $V = [n]$ and $E = \{12, 23, 34, \ldots, (n-1)n\}$. A *cycle* graph is a graph isomorphic to the graph $G = (V, E)$ with $V = [n]$ and $E = \{12, 23, 34, \ldots, (n-1)n, n1\}$.

A *complete* graph is a graph in which all pairs of distinct vertices are joined by an edge. Thus, a complete graph of order n contains exactly $n(n-1)/2$ edges. Of course, any two complete graphs of order n are isomorphic, and we use K_n to denote this class.

A graph $G = (V, E)$ is called *bipartite* if the vertex set can be partitioned into two sets $V = V_1 \cup V_2$ such that there are no edges between any two vertices in the same set V_1 or V_2. Any such partition is called *bipartition*. Often a bipartite graph is given by a specific bipartition, and thus may be denoted as $G = (V_1, V_2, E)$.

Problem 5.3.1 *Prove that no bipartite graph can contain a cycle of odd length. (In fact, this property is sufficient and thus characterizes bipartite graphs. How can we prove the sufficiency?)*

A *complete bipartite* graph is a bipartite graph with bipartition (V_1, V_2) in which every vertex in V_1 is joined (by an edge) to each vertex in V_2. When $|V_1| = p$ and $|V_2| = q$, such a graph is denoted by $K_{p,q}$.

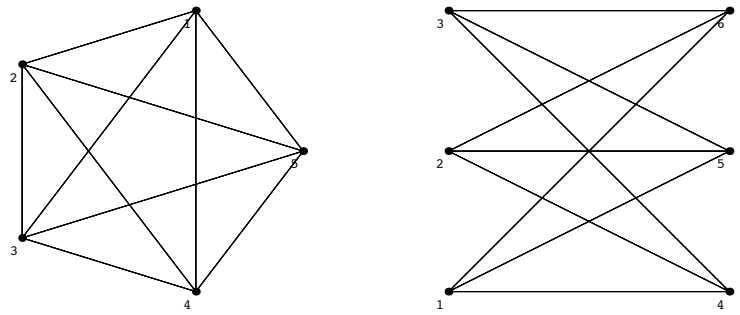


Figure 5.4: K_5 and $K_{3,3}$

A *walk* (of *length* k) in a graph $G = (V, E)$ is a sequence $(v_0, e_1, v_1, e_2, v_2, \ldots, v_{k-1}, e_k, v_k)$ in which all v_i's are in V and each e_i is an edge joining the vertices v_{i-1} and v_i (i.e. $e_i = v_{i-1}v_i \in E$ for $i = 1, 2, \ldots, k$). The vertices v_0 and v_k are called the *origin* and the *terminus* of the walk, and the vertices $v_1, \ldots, v_{k-1}$ are *internal* vertices of the walk. We say a walk *connects* v_0 and v_k. A walk is *closed* if $k > 0$ and $v_0 = v_k$. Since we do not allow parallel edges, a walk can be denoted simply by the sequence of its vertices: $(v_0, v_1, v_2, \ldots, v_k)$.

A *trail* is a walk with distinct edges (but possibly with some vertices used more than once). A *path* is a walk with distinct vertices (and thus distinct edges). A *cycle* is a closed trail whose origin and the internal vertices are distinct.

A path (or a cycle) is often identified with their underlying graph consisting of its vertices and edges.

A graph G is called *connected* if for every pair u, v of vertices, there exists a path connecting u and v. A *connected component* of a graph G is a connected subgraph G' of G which is maximal, i.e. there is no connected subgraph of G properly containing G'.

A *forest* is a graph containing no cycle. A *tree* is a connected forest. A *tree* (*forest*, *cycle*, etc.) in a graph is a subgraph of G which is tree (forest, cycle, etc.).

Proposition 5.8 *Let T be a tree of order n. Then the following properties hold:*

(a) *T contains at least two leaves for $n \geq 2$.*

(b) *T has exactly $n - 1$ edges.*

(c) *There is a unique path between any two vertices in T.*

(d) *For any edge $f \in E(T)$, $T - f$ contains exactly two connected components.*

Proof. First we show (a). Assume $n \geq 2$. Take any vertex v_0. Then take any edge e_1 incident to v_0. e_1 exists because G is connected. Let v_1 be the other endvertex of e_1. If $deg(v_1) \neq 1$, one can extend the sequence to generate a walk $(v_0, e_1, v_1, e_2, v_2)$. This procedure

can continue to generate a walk $P = (v_0, e_1, v_1, e_2, v_2, \ldots, e_k, v_k)$ as long as the last vertex v_k is not a leaf. Since T contains no cycle, P contains no repeated vertices and is thus a path. Since V is finite, this sequence must end with a vertex of degree 1. If $deg(v_0) = 1$, the claim is proved. Otherwise, one can apply the same process using another edge incident to v_0 to find another vertex of degree 1.

To prove (b), use induction on n. For $n = 1$, the statements are trivial. Let $n \geq 2$, and assume by induction that (b) is true for any smaller n. By (a), we have a vertex v of degree 1. By removing the vertex and the unique edge incident to it from the graph, we obtain a tree of size $n - 1$. By the induction hypothesis, this graph has $n - 2$ edges. Since we have one more edge in T, we have $n - 1$ edges in T.

(c) and (d) are left for exercise. ∎

Problem 5.3.2 *Let T be a connected graph of order n. Prove that the following statements are equivalent:*

(1) *T is a tree.*

(2) *T has exactly $n - 1$ edges.*

(3) *For each edge $f \in E(T)$, $T - f$ is not connected.*

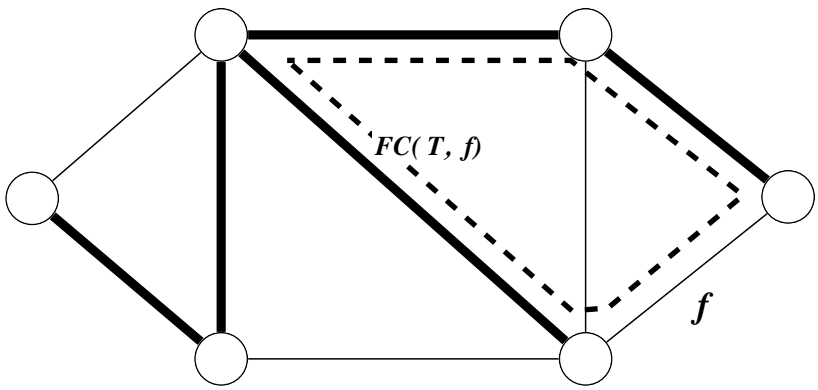
For any graph $G = (V, E)$ and any subset S of V, we denote by $\delta(S)$ the set of edges which connect a vertex in S and a vertex in $V - S$. The set $\delta(S)$ is called the *cut set* with respect to S.

It follows from Proposition 5.8 (c) and (d) we have two basic operations for spanning trees in a graph.

Proposition 5.9 *Let T be a spanning tree in a connected graph G, and f be any edge not in T. Then,*

(a) *$T + f$, the graph obtained by adding the edge f to T, contains a unique cycle, called the* fundamental cycle *of T with f, denoted by* $\mathrm{FC}_G(T, f)$.

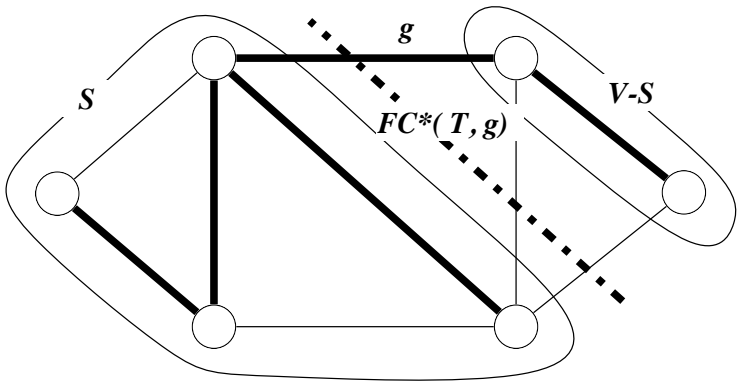
(b) *For each $g \in E$, $T + f - g$ is a spanning tree if and only if $g \in \mathrm{FC}_G(T, f)$.*



Proposition 5.10 *Let T be a spanning tree in a connected graph G, and g be any edge in T. Then,*

(a) *$T - g$ contains exactly two connected components that are of form $T[S]$ and $T[V - S]$, for some $S \subset V$. (The cut set $\delta(S)$ is called the* fundamental cut *of T with g, denoted by $\mathrm{FC}^*_G(T, g)$).*

(b) *For any $f \in E$, $T - g + f$ is a spanning tree if and only if $f \in \mathrm{FC}^*_G(T, g)$.*



Chapter 6

Polynomially Solvable Problems

The easy problems listed in the first part of Section 5.1 are in the class P, that is, they all admit a polynomial (= efficient, good) algorithm. Known polynomial algorithms, each of which is specific to the target problem, look all so different from each other. The purpose of the present chapter is to present a succinct certificate of optimality for some of these problems. It is important to observe that all efficient algorithms are the same in one sense: they all find an optimal solution and a succinct certificate in polynomial time.

6.1 Minimum-weight Spanning Tree Problem

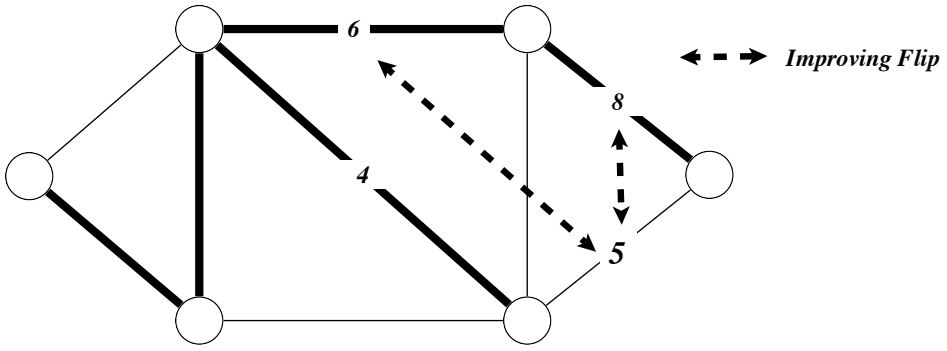
- Minimum-weight Spanning Tree (MST) Problem

  Given a graph $G = (V, E)$ with edge weights $w(e)$ $(e \in E)$, find a minimum-weight spanning tree (MST), i.e. a spanning tree T^* that minimizes the total weight $w(T) = \sum_{e \in T} w(e)$.

  (Note that we represent a tree as a subset of edges.)

- This problem is clearly in NP. The following theorem not only shows it is in co-NP, but suggests an obvious algorithm and proves the correctness of many algorithms.

  For a spanning tree $T \subseteq E$, $f \in E \backslash T$ and $g \in \mathrm{FC}(T, f)$, the replacement of T by the spanning tree $T' = T + f - g$ is called an *flip operation*. An edge flip is called *improving* if $w(T') < w(T)$.



Theorem 6.1 *A spanning tree T^* is an MST if and only if it admits no improving flip.*

Proof. (Here we assume that all weights are distinct.) Let T be a spanning tree with no improving flip, and suppose it is not optimal. Take any MST T^*. Without loss of generality, we may assume $T \cap T^* = \emptyset$, since we can contract all edges in common. Let f be the edge of minimum weight. It is clear that $f \in T^*$. Then any edge flip on T with f and $g \in \mathrm{FC}(T, f)$ is improving. ∎

- This theorem immediately shows the following algorithms find an MST correctly.

  Flip Algorithm Start from any spanning tree $G(T)$ and apply any improving flips until no such flip is possible.

  Greedy Algorithm 1: Kruskal's Algorithm Start from the forest $G(T) = (V, \emptyset)$, and add to T any smallest weight edge $e \in E \backslash T$ that can be added to T without creating a cycle.

  Greedy Algorithm 2: Prim's Algorithm Start from the any single node tree $G(T) = (\{v\}, \emptyset)$, and add to T any smallest weight edge $e \in E \backslash T$ that can be added to T without creating a cycle and preserve the connectivity.

- Exercise How many flips does Flip algorithm perform? Upper bounds?

- Extensions?

  One can solve a more general class of problems by the greedy algorithms. This include the minimum weight basis problem:

  Given $m \times E$ matrix A and weights $w(e), e \in E$, find $B \subseteq E$ such that $A_{\cdot B}$ is a basis of A (i.e. nonsingular) and $w(B)$ is minimum over all bases.

  There is a characterization of problems that can be solved by the greedy algorithm in terms of matroids. Matroids is a combinatorial abstraction of linear independence/dependence.

6.2 Bipartite Perfect Matching Problem

- Matching

  A *matching* in a graph $G = (V, E)$ is a subset M of edges such that each node of G is the endnode of at most one member of M. An endnode of an edge in M is called *saturated by* M. A matching M is called *perfect* if every node is saturated.

- Bipartite Perfect Matching Problem Given a bipartite graph $G = (V_1, V_2, E)$, find a perfect matching M in G.

- Hall's Theorem

Theorem 6.2 *There exists a matching M that saturates every node of V_1 if and only if*

$$|N(S)| \geq |S| \text{ for all } S \subseteq V_1, \tag{6.1}$$

where $N(S) = \{v \in V_2 : v \text{ is adjacent to some node in } S\}$, the neighbor set of S.

Proof. The only if part is trivial, since if there is a matching saturating all V_1 nodes, $N(S)$ contains all nodes matched with S.

To prove the if part, we present a finite algorithm which for any bipartite graph G finds either a matching meeting every node of V_1 or a set $S \subseteq V_1$ violating the condition (6.1).

The general stage of the algorithm starts with any matching M and a node $v \in V_1$ not saturated by M. We set $M = \emptyset$ at the beginning. The algorithm tries to augment M to saturate v. For this, we start to grow a tree T starting at v.

(0) Let $S = S_1 = \{v\}$, $T = (S, \emptyset)$ and $S_2 = \emptyset$. (Note $S_1 \subseteq V_1$ and $S_2 \subseteq V_2$.)

(1) Let $W_2 = N(S_1) \backslash S_2$ and add the edges connecting S_1 to W_2 to the tree T. If $W_2 = \emptyset$, the set S violates (6.1) and stop. (See Fig. 6.1.)

(2) If W_2 contains an M-unsaturated node w, we find a path from v to w that uses non-matching and matching edges alternately. This means we can reverse the matching edges and the non-matching edges along the path to augment M so that v is newly saturated by the new M. (See Fig. 6.2.)

(3) Otherwise, all nodes in W_2 have their matched nodes, which we newly call S_1. Add S_1 to S, W_2 to S_2, the maching edges incident with W_2 and S_1 to the tree T. Repeat from (1).

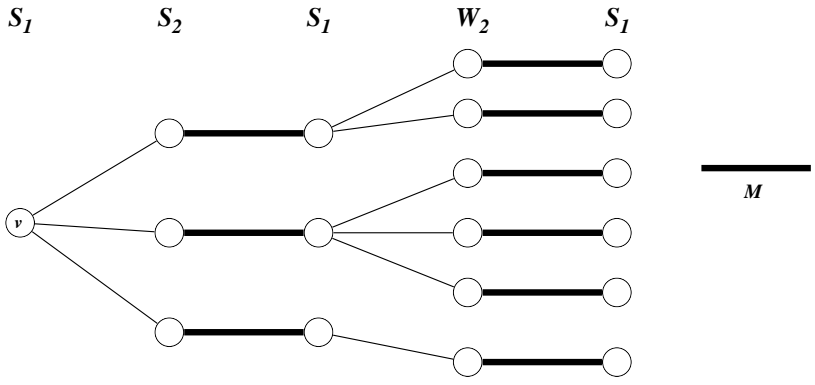


Figure 6.1: Case (1): a violating set $S = \cup S_1$ is found

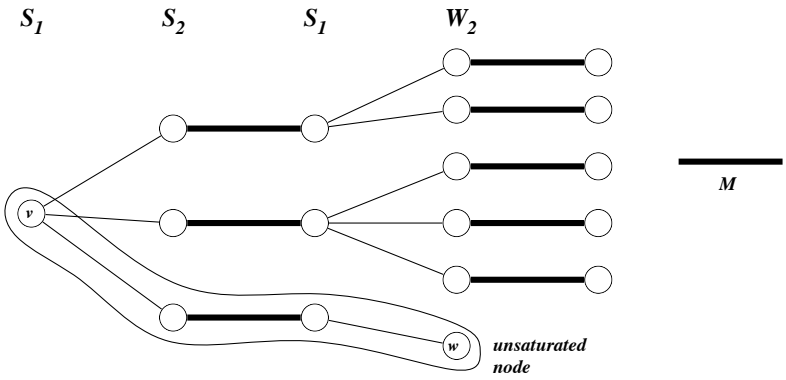


Figure 6.2: Case (2): an augmenting path is found

6.3 Assignment Problem

- Assignment Problem

  Given an $n \times n$ matrix $W = [w_{ij}]$, select n entries one from each row and each column so as to maximize (or minimize) the sum.

  Equivalently, given a weighted complete bipartite graph $G = (V_1, V_2, E)$ with $|V_1| = |V_2| = n$, find a perfect matching $M \subseteq E$ of maximum total weight.

- IP Formulation of the Assignment Problem

  Let $X = [x_{ij}]$ be an $n \times n$ variable matrix. One can also consider X as a vector of size $n \times n$. The meaning of each variable x_{ij} is:

$$x_{ij} = \begin{cases} 1 & \text{if the position } (i,j) \text{ is chosen} \\ 0 & \text{if the position } (i,j) \text{ is not chosen} \end{cases}$$

$$\begin{array}{lll} \text{(IP-A)} & \max w(X) := \displaystyle\sum_{i,j} w_{ij}x_{ij} & \\ & \text{subject to} \quad \displaystyle\sum_{j=1}^{n} x_{ij} = 1 & \forall i = 1, \ldots, n \\ & \qquad\qquad \displaystyle\sum_{i=1}^{n} x_{ij} = 1 & \forall j = 1, \ldots, n \\ & \qquad\qquad x_{ij} = 0 \text{ or } 1, & \forall i, j = 1, \ldots, n. \end{array}$$

This is an integer programming (IP) formulation of the assignment problem. A feasible solution to (IP-A) can be identified with an assignment, since any feasible solution X has exactly one in each row and each column, and such an X is feasible for (IP-A).

In general, an IP is hard to solve. However some IP problems have certain favorable structures which help us to find an optimal solution efficiently. The above IP is fortunately of this kind. In fact the IP problem above is essentially equivalent to an LP where the integer restriction $x_{ij} = 0$ or 1 is replaced by $0 \le x_{ij} \le 1$. More precisely,

- LP Relaxation of the Assignment Problem

$$
\begin{array}{lll}
\text{(LP-A)} & \max w(X) := \sum_{i,j} w_{ij} x_{ij} & \\
& \text{subject to} & \\
& \qquad \sum_{j=1}^{n} x_{ij} = 1, & \forall i = 1, \ldots, n \\
& \qquad \sum_{i=1}^{n} x_{ij} = 1, & \forall j = 1, \ldots, n \\
& \qquad x_{ij} \ge 0, & \forall i, j = 1, \ldots, n.
\end{array}
$$

Since (LP-A) has a less restricted constraint set, its optimal value is at least as large as that of (IP-A). An important fact is

Theorem 6.3 *There is an integral optimal solution to the linear programming problem* (LP-A). *Consequently, the optimal values of* (LP-A) *and* (IP-A) *are equal.*

Exercise: Prove Theorem 6.3 by an elementary argument. More precisely, for any given fractional solution X to (LP-A), find an algorithm to find an integral solution $\tilde{X}$ which is as good as X, i.e. satisfies $w(\tilde{X}) \ge w(X)$. Is it polynomial?

These together imply

Remark 6.4 *The assignment problem can be solved by an LP algorithm applied to* (LP-A) *and the "integerizer". Furthermore any pivot methods return an integral solution in this case (see Theorem 6.6).*

- Certificate for Optimality

There is a direct algorithm for the assignment problem, known as the Hungarian method, which is a polynomial algorithm and practically more efficient. While the simplex method works reasonably well for the assignment problem, if one needs the most efficient algorithm, the Hungarian method should be chosen. The Hungarian method exploits a certificate for optimality that coincides with the certificate (the LP strong duality theorem, Theorem 2.2) applied to the (LP-A).

Theorem 6.5 *An assignment $X^* = [x^*_{ij}]$ is optimal if and only if there exist two vectors $u \in R^n$ and $v \in R^n$ such that*

$$u_i + v_j \ge w_{ij} \qquad \forall i, j = 1, 2, \ldots, n \tag{6.2}$$

$$w(X^*) = \sum_i u_i + \sum_j v_j. \tag{6.3}$$

As for the optimality certificate for linear programming, one direction of the equivalence above is easy and the other is nontrivial. Namely, if there exist two vectors $u \in R^n$ and $v \in R^n$ satisfying the conditions above, then for any assignment $X = [x_{ij}]$

$$\begin{aligned} w(X) = \sum_i \sum_j w_{ij} x_{ij} &\le \sum_i \sum_j (u_i + v_j) x_{ij} \\ &= \sum_i \left(u_i \sum_j x_{ij} \right) + \sum_j \left(v_j \sum_i x_{ij} \right) \\ &= \sum_i u_i + \sum_j v_j \\ &= w(X^*). \end{aligned} \tag{6.4}$$

This means X^* is an optimal assignment.

- Theorem 6.3 is "equivalent to" a more geometric theorem (Here we mean by "equivalent" easily reducible from each other):

  Theorem 6.6 *Every extreme point of the feasible region*

  $$\Omega = \{X \in R^{n \times n} : X \text{ satisfies the constraints of (LP-A)}\}$$

  of (LP-A) *is integral. (Such a convex polytope is called integral.)*

  Recall that a point $X \in \Omega$ is called a extreme point of Ω if they are not on the line segment connecting two other points X^1 and X^2 of Ω. As it was shown in Section 4.9.3, any pivot algorithms such as the simplex method (see, Section 4.4.3) always finds an extreme optimal solution.

- Theorem 6.6 can be considered as a special case of a more general theorem:

  Theorem 6.7 *For any totally unimodular matrix $A \in Z^{m \times d}$ and any integral vector $b \in Z^m$, the feasible region*

  $$\Omega(A, b) = \{x \in R^d : Ax = b \text{ and } x \ge \mathbf{0}\}$$

  is integral.

  Proof. (Outline) This follows from the fact that any extreme solution is elementary (see Section 4.9.3). This implies it is a unique solution to a subsystem $A_{\cdot B} x_B = b, x_N = \mathbf{0}$ for some partition (B, N) of the column index set. The total unimodularity and Cramer's rule imply the result. ∎

  Here, a matrix is called *totally unimodular* if all its subdeterminants are either 0, -1 or 1. (A *subdeterminant* is the determinant of a square submatrix of A).

  To see that Theorem 6.6 is a special case, write the matrix A for (LP-A) by considering the matrix X as a long vector $x = (x_{11}, x_{12}, \ldots, x_{1n}, x_{21}, \ldots, \ldots, x_{nn})$.

6.4 Optimal Matching Problem

- Optimal Matching Problem

  Given a graph $G = (V, E)$ with edge weight w_{ij} for each $(i, j) \in E$, find a matching M with maximum total weight.

- Optimal Perfect Matching Problem?

  The optimal perfect matching problem can be defined similarly. Is it more difficult than the optimal matching problem? In fact, they are essentially the equivalent. How can we solve the optimal perfect matching problem by using an algorithm for the optimal matching problem? (Exercise) Because of the equivalence, we discuss the optimal matching problem only.

- IP Formulation of the Optimal Matching Problem

  Let x_{ij} denote a variable for each edge $(i, j) \in E$ to represent a matching. The meaning of each variable x_{ij} is:

$$x_{ij} = \begin{cases} 1 & \text{if } (i,j) \text{ is a matching edge} \\ 0 & \text{otherwise} \end{cases}$$

$$\begin{array}{lll} \text{(IP-M)} & \max w(x) := \displaystyle\sum_{(i,j)\in E} w_{ij}x_{ij} & \\ & \text{subject to} \displaystyle\sum_{(i,j)\in E} x_{ij} \le 1 & \forall i \in V \\ & x_{ij} = 0 \text{ or } 1, & \forall (i,j) \in E. \end{array}$$

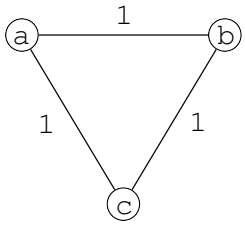
  This is an integer programming (IP) formulation of the assignment problem. A feasible solution to (IP-M) can be identified with a matching.

$$\begin{array}{lll} \text{(LP-M)} & \max w(x) := \displaystyle\sum_{(i,j)\in E} w_{ij}x_{ij} & \\ & \text{subject to} \displaystyle\sum_{(i,j)\in E} x_{ij} \le 1 & \forall i \in V \\ & x_{ij} \ge 0, & \forall (i,j) \in E. \end{array}$$

  This LP relaxation is not as good as the LP relaxation (LP-A) for the assignment problem given in Section 6.3. Why? This is because a statement corresponding to Theorem 6.3 is not true.

- Small Example

  Consider the following graph. It is a triangle of equal edge weight 1. Clearly we cannot select two edges for a matching, and so any single edge forms an optimal matching of weight 1.



On the other hand, the relaxation (LP-A) has a better solution. Namely, the vector $(x_{ab}, x_{bc}, x_{ac}) = (1/2, 1/2, 1/2)$ is feasible since the constraints say:

$$x_{ab} + x_{ac} \leq 1 \tag{6.5}$$
$$x_{ab} + x_{bc} \leq 1 \tag{6.6}$$
$$x_{ac} + x_{bc} \leq 1 \tag{6.7}$$
$$x_{ac}, x_{bc}, x_{ac} \geq 0. \tag{6.8}$$

This fractional solution has a better weight, 1.5, and can be shown to be optimal for (LP-M). (Why?)

- How to cut off the fractional solutions?

  The small example shows that the LP relaxation (LP-M) is not very useful (yet) to solve the optimal matching problem. Edmonds showed that we only need to add one extra class of constraints. The idea can be seen in the example. Since there are three nodes, the sum of all variables for the edges cannot be more than 1, if x represents a matching.

  In general, if we pick up any subset S of nodes with odd cardinality $2k + 1$ for some k, any vector x representing a matching must satisfy the inequality:

$$\sum_{i,j \in S, (i,j) \in E} x_{ij} \leq k. \tag{6.9}$$

This observation leads to an appropriate LP relaxation of the optimal matching problem.

(LP-M2)

$$\begin{aligned}
&\max w(x) := \sum_{(i,j)\in E} w_{ij} x_{ij} \\
&\text{subject to} \quad \sum_{(i,j)\in E} x_{ij} \leq 1 \qquad \forall i \in V \\
&\text{subject to} \quad \sum_{i,j\in S,(i,j)\in E} x_{ij} \leq \frac{|S|-1}{2} \quad \forall \text{ odd set } S \subseteq V \\
&\qquad\qquad x_{ij} \geq 0, \qquad \forall (i,j) \in E.
\end{aligned}$$

Theorem 6.8 *The optimal values of* (IP-M) *and* (LP-M2) *are equal. That is, there is an integral optimal solution to the linear programming problem* (LP-M2).

- Certificate for Optimality

  Theorem 6.9 *A matching $x^* = (x^*_{ij})$ is optimal if and only if there exists a feasible solution to the dual of (LP-M2) whose objective value is equal to $w(x^*)$.*

  Edmonds found an ingenious algorithm, known as Blossom Algorithm, for the optimal matching problem which finds an optimal matching with a dual feasible solution in polynomial time. Unlike the assignment problem, one should not apply the simplex method to the LP relaxation (LP-M2), because there are just too many (exponential in n) odd set inequalities and therefore too many dual variables. Edmonds' algorithm considers only a small set of such inequalities at a time, and update the set whenever necessary. Fortunately it is possible to show that there is a dual optimal solution with a polynomial number of nonzero components.

- Using an LP relaxation is a very powerful technique, even when a system of linear inequalities to determine the integral polytope is not known. Here is a general approach to the combinatorial optimization:

  (a) Formulate a combinatorial optimization problem as an IP whose feasible solutions represent the feasible solutions to the original problem;

  (b) Relax the IP as an LP;

  (c) Solve the LP by the simplex method or any LP algorithm that returns an extreme optimal solution;

  (d) If the solution to the LP is integral, it solves the original problem;

  (e) Otherwise, try to add constraints that cut off this fractional extreme point but not any integral feasible points. Repeat from (c).

  This technique is known as the cutting plane technique for combinatorial optimization, and has been used together with the branch-and-bound technique to solve a large-scale hard optimization problems such as the TSP. We shall discuss this in Chapter 7.

6.5 Maximal Flow Problem

- Network

  – a graph where the nodes and/or the edges are associated with some given numbers that (usually) have some physical or economic meaning.

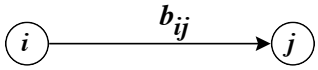
- Maximum Flow Problem

  Graph

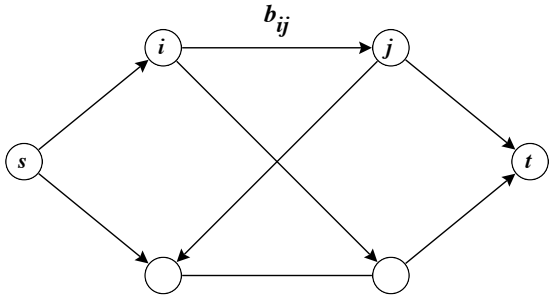
$G = (V, E)$	a directed graph
$V = \{1, 2, \cdots, n\}$	the node set
$E =$ the set of pairs (i, j)	the edge set

  Edge capacity

b_{ij}	the capacity of flow per unit time for the edge (i, j)



Problem Description
Determine a flow from the source s to the sink t maximizing the amount leaving the source node s. Here we assume that at all nodes other than s and t the conservation law is satisfied: the entering total flow = the leaving total flow.



Variables

x_{ij}	a flow on the edge (i, j)
v	the value of a flow x

$$\begin{aligned} &\max && v \\ &\text{subject to} && \sum_{(j,i)\in E} x_{ji} - \sum_{(i,j)\in E} x_{ij} = \begin{cases} -v & i = s \\ 0 & \forall i \neq s, t \\ v & i = t \end{cases} \\ & && 0 \leq x_{ij} \leq b_{ij} \quad \forall (i,j) \in E \end{aligned}$$

- If all b_{ij} are integers, there exists an integral optimal solution. Prove this by an elementary argument similar to that for the assignment problem (Theorem 6.3). In fact, all extreme solutions are integral, and thus the simplex method finds an integral optimal solution.

- There are simple direct methods for the maximum flow problem that are more efficient than the simplex method.

- Certificate for Optimality

  A partition $(S, \overline{S} = V \backslash S)$ of the node set V is called an (s,t)-*cut* if $s \in S$ and $t \in \overline{S}$. The value $w(S, \overline{S})$ of an (s,t)-cut $(S, \overline{S})$ is defined as the sum of capacities of the edges directed from S to its complement.

  (6.10) $$w(S, \overline{S}) = \sum_{\substack{i \in S, j \in \overline{S} \\ (i,j) \in E}} b_{ij}.$$

  It is easy to show that for any feasible flow of value v and any (s,t)-cut of value w, $v \leq w$. The following theorem shows the equality for max-flow and min-cut pairs.

Theorem 6.10 *A feasible flow $x^* = (x^*_{ij})$ with value v^* is optimal if and only if there exists an (s,t)-cut with the same value.*

6.6 Minimum Cost Flow Problem

Given a graph with edge weight (e.g. cost) and demands for each nodes, find a flow satisfying all demands that minimizes the total cost.

Node Demand

s_i demand of node i

(If $s_i < 0$, we consider that i is a supply node and its supply quantity is $-s_i$.)

Edge Cost

c_{ij} cost of running a unit flow on edge (i,j)

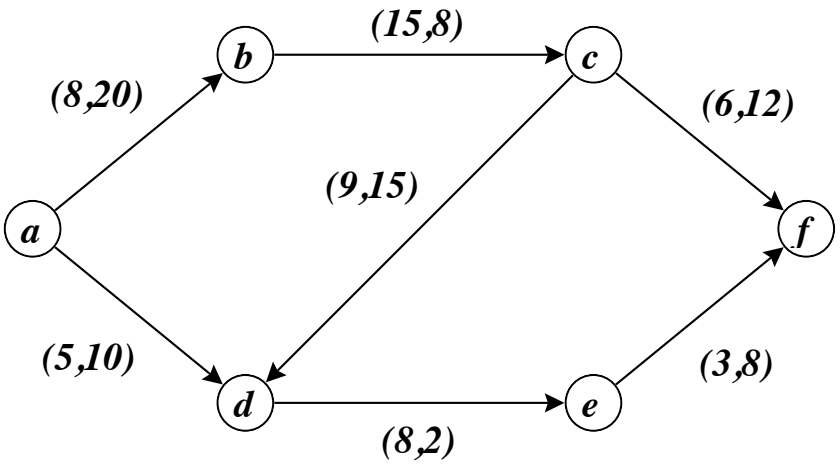
$$\begin{aligned} \max \quad & \sum_{(i,j)\in E} c_{ij}x_{ij} \\ \text{subject to} \quad & \sum_{(j,i)\in E} x_{ji} - \sum_{(i,j)\in E} x_{ij} = s_i \qquad \forall i \in V \\ & 0 \le x_{ij} \le b_{ij} \qquad \forall (i,j) \in E \end{aligned}$$

- This problem is more general than the maximum flow problem. (Why?)
- When b_{ij} and s_i are integers and the problem has an optimal solution then there exists an integral optimal solution. Prove this by an elementary argument similar to that for the assignment problem (Theorem 6.3). In fact all extreme solutions are integral, and thus any pivot methods for LP find an integral optimal solution.
- Certificate of Optimality

  The minimum cost flow problem is an LP with a special structure. Thus the certificate of optimality using the dual optimal solution (Theorem 2.2) applies here.
- There are direct polynomial methods for the minimum cost flow problem that are more efficient than the simplex method. There are even *strongly polynomial* algorithms for this problem as well, for which the number of arithmetic operations to solve any instance is bounded above by a polynomial in the number of nodes and edges only (i.e. independent of the sizes of b_{ij}'s, c_{ij}'s and s_i's. Such methods are not yet known to be practical, but the existence is in great contrast to the general LP case for which no strong polynomial algorithms are known.

6.6.1 Exercise (Toy Network Flow Problems)

Six cities a, b, c, d, e, f are linked by pipelines. Capacities $(kl/second)$ and costs $(francs/kl)$ are given below.



Here a label (p, q) indicates cost p and capacity q.

Modelling by the Max Flow Problem

Using the network of pipelines, how can one maximize the flow from the city a to the city f? We assume flows satisfy the conservation law at each cities other than a and f.

Modelling by the Minimum Cost Flow

Assuming that the demands of each cities are given below, how can one find a minimum cost flow satisfying the demands.

city	a	b	c	d	e	f
demand	-20	-4	0	6	8	10

Chapter 7

Exhaustive Search and Branch-and-Bound Algorithms

7.1 Branch-and-Bound Technique

NP-complete optimization problems, such as TSP, the knapsack problem and the three-dimensional assignment problem, are all alike in the sense that there is no known succinct certificate for optimality. This implies at least in practice (quite likely in theory as well) that in order to verify the optimality of a feasible solution x^*, we must depend on a proof that has no polynomial bound in the input size. Typically such a proof is the direct verification of optimality by enumeration of all feasible solutions.

The Branch-and-Bound (B&B) algorithm is a general scheme of algorithms to solve hard (mainly NPC or harder) optimization problems that avoids unnecessary enumeration of certain feasible solutions using previously computed solutions and simple logics. The best way to illustrate the idea is to look at a simple example of the knapsack problem.

- Knapsack Problem
  From n items with given positive values $c_1, c_2, \ldots, c_n$ and positive weights $w_1, w_2, \ldots, w_n$, select items with total weight at most b (a given constant) so as to maximize the total value.

- Example

j	1	2	3	4	5	
c_j	10	80	40	30	22	
w_j	1	9	5	4	3	$(\leq b = 13)$

- IP Formulation

$$
\begin{array}{lll}
(P) & \max f(x) := & 10x_1+80x_2+40x_3+30x_4+22x_5 \\
& \text{subject to} & \\
& E: & x_1+\ 9x_2+\ 5x_3+\ 4x_4+\ 3x_5 \ \leq \ 13 \\
& E_I: & x_1, x_2, x_3, x_4, x_5 \ = \ 0 \text{ or } 1
\end{array}
$$

- Two Basic Conditions for the B&B

  To define a B&B algorithm for an optimization problem P, the following two conditions are necessary (and sufficient):

  (a) Problem Decomposability: A given problem P can be easily decomposed into two or more subproblems $P_1,\ldots,\ P_k$ so that the optimal solution of P can be found by solving the subproblems and selecting the best solution. In particular,

  $$\text{ov}(P) = \max\{\text{ov}(P_1), \text{ov}(P_2), \ldots, \text{ov}(P_k)\},$$

  where ov denotes the optimal value.

  (b) Upper and Lower Bounds Easily Computable: It is easy to compute some lower bound LB(P) and upper bound UB(P) of the optimal value ov(P). (This should apply to all subproblems as well.)

  Knapsack case

  (a) One can easily decompose the problem into two subproblems $P_1 = P|_{x_3=0}$ and $P_2 = P|_{x_3=1}$ by fixing one variable, say x_3, to 0 and 1 respectively.

  (b) For lower bound computation, for example, we can use the trivial bound 0. A better bound can be found by a **heuristic search**. For example, add items one by one until one cannot add any more, and good heuristics on a priority order of items can help to improve this bound.

  For upper bound computation, one can use the LP relaxation. The optimal value of the LP relaxation is an upper bound of ov(P).

- LP Relaxation

$$\begin{array}{lllllllll}
(LP-P) & \max f(x) := & 10x_1 & +80x_2 & +40x_3 & +30x_4 & +22x_5 & & \\
& \text{subject to} & & & & & & & \\
& E: & x_1 & +9x_2 & +5x_3 & +4x_4 & +3x_5 & \leq & 13 \\
& E_1: & x_1 & & & & & \leq & 1 \\
& E_2: & & x_2 & & & & \leq & 1 \\
& E_3: & & & x_3 & & & \leq & 1 \\
& E_4: & & & & x_4 & & \leq & 1 \\
& E_5: & & & & & x_5 & \leq & 1 \\
& E_0: & & & x_1, x_2, x_3, x_4, x_5 & & & \geq & 0
\end{array}$$

- If the LP has an integral optimal solution, then it solves the original problem. This is not the case in general.

  The variables are already sorted by the efficiency $\frac{c_j}{w_j}$:

$$\frac{10}{1} > \frac{80}{9} > \frac{40}{5} > \frac{30}{4} > \frac{22}{3}$$

By filling the knapsack with items $1, 2, \ldots$ consecutively as much as one can, we obtain an integral feasible solution

$$x^I(P) = (1, 1, 0, 0, 0), \qquad \text{value} = 10 + 80 = 90 \tag{7.1}$$

Moreover, by filling the best unused item 3 with the maximum fraction, we obtain a solution

$$x^L(P) = (1, 1, 3/5, 0, 0), \qquad \text{value} = 90 + 40 \times 3/5 = 114 \tag{7.2}$$

which can be shown to be optimal for the LP relaxation (LP-P).

Proposition 7.1 *The solution $x^L(P)$ obtained by the above algorithm is an optimal solution to* (LP-P).

For the rest of discussion, we fix the upper bound and the lower bound functions as:

$$\text{LB}(P) = f(x^I(P)) \tag{7.3}$$

$$\text{UB}(P) = f(x^L(P)) \tag{7.4}$$

- B&B Tree

  The B&B algorithm starts with the original problem P as the **root** node of its dynamically constructed tree T. If $\text{LB}(P) = \text{UB}(P)$, we have solved the problem. Otherwise (the case for our example) we set the *currently best value* CurrBestVal := $\text{LB}(P) = 90$. Then decompose P into two problems. The natural choice is by using x_3, since x_3^L is not integer. See Fig. 7.1.

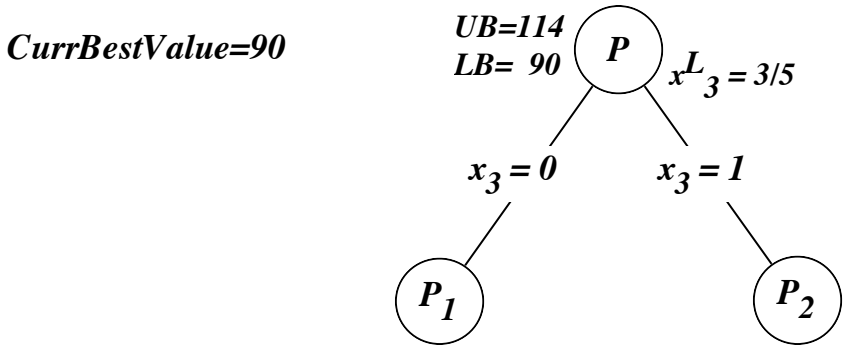


Figure 7.1: Initial Branch-and-Bound Tree

Then we compute the bounds for both $P_1 = P|_{x_3=0}$ and $P_2 = P|_{x_3=1}$. Both problems have slightly smaller upper bounds but they do not generate any better integral solution. We must decompose again one of P_1 or P_2. Take the one with better upper bound, hoping that we obtain a better solution.

In particular, for P_{11}, we obtain an integral solution

$$x^L(P) = (1, 1, 0, 0, 1), \qquad \text{value} = 112 \tag{7.5}$$

Here we can update the currently best value with 112.

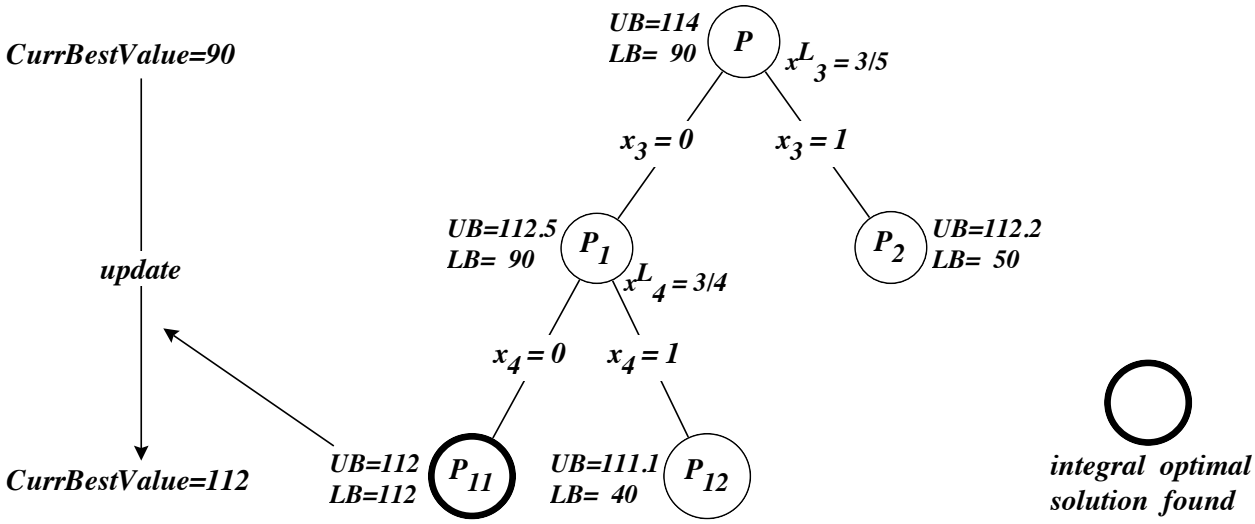


Figure 7.2: Intermediate Branch-and-Bound Tree

j	1	2	3	4	5
c_j	10	80	40	30	22
w_j	1	9	5	4	3

P_1	x^I	1	1	0	0	0	LB $= 10 + 80 = 90$
	x^L	1	1	0	3/4	0	UB $= 90 + 30 \times (3/4) = 112.5$
P_2	x^I	1	0	1	0	0	LB $= 10 + 40 = 50$
	x^L	1	7/9	1	0	0	UB $= 50 + 80 \times (7/9) = 112.2$
P_{11}	x^I	1	1	0	0	1	LB $= 10 + 80 + 22 = 112$
	x^L	1	1	0	0	1	UB $= 112$
P_{12}	x^I	1	0	0	1	0	LB $= 10 + 30 = 40$
	x^L	1	8/9	0	1	0	UB $= 40 + 80 \times (8/9) = 111.1$

Table 7.1: Computation of lower and upper bounds

By evaluating the bounds for P_{12}, we obtain an upper bound 111.1 which is smaller than CurrBestVal. This means there is no hope to find a better solution to P, and we can terminate the investigation of this node.

Now we are left with the node P_2 whose upper bound is 112.2 which is still larger than CurrBestVal= 112. Thus we decompose[1] it by using x_2 which takes a fractional value at the LP solution. This creates two nodes P_{21} and P_{22}. Calculation shows that the upper bounds are 61 and $-\infty$, respectively. $-\infty$ means that the LP is infeasible. Since both bounds are smaller than CurrBestVal, there is no hope to find a solution whose value is better (larger) than CurrBestVal.

[1]In our particular example in which all c_j's are integer, we do not need to decompose the node P_2.

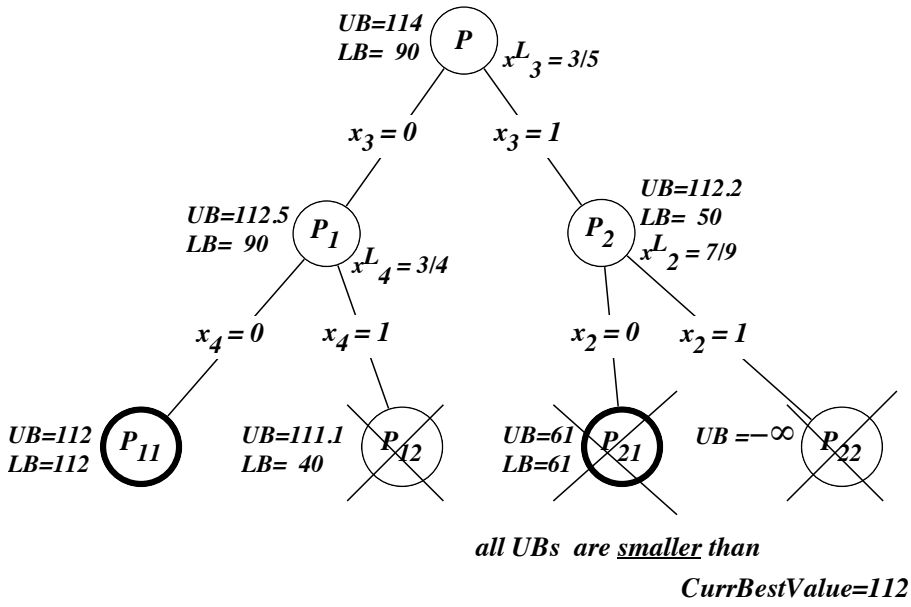


Figure 7.3: Final Branch-and-Bound Tree

- General Description

```
procedure Branch&Bound(P);
begin
    ProblemQueue := {P};
    CurrBestVal := LB(P);
    while ProblemQueue ≠ ∅ do
        select and remove one problem P' from ProblemQueue;
        if (UB(P') > CurrBestVal) then
            if (LB(P') > CurrBestVal ) then
                CurrBestVal := LB(P');
            endif;
            if (UB(P') > LB(P')) then
                Decompose P' into P_1, P_2, ..., P_k;
                Compute bounds for P_1, P_2, ..., P_k;
                Add P_1, P_2, ..., P_k to ProblemQueue;
            endif;
        endif;
    endwhile;
    output CurrBestVal; /* CurrBestVal is the optimal value. */
end.
```

- Certificate of Optimality
  The certificate of optimality generated by a B&B algorithm is the final B&B tree. In this tree, one of the leaf nodes (i.e. degree 1 nodes) contains a feasible solution that

realizes the optimal value (= final CurrBestVal). And all other leaf nodes have upper bounds smaller or equal to this value.

This certificate might be very large since the complete enumeration tree with all leaf nodes decomposed into trivial subproblems (with all variables fixed) is exponential in size. For the knapsack case, this is of order 2^n.

- Strategies for Reducing the B&B tree size

There is much flexibility in B&B algorithms that one can exploit for reducing the size of the final B&B tree. One should experiment with different strategies to see what is best for any specific problem and problem instances.

Decomposition For the knapsack problem, there is always a unique fractional component in the LP solution for natural decomposition. In general, there are many different natural variables for decomposition. Also one might be able to decompose the problem by a completely different logic.

Priority Which problem in the ProblemQueue should one investigate? The most important objective is to recognize problem nodes that contain better feasible solutions.

One should use some heuristics when selecting a most hopeful branch. This varies from problem to problem. Selecting a node with the largest upper bound is one natural choice. However one must be very careful to fix any strategy, since intuitively sound rules might fail very badly.

Upper Bounding In general, computing a better upper bound means more time. "When should one use more sophisticated bounds?" is an important question that is hard to answer. Yet, an accurate upper bound becomes critical in the following situation.

Suppose the algorithm already found an (almost) optimal value at some stage, and there are still many problems in the queue. The remaining task is to eliminate as quickly as possible those "hopeless" problems in the queue by the logic $\mathrm{UB}(P') \leq$ CurrBestVal.

In this situation, one might be able to speed up by a better (but more time consuming) upper bounding algorithm. This is typically done by the cutting plane technique described at the end of Section 6.4. In particular, the cutting plane technique has been heavily used to solve large-scale TSPs, and it appears to be essential for successful termination of the B&B algorithm. The technique combining B&B and the cutting plane technique is called the *Branch-and-Cut* technique.

Lower Bounding This is where *heuristics* can help enourmously. Finding a reasonable solution in the earlier stage usually reduces the size of the B&B tree drastically. The typical techniques are local search algorithms with different strategies, such as, simulated annealing, tabu search and genetic algorithms.

Chapter 8

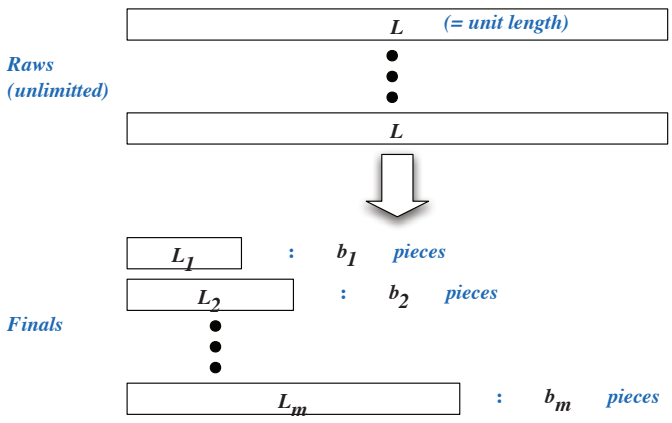
Cutting-Stock Problem and Column Generation

In this chapter, we present a technique to solve a certain type of large scale linear programming problems, known as the column generation technique. We shall present this general technique through its application to the Cutting-Stock Problem, an NP-hard combinatorial optimization problem.

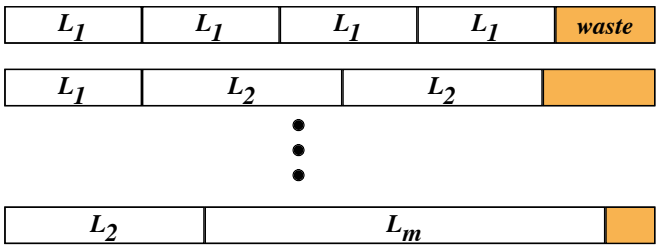
8.1 Cutting-Stock Problem

The cutting-stock problem arises often in industries that need to cut basic raws (e.g. wood or metal sheets, paper rolls, textiles) of unit length into various pieces (called finals) of demanded lengths. The main goal is to find the most economical way of cutting-out the finals requiring the smallest number of raws.

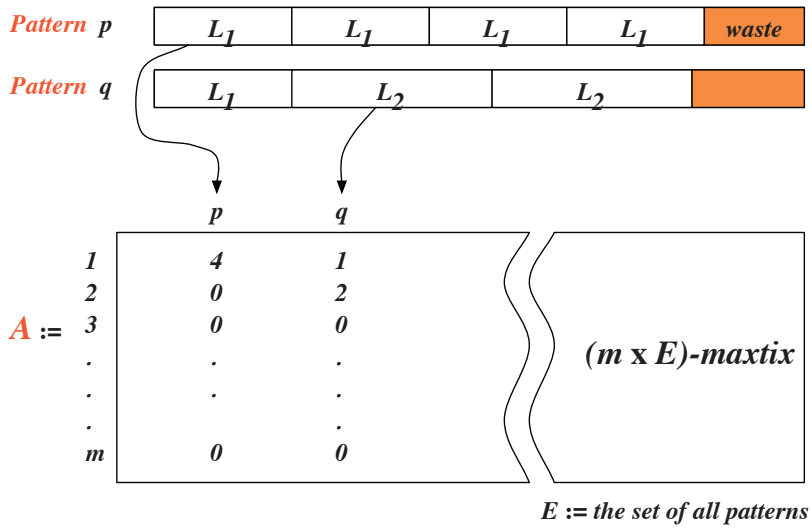
More formally, the *cutting-stock problem* is the following: Given raws of unit length L, and given demands of b_i finals of lengths L_i for $i = 1, 2, \ldots, m$, find a most economical way to satisfy the demands.



A key obstacle in finding an optimal solution for the cutting-stock problem is that in general there are exponentially many patterns to cut a raw.



Let us first formulate the optimization problem as an integer programming problem. A *cutting pattern* is given by specifying how many of pieces of length L_i for each $i = 1, 2, \ldots, m$ is cut out (so that the total length does not exceed L). Let E be the set of all cut patterns and let A be the $(m \times E)$ pattern matrix where a_{ij} represents the number of pieces of length L_i in pattern $j \in E$.



An IP formulation of the cutting-stock problem is thus to find an optimal combination of frequency x_j of using pattern j so that the demand is met.

$$\begin{array}{lrcl} \text{minimize} & e^T x & & (e^T = (1, 1, \ldots 1)) \\ \text{subject to} & & & \\ & A\,x & = & b \\ & x & \geq & 0 \\ & x_j & \text{is} & \text{integer} \quad (\forall j \in E). \end{array}$$

The *LP relaxation* is the LP obtained from this by removing the integrality condition on x_j's. It is important to note that the LP relaxation itself is already challenging as the number of columns of A can be very large and one cannot expect the matrix A to be explicitly given.

The column generation technique to be described below is to solve the LP relaxation, of course, it may generate a fractional solution. How to get around with this problem depends very much on specific applications.

8.2 The Simplex Method Review

- Linear Programming:

$$\begin{array}{lrcl} \text{minimize} & f = c^T\, x & & \\ \text{subject to} & & & \\ & A\, x & = & b \\ & x & \geq & 0. \end{array}$$

- Basis ($A_{\cdot B}$: nonsingular, $B \subseteq E$):

$$A_{\cdot B} x_B + A_{\cdot N} x_N = b$$

$$A = \left[\begin{array}{c|c} A_{\cdot B} & A_{\cdot N} \end{array}\right]$$

$$\Longrightarrow \begin{array}{rcl} x_B & = & A_{\cdot B}^{-1} b - A_{\cdot B}^{-1} A_{\cdot N} x_N \\ f & = & c_B^T A_{\cdot B}^{-1} b + (c_N^T - c_B^T A_{\cdot B}^{-1} A_{\cdot N}) x_N \end{array}$$

- Basic solution ($\overline{x}$):

$$\overline{x}_N = 0, \qquad \overline{x}_B = A_{\cdot B}^{-1} b$$
$$\overline{f} = c_B^T A_{\cdot B}^{-1} b$$

Simplex Method

- Start with a feasible basic solution $\overline{x}$:

$$\overline{x}_N = 0, \quad \overline{x}_B = A_{\cdot B}^{-1} b \geq 0.$$

- The current solution is optimal if

$$(c_N^T - c_B^T A_{\cdot B}^{-1} A_{\cdot N}) \geq 0$$

  or equivalently

$$(c_j - c_B^T A_{\cdot B}^{-1} A_{\cdot j}) \geq 0 \text{ for all } j \in N.$$

- If the optimality is violated for some $s \in N$:

$$(c_s - c_B^T A_{\cdot B}^{-1} A_{\cdot s}) < 0,$$

  then the column s enters the basis and some basic $r \in B$ leaves to produce a new feasible basis (=pivot operation). If no proper r exists, then the LP is unbounded.

8.3 Column Generation Technique

Assume A (and c) is very wide and implicitly given.

- If the optimality:

  $$(*) \qquad (c_j - c_B^T A_{.B}^{-1} A_{.j}) \geq 0 \text{ for all } j \in N.$$

  can be checked quickly without direct reference to c_N, $A_{.N}$, each simplex iteration is simple.

- In practice, the number of simplex pivots depends mainly on m, roughly $O(m)$. Thus, the resulting algorithm is expected to perform efficiently.

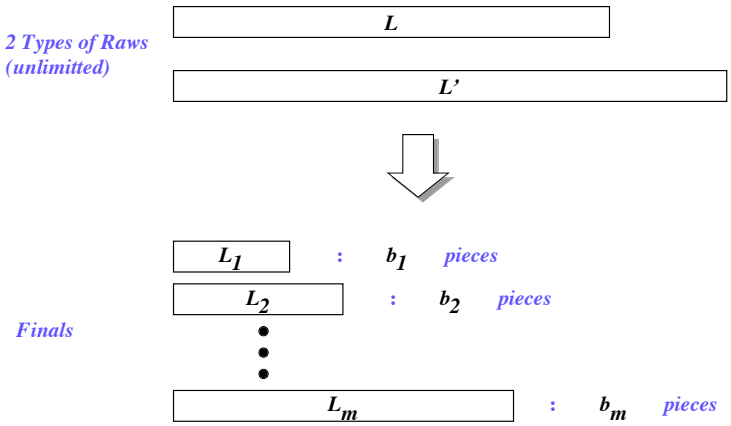
Example: Cutting-Stock Case

- $c^T = e^T \equiv (1, 1, \ldots 1)$. By setting $p = c_B^T A_{.B}^{-1}$,

$$\begin{aligned}
(*) &\iff (1 - pA_{.j}) \geq 0 \text{ for all } j \in N \\
&\iff \max_{j \in N} pA_{.j} \leq 1 \\
&\iff z \leq 1 \text{ where} \\
&\qquad z := \max \sum_{i=1}^{m} p_i \alpha_i \\
&\qquad \text{subject to } \sum_{i=1}^{m} L_i \alpha_i \leq L \\
&\qquad \alpha_i \text{ is nonneg. integer } \forall i = 1, \ldots, m. \\
&\qquad (= \text{Knapsack Problem})
\end{aligned}$$

Exercise

Extend the column generation algorithm to the cutting stock problem with two different types of raws, L and L'. How can one generalize the algorithm to minimize the total waste? What happens if there are k different raws?



- Generalized Assignment Problem: Find the best assignment of m tasks to n machines subject to capacity constraints.
- Crew Scheduling for Airlines: Select the best flight pairings to cover a given flight schedule.

A Little History

- The basic idea of column generation was first used by Dantzig and Wolfe (1960) in a different setting: solving large-scale structured LP's by decomposing the constraints.
- The column generation technique discussed in this lecture was due to Gilmore and Gomory (1961).
- Column generation is sometimes called *delayed column generation.*

Chapter 9

Approximation Algorithms

In this chapter, we present some basic techniques to solve combinatorial optimization problems approximately. Here the term "approximation algorithm" is used for algorithm that has certain performance guarantee.

Formally, for any positive number p, when we say a method is a *factor p approximation algorithm* for a class of minimization (maxization) problems, it means that the algorithm generate a feasible solution with objective value at most (at least) p times the optimal value. Note that this definition makes sense only if the optimal value is positive, but it is usually quite easy to convert problems so that this condition is satisfied.

There are many different techniques to design approximation algorithms. We shall introduce three basic approaches, namely, the greedy, the primal-dual and the LP-rounding techniques. We shall present the techniques through the set cover problem, although they can be applied to many different types of combinatorial optimization problems.

9.1 Set Cover Problem

The set cover problem is a NP-hard optimization problem. Let U be a finite set of cardinality n and let $\mathcal{S}$ be a set of subsets of U. A *set cover problem* is to find a minimum-cardinality subset $\mathcal{C}$ of $\mathcal{S}$ that *covers* the ground set U, i.e. the union of all sets in $\mathcal{C}$ is U.

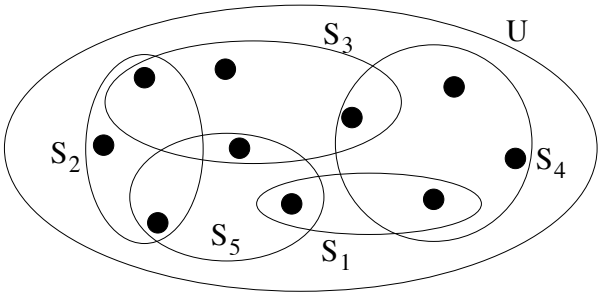


Figure 9.1: Set Cover Problem

Clearly, such a cover exists if and only if the union of all sets in $\mathcal{S}$ is U, and we assume

this for the sequel of the chapter.

It is not hard to imagine applications of the set cover problem. For example, when a company needs to set up client service centers to cover all clients, a problem of this kind arises. In particular, in the case that there are finite candidate locations for centers and the set of clients that can be serviced by a given location is known, the problem of selecting a smallest number of locations to service all clients is exactly a set cover problem.

Naturally, one might want to extend the set cover problem to the weighted set cover problem where each set $S \in \mathcal{S}$ is associated with positive cost $w(S)$ and the objective is to find a cover with minimum total cost. For simplicity of discussion, we shall not deal with the general problem, but many techniques, such as the greedy algorithm in the next section, can be generalized naturally.

Let us first formulate the set cover problem as an IP. By setting a binary variable x_S for each $S \in \mathcal{S}$, the problem is equivalent to the 0/1 IP:

$$\begin{array}{lll} \text{minimize} & \sum_{S\in\mathcal{S}} x_S & \\ \text{subject to} & & \\ & \sum_{e\in S\in\mathcal{S}} x_S \;\geq\; 1, & \forall e \in U \\ & x_S \;\in\; \{0,1\}, & \forall S \in \mathcal{S}. \end{array} \tag{9.1}$$

The LP relaxation is obtained from this by replacing the integrality condition with nonnegativity.

$$\begin{array}{lll} \text{minimize} & \sum_{S\in\mathcal{S}} x_S & \\ \text{subject to} & & \\ & \sum_{e\in S\in\mathcal{S}} x_S \;\geq\; 1, & \forall e \in U \\ & x_S \;\geq\; 0, & \forall S \in \mathcal{S}. \end{array} \tag{9.2}$$

It is easy to see that every optimal solution x to the LP relaxation satisfies $x \leq 1$.

There is a specialization of the set cover problem to graphs, known as the vertex cover problem. For a given graph $G = (E, V)$, a *vertex cover problem* is to find a smallest number of vertices to cover all edges, equivalently, it is the set cover problem where $U := E$ and

$$\mathcal{S} := \Big\{ \underbrace{\{e \in E : i \in e\}}_{S_i} : i \in V \Big\}.$$

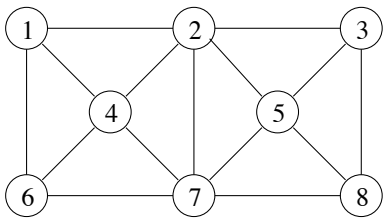


Figure 9.2: Vertex Cover Problem

The integer programming formulation is

$$\begin{array}{lrcl} \text{minimize} & \sum_{j\in V} x_j & & \\ \text{subject to} & & & \\ & x_i + x_j & \geq & 1, \quad \forall \{i,j\} \in E \\ & x_j & \in & \{0,1\}, \quad \forall j \in V. \end{array} \tag{9.3}$$

One can classify the set cover problem in a way that the vertex cover is the easiest among its nontrivial specializations. he *frequency* of an element $e \in U$ in $\mathcal{S}$ is the number of subsets in $\mathcal{S}$ containing e. Denote by $f = f(\mathcal{S})$ the frequency of a most frequent element. It is easy to see that $f(\mathcal{S}) = 2$ for the vertex cover, and the set cover becomes trivial if $f(\mathcal{S}) = 1$.

The greedy algorithm in Section 9.2 achieves the approximation factor of $1 + \ln n$, while both the primal-dual algorithm in Section 9.3 and the LP-rounding Section 9.4 attains the factor of $f = f(\mathcal{S})$. It is worthwhile to remark that none of these is superior to the other.

9.2 Greedy Algorithm

Here we consider a greedy algorithm for the set cover. The key idea is quite simple. The algorithm selects a most efficient set at each step, and stops as soon as all elements of U is covered. The natural definition of *efficiency* $e(S, C)$ of a set S when a subset C of U is already covered is the cardinality of uncovered part in S:

$$e(S, C) := |S \backslash C|. \tag{9.4}$$

The algorithm can be written as follows.

```
procedure GreedySetCover(U, S);
begin
    C := ∅; C := ∅;
    while C ≠ U do
        take a set S ∈ S with largest efficiency e(S, C);
        C := C ∪ S;
        C := C ∪ {S};
    endwhile;
    output C;
end.
```

The following gives an approximation factor of the greedy algorithm.

Theorem 9.1 *The greedy algorithm* `GreedySetCover` *achieves an approximation factor of* $H_n := 1 + 1/2 + 1/3 + \cdots + 1/n$.

Proof. Let $\overline{\mathcal{C}}$ denote the output of the algorithm, $\mathcal{C}^*$ denote a minimum set cover and let z^* be the optimum cost $|\mathcal{C}^*|$. We list the elements of U in the order in which they are

covered by the algorithm as e_1, e_2,..., e_n. We break ties arbitrarily. Let $p(e_k)$ denote the "shared price" of e_k, that is $1/e(S_k, C_k)$, where S_k is the set chosen to cover e_k and C_k is the part which was previously covered. It follows that

$$|\overline{\mathcal{C}}| = \sum_{k=1}^{n} p(e_k). \tag{9.5}$$

Consider the stage when e_k was first covered by a selected set S_k, for a fixed $k = 1, \ldots, n$. This means $e(S, C_k)$ attains maximum at $S = S_k$. Because $\mathcal{C}^*$ covers $U \backslash C_k$ at cost z^*, there must be a set $S \in \mathcal{S}$ that has efficiency at least $|U \backslash C_k|/z^*$. This implies that $e(S_k, C_k) \geq |U \backslash C_k|/z^*$ and $p(e_k) \leq z^*/|U \backslash C_k|$.

On the other hand, $|C_k| \leq k - 1$, and thus we have

$$p(e_k) \leq \frac{z^*}{|U \backslash C_k|} \leq \frac{z^*}{n - k + 1}. \tag{9.6}$$

It follows from (**??**) and (**??**) that

$$\begin{aligned} |\overline{\mathcal{C}}| &= \sum_{k=1}^{n} p(e_k) \leq \sum_{k=1}^{n} \frac{z^*}{n - k + 1} \\ &= (1 + \frac{1}{2} + \cdots + \frac{1}{n}) z^* = H_n z^*. \end{aligned}$$

This proves the theorem. ∎

Note that the Harmonic number H_n is bounded below and above by two log functions as

$$\ln(n + 1) \leq H_n \leq 1 + \ln n.$$

This means the greedy algorithm is a factor $(1 + \ln n)$ approximation algorithm.

Can one find a polynomial-time algorithm that has a better approximation factor? It turns out that there is no approximation algorithm with a constant factor, unless P=NP. Furthermore, it has been shown that the approximation factor H_n of the greedy algorithm cannot be improved beyond a constant factor.

There is a deep theorem, known as the PCP theorem (where PCP stands for probabilistically checkable proof systems), in the theory of approximation algorithms that implies the hardness of various optimizations, including that of the set cover mentioned above. There are similar results known for the vertex cover, the MAX-3SAT (an optimization version of the 3SAT) and the MAX CLIQUE problem. To present these results is beyond the scope of this lecture notes. See Appendix A for further readings on the subject.

9.3 Primal-Dual Algorithm

Consider a dual pair of linear programs:

$$\begin{array}{lrl} \min & c^T x & \\ \text{subject to} & Ax & \geq b \\ & x & \geq \mathbf{0}, \end{array} \tag{9.7}$$

$$\begin{array}{lll} \max & b^T y & \\ \text{subject to} & A^T y & \leq c \\ & y & \geq \mathbf{0}. \end{array} \tag{9.8}$$

The LP relaxation (9.2) of the set cover problem is a special case of the primal LP (9.5) where c and b are both vectors of all 1's and A is a 0/1 matrix.

When we need to solve the primal LP, the dual LP plays a crucial role of certifying the optimality though the complementarity slackness condition, Theorem 2.3. Note that our primal LP here is minimization and in the form of dual in the theorem.

When we need to find a "good" integer solution to the primal LP, we can still make use of a relaxed complementarity slackness condition as follows.

Theorem 9.2 (Relaxed Complementary Slackness Conditions) *Suppose a primal and dual feasible solutions $\overline{x}$ and $\overline{y}$ satisfy the following conditions (a) and (b) for some positive numbers α and β:*

(a) $\overline{x}_j > 0$ *implies* $c_j/\alpha \leq (A^T\overline{y})_j \leq c_j$, *for all* j,

(b) $\overline{y}_i > 0$ *implies* $b_i \leq (A\overline{x})_i \leq \beta\, b_i$, *for all* i.

Then we have

$$b^T\overline{y} \leq c^T\overline{x} \leq (\alpha\beta)\, b^T\overline{y}. \tag{9.9}$$

Proof. Let $\overline{x}$ and $\overline{y}$ be feasible vectors satisfying both (a) and (b). The first inequality $b^T\overline{y} \leq c^T\overline{x}$ is nothing but the weak duality theorem, Theorem 2.7. The second inequality follows from simple substitutions and manipulations:

$$\begin{aligned} c^T\overline{x} = \sum_j c_j\overline{x}_j &\leq \alpha \sum_j (A^T\overline{y})_j\overline{x}_j && (\because \overline{x}_j \geq 0 \text{ and (a)}) \\ &= \alpha \sum_i (A\overline{x})_i\overline{y}_i && (\text{ rewriting } \overline{y}^T A\overline{x}) \\ &\leq \alpha\,\beta \sum_i b_i\overline{y}_i && (\because \overline{y}_i \geq 0 \text{ and (b)}) \\ &\leq (\alpha\beta)\, b^T\overline{y} \end{aligned}$$

■

The inequalities (9.7) imply that the primal solution $\overline{x}$ attains the approximation factor of $\alpha\beta$. Thus, if one manages to design an algorithm that finds a dual pair $(\overline{x}, \overline{y})$ of feasible solutions satisfying the relaxed complementarity conditions such that $\overline{x}$ is integral, then it is a factor $(\alpha\beta)$ approximation algorithm for the associated integer programming problem.

A Primal-Dual Algorithm for Set Cover

Based on the ideas above, we shall present a concrete algorithm for the set cover problem. Consider the LP relaxation (9.2) of the set cover, and its dual LP:

$$\begin{array}{lrcl} \text{maximize} & \sum_{e\in U} y_e & & \\ \text{subject to} & & & \\ & \sum_{e\in S} y_e & \leq & 1, \quad \forall S \in \mathcal{S} \\ & y_e & \geq & 0, \quad \forall e \in U. \end{array} \tag{9.10}$$

In our application of Theorem 9.2, we set $\alpha = 1$ and $\beta = f$, where $f = f(\mathcal{S})$ is the frequency of the most frequent element. With this setting, the relaxed complementarity slackness conditions (a) and (b) in Theorem 9.2 are reduced to:

(SC-a) $\overline{x}_S > 0$ implies $\sum_{e\in S} y_e = 1$, for all $S \in \mathcal{S}$,

(SC-b) $\overline{y}_e > 0$ implies $1 \leq \sum_{e\in S\in\mathcal{S}} x_S \leq f$, for all $e \in U$.

The algorithm we present below finds a dual pair of integral feasible solutions satisfying both (SC-a) and (SC-b).

```
procedure PrimalDualSetCover(U, S);
begin
    x := 0; y := 0; mark all elements of U as uncovered;
    while x is infeasible do
        take any uncovered element e ∈ U ;
        increase y_e until some dual constraint(s) become tight;
        set x_S := 1 for all S's that have become tight;
        mark all elements in those tight sets as covered;
    endwhile;
    output (x, y);
end.
```

Theorem 9.3 (Primal-Dual Performance) *The algorithm* `PrimalDualSetCover` *is a factor f approximation algorithm for the set cover problem.*

Proof. Left to the reader. ∎

9.4 LP-Rounding

The LP-rounding technique is an intuitively appealing method to convert an optimal (fractional) solution of the LP relaxation to a "near-by" integral solution of a combinatorial optimization problem.

While the rounding technique may not work at all in general (for example, when there are equality constraints), it does work quite well for the set cover problem.

For rounding, we use $f = f(\mathcal{S})$ the largest frequency. Here is an algorithm that relies on an external LP solver.

```
procedure LProundingSetCover(U, S);
begin
    solve the LP relaxation (9.2) to find an optimal solution x;
    select each set S with x_S ≥ 1/f;
    output the set C of selected sets;
end.
```

Theorem 9.4 (LP-Rounding Performance) *The algorithm* `LProundingSetCover` *is a factor f approximation algorithm for the set cover problem.*

Proof. Let x be the LP optimal solution found and let $\mathcal{C}$ be the output of the algorithm. Because each fixed element e is contained in at most f sets and x is a feasible solution to the LP, among all x_S's with $e \in S$ there is one with value at least $1/f$. This means every element e is covered by $\mathcal{C}$. Clearly, the relative ratio of the resulting objective value and the LP optimal value is at most f. ∎