IEOR 269, Spring 2010
Integer Programming and Combinatorial Optimization

Professor Dorit S. Hochbaum

Contents

These notes are based on "scribe" notes taken by students attending Professor Hochbaum's course IEOR 269 in the spring semester of 2010. The current version has been updated and edited by Professor Hochbaum 2010

Lec1

1 Introduction

Consider the general form of a linear program:

$$\begin{aligned} \max \quad & \sum_{i=1}^{n} c_i x_i \\ \text{subject to} \quad & Ax \leq b \end{aligned}$$

In an *integer* programming optimization problem, the additional restriction that the x_i must be integer-valued is also present.

While at first it may seem that the integrality condition limits the number of possible solutions and could thereby make the integer problem easier than the continuous problem, the opposite is actually true. Linear programming optimization problems have the property that there exists an optimal solution at a so-called *extreme point* (a basic solution); the optimal solution in an integer program, however, is not guaranteed to satisfy any such property and the number of possible integer valued solutions to consider becomes prohibitively large, in general.

While linear programming belongs to the class of problems P for which "good" algorithms exist (an algorithm is said to be good if its running time is bounded by a polynomial in the size of the input), integer programming belongs to the class of *NP-hard* problems for which it is considered highly unlikely that a "good" algorithm exists. For some integer programming problems, such as the *Assignment Problem* (which is described later in this lecture), efficient algorithms do exist. Unlike linear programming, however, for which effective general purpose solution techniques exist (ex. simplex method, ellipsoid method, Karmarkar's algorithm), integer programming problems tend to be best handled in an ad hoc manner. While there are general techniques for dealing with integer programs (ex. branch-and-bound, simulated annealing, cutting planes), it is usually better to take advantage of the structure of the specific integer programming problems you are dealing with and to develop special purposes approaches to take advantage of this particular structure.

Thus, it is important to become familiar with a wide variety of different classes of integer programming problems. Gaining such an understanding will be useful when you are later confronted with a new integer programming problem and have to determine how best to deal with it. This will be a main focus of this course.

One natural idea for solving an integer program is to first solve the "LP-relaxation" of the problem (ignore the integrality constraints), and then round the solution. As indicated in the class handout, there are several fundamental problems to using this as a general approach:

1. The rounded solutions may not be feasible.

2. The rounded solutions, even if some are feasible, may not contain the optimal solution. The solutions of the ILP in general can be arbitrarily far from the solutions of the LP.

3. Even if one of the rounded solutions is optimal, checking all roundings is computationally expensive. There are 2^n possible roundings to consider for an n variable problem, which becomes prohibitive for even moderate sized n.

2 Formulation of some ILP

2.1 0-1 knapsack problem

Given n items with utility u_i and weight w_i for $i \in \{1, \ldots, n\}$, find of subset of items with maximal utility under a weight constraint B. To formulate the problem as an ILP, the variable x_i, $i = 1, \ldots, n$ is defined as follows:

$$x_i = \begin{cases} 1 & \text{if item } \ i \ \text{ is selected} \\ 0 & \text{otherwise} \end{cases}$$

and the problem can be formulated as:

$$\begin{aligned} \max \ & \sum_{i=1}^{n} u_i x_i \\ \text{subject to } & \sum_{i=1}^{n} w_i \, x_i \leq B \\ & x_i \in \{0,1\}, \ \ i \in \{1, \ldots, n\}. \end{aligned}$$

The solution of the LP relaxation of this problem is obtained by ordering the items in decreasing order of u_i/w_i, choosing $x_i = 1$ as long as possible, then choosing the remaining budget for the next item, and 0 for the other ones.

This problem is NP-hard but can be solved efficiently for small instances, and has been proven to be *weakly NP-hard* (more details later in the course).

2.2 Assignment problem

Given n persons and n jobs, if we note $w_{i,j}$ the utility of person i doing job j, for $i, j \in \{1, \ldots, n\}$, find the assignment which maximizes utility. To formulate the problem as an ILP, the variable $x_{i,j}$, for $i, j \in \{1, \ldots, n\}$ is defined as follows:

$$x_{i,j} = \begin{cases} 1 & \text{if person } \ i \ \text{ is assigned to job } \ j \\ 0 & \text{otherwise} \end{cases}$$

and the problem can be formulated as:

$$\begin{aligned} \max \ & \sum_{i=1}^{n} \sum_{j=1}^{n} w_{i,j} \, x_{i,j} \\ \text{subject to } & \sum_{j=1}^{n} x_{i,j} = 1 \ \ i \in \{1 \ldots n\} && (1) \\ & \sum_{i=1}^{n} x_{i,j} = 1 \ \ j \in \{1 \ldots n\} && (2) \\ & x_{i,j} \in \{0,1\}, \ \ i, j \in \{1, \ldots, n\} \end{aligned}$$

where constraint (1) expresses that each person is assigned exactly one job, and constraint (2) expresses that one job is assigned to one person exactly. The constraint matrix $A \in \{0,1\}^{2n \times n^2}$ of this problem has exactly two 1 in each column, one in the upper part and one in the lower part. A column of A representing variable $x_{i',j'}$ has coefficient 1 at line i' and 1 at line $n + j'$, and 0

otherwise. This is illustrated for the case of $n = 3$ in equation (3), where the columns of A are ordered in lexicographic order for $(i, j) \in \{1, \ldots, n\}^2$.

$$A = \begin{pmatrix} 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 \\ 1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 \end{pmatrix} \tag{3}$$

and if we note $b = \big(1, 1, 1, 1, 1, 1, 1, 1, 1\big)^T$ and $X = \big(x_{1,1}, x_{1,2}, x_{1,3}, \ldots, x_{3,1}, x_{3,2}, x_{3,3}\big)^T$ the constraints (1)- (2) in the case of $n = 3$ can be rewritten as $AX = b$.

This property of the constraint matrix can be used to show that the solutions of the LP-relaxation of the assignment problem are integers.

Definition 2.1. *A matrix A is called totally unimodular (TUM) if for any square sub-matrix A' of A,* $\det A' \in \{-1, 0, 1\}$.

Lemma 2.2. *The solutions of a LP with integer objective, integer right-hand side constraint, and TUM constraint matrix are integers.*

Proof. Consider a LP in classical form:

$$\begin{aligned} &\max\ c^T x \\ &\text{subject to}\ \ A\,x \le b \end{aligned}$$

where $c, x \in \mathbf{Z}^n$, $A \in \mathbf{Z}^{m\times n}$ totally unimodular, $b \in \mathbf{Z}^m$. If $m < n$, a basic solution x^*_{basic} of the LP is given by $x^*_{\text{basic}} = B^{-1}\, b$ where $B = (b_{i,j})_{1\le i,j\le m}$ is a non-singular square submatrix of A (columns of B are not necessarily consecutive columns of A).

Using Cramer's rule, we can write: $B^{-1} = C^T / \det B$ where $C = (c_{i,j})$ denotes the cofactor matrix of B. As detailed in (4), the absolute value of $c_{i,j}$ is the determinant of the matrix obtained by removing line i and column j from B.

$$c_{i,j} = (-1)^{i+j} \det \begin{pmatrix} b_{1,1} & \ldots & b_{1,j-1} & X & b_{1,j+1} & \ldots & b_{1,m} \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ b_{i-1,1} & \ldots & b_{i-1,j-1} & X & b_{i-1,j+1} & \ldots & b_{i-1,m} \\ X & X & X & X & X & X & X \\ b_{i+1,1} & \ldots & b_{i+1,j-1} & X & b_{i+1,j+1} & \ldots & b_{i+1,m} \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ b_{m,1} & \ldots & b_{m,j-1} & X & b_{m,j+1} & \ldots & b_{m,m} \end{pmatrix} \tag{4}$$

Since $\det(\cdot)$ is a polynomial function of the matrix coefficients, if B has integer coefficients, so does C. If A is TUM, since B is non-singular, $\det(B) \in \{-1, 1\}$ so B^{-1} has integer coefficients. If b has integer coefficients, so does the optimal solution of the LP. □ □

ILP with TUM constraint matrix can be solved by considering their LP relaxation, whose solutions are integral. The minimum cost network flow problem has a TUM constraint matrix.

3 Non-linear Objective functions

3.1 Production problem with set-up costs

Given n products with revenue per unit p_i, marginal cost c_i, set-up cost f_i, $i \in \{1, \ldots, n\}$, find the production level which maximizes net profit (revenues minus costs). In particular the cost of producing 0 item i is 0, but the cost of producing $\epsilon > 0$ item i is $f_i + \epsilon\, c_i$. The problem is represented in Figure 1.

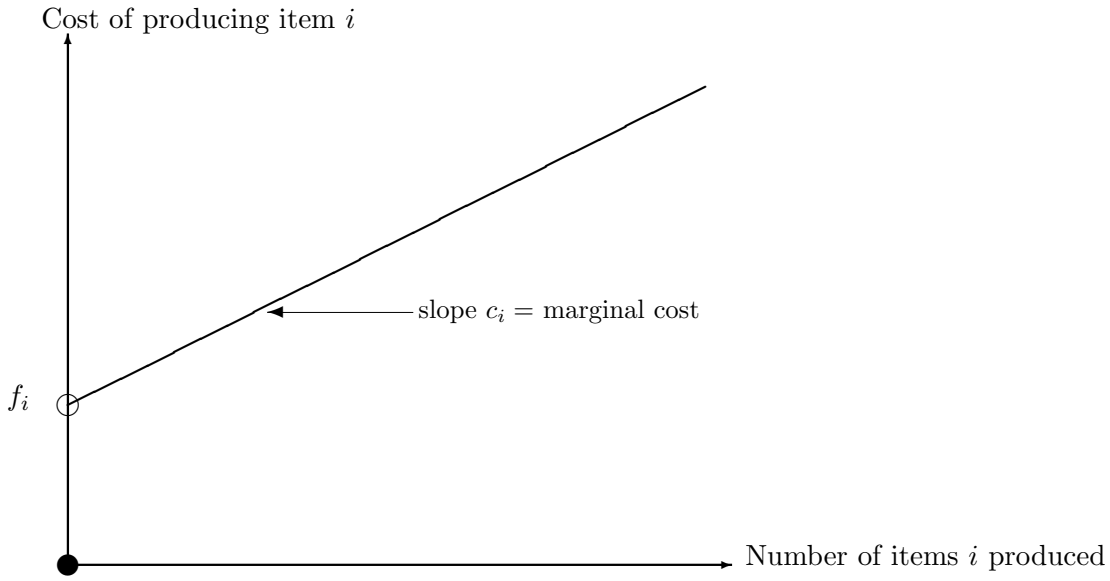


Figure 1: Cost of producing item i

To formulate the problem, the variable y_i, $i \in \{1, \ldots, n\}$ is defined as follows:

$$y_i = \begin{cases} 1 & \text{if } x_i > 0 \\ 0 & \text{otherwise} \end{cases}$$

and the problem can be formulated as:

$$\begin{aligned} \max \ & \sum_{i=1}^{n} (p_i\, x_i - c_i\, x_i - f_i\, y_i) \\ \text{subject to } & x_i \leq M\, y_i, \ \ i \in \{1, \ldots, n\} \\ & x_i \in \mathbf{N}, \ \ i \in \{1, \ldots, n\} \end{aligned}$$

where M is a 'large number' (at least the maximal production capacity). Because of the shape of the objective function, y_i will tend to be 0 in an optimal solution. The constraint involving M enforces that $x_i > 0 \Rightarrow y_i = 1$. Reciprocally if $y_i = 0$ the constraint enforces that $x_i = 0$.

3.2 Piecewise linear cost function

Consider the following problem:

$$\min \sum_{i=1}^{n} f_i(x_i)$$

where $n \in \mathbf{N}$, and for $i \in \{1, \ldots, n\}$, f_i is a piecewise linear function on k_i successive intervals. Interval j for function f_i has length w_i^j, and f_i has slope c_i^j on this interval, $(i, j) \in \{1, \ldots, n\} \times \{1, \ldots, k_i\}$. For $i \in \{1, \ldots, n\}$, the intervals for function f_i are successive in the sense that the supremum of interval j for function f_i is the infimum of interval $j+1$ for function f_i, $j \in \{1, \ldots, k_i - 1\}$. The infimum of interval 1 is 0, and the value of f_i at 0 is f_i^0, for $i \in \{1, \ldots, n\}$. The notations are outlined in Figure 2.

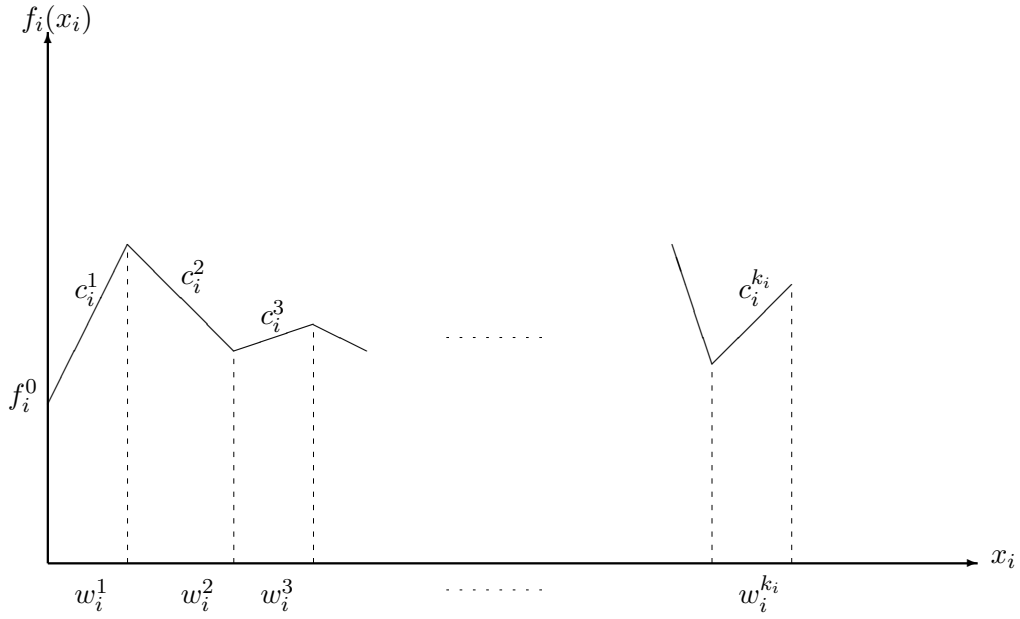


Figure 2: Piecewise linear objective function

We define the variable δ_i^j, $(i, j) \in \{1, \ldots, n\} \times \{1, \ldots, k_i\}$, to be the length of interval j at which f_i is estimated. We have $x_i = \sum_{j=1}^{k_i} \delta_i^j$.

The objective function of this problem can be rewritten $\sum_{i=1}^{n} \left(f_i^0 + \sum_{j=1}^{k_i} \delta_i^j \, c_i^j \right)$. To guarantee that the set of δ_i^j define an interval, we introduce the binary variable:

$$y_i^j = \begin{cases} 1 & \text{if } \delta_i^j > 0 \\ 0 & \text{otherwise} \end{cases}$$

and we formulate the problem as:

$$\begin{aligned} \min \ & \sum_{i=1}^{n} \left(f_i^0 + \sum_{j=1}^{k_i} \delta_i^j \, c_i^j \right) \\ \text{subject to } & w_i^j \, y_i^{j+1} \leq \delta_i^j \leq w_i^j \, y_i^j \ \text{ for } \ (i, j) \in \{1, \ldots, n\} \times \{1, \ldots, k_i\} \\ & y_i^j \in \{0, 1\}, \ \text{ for } \ (i, j) \in \{1, \ldots, n\} \times \{1, \ldots, k_i\} \end{aligned} \tag{5}$$

The left part of the first constraint in (5) enforces that $y_i^{j+1} = 1 \Rightarrow \delta_i^j = w_i^j$, i.e. that the set of δ_i^j defines an interval. The right part of the first constraint enforces that $y_i^j = 0 \Rightarrow \delta_i^j = 0$, that $\delta_i^j \leq w_i^j$, and that $\delta_i^j > 0 \Rightarrow y_i^j = 1$.

In this formulation, problem (5) has continuous and discrete decision variables, it is called a *mixed integer program*. The production problem with set-up cost falls in this category too.

3.3 Piecewise linear convex cost function

If we consider problem (5) where f_i is piecewise linear and convex, $i = \{1 \dots n\}$, the problem can be formulated as a LP:

$$\min \sum_{i=1}^{n} \left(f_i^0 + \sum_{j=1}^{k_i} \delta_i^j \, c_i^j \right)$$
$$\text{subject to } 0 \leq \delta_i^j \leq w_i^j \, y_i^j \text{ for } (i,j) \in \{1,\dots,n\} \times \{1,\dots,k_i\}$$
$$y_i^j \in \{0,1\}, \text{ for } (i,j) \in \{1,\dots,n\} \times \{1,\dots,k_i\}$$

Indeed we don't need as in the piecewise linear case to enforce that for $i \in \{1,\dots,n\}$, the δ_i^j are filled 'from left to right' (left part of first constraint in previous problem) because for $i \in \{1,\dots,n\}$ the slopes c_i^j increase with increasing values of j. For $i \in \{1,\dots,n\}$, an optimal solution will set δ_i^j to its maximal value before increasing the value of δ_i^{j+1}, $\forall j \in \{1,\dots,k_i - 1\}$. This is illustrated in figure 3.

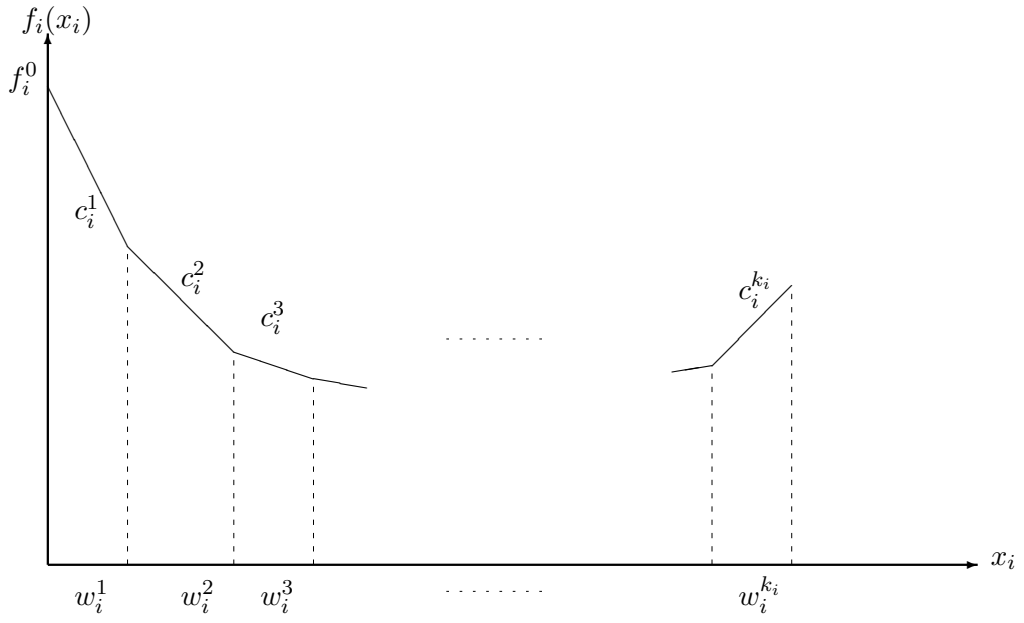


Figure 3: Piecewise linear convex objective function

3.4 Disjunctive constraints

Disjunctive (or) constraints on continuous variables can be written as conjunctive (and) constraints involving integer variables. Given $u_2 > u_1$, the constraints:

$$\begin{cases} x \leq u_1 \text{ or} \\ x \geq u_2 \end{cases}$$

can be rewritten as:

$$\begin{cases} x \leq u_1 + M\,y \text{ and} \\ x \geq u_2 - (M + u_2)\,(1 - y) \end{cases} \tag{6}$$

where M is a 'large number' and y is defined by:

$$y = \begin{cases} 1 & \text{if } x \geq u_2 \\ 0 & \text{otherwise} \end{cases}$$

If $y = 1$, the first line in (6) is automatically satisfied and the second line requires that $x \geq u_2$. If $y = 0$, the second line in (6) is automatically satisfied and the first line requires that $x \leq u_1$. Reciprocally if $x \leq u_1$ the first line is satisfied $\forall y \in \{0, 1\}$, and the second line is satisfied for $y = 0$. If $x \geq u_2$, the second line is satisfied $\forall y \in \{0, 1\}$ and the first line is satisfied for $y = 1$.

4 Some famous combinatorial problems

4.1 Max clique problem

Definition 4.1. *Given a graph $G(V, E)$ where V denotes the set of vertices and E denotes the set of edges, a clique is a subset S of vertices such that $\forall i, j \in S$, $(i, j) \in E$.*

Given a graph $G(V, E)$, the max clique problem consists in finding the clique of maximal size.

4.2 SAT (satisfiability)

Definition 4.2. *A disjunctive clause c_i on a set of binary variables $\{x_1, \ldots, x_n\}$ is an expression of the form $c_i = y_i^1 \vee \ldots \vee y_i^{k_i}$ where $y_i^j \in \{x_1, \ldots, x_n, \neg x_1 \ldots, \neg x_n\}$.*

Given a set of binary variables $\{x_1, \ldots, x_n\}$ and a set of clauses $\{c_1, \ldots, c_p\}$, SAT consists of finding an assignment of the variables $\{x_1, \ldots, x_n\}$ such that c_i is true, $\forall i \in \{1, \ldots, p\}$.

If the length of the clauses is bounded by an integer n, the problem is called n-SAT, and has been proven to be NP-hard in general. Famous instance are 2-SAT and 3-SAT. The different level of difficulty between these two problems illustrates general behavior of combinatorial problems when switching from 2 to 3 dimensions (more later). 3-SAT is a NP-hard problem, but algorithms exist to solve 2-SAT in polynomial time.

4.3 Vertex cover problem

Definition 4.3. *Given a graph $G(V, E)$, S is a vertex cover if $\forall (i, j) \in E$, $i \in S$ or $j \in S$.*

Given a graph $G(V, E)$, the vertex cover problem consists of finding the vertex cover of G of minimal size.

5 General optimization

An optimization problem takes the general form:

$$\begin{aligned} &\text{opt} \;\; f(x) \\ &\text{subject to} \;\; x \in S \end{aligned}$$

where opt $\in \{\min, \max\}$, $x \in \mathbf{R}^n$, $f : \mathbf{R}^n \mapsto \mathbf{R}$. $S \in \mathbf{R}^n$ is called the feasible set. The optimization problem is 'hard' when:

- The function is not convex for minimization (or concave for maximization) on the feasible set S.
- The feasible set S is not convex.

The difficulty with a non-convex function in a minimization problem comes from the fact that a neighborhood of a local minimum and a neighborhood of a global minimum look alike (bowl shape). In the case of a convex function, any local minimum is the global minimum (respectively concave, max).

Lec2

6 Neighborhood

The concept of "local optimum" is based on the concept of neighborhood: We say a point is locally optimal if it is (weakly) better than all other points in the neighbor. For example, in the simplex method, the neighborhood of a basic solution $\bar{x}$ is the set of basic solutions that share all but one basic columns with $\bar{x}$.[1]

For a single problem, we may have different definitions of neighborhood for different algorithms. For example, for the same LP problem, the neighborhoods in the simplex method and in the ellipsoid method are different.

6.1 Exact neighborhood

Definition 6.1. *A neighborhood is* **exact** *if a local optimum with respect to the neighbor is also a global optimum.*

It seems that it will be easier for us to design an efficient algorithm with an exact neighborhood. However, if a neighborhood is exact, does that mean the algorithm using this neighborhood is efficient? The answer is NO! Again, the simplex method provides an example. The neighborhood of the simplex method is exact, but the simplex method is theoretically inefficient.

Another example is the following "trivial" exact neighborhood. Suppose we are solving a problem with the feasible region S. If for any feasible solution we define its neighborhood as the whole set S, we have this neighborhood exact. However, such a neighborhood actually gives us no information and does not help at all.

The concept of exact neighborhood will be used later in this semester.

[1] We shall use the following notations in the future. For a matrix A,

$$\begin{array}{rcccl} A_{i\cdot} & = & (a_{ij})_{j=1,\ldots,n} & = & \text{the } i\text{-th row of } A. \\ A_{\cdot j} & = & (a_{ij})_{i=1,\ldots,m} & = & \text{the } j\text{-th column of } A. \\ B & = & [A_{\cdot j_1} \cdots A_{\cdot j_m}] & = & \text{a basic matrix of } A. \end{array}$$

We say that B' is adjacent to B if $|B' \setminus B| = 1$ and $|B \setminus B'| = 1$.

7 Complexity of algorithms

It is meaningless to use an algorithm that is "inefficient" (the concept of efficiency will be defined precisely later). The example in he handout of "Reminiscences and Gary & Johnson" shows that an inefficient algorithm may not result in a solution in a reasonable time, even if it is correct. Therefore, the complexity of algorithms is an important issue in optimization.

We start by defining polynomial functions.

Definition 7.1. *A function $p(n)$ is a polynomial function of n if $p(n) = \sum_{i=0}^{k} a_i n^i$ for some constants $a_0, a_1, ...,$ and a_k.*

The complexity of an algorithm is determined by the number of operations it requires to solve the problem. Operations that should be counted includes addition, subtraction, multiplication, division, comparison, and rounding. While counting operations is addressed in most of the fundamental algorithm courses, another equally important issue concerning the length of the problem input will be discussed below.

We illustrate the concept of input length with the following examples.

7.1 Finding the maximum element

The problem is to find the maximum element in the set of integers $\{a_1, ..., a_n\}$. An algorithm for this problem is

$i = 1$, $a_{\max} = a_1$
until $i = n$
  $i \leftarrow i + 1$
  $a_{\max} \leftarrow \max\{a_i, a_{\max}\}$[2]
end

The input of this problem includes the numbers n, a_1, a_2, ..., and a_n. Since we need $1 + \lceil \log_2 x \rceil$ bits to represent a number x in the binary representation used by computers, the size of the input is

$$1 + \lceil \log_2 n \rceil + 1 + \lceil \log_2 a_1 \rceil + \cdots + 1 + \lceil \log_2 a_n \rceil \geq n + \sum_{i=1}^{n} \lceil \log_2 a_i \rceil. \tag{7}$$

To complete the algorithm, the operations we need is $n - 1$ comparisons. This number is even smaller than the input length! Therefore, the complexity of this algorithm is at most linear in the input size.

7.2 0-1 knapsack

As defined in the first lecture, the problem is to solve

$$\begin{aligned} \max \quad & \textstyle\sum_{j=1}^{n} u_j x_j \\ \text{s.t.} \quad & \textstyle\sum_{j=1}^{n} v_j x_j \leq B \\ & x_j \in \{0, 1\} \quad \forall j = 1, 2, ..., n, \end{aligned}$$

[2] this statement is actually implemented by

if $a_i \geq a_{\max}$
  $a_{\max} \leftarrow a_i$
end

we restrict parameters to be integers in this section.

The input includes $2n+2$ numbers: n, set of v_j's, set of u_j's, and B. Similar to the calculation in (7), the input size is at least

$$2n + 2 + \sum_{j=1}^{n} \lceil \log_2 u_j \rceil + \sum_{j=1}^{n} \lceil \log_2 v_j \rceil + \lceil \log_2 B \rceil. \tag{8}$$

This problem may be solved by the following dynamic programming (DP) algorithm. First, we define

$$\begin{array}{rcll} f_k(y) & = & \max & \sum_{j=1}^{k} u_j x_j \\ & & \text{s.t.} & \sum_{j=1}^{k} v_j x_j \le y \\ & & & x_j \in \{0,1\} \quad \forall j = 1,2,...,k. \end{array}$$

With this definition, we may have the recursion

$$f_k(y) = \max\Big\{ f_{k-1}(y), f_{k-1}(y - v_k) + u_k \Big\}.$$

The boundary condition we need is $f_k(0) = 0$ for all k from 1 to n. Let $g(y,u,v) = u$ if $y \ge v$ and 0 otherwise, we may then start from $k = 1$ and solve

$$f_1(y) = \max\Big\{ 0, g(y, u_1, v_1) \Big\}.$$

Then we proceed to $f_2(\cdot)$, $f_3(\cdot)$, ..., and $f_n(\cdot)$. The solution will be found by looking into $f_n(B)$.

Now let's consider the complexity of this algorithm. The total number of f functions is nB, and for each f function we need 1 comparison and 2 additions. Therefore, the number of total operations we need is $O(nB) = O(n2^{\log_2 B})$. Note that $2^{\log_2 B}$ is an exponential function of the term $\lceil \log_2 B \rceil$ in the input size. Therefore, if B is large enough and $\lceil \log_2 B \rceil$ dominates other terms in (8), then the number of operations is an exponential function of the input size! However, if B is small enough, then this algorithm works well.

An algorithm with its complexity similar to this $O(nB)$ is called *pseudo-polynomial*.

Definition 7.2. *An algorithm is* **pseudo-polynomial** *time if it runs in a polynomial time with a unary-represented input.*

For the 0-1 knapsack problem, the input size will be $n + \sum_{j=1}^{n}(v_i + u_i) + B$ under unary representation. It then follows that the number of operations becomes a quadratic function of the input size. Therefore, we conclude that the dynamic programming algorithm for the 0-1 knapsack is pseudo-polynomial.

7.3 Linear systems

Given an $n \times n$ matrix A and an $n \times 1$ vector b, the problem is to solve the linear system $Ax = b$. As several algorithms have been proposed for solving linear systems, here we discuss Gaussian elimination: through a sequence of elementary row operations, change A to a lower-triangular matrix.

For this problem, the input is the matrix A and the vector b. To represent the $n^2 + n$ numbers, the number of bits we need is bounded below by

$$n^2 + \sum_{i=1}^{n} \sum_{j=1}^{n} \lceil \log_2 a_{ij} \rceil + \sum_{i=1}^{n} \lceil \log_2 b_i \rceil.$$

On the other hand, the number of operations we need is roughly $O(n^3)$: For each pair of the $O(n^2)$ rows, we need n additions (or subtractions) and n multiplications. Therefore, the number of operations is a polynomial function on the input size.

However, there is an important issue we need to clarify here. When we are doing complexity analysis, we must be careful if multiplications and divisions are used. While additions, subtractions, and comparisons will not bring this problem, multiplications and divisions may increase the size of numbers. For example, after multiplying two numbers a and b, we need $\lceil \log_2 ab \rceil \doteqdot \lceil \log_2 a \rceil + \lceil \log_2 b \rceil$ bits to represent a single number ab. If we further multiply ab by another number, we may need even more bits to represent it. This exponentially increasing length of numbers may bring two problems:

- The length of the number may go beyond the storage limit of a computer.
- It actually takes more time to do operations for "long" numbers.

In short, when ever we have multiplications and divisions in our algorithms, we must make sure that the length of numbers does not grow exponentially, so that our algorithm is still polynomial-time, as we desire.

For Gaussian elimination, the book by Schrjiver has Edmond's proof (Theorem 3.3) that, at each iteration of Gaussian elimination, the numbers grow only polynomially (by a factor of ≤ 4 with each operation). With this in mind, we can conclude that Gaussian elimination is a polynomial-time algorithm.

7.4 Linear Programming

Given an $m \times n$ matrix A, an $m \times 1$ vector b, and an $n \times 1$ vector c, the linear program to solve is

$$\begin{aligned} \max \quad & c^T x \\ \text{s.t.} \quad & Ax = b \\ & x \geq 0. \end{aligned}$$

With the parameters A, b, and c, the input size is bounded below by

$$mn + \sum_{i=1}^{m} \sum_{j=1}^{n} \lceil \log_2 a_{ij} \rceil + \sum_{i=1}^{m} \lceil \log_2 b_i \rceil + \sum_{j=1}^{n} \lceil \log_2 c_j \rceil.$$

Now let's consider the complexity of some algorithms for linear programming. As we already know, the simplex method may need to go through almost all basic feasible solutions in some instances. This fact makes the simplex method an exponential-time algorithm. On the other hand, the ellipsoid method has been proved to be a polynomial-time algorithm for linear programming. Let n be the number of variables and

$$L = \log_2 \Big(\max\{\det B \mid B \text{ is a submatrix of } A\} \Big),$$

it has been shown that the complexity of the ellipsoid method is $O(n^6 L^2)$.

It is worth mentioning that in practice we still prefer the simplex method to the ellipsoid method, even if theoretically the former is inefficient and the latter is efficient. In practice, the simplex method is usually faster than the ellipsoid method. Also note that the complexity of the ellipsoid method depends on the values of A. In other words, in we keep the numbers of variables and constraints the same but change the coefficients, we may result in a different running time. This does not happen in running the simplex method.

8 Some interesting IP formulations

8.1 The fixed cost plant location problem

We are given a set of locations, each with a market on it. The problem is to choose some locations to build plants (facilities), which require different fixed costs. Once we build a plant, we may serve a market by this plant with a service costs proportional to the distance between the two locations. The problem is to build plants and serve all markets with the least total cost.

Let $G = (V, E)$ be an instance of this problem, where V is the set of locations and E is the set of links. For each link $(i, j) \in E$, let d_{ij} be the service cost; for each node $i \in V$, let f_i be the fixed construction cost. It is assumed that G is a complete graph.

To model this problem, we define the decision variables

$$y_j = \begin{cases} 1 & \text{if location } j \text{ is selected to build a plant} \\ 0 & \text{otherwise} \end{cases} \quad \text{for all } j \in V, \text{ and}$$

$$x_{ij} = \begin{cases} 1 & \text{if market } i \text{ is served by plant } j \\ 0 & \text{otherwise} \end{cases} \quad \text{for all } i \in V, j \in V.$$

Then we may formulate the problem as

$$\begin{aligned} \min \quad & \sum_{j \in V} f_j y_j + \sum_{i \in V} \sum_{j \in V} d_{ij} x_{ij} \\ \text{s.t.} \quad & x_{ij} \le y_j \quad \forall\, i \in V, j \in V \\ & \sum_{j \in V} x_{ij} = 1 \quad \forall\, i \in V \\ & x_{ij}, y_j \in \{0, 1\} \quad \forall\, i \in V, j \in V \end{aligned}$$

An alternative formulation has the first set of constraints represented more compactly:

$$\sum_{i \in V} x_{ij} \le n \cdot y_j \quad \forall j \in V.$$

The alternative formulation has fewer constraints. However, *this does not imply it is a better formulation.* In fact, for integer programming problems, usually we prefer a formulation with tighter constraints. This is because tighter constraints typically result in an LP polytope that is closer to the convex hull of the IP feasible solutions.

8.2 Minimum/maximum spanning tree (MST)

The minimum (or maximum, here we discuss the minimum case) spanning tree problem is again defined on a graph $G = (V, E)$. For each (undirected) link $[i, j] \in E$, there is a weight w_{ij}. The problem is to find a spanning tree for G, which is defined below.

Definition 8.1. *Given a graph $G = (V, E)$, $T = (V, E_T \subseteq E)$ is a spanning tree (edge induced acyclic subgraph) if T is connected and acyclic.*

The weight of a spanning tree T is defined as the total weights of the edges in T. The problem is to find the spanning tree for G with the minimum weight.

This problem can be formulated as follows. First we define

$$x_{ij} = \begin{cases} 1 & \text{if edge } [i, j] \text{ is in the tree} \\ 0 & \text{otherwise} \end{cases} \quad \text{for all } [i, j] \in E.$$

Then the formulation is

$$\begin{aligned}
\min \quad & \sum_{[i,j]\in E} w_{ij}x_{ij} \\
\text{s.t.} \quad & \sum_{i\in S, j\in S, [i,j]\in E} x_{ij} \leq |S|-1 \quad \forall\, S\subset V \\
& \sum_{[i,j]\in E} x_{ij} = |V|-1 \\
& x_{ij}\geq 0 \quad \forall\, [i,j]\in E.
\end{aligned}$$

We mention two things here:

- Originally, the last constraint $x_{ij} \geq 0$ should be $x_{ij} \in \{0, 1\}$. It can be shown that this binary constraint can be relaxed without affecting the optimal solution.
- For each subset of V, we have a constraint. Therefore, the number of constraints is $O(2^n)$, which means we have an exponential number of constraints.

The discussion on MST will be continued in the next lecture.

Lec3

9 The Minimum Spanning Tree (MST) Problem

Definition: Let there be a graph $G = (V, E)$ with weight c_{ij} assigned to each edge $e = (i, j) \in E$, and let $n = |V|$.
The MST problem is finding a tree that connects all the vertices of V and is of minimum total edge cost.

The MST has the following linear programming (LP) formulation:

Decision Variables:

$$X_{ij} = \begin{cases} 1 & \text{if edge e=(i,j) is selected} \\ 0 & \text{o/w} \end{cases}$$

$$\begin{aligned}
\min \quad & \sum_{(i,j)\in E} c_{ij}.X_{ij} \\
\text{s.t} \quad & \sum_{(i,j)\in E} X_{ij} = n-1 \\
& \sum_{\substack{i,j\in S \\ (i,j)\in E}} X_{ij} \leq |S|-1 \qquad\qquad (\forall S\subset V) \\
& X_{ij}\geq 0
\end{aligned}$$

The optimum solution to the (LP) formulation is an integer solution, so it is enough to have nonnegativity constraints instead of binary variable constraints (Proof will come later). One significant property of this (LP) formulation is that it contains exponential number of constraints with respect to the number of vertices. With Ellipsoid Method, however, this model can be solved optimally in polynomial time. As we shall prove later, given a polynomial time *separation oracle*, the ellipsoid method finds the optimal solution to an LP in polynomial time. A separation oracle is an algorithm that given a vector, it finds a violated constraint or asserts that the vector is feasible.

There are other problem formulations with the same nature, meaning that nonnegativity constraints are enough instead of the binary constraints. Before proving that the MST formulation is in fact correct, we will take a look at these other formulations:

10 General Matching Problem

Definition: Let there be a graph $G = (V, E)$. A set $M \subseteq E$ is called a matching if $\forall v \in V$ there exists at most one $e \in M$ adjacent to v.
General Matching Problem is finding a feasible matching M so that $|M|$ is maximized. This Maximum Cardinality Matching is also referred to as the Edge Packing Problem.

10.1 Maximum Matching Problem in Bipartite Graphs

Definition: In a bipartite graph, $G = (V, E_B)$ the set of vertices V can be partitioned into two disjoint sets V_1 and V_2 such that every edge connects a vertex in V_1 to another one in V_2. That is, no two vertices in V_1 have an edge between them, and likewise for V_2 (Please see Figure 4).

The formulation of Maximum Matching Problem in Bipartite Graph is as follows:

$$X_{ij} = \begin{cases} 1 & \text{if edge e=(i,j) is selected} \\ 0 & \text{o/w} \end{cases}$$

$$\begin{aligned} \max \quad & \sum_{(i,j)\in E_B} X_{ij} \\ \text{s.t} \quad & \sum_{\substack{(i,j)\in E_B \\ j\in V}} X_{ij} \leq 1 \\ & X_{ij} \geq 0 \end{aligned}$$

Remark: Assignment Problem is a Matching Problem on a complete Bipartite Graph with edge weights not necessarily 1.

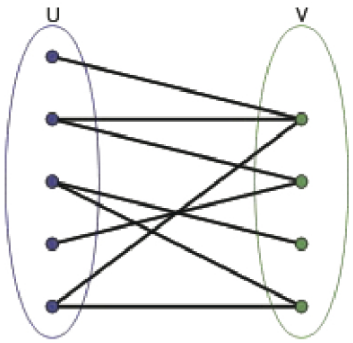


Figure 4: Bipartite Graph

10.2 Maximum Matching Problem in Non-Bipartite Graphs

When the graph $G = (V, E)$ is non-bipartite, the formulation above, replacing E_B by E does not have integer extreme points and therefore does not solve the integer problem unless we add the integrality requirement. In a non-bipartite graph there are odd cycles. In that case, the optimum solution could be non-integer. As an example, consider a cycle with 3 vertices i, j, k. The best integer solution for this 3-node cycle would be 1, as we can select at most one edge. However, in case of Linear Model, the optimum would be 1.5, where each edge is assigned 1/2 (Please see Figure 5).

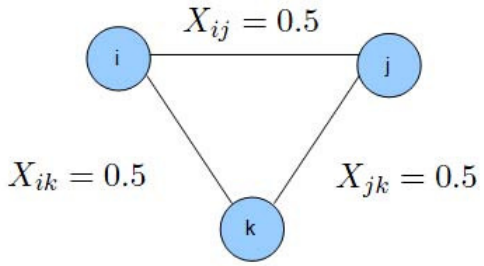


Figure 5: Odd Cycle Problem

In order to exclude fractional solutions involving odd cycles, we need to add the following (exponentially many) constraints to the above formulation.

$$\sum_{\substack{(i,j)\in E \\ j\in S}} X_{ij} \leq \frac{|S|-1}{2} \quad S \subseteq V (\text{where } |S| \text{ is odd})$$

Jack Edmonds showed that adding this set of constraints is sufficient to guarantee the integrality of the solutions to the LP formulation of the general matching problem.

10.3 Constraint Matrix Analysis for Matching Problems

In the Bipartite Matching Problem, the constraint matrix is totally unimodular. Each column has exactly two 1's, moreover the set rows can be partitioned into two sets such that on each set there is only one 1 on each column (Figure 6). In the Nonbipartite Matching Problem, however, although each column has again two 1's, it is impossible to partition the rows of the constraint matrix as in the bipartite case (see Figure 7).
Indeed, the non-bipartite matching constraint matrix (without the odd sets constraints) is *not* totally unimodular. For example for a graph that is a triangle, the determinant of the corresponding constraint matrix is 2, hence it is not unimodular:

$$\begin{matrix} 1 & 0 & 1 \\ 1 & 1 & 0 \\ 0 & 1 & 1 \end{matrix}$$

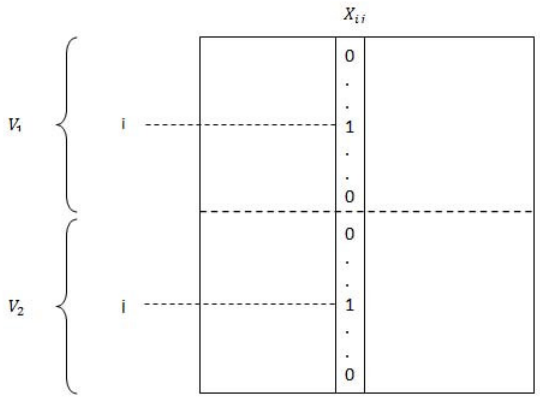


Figure 6: Bipartite Graph Constraint Matrix

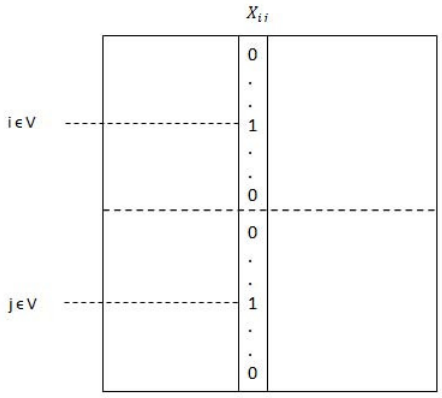


Figure 7: Non-Bipartite Graph Constraint Matrix

11 Traveling Salesperson Problem (TSP)

Given a graph $G = (V, E)$ with each edge $(i, j) \in E$ having a weight c_{ij}. Our aim is to find a tour which visits each node exactly once with minimum total weighted cost.

Definition: Degree of a node is the number of adjacent edges to the node. In the Figure 8, degree of the nodes are as follows:

$$deg_1 = 2, deg_2 = 3, deg_3 = 2, deg_4 = 3, deg_5 = 3, deg_6 = 1$$

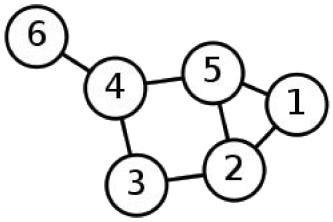


Figure 8: Example Graph

Remark:

$$\sum_{i \in V} deg_i = 2\,|E|$$

Definition: A directed Graph is a graph where each edge is directed from one vertex to another, and is called an arc. A directed Graph is denoted by $G = (V, A)$.
To convert an undirected graph into a directed one, we replace each edge $(i, j) \in E$ by two arcs $(i, j)\&(j, i)$ in A.
Definition: The *indegree* of a node is the number of the incoming arcs to the node; the *outdegree* of a node is the number of the outgoing arcs from the node.
For each node in a cycle, the outdegree and indegree are both equal to 1.

11.1 IP Formulation for TSP

It is easier to give a formulation for the TSP if we represent each edge with two arcs.
Decision Variable:

$$X_{ij} = \begin{cases} 1 & \text{if arc a=(i,j) is traversed} \\ 0 & \text{o/w} \end{cases}$$

The formulation is:

$$\min \sum_{(i,j)\in A} c_{ij}.X_{ij}$$

s.t

$$(1) \sum_{\substack{(i,j)\in A \\ j\in V}} X_{ij} = 1 \qquad \forall i \in V$$

$$(2) \sum_{\substack{(i,j)\in A \\ i\in V}} X_{ij} = 1 \qquad \forall j \in V$$

$$(3) \sum_{\substack{i\in S \\ j\in \overline{S}}} X_{ij} \geq 1 \qquad \forall S \subset V, \overline{S} = V/S$$

$$(4) X_{ij} \in \{0,1\} \qquad \forall (i,j) \in A$$

Constraint sets (1) and (2) state that the outdegree and the indegree, respectively, for each node will be 1. Constraint set (3) is the Subtour Elimination constraints which prevents a solution with several (not connected) subtours instead of a single tour.
This formulation has an exponential number of constraints, and the binary requirement cannot be omitted – without it there are optimal fractional solutions (try to find an example). There are other alternative formulations to the TSP problem some of which are compact. For instance, the Tucker formulation (given in the book of Papadimitriou and Stiglitz) is compact – has a polynomial number of constraints. However, that formulation is inferior in terms of the quality of the respective LP relaxation and is not used in practice. We will see later in the course, the 1-tree formulation of the TSP, also with an exponential number of constraints.

12 Discussion of LP-Formulation for MST

Theorem 12.1. *The formulation is correct, i.e.* X_{ij} *is a* $(0-1)$ *vector and the edges on which* $X_{ij} = 1$ *form a minimum spanning tree.*

Proof: Let E^* be the set of edges for which the solution to (LP) is positive:

$$E^* = \{e \in E | X_e > 0\}$$

.

Proposition 12.1. $X_e \leq 1$, $\forall e \in E^*$.

Proof of Proposition 12.1: For edge $e = (i,j)$ set $S = \{i,j\}$. Then, the corresponding constraint yields $X_{ij} \leq 2 - 1 = 1$.

Proposition 12.2. (V, E^*) *contains no cycles.*

Proof of Proposition 12.2:

Step 1: *There cannot be a cycle with only integer values (1's).*

Proof of Step 1: Let the set of vertices the cycle passes through be denoted by R, then:

$$|R| \leq \sum_{\substack{e=(i,j) \\ i,j \in R}} X_e \leq |R| - 1$$

Contradiction!

Before Step 2, let's prove a lemma which we will use later:

Definition: We call a set of vertices R tight if

$$\sum_{\substack{e=(i,j) \\ i,j \in R}} X_e = |R| - 1$$

Lemma 12.3. *No fractional tight cycles share a fractional edge.*

Proof of Lemma 12.3: Suppose there were two such cycles C_1 and C_2 with k and ℓ vertices, respectively, sharing p edges as shown in Figure 9.

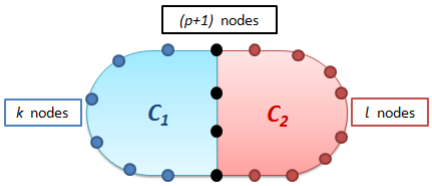


Figure 9: 2 tight fractional cycles

Let P denote the p-path consisting of the common edges, i.e. contains $(p+1)$ vertices.

The total value of X_e summed on these edges of the union of cycles is:

$$\sum_{e \in C_1 \cup C_2} X_e = (\ell - 1) + (k - 1) - \sum_{e \in P} X_e$$

For all subset $S \subset V$, we had the following constraint in our formulation:

$$\sum_{\substack{i,j \in S \\ (i,j) \in E}} X_{ij} \leq |S| - 1$$

Let define $S = C_1 \cup C_2$, then:

$$(\ell - 1) + (k - 1) - \sum_{e \in P} X_e \leq k + \ell - (p + 1) - 1$$

$$p \leq \sum_{e \in P} X_e$$

$\Rightarrow X_e = 1 \ \forall e \in P$

Done!

Step 2: *There is no fractional tight cycles.*

Proof of Step 2: Suppose there were fractional tight cycles in (V, E^*). Consider the one containing the least number of fractional edges. Each such cycle contains at least two fractional edges $e_1 \& e_2$ (else, the corresponding constraint is not tight).
Let $c_{e_1} \leq c_{e_2}$ and $\theta = \min\{1 - X_{e_1}, X_{e_2}\}$.
The solution $\{X_e^{'}\}$ with:

$$(X_{e_1}^{'} = X_{e_1} + \theta), (X_{e_2}^{'} = X_{e_2} - \theta) \text{ and } (X_e^{'} = X_e, \forall e \neq e_1, e_2)$$

is feasible and at least as good as the optimal solution $\{X_e\}$. The feasibility comes from the Lemma 12.3. We are sure that we do not violate any other nontight fractional cycles which share the edge e_1. Otherwise, we should have selected θ as the minimum slack on that nontight fractional cycle and update X_e accordingly. But, this would incur a solution where two fractional tight cycle share the fractional edge $e_1^{'}$ which cannot be the case due to Lemma 12.3.
If there were nontight fractional cycles, then we could repeat the same modification for the fractional edges without violating feasibility. Therefore, there are no fractional cycles in (V, E^*).

The last step to prove that (LP) is the correct form will be to show that the resulting graph is connected. (Assignment 2)
Lec4

13 Branch-and-Bound

13.1 The Branch-and-Bound technique

The Branch-and-Bound Technique is a method of implicitly (rather than explicitly) enumerating all possible feasible solutions to a problem.

Summary of LP-based Branch-and-Bound Technique (for maximization of pure integer linear programs)

Step 1 *Initialization*: Begin with the entire set of solutions under consideration as the only **remaining set**. Set $Z_L = -\infty$. Throughout the algorithm, the best know feasible solution is referred to as the **incumbent solution**, and its objective value Z_L is a lower bound on the maximal

objective value. Compute an upper bound Z_U on the maximal objective value by solving the LP-relaxation.

Step 2 *Branch*: Use some *branch rule* to select *one* of the *remaining* subsets (those neither fathomed nor partitioned) and partition into two new subsets of solutions. A popular branch rule is the **best bound rule**.

The **best bound rule** says to select the subset having the *most favorable bound* (the highest upper bound Z_U) because this subset would seem to be the most promising one to contain an optimal solution.

Partition is performed by arbitrarily selecting an integer variable x_j with a non integer value v and partitioning the subsets into two distinct subsets, one with the additional constraint $x_j \geq \lceil v \rceil$ and one with the additional constraint $x_j \leq \lfloor v \rfloor$.

Step 2 *Bound*: For each new subsets, obtain an upper bound Z_U on the value of the objective function for the feasible solutions in the subsets. Do this by solving the appropriate LP-relaxation.

Step 3 *Fathoming*: For each new subset, exclude it from further consideration if

1 $Z_U \leq Z_L$ (the best Z-value that could be obtained from continuing on is not any better than the Z-value of the best **known** feasible solution); or

2 The subset is found to contain no feasible solutions. (i.e., LP relaxation is infeasible); or

3 The optimal solution to the LP relaxation satisfies the integrality requirements and, therefore, must be the best solution in the subset (Z_U corresponds to its objective function value); if $Z_U \geq Z_L$, then reset $Z_L = Z_U$, store this solution as the incumbent solution, and reapply fathoming step 1 to all remaining subsets.

Step 5 *Stopping rule*: Stop the procedure when there are no *remaining* (unfathomed) subsets; the current *incumbernt* colution is *optimal*. Otherwise, return to the branch step. (If Z_L still equals $-\infty$, then the problem possesses no feasible solutions)

Alternatively, if Z_U^* is the largest Z_U among the remaining subsets (unfathomed and upartitioned), stop when the maximum error $\frac{Z_U^* - Z_L}{Z_L}$ is sufficiently small

Example:

$$\begin{aligned} \max z = \quad & 5x_1 + 4x_2 \\ \text{s.t.} \quad & 6x_1 + 13x_2 \leq 67 \\ & 8x_1 + 5x_2 \leq 55 \\ & x_1, x_2 \text{ non-negative integer} \end{aligned}$$

The solution technique for example is illustrated in Figure 10
The updates of the bounds are shown in the following table. Notice that, at any point in the execution of the algorithm, the optimality gap (relative error) is $\frac{Z_{IP}^* - Z_L}{Z_{IP}^*}$. However, since we do not know Z_{IP}^* then we use an upper bound on the optimality gap instead. In particular note that $\frac{Z_{IP}^* - Z_L}{Z_{IP}^*} \leq \frac{Z_U - Z_L}{Z_L}$, therefore $\frac{Z_U - Z_L}{Z_L}$ is used as an upper bound to the optimality gap.

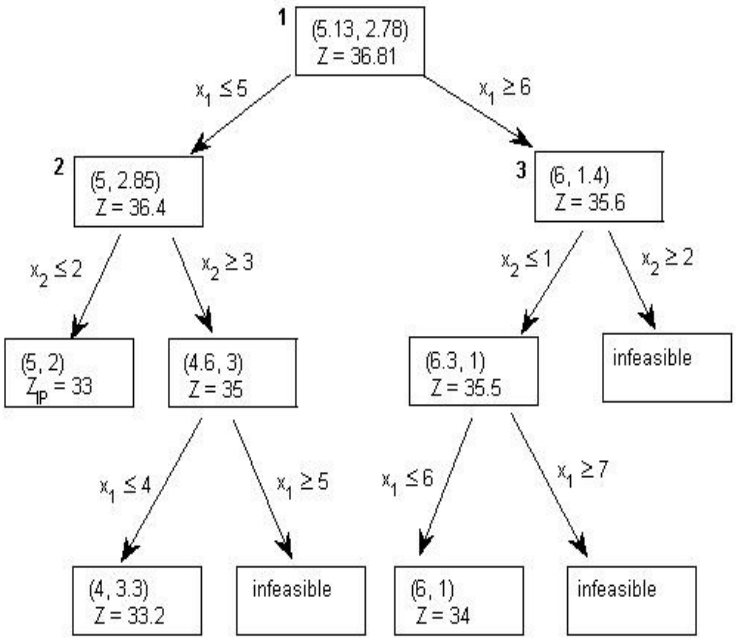


Figure 10: Solution of the example for LP-based Branch-and-Bound

Subproblem	Z_L	Z_U	$\frac{Z_U - Z_L}{Z_L}$
1	$-\infty$	36	∞
2	33	35	6.06%
3	34	34	0%

13.2 Other Branch-and-Bound techniques

There are other branch and bound techniques. The main difference between the different branch and bound techniques is how are the bounds obtained. In particular, there are several ways of relaxing an integer program, and each of these relaxations will give a different bound. In most cases there is a balance between how fast a particular relaxation can be solved and the quality of the bound we get. Usually the harder the relaxation, the better bound we obtain. An example of a different type of branch and bound technique for the directed traveling salesperson problem (TSP) is given below.

For the directed version of traveling salesman problem(TSP), a particular formulation discussed in class has the following decision variables: x_{ij} is the binary variable indicating whether node j follows node i in the tour. In this formulation we have two kinds of constraints :

- $\sum_i x_{ij} = 1$ and $\sum_j x_{ij} = 1$ (each node must have degree of 2)
- $\sum_{i \in S, j \notin S} x_{ij} \geq 1 \; \varnothing \subsetneq S \subsetneq V$ (subtour elimination constaints)

Notice without the subtour elimination constraints, the problem becomes assignment problem which can be solved efficiently. So a bound for TSP may be obtained by relaxing the subtour elimination constraints.

14 Basic graph definitions

- A graph or undirected graph G is an ordered pair $G := (V, E)$. Where V is a set whose elements are called vertices or nodes, and E is a set of unordered pairs of vertices of the form $[i, j]$, called edges.
- A directed graph or digraph G is an ordered pair $G := (V, A)$. Where V is a set whose elements are called vertices or nodes, and A is a set of ordered pairs of vertices of the form (i, j), called arcs. In an arc (i, j) node i is called the *tail* of the arc and node j the *head* of the arc. We sometimes abuse of the notation and refer to a digraph also as a graph.
- A path (directed path) is an ordered list of vertices $(v_1, \ldots, v_k)$, so that $(v_i, v_{i+1}) \in E$ $((v_i, v_{i+1}) \in A)$ for all $i = 1 \ldots, k$. The length of a path is $|(v_1, \ldots, v_k)| = k$.
- A cycle (directed cycle) is an ordered list of vertices $v_0, \ldots, v_k$, so that $(v_i, v_{i+1}) \in E$ $((v_i, v_{i+1}) \in A)$ for all $i = 1, 2, \ldots, n$ and $v_0 = v_k$. The length of a cycle is $|(v_0, \ldots, v_k)| = k$.
- A simple path (simple cycle) is a path (cycle) where all vertices $v_1, \ldots, v_k$ are distinct.
- An (undirected) graph is said to be *connected* if, for every pair of nodes, there is an (undirected) path starting at one node and ending at the other node.
- A directed graph is said to be *strongly connected* if, for every (ordered) pair of nodes (i, j), there is a directed path in the graph starting in i and ending in j.
- The *degree* of a vertex is the number of edges incident to the vertex. $\sum_{v \in V} degree(v) = 2|E|$.
- In a directed graph the *indegree* of a node is the number of incoming arcs that have that node as a head. The outdegree of a node is the number of outgoing arcs from a node, that have that node as a tail.

  Be sure you can prove,
  $$\sum_{v \in V} indeg(v) = |A|,$$
  $$\sum_{v \in V} outdeg(v) = |A|.$$
- A *tree* can be characterized as a connected graph with no cycles. The relevant property for this problem is that a tree with n nodes has $n - 1$ edges.

  Definition: An undirected graph $G = (V, T)$ is a tree if the following three properties are satisfied:

  Property 1: $|T| = |V| - 1$.
  Property 2: G is connected.
  Property 3: G is acyclic.

  (Actually, any two of the properties imply the third as you are to prove in Assignment 2).
- A graph is *bipartite* if the vertices in the graph can be partitioned into two sets in such a way that no edge joins two vertices in the same set.
- A *matching* in a graph is set of graph edges such that no two edges in the set are incident to the same vertex.
  The *bipartite (nonbipartite) matching problem*, is stated as follows: Given a bipartite (nonbipartite) graph $G = (V, E)$, find a maximum cardinality matching.

15 Complexity analysis

15.1 Measuring quality of an algorithm

Algorithm: One approach is to enumerate the solutions, and select the best one.
Recall that for the assignment problem with 70 people and 70 tasks there are $70! \approx 2^{332.4}$ solutions.
The existence of an algorithm does not imply the existence of a good algorithm!
To measure the complexity of a particular algorithm, we count the number of operations that are performed as a function of the 'input size'. The idea is to consider each elementary operation (usually defined as a set of simple arithmetic operations such as $\{+, -, \times, /, \leq\}$) as having unit cost, and measure the number of operations (in terms of the size of the input) required to solve a problem. The goal is to measure the rate, ignoring constants, at which the running time grows as the size of the input grows; it is an asymptotic analysis.
Complexity analysis is concerned with counting the number of operations that must be performed in the worst case.

Definition 15.1 (Concrete Complexity of a problem)**.** *The complexity of a problem is the complexity of the algorithm that has the lowest complexity among all algorithms that solve the problem.*

15.1.1 Examples

Set Membership - Unsorted list: We can determine if a particular item is in a list of n items by looking at each member of the list one by one. Thus the number of comparisons needed to find a member in an unsorted list of length n is n.
Problem: given a real number x, we want to know if $x \in S$.
Algorithm:
1. Compare x to s_i
2. Stop if $x = s_i$
3. else if $i \leftarrow i + 1 < n$ goto 1 else stop x is not in S

Complexity = n comparisons in the worst case. This is also the **concrete complexity** of this problem. Why?

Set Membership - Sorted list: We can determine if a particular item is in a list of n elements via **binary search**. The number of comparisons needed to find a member in a sorted list of length n is proportional to $\log_2 n$.
Problem: given a real number x, we want to know if $x \in S$.
Algorithm:
1. Select $s_{med} = \lfloor \frac{first+last}{2} \rfloor$ and compare to x
2. If $s_{med} = x$ stop
3. If $s_{med} < x$ then $S = (s_{med+1}, \ldots, s_{last})$ else $S = (s_{first}, \ldots, s_{med-1})$
4. If $first < last$ goto 1 else stop

Complexity: after k^{th} iteration $\frac{n}{2^{k-1}}$ elements remain. We are done searching for k such that $\frac{n}{2^{k-1}} \leq 2$, which implies:

$$\log_2 n \leq k$$

Thus the total number of comparisons is at most $\log_2 n$.

Aside: This binary search algorithm can be used more generally to find the *zero* in a *monotone increasing* and *monotone nondecreasing* functions.

Matrix Multiplication: The straightforward method for multiplying two $n \times n$ matrices takes n^3 multiplications and $n^2(n-1)$ additions. Algorithms with better complexity (though not

necessarily practical, see comments later in these notes) are known. Coppersmith and Winograd (1990) came up with an algorithm with complexity $Cn^{2.375477}$ where C is large. Indeed, the constant term is so large that in their paper Coppersmith and Winograd admit that their algorithm is impractical in practice.

Forest Harvesting: In this problem we have a forest divided into a number of cells. For each cell we have the following information: H_i - benefit for the timber company to harvest, U_i - benefit for the timber company not to harvest, and B_{ij} - the border effect, which is the benefit received for harvesting exactly one of cells i or j. This produces an m by n grid. The way to solve is to look at every possible combination of harvesting and not harvesting and pick the best one. This algorithm requires (2^{mn}) operations.

An algorithm is said to be *polynomial* if its running time is bounded by a polynomial in the size of the input. All but the forest harvesting algorithm mentioned above are polynomial algorithms.

An algorithm is said to be *strongly polynomial* if the running time is bounded by a polynomial in the size of the input *and* is independent of the numbers involved; for example, a max-flow algorithm whose running time depends upon the size of the arc capacities is *not* strongly polynomial, even though it may be polynomial (as in the scaling algorithm of Edmonds and Karp). The algorithms for the sorted and unsorted set membership have strongly polynomial running time. So does the greedy algorithm for solving the minimum spanning tree problem. This issue will be returned to later in the course.

Sorting: We want to sort a list of n items in nondecreasing order.

Input: $S = \{s_1, s_2, \ldots, s_n\}$

Output: $s_{i1} \leq s_{i2} \leq \cdots \leq s_{in}$

Bubble Sort: $n' = n$

```
While n' ≥ 2
    i = 1
    while i ≤ n' − 1
        If s_i > s_{i+1} then t = s_{i+1}, s_{i+1} = s_i, s_i = t
        i = i + 1
    end while
    n' ← n' − 1
end while
Output {s_1, s_2, ..., s_n}
```

Basically, we iterate through each item in the list and compare it with its neighbor. If the number on the left is greater than the number on the right, we swap the two. Do this for all of the numbers in the array until we reach the end. Then we repeat the process. At the end of the first pass, the last number in the newly ordered list is in the correct location. At the end of the second pass, the last and the penultimate numbers are in the correct positions. And so forth. So we only need to repeat this process a maximum of n times.

The complexity of this algorithm is: $\sum_{k=2}^{n}(k-1) = n(n-1)/2 = O(n^2)$.

Merge sort (a recursive procedure):

This you need to analyze in Assignment 2, so it is omitted here.

15.2 Growth of functions

n	$\lceil \log n \rceil$	$n-1$	2^n	$n!$
1	0	0	2	1
3	2	2	8	6
5	3	4	32	120
10	4	9	1024	3628800
70	7	69	2^{70}	$\approx 2^{332}$

We are interested in the asymptotic behavior of the running time.

15.3 Definitions for asymptotic comparisons of functions

We define for functions f and g,

$$f, g \ : \ Z^+ \to [0, \infty) \ .$$

1. $f(n) \in O(g(n))$ if $\exists$ a constant $c > 0$ such that $f(n) \leq cg(n)$, for all n sufficiently large.
2. $f(n) \in o(g(n))$ if, for any constant $c > 0$, $f(n) < cg(n)$, for all n sufficiently large.
3. $f(n) \in \Omega(g(n))$ if $\exists$ a constant $c > 0$ such that $f(n) \geq cg(n)$, for all n sufficiently large.
4. $f(n) \in \omega(g(n))$ if, for any constant $c > 0$, $f(n) > cg(n)$, for all n sufficiently large.
5. $f(n) \in \Theta(g(n))$ if $f(n) \in O(g(n))$ and $f(n) \in \Omega(g(n))$.

Examples:

- Bubble sort has complexity $O(n^2)$
- Matrix multiplication has complexity $O(n^3)$
- Gaussian elimination $O(n^3)$
- $2^n \notin O(n^3)$
- $n^3 \in 0(2^n)$
- $n^3 \in o(2^n)$
- $n^4 \in \Omega(n^3.5)$
- $n^4 \in \Omega(n^4)$

15.4 Properties of asymptotic notation

We mention a few properties that can be useful when analyzing the complexity of algorithms.

Proposition 15.2. *$f(n) \in \Omega(g(n))$ if and only if $g(n) \in O(f(n))$.*

The next property is often used in conjunction with *L'hôpital's Rule.*

Proposition 15.3. *Suppose that $\lim_{n\to\infty} \frac{f(n)}{g(n)}$ exists:*

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

Then,

1. *$c < \infty$ implies that $f(n) \in O(g(n))$.*
2. *$c > 0$ implies that $f(n) \in \Omega(g(n))$.*
3. *$c = 0$ implies that $f(n) \in o(g(n))$.*
4. *$0 < c < \infty$ implies that $f(n) \in \Theta(g(n))$.*

5. $c = \infty$ *implies that* $f(n) \in \omega(g(n))$.

An algorithm is *good* or **polynomial-time** if the complexity is O(polynomial(length of input)). This polynomial must be of fixed degree, that is, its degree must be independent of the input length. So, for example, $O(n^{\log n})$ is not polynomial.

Example: K-cut
A graph $G = (V, E)$

- partition into k parts i.e. $V = \uplus_{i=1..k} V_i$
- the cost between two nodes i, j is $c_{i,j}$

The problem is to find

$$\min_{V_i \text{partition}} \sum_{P_1, P_2 = 1 \ldots k} \sum_{j \in V_{P_2}} \sum_{i \in V_{P_1}, i \neq j} c_{ij} \tag{9}$$

Or in words, to find the partitions that minimize the costs of the cuts (the edges between any two different partitions). The problem has the complexity of $O(n^{k^2})$. Unless k is fixed (for example, $2-cut$), this problem is *not* polynomial. In fact it is NP-hard and harder than max clique problem.

15.5 Caveats of complexity analysis

One should bear in mind a number of caveats concerning the use of complexity analysis.

1. **Ignores the size of the numbers.** The model presented is a poor one when dealing with very large numbers, as each operation is given unit cost, regardless of the size of the numbers involved. But multiplying two huge numbers, for instance, may require more effort than multiplying two small numbers.

2. **Is worst case analysis.** Complexity analysis does not say much about the average case. Traditionally, complexity analysis has been a pessimistic measure, concerned with worst-case behavior. The *simplex method* for linear programming is known to be exponential (in the worst case), while the *ellipsoid* algorithm is polynomial; but, for the ellipsoid method, the average case behavior and the worst case behavior are essentially the same, whereas the average case behavior of simplex is much better than it's worst case complexity, and in practice is preferred to the ellipsoid method.

   Similarly, Quicksort, which has $O(n^2)$ worst case complexity, is often chosen over other sorting algorithms with $O(n \log n)$ worst case complexity. This is because QuickSort has $O(n \log n)$ average case running time and, because the constants (that we ignore in the O notation) are smaller for QuickSort than for many other sorting algorithms, it is often preferred to algorithms with "better" worst-case complexity and "equivalent" average case complexity.

3. **Ignores constants.** We are concerned with the asymptotic behavior of an algorithm. But, because we ignore constants (as mentioned in QuickSort comments above), it may be that an algorithm with better complexity only begins to perform better for instances of inordinately large size.

   Indeed, this is the case for the $O(n^{2.375477})$ algorithm for matrix multiplication, that is *"...wildly impractical for any conceivable applications."* [3]

[3] See *Coppersmith D. and Winograd S.*, Matrix Multiplication via Arithmetic Progressions. *Journal of Symbolic Computation*, 1990 Mar, V9 N3:251-280.

4. **$O(n^{100})$ is polynomial.** An algorithm that is *polynomial* is considered to be "good". So an algorithm with $O(n^{100})$ complexity is considered good even though, for reasons already alluded to, it may be completely impractical.

Still, complexity analysis is in general a very useful tool in both determining the intrinsic "hardness" of a *problem* and measuring the quality of a particular *algorithm*.

16 Complexity classes and *NP*-completeness

In optimization problems, there are two interesting issues: one is *evaluation*, which is to find the optimal *value* of the objective function (evaluation problems); the other one is *search*, which is to find the optimal solution (optimization problems).

16.1 Search vs. Decision

Decision Problem - A problem to which there is a yes or no answer.
Example. SAT = {Does there exist an assignment of variables which satisfies the boolean function ϕ; where ϕ is a conjunction of a set of clauses, and each clause is a disjunction of some of the variables and/or their negations?}
Evaluation Problem - A problem to which the answer is the cost of the optimal solution.
Note that an evaluation problem can be solved by solving a auxiliary decision problems of the form "Is there a solution with value less than or equal to M?". Furthermore, using binary search, we only have to solve a polynomial number of auxiliary decision problems.
Optimization Problem - A problem to which the answer is an optimal solution.

Optimization problem and evaluation problem are equivalent. [4]
To illustrate, consider the Traveling Salesperson Problem (**TSP**). TSP is defined on an undirected graph, $G = (V, E)$, where each edge $(i, j) \in E$ has an associated distance c_{ij}.
TSP_OPT = { Find a tour (a cycle that visits each node exactly once) of total minimum distance. }
TSP_EVAL = { What is the total distance of the tour with total minimum distance in $G = (V, E)$? }
TSP_DEC = { Is there a tour in $G = (V, E)$ with total distance $\leq M$? }
Given an algorithm to solve **TSP_DEC**, we can solve **TSP_EVAL** as follows.

1. Find the upper bound and lower bound for the TSP optimal objective value. Let $C_{min} = \min_{(i,j)\in E} c_{ij}$, and $C_{max} = \max_{(i,j)\in E} c_{ij}$. Since a tour must contain exactly n edges, then an upper bound (lower bound) for the optimal objective value is $n \cdot C_{max}$, $(n \cdot C_{min})$. One upper
2. Find the optimal objective value by binary search in the range $[n \cdot C_{min}, n \cdot C_{max}]$. This binary search is done by calling the algorithm to solve **TSP_DEC** $O(\log_2 n(C_{max} - C_{min})$ times, which is a polynomial number of times.

In Assignemnt 2 you are to show that if you have an algorithm for **TSP_EVAL** then in polynomial time you can solve **TSP_OPT**.
An important problem for which the distinction between the decision problem and giving a solution to the decision problem – the *search problem* – is significant is primality. It was an open question (until Aug 2002) whether or not there exist polynomial-time algorithms for testing whether or not

[4] J.B. Orlin, A.P. Punnen, A.S. Schulz, Integer programming: Optimization and evaluation are equivalent, WADS 2009.

an integer is a prime number. [5] However, for the corresponding search problem, finding all factors of an integer, no similarly efficient algorithm is known.
Another example: A graph is called k-connected if one has to remove at least k vertices in order to make it disconnected. A theorem says that there exists a partition of the graph into k connected components of sizes $n_1, n_2 \ldots n_k$ such that $\sum_{i=1}^{k} n_k = n$ (the number of vertices), such that each component contains one of the vertices $v_1, v_2 \ldots v_k$. The optimization problem is finding a partition such that $\sum_i |n_i - \bar{n}|$ is minimal, for $\bar{n} = n/k$. No polynomial-time algorithm is known for $k \geq 4$. However, the corresponding decision problem is trivial, as the answer is always *Yes* once the graph has been verified (in polynomial time) to be k-connected. So is the evaluation problem: the answer is always the sum of the averages rounded up with a correction term, or the sum of the averages rounded down, with a correction term for the residue.
For the rest of the discussion, we shall refer to the decision version of a problem, unless stated otherwise.

16.2 The class *NP*

A very important class of decision problems is called *NP*, which stands for nondeterministic polynomial-time. It is an abstract class, not specifically tied to optimization.

Definition 16.1. *A decision problem is said to be in NPif for all "Yes" instances of it there exists a polynomial-length "certificate" that can be used to verify in polynomial time that the answer is indeed Yes.*

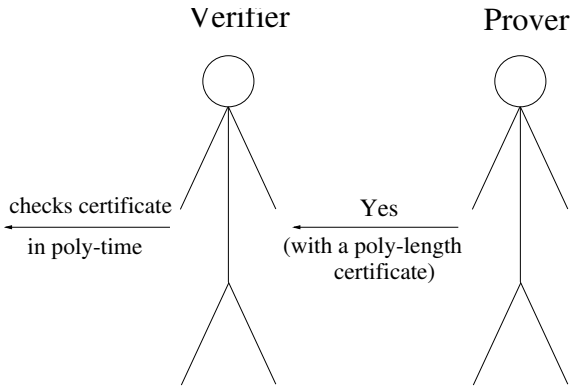


Figure 11: Recognizing Problems in *NP*

Imagine that you are a *verifier* and that you want to be able to confirm that the answer to a given decision problem is indeed "yes". Problems in *NP*have poly-length certificates that, without necessarily indicating how the answer was obtained, allow for this verification in polynomial time (see Figure 11).
To illustrate, consider again the decision version of TSP. That is, we want to know if there exists a tour with total distance $\leq M$. If the answer to our problem is "yes", the prover can provide us with

[5] Prior to 2002 there were reasonably fast superpolynomial-time algorithms, and also Miller-Rabin's randomized algorithm, which runs in polynomial time and tells either "I don't know" or "Not prime" with a certain probability. In August 2002 (here is the official release) "Prof. Manindra Agarwal and two of his students, Nitin Saxena and Neeraj Kayal (both BTech from CSE/IITK who have just joined as Ph.D. students), have discovered a polynomial time deterministic algorithm to test if an input number is prime or not. Lots of people over (literally!) centuries have been looking for a polynomial time test for primality, and this result is a major breakthrough, likened by some to the P-time solution to Linear Programming announced in the 70s."

such a tour and we can verify in polynomial time that 1) it is a valid tour and 2) its total distance is $\leq M$. However if the answer to our problem is "no", then (as far as we know) the only way to verify this would be to check every possible tour (this is certainly not a poly-time computation).
Lec5

17 Addendum on Branch and Bound

In general, the idea of Branch and Bound is implicit enumeration. Consequently, this can create serious problems for general IP.

Example 17.1. *Consider the following 0-1 Knapsack problem.*

$$\begin{aligned} \max \quad & 2x_1 + 2x_2 + \cdots + 2x_{100} \\ \text{subject to} \quad & 2x_1 + 2x_2 + \cdots + 2x_{100} \leq 101 \qquad (10) \\ & x_i \in \{0,1\} \qquad \text{for } i = 1, \ldots, 100 \qquad (11) \end{aligned}$$

One optimal solution to this problem is $x_1 = x_2 = \cdots = x_{50} = 1$ and $x_{51} = \cdots = x_{100} = 0$. In fact, setting any subset of 50 variables to 1 and the remaining to 0 yields an optimal solution.

For a 0-1 Knapsack problem of 100 variables, there are 2^{100} possible solutions. The naive way to find the optimal solution would to enumerate all of them, but this would take *way* too much time. Although Branch and Bound is an enumeration algorithm, the idea is to enumerate the solutions but discarding most of them before they are even considered.
Let us consider the LP-based Branch and Bound algorithm on the problem defined in Example 17.1. At the top node of the Branch and Bound tree, we don't make any assumptions about the bounds. The LP relaxation of the problem gives an objective value of 101, which is obtained by setting any subset of 50 variables to 1, another 49 variables to 0, and the remaining variable to $\frac{1}{2}$; 101 is now the upper bound on the optimal solution to the original problem. Without loss of generality, let the solution to the LP relaxation be

$$\begin{cases} x_1 = x_2 = \cdots = x_{50} = 1 \\ x_{51} = \dfrac{1}{2} \\ x_{52} = x_{53} = \cdots = x_{100} = 0. \end{cases}$$

Branching on x_{51} (the only fractional variable) gives no change in the objective value for either branch. In fact, using Branch and Bound we would have to fix 50 variables before we found a feasible integer solution. Thus, in order to find a feasible solution we would need to search $2^{51} - 1$ nodes in the Branch and Bound tree.
Although this is a pathological example where we have to do a lot of work before we get a lower bound or feasible solution, it illustrates the potential shortfalls of enumeration algorithms. Further, it is important to note that, in general, the rules for choosing which node to branch on are not well understood.

18 Complexity classes and *NP*-completeness

In optimization problems, there are two interesting issues: one is *evaluation*, which is to find the optimal *value* of the objective function (evaluation problems); the other one is *search*, which is to find the optimal solution (optimization problems).

18.1 Search vs. Decision

Decision Problem: A problem to which there is a yes or no answer.

Example 18.1. *SAT* = {*Does there exist an assignment of variables which satisfies the boolean function ϕ; where ϕ is a conjunction of a set of clauses, and each clause is a disjunction of some of the variables and/or their negations?*}

Evaluation Problem: A problem to which the answer is the cost of the optimal solution.

Note that an evaluation problem can be solved by solving auxiliary decision problems of the form "Is there a solution with value less than or equal to M?" Furthermore, using binary search we only have to solve a polynomial number of auxiliary decision problems.

Optimization Problem: A problem to which the answer is an optimal solution.

To illustrate, consider the Traveling Salesperson Problem (**TSP**). TSP is defined on an undirected graph, $G = (V, E)$, where each edge $(i, j) \in E$ has an associated distance c_{ij}.

TSP_OPT = { Find a tour (a cycle that visits each node exactly once) of total minimum distance. }

TSP_EVAL = { What is the total distance of the tour in $G = (V, E)$ with total minimum distance? }

TSP_DEC = { Is there a tour in $G = (V, E)$ with total distance $\leq M$? }

Given an algorithm to solve **TSP_DEC**, we can solve **TSP_EVAL** in polynomial time as follows.

1. Find the upper bound and lower bound for the TSP optimal objective value. Let $C_{min} = \min_{(i,j)\in E} c_{ij}$, and $C_{max} = \max_{(i,j)\in E} c_{ij}$. Since a tour must contain exactly n edges, then an upper bound (lower bound) for the optimal objective value is $n \cdot C_{max}$, $(n \cdot C_{min})$.
2. Find the optimal objective value by binary search in the range $[n \cdot C_{min}, n \cdot C_{max}]$. This binary search is done by calling the algorithm to solve **TSP_DEC** $O(\log_2 n(C_{max} - C_{min})$ times, which is a polynomial number of times.

In Assignment 2 you are to show that if you have an algorithm for **TSP_EVAL** then in polynomial time you can solve **TSP_OPT**.
Primality is an important problem for which the distinction between the decision problem and giving a solution to the decision problem – the *search problem* – is significant. It was an open question (until Aug 2002) whether or not there exists a polynomial-time algorithm for testing whether or not an integer is a prime number.[6] However, for the corresponding search problem to find all factors of an integer, no similarly efficient algorithm is known.

[6]Prior to 2002 there were reasonably fast superpolynomial-time algorithms, and also Miller-Rabin's randomized algorithm, which runs in polynomial time and tells either "I don't know" or "Not prime" with a certain probability. In August 2002 (here is the official release) "Prof. Manindra Agarwal and two of his students, Nitin Saxena and Neeraj Kayal (both BTech from CSE/IITK who have just joined as Ph.D. students), have discovered a polynomial time deterministic algorithm to test if an input number is prime or not. Lots of people over (literally!) centuries

Example 18.2. *A graph is called k-connected if one has to remove at least k vertices in order to make it disconnected. A theorem says that there exists a partition of the graph into k connected components of sizes $n_1, n_2 \ldots n_k$ such that $\sum_{i=1}^{k} n_k = n$ (the number of vertices), such that each component contains one of the vertices $v_1, v_2 \ldots v_k$. The optimization problem is finding a partition such that $\sum_i |n_i - \bar{n}|$ is minimal, for $\bar{n} = n/k$. No polynomial-time algorithm is known for $k \geq 4$. However, the corresponding recognition problem is trivial, as the answer is always Yes once the graph has been verified (in polynomial time) to be k-connected. So is the evaluation problem: the answer is always the sum of the averages rounded up with a correction term, or the sum of the averages rounded down, with a correction term for the residue.*

For the rest of the discussion, we shall refer to the decision version of a problem, unless stated otherwise.

18.2 The class *NP*

A very important class of decision problems is called *NP*, which stands for nondeterministic polynomial-time. It is an abstract class, not specifically tied to optimization.

Definition 18.3. *A decision problem is said to be in NPif for all "Yes" instances of it there exists a polynomial-length "certificate" that can be used to verify in polynomial time that the answer is indeed Yes.*

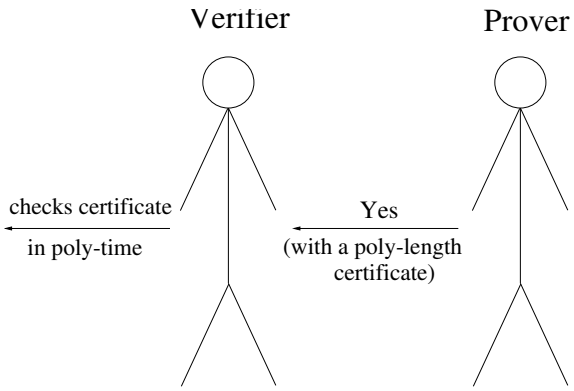


Figure 12: Recognizing Problems in *NP*

Imagine that you are a *verifier* and that you want to be able to confirm that the answer to a given decision problem is indeed "yes". Problems in *NP*have poly-length certificates that, without necessarily indicating how the answer was obtained, allow for this verification in polynomial time (see Figure 12).
To illustrate, consider again the decision version of TSP. That is, we want to know if there exists a tour with total distance $\leq M$. If the answer to our problem is "yes", the prover can provide us with such a tour expressed as a permutation of the nodes. The length of this tour is the number of cities, n, which is less than the length of the input. However, we still have to do some work to read the certificate: given a permutation of the nodes, we need to verify that the tour is valid (there are no subtours), and we need to sum the cost of the edges to verify that the total length of the tour is $\leq M$. This can be done in polynomial time. If the answer to our problem is "no," then (as far

have been looking for a polynomial time test for primality, and this result is a major breakthrough, likened by some to the P-time solution to Linear Programming announced in the 70s."

as we know) the only way to verify this would be to check every possible tour, which is certainly not a poly-time computation.

18.2.1 Some Problems in *NP*

Hamiltonian Cycle = { Does the incomplete, undirected graph $G = (V, E)$ contain a Hamiltonian cycle (a tour that visits each vertex exactly once)? }
Certificate: the tour. Observe that TSP is harder than Hamiltonian cycle: let the edges in E have a weight of 1, and let the missing edges in E have a weight of ∞. We can then ask whether the modified graph contains a tour of length $\leq n$.

Clique = { Is there a clique of size $\leq k$ in the graph $G = (V, E)$? }
Certificate: the clique; we must check that all pairwise edges are present.

Compositeness = { Is N composite (can N be written as $N = ab$, a and b are integers, s.t. $a, b > 1$)? }
Certificate: the factors a and b. (*Note 1:* $\log_2 N$ is the length of the input, length of certificate $\leq 2 \log_2 N$; and we can check in poly-time (multiply). *Note 2:* Giving only one of the factors is also OK.)

Primality = { Is the number N prime? }
Certificate: not so trivial. It was only in 1976 that Pratt showed that one exists in the form of a test. Interestingly, this problem was proved to be in *NP*26 years before it was known to be polynomial time solvable.

TSP = { Is there a tour in $G = (V, A)$ the total weight of which is $\leq m$?}
Certificate - the tour.

Dominating Set = { Given an incomplete, undirected graph $G = (V, E)$, is there a subset of the vertices $S \subset V$ such that every node in the graph is either in S or is a neighbor of S (i.e. S is a dominating set), and $|S| \leq k$? }
Certificate: the set of vertices, S. (A vertex is said to dominate itself and its neighbors. A dominating set dominates all vertices)

LP = { Is there a vector $\vec{x}$, $\vec{x} \geq 0$ and $A\vec{x} \leq b$ such that $C\vec{x} \leq k$? }
Certificate: a basic feasible solution x_B. (A general solution to the problem may be arbitrarily large. However, every basic feasible solution $\vec{x}$ can be shown to be poly-length in terms of the input. To see this, recall that any basic feasible solution x_B can be described in terms of its basis: $x_B = B^{-1}b = \left(\frac{[\text{cofactor matrix}]}{\det B}\right) b$. The cofactor matrix consists of sub-determinants, and we know that the determinant of A is a sum of products of terms. Let $a_{\max}$ be the element in A whose absolute value is largest. Then $\det B \leq (m!)\, a_{\max}^m \leq (m^m)\, a_{\max}^m$. Observe that $\log_2 (\det B) \leq m \log_2 m + m \log_2 a_{\max}$, which is of polynomial length. Therefore, both the denominator and the numerator are polynomial length, and we can present the solution to the LP in polynomial length. Clearly, it is also checkable in polynomial-time.)

IP = { Is there a vector $\vec{x}$, $\vec{x} \geq 0, \vec{x} \in Z$ and $A\vec{x} \leq b$ such that $C\vec{x} \leq k$? }
Certificate: a feasible vector $\vec{x}$. (just like LP, it can be certified in polynomial time; it is also necessary to prove that the certificate is only polynomial in length. See AMO, pg 795.)

k-center = { Given a complete graph $G = (V, E)$ with edge weights $w : V \times V \mapsto R^+$, is there a subset $S \subseteq V$, $|S| = k$, such that $\forall v \in V \setminus S$, $\exists s \in S$ such that $w(v, s) \leq M$? }
Certificate: the subset.

18.3 The class *co-NP*

Suppose the answer to the recognition problem is *No*. How would one certify this?

Definition 18.4. *A decision problem is said to be in co-NPif for all "No" instances of it there exists a polynomial-length "certificate" that can be used to verify in polynomial time that the answer is indeed No.*

18.3.1 Some Problems in *co-NP*

Primality = { Is N prime? }
"No" Certificate: the factors.

LP = { Is there a vector $\vec{x}$, $\vec{x} \geq 0$ and $A\vec{x} \leq b$ such that $C\vec{x} \leq k$? }
"No" Certificate: the feasible *dual* vector $\vec{y}$ such that $\vec{y} \leq 0$, $\vec{y}^T A \leq c$ and $\vec{y}^T b > k$ (from Weak Duality Thm.)

MST = { Is there a spanning tree with total weight $\leq M$? }
"No" Certificate: Although the prover gives us a no answer, we can just ignore her and solve the MST with a known polynomial-time algorithm and check whether the solution is less than M.

18.4 *NP*and *co-NP*

As a verifier, it is clearly nice to have certificates to confirm both "yes" *and* "no" answers; that is, for the problem to be in both *NP* and *co-NP*. From Sections 18.2.1 and 18.3.1, we know that the problems **Prime** and **LP** are in both *NP* and *co-NP*. However, for many problems in *NP*, such as **TSP**, there is no obvious polynomial "no" certificate.
Before we look at these 2 types of problems in detail, we define P, the class of polynomially solvable problems.

Definition 18.5. *Let Pbe the class of polynomial-time (as a function of the length of the input) solvable problems.*

Theorem 18.1. *If a decision problem is in Pthen it is also in NP.*

It is easy to see that $P \subseteq NP \cap co\text{-}NP$- just find an optimal solution. It is not known, however, whether, $P = NP \cap co\text{-}NP$; the problems that are in the intersection but for which no polynomial-time algorithm is known are usually very pathological. Primality $\in NP \cap co\text{-}NP$; this gave rise to the conjecture that primality testing can be carried out in polynomial time, as indeed it was recently verified to be.
Notice that the fact that a problem belongs to $NP \cap co\text{-}NP$ does not automatically suggest a polynomial time algorithm for the problem.
However, being in $NP \cap co\text{-}NP$ is usually considered as strong evidence that there *exists* a polynomial algorithm. Indeed, many problems were known to be in $NP \cap co\text{-}NP$ well before a polynomial algorithm was developed (ex. **LP**, **primality**); this membership is considered to be strong evidence that there was a polynomial algorithm out there, giving incentive for researchers to put their efforts

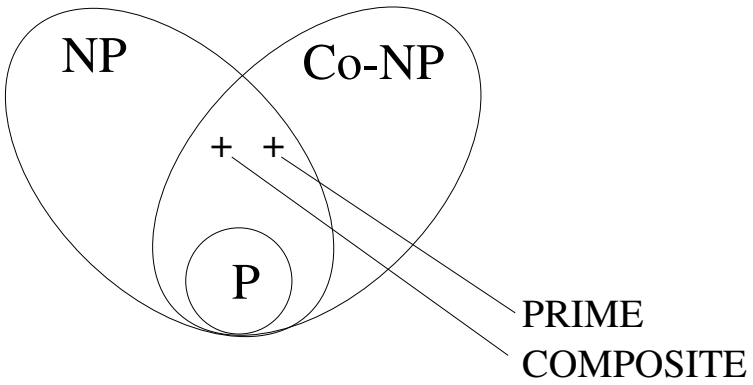


Figure 13: *NP* , *co-NP* , and *P*

into finding one. Both **primality** and **compositeness** lie in $\in NP \cap co\text{-}NP$; this gave rise to the conjecture that both can be solved in polynomial time.
On the flip-side, *NP*-Complete problems, which we turn to next, are considered to be immune to any guaranteed polynomial algorithm.

18.5 *NP*-completeness and reductions

A major open question in contemporary computer science is whether *P=NP*. It is currently believed that *P≠NP*.
As long as this conjecture remains unproven, instead of proving that a certain *NP*problem is not in *P*, we have to be satisfied with a slightly weaker statement: if the problem is in *P*then *P=NP*, that is, a problem is "at least as hard" as any other *NP*problem. Cook has proven this for a problem called *SAT*; for other problems, this can be proven using *reductions*.
In the rest of this section, we formally define *reducibility* and the complexity class *NP-complete*. Also, we provide some examples.

18.5.1 Reducibility

There are two definitions of reducibility, Karp and Turing; they are known to describe the same class of problems. The following definition uses Karp reducibility.

Definition 18.6. *A problem P_1 is said to reduce in polynomial time to problem P_2 (written as "$P_1 \propto P_2$") if there exists a polynomial-time algorithm A_1 for P_1 that makes calls to a subroutine solving P_2 and each call to a subroutine solving P_2 is counted as a single operation.*

We then say that P_2 is at least as hard as P_1. (Turing reductions allow for only *one* call to the subroutine.)
Important: In all of this course when we talk about reductions, we will always be referring to polynomial time reductions.

Theorem 18.7. *If $P_1 \propto P_2$ and $P_2 \in P$then $P_1 \in P$*

Proof. Let the algorithm A_1 be the algorithm defined by the $P_1 \propto P_2$ reduction. Let A_1 run in time $O(p_1(|I_1|))$ (again, counting each of the calls to the algorithm for P_2 as one operation), where $p_1()$ is some polynomial and $|I_1|$ is the size of an instance of P_1.
Let algorithm A_2 be the poly-time algorithm for problem P_2, and assume this algorithm runs in $O(p_2(|I_2|))$ time.

The proof relies on the following two observations:

1. The algorithm A_1 can call at most $O(p_1(|I_1|))$ times the algorithm A_2. This is true since each call counts as one operation, and we know that A_1 performs $O(p_1(|I_1|))$ operations.
2. Each time the algorithm A_1 calls the algorithm A_2, it gives it an instance of P_2 of size at most $O(p_1(|I_1|))$. This is true since each bit of the created P_2 instance is either a bit of the instance $|I_1|$, or to create this bit we used at least one operation (and recall that A_1 performs $O(p_1(|I_1|)$ operations).

We conclude that the resulting algorithm for solving P_1 (and now counting all operations) performs at most $O(p_1(|I_1|) + p_1(|I_1|) * p_2((p_1(|I_1|))))$ operations. Since the multiplication and composition of polynomials is still a polynomial, this is a polynomial time algorithm. □

Corollary 18.8. *If $P_1 \propto P_2$ and $P_1 \notin P$then $P_2 \notin P$*

18.5.2 *NP*-Completeness

Definition 18.9. *A problem $\mathcal{Q}$ is said to be NP-hard if $B \propto \mathcal{Q}\ \forall B \in NP$. That is, if all problems in NP are polynomially reducible to $\mathcal{Q}$.*

Definition 18.10.
A decision problem $\mathcal{Q}$ is said to be NP-Complete if:

1. *$\mathcal{Q} \in NP$, and*
2. *$\mathcal{Q}$ is NP-Hard.*

It follows from Theorem 18.7 that if *any NP*-complete problem were to have a polynomial algorithm, then all problems in *NP*would. Also it follows from Corollary 18.8 that is we prove that *any NP*-complete problem has no polynomial time algorithm, then this would prove that no *NP*-complete problem has a polynomial algorithm.

Note that when a decision problem is *NP*-Complete, it follows that its optimization problem is *NP*-Hard.

Conjecture: P $\neq$ *NP*. So, we do not expect any polynomial algorithm to exist for an *NP*-complete problem.

Now we will show specific examples of *NP*-complete problems.

SAT = { Given a boolean function in conjunctive normal[7] form (CNF), does it have a satisfying assignment of variable? }

$$(X_1 \vee X_3 \vee \bar{X_7}) \wedge (X_{15} \vee \bar{X_1} \vee \bar{X_3}) \wedge (\qquad\qquad) \ldots \wedge (\qquad\qquad)$$

Theorem 18.11 (Cook-Levin (1970))**.** *SAT is NP-Complete*

Proof.

[7]In boolean logic, a formula is in conjunctive normal form if it is a conjunction of clauses (i.e. clauses are "linked by and"), and each clause is a disjunction of literals (i.e. literals are "linked by or"; a literal is a variable or the negation of a variable).

SAT is in *NP***:** Given the boolean formula and the assignment to the variables, we can substitute the values of the variables in the formula, and verify in linear time that the assignment indeed satisfies the boolean formula.

$B \propto SAT\ \forall B \in NP$**:** The idea behind this part of the proof is the following: Let X be an instance of a problem $\mathcal{Q}$, where $\mathcal{Q} \in NP$. Then, there exists a SAT instance $F(X, \mathcal{Q})$, whose size is polynomially bounded in the size of X, such that $F(X, \mathcal{Q})$ is satisfiable if and only if X is a "yes" instance of $\mathcal{Q}$.

The proof, in very vague terms, goes as follows. Consider the process of verifying the correctness of a "yes" instance, and consider a Turing machine that formalizes this verification process. Now, one can construct a SAT instance (polynomial in the size of the input) mimicking the actions of the Turing machine, such that the SAT expression is satisfiable if and only if verification is successful.

□

Karp[8] pointed out the practical importance of Cook's theorem, via the concept of reducibility. To show that a problem $\mathcal{Q}$ is *NP*-complete, one need only demonstrate two things:

1. $\mathcal{Q} \in NP$, and
2. $\mathcal{Q}$ is *NP*-Hard. However, now to show this we only need to show that $\mathcal{Q}' \propto \mathcal{Q}$, for some *NP*-hard or *NP*-complete problem $\mathcal{Q}'$. (A lot easier than showing that $B \propto \mathcal{Q}\ \forall B \in NP$, right?)

Because all problems in *NP* are polynomially reducible to $\mathcal{Q}_1$, if $\mathcal{Q}_1 \propto \mathcal{Q}$, it follows that $\mathcal{Q}$ is at least as hard as $\mathcal{Q}_1$ and that all problems in *NP* are polynomially reducible to $\mathcal{Q}$.

Starting with SAT, Karp produced a series of problems that he showed to be *NP*-complete.[9] To illustrate how we prove *NP*-Completeness using reductions, and that, apparently very different problems can be reduced to each other, we will prove that the Independent Set problem is *NP*-Complete.

Independent Set (IS) = {Given a graph $G = (V, E)$, does it have a subset of nodes $S \subseteq V$ of size $|S| \geq k$ such that: for every pair of nodes in S there is no edge in E between them?}

Theorem 18.12. *Independent Set is NP-complete.*

Proof.

IS is in *NP***:** Given the graph $G = (V, E)$ and S, we can check in polynomial time that: 1) $|S| \geq k$, and 2) there is no edge between any two nodes in S.

IS is *NP***-Hard:** We prove this by showing that $SAT \propto IS$.

i) Transform input for IS from input for SAT in polynomial time

Suppose you are given an instance of SAT, with k clauses. Form a graph that has a *component* corresponding to each clause. For a given clause, the corresponding component is a complete graph with one vertex for each variable in the clause. Now, connect

[8]Reference: R. M. Karp. *Reducibility Among Combinatorial Problems*, pages 85–103. Complexity of Computer Computations. Plenum Press, New York, 1972. R. E. Miller and J. W. Thatcher (eds.).

[9]For a good discussion on the theory of *NP*-completeness, as well as an extensive summary of many known *NP*-complete problems, see: M.R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-completeness.* Freeman, San Francisco, 1979.

two nodes in different components if and only if they correspond to a variable and its negation. This is clearly a polynomial time reduction.

ii) Transform output for SAT from output for IS in polynomial time

If the graph has an independent set of size $\geq k$, then our the SAT formula is satisfiable.

iii) Prove the correctness of the above reduction

To prove the correctness we need to show that the resulting graph has an independent set of size at least k **if and only if** the *SAT* instance is satisfiable.

($\rightarrow$) [G has an independent set of size at least $k \rightarrow$ the *SAT* instance is satisfiable]

We create a satisfiable assignment by making true the literals corresponding to the nodes in the independent set; and we make false all other literals. That this assignment satisfies the SAT formula follows from the following observations:

1. The assignment is valid since the independent set may not contain a node corresponding to a variable and a node corresponding to the negation of this variable. (These nodes have an edge between them.)
2. Since 1) the independent set may not contain more than one node of each of the *components* of G, and 2) the value of the independent set is $\geq k$ (it is actually equal to k); then it follows that the independent set must contain exactly one node from each component. Thus every clause is satisfied.

($\leftarrow$) [the *SAT* instance is satisfiable $\rightarrow$ G has an independent set of size at least k]

We create an independent set of size equal to k by including in S one node from each clause. From each clause we include in S exactly one of the nodes that corresponds to a literal that makes this clause true. That we created an independent set of size k follows from the following observations:

1. The constructed set S is of size k since we took exactly one node "from each clause", and the SAT formula has k clauses.
2. The set S is an independent set since 1) we have exactly one node from each component (thus we can't have intra-component edges between any two nodes in S); and 2) the satisfying assignment for SAT is a valid assignment, so it cannot be that both a variable and its negation are true, (thus we can't have inter-component edges between any two nodes in S).

□

An example of the reduction is illustrated in Figure 14 for the 3-SAT expression $(x_1 \vee x_2 \vee x_3) \wedge (\bar{x_1} \vee \bar{x_2} \vee x_4) \wedge (x_1 \vee \bar{x_3} \vee x_4) \wedge (\bar{x_2} \vee x_3 \vee \bar{x_4})$.

Partition = { Given a set of n numbers $a_1, a_2, \ldots, a_n$ such that $\sum_{i=1}^{n} a_i = B$, is there a subset S of the numbers such that $\sum_{i \in S} a_i = \frac{B}{2}$? }

Theorem 18.13. *0-1 Knapsack is NP-complete.*

Proof. First of all, we know that it is in *NP*, as a list of items acts as the certificate and we can verify it in polynomial time.
Now, we show that *Partition* $\propto$ *0-1 Knapsack*. Consider an instance of *Partition*. Construct an instance of *0-1 Knapsack* with $w_i = c_i = b_i$ for $i = 1, 2, \ldots, n$, and $W = K = \frac{1}{2}\sum_{i=1}^{n} b_i$. Then, a solution exists to the *Partition* instance if and only if one exists to the constructed *0-1 Knapsack* instance.
Exercise: Try to find a reduction in the opposite direction. □

Theorem 18.14. *k-center is NP-complete.*

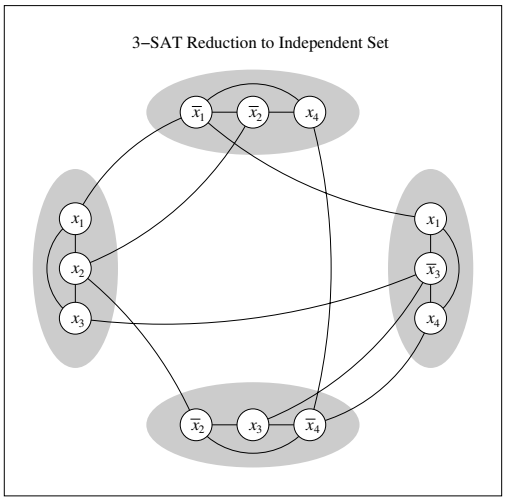


Figure 14: Example of 3-SAT reduction to Independent set

Proof. The *k-center* problem is in *NP* from Section 18.2. The *dominating set* problem is known to be *NP-complete*; therefore, so is *k-center* (from Section 18.5.1). □

This algorithm was given earlier in Fall 2008 **Additional remarks about the Knapsack problem:**
The problem can be solved by a longest path procedure on a graph – DAG– where for each $j \in \{1, \ldots, n\}$ and $b \in \{0, 1, \ldots, B\}$ we have a node. This algorithm can be viewed alternatively also as a dynamic programming with $f_j(b)$ the value of $\max \sum_{i=1}^{j} u_i x_i$ such that $\sum_{i=1}^{j} v_i x_i \leq b$. This is computed by the recursion:

$$f_j(b) = \max\{f_{j-1}(b); f_{j-1}(b - v_j) + u_j\}.$$

There are obvious boundary conditions,

$$f_1(b) = \begin{cases} 0 & \text{if } b \leq v_1 - 1 \\ u_1 & \text{if } b \geq v_1 \end{cases}$$

The complexity of this algorithm is $O(nB)$. So if the input is represented in *unary* then the input length is a polynomial function of n and b and this run time is considered polynomial. The knapsack problem, as well as any other *NP*-complete problem that has a poly time algorithm for unary input is called *weakly NP-complete*.

19 The Chinese Checkerboard Problem and a First Look at Cutting Planes

In this problem[10] we are given a standard Chinese checkerboard and three different types of diamond tiles (Figure 15). Each type of tile has a prescribed orientation, and each tile covers exactly four circles on the board. The Chinese checkerboard problem is a special case of the Set Packing problem and asks:

> *What is the maximum number of diamonds that can be packed on a Chinese checkerboard such that no two diamonds overlap or share a circle?*

19.1 Problem Setup

Let D be the collection of diamonds, and let O be a set containing all the pairs of diamonds that overlap; that is, $(d, d') \in O$ implies that diamonds d and d' have a circle in common. For every $d \in D$, define a decision variable x_d as follows.

$$x_d = \begin{cases} 1 & \text{if diamond } d \text{ is selected} \\ 0 & \text{if diamond } d \text{ is not selected} \end{cases}$$

To determine the total number of decision variables, we consider all possible placements for each type of diamond. Starting with the yellow diamond, we sweep each circle on the checkerboard and increment our count if a yellow diamond can be placed with its upper vertex at that circle. Figure 16 illustrates this process; a yellow diamond can be "hung" from every black circle in the image, and there are 88 total black circles. Using the same procedure, we find that there are 88 legal placements for green diamonds and another 88 for yellow diamonds. This brings the total number of decision variables to 264.

19.2 The First Integer Programming Formulation

We can formulate the problem as the following integer program, which is also known as the set packing problem.

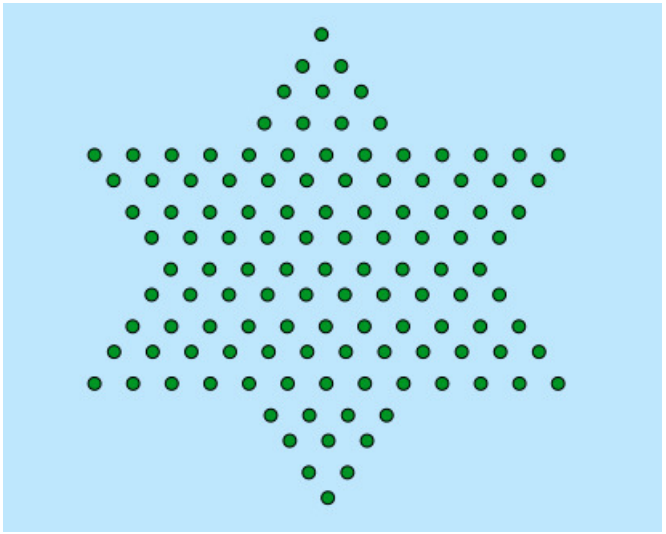
$$\begin{aligned} \max \quad & \sum_{d \in D} x_d \\ \text{subject to} \quad & x_d + x_{d'} \leq 1 \qquad \text{for } (d, d') \in O \\ & x_d \in \{0, 1\} \qquad \text{for } d \in D \end{aligned}$$

The optimal objective value for this IP is 27, but the optimal objective for the LP relaxation is 132. This is a huge difference!

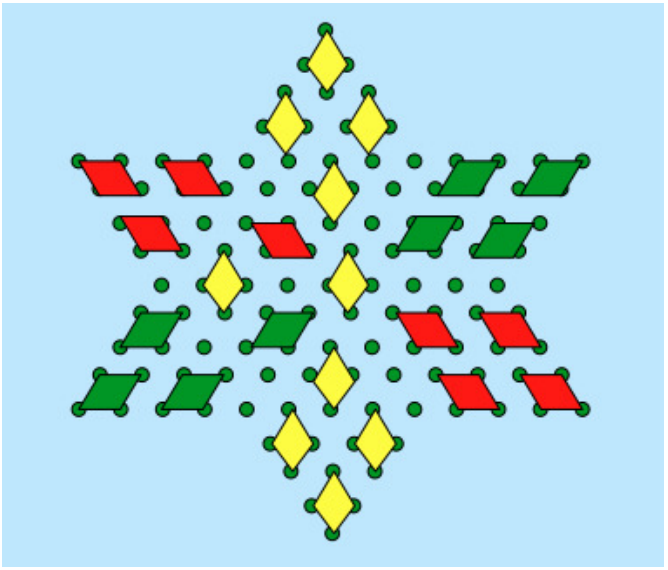
19.3 An Improved ILP Formulation

As it turns out, we can do much better by reformulating the constraints to make them tighter. We begin by observing that for many circles on the board, there are 12 different diamonds competing for that space. (See Figure 17.)

[10] Thanks to Professor Jim Orlin for the use of his slides and to Professor Andreas Schulz for this example.



(a) Empty



(b) Packed

Figure 15: (a) Image of empty Chinese checkerboard. (b) Example packing of the checkerboard using the three types of diamond tiles.

Formally, we say that S is the set of diamonds overlapping at a given circle. So if $S = \{x_1, x_2, x_3, x_4\}$ is an overlapping set, it would correspond to the following constraints in our initial IP formulation:

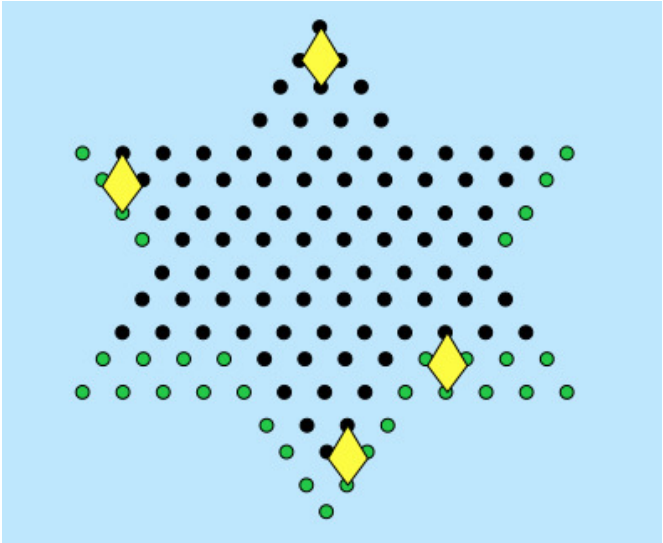


Figure 16: Enumerating the number of possible yellow diamonds; black circles show all legal positions from which a yellow diamond can be "hung."

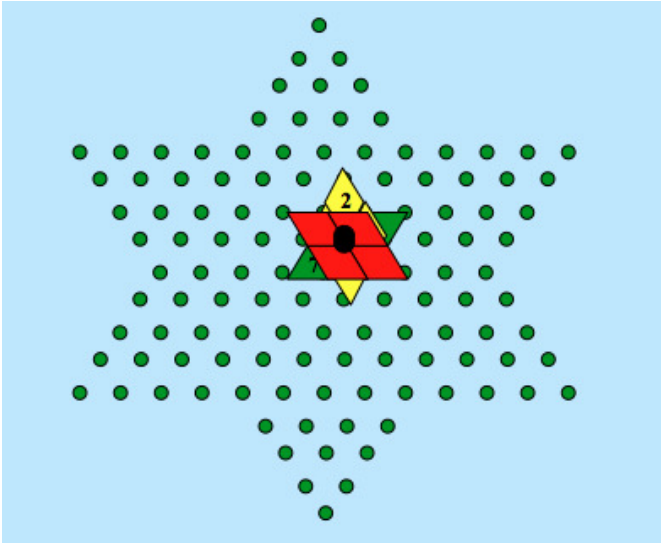


Figure 17: Example point where 12 different diamonds can fit.

$$
\begin{aligned}
x_1 + x_2 &\leq 1 \\
x_1 + x_3 &\leq 1 \\
x_1 + x_4 &\leq 1 \\
x_2 + x_3 &\leq 1 \\
x_2 + x_4 &\leq 1 \\
x_3 + x_4 &\leq 1 \\
x_1, x_2, x_3, x_4 &\in \{0, 1\}
\end{aligned}
$$

Although this is an accurate formulation, from Figure 17 we observe that we can replace the pairwise overlapping constraints with

$$x_1 + x_2 + x_3 + x_4 \leq 1 \tag{12}$$

which is much tighter. (Any solution satisfying this constraint is feasible for the original constraints, but the converse is not true.) Hence, the following is a much stronger IP formulation. Let S_i be the set of *all* diamonds that cover circle i.

$$\begin{aligned} \max \quad & \sum_{d \in D} x_d & \\ \text{subject to} \quad & \sum_{d \in S_i} x_d \leq 1 & \text{for every circle } i \\ & x_d \in \{0,1\} & \text{for } d \in D \end{aligned}$$

The optimal objective value for this IP is 27 (identical to the original formulation), but the optimal objective value for the LP relaxation is 27.5; clearly, this is a significantly better bound.
Lec6

20 Cutting Planes

Recall that the principal approach in solving an integer program is to obtain the resulting linear programming relaxation. Cutting planes are constraints which can be added to the LP relaxation without excluding any integer feasible solutions. Cutting planes are extremely useful because they reduce the gap between the LP relaxation and the integer optimal solution. The goals of this lecture are to illustrate how valuable it can be to obtain tighter LPs, and illustrate various methods for obtaining better bounds in problems such as set packing, capital budgeting (the knapsack problem), the traveling salesman problem and general integer programs. We also describe how cutting planes can be utilized to accelerate branch and bound algorithms.

20.1 Chinese checkers

In the Chinese checker board problem which we introduced in the previous lecture we seek the maximum number of diamonds that can be placed in the checker board. The diamonds are not permitted to overlap, or even to share a single circle on the board. To provide an integer programming formulation of the problem, we number diamonds and we describe the set of diamonds as D. The binary variable x_d indicates whether diamond d is included in the checker board. We then formulate the problem

$$\begin{aligned} & \max \textstyle\sum_{d \in D} x_d & \\ & \text{s.t.} & \\ & x_d + x_{d'} \leq 1 & \forall (d, d') \in O \\ & 0 \leq x_d \leq 1 & x_d \in \mathbb{Z} \end{aligned}$$

where O represents the pairs of diamonds that overlap on the checker board. There are 268 diamonds in total, which is also the number of our decision variables.
This problem is a special case of the set packing problem, particularly this is the independent set problem. The independent set problem is to find the maximum number of nodes in a graph such

that no two of which are adjacent. The above problem fits this interpretation since the set O can be thought of as the edges of a graph and D as the nodes of the graph.
The set packing problem is formulated as follows: there is a universal set $J = \{1, ..., n\}$ and subsets $S_i \subset J, i = 1, ..., m$. The task is to find a collection of subsets $\mathcal{C} = \{S_i\}_{i \in \{1,...,m\}}$, such that $S_{i_1} \cap S_{i_2} = \emptyset, S_{i_1}, S_{i_2} \in \mathcal{C}$. That is, find a maximum number of subsets such that no two of these subsets have an item in common.
The optimization version of the problem seeks the largest number of subsets such that the aforementioned condition is true, and there is also weighted version of the problem. Notice that our decision variables in the optimization version of the problem are the subsets S_i which we wish to include.
To see that independent set is a set packing problem consider a graph $G = (V, E)$ for which we want to determine an independent set. The decision variables in the optimization version of the independent set problem are the vertices which we wish to include in the independent set. Let us consider a set packing problem in which the universal set consists of all the edges of the graph G, i.e. the universal set is the set E. For each vertex $i \in V$ of the graph, let us form a subset S_i which consists of all edges that are adjacent to node i, i.e. let $S_i = \{[i, j] | [i, j] \in E\}$ for all $i \in V$. As we mentioned before, the decision variables in the set packing problem are the subsets which we want to include, and since each set corresponds to a vertex of the graph of the independent set problem, each choice of subset S_i in the set packing problem corresponds to a choice of vertex i for the independent set problem. We are seeking to choose the maximum number of subsets S_i such that no two overlap, i.e. the maximum number of vertices such that no two share any edges, which is the independent set problem.
For the formulation of the set packing optimization problem we define a binary variable x_i which indicates if set S_i is selected. For each subset S_i we also define J_i, the collection of subsets $S_j, j \neq i$, that S_i shares an element with. That is, if S_i is chosen then none of the other subsets $S_j \in J_i$, $j \neq i$, can be selected. We get the following problem:

$$\begin{aligned} &\max \textstyle\sum_{i=1}^{m} x_i \\ &\text{s.t.} \\ &\sum_{i \in J_i} x_i \leq 1 \qquad i = 1, ..., m \\ &x_i \in \{0, 1\} \end{aligned}$$

Set packing appears in various contexts. Examples include putting items in storage (packing a truck), manufacturing (number of car doors in a metal sheet, rolls of paper that can be cut from a giant roll), the fire station problem and the forestry problem.

20.2 Branch and Cut

As we mentioned in the introduction of this section, cutting planes are useful in getting better bounds to an integer problem. This is especially useful for branch and bound algorithms since more nodes of the branch and bound tree can be fathomed, and so that the fathoming can be done earlier in the process. Recall that we can eliminate node j of the tree if the LP bound for node j is no more than the value of the incumbent. Hence, the lower the bound of a node that we can produce the better, and cutting planes are useful in lowering these bounds.
For cutting planes, as for any other known technique for solving general IPs, we can come up with an instance of a problem which takes an exponential amount of time to solve. The tradeoff of using cutting planes is that although we can get better bounds at each node of the branch and bound tree, we end up spending more time in each node of the tree.

To reiterate, there may be different ways of formulating an IP. Each way gives the same IP, however the resulting LPs may be very different and some LPs have different bounds so the choice of formulation is critical.

21 The geometry of Integer Programs and Linear Programs

Given an integer program, we often desire to relax the problem to an LP and add a separating hyperplane (a constraint) which isolates the optimal solution of the LP without removing any of the integer feasible solutions from the feasible region. If we are able to add the right inequalities to the problem, then any corner of the resulting polyhedron becomes integer and the LP relaxation will yield an optimal integer solution. These 'right inequalities' are called facets, and are defined as cutting planes which cannot be tightened any more. Although this strategy is correct in principle, it cannot be applied in a practical setting for two reasons. Firstly, it has been shown that the problem of finding violated facets is an NP-hard problem. Moreover, even if we could find facets in polynomial time, the number of facets that are required to contain the feasible region are frequently exponential. Nevertheless, valid inequalities are of great utility for branch and bound and can significantly reduce the upper bounds of the tree and speed up computations.
The pure cutting plane approach adds cutting planes iteratively, sustaining a single LP instead of splitting the feasible region, however valid inequalities can be used for the same purpose in branch and bound.
There are two approaches for generating cutting planes. Problem specific cutting planes have been developed for problems with special structure such as set packing, the knapsack problem and TSP. On the other hand, there is an LP-based approach, that works for general integer programs named Gomory cutting planes.

21.1 Cutting planes for the Knapsack problem

We demonstrate an example of finding cutting planes for the knapsack problem. Consider the following LP relaxation of an instance of the knapsack problem:

$$\begin{aligned}
&\max 16x_1 + 22x_2 + 12x_3 + 8x_4 + 11x_5 + 19x_6 \\
&\text{s.t.} \\
&5x_1 + 7x_2 + 4x_3 + 3x_4 + 4x_5 + 6x_6 \leq 14 \\
&0 \leq x_j \leq 1
\end{aligned}$$

The optimal solution is (1,3/7,0,0,0,1) with an objective value of 44 3/7. The trick that we employ is to identify covers, i.e. subsets such that the sum of the weights in the subset exceeds the budget. For example, the constraint $x_1 + x_2 + x_3 \leq 2$ is a cover constraint. Cover constraints are cutting planes, i.e. they do not eliminate any IP solutions, but they cut off fractional LP solutions. In general, for each cover S we obtain the constraint $\sum_{j \in S} x_j \leq |S| - 1$.
Our strategy will be to iterate between solving the resulting LP relaxation and finding violated cover constraints in the resulting solution in order to tighten the problem. For example, $\{1, 2, 6\}$ is a violated cover from the optimal solution which we presented. Solving the problem with this added cover constraint yields an optimal solution of $(0, 1, 1/4, 0, 0, 1)$ and $\{2, 3, 6\}$ becomes a violated constraint. The resulting optimal solution after adding this new cover constraint becomes (1/3,1,1/3,0,0,1), and after adding the corresponding cover constraint for the violated cover $\{1, 2, 3, 6\}$ we obtain the solution (0,1,0,0,1/4,1) with an objective value of 43 3/4, which bounds the optimal integer value below 43, which is in fact the optimal objective value.

Notice that we required 3 cuts. Had we been smarter it would have taken 1 cut, the last one which we presented. Nevertheless, we had a simple approach for finding cuts. This does not find all of the cuts, but is practical. Recall, it took 25 nodes of a branch and bound tree to solve the same problem. In fact, researchers have found cutting plane techniques to be quite useful to solve large integer programs (usually as a way of getting better bounds.)

21.2 Cutting plane approach for the TSP

Moving on to TSP, this is a very well studied problem. It is often the problem for testing out new algorithmic ideas. The reason is that although the problem is NP-complete (it is intrinsically difficult in some technical sense), large instances have been solved optimally (5000 cities and larger). Very large instances have been solved approximately (10 million cities to within a couple of percent of optimum). We will formulate the problem by adding constraints that look like cuts.
In the following formulation, x_e is a binary variable which denotes whether edge e belongs in the optimal tour and $A(i)$ are the arcs which are incident to node i. Consider the problem

$$\begin{array}{l}\min \sum_e c_e x_e \\ \text{s.t.} \\ \sum_{e \in A(i)} x_e = 2 \\ x_e \in \{0,1\}\end{array}$$

This problem is easy to solve since its linear relaxation will yield an optimal integer solution, at the expense of ignoring the subtour elimination constraints. By noting the fact that any integer solution with exactly two arcs incident to every node is the union of cycles, we can derive an improved formulation: for each possible subtour, add a constraint that makes the subtour infeasible for the IP. These are called subtour breaking constraints. For any proper subset S of the graph nodes, a subtour breaking constraint can be formulated as

$$\sum_{i \in S, j \in S} x_{ij} \leq |S| - 1$$

The reason that this formulation will work is that a subtour that includes all nodes of S has $|S|$ arcs, whereas a tour for the entire network has at most $|S| - 1$ arcs with two endpoints in S, thus ensuring that the set S will not have a subtour going through all its nodes.
Producing these constraints for all possible subtours will result in exponentially many constraints, too many to include in an IP or an LP. In practice it is worth including only some of the constraints, solving the resulting LP and if a subtour appears as part of the LP solution, then add a new subtour elimination constraint to the LP, and iterate the process.
The IP formulation which we presented is quite successful in practice, as it usually achieves an LP bound within 1% to 2% from the optimal TSP tour length. It has also been observed in practice that adding even more complex constraints yields better LP formulations.

22 Gomory cuts

Gomory cuts is a method for generating cuts using the simplex tableau. Consider the following example:

$$\begin{array}{llll} x_1 & x_2 & x_3 & x_4 \\ 1+3/5 & 4+1/5 & 3 & 2/5 \end{array} \quad = 9+4/5$$

One can always bring the LP to this standard form. Now notice that if we bring the integers of the equality on the right hand side, then by the fact that the variables are nonnegative it must be the left hand side below is nonnegative:

$$\tfrac{3}{5}x_1 + \tfrac{1}{5}x_2 + \tfrac{2}{5}x_4 = \text{integer} + \tfrac{4}{5}$$

Therefore the right hand side has to be nonnegative as well, and this implies that the integer part has to be at least zero. Therefore the right hand side has to be greater than or equal to its fractional component, which results in the following inequality:

$$\tfrac{3}{5}x_1 + \tfrac{1}{5}x_2 + \tfrac{2}{5}x_4 \geq \tfrac{4}{5}$$
$$3x_1 + x_2 + 2x_4 \geq 4$$

Observe that in order to obtain the Gomory cut we relied on there being a single constraint with a fractional right hand side, all coefficients being nonnegative, and all variables being integral. Nevertheless the method also works if the coefficients are negative. This is true because we can rewrite the equality such that the coefficients of the fractional parts are nonnegative. Consider the following example:

$$\begin{array}{llll} x_1 & x_2 & x_3 \quad x_4 & \\ 1+3/5 & -4+1/5 & -3 \quad -2/5 & = -1-1/5 \\ 1+3/5 & -5+2/5 & -3 \quad -1+3/5 & = -2+4/5 \end{array}$$

which leads to the following cut:

$$\tfrac{3}{5}x_1 + \tfrac{2}{5}x_2 + \tfrac{3}{5}x_4 \geq \tfrac{4}{5}$$
$$3x_1 + 2x_2 + 3x_4 \geq 4$$

In general, let $fr(a)$ be the positive fractional part of a, then we substitute $fr(a) = a - \lfloor a \rfloor$.
The steps for generating Gomory cuts in general are as follows: after pivoting, find a basic variable that is fractional. Write a Gomory cut. This is an inequality constraint, leading to a new slack variable. Note that the only coefficients that are fractional correspond to non-basic variables. It is also true that the Gomory cut makes the previous basic feasible solution infeasible. Resolve the LP with the new constraint, and iterate.

23 Generating a Feasible Solution for TSP

As we mentioned previously, branch and bound requires generating an integer feasible solution to the problem in order to provide a lower bound apart from the upper bound which is generated by the LP relaxation (for maximization problems). In general this may be hard. We present a polynomial time algorithm for generating a feasible solution for TSP. The solution is guaranteed to be within a factor of 2 from optimal, as long as the triangle inequality is satisfies, which we will assume is true for the rest of our discussion.
Consider a complete graph $G = (V, E)$ for which the triangle inequality holds, i.e. $\forall i, j, k \in V$ we have $c_{ij} + c_{jk} \geq c_{ik}$. Selecting the nearest neighbor as the next city to visit can be a very bad approach (it can do as bad as $\log nOPT$ where n is the number of cities). The following approximation algorithm relies on the minimum spanning tree. Define an Eulerian graph as a graph for which $\forall j \in V$ deg_j is even. An Eulerian tour is a tour that traverses each edge exactly once. Take the MST of the graph in question (which we can do in polynomial time) and make it in to an Eulerian graph by doubling each edge. In the resulting graph we have an Eulerian tour which

we can make into a tour by creating shortcuts. We do this by jumping over nodes which appear in the list of edges which describe the Eulerian tour. The length of the tour we get must be within a factor of 2 of $|MST|$ due to the triangle inequality. On the other hand, $TSP^{\text{opt}} \geq |MST|$ because by removing one edge from the tour we get an MST which has to be at least as long as MST. Therefore we have developed an algorithm for generating a tour within a factor of 2 of TSP^{opt}. There is also a polynomial time algorithm for generating a tour within a factor of 1.5 from optimal, which we do not present here.

24 Diophantine Equations

We will now discuss the problem of determining whether the following system of equations:

$$Ax = b \quad x \in \mathbb{Z}^n$$

has a feasible solution, where we assume that the coefficients are integral. Consider the following equality:

$$a_1x_1 + ... + a_nx_n = b \quad x \in \mathbb{Z}^n \quad (D)$$

For $n = 1$ the existence of a feasible solution is equivalent to the condition that $\frac{b}{a_1} \in \mathbb{Z}^n$. For $n = 2$ the idea is to find the greatest common divisor of a_1, a_2, which will function as the step size in our system. The Euclidean algorithm is a polynomial time algorithm for calculating the gcd, which relies on the following fact: for $a \geq b > 0$, $a, b \in \mathbb{Z}^+$ we have $gcd(a, b) = gcd(b, r)$. Stop at $gcd(a, 0) = a$ and output a as the result. Since r is at most half of a we require $\log a$ operations to complete the algorithm which makes the algorithm linear in the size of the input when the input is provided in binary form.
There is a useful extension of the Euclidean algorithm which keeps track of additional intermediate quantities in order to output r, p, q such that $gcd(a, b) = r = pa + qb$ where p and q are relatively prime and integer. The algorithm is presented below and again, without loss of generality, we assume that $a \geq b$.

Intitialize
$r_{-1} = a \qquad r_0 = b$
$p_{-1} = 1 \qquad p_0 = 0$
$q_{-1} = 0 \qquad q_0 = 1$
$t = 0$
While $r_t \neq 0$
$t \leftarrow t + 1$
$d_t \leftarrow \left\lfloor \frac{r_{t-2}}{r_{t-1}} \right\rfloor$
$r_t \leftarrow r_{t-2} - d_t r_{t-1}$
$p_t \leftarrow p_{t-2} + d_t p_{t-1}$
$q_t \leftarrow q_{t-2} + d_t q_{t-1}$
Loop
Return $r = r_{t-1} \qquad p = (-1)^{t+1} p_t \qquad q = (-1)^t q_t$

Returning to $a_1x_1 + a_2x_2 = b$, we will use the extended Euclidean algorithm which has the same run time as the Euclidean algorithm but in addition generates $p, q \in \mathbb{Z}$ such that $gcd(a, b) = r = pa + qb$ where p, q are relatively prime. Consider the example of $10x_1 + 3x_2 = 7$. We have $gcd(10, 3) =$

$r = 10p + 3q$ where $r = 1, p = 1, q = -3$. Substitute the gcd for both a_1 and a_2 and obtain $10x_1 + 3x_2 = ry_1 = 7$, which implies $y_1 = 7$. We can now backtrack y_1 to obtain the desired feasible solution: $x_1 = py_1 = 7$, $x_2 = qy_1 = -21$.
The same logic applies to any diophantine equation. From equation (D) above, we have $r = gcd(a_1, a_2) = pa_1 + qa_2$, from which we get a new equation:

$$ry + a_3x_3... + a_nx_n = b \quad y \in \mathbb{Z}, x \in \mathbb{Z}^{n-2} \quad (D')$$

It can now be seen that (D) has a solution iff (D') has a solution. One direction is obvious. To see the other direction, note that if $(\bar{y}, \bar{x}_3, ..., \bar{x}_n)$ is a solution for (D') then $(p\bar{y}, q\bar{y}, \bar{x}_3, ..., \bar{x}_n)$ is a solution for (D). Therefore, the existence of a solution for an equation in $\mathbb{Z}^n$ can be reduced to checking if there is a solution to an equation for $\mathbb{Z}^2$.

24.1 Hermite Normal Form

Definition 24.1. *An integral matrix of full rank is said to be in Hermite normal form if it has the form $(H|0)$ where H is non-singular, lower triangular, in which $|h_{ij}| \leq h_{ii}$ for $1 \leq j < i \leq n$.*

Theorem 24.1. *If A is an $m \times n$ integer matrix with $rank(A) = m$ then there exists an $n \times n$ unimodular matrix C such that*

1. *$AC = [H|0]$ and H is Hermite normal form*
2. *there exists $x \in \mathbb{Z}^n$ such that $Ax = b$ iff $H^{-1}b \in \mathbb{Z}^n$.*

Proof. Elementary unimodular column operations include the following operations:

1. Exchange columns
2. Multiply a column by -1
3. Add integer multiples of one column to another

All of these operations can be represented by a unimodular matrix, that is the matrix has a determinant of ± 1. Thus the matrix representing multiple elementary unimodular column operations also has determinant of ± 1.
Part (1) can be proved by construction, according to the following algorithm, which converts an $m \times n$ matrix A with integral coefficients and full row rank into the form$[H|0]$ using only elementary unimodular column operations, where H is an $m \times m$ matrix in Hermite normal form. We first introduce the following operations:

$A' =$ **Operation1**(A, i, j):
let $r := gcd(a_{i,i}, a_{i,j})$, $\quad pa_{i,i} + qa_{i,j} = r$
$a'_i = pa_i + qa_j$
$a'_j = \frac{a_{i,i}}{r}a_j + \frac{a_{i,j}}{r}a_i$
return $A' = [a_1, a_2, ..., a_{i-1}, a'_i, a_{i+1}, ..., a'_j, ..., a_n]$

$A' =$ **Operation2**(A, i):
if $a_{i,i} < 0$, then $a'_i = -a_i$
return $A' = [a_1, a_2, ..., a_{i-1}, a'_i, a_{i+1}, ..., a_n]$

$A' =$ **Operation3**(A, i, j):
$a'_j = a_j - \left\lceil \frac{a_{i,j}}{a_{i,i}} \right\rceil a_i$
return $A' = [a_1, a_2, ..., a_{j-1}, a'_j, a_{j+1}, ..., a_n]$

The following is the Hermite normal form algorithm:
$H =$ **Hermite**(A)
For row $i = 1, ..., m$
For column $j = i + 1, ..., n$
A=**Operation1**(A, i, j)
Loop
A=**Operation2**(A,i)
For column $j = 1, ..., i - 1$
A=**Operation3**(A, i, j)
Loop
Loop
Return $H = A$

Notice that the Hermitian matrix is constructed with a polynomial number of operations. Consider, for example:

$$A = \begin{bmatrix} 2 & 6 & 1 \\ 4 & 7 & 7 \end{bmatrix}, b = \begin{bmatrix} 7 \\ 4 \end{bmatrix}$$

Implement the algorithm in the handout to finally obtain

$$AC = \begin{bmatrix} 1 & 0 & 0 \\ -3 & 5 & 0 \end{bmatrix} = [H|0], C = \begin{bmatrix} -6 & 3 & 7 \\ 2 & -1 & -2 \\ 1 & 0 & -2 \end{bmatrix}.$$

By then taking

$$H^{-1} = \begin{bmatrix} 1 & 0 \\ 3/5 & 1/5 \end{bmatrix}$$

we get $\bar{x} = C \begin{bmatrix} \frac{H^{-1}b}{0} \end{bmatrix}$ as a possible solution.
We will now prove part (2) of the previous theorem. In general, asking whether the Diophantine equation has a feasible solution is equivalent to asking if $b \in L(A)$, where the lattice of an $m \times n$ matrix A is defined as $L(A) := \{y \in \mathbb{R}^m : y = Ax, x \in \mathbb{Z}^n\}$. The following lemma is true:

Lemma 24.1. *If A is $m \times n$, C is $n \times n$ unimodular, then $L(A) = L(AC)$.*

Proof. Let $x = Cw$. Then

$$\begin{aligned} L(A) &= \{y \in \mathbb{R}^m : y = ACw, Cw \in \mathbb{Z}^n\} \\ &= \{y \in \mathbb{R}^m : y = ACw, w \in \mathbb{Z}^n\} \\ &= L(AC) \end{aligned}$$

where the second equality follows from the fact that C is unimodular. □ □

Therefore $Ax = b$ has in integer solution iff $ACw = b$ has an integer solution. This is equivalent to $[H|0]w = b$ having an integer solution, which implies that $H^{-1}b$ is integer iff $Ax = b$ has an integer solution, proving part (2). □

Since $w = H^{-1}b$, solves $ACw = b$, then a solution to $Ax = b$ is $\bar{x} = C\begin{bmatrix} H^{-1}b \\ 0 \end{bmatrix}$.
Lec7

25 Optimization with linear set of equality constraints

When the constraints are a set of linear equations over real variables (and not inequalities), optimization is independent of the cost function[11]. There can be following three cases:

1. *optimal cost is unbounded:* If the system of linear equations is under-constrained.
2. *no feasible solution:* If the system of linear equations is over-constrained and no feasible solution exists.
3. *exactly one solution:* If the system of linear equation has a unique solution.

26 Balas's additive algorithm

Balas's additive algorithm is a Branch and Bound algorithm that does not use LP. It considers binary ILP in the following standard form.

$$\begin{aligned} \min \quad & \sum_{j=1}^{n} c_j x_j \\ \text{subject to} \quad & \sum_{j=1}^{n} a_{ij} x_j \geq b_i \qquad \text{for } i = 1, \ldots, m \\ & x_j \in \{0,1\} \qquad \text{for } j = 1, \ldots, n \\ & 0 \leq c_1 \leq c_2 \leq \cdots \leq c_n \end{aligned}$$

All the coefficients c_i are non-negative and in increasing order.

Theorem 26.1. *Any ILP can be reduced to the standard form above.*

Proof. We give a proof by construction. The following series of transformations can reduce an ILP into the standard form.

1. *Integer variables to binary variables:* Any integer variable x_j can be coded in binary using $\lfloor \log u_j \rfloor + 1$ binary variables[12], where $0 \leq x_j \leq u_j$. Define $\lfloor \log u_j \rfloor + 1$ new variables, such that
$$x_j = \sum_{k=0}^{\lfloor \log u_j \rfloor} 2^k x_{kj}$$
If the ILP has a finite optimal feasible solution, we can compute the upper bound of each component (u_j) as a polynomial function of the input (Theorem 13.5 in [21]).

[11] If the cost function is $\sum_i a_i x_i$ and the linear equation is $\sum_i a_i x_i = d$, then the optimal cost is d
[12] $0 \leq u_j \leq 2^n - 1$ can be expressed using n bits. So, $n \geq \log(u_j + 1)$

2. *Optimization Objective:* If the optimization objective is to maximize some linear expression $c_j x_j$, that is,

$$OPT = \max \sum_{j=1}^{n} c_j x_j$$

we rewrite it as a minimization problem by defining the objective

$$OPT' = \min \ (-\sum_{j=1}^{n} c_j x_j)$$

The optimum for the maximization problem can be obtained from the computed optimum for the minimization problem as $OPT = -OPT'$. Hence, we can reduce any optimization problem with maximization objective to one with minimization objective.

3. *All constraints as $\geq$ constraints:*

   - if there is an equality, we replace it with two constraints $\leq$ and $\geq$.
   - if the inequality is $\leq$, multiply the inequality by -1 to get it in the desired $\geq$ form.
   - Write all constraints as $\geq$ constraints with the RHS of the inequalities being just the constant term.

4. *Positive costs:* All coefficients in the objective function must be positive. So, if a variable x has a negative coefficient in the objective function, then we must make a 'change of variable', say to a (binary) variable y, such that $y = 1 - x$. We need to change all occurrences of the variable x with $1 - y$ in our problem formulation (this includes both: the objective function and the constraints).

5. *Ordered costs:* Change the indices of the variables so that the indices are ordered according to monotone non-decreasing coefficients. Ties can be broken arbitrarily. The coefficients with new indices would be ordered in an increasing order, that is,

$$c_1 \leq c_2 \leq \cdots \leq c_n$$

Steps of the Algorithm

1. *Initialization*:

   (a) Check the problem for infeasibility. If the problem is infeasible then stop.

   (b) Set the lower bound to zero and the upper bound to ∞.

2. Branch on the node with the smallest lower bound, and in the order of increasing index of the variables.

3. If a 0-completion is feasible, then we have found a feasible solution to our sub-problem. In particular, if this feasible solution is the best one found so far, then update the upper bound to its optimal value.

4. - When the most recent variable was fixed to 1, and a 0-completion is not feasible, then the value of the optimization objective at the node is a lower bound.

- When the most recent variable was fixed to 0, and a 0-completion is not feasible, then the lower bound for such a node is obtained by setting the next unfixed variable to be equal to one[13].

5. Check the new sub-problem for infeasibility.
6. As in the general branch and bound algorithm for minimization, we fathom a node if:
   - it has a feasible solution
   - the problem is infeasible
   - the lower bound of the current node is higher than the current upper bound
7. Keep branching until all nodes are fathomed.

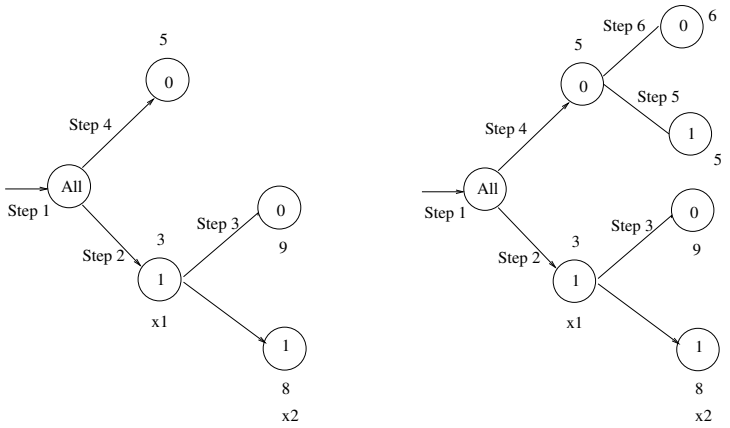
Illustrative Example

Let us consider the following ILP problem.

$$\begin{aligned} \min \quad & Z = 3x_1 + 5x_2 + 6x_3 + 9x_4 + 10x_5 + 10x_6 \\ \text{subject to} \quad & -2x_1 + 6x_2 - 3x_3 + 4x_4 + x_5 - 2x_6 \geq +2 \\ & -5x_1 - 3x_2 + x_3 + 3x_4 - 2x_5 + x_6 \geq -2 \\ & 5x_1 - x_2 + 4x_3 - 2x_4 + 2x_5 - x_6 \geq +3 \\ & x_j \in \{0,1\} \qquad \text{for } \; j = 1, 2, \ldots, 6 \end{aligned}$$

In Figure 18, we illustrate how Balas' Additive Method work works by showing two intermediate steps for the above ILP problem. After step 4, the node with the least lower bound of the objective function is the node with $x_1 = 0$. The lower bound for this node is 5. We branch on x_2 setting x_2 as 1 in step 5. The lower bound of the objective function is again 5. At this point, there are two nodes with the same lower bound. We choose the node with smaller index variable. We branch on x_2 again setting x_2 as 0 in step 6. The lower bound for this node is 6. Now, the best node to branch would be the one with $x_2 = 1$. The lower bound of objective function is 5 at this node. Since the next index is 3, we would branch on x_3.

[13] As noted in class, this can also be done when the most recent variable was fixed to 1, and the 0-completion is not feasible. In Balas' additive algorithm however, this improvement on the bound was used only when the recent variable to be fixed was set to 0.



(a) Step 1 to 4 (b) Step 5 and 6

Figure 18: Balas' Additive Method Example

27 Held and Karp's algorithm for TSP

One way of formulating the TSP problem on a graph $G = (V, E)$, where $V = \{1, 2, \ldots, n\}$ is a set of nodes and E is the set of edges, is to consider an 1-tree. An 1-tree is a spanning tree over $V \setminus \{1\}$ nodes + 2 edges adjacent to node 1. 1-tree with least cost can be computed by finding the minimum spanning tree and then adding the cheapest edges adjacent to node 1. So, cost of an optimal 1-tree T^* is at-most the cost of the optimal tour TSP^*.

$$| T^* | \leq | TSP^* |$$

Proof sketch: Consider an optimal tour. Remove edges adjacent to node 1. The remaining edges of the tour form a spanning tree of nodes $i = \{2, \ldots, n\}$. The cost of this spanning tree is greater than or equal to the cost of the minimum spanning tree. Optimal 1-tree is obtained by adding 2 cheapest edge adjacent to node 1. Thus, the cost of the optimal tour is greater than or equal to the cost of the 1-tree.

An important observation:

The following constraints suffice to enforce that every node is of degree 2.

$$\begin{aligned} \sum_{j>i} x_{ij} + \sum_{j<i} x_{ji} &= 2, \quad i = 1, \ldots, n-1 \\ \sum_{1 \leq i < j \leq n} x_{ij} &= n \end{aligned}$$

The first constraint ensures that all nodes $i = 1, \ldots, n-1$ are of degree 2. There is no need to specify that node n is of degree 2 because of the following lemma from graph theory:

Given a graph with nodes of degrees $d_1, d_2, \ldots, d_n$, then, $m = |E| = \sum_{i=1}^{n} d_i/2$.

Problem of finding minimum weight 1-tree:

This problem is solvable in polynomial time. We describe the two-step method below:

1. Find MST on $\{2,\ldots,n\}$. Greedy methods for MST give optimal answer and run in polynomial time ($O(m\log n)$). Some examples are Prim's algorithm for MST [23][14] and Kruskal's algorithm [16]

2. Add 2 cheapest edges adjacent to 1. The two cheapest edges can be found by a linear pass over all the neighbors of node 1. The neighbors can be atmost $n-1$. So, the complexity of this step is ($O(n)$).

Complexity: So, the overall complexity of finding optimum 1-tree is $O(m\log n)$
In order to guarantee that the computed 1-tree is a tour, we require that there is no cycle in $\{2,\ldots,n\}$ (subtour elimination).

Subtour elimination to transform a 1-tree to a tour:

The following ILP formulation uses the idea of 1-tree with no cycles in nodes $i=\{2,\ldots,n\}$ to find an optimal TSP.

$$\begin{aligned}
\min \quad & \sum_{1\le i<j\le n} c_{ij}x_{ij} \\
\text{subject to} \quad & \sum_{j>i} x_{ij} + \sum_{j<i} x_{ji} = 2 \quad \text{for} \quad i=2,\ldots,n-1 \\
& \sum_{i<j,i,j\in S} x_{ij} \le |S|-1 \quad \text{for} \quad S\subset\{2,\ldots,n\} \\
& \sum_{j} x_{1j} = 2 \\
& \sum_{1\le i<j\le n} x_{ij} = n, \\
& x_{ij}\in\{0,1\} \quad \text{for} \quad i<j
\end{aligned}$$

Note: Constraint $\sum_{i<j,i,j\in S} x_{ij} \le |S|-1$ is added for all $S\subset\{2,\ldots,n\}$. So, the number of constraints in this formulation is exponential in the number of nodes.

28 Lagrangian Relaxation

This is based on relaxing constraint the constraint

$$\sum_{j>i} x_{ij} + \sum_{j<i} x_{ji} = 2 \quad \text{for} \quad i=2,\ldots,n-1$$

which imposes the restriction that the degree of each node is exactly 2.

$$L(\vec{\pi}) = \sum_{1\le i<j\le n} c_{ij}x_{ij} + \sum_{i=2}^{n-1} \pi_i(2-[\sum_{j>i} x_{ij} + \sum_{j<i} x_{ji}])$$

where π_i unrestricted and $\sum_{j>i} x_{ij} + \sum_{j<i} x_{ji} = d_i$, the degree of node i.

[14] Complexity depends on data structures used. The best is $O(m\log n)$.

Consider the following integer program,

$$\begin{aligned} \min \quad & L(\vec{\pi}) \\ \text{subject to} \quad & \sum_{i<j,i,j\in S} x_{ij} \leq |S| - 1 \quad \text{for} \quad S \subset \{2,\ldots,n\} \\ & \sum_{j} x_{1j} = 2 \\ & \sum_{1\leq i<j\leq n} x_{ij} = n, \\ & x_{ij} \in \{0,1\} \quad \text{for} \quad i<j \end{aligned}$$

We can rewrite $L(\vec{\pi})$ as:

$$L(\vec{\pi}) = \sum_{1\leq i<j\leq n} c_{ij}x_{ij} - 2\sum_{i=2}^{n-1}\pi_i - \sum_{i=2}^{n-1}\pi_i d_i$$

that is,

$$L(\vec{\pi}) = 2\sum_{i=2}^{n-1}\pi_i + \sum_{1\leq i<j\leq n}(c_{ij} - \pi_i - \pi_j)x_{ij}$$

Let $L^* = \min \ \ L(\vec{\pi})$ and TSP^* be the cost of the optimal tour. We know that

$$|L^*| \leq |TSP^*|$$

An Illustrative Example:

The example is presented in Figure 19.

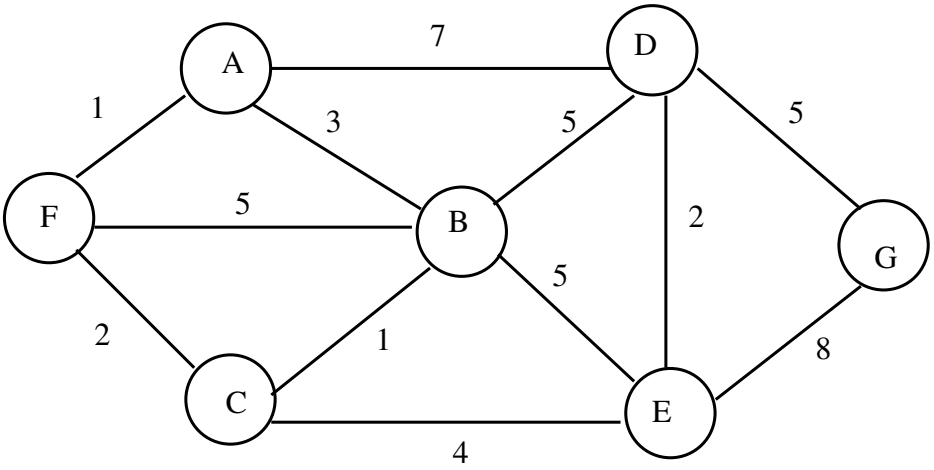


Figure 19: Example for Lagrangian Relaxation

We assume that edges not present have cost = 20. One tour found by inspection, with total weight 28, is: GDBAFCEG

Let node $1 = G$. We compute the minimum spanning tree for nodes $A - F$ and add the edges DG and EG to it to form the 1-tree. The found 1-tree is illustrated in Figure 20.

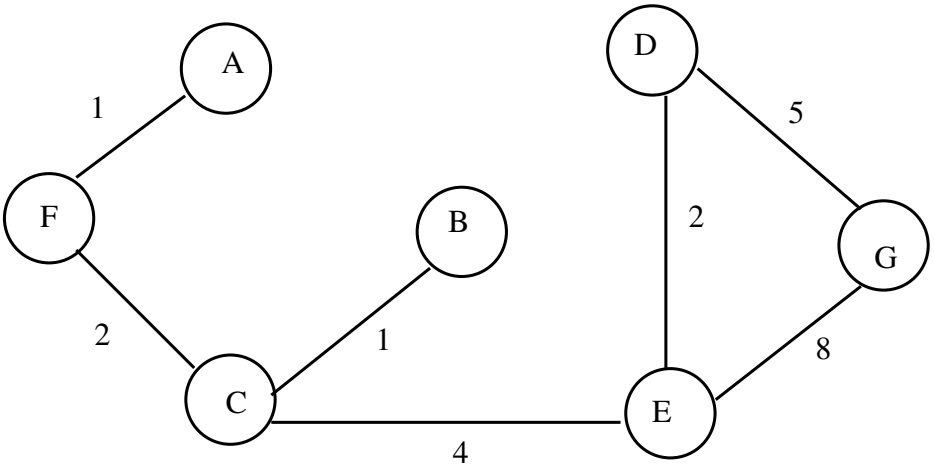


Figure 20: 1-Tree with $L((A, B, C, D, E, F) = (0, 0, 0, 0, 0, 0))$

The cost for this 1-tree is 21

$$L((A, B, C, D, E, F) = (0, 0, 0, 0, 0)) = 21$$

$$L(\vec{\pi}^*) = \min_{\vec{\pi}} \quad L(\vec{\pi})$$

The search for $\vec{\pi}^*$ is done using the sub-gradient method. For $i = 2, \ldots, n$,

- if $\sum_j x_{ij} < 2$, π_i is increased, that is, $\pi_i \nearrow$
- if $\sum_j x_{ij} > 2$, π_i is decreased, that is, $\pi_i \searrow$

Given, the solution for $L((A, B, C, D, E, F) = (0, 0, 0, 0, 0, 0))$, we make the following changes:

- $\pi_B \searrow, \pi_C \nearrow, \pi_E \searrow, \pi_F \nearrow$
- π_A, π_D remain unchanged.

The graph obtained after revision of the weights is illustrated in Figure 21. The 1-tree obtained is illustrated in Figure 22. The cost for this 1-tree is 23.

$$L((A, B, C, D, E, F) = (0, -1, 1, 0, -1, 1)) = 23 + (0 + (-1) + 3(-1) + 0 + 3(-1) + 2(1)) = 24$$

Given, the computed tree for $L((A, B, C, D, E, F) = (0, -1, 1, 0, -1, 1))$, we can make the following changes:

- $\pi_A \nearrow, \pi_B \nearrow, \pi_C \searrow, \pi_E \searrow$
- π_D, π_F remain unchanged.

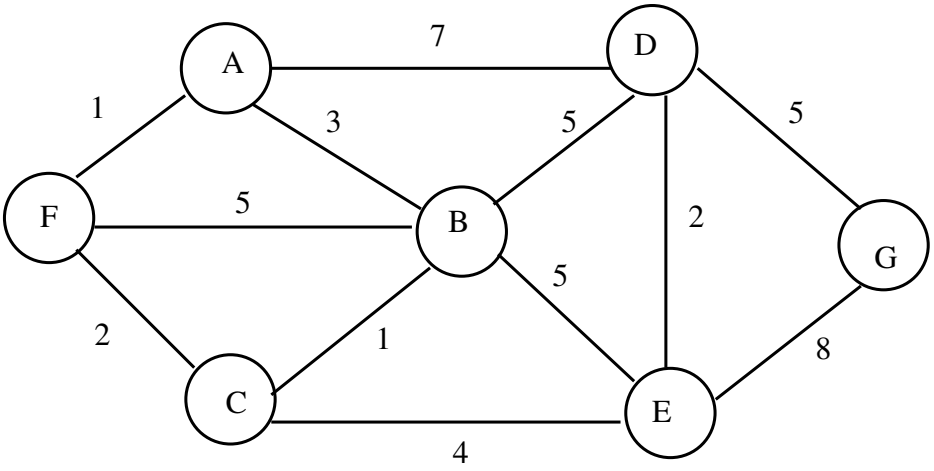


Figure 21: Recomputed edge weights for $L((A, B, C, D, E, F) = (0, -1, 1, 0, -1, 1))$

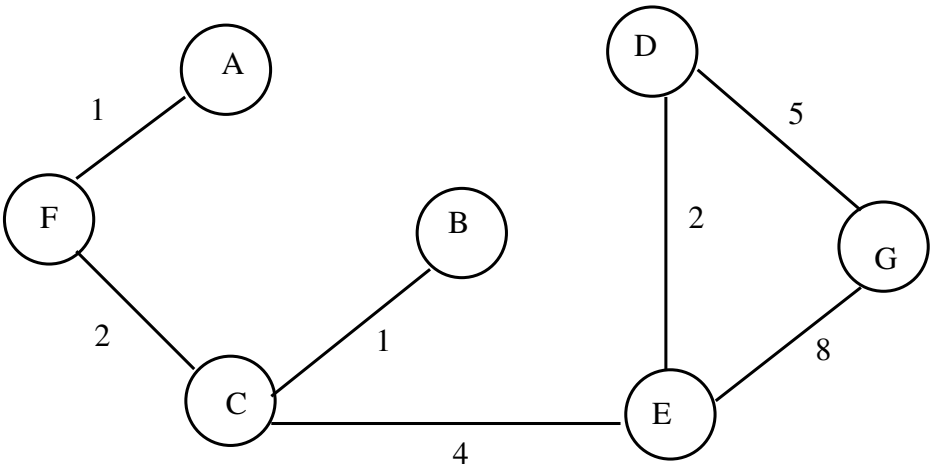


Figure 22: 1-tree with $L((A, B, C, D, E, F) = (0, -1, 1, 0, -1, 1))$

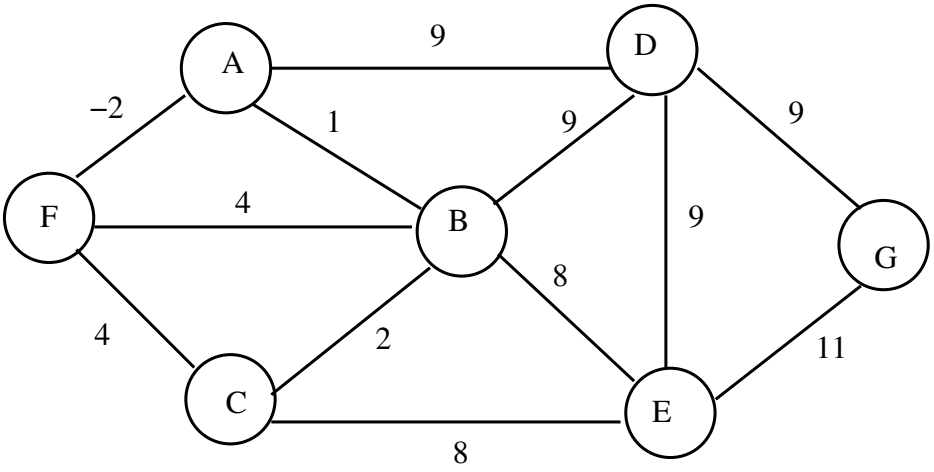


Figure 23: Recomputed edge weights for $L((A, B, C, D, E, F) = (2, -1, 0, -4, -4, 2))$

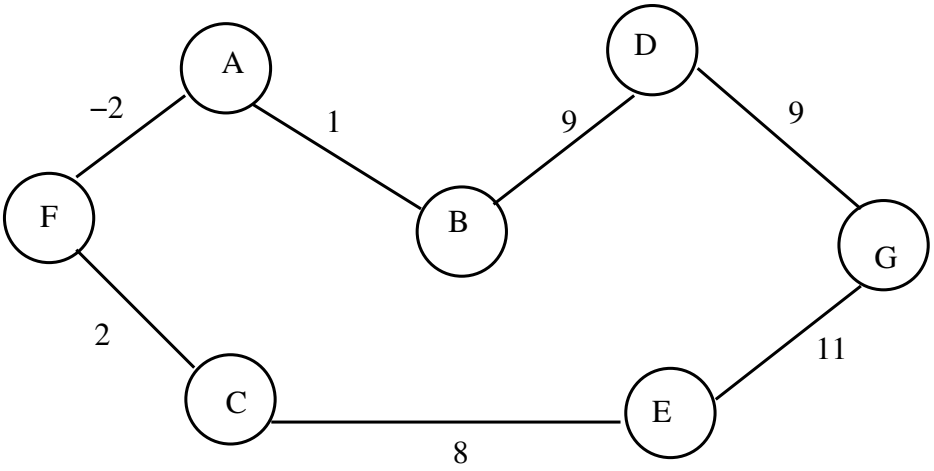


Figure 24: 1-tree with $L((A, B, C, D, E, F) = (2, -1, 0, -4, -4, 2))$

Let us consider the following set of multipliers.

$$\pi_A = 2, \pi_B = -1, \pi_C = 0, \pi_D = -4, \pi_E = -4, \pi_F = 2$$

Revising the costs revises the weight of the edges and the problem reduces to finding 1-tree of the graph illustrated in Figure 23. The 1-tree obtained is illustrated in Figure 24. The cost for this 1-tree is 38.

$$L((A, B, C, D, E, F) = (2, -1, 0, -4, -4, 2)) = 38 + 2(2 + (-1) + 0 + (-4) + (-4) + 2) = 28$$

The obtained 1-tree is also a tour and is in fact, an optimal tour. So, we find the cost of the most optimal tour in this example to be 28 and a tour with this cost is GDBAFCEG. Lec8

29 Lagrangian Relaxation

One of the most computationally useful ideas of the 1970's is the observation that many hard problems can be viewed as easy problems complicated by a relatively small set of side constraints. Define the problem

$$\begin{aligned} \max \quad & cx \\ \textit{subject to} \quad & Ax \leq b \\ & Cx \leq d \\ & x \geq 0, \textit{integer} \end{aligned}$$

We relax the set of constraints $Ax \leq b$ and add them to the objective function as a 'penalty term' to obtain the following problem, $L(\lambda)$

$$\begin{aligned} L(\lambda)_{\lambda \geq 0} = \max \quad & cx + \lambda(b - Ax) = (c - \lambda A)x + \lambda b \\ \textit{subject to} \quad & Cx \leq d \\ & x \geq 0, \textit{integer} \end{aligned} \tag{13}$$

We restrict the λ's to be positive to add a penalty when the constraint $Ax \leq b$ is not satisfied. If $Ax \leq b$ is changed to $Ax \geq b$ then we restrict the λ's to be negative. If the constraint is $Ax = b$

then the λ's are unrestricted.

We have the following relationship between $L(\lambda)$ and Z;

Theorem 29.1. $L(\lambda) \geq Z$

Proof. Let x^* be the optimal solution for (1). Then we have that x^* is also a feasible solution to $L(\lambda)$. Therefore, $L(\lambda)$ is an upper bound for Z. □

We would like the lowest upper bound on Z so we find the minimum of $L(\lambda)$ over all $\lambda \geq 0$.

$$Z_D = \min_{\lambda \geq 0} L(\lambda) \tag{14}$$

In the case of one variable, this problem can easily be seen to be a piecewise linear convex function of a single variable. However, it is not easy to see for larger dimensions, so we prove the following.

Theorem 29.2. *$L(\lambda)$ is convex.*

Proof. Let $\lambda \geq 0$ and $\lambda = \theta\lambda^1 + (1-\theta)\lambda^2$ for $0 \leq \theta \leq 1$ and $\lambda^1, \lambda^2 \geq 0$. Also, let $\bar{x}$ be an optimal solution to $L(\lambda)$, and $X = \{x \in \mathbb{Z} | Dx \leq e, x \geq 0\}$

$$\begin{aligned} L(\lambda) &= \max_{x \in X} \; cx + \sum_i \lambda_i (b - A_i x) \\ &= c\bar{x} + \sum_i \lambda_i (b - A_i \bar{x}) \end{aligned}$$

Since $\bar{x}$ is the optimal solution for λ, $\bar{x}$ is not necessarily optimal for λ^1 or λ^2. So we have the following inequalities,

$$\begin{aligned} L(\lambda^1) &\geq c\bar{x} + \sum_i \lambda_i^1 (b - A_i \bar{x}) \\ L(\lambda^2) &\geq c\bar{x} + \sum_i \lambda_i^2 (b - A_i \bar{x}) \end{aligned}$$

Therefore,

$$\theta L(\lambda^1) + (1-\theta) L(\theta^2) \geq L(\theta\lambda^1 + (1-\theta)\lambda^2)$$

□

To solve (2), one such method is the steepest descent algorithm. To begin this algorithm, choose a starting lambda, λ_0

$$L(\lambda_0) = \max_{x \in X} cx + \lambda_0 (b - Ax)$$

We solve this problem and denote the optimal solution by x_0. We now want to update λ_0. The steepest descent method chooses the negative of the gradient of the objective function of (2) as the direction of descent. Hence, we take the gradient of $cx_0 + \lambda(b - Ax_0)$ with respect to λ. This gradient is easily seen to be $(b - Ax_0)$. We update the λ's using the following relation (where we are now updating from λ_{i-1} to λ_i):

$$\lambda_i = \lambda_{i-1} - [\frac{(b - A\bar{x}_{i-1})}{||b - A\bar{x}_{i-1}||}] s_{i-1}$$

Where s_{i-1} is the step size at the i^{th} iteration.

For the steepest descent algorithm to converge, the step sizes must satisfy the following sufficient conditions (although they are typically not the fastest for converging, e.g. $s_k = \frac{1}{k}$),

$$\sum_{k=1}^{\infty} s_k = \infty$$
$$\lim_{k \to \infty} s_k = 0$$

However, Fisher recommends the following step sizes,

$$s_k = \frac{\lambda_k (L(\lambda_k) - Z^*)}{\sum_i (b_i - A_i \bar{x}_i^k)^2}$$

Where Z^* is the value of the best feasible solution found so far. Note, if $L(\lambda_k) = Z^*$ then, by Theorem 1.1, we have found a feasible solution that is also optimal. This will cause the next step size to be zero in Fisher's formula, which is an indication that we are at the optimal solution. Therefore, the algorithm terminates when we have indeed found an optimal feasible solution.

In Practice, Lagrangian Relaxation is typically done in the following manner:
1.) Choose a subset of constraints to relax (typically the constraints that makes the resulting problem easier to solve).
2.) Solve the sub problem with the relaxed constraints.
3.) Find a proper step size (e.g. Fisher's) and solve using Steepest Descent or another algorithm.

30 Complexity of Nonlinear Optimization

30.1 Input for a Polynomial Function

To input a function that is a polynomial of the form $a_0 + a_1 x + a_2 x^2 + \ldots + a_k x^k$, we form a $(k+1) \times 1$ vector whose entries are the $k+1$ coefficients of the polynomial. The i^{th} entry of the vector is a_{i-1}.
For a polynomial with two variables, we form a two dimensional matrix with the ij^{th} entry equal to a_{ij} where a_{ij} is the coefficient of the term $x^{i-1}y^{j-1}$ (i.e. $a_{ij}x^{i-1}y^{j-1}$). The resulting matrix will be $k_1 + 1 \times k_2 + 1$, where k_1 is the highest exponent of x and k_2 is the highest exponent of y.
For polynomials with more variables, we follow the same method by using a matrix with the same dimensions as the number of variables.
Note, only polynomial functions can be represented with this method. For a general non-linear function, we do not have a good representation for input.

30.2 Table Look-Up

When working with computers, we restrict the accuracy of the numbers we are working with since we can only compute with finite precision.
Even though there is no nice representation for general non-linear functions, we can use a method called Table Look-Up during the execution of an algorithm. If we have a function of several variables, $f(x_1, \ldots, x_n)$, table look-up takes an argument that is a vector, $\bar{x} \in \mathbb{R}^n$, and delivers the

value of the function at $\bar{x}$, $f(\bar{x})$, with specified accuracy. Table look-up can be visualized in the following table format,

$$\begin{array}{cc} x & f(x) \\ \vdots & \vdots \end{array}$$

30.3 Examples of Non-linear Optimization Problems

When dealing with non-linear optimization problems there are two alternative characterizations of optimal solutions,

$$|f(x) - f(x^*)| \leq \epsilon \tag{15}$$

$$||x - x^*|| \leq \epsilon \tag{16}$$

(3) indicates that x is an ϵ-approximate solution, where x^* is the optimal solution. (4) indicates that x is an ϵ-accurate solution, where x^* is the optimal solution as well. Note: for an ϵ-approximate solution, the values of x that satisfy this inequality could be arbitrarily far from the optimal solution, x^*. Also note: for an ϵ-accurate solution, the value of the function at values of x that satisfy this inequality could be arbitrarily far from the value of the function at the optimal solution, $f(x^*)$.

An example that involves separable convex optimization over linear constraints is,

$$\begin{array}{rl} \min & \sum_{j=1}^{n} f_j(x_j) \\ \textit{subject to} & Ax = b \\ & l \leq x \leq u \end{array} \tag{17}$$

where the f_j's are convex functions $\forall j$.
Note: A problem of the form $P = \{min\ f(x) : x \in \mathbb{R}^n\}$ cannot be addressed with finite complexity. This is because we are dealing with real numbers, which we cannot represent. We can only use a rational approximation which limits the accuracy of the numbers in our computations.

A problem in the form of (5), is the following optimization problem with piecewise linear functions. In the new problem the $f_j(x)$'s are piecewise linear functions with k-pieces (i.e. the function f_j is made up of k linear pieces):

$$\begin{array}{rl} \min & \sum_{j=1}^{n} f_j(x_j) \\ \textit{subject to} & Ax = b \\ & x \geq 0 \end{array} \tag{18}$$

We transform problem (6) into the following:

$$\begin{array}{rl} \min & \sum_{j=1}^{n} b_{0,j} + \sum_{i=1}^{k} c_{ij}\delta_{ij} \\ \textit{subject to} & \sum_{j=1}^{n} A_j(\sum_{i=1}^{k} \delta_{ij}) = b \\ & d_{ij} \geq \delta_{ij} \geq 0, \quad \forall i \in \{1,\ldots,k\}, j \in \{1,\ldots,n\} \end{array}$$

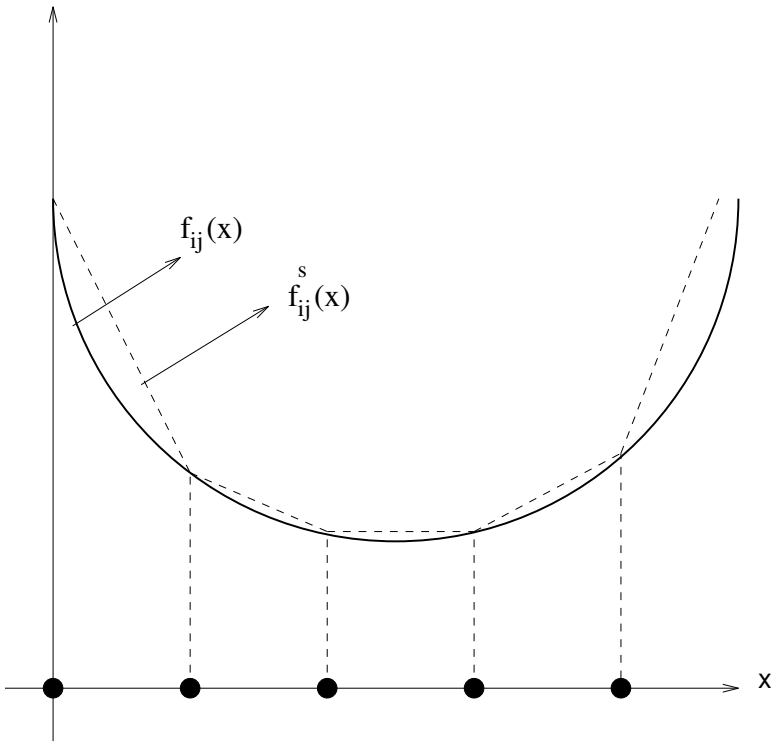


Figure 25: Linear Piecewise Approximation

Where d_{ij} is the length of the i^{th} linear piece of the j^{th} function; A_j is the j^{th} column of A; δ_{ij} is the length of the i^{th} linear piece of the j^{th} function that is actually used; $x_j = \sum_{i=1}^{k} \delta_{ij}$; $b_{0,j}$ is the value $f_j(\mathbf{0})$ function; and c_{ij} is the slope of the i^{th} linear piece of the j^{th} function. Notice that we can get rid of the constant term in the objective function since this term will not effect the minimization. Also note that we do not need to enforce the requirement that if $\delta_{ij} > 0$ then $\delta_{i-1,j} = d_{i-1,j}$. This is because since we are dealing with a convex function, the slopes (c_{ij}'s) are increasing. Hence, the minimization will force the δ_{ij}'s corresponding to the lower c_{ij}'s to reach d_{ij} before the next one becomes non-zero.

The following theorem, stated without proof, is known as the Proximity Theorem. The s-piecewise linear optimization problem mentioned in the theorem is problem (6) with each segment having length s.

Theorem 30.1. *Let $x^* \in \mathbb{R}^n$ be the optimal solution vector to problem* (6)*, and $x^s \in \mathbb{R}^n$ be the optimal solution to the s-piecewise linear optimization problem, where s is the length of each linear piece segment. Then $||x^* - x^s||_\infty \leq n\Delta s$ where Δ is the maximum sub-determinant of A.*

From the above theorem, we see that in order to have an ϵ-accurate solution to problem (6) we need $n\Delta s \leq \epsilon$. Therefore, if $s \leq \frac{\epsilon}{n\Delta}$, then x^s will indeed be an ϵ-accurate solution to problem (6) (i.e. $\epsilon \geq n\Delta s$). We can subdivide the interval $[\ell, u]$ into $4n\Delta$ pieces.

$$\frac{u-\ell}{4n\Delta} = s$$

Notice, the length of each linear piece depends on the sub-determinant of A. This value, Δ takes a polynomial number of operations to compute. Substituting Δ pieces makes the number of variables

pseudopolynomial.
Instead, we can iteratively divide the interval $u - \ell$ into halves until we get the length of each linear piece that will yield an ϵ-accurate solution. More precisely, the Proximity-Scaling Algorithm (Hochbaum and Shantikumar):

The proximity-scaling algorithm can be employed whenever there is a valid proximity theorem. For convex network flow the proximity theorem is $||\mathbf{x}^s - \mathbf{x}^{\frac{s}{2}}||_\infty \leq ms$. We call α the *proximity factor* (In our case, $\alpha = n\Delta$) if $\| \mathbf{x}^s - \mathbf{x}^{\frac{s}{2}} \|_\infty \leq \alpha s$.
The algorithm is implemented as follows. The scaling unit is selected initially to be $s = \lceil \frac{U}{4\alpha} \rceil$ for $U = \max_{(i,j)\in A}\{u_{ij} - \ell_{ij}\}$. The interval for variable x_{ij}, $[\ell_{ij}, u_{ij}]$ is thus replaced by up to 4α intervals of length s each.

Proximity-scaling algorithm:

Step 0: Let $s = \lceil \frac{U}{4\alpha} \rceil$.

Step 1: Solve (LNF-s) or (INF-s) with an optimal solution $\mathbf{x}^s$. If $s = 1$ output the solution and stop.

Step 2: Set $\ell_{ij} \leftarrow \max\{\ell_{ij}, x^s_{ij} - \alpha s\}$ and $u_{ij} \leftarrow \min\{u_{ij}, x^s_{ij} + \alpha s\}$, for $(i, j) \in A$.

Step 3: $s \leftarrow \lceil \frac{s}{2} \rceil$. Go to step 1.

The total number of iterations will be,

$$(\frac{1}{2})^q (u - \ell) \leq \epsilon$$
$$\frac{u - \ell}{\epsilon} \leq 2^q$$
$$\lceil (\log_2 \frac{U}{\epsilon}) \rceil = q$$

Where $U = u - \ell$

30.4 Impossibility of strongly polynomial algorithms for nonlinear (non-quadratic) optimization

Consider the Allocation Problem, where the $f_j(x)$'s are piecewise linear functions with k-pieces:

$$\begin{aligned} \max \quad & \sum_{j=1}^{n} f_j(x_j) \\ \textit{subject to} \quad & \sum_{j=1}^{n} x_j \leq B \\ & u_j \geq x_j \geq 0 \end{aligned}$$

To solve this problem we perform a Greedy Algorithm by calculating the gradients (i.e. $\Delta_1(1), \Delta_1(2), \ldots, \Delta_1(k)$ for $f_1()$; $\Delta_2(1), \Delta_2(2), \ldots \Delta_2(k)$ for $f_2()$; $\ldots$; $\Delta_n(1), \Delta_n(2), \ldots, \Delta_n(k)$ for $f_n()$), where (letting d_{ij} denote the length of the i^{th} linear piece of the j^{th} function)

$$\Delta_j(i) = \frac{f_j(\sum_{l=0}^{i} d_{ij}) - f_j(\sum_{l=0}^{i-1} d_{ij})}{d_{ij}}.$$

After we calculate the gradients, we add the highest gradients using the procedure below until we run out of capacity (i.e. we reach B).

(1) Take the largest gradient, say $\Delta_j(i)$.
(2) Update $x_j \leftarrow x_j + \min\{d_{ij}, B, u_j\}$.
(2′) Temporarily store B for step (4). $\bar{B} \leftarrow B$.
(3) Update $B \leftarrow [B - \min\{d_{ij}, B, u_j\}]$. If $B = 0$, STOP. We have reached the optimal solution.
(4) Update $u_j \leftarrow [u_j - \min\{d_{ij}, \bar{B}, u_j\}]$.
(5) Delete $\Delta_j(i)$. If there are remaining gradients, return to Step (1). Else, STOP.

This greedy algorithm works because the coefficient of each x_j is one, so only the gradients need to be considered when determining optimality. Any algorithm solving the simple allocation problem for f_j nonlinear and non-quadratic must depend on $log_2 B$

Please refer to the handout "The General Network Flow Problem," for the example explained in the rest of class. Lec9

31 The General Network Flow Problem Setup

A common scenario of a network-flow problem arising in industrial logistics concerns the distribution of a single homogenous product from plants (origins) to consumer markets (destinations). The total number of units produced at each plant and the total number of units required at each market are assumed to be known. The product need not be sent directly from source to destination, but may be routed through intermediary points reflecting warehouses or distribution centers. Further, there may be capacity restrictions that limit some of the shipping links. The objective is to minimize the variable cost of producing and shipping the products to meet the consumer demand.
The sources, destinations, and intermediate points are collectively called *nodes* of the network, and the transportation links connecting nodes are termed *arcs*. Although a production/distribution problem has been given as the motivating scenario, there are many other applications of the general model. Figure 31 indicates a few of the many possible alternatives.

Table 1: Other applications

	Urban Transportation	Communication System	Water Resources
Product	Buses, autos, etc.	Messages	Water
Nodes	Bus stops, street intersections	Communication centers, relay stations	Lakes, reservoirs, pumping stations
Arc	Streets (lanes)	Communication channels	Pipelines, canals, rivers

Here is a general instance statement of the minimum-cost flow problem: Given a network $G = (N, A)$, with a cost c_{ij}, upper bound u_{ij}, and lower bound l_{ij} associated with each directed arc (i, j), and supply b_v at each node. Find the cheapest integer valued *flow* such that it satisfies: 1) the capacity constraints on the arcs, 2) the supplies/demands at the nodes, 3) that the flow is conserved through the network.

A numerical example of a network-flow problem is given in Figure 26. The nodes are represented by numbered circles and the arcs by arrows. The arcs are assumed to be *directed* so that, for instance,

material can be sent from node 1 to node 2, but not from node 2 to node 1. Generic arcs will be denoted by (i, j), so that $(4, 5)$ means the arc *from* 5 *to* 5. Note that some pairs of nodes, for example 1 and 5, are not connected directly by an arc.

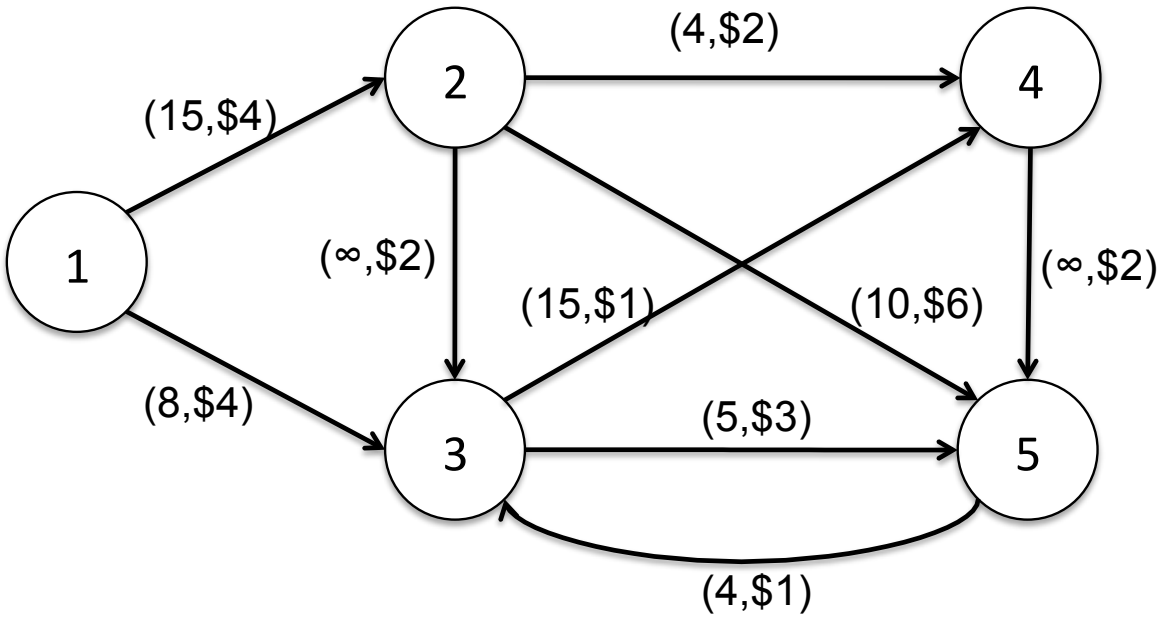


Figure 26: Numerical example of a network-flow problem

Figure 26 exhibits several additional characteristics of network flow problems. First, a *flow capacity (u_{ij})* is assigned to each arc *(i,j)*, and second, *a per-unit cost (c_{ij})* is specified for shipping along each arc *(i,j)*. There characteristics are shown next to each arc. This, the flow on arc 2-4 must be between 0 and 4 units, and each unit flow on this arc costs \$2.00. The ∞'s in the figure have been used to denote unlimited flow capacity on arcs $(2, 3)$ and $(4, 5)$. Finally, the numbers in parenthesis next to the nodes give the *material supplied or demanded (b_i)* at that node i. In this case, node 1 is an origin or *source node* supplying 20 units, and nodes 4 and 5 are destinations or *sink nodes* requiring 5 and 15 units, respectively, as indicated by the negative signs. The remaining nodes have no net supply or demand; they are intermediate points, often referred to as *transshipment nodes*.

The objective is to find the minimum-cost flow pattern to fulfil demands from the source nodes. such problems usually are referred to as *minimum-cost flow* or *capacitated transhipment* problems. To transcribe the problem into a formal linear program, let

$$f_{ij} = \text{feasible flow} = \text{Number of units shipped from node } i, \text{ to } j \text{ using arc } (i, j).$$

Then the tabular form of the linear-programming formulation associated with the network of Figure 26 is as shown in Figure 31.

The first five equations are flow-balance equations at the nodes. They state the conservation-of-flow law.
The distinctive feature of this matrix of the flow balance constraint matrix is that each column has precisely one 1 and one -1 in it.

Table 2: Tabular form associated with the network of Figure 26

	f_{12}	f_{13}	f_{23}	f_{24}	f_{25}	f_{34}	f_{35}	f_{45}	f_{53}	Right-hand Side
Node 1	1	1								20
Node 2	-1		1	1	1					0
Node 3		-1	-1			1	1		-1	0
Node 4				-1		-1		1		-5
Node 5					-1		-1	-1	1	-15
Capacities	15	8	+inf	4	10	15	5	+inf	4	
Objective Function	4	4	2	2	6	1	3	2	1	(Min)

$$\left(\begin{array}{l}\text{Flow out}\\ \text{of a node}\end{array}\right) - \left(\begin{array}{l}\text{Flow into}\\ \text{a node}\end{array}\right) = \left(\begin{array}{l}\text{Net supply}\\ \text{at a node}\end{array}\right)$$

$$A = \begin{bmatrix} 0 & 1 & \cdots \\ 0 & -1 & \cdots \\ 1 & 0 & \\ 0 & \vdots & \\ -1 & \vdots & \\ 0 & & \\ \vdots & & \end{bmatrix} \tag{19}$$

It is important to recognize the special structure of these balance equations. This type of tableau is referred to as a *node-arc incident matrix* or *totally unimodular constraint matrix*; it completely describes the physical layout of the network. It turns out that all extreme points of the linear programming relaxation of the Minimum Cost Network Flow Problem are integer valued (assuming that the supplies and upper/lower bounds are integer valued). Therefore MCNF problems can be solved using LP techniques. However, we will investigate more efficient techniques to solve MCNF problems, which found applications in many areas. In fact, the assignment problem is a special case of MCNF problem, where each person is represented by a supply node of supply 1, each job is represented by a demand node of demand 1, and there are an arc from each person j to each job i, labeled by $c_{i,j}$, that denotes the cost of person j performing job i.

The remaining two rows in the table give the upper bounds on the variables and the cost of sending one unit of flow across an arc. For example, f_{12} is constrained by $0 \leq f_{12} \leq 15$ and appears in the objective function as $2f_{12}$. In this example the lower bounds on the variables are taken implicitly to be zero, although in general there may be nonzero lower bounds.

With the setup given above, we can state the problem in a bit more general form for Figure 26

$$\begin{aligned}
\min \quad & \sum_{(i,j)\in A} c_{ij} f_{ij} \\
\textit{subject to} \quad & (f_{23} + f_{25} + f_{24}) - f_{12} = 0 && (20)\\
& (f_{35} + f_{34}) - (f_{13} + f_{23} + f_{53}) = 0 && (21)\\
& f_{45} - (f_{24} + f_{34}) = -5 && (22)\\
& f_{53} - (f_{25} + f_{35} + f_{45}) = -15 && (23)\\
& 0 \le f_{ij} \le U_{ij} \quad \forall_{(i,j)\in A} && (24)
\end{aligned}$$

To further generalize a Minimum-cost Flow Problem with n nodes, we have the following formulation:

$$\begin{aligned}
\min \quad & z = \sum_i \sum_j c_{ij} f_{ij} \\
\textit{subject to} \quad & \sum_j f_{ij} - \sum_k f_{ki} = b_i \ \ \forall_{i=1,2,\ldots,n} \ \text{[Flow Balance]} && (25)\\
& l_{ij} \le f_{ij} \le u_{ij} \ \ \text{[Flow Capacities]} && (26)
\end{aligned}$$

Comments: The first set of constraints is formed by the flow balance constraints. The second set is formed by the the capacity constraints. Nodes with negative supply are sometimes referred to as *demand* nodes and nodes with 0 supply are sometimes referred to as *transshipment* nodes. Notice that the constraints are dependent (left hand sides add to 0, so that the right hand sides must add to for consistency) and, as before, one of the constraints can be removed. Another thing that worth thinking is that what would happen if we have a negative cost on an arc. For example, let $C_{35} = 3$ and $C_{53} = -6$, we will then have a negative cycle in our network. Therefore we could just use this cycle repeatedly to minimize our total cost. Hence our problem becomes unbounded. Hence we conclude that as long as we don't have a negative cycle in our network, we are then ok with having negative costs on our arc(s).

32 Shortest Path Problem

The *Shortest Path* problem is defined on a directed, weighted graph, where the weights may be thought of as distances. The objective is to find a path from a source node, s, to node a sink node, t, that minimizes the sum of weights along the path. To formulate as a network flow problem, let $x_{i,j}$ be 1 if the arc (i, j) is in the path and 0 otherwise. Place a supply of one unit at s and a demand of one unit at t. The formulation for the problem is:

$$\min \quad \sum_{((i,j)\in A} d_{i,j}x_{i,j}$$

$$\textit{subject to} \quad \sum_{(i,k)\in A} x_{i,k} - \sum_{(j,i)\in A} x_{j,i} = 0 \qquad \forall i \neq s,t \tag{27}$$

$$\sum_{(s,i)\in A} x_{s,i} = 1 \tag{28}$$

$$\sum_{(j,t)\in A} x_{j,t} = 1 \tag{29}$$

$$0 \leq x_{i,j} \leq 1 \tag{30}$$

This problem is often solved using a variety of search and labeling algorithms depending on the structure of the graph.

33 Maximum Flow Problem

33.1 Setup

The *Maximum Flow* problem is defined on a directed network $G = (V, A)$ with capacities u_{ij} on the arcs, and no costs. In addition two nodes are specified, a source node, s, and sink node, t. The objective is to find the maximum flow possible between the source and sink while satisfying the arc capacities. (We assume for now that all lower bounds are 0.)

Definitions:

- A *feasible flow* f is a flow that satisfied the flow-balance and capacity constraints.
- A *cut* is a partition of the nodes (S, T), such that $S \subseteq V$, $s \in S$, $T = V \setminus S$, $t \in T$.
- *Cut capacity:* $U(S,T) = \sum_{i\in S}\sum_{j\in T} u_{ij}$. (**Important:** note that the arcs in the cut are **only** those that go from S to T.)

An Example

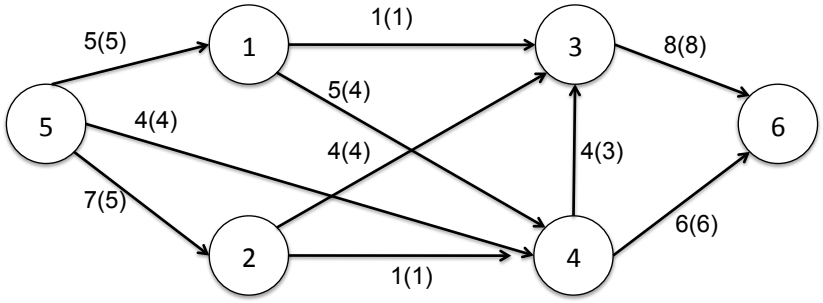


Figure 27: Maximum Flow Setup Example

See Figure 27, on each arc, we have a capacity and the amount of actual flow(within the parenthesis). The *source node* in the graph is node 5 and the *sink node* is in the graph is node 6. The

current solution is optimal; next, we will find a certificate of optimality.

Intuition:
Find a Minimum Cut.
A cut is said to be a *minimum cut* if it has the minimum *cut capacity*.
In this particular example, we can tell that node 2 and node 5 form a minimum cut (i.e. minimal source set), that is a source set S. We can also find that node $1, 3, 4$ and 6 form a sink set T. Since from any node (2 *and* 5) in the source set, we can't send any flow to T (all the capacities of the arcs coming out of this cut has been used). For the same reasoning, we can also tell that node 1, node 3, node 4 and node 6 form a minimum cut (i.e. maximal sink set).

Claim:
Given a maximum flow, we can always find a minimum cut (with minimal source set or with maximal source set)

Clarification: What is the difference between *maximum* and *maximal*?
An example of *maximum/maximal independence set (IS)*:
See Figure 28 A maximum IS (node 1, 3and 5) is a largest independent set for a given graph G (i.e. maximum independence)
A maximal IS (node 2 and 5) is an independent set that is not a subset of any other independent set (i.e. we can't add or remove any nodes from the set to maintain its independence).

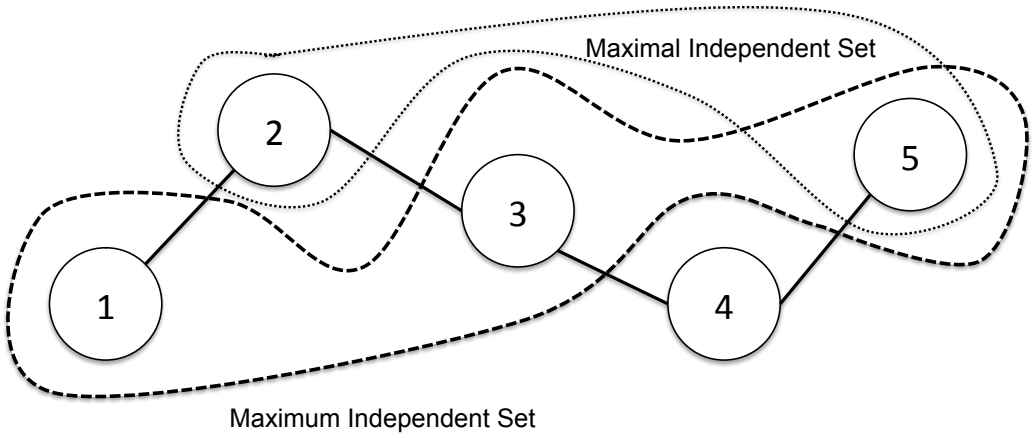


Figure 28: Maximum Independent Set Example

Let $|f|$ be the total amount of flow out of the source (equivalently into the sink). It is easy to observe that any feasible flow satisfies

$$|f| \leq U(S,T) \tag{31}$$

for any (S,T) cut. This is true since all flow goes from s to t, and since $s \in S$ and $t \in T$ (by definition of a cut), then all the flow must go through the arcs in the (S,T) cut. Inequality 31 is a special case of the *weak duality theorem* of linear programming.

The following theorem, which we will establish algorithmically, can be viewed as a special case of the strong duality theorem in linear programming.

Theorem 33.1 (Max-Flow Min-Cut)**.** *The value of a maximum flow is equal to the capacity of a cut with minimum capacity among all cuts. That is,*

$$\max |f| = \min U(S,T)$$

33.2 Algorithms

Next we introduce the terminology needed to prove the Max-flow/Min-cut duality and to give us an algorithm to find the max flow of any given graph.

Residual graph: The residual graph, $G_f = (V, A_f)$, with respect to a flow f, has the following arcs:

- forward arcs: $(i,j) \in A_f$ if $(i,j) \in A$ and $f_{ij} < u_{ij}$. The residual capacity is $u^f_{ij} = u_{ij} - f_{ij}$. The residual capacity on the forward arc tells us how much we can increase flow on the original arc $(i,j) \in A$.
- reverse arcs: $(j,i) \in A_f$ if $(i,j) \in A$ and $f_{ij} > 0$. The residual capacity is $u^f_{ji} = f_{ij}$. The residual capacity on the reverse arc tells us how much we can decrease flow on the original arc $(i,j) \in A$.

Intuitively, residual graph tells us how much **additional** flow can be sent through the original graph with respect to the given flow. This brings us to the notion of an augmenting path. In the presence of lower bounds, $\ell_{ij} \leq f_{ij} \leq u_{ij}$, $(i,j) \in A$, the definition of forward arc remains, whereas for reverse arc, $(j,i) \in A_f$ if $(i,j) \in A$ and $f_{ij} > \ell_{ij}$. The residual capacity is $u^f_{ji} = f_{ij} - \ell_{ij}$.

Augmenting path: An augmenting path is a path from s to t in the residual graph. The *capacity of an augmenting path* is the minimum residual capacity of the arcs on the path – the bottleneck capacity.
If we can find an augmenting path with capacity δ in the residual graph, we can increase the flow in the original graph by adding δ units of flow on the arcs in the original graph which correspond to forward arcs in the augmenting path and subtracting δ units of flow on the arcs in the original graph which correspond to reverse arcs in the augmenting path.
Note that this operation does not violate the **capacity constraints** since δ is the smallest arc capacity in the augmenting path in the residual graph, which means that we can always add δ units of flow on the arcs in the original graph which correspond to forward arcs in the augmenting path and subtract δ units of flow on the arcs in the original graph which correspond to reverse arcs in the augmenting path without violating capacity constraints.
The **flow balance** constraints are not violated either, since for every node on the augmenting path in the original graph we either increment the flow by δ on one of the incoming arcs and increment the flow by δ on one of the outgoing arcs (this is the case when the incremental flow in the residual graph is along forward arcs) or we increment the flow by δ on one of the incoming arcs and decrease the flow by δ on some other incoming arc (this is the case when the incremental flow in the residual graph comes into the node through the forward arc and leaves the node through the reverse arc) or we decrease the flow on one of the incoming arcs and one of the outgoing arcs in the original graph (which corresponds to sending flow along the reverse arcs in the residual graph).

33.2.1 Ford-Fulkerson algorithm

The above discussion can be summarized in the following algorithm for finding the maximum flow on a give graph. This algorithm is is also known as the **augmenting path algorithm**. Below is a pseudocode-like description.

Pseudocode:
f: flow;
G_f: the residual graph with respect to the flow;
P_{st}: a path from s to t;
U_{ij}: capacity of arc from i to j.

Initialize $f = 0$
If $\exists P_{st} \in G_f$ do
find $\delta = \min_{(i,j)\in P_{st}} U_{ij}$
$f_{ij} = f_{ij} + \delta \forall (i,j) \in P_{st}$
else stop f is max flow.

Detailed description:

1. Start with a feasible flow (usually $f_{ij} = 0\ \forall (i,j) \in A$).
2. Construct the residual graph G_f with respect to the flow.
3. Search for augmenting path by doing breadth-first-search from s (we consider nodes to be adjacent if there is a positive capacity arc between them in the residual graph) and seeing whether the set of s-reachable nodes (call it S) contains t.
   If S contains t then there is an augmenting path (since we get from s to t by going through a series of adjacent nodes), and we can then increment the flow along the augmenting path by the value of the smallest arc capacity of all the arcs on the augmenting path.
   We then update the residual graph (by setting the capacities of the forward arcs on the augmenting path to the difference between the current capacities of the forward arcs on the augmenting path and the value of the flow on the augmenting path and setting the capacities of the reverse arcs on the augmenting path to the sum of the current capacities and the value of the flow on the augmenting path) and go back to the beginning of step 3.
   If S does not contain t then the flow is maximum.

To establish correctness of the augmenting path algorithm we prove the following theorem which is actually stronger than the the max-flow min-cut theorem.
Reminder: f_{ij} is the flow from i to j, $|f| = \sum_{(s,i)\in A} f_{si} = \sum_{(i,t)\in A} f_{it} = \sum_{u\in S, v\in T} f_{uv}$ for any cut (S,T).

Theorem 33.2 (Augmenting Path Theorem)**.** *(generalization of the max-flow min-cut theorem) The following conditions are equivalent:*

1. *f is a maximum flow.*
2. *There is no augmenting path for f.*
3. *$|f| = U(S,T)$ for some cut (S,T).*

Proof.
$(1 \Rightarrow 2)$ If $\exists$ augmenting path p, then we can strictly increase the flow along p; this contradicts that the flow was maximum.

$(2 \Rightarrow 3)$ Let G_f be the residual graph w.r.t. f. Let S be a set of nodes reachable in G_f from s. Let $T = V \setminus S$. Since $s \in S$ and $t \in T$ then (S, T) is a cut. For $v \in S, w \in T$, we have the following implications:

$$\begin{aligned}(v,w) \notin G_f &\Rightarrow f_{vw} = u_{vw} \text{ and } f_{wv} = 0 \\ &\Rightarrow |f| = \sum_{v\in S, w\in T} f_{vw} - \sum_{w\in T, v\in S} f_{wv} = \sum_{v\in S, w\in T} u_{vw} = U(S,T)\end{aligned}$$

$(3 \Rightarrow 1)$ Since $|f| \leq U(S,T)$ for any (S,T) cut, then $|f| = U(S,T) \implies f$ is a maximum flow. □

Note that the equivalence of conditions 1 and 3 gives the max-flow min-cut theorem.

Given the max flow (with all the flow values), a min cut can be found in by looking at the residual network. The set S, consists of s and the all nodes that s can reach in the final residual network and the set $\bar{S}$ consists of all the other nodes. Since s can't reach any of the nodes in $\bar{S}$, it follows that any arc going from a node in S to a node in $\bar{S}$ in the original network must be saturated which implies this is a minimum cut. This cut is known as the minimal source set minimum cut. Another way to find a minimum cut is to let $\bar{S}$ be t and all the nodes that can reach t in the final residual network. This a *maximal source set minimum cut* which is different from the minimal source set minimum cut when the minimum cut is not unique.
While finding a minimum cut given a maximum flow can be done in linear time, $O(m)$, we have yet to discover an efficient way of finding a maximum flow given a list of the edges in a minimum cut other than simply solving the maximum flow problem from scratch. Also, we have yet to discover a way of finding a minimum cut without first finding a maximum flow. Since the minimum cut problem is asking for less information than the maximum flow problem, it seems as if we should be able to solve the former problem more efficiently than the later one. More on this in Section 56.
In a previous lecture we already presented Ford-Fulkerson algorithm and proved its correctness. In this section we will analyze its complexity. For completeness purposes we give a sketch of the algorithm.

Ford-Fulkerson Algorithm
Step 0: $f = 0$
Step 1: Construct G_f (Residual graph with respect to flow f)
Step 2: Find an augmenting path from s to t in G_f
Let path capacity be δ
Augment f by δ along this path
Go to step 1
If there does not exist any path
Stop with f as a maximum flow.

Theorem 33.3. *If all capacities are integer and bounded by a finite number U, then the augmenting path algorithm finds a maximum flow in time $O(mnU)$, where $U = \max_{(v,w)\in A} u_{vw}$.*

Proof. Since the capacities are integer, the value of the flow goes up at each iteration by at least one unit.
Since the capacity of the cut $(s, N \setminus \{s\})$ is at most nU, the value of the maximum flow is at most nU.
From the two above observations it follows that there are at most $O(nU)$ iterations. Since each iteration takes $O(m)$ time–find a path and augment flow, then the total complexity is $O(nmU)$. □

The above result also applies for rational capacities, as we can scale to convert the capacities to integer values.

Drawbacks of Ford-Fulkerson algorithm:

1. The running time is not polynomial. The complexity is exponential since it depends on U.
2. In Ford-Fulkerson algorithm which augmenting path to pick is not specified. Given the graph in Figure 29, the algorithm could take 4000 iterations for a problem with maximum capacity of 2000.

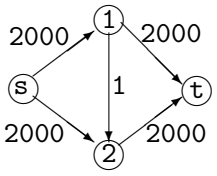


Figure 29: Graph leading to long running time for Ford-Fulkerson algorithm.

3. Finally, for irrational capacities, this algorithm may converge to the wrong value (see Papadimitriou and Steiglitz p.126-128)

Improvements for enhancing the performance of Ford-Fulkerson algorithm include:

1. Augment along maximum capacity augmenting paths. (See homework 5).
2. Using the concept of capacity scaling to cope with the non-polynomial running time. (Next section)
3. Augmenting along the shortest (in number of edges) path. (Later).

Each alternative leads to different types of max-flow algorithms.

33.2.2 Capacity scaling algorithm

We note that the Ford-Fulkerson algorithm is good when our capacities are small. Most algorithms with this type of properties can be transformed to polynomial time algorithms using a *scaling* strategy.

The idea in this algorithm is to construct a series of max-flow problems such that: the number of augmentations on each problem is small; the maximum flow on the previous problem to find a feasible flow to the next problem with a small amount of work; the last problem gives the solution to our original max-flow problem.

In particular, at each iteration k, we consider the network P_k with capacities restricted to the k most significant digits (in the binary representation). (We need $p = \lfloor \log U \rfloor + 1$ digits to represent the capacities.) Thus, at iteration k, we will consider the network where the capacities are given by the following equation:

$$u_{ij}^{(k)} = \left\lfloor \frac{u_{ij}}{2^{p-k}} \right\rfloor$$

Alternatively, note that the capacities of the kth iteration can be obtained as follows:

$$u_{ij}^{(k)} = 2u_{ij}^{(k-1)} + \text{next significant digit}$$

From the above equation it is clear that we can find a feasible flow to the current iteration by doubling the max-flow from the previous iteration.

Finally, note that the residual flow in the current iteration can't be more than m. This is true since in the previous iteration we had a max-flow and a corresponding min-cut. The residual capacity at each arc in the min-cut can grow at most by one unit. Therefore the max-flow in the current iteration can be at most m units bigger than the max flow at the previous operations.
The preceding discussion is summarized in the following algorithm.

Capacity scaling algorithm
$f_{ij} := 0$;
Consider P_0 and apply the augmenting path algorithm.
For $k := 1$ to n Do
    Multiply all residual capacities and flows of residual graph from previous iteration by 2;
    Add 1 to all capacities of arcs that have 1 in the (k+1)st position
        (of the binary representation of their original capacity);
    Apply the augmenting path algorithm starting with the current flow;
End Do

Theorem 33.4. *The capacity scaling algorithm runs in $O(m^2 \log_2 U)$ time.*

Proof. For arcs on the cut in the previous network residual capacities were increased by at most one each, then the amount of residual max flow in P_i is bounded by the number of arcs in the cut which is $\leq m$. So the number of augmentations at each iteration is $O(m)$.
The complexity of finding an augmenting path is $O(m)$.
The total number of iterations is $O(\log_2 U)$.
Thus the total complexity is $O(m^2 \log_2 U)$. □

Notice that this algorithm is polynomial, but still not strongly polynomial. Also it cannot handle real capacities.
We illustrate the capacity scaling algorithm in Figure 30

Theorem 33.5. *Dinic's algorithm solves correctly the max-flow problem.*

Proof. The algorithm finishes when s and t are disconnected in residual graph and so there is no augmenting path. Thus by the augmenting path theorem the flow found by Dinic's algorithm is a maximum flow. □

33.3 Maximum-flow versus Minimum Cost Network Flow

Question: How is a Max-flow problem a Minimum Cost Network Flow problem?
See Figure 31, we can simply add a circulation arc from the sink to the source node, $f(t, s)$. On this arc, we make the cost -1 and the capacity ∞. Then by finding the minimum cost in the graph, we are also finding the maximum flow in our original graph.

33.4 Formulation

Recall the IP formulation of the *Minimum Cost Network Flow* problem.

$$\begin{array}{lllll} & \text{Min} & cx & \\ (\text{ MCNF}) & \text{subject to} & Tx = b & \text{Flow balance constraints} \\ & & l \leq x \leq u, & \text{Capacity bounds} \end{array}$$

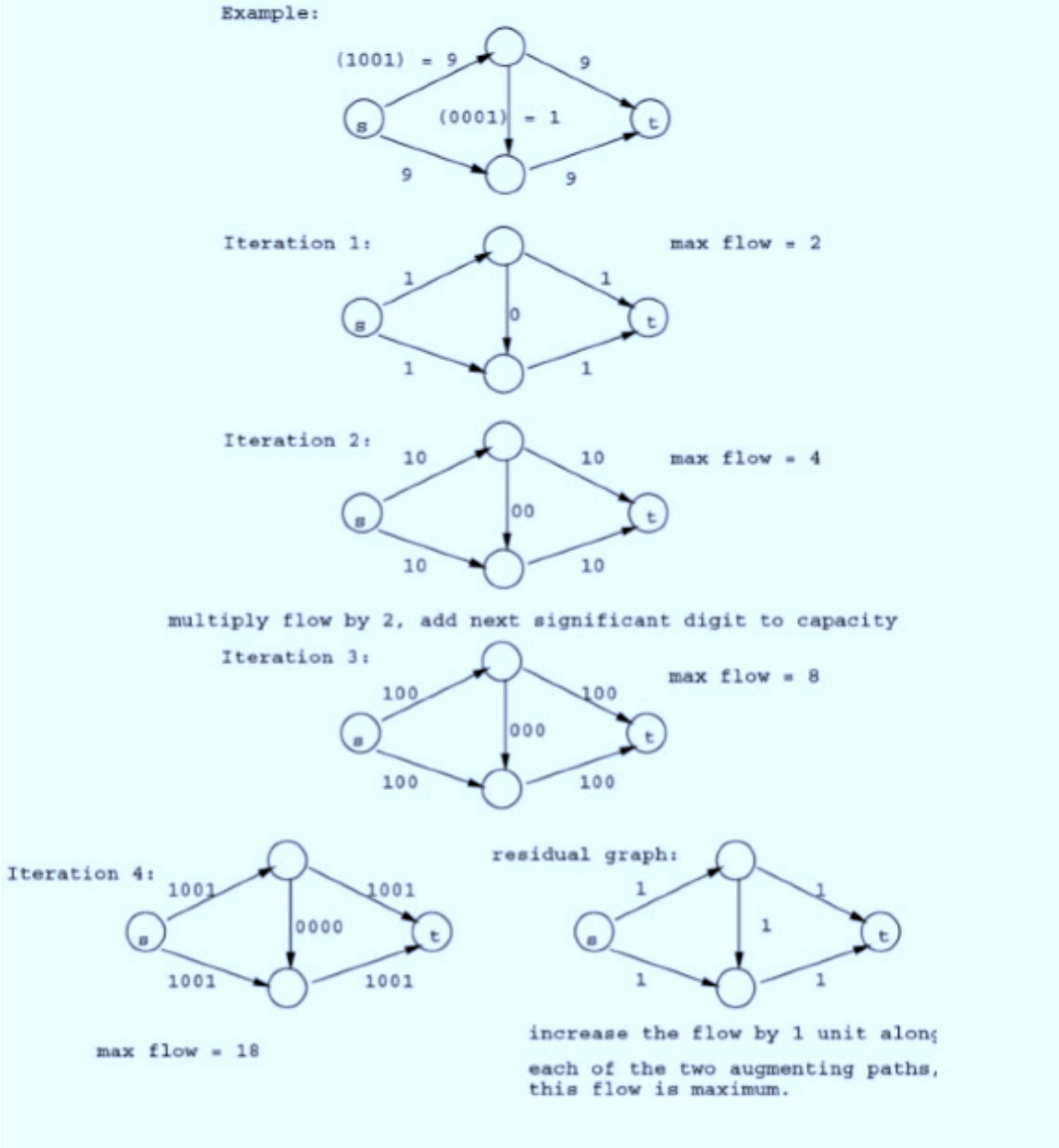


Figure 30: Example Illustrating the Capacity Scaling Algorithm

Here $x_{i,j}$ represents the flow along the arc from i to j. We now look at some specialized variants. The *Maximum Flow* problem is defined on a directed network with capacities u_{ij} on the arcs, and no costs. In addition two nodes are specified, a source node, s, and sink node, t. The objective is to find the maximum flow possible between the source and sink while satisfying the arc capacities. If $x_{i,j}$ represents the flow on arc (i, j), and A the set of arcs in the graph, the problem can be formulated as:

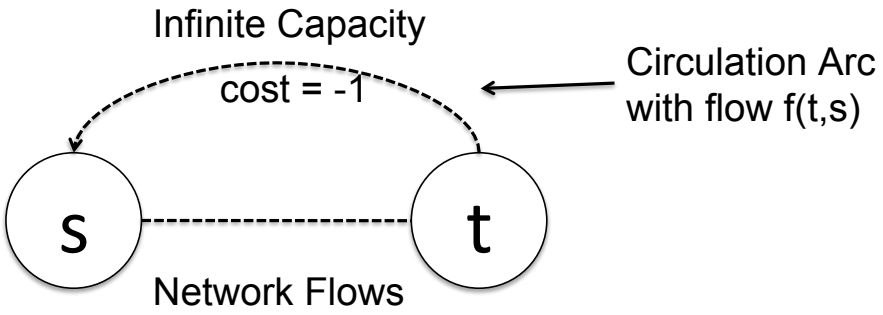


Figure 31: Maximum Flow versus Minimum Cost

$$
\begin{array}{lll}
 & \text{Max} & V \\
 & \text{subject to} & \sum_{(s,i)\in A} x_{s,i} = V \qquad \text{Flow out of source} \\
\text{(MaxFlow)} & & \sum_{(j,t)\in A} x_{j,t} = V \qquad \text{Flow in to sink} \\
 & & \sum_{(i,k)\in A} x_{i,k} - \sum_{(k,j)\in A} x_{k,j} = 0, \qquad \forall k \neq s,t \\
 & & l_{i,j} \leq x_{i,j} \leq u_{i,j}, \quad \forall (i,j) \in A.
\end{array}
$$

The *Maximum Flow* problem and its dual, known as the *Minimum Cut* problem have a number of applications and will be studied at length later in the course.

34 Minimum Cut Problem

34.1 Minimum s-t Cut Problem

Setup
$G = (V, A)$ with arc capacities u_{ij}'s
Want to find a partition $V = S \cup T$ such that,
$s \in S, t \in T$ and $\sum_{i\in S, j\in T, (i,j)\in A} u_{ij}$ is minimum.
In other words, we want to find a cut with the minimum capacity.
Observations:

- Which is easier? *2-cut* or *s-t cut*?
  Definitions:
  k-cut: a cut that partition a set into k subset.
  S1-S2-S3-cut: a cut that partition a set into 3 subset s1 s2 and s3 while each subset either contains the source node or the sink node in the graph.
  Answer: The *2-cut* is easier.

  From Figure 32, we can see that solving the *min S1-S2-S3-cut* will help solve the *min 3-cut*. Reasoning: a S1-S2-S3-cut is a special case of 3-cut. Then, we know that since *min 2-cut* is easier than *min 3-cut*, we conclude that the *min 2-cut* can be solved strictly more efficiently than the *s-t cut*.

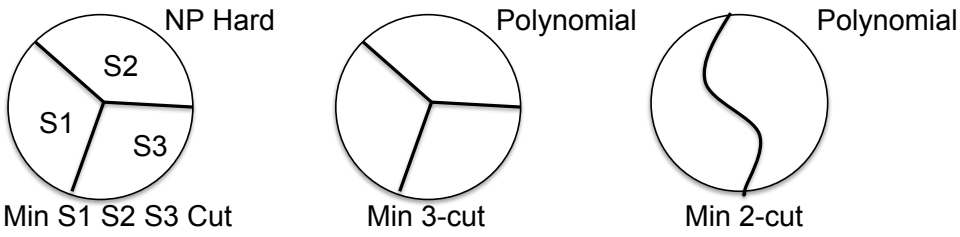


Figure 32: Different cuts

34.2 Formulation

Idea: Let's take a look at the dual of the max-flow formulation
Given $G = (N, A)$ and capacities u_{ij} for all $(i, j) \in A$:
Consider the formulation of the maximum flow problem given earlier.

$$\begin{aligned} \max \quad & f_{ts} \\ \textit{subject to} \quad & Outflow_s - Inflow_s = f_{ts} && (32)\\ & Outflow_i - Inflow_i = 0 \quad \forall i \in V \backslash \{s,t\} && (33)\\ & Outflow_t - Inflow_t = -f_{ts} && (34)\\ & 0 \le f_{ij} \le U_{ij} && (35) \end{aligned}$$

For dual variables we let $\{z_{ij}\}$ be the nonnegative dual variables associated with the capacity upper bounds constraints, and $\{\lambda_i\}$ the variables associated with the flow balance constraints.

$$\begin{aligned} \text{Min} \quad & \textstyle\sum_{(i,j)\in A} u_{ij} z_{ij} \\ \text{subject to} \quad & z_{ij} - \lambda_i + \lambda_j \ge 0 \quad \forall (i,j) \in A \\ & \lambda_s - \lambda_t \ge 1 \\ & z_{ij} \ge 0 \qquad \forall (i,j) \in A. \end{aligned}$$

Observations:
The dual problem has an infinite number of solutions: if (λ^*, z^*) is an optimal solution, then so is $(\lambda^* + \delta, z^*)$ for any constant δ. To avoid this, we set $\lambda_t = 0$ (or to any other arbitrary value). Observe now that with this assignment there is an optimal solution with $\lambda_s = 1$ and a partition of the nodes into two sets: $S = \{i \in V | \lambda_i = 1\}$ and $\bar{S} = \{i \in V | \lambda_i = 0\}$.

The complementary slackness condition states that the primal and dual optimal solutions x^*, λ^*, z^* satisfy,

$$x^*_{ij} \cdot [z^*_{ij} - \lambda^*_i + \lambda^*_j] = 0 \quad (36)$$
$$[u_{ij} - x^*_{ij}] \cdot z^*_{ij} = 0 \quad (37)$$

In an optimal solution $z^*_{ij} - \lambda^*_i + \lambda^*_j = 0$ so the first set of complementary slackness conditions (18) do not provide any information on the primal variables $\{x_{ij}\}$. As for the second set (19), $z^*_{ij} = 0$ on

all arcs other than the arcs in the cut (S, T). So we can conclude that the cut arcs are saturated, but derive no further information on the flow on other arcs.

The only method known to date for solving the minimum cut problem requires finding a maximum flow first, and then recovering the cut partition by finding the set of nodes reachable from the source in the residual graph (or reachable from the sink in the reverse residual graph). That set is the source set of the cut, and the recovery can be done in linear time in the number of arcs, $O(m)$.

On the other hand, if we are given a minimum cut, there is no efficient way of recovering the flow values on each arc other than solving the maximum flow problem from scratch. The only information given by the minimum cut, is the value of the maximum flow and the fact that the arcs on the cut are saturated. Beyond that, the flows have to be calculated with the same complexity as would be required without the knowledge of the minimum cut.

This asymmetry implies that it may be easier to solve the minimum cut problem than to solve the maximum flow problem. Still, no minimum cut algorithm has ever been discovered, in the sense that every known so-called minimum cut algorithm computes first the maximum flow.

35 Selection Problem

Suppose you are going on a hiking trip. You need to decide what to take with you. Many individual items are useless unless you take something else with them. For example, taking a bottle of wine without an opener does not make too much sense. Also, if you plan to eat soup there, for example, you might want to take a set of few different items: canned soup, an opener, a spoon and a plate. The following is the formal definition of the problem.

Given a set of items $J = \{1, \ldots, n\}$, a cost for each item c_j, a collection of sets of items $S_i \subseteq J$ for $i = 1, \ldots, m$, and a benefit for each set b_i. We want to maximize the net benefit (= total benefit - total cost of items) selected.

We now give the mathematical programming formulation:

$$\begin{aligned} \max \quad & \sum_{j=1}^{m} b_i x_i - \sum_{i=1}^{n} c_j y_j \\ & x_i \leq y_j \quad \text{for } i = 1, \ldots, m, \forall j \in S_i \\ & x_i \in \{0, 1\} \quad \text{for } i = 1, \ldots, m \\ & y_j \in \{0, 1\} \quad \text{for } j = 1, \ldots, n. \end{aligned}$$

Where,

$$x_i = \begin{cases} 1 & \text{if set } j \text{ is selected} \\ 0 & \text{otherwise} \end{cases}$$

$$y_j = \begin{cases} 1 & \text{if item } j \text{ is included} \\ 0 & \text{otherwise} \end{cases}$$

We immediately recognize that the constraint matrix is totally unimodular (each row has one 1 and one -1). Thus when solving the LP relaxation of the problem we will get a solution for the above IP.

Nevertheless, we can do better than solving the LP. In particular we now show how the Selection Problem can be solved by finding a minimum s-t cut in an appropriate network.

First we make the following observation. The optimal solution to the selection problem will consist of a collection of sets, $S_j, j \in J$, and the union of their elements ($\bigcup S_j, j \in J$). In particular we will never pick "extra items" since this only adds to our cost. Feasible solutions with this structure are called *selections*.

As per the above discussion, we can limit our search to selections, and we can formulate the selection problem as follows. Find the collection of sets S such that:

$$\max_S \sum_{i \in S} b_i - \sum_{j \in \bigcup_{i \in S} S_i} c_j \tag{38}$$

The following definition will be useful in our later discussion. Given a directed graph $G = (V, A)$, a *closed set* is a set $C \subseteq V$ such that $u \in C \text{and} (u, v) \in A \implies v \in C$. That is, a closed set includes all of the successors of every node in C. Note that both $\emptyset$ and V are closed sets.

Now we are ready to show how we can formulate the selection problem as a minimum cut problem. We first create a bipartite graph with sets on one side, items on the other. We add arcs from each set to all of its items. (See Figure 33) Note that a selection in this graph is represented by a collection of set nodes and all of its successors. Indeed there is a one-to-one correspondence between selections and closed sets in this graph.

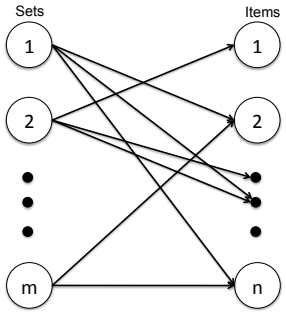


Figure 33: Representing selections as closed sets

Next we transform this graph into a maxflow-mincut graph. We set the capacity of all arcs to infinity. We add a source, a sink, arcs from s to each set i with capacity b_i, and arcs from from each set j to t with capacity c_j. (See Figure 34)
We make the following observations. There is a one-to-one correspondence between finite cuts and selections. Indeed the source set of any finite (S, T) cut is a selection. (If $S_j \in S$ then it must be true that all of its items are also in S—otherwise the cut could not be finite.) Now we are ready to state our main result.

Theorem 35.1. *The source set of a minimum cut (S, T) is an optimal selection.*

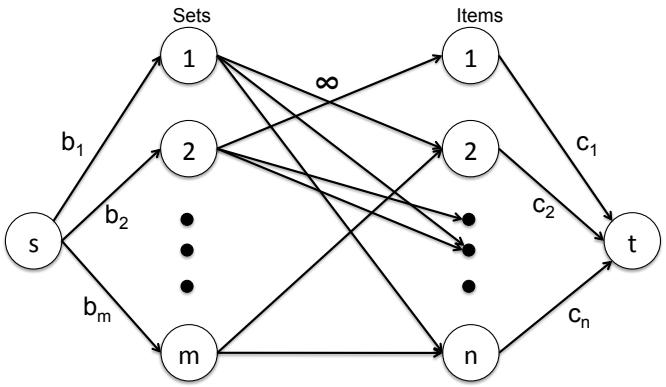


Figure 34: Formulation of selection problem as a min-cut problem

Proof.

$$\begin{aligned}
\min_S U(S,T) &= \min \sum_{i\in T} b_i + \sum_{j\in S} c_j \\
&= \min \sum_{i=1}^{m} b_i - \sum_{i\in S} b_i + \sum_{j\in S} c_j \\
&= B + \min - \sum_{i\in S} b_i + \sum_{j\in S} c_j \\
&= B - \max \sum_{i\in S} b_i - \sum_{j\in S} c_j.
\end{aligned}$$

Where $B = \sum_{i=1}^{m} b_i$. □

Lec10

36 A Production/Distribution Network: MCNF Formulation

36.1 Problem Description

A company manufacturing chairs has four plants located around the country. The cost of manufacture, excluding raw material, per chair and the minimum and maximum monthly production for each plant is shown in the following table.

Table 3: Production facts

		Production	
Plant	Cost per chair	Maximum	Minimum
1	\$5	500	0
2	\$7	750	400
3	\$3	1000	500
4	\$4	250	250

Twenty pounds of wood is required to make each chair. The company obtains the wood from two sources. The sources can supply any amount to the company, but contracts specify that the company must buy at least eight tons of wood from each supplier. The cost of obtaining the wood at the sources is:

- Source 1 = \$0.10/pound
- Source 2 = \$0.075/pound

Shipping cost per pound of wood between each source and each plant is shown in cents by the following matrix:

Table 4: Raw material shipping cost

		Plant			
		1	2	3	4
Wood	1	1	2	4	4
Source	2	4	3	2	2

The chairs are sold in four major cities: New York, Chicago, San Francisco and Austin. Transportation costs between the plants and the cities are shown in the following matrix: (All costs are in dollars per chair.)

Table 5: Product shipping cost

		Cities			
		NY	A	SF	C
Plant	1	1	1	2	0
	2	3	6	7	3
	3	3	1	5	3
	4	8	2	1	4

The maximum and minimum demand and the selling price for the chairs in each city is shown in the following table:

		Demand	
City	Selling price	Maximum	Minimum
New York	\$20	2000	500
Austin	\$15	400	100
San Francisco	\$20	1500	500
Chicago	\$18	1500	500

Table 6: Demand facts

Now, as the manager of the company, you must decide the following:

1. Where should each plant buy its raw materials?
2. How many chairs should be made at each plant?
3. How many chairs should be sold at each city?

4. Where should each plant ship its product?

36.2 Formulation as a Minimum Cost Network Flow Problem

- **Units and upper-bounds**

  The units of arc capacity and cost are not consistent. First we need to convert the units of the arc capacities (upper and lower bounds) into chairs, and convert the units of the arc cost into dollars per chair.

  Also, note that the maximum total production capacity of the four plants is 2500. Thus, if not otherwise specified, the upper-bound for arc capacities can be set to 2500.

- **Nodes**

  Based on the problem description, we have three types of actual nodes in this network: the wood sources ($WS1$ and $WS2$), plants ($P1$,...,$P4$), and demands (NY, Au, SF, Ch). These nodes correspond to actual locations.

  We also need some dummy/artificial nodes to formulate it as a MCNF problem. These nodes include a source node, and a sink node, and dummy nodes used to model the constraints of the flow volume through some of the nodes,

- **Arcs from wood sources to plants**

  These arcs correspond to buying raw materials from wood sources and shipping them to the plants. The upper-bound is 2500, the lower-bound is 0, and the cost is the unit purchasing and shipping cost. The cost is different for each wood sources - plant pair.

- **Arcs from plants to demands**

  These arcs correspond to shipping the chairs from the plants to the demand nodes. The upper and lower bounds are given by the maximum and minimum production volume of the plant from which each arc is originated. The cost is the shipping cost from plants to cities.

- **Constraints of flow volume through the nodes**

  In MNCF problem, we can specify upper and lower bounds on the flows through the arcs. However, we can usually come up with problems that require some constraints on the flow volume through some nodes. For instance, in this problem, the company is required to buy at least eight tons of wood from each supplier; the plants have maximum and minimum production constraints; the cities have maximum and minimum demand levels.

  A trick for modeling node flow volume constraints is splitting the node into two, or, in other words, adding a dummy node. Assume we have a node n. The maximum and minimum flow volume through n is u and l, respectively. We can add a dummy node n', and add an arc between n and n', set the upper and lower bounds of arc (n, n') to u and l, and the unit cost to 0, as shown in Figure 35. The incoming arcs to node n are still incoming to node n, while the outgoing arcs from node n will now be outgoing from node n'.

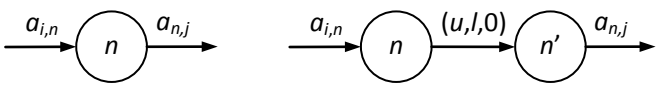


Figure 35: Modeling node flow volume constraints

For this problem, we need to add dummy nodes for the plants ($P1'$,...,$P4'$). The flow volume constraints for wood sources and cities are treated in a similar way. However, no dummy node is needed, because we can link the arc corresponding to each wood source/city to the source/sink nodes.

- **Arcs from the *source* node to the wood sources**

  These arcs specify the minimum amount of raw materials purchased from each wood source. The upper bound is 2500, the lower bound is 800, and the cost is 0.

  The cost is 0 because we have already counted the purchase cost in the arcs from the wood sources to the plants. Otherwise, we can also represent the purchase cost here, and let the arcs from wood sources to the plants only represent the raw material shipping cost.

- **Arcs from the cities to the *sink* node**

  These arcs correspond to selling chairs in each city and earning the selling price. Note that these profits are represented by negative costs.

- **Balancing the entire network: circulation/draining arc**

  So far, we have constructed a network. The upper/lower bounds and unit cost of the arcs in this network guarantee that all the purchasing, shipping, production, distribution, selling requirements are met. However, we haven't specified the amount of total supply and total demand. Without this, there will be no feasible flow to the networks.

  We cannot assign an arbitrary total supply and demand, because this will impose extra constraints, and completely change the problem, even making optimal solutions infeasible. But we cannot decide the total supply and total demand until we know the optimal production/distribution quantities.

  There are two ways to tackle this problem. The first one is by adding an artificial arc from the sink node back to the source node, called the *circulation arc*. This arc has infinite capacity and zero cost. By adding this arc, everything flowing from the source to the sink can flow back to the source without incurring any cost. Thus the entire network is balanced.

  The second method is to assign sufficiently large supply and demand at the source and sink nodes, and to add an artificial arc from the source node to the sink node, called *drainage arc*. The supply and demand must be sufficiently large, so that the optimal solution is guaranteed to be feasible. The drainage arc will deliver the extra supply and demand that do not actually occur. There is no capacity constraint or cost on this arc. Thus the extra supply can flow to the sink and meet the extra demand without incurring any cost.

  See Figure 36 for illustration of circulation and drainage arcs.

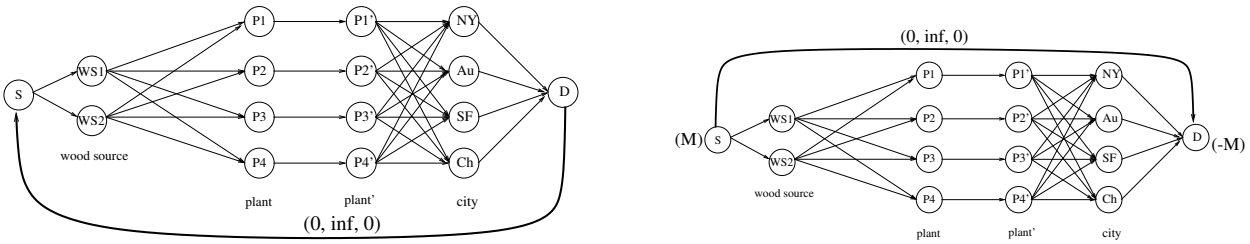
- **Complete MCNF network**

  The complete MCNF network is shown in Figure 37. The circular arc approach is used.

  The parameters of the arcs are summarized in Figure 38

36.3 The Optimal Solution to the Chairs Problem

Solving this MCNF problem, we can get the optimal solutions to the chairs problem, as shown in Figure 39.
The original questions can now be answered in terms of the f_{ij} solution variables.



(a) circulation arc (b) drainage arc

Figure 36: Circulation arc and drainage arc

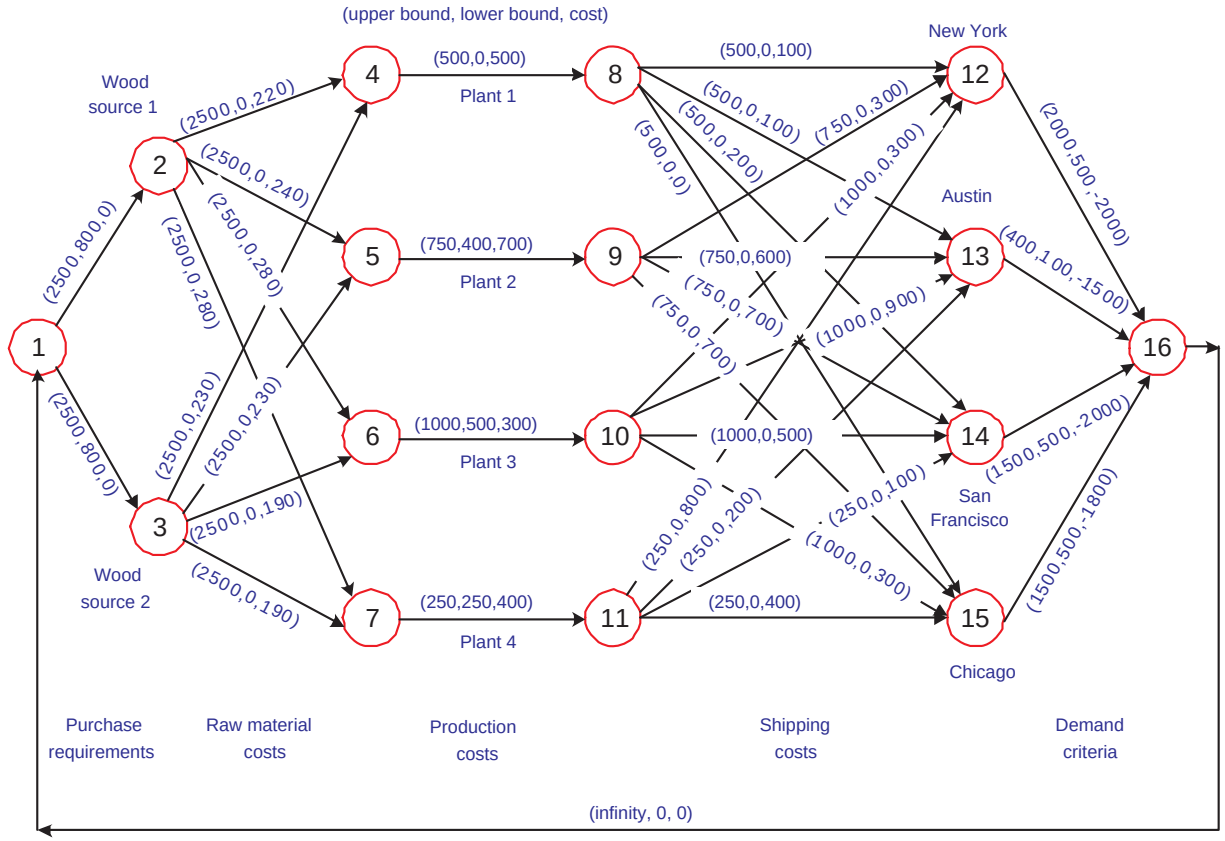


Figure 37: Complete MCNF network for the chairs problem

1. Where should each plant buy its raw materials?

   (a) Plant 1 should buy 500 chairs or 10,000 lb from wood source 1 and 0 chairs or 0 lb from wood source 2.

   (b) Plant 2 should buy 300 chairs or 6000 1b from wood source 1 and 450 chairs or 9000 lb from wood source 2.

   (c) Plant 3 should buy 0 chairs or 0 lb from wood source 1 and 1000 chairs or 20,000 lb from wood source 2.

Table 3.8. NETWORK LABELS

Arc	*From Node*	*To Node*	*Upper Bound*	*Lower Bound*	*Cost*
1	1	2	2500	800	0
2	1	3	2500	800	0
3	2	4	2500	0	220
4	2	5	2500	0	240
5	2	6	2500	0	280
6	2	7	2500	0	280
7	3	4	2500	0	230
8	3	5	2500	0	210
9	3	6	2500	0	190
10	3	7	2500	0	190
11	4	8	500	0	500
12	5	9	750	400	700
13	6	10	1000	500	300
14	7	11	250	250	400
15	8	12	500	0	100
16	8	13	500	0	100
17	8	14	500	0	200
18	8	15	500	0	0
19	9	12	750	0	300
20	9	13	750	0	600
21	9	14	750	0	700
22	9	15	750	0	300
23	10	12	1000	0	300
24	10	13	1000	0	100
25	10	14	1000	0	500
26	10	15	1000	0	300
27	11	12	250	0	800
28	11	13	250	0	200
29	11	14	250	0	100
30	11	15	250	0	400
31	12	16	2000	500	−2000
32	13	16	400	100	−1500
33	14	16	1500	500	−2000
34	15	16	1500	500	−1800
35	16	1	5400	1600	0

Number of nodes = 16
Number of arcs = 35

Figure 38: Arcs of the MCNF network for the chairs problem

(d) Plant 4 should buy 0 chairs or 0 lb from wood source 1 and 250 chairs or 5000 lb from wood source 2.

2. How many chairs should be made at each plant?

   (a) Plant 1 should make 500 chairs.
   (b) Plant 2 should make 750 chairs.
   (c) Plant 3 should make 1000 chairs.
   (d) Plant 4 should make 250 chairs.

3. How many chairs should be sold at each city?

   (a) New York should sell 1400 chairs.
   (b) Austin should sell 100 chairs.
   (c) San Francisco should sell 500 chairs.
   (d) Chicago should sell 500 chairs.

4. Where should each plant ship its product?

   (a) Plant 1 ships 500 chairs to Chicago.

Table 3.9. THE OPTIMAL SOLUTION

Node k	π_k	Arc (i,j)	f_{ij}	Arc (i,j)	f_{ij}
1	1230	1	800	19	750
2	1200	2	1700	20	0
3	1230	3	500	21	0
4	1420	4	300	22	0
5	1440	5	0	23	650
6	1420	6	0	24	100
7	1420	7	0	25	250
8	3230	8	450	26	0
9	2930	9	1000	27	0
10	2930	10	250	28	0
11	3330	11	500	29	250
12	3230	12	750	30	0
13	3030	13	1000	31	1400
14	3430.	14	250	32	100
15	3230	15	0	33	500
16	1230	16	0	34	500
		17	0	35	2500
		18	500		

Figure 39: The optimal solution to the chairs problem

(b) Plant 2 ships 750 chairs to New York.

(c) Plant 3 ships 650 chairs to New York, 100 to Austin, and 250 to San Francisco.

(d) Plant 4 ships 250 chairs to San Francisco.

37 Transhipment Problem

A special case of the MCNF problem is the *Transhipment Problem*, where there is no lower or upper bound on the arcs.

Transhipment Problem Given a network $G = (V, A)$, with a cost c_{ij}, associated with each arc (i, j), and supply b_i at each node $i \in V$. There is no upper or lower bound on the arcs. Find the cheapest integer valued flow such that it satisfies: 1) the supplies/demands at the nodes, 2) that the flow is conserved through the network.

38 Transportation Problem

We are interested in a special case of the transhipment problem, called the *Transportation Problem*. Given a bipartite network $G = (V^1 \cup V^2, A)$, such that $u \in V^1, v \in V^2, \forall (u, v) \in A$. There is a cost c_{ij} associated with each arc $(i, j) \in A$, no lower or upper bound on the arc. Supply $b_i > 0$ for all $i \in V^1$, $b_i < 0$ for all $i \in V^2$. Thus, V^1 can be considered as the suppliers, and V^2 can be considered as customers. There are no transhipment arcs between supplier or customers. The goal is to meet all demand at minimum cost.
The transportation problem can be formulated as an LP as follows. Let s_i be the supply of supplier $i \in V_1$, and d_j the demand of customer $j \in V_2$. Let x_{ij} be the amount of products shipped from

supplier i to customer j.

$$\begin{array}{lll} \min & \sum_{i\in V_1, j\in V_2} c_{i,j}x_{i,j} & \\ \text{s.t.} & \sum_{j\in V_2} x_{i,j} = s_i & \forall i \in V_1 \\ & \sum_{i\in V_1} x_{i,j} = d_j & \forall j \in V_2 \\ & x_{i,j} \geq 0 & \end{array} \tag{39}$$

The optimal solution to the above formulation is guaranteed to be integral, as long as all supplies and demand are integral. This follows from the total unimodularity of the constraint matrix.
It is obvious that the transportation problem is a special case of the transhipment problem. However, the converse is also true. One can show that the transhipment problem is polynomial-time reducible to the transportation problem. We will not give this reduction here.

38.1 Production/Inventory Problem as Transportation Problem

Problem Description Sailco Corporation must determine how many sailboats should be produced during each of the next four quarters.
Demand is as follows: first quarter 40 sailboats; second quarter, 60 sailboats; third quarter, 75 sailboats; fourth quarter, 25 sailboats.
Sailco must meet demand on time.
At the beginning of the first quarter, Sailco has an inventory of 10 sailboats. At the beginning of each quarter, Sailco must decide how many sailboats should he produced during the current quarter. For simplicity, we assume that sailboats manufactured during a quarter can be used to meet demand for the current quarter.
During each quarter, Sailco can produce up to 40 sailboats at a cost of $400 sailboat. By having employees work overtime during a quarter, Sailco can produce additional sailboats at a cost of $450 per sailboat. At the end of each quarter (after production has occurred and the current quarter's demand has been satisfied), a carrying or holding cost of $20 per sailboat is incurred.
Formulate a transportation problem to minimize the sum of production and inventory costs during the next four quarters.

Transportation Problem Formulation In this problem, we need to provide a production schedule, which includes 1) the regular production quantity in each month, and 2) the overtime production quantity in each month. To formulate it as a transportation problem, we need to decompose this production schedule. In each month, for both regular and overtime production quantity, we need to specify the month in which the product will be sold. By doing this, each product we produce will contain the following information: 1) in which month it is produced, 2) whether it is produced with regular or overtime production capacity, and 3) in which month it is sold.
Now, we can construct a transportation network for this problem.

- **Nodes**

  In the transportation problem, the nodes are partitioned into suppliers and customers. In this problem, there are two types of suppliers: 1) regular production in each month ($Q1_P, \ldots, Q4_P$), and 2) overtime production in each month ($Q1_O, \ldots, Q4_O$). The customers are the demands in each month ($Q1_D, \ldots, Q4_D$).

  For the regular production nodes, the supply is simply the regular capacity 40. For the demand nodes, the demand is simply the demand in each month. For the overtime production

nodes, the supply can be set to a sufficiently large number, e.g. sum of all the demands, 200. This upper-bound seems quite loose. But, since the transportation problem can be solved in strongly polynomial time, the upper-bound does not matter.

However, since we have set the supply of overtime production nodes to be sufficiently large, the total amount of supply is now greater than the total amount of demand. To make the network balanced, we need to add a dummy sink node T. The demand of this dummy sink is exactly the difference between total supply and total demand, $40 \times 4 + 200 \times 4 - 200 = 760$. All the extra production capacity will be consumed by this dummy sink.

- **Arcs**

  An arc (Qi_P, Qj_D) corresponds to producing with regular capacity in the i-th month to satisfy the demand in the j-th month. An arc (Qi_O, Qj_D) corresponds to the overtime case. Since we can only satisfy the demand in the current month or in the future, $j \geq i$ for all the arcs. Also, since overtime production has infinite capacity, overtime production will only be used to satisfy demand in the current month. We will never produce some products by overtime, and then keep them in inventory.

  The cost of an arc corresponds to the total production and inventory holding cost. For instance, consider arc (Qi_O, Qj_D), for some $j > i$. Each product that flows in this arc is produced with overtime capacity, thus incurring production cost \$450, and held in inventory through month $i, \ldots, j-1$, incurring inventory holding cost \$20 in each of these months, $\$20(j-i)$ in total.

  We also need to add an arc from each production node to the dummy sink node to balance the total supply and demand. Since these production did not actually take place, the cost should be zero.

- **Complete network**

  The complete network is shown in Figure 40 and Table 7.

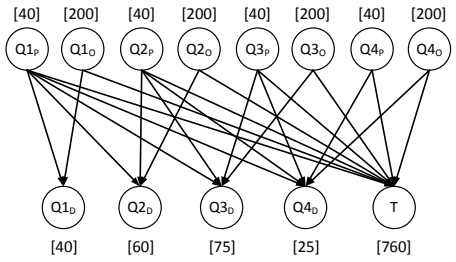


Figure 40: The transportation problem formulation for the production/inventory problem

Table 7: Arcs in the production/inventory problem

Arc		Cost	Arc		Cost	Arc		Cost
1_P	1_D	400	2_P	3_D	420	3_O	3_D	450
1_P	2_D	420	2_P	4_D	440	3_O	T	0
1_P	3_D	440	2_P	T	0	4_P	4_D	400
1_P	4_D	460	2_O	2_D	450	4_P	T	0
1_P	T	0	2_O	T	0	4_O	4_D	450
1_O	1_D	450	3_P	3_D	400	4_O	T	0
1_O	T	0	3_P	4_D	420			
2_P	2_D	400	3_P	T	0			

39 Assignment Problem

Now we will look at a special case of the transportation problem, the *Assignment Problem*. The assignment problem appears in many other problems as part of the constraints, such as the TSP.

Problem Description Given n jobs and n machines, cost c_{ij} for assigning job i to machine j. Find the minimum cost assignment such that each job is assigned to one machine, and each machine is assigned with one job.
The assignment problem is a transportation problem with $|V_1| = |V_2|$, i.e. the number of suppliers is equal to the number of customers, and $b_i = 1, \forall i \in V_1, b_i = -1, \forall i \in V_2$.

40 Maximum Flow Problem

Maximum Flow Problem Given a directed network $G = (V, A)$ with capacities u_{ij} on arc (i, j), and no cost. Two nodes are specified, a source node s, and sink node t. The objective is to find the maximum flow possible from the source to the sink while satisfying the arc capacities. If f_{ij} represents the flow on arc (i, j), and A the set of arcs in the graph, the problem can be formulated as:

$$(\textit{max flow}) \qquad \begin{array}{ll} \max & V \\ \text{subject to} & \sum_{(s,i)\in A} f_{si} = V \\ & \sum_{(j,t)\in A} f_{jt} = V \\ & \sum_{(i,k)\in A} f_{ik} - \sum_{(k,j)\in A} f_{kj} = 0, \qquad \forall k \neq s, t \\ & 0 \leq f_{ij} \leq u_{ij}, \quad \forall (i,j) \in A. \end{array}$$

40.1 A Package Delivery Problem

Problem Description Seven types of packages are to be delivered by five trucks. There are three packages of each type, and the capacities of the five trucks are 6, 4, 5, 4, and 3 packages, respectively.
Set up a maximum-flow problem that can be used to determine wether the packages can be loaded so that no truck carries two packages of the same type.

Maximum flow problem formulation We now construct a network to formulate the problem as *max flow*. We consider the packages as particles flowing in this network.

- **Nodes**

  The nodes include 1) seven nodes representing different package types $P_1, \ldots, P_7$; 2) five nodes representing the trucks $T_1, \ldots, T_5$; and 3) the source s and sink t.

- **Arcs**

  (s, P_i): an arc from the source s to type i package specifies the number of type i packages to be delivered. For this problem, every arc of this kind has an upper-bound equal to 3.

  (P_i, T_j): an arc from type i package to truck j specifies the maximum number packages of type i that could be delivered by truck j. For this problem, every arc of this kind has upper-bound equal to 1, since no truck can carry two packages of the same type.

  (T_j, t): an arc from truck j to the sink t specifies the capacity of the truck. For this problem, the corresponding upper-bounds are 6, 4, 5, 4, and 3 packages, respectively.

- **Complete network**

  The complete network is shown in Figure 41.

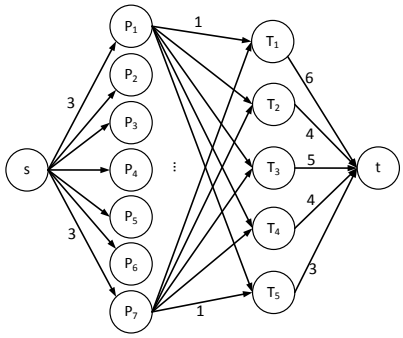


Figure 41: Max flow formulation for the package delivery problem

- **Solution**

  The packages can be loaded in a way such that no truck carries two packages of the same type, *iff* the maximum flow of the network shown in Figure 41 is 21 (total number of packages).

41 Shortest Path Problem

Shortest Path Problem Given a directed, weighted graph, where the weights d_{ij} may be thought of as distances. The objective is to find a path from a source node, s, to a sink node, t, that minimizes the sum of weights along the path. The shortest path problem can be viewed as a MCNF problem with $b_s = 1$ and $b_t = -1$, costs the same as the distances, and no upper bound and zero lower bound on arcs. Let x_{ij} be 1 if the arc (i, j) is in the path and 0 otherwise (x_{ij} can also be viewed as the flow along arc (i, j)).

$$
(SP) \qquad \begin{array}{ll} \min & \sum_{((i,j)\in A} d_{ij} x_{ij} \\ \text{subject to} & \sum_{(i,k)\in A} x_{ik} - \sum_{(j,i)\in A} x_{ji} = 0 \qquad \forall i \neq s, t \\ & \sum_{(s,i)\in A} x_{si} = 1 \\ & \sum_{(j,t)\in A} x_{jt} = 1 \\ & 0 \leq x_{ij} \leq 1 \end{array}
$$

41.1 An Equipment Replacement Problem

Problem Description A single machine is needed to perform a specified function for the next four years, after which the function and machine will no longer be needed. The purchase price of a machine varies over the next four years according to the following table.

Year	Now	One Year From now	Two Years From now	Three Years From now
Purchase price	\$25,000	\$33,000	\$38,000	\$47,000

The salvage value of a machine depends only on its length of service and is given by the following table.

Length of Service	One Year	Two Years	Three Years	Four Years
Salvage Value	\$17,000	\$6,000	\$3,000	\$1,000

The annual operating cost varies with length of service, as follows.

Length of Service	New	One Year	Two Years	Three Years
Annual operating cost	\$3,000	\$5,000	\$8,000	\$18,000

Construct a network in which the shortest path will yield an optimal policy of purchasing, operating, and salvaging machines over the next four years if management wishes to minimize the total cost.

Shortest path problem formulation We now construct a network to formulate the equipment replacement problem as a shortest path problem.

- **Nodes**

  There are five nodes in the network. The first four represent the four years, and the fifth one represents the end of the planning horizon.

- **Arcs**

  An arc (i, j) means buying a new machine at the beginning of the i-th year, and selling it at end of $(j-1)$-th year. The cost for this arc is 1) the purchase price in year i, plus 2) the maintaining cost through the $j-i$ years when this machine is used, minus 3) the salvage value after $j-i$ years.

- **Complete network**

  The complete network is shown in Figure 42.

- **Solution**

  Solving the shortest path problem, if the solution is a path $p = (1, v_1, \ldots, v_k, 5)$, then the company should buy a new machine at the beginning of year $1, \ldots, v_k$, and sell it at the end of year $v_1 - 1, \ldots, v_k - 1, 5$.

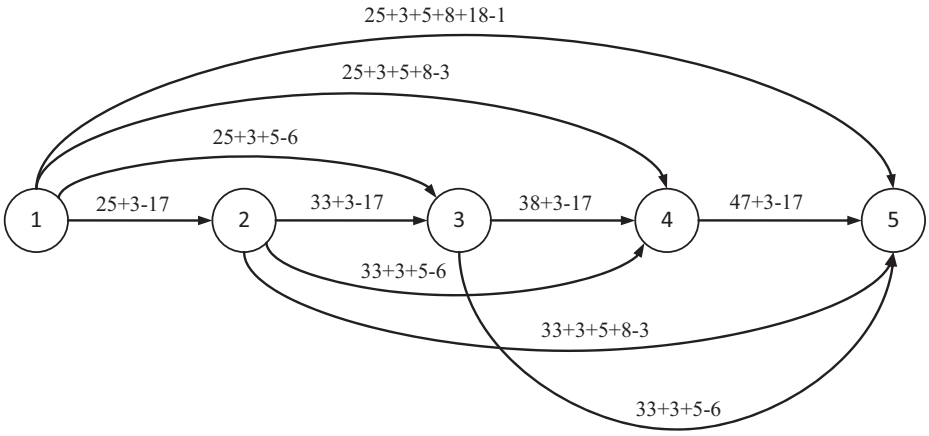


Figure 42: Shortest path formulation for the equipment replacement problem

42 Maximum Weight Matching

Maximum weight matching Given a graph $G = (V, E)$, a set of edges M is a matching if each node has at most one edge in M adjacent to it. The *Maximum Weight Matching Problem* is to find in a edge-weighted graph the matching of maximum weight.

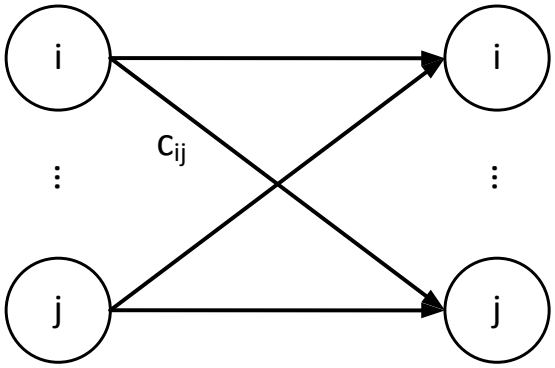
Bipartite matching The *Bipartite Matching Problem* is a special case of the maximum matching problem, where we are given a bipartite graph $G = (V_1 \cup V_2, E)$. Let $x_{ij} = 1$, if edge $(i, j) \in M$; $x_{ij} = 0$, otherwise. The formulation for the problem is

$$\begin{array}{lll} & \max & \sum_{(i,j)\in E} w_{ij}x_{ij} \\ (\textit{bipartite matching}) & \text{subject to} & \sum_{(i,j)\in E} x_{ij} \leq 1 \qquad \forall i \in V \\ & & 0 \leq x_{ij} \leq 1, \quad \forall\ (i,j) \in E. \end{array}$$

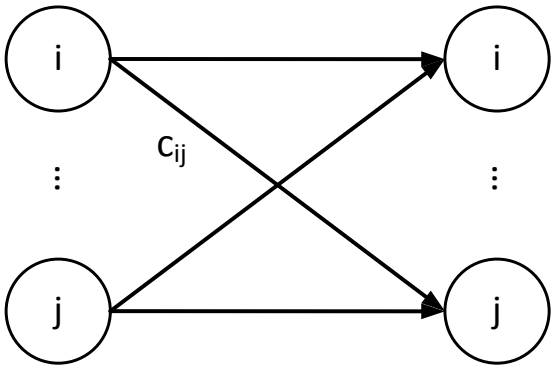
We can show that the assignment problem is a special case of bipartite matching. Given an assignment problem with n jobs and n machines, cost c_{ij}, we can construct a bipartite graph, of which the maximum weight matching gives the optimal assignment. Let $w_{ij} = M - c_{ij}$, where M is a sufficiently large number so that $w_{ij} > 0$ for all (i, j), as shown in Figure 43.
Since M is sufficiently large, such that $M - c_{ij} > 0$ for all (i, j), the maximum weight matching must contain n edges. Thus, the maximum weight matching M is a feasible assignment. Also, every feasible assignment is a matching. Maximizing $\sum_{(i,j)\in M}(M - c_{ij})$ is equivalent to minimizing $\sum_{(i,j)\in M} c_{ij})$. Thus the maximum weight matching in the constructed bipartite graph is an optimal solution to the original assignment problem.

42.1 An Agent Scheduling Problem with Reassignment

Problem Description The table below shows the required start and end times for ten tasks, as well as the minimum time it takes to reassign to task j an agent who has just completed task i.
The problem is to find the minimum number of agents required to complete the 10 tasks with no deviation from the schedule and find a schedule for each individual.



(a) assignment problem



(b) bipartite matching

Figure 43: Assignment problem and bipartite matching

Bipartite matching formulation We can use bipartite matching to solve this problem. To do this wee need to construct the following bipartite graph.

- **Nodes**

  We have two sets of nodes V_1 and V_2. They both contain 10 nodes representing the 10 tasks. Node $i \in V_1$ represents the starting of task i. Node $i' \in V^2$ represents the completion of task i

- **Arcs**

  In this network, we only have arcs from V_2 to V_1. There is an arc from $i' \in V^2$ to $j \in V^1$, if an agent can be reassigned to task j after completing task i, i.e. $e_i + R_i j \leq b_j$.

  For instance, consider the first 3 tasks in Table 8. An agent can be reassigned to task 2 or 3 after completing task 1, and to task 3 after completing task 2. The corresponding network is shown in

- **Solution**

  The maximum cardinality bipartite matching provides a schedule with the minimum number of agents.

  This follows from the fact that each arc in the graph corresponds to a reassignment of agents to tasks, and minimizing the number of agents required is equivalent to maximizing the number

Table 8: Task start/end and reassignment time

i	b_i	e_i	R_{ij}: Reassignment time from task i to task j (minutes)									
Task	**Start**	**End**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
1	1:00 p.m.	1:30 p.m.	----	60	10	230	180	20	15	40	120	30
2	6:00 p.m.	8:00 p.m.	10	----	40	75	40	5	30	60	5	15
3	10:30 p.m.	11:00 p.m.	70	30	----	0	70	30	20	5	120	70
4	4:00 p.m.	5:00 p.m.	0	50	75	----	20	15	10	20	60	10
5	4:00 p.m.	7:00 p.m.	200	240	150	70	----	15	5	240	90	65
6	12:00 p.m	1:00 p.m.	20	15	20	75	120	----	30	30	15	45
7	2:00 p.m.	5:00 p.m.	15	30	60	45	30	15	----	10	5	0
8	11:00 p.m.	12:00 a.m.	20	35	15	120	75	30	45	----	20	15
9	8:10 p.m.	9:00 p.m.	25	60	15	10	100	70	80	60	----	120
10	1:45 p.m.	3:00 p.m.	60	60	30	30	120	40	50	60	70	----

of reassignments. Also, a feasible schedule must be a matching, because after completing a task, an agent can at most be reassigned to one task, and any task can at most have one agent assigned to it.

Given the maximum bipartite matching M, we can obtain the optimal schedule in the following way:

1. Identify the nodes in V_1 that are not matched (have no edge in M adjacent to it), let the set of these nodes be V_N.
2. For each node in V_N, we will have a new agent assigned to this task (i.e. the agent is not reassigned from somewhere else). $|V_N|$ is the minimum number of agents required.
3. For each agent, assume it begins with task $i \in V_N$, then the next task is task j, such that arc $(i', j) \in M$. Now, task j is the agent's current task, and we can find the next task in the same way. If arc $(j', k) \notin M$ for all $k \in V_1$, i.e. the agent is not reassigned to any task after completing task j, then task j is this agent's last task. We have found the complete schedule for the agent who starts with task i.

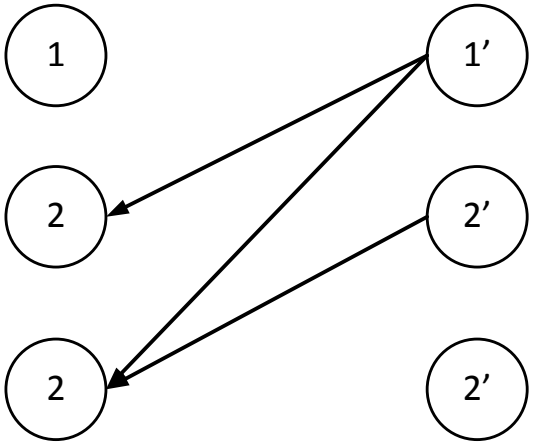


Figure 44: Bipartite matching formulation for the agent scheduling problem with reassignment

43 MCNF Hierarchy

The following diagram summarizes the relationship among all the problems we have mentioned so far. Note that the assignment problem is a special case of both the transportation problem and the maximum weights bipartite matching.

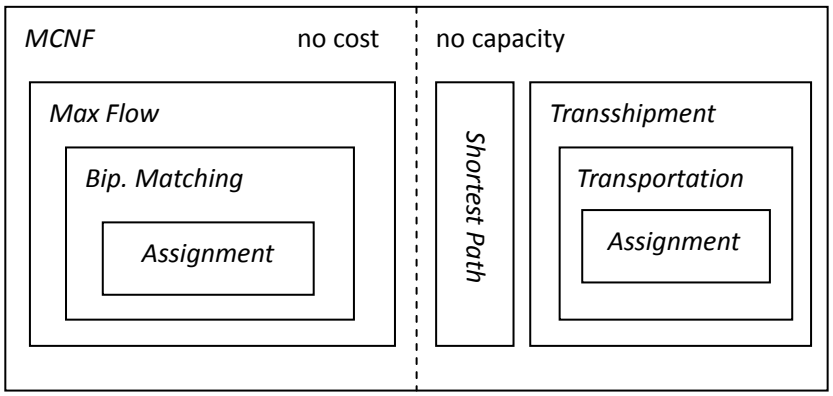


Figure 45: MCNF Hierarchy

Lec11

44 The maximum/minimum closure problem

44.1 A practical example: open-pit mining

Open-pit mining is a surface mining operation in which blocks of earth are extracted from the surface to retrieve the ore contained in them. During the mining process, the surface of the land is

being continuously excavated, and a deeper and deeper pit is formed until the operation terminates. The final contour of this pit mine is determined before mining operation begins. (Figure 44.1) To

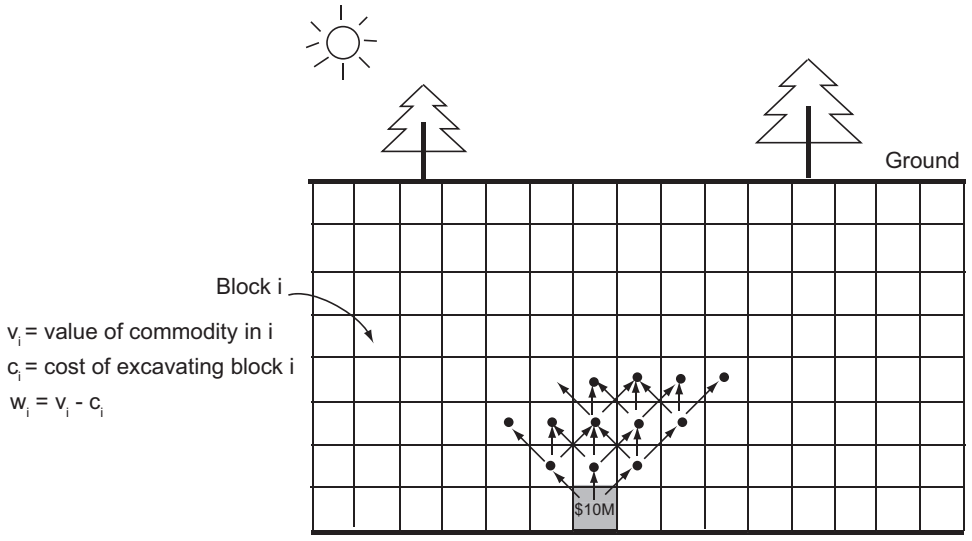


Figure 46: Open pit mining

design the optimal pit – one that maximizes profit, the entire area is divided into blocks, and the value of the ore in each block is estimated by using geological information obtained from drill cores. Each block has a weight associated with it, representing the value of the ore in it, minus the cost involved in removing the block. While trying to maximize the total weight of the blocks to be extracted, there are also contour constraints that have to be observed. These constraints specify the slope requirements of the pit and precedence constraints that prevent blocks from being mined before others on top of them. Subject to these constraints, the objective is to mine the most profitable set of blocks.

The problem can be represented on a directed graph $G = (V, A)$. Each block i corresponds to a node with weight w_i representing the net value of the individual block. The net value w_i is computed as the assessed value v_i of the ore in that block, from which the cost c_i of excavating that block alone is deducted. There is a directed arc from node i to node j if block i cannot be excavated before block j which is on a layer right above block i. This precedence relationship is determined by the engineering slope requirements. Suppose block i cannot be excavated before block j, and block j cannot be excavated before block k. By transitivity this implies that block i cannot be excavated before block k. We choose in this presentation not to include the arc from i to k in the graph and the existence of a directed path from i to k implies the precedence relation. Including only arcs between immediate predecessors reduces the total number of arcs in the graph. Thus to decide which blocks to excavate in order to maximize profit is equivalent to finding a maximum weighted set of nodes in the graph such that all successors of all nodes are included in the set.

Notice that the problem is trivial if $w_i \leq 0, \forall i \in V$ (in which no block would be excavated) or if $w_i \geq 0, \forall i \in V$ (in which all the blocks would be excavated).

Let $x_i = \begin{cases} 1 & \text{if block } i \text{ is selected} \\ 0 & \text{otherwise} \end{cases}$.

Then the open pit mining problem can be formulated as follows:

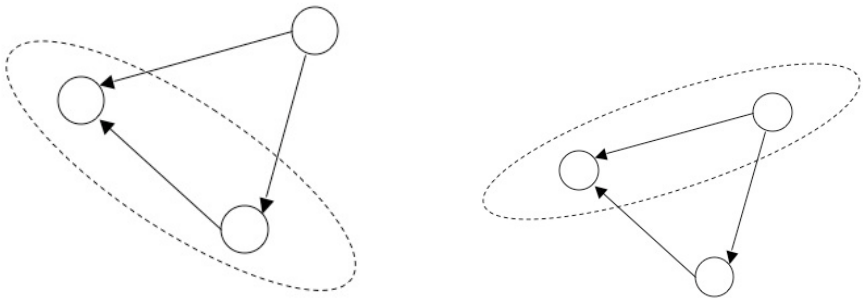
$$\begin{aligned} \max \quad & \sum_{i \in V} w_i x_i \\ \text{s.t.} \quad & x_i \leq x_j \quad \forall (i,j) \in A \\ & 0 \leq x_i \leq 1 \quad \forall i \in V \end{aligned}$$

Notice that each inequality constraint contains exactly one 1 and one -1 in the coefficient matrix. The constraint matrix is totally unimodular. Therefore, we do not need the integrality constraints. The following website `http://riot.ieor.berkeley.edu/riot/Applications/OPM/OPMInteractive.html` offers an interface for defining, solving, and visualizing the open pit mining problem.
The open-pit mining problem is a special case of the *maximum closure problem.* The next subsection discusses the maximum closure problem in detail.

44.2 The maximum closure problem

We begin by defining a closed set on a graph.

Definition 44.1. *Given a directed graph $G = (V, A)$, a subset of the nodes $D \subseteq V$ is* closed*, if for every node in D, its successors are also in D.*



(a) Closed set (b) Not a closed set

Figure 47: Closed set v.s. unclosed set

Consider a directed graph $G = (V, A)$ where every node $i \in V$ has a corresponding weight w_i. The *maximum closure problem* is to find a closed set $V' \subseteq V$ with maximum total weight. That is, the maximum closure problem is:

Problem Name: *Maximum closure*

Instance: *Given a directed graph $G = (V, A)$, and node weights (positive or negative) w_i for all $i \in V$.*

Optimization Problem: *find a closed subset of nodes $V' \subseteq V$ such that $\sum_{i \in V'} w_i$ is maximum.*

We can formulate the maximum closure problem as an integer linear program (ILP) as follows.

$$\begin{aligned} \max \quad & \sum_{i \in V} w_i x_i \\ \text{s.t.} \quad & x_i \leq x_j \quad \forall (i,j) \in A \\ & x_i \in \{0,1\} \quad \forall i \in V \end{aligned}$$

where x_i is a binary variable that takes the value 1 if node i is in the maximum closure, and 0 otherwise. The first set of constraints imposes the requirement that for every node i included in the set, its successor is also in the set. Observe that since every row has exactly one 1 and one -1, the constraint matrix is totally unimodular (TUM). Therefore, its linear relaxation formulation results in integer solutions. Specifically, this structure also indicates that the problem is the dual of a flow problem.

Johnson [14] seems to be the first researcher who demonstrated the connection between the maximum closure problem and the selection problem (i.e., maximum closure on bipartite graphs), and showed that the selection problem is solvable by max flow algorithm. Picard [22], demonstrated that a minimum cut algorithm on a related graph, solves the maximum closure problem.

Let $V^+ \equiv \{j \in V | w_i > 0\}$, and $V^- \equiv \{i \in V | w_i \leq 0\}$. We construct an s,t-graph G_{st} as follows. Given the graph $G = (V, A)$ we set the capacity of all arcs in A equal to ∞. We add a source s, a sink t, set A_s of arcs from s to all nodes $i \in V^+$ (with capacity $u_{s,i} = w_i$), and set A_t of arcs from all nodes $j \in V^-$ to t (with capacity $u_{j,t} = |w_j| = -w_j$). The graph $G_{st} = \{V \cup \{s,t\}, A \cup A_s \cup A_t\}$ is a *closure graph* (a closure graph is a graph with a source, a sink, and with all finite capacity arcs adjacent only to either the source or the sink.) This construction is illustrated in Figure 48.

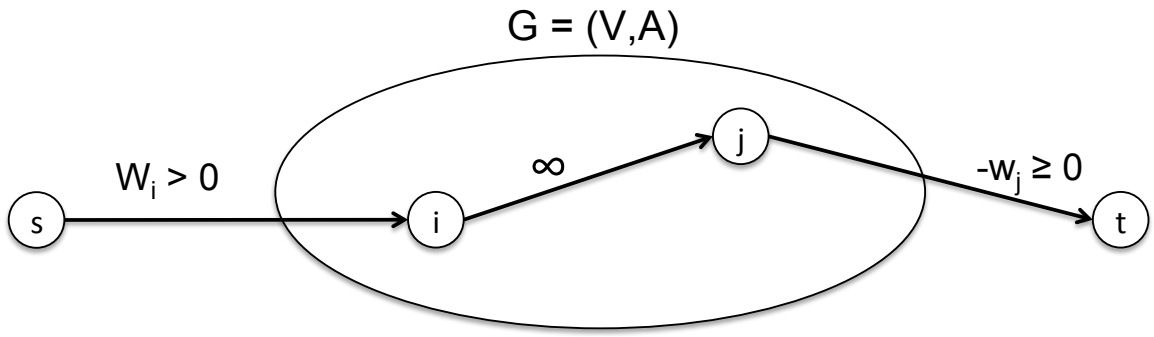


Figure 48: Visual representation of G_{st}.

Claim 44.2. *If $(s \cup S, t \cup T)$ is a finite $s-t$ cut on $G_{s,t}$, then S is a closed set on G.*

Proof. Assume by contradiction that S is not closed. This means that there must be an arc $(i,j) \in A$ such that $i \in S$ and $j \in T$. This arc must be on the cut (S,T), and by construction $u_{i,j} = \infty$, which is a contradiction on the cut being finite. □

Theorem 44.3. *If $(s \cup S, t \cup T)$ is an optimal solution to the minimum $s-t$ cut problem on $G_{s,t}$, then S is a maximum closed set on G.*

Proof.

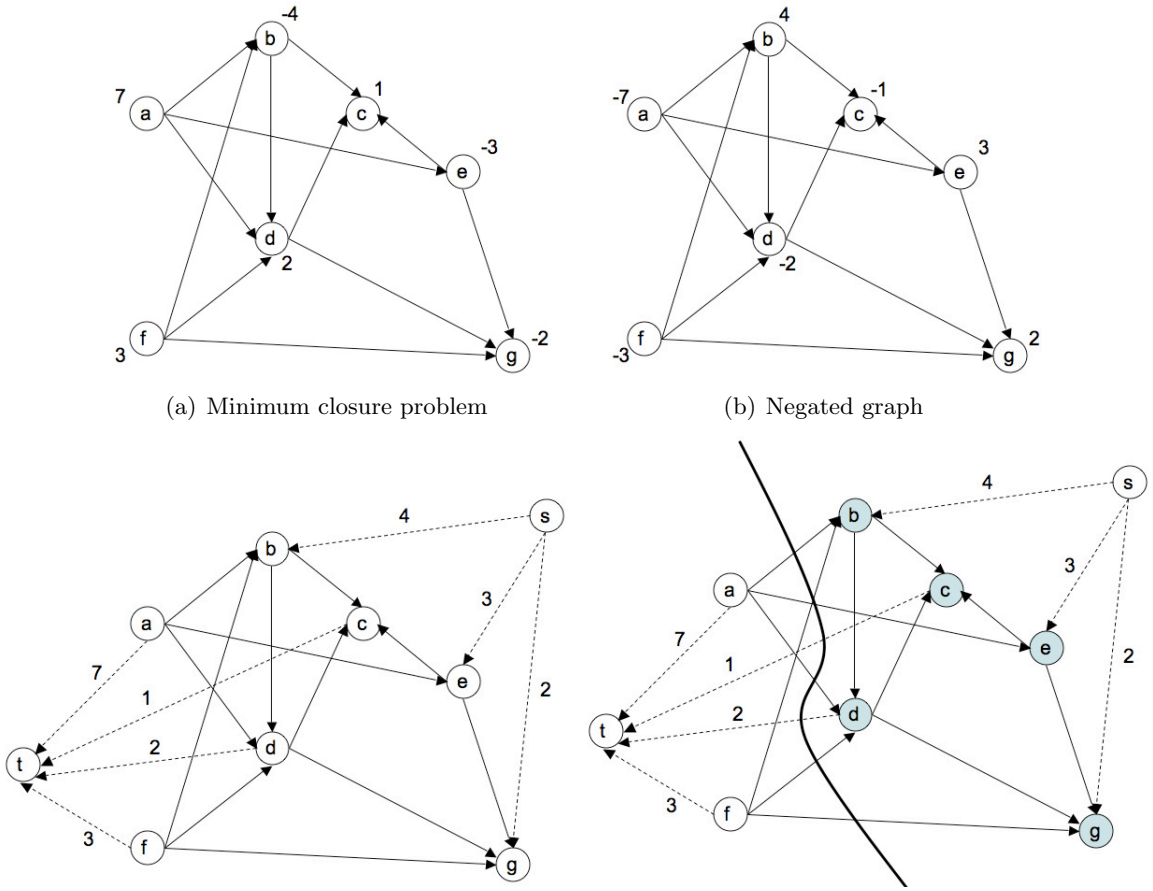
$$\begin{aligned} C\left(s \cup S, t \cup T\right) &= \sum_{(s,i)\in A_{st},\ i\in T} u_{s,i} + \sum_{(j,t)\in A_{st},\ j\in S} u_{j,t} \\ &= \sum_{i\,\in\, T\cap V^+} w_i + \sum_{j\,\in\, S\cap V^-} -w_j \\ &= \sum_{i\in V^+} w_i - \sum_{i\in S\cap V^+} w_i - \sum_{j\in S\cap V^-} w_j \\ &= W^+ - \sum_{i\in S} w_i \end{aligned}$$

(Where $W^+ = \sum_{i\in V^+} w_i$, which is a constant.) This implies that minimizing $C\left(s \cup S, t \cup T\right)$ is equivalent to minimizing $W^+ - \sum_{i\in S} w_i$, which is in turn equivalent to $\max_{S\subseteq V} \sum_{i\in S} w_i$. Therefore, any source set S that minimizes the cut capacity also maximizes the sum of the weights of the nodes in S. Since by Claim 44.2 any source set of an $s-t$ cut in $G_{s,t}$ is closed, we conclude that S is a maximum closed set on G. □

Variants/Special Cases

- In the minimum closure problem we seek to find a closed set with minimum total weight. This can be solved by negating the weights on the nodes in G to obtain G^-, constructing $G^-_{s,t}$ just as before, and solving for the maximum closure. Under this construction, the source set of a minimum $s-t$ cut on $G_{s,t}$ is a minimum closed set on G. See Figure 49 for a numerical example.
- The selection problem is a special case of the maximum closure problem.

Numerical Example



(a) Minimum closure problem

(b) Negated graph

(c) Modified (max flow) graph

(d) Source of min cut = min closure set on G

Figure 49: Converting a minimum closure problem to a maximum closure problem. Under this transformation, the source set of a minimum cut is also a minimum closed set on the original graph.

45 Integer programs with two variables per inequality

45.1 Monotone IP2

For now, let's focus our attention on integer programming with monotone inequalities (Monotone IP2).

Definition 45.1. *An inequality is said to be* monotone *if it is of the form* $ax - by \geq c$ *or* $ax - by \leq c$, *where* x, y *are variables and* a *and* b *of the same sign.*

A typical Monotone IP2 is shown below:

$$\begin{aligned} \text{(Monotone IP2)} \quad \max \quad & \sum_{i \in V} w_i x_i \\ \text{s.t.} \quad & a_{ij} x_i - b_{ij} x_j \geq c_{ij} \quad \forall\ (i,j) \in E \\ & l_i \leq x_i \leq u_i,\ \textit{integer} \quad \forall\ i \in V \end{aligned}$$

where $a_{ij} \geq 0, b_{ij} \geq 0, \forall (i,j) \in E$.

Note that assuming a_{ij}, b_{ij} in all monotone inequalities of the problem to be positive and the inequality relationship to be "$\geq$" does not impair the generality of Monotone IP2. It is easy to see that monotone inequality $ax - by \leq c$ can be transformed into $by - ax \geq -c$, and monotone inequality $ax - by \geq c$ with $a \leq 0, b \leq 0$ can be transformed into $(-b)y - (-a)x \geq c$ with $-b \geq 0, -a \geq 0$.

Monotone IP2 is *NP*-hard. However, as in the case of the knapsack problem, we can give an algorithm that depends on the "numbers" in the input (recall that knapsack can be solved in $O(nB)$ time, where B is the size of the knapsack)—these algorithms are sometimes referred to as pseudo-polynomial time algorithms. For this purpose we next describe how to solve Monotone IP2 as a maximum closure problem.

We first make a change of variables in our original problem. In particular we write variable x_i as a summation of binary variables $x_i = l_i + \sum_{p=l_i+1}^{u_i} x_i^{(p)}$, and impose the restriction that $x_i^{(p)} = 1 \implies x_i^{(p-1)} = 1$ for all $p = l_i + 1, \ldots, u_i$.

With this change of variables we rewrite Monotone IP2 as follows:

$$\max \quad \sum_{i \in V} w_i l_i + \sum_{i \in V} w_i \sum_{p=l_i+1}^{u_i} x_i^{(p)} \tag{40a}$$

$$\text{s.t.} \quad x_j^{(p)} \leq x_i^{(q(p))} \quad \forall\ (i,j) \in E \quad \text{for } p = l_j + 1, \ldots, u_j \tag{40b}$$

$$x_i^{(p)} \leq x_i^{(p-1)} \quad \forall\ i \in V \quad \text{for } p = l_i + 1, \ldots, u_i \tag{40c}$$

$$x_i^{(p)} \in \{0,1\} \qquad \forall\ i \in V \quad \text{for } p = l_i + 1, \ldots, u_i, \tag{40d}$$

where $q(p) \equiv \left\lceil \frac{c_{ij} + b_{ij} p}{a_{ij}} \right\rceil$.

Inequality (40c) guarantees the restriction that $x_i^{(p)} = 1 \implies x_i^{(p-1)} = 1$ for all $p = l_i + 1, \ldots, u_i$; Inequality (40b) follows from the monotone inequalities in the original problem. In particular, for any monotone inequality $a_{ij} x_i - b_{ij} x_j \geq c_{ij}$ with $a_{ij}, b_{ij} \geq 0$, $a_{ij} x_i - b_{ij} x_j \geq c_{ij} \iff x_i \geq \frac{c_{ij} + b_{ij} x_j}{a_{ij}} \overset{x_i \text{ integer}}{\implies} x_i \geq \left\lceil \frac{c_{ij} + b_{ij} x_j}{a_{ij}} \right\rceil$. Equivalently, if $x_j \geq p$, we must have $x_i \geq q(p) = \left\lceil \frac{c_{ij} + b_{ij} p}{a_{ij}} \right\rceil$. In terms of the newly defined binary variables, this is further equivalent to $x_j^p = 1 \implies x_i^{q(p)} = 1$, i.e., $x_j^{(p)} \leq x_i^{(q(p))}$.

Now it's obvious that Monotone IP2 is the maximum closure problem of an s,t-Graph G_{st} defined as follows. First still define $V^+ \equiv \{i \in V | w_i > 0\}$ and $V^- \equiv \{j \in V | w_j \leq 0\}$ as before, and the sets of nodes and arcs are constructed below.

Set of nodes:

Add a source s, a sink t, and nodes $x_i^{(l_i)}, x_i^{(l_i+1)}, \ldots, x_i^{(u_i)}$ for each $i \in V$.

Set of arcs:

1) For any $i \in V^+$, connect s to $x_i^{(p)}, p = l_i + 1, \ldots, u_i$ by an arc with capacity w_i.

2) For any $i \in V^-$, connect $x_i^{(p)}, p = l_i + 1, \ldots, u_i$ to t by an arc with capacity $|w_i| = -w_i$.

3) For any $i \in V$, connect $x_i^{(p)}$ to $x_i^{(p-1)}$, $p = l_i + 1, \ldots, u_i$, by an arc with capacity ∞, and connect s to $x_i^{(l_i)}$ with an arc with capacity ∞.

4) For any $(i,j) \in E$, connect $x_j^{(p)}$ to $x_i^{q(p)}$ by an arc with capacity ∞ for all $p = l_i + 1, \ldots, u_j$. (Note that for situations where $q(p) > u_i$, we must have $x_j^{(p)} = 0$. Therefore, we can either remove the node $x_j^{(p)}$ by redefining a tighter upper bound for x_j, or simply fix $x_j^{(p)}$ to be zero by introducing an arc from $x_j^{(p)}$ to t with capacity ∞.)

This construction is illustrated in Figure 50.

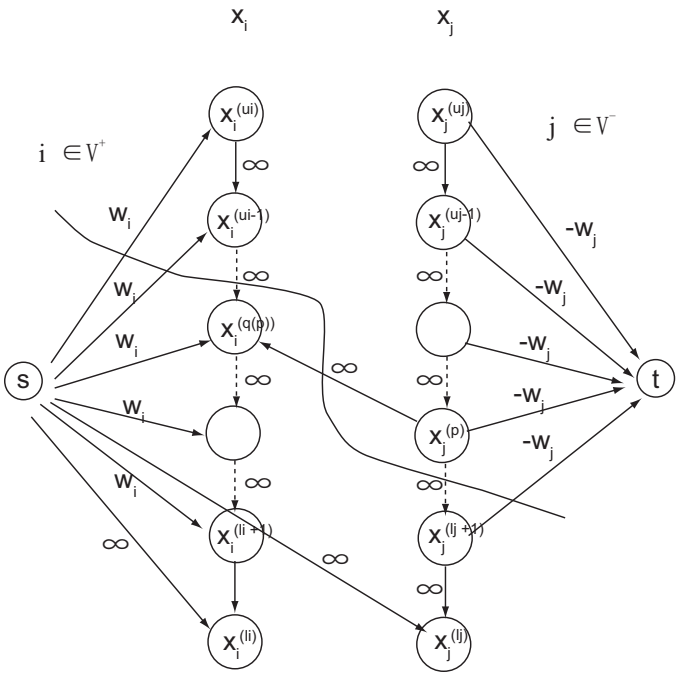


Figure 50: Illustration of G_{st} for Monotone IP2 and example of a finite $s-t$ cut

Remarks:

- It should be noted that the maximum closure problem is defined on an s,t-Graph with $2+\sum_{i\in V}(u_i-l_i)$ nodes. The size of the graph is not a polynomial function of the length of the input. Therefore, the original Monotone IP2 is weakly NP-hard and can be solved by pseudo-polynomial time algorithm based on the construction of a maximum closure problem.
- For the min version of Monotone IP2, we can construct the s,t-Graph in the same way and define closure with respect to the sink set instead.

45.2 Non–monotone IP2

Now let's look at the non–monotone integer program with two variables per inequality:

$$\begin{aligned} \text{(IP2)}\ \min \quad & \sum_{i\in V} w_i x_i \\ \text{s.t.} \quad & a_{ij}x_i + b_{ij}x_j \geq c_{ij} \quad \forall\ (i,j)\in E \\ & l_i \leq x_i \leq u_i,\ \textit{integer} \quad \forall\ i\in V \end{aligned}$$

This problem is more general than Monotone IP2, as we don't impose any restrictions on the signs of a_{ij} and b_{ij}. The problem is clearly *NP*-hard, since vertex cover is a special case. We now give a 2-approximation algorithm for it.

We first "monotonize" IP2 by replacing each variable x_i in the objective by $x_i = \frac{x_i^+ - x_i^-}{2}$ where $l_i \leq x_i^+ \leq u_i$ and $-u_i \leq x_i^- \leq -l_i$, and each inequality $a_{ij}x_i + b_{ij}x_j \geq c_{ij}$ by the following two

inequalities:

$$a_{ij}x_i^+ - b_{ij}x_j^- \leq c_{ij}$$
$$-a_{ij}x_i^- + b_{ij}x_j^+ \leq c_{ij}$$

With such transformations we get the following problem:

$$\begin{aligned}
\text{(IP2') min} \quad & \frac{1}{2}\left(\sum_{i\in V} w_i x_i^+ + \sum_{i\in V}(-w_i)x_i^-\right) \\
\text{s.t.} \quad & a_{ij}x_i^+ - b_{ij}x_j^- \leq c_{ij} \quad \forall\ (i,j)\in E \\
& -a_{ij}x_i^- + b_{ij}x_j^+ \leq c_{ij} \quad \forall\ (i,j)\in E \\
& l_i \leq x_i^+ \leq u_i,\ \textit{integer} \quad \forall\ i\in V \\
& -u_i \leq x_i^- \leq -l_i,\ \textit{integer} \quad \forall\ i\in V
\end{aligned}$$

Let's examine the relationship between IP2 and IP2'. Given any feasible solution $\{x_i\}_{i\in V}$ for IP2, we can construct a corresponding feasible solution for IP2' with the same objective value, by letting $x_i^+ = x_i, x_i^- = -x_i, i \in V$; On the other hand, given any feasible solution $\{x_i^+, x_i^-\}_{i\in V}$ for IP2', $\{x_i = \frac{x_i^+ - x_i^-}{2}\}_{i\in V}$ satisfies the inequality constraint in IP2 (To see this, simply sum up the two inequality constraints in IP2' and divide the resulted inequality by 2.) but may not be integral. Therefore, IP2' provides a lower bound for IP2.
Since IP2' is a monotone IP2, we can solve it in integers (we can reduce it to maximum closure, which in turn reduces to the minimum s,t-cut problem). Its solution is a lower bound on IP2. In general it is not trivial to round the solution obtained and get an integer solution for IP2. However we can prove that there always exists a way to round the variables and get our desired 2-approximation.

46 Vertex cover problem

The vertex cover problem is defined as follows: Given an undirected graph $G = (V, E)$ we want to find the set $S \subseteq V$ with minimum cardinality such that every edge in E is adjacent to (at least) one node in S. In the weighted version, each node $i \in V$ has an associated weight $w_i \geq 0$ and we want to minimize the total weight of the set S. Note that, if the graph has negative or zero weights, then we simply include all those nodes in S and remove their neighbors from V.
Let $x_i = \begin{cases} 1 & \text{if } i \in V \text{ is in the cover} \\ 0 & \text{otherwise} \end{cases}$.
The (weighted) vertex cover problem can be formulated as the integer program below.

$$\min \sum_{i\in V} w_i x_i \qquad (\mathcal{VC}.0)$$
$$\text{s.t.} x_i + x_j \geq 1 \quad \forall\ (i,j)\in E \qquad (\mathcal{VC}.1)$$
$$x_i \in \{0,1\} \quad \forall\ i \in V$$

46.1 Vertex cover on bipartite graphs

The vertex cover problem on bipartite graphs $G = (V_1 \cup V_2, E)$ can be solved in polynomial time. This follows since constraint matrix of the vertex cover problem on bipartite graphs is totally unimodular. To see this, observe that the constraint matrix can be separated in columns into two parts corresponding to V_1 and V_2 respectively, each row within which contains exactly one 1.
That the vertex cover problem on bipartite graph is polynomially solvable can also be seen from our discussion on monotone IP2.
To see this, let $x_i = \begin{cases} 1 & \text{if } i \in V_1 \text{ is in the cover} \\ 0 & \text{otherwise} \end{cases}$,
$y_i = \begin{cases} -1 & \text{if } i \in V_2 \text{ is in the cover} \\ 0 & \text{otherwise} \end{cases}$.
The integer programming formulation for the vertex cover problem on bipartite graph can then be transformed to the following:

$$\begin{aligned} \min & \sum_{j \in V_1} w_j x_j + \sum_{j \in V_2} (-w_j) y_j \qquad (\mathcal{VC}_B) \\ \text{s.t.} & x_i - y_j \geq 1 \quad \forall\ (i,j) \in E \\ & x_i \in \{0,1\} \quad \forall\ i \in V_1 \\ & y_j \in \{-1,0\} \quad \forall\ j \in V_2 \end{aligned}$$

Evidently, this formulation is a Monotone IP2. Furthermore since both x_i and y_j can take only two values, the size of the resulted s,t-Graph is polynomial. In fact, the following s,t-Graph (Figure 46.1) can be constructed to solve the vertex cover problem on bipartite graph.
1. Given a bipartite graph $G = (V_1 \cup V_2, E)$, set the capacity of all edges in A equal to ∞.
2. Add a source s, a sink t, set A_s of arcs from s to all nodes $i \in V_1$ (with capacity $u_{s,i} = w_i$), and set A_t of arcs from all nodes $j \in V_2$ to t (with capacity $u_{j,t} = w_j$).
It is easy to see that if $(s \cup S, t \cup T)$ is a finite $s-t$ cut on $G_{s,t}$, then $(V_1 \cap T) \cup (V_2 \cap S)$ is a vertex cover. Furthermore, if $(s \cup S, t \cup T)$ is an optimal solution to the minimum $s-t$ cut problem on $G_{s,t}$, then $(V_1 \cap T) \cup (V_2 \cap S)$ is a vertex cover with the minimum weights, where

$$C(s \cup S, t \cup T) = \sum_{i \in V_1 \cap T} w_i + \sum_{j \in V_2 \cap S} w_j = \sum_{j \in VC^*} w_j$$

46.2 Vertex cover on general graphs

The vertex cover problem on general graphs is *NP*-hard. For the unweighted version, there is a 2-approximation algorithm due to Gavril for finding a solution at most twice as large as the minimum solution.
The algorithm goes as follows.
1. Find a maximal (but not necessarily maximum) matching in G, i.e., a maximal subset of edges, M, such that no two edges in M have an endpoint in common.
2. Return $VC^M = \cup_{(i,j) \in M} \{i \cup j\}$, i.e., the set of all endpoints of edges in M.
The correctness of the algorithm can be easily shown. Clearly, since M is maximal, there cannot be an edge in $E \backslash M$ with no endpoint in VC^M. Thus, VC^M is a vertex cover. Obviously, $|VC^M| = 2|M|$. Also, for each of the edges in M, one of the endpoints must be in every vertex cover. Therefore, $VC_{opt} \geq |M|$. In conclusion, $|VC^M| = 2|M| \leq 2|VC_{opt}|$.

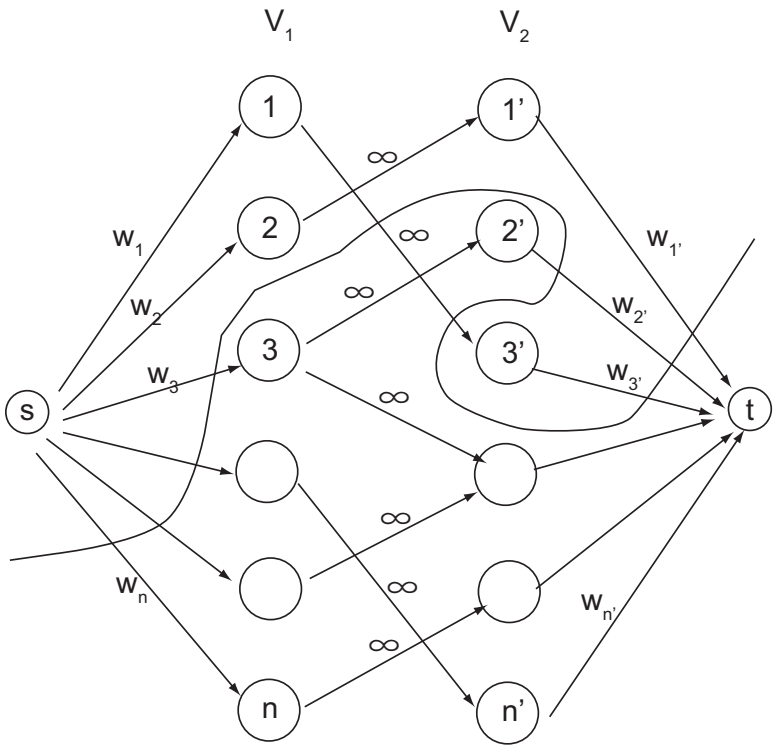


Figure 51: An illustration of the s,t-Graph for the vertex cover problem on bipartite graph and an example of a finite $s-t$ cut

Moreover, the LP-relaxation of the vertex cover problem on general graphs can be solved by solving the vertex cover problem in a related bipartite graph. Specifically, as suggested by Edmonds and Pulleyblank and noted in [NT75], the LP-relaxation can be solved by finding an optimal cover C in a bipartite graph $G_b = (V_{b1} \cup V_{b2}, E_b)$ having vertices $a_j \in V_{b1}$ and $b_j \in V_{b2}$ of weight w_j for each vertex $j \in V$, and two edges $(a_i, b_j), (a_j, b_i)$ for each edge $(i,j) \in E$. Given the optimal cover C on G_b, the optimal solution to the LP-relaxation of our original problem is given by:

$$x_j = \begin{cases} 1 & \text{if} \quad a_j \in C \text{ and } b_j \in C, \\ \frac{1}{2} & \text{if} \quad a_j \in C \text{ and } b_j \notin C, \text{ or} \quad a_j \notin C \text{ and } b_j \in C, \\ 0 & \text{if} \quad a_j \notin C \text{ and } b_j \notin C. \end{cases}$$

In turn, the problem of solving the vertex cover problem in G_b (or, on any bipartite graph) can be reduced to the minimum cut problem as we showed previously. For this purpose we create a (directed) st-graph $G_{st} = (V_{st}, A_{st})$ as follows: (1) $V_{st} = V_b \cup \{s\} \cup \{t\}$, (2) A_{st} contains an infinite-capacity arc (a_i, b_j) for each edge $(a_i, b_j) \in E_b$, (3) A_{st} contains an arc (s, a_i) of capacity w_i for each node $i \in V_{b1}$, and (4) A_{st} contains an arc (b_i, t) of capacity w_i for each node $i \in V_{b2}$.
Given a minimum (S,T) cut we obtain the optimal vertex cover as follows: let $a_i \in C$ if $a_i \in T$ and let $b_j \in C$ if $b_j \in S$.
We now present an alternative method of showing that the LP-relaxation of the vertex cover problem can be reduced to a minimum cut problem, based on our discussion on integer programming with two variables per inequality (IP2).
As we did for the non–monotone integer program with two variables per inequality, we can "mono-

tonize" ($\mathcal{VC}$) by replacing each binary variable x_i in the objective by $x_i = \frac{x_i^+ - x_i^-}{2}$ where $x_i^+ \in \{0,1\}$ and $x_i^- \in \{-1,0\}$, and each inequality $x_i + x_j \geq 1$ by the following two inequalities:

$$x_i^+ - x_j^- \geq 1$$
$$-x_i^- + x_j^+ \geq 1$$

The "monotonized" formulation is shown below:

$$\min \frac{1}{2}\left\{\sum_{i\in V} w_i x_i^+ + \sum_{i\in V}(-w_i)x_i^-\right\} \qquad (\mathcal{VC}'.0)$$
$$\text{s.t. } x_i^+ - x_j^- \geq 1 \quad \forall (i,j) \in E \qquad (\mathcal{VC}'.1)$$
$$-x_i^- + x_j^+ \geq 1 \quad \forall (i,j) \in E \qquad (\mathcal{VC}'.2)$$
$$x_i^+ \in \{0,1\} \quad \forall i \in V$$
$$x_i^- \in \{-1,0\} \quad \forall i \in V$$

The relationship between ($\mathcal{VC}$) and ($\mathcal{VC}'$) can be easily seen. For any feasible solution $\{x_i\}_{i\in V}$ for ($\mathcal{VC}$), a corresponding feasible solution for ($\mathcal{VC}'$) with the same objective value can be constructed by letting $x_i^+ = x_i, x_i^- = -x_i, \forall i \in V$; On the other hand, for any feasible solution $\{x_i^+, x_i^-\}_{i\in V}$ for ($\mathcal{VC}'$), $x_i = \frac{x_i^+ - x_i^-}{2}$ satisfies inequality ($\mathcal{VC}$.1) (To see this, simply sum up inequalities ($\mathcal{VC}$.1) and ($\mathcal{VC}$.2) and divide the resulted inequality by 2.) but may not be integral. Therefore, given an optimal solution $\{x_i^+, x_i^-\}_{i\in V}$ for ($\mathcal{VC}'$), the solution $\{x_i = \frac{x_i^+ - x_i^-}{2}\}$ is feasible, half integral (i.e., $x_i \in \{0, \frac{1}{2}, 1\}$) and super-optimal (i.e., its value provides a lower bound) for ($\mathcal{VC}$).
Comparing ($\mathcal{VC}'$) with $(\mathcal{VC})_{\mathcal{B}}$, we can see that without the coefficient $\frac{1}{2}$ in the objective ($\mathcal{VC}'$) can be treated as a vertex cover problem on a bipartite graph $G(V_{b_1} \cup V^{b_2}, E_b)$ where $V_{b_1} = a_j, \forall j \in V, V_{b_2} = b_j, \forall i \in V, E_b = \{(a_i, b_j), (a_j, b_i), \forall (i,j) \in E\}$.
Therefore, we obtain the following relationship

$$VC^{+-} \leq 2VC^*$$

where VC^* is the optimal value of the original vertex cover problem on a general graph $G(V,E)$, and VC^{+-} is the optimal value of the vertex cover problem defined on the bipartite graph $G(V_{b_1} \cup V^{b_2}, E_b)$.

47 The convex cost closure problem

We now consider a generalized version of the maximum closure problem, referred to as the convex cost closure (ccc) problem. In this variation, the weight for every node $i \in V$ is given by a convex function $w_i(x_i)$. Further, we restrict x_i to be within the range $[l,u]$ and integral.

$$\begin{aligned} \min \quad & \sum_{i\in V} w_i(x_i) \\ \text{subject to} \quad & x_i - x_j \leq 0 \quad (i,j) \in E \\ & l \leq x_i \leq u \quad i \in V \\ & x_i \text{ integer} \quad i \in V \end{aligned}$$

Proposition 47.1. *If node weights are linear functions, then in an optimal solution every x_i is either l or u.*

Proof. Observe that the problem can be converted to the minimum closure problem by translating the x variables as follows. For every node $i \in V$, $y_i = \frac{x_i - l}{u - l}$, which yields

$$\begin{aligned} \min \quad & \sum_{i \in V} w_i y_i \\ \text{subject to} \quad & y_i - y_j \leq 0, \quad (i,j) \in E \\ & 0 \leq y_i \leq 1 \quad i \in V \end{aligned}$$

Since the constraint matrix is totally unimodular, the optimal solution is guaranteed to be integer. Hence, every y_i is either 0 or 1. To get the optimal solution to the original problem we solve for x_i and find that

$$x_i = \begin{cases} l & \text{if } y_i = 0 \\ u & \text{if } y_i = 1 \end{cases}$$

□

We therefore conclude that solving (ccc) with a linear objective is no more interesting than solving the minimum closure problem.
Solving (ccc) with a convex nonlinear objective is not nearly as straight-forward. We introduce the notion of a threshold theorem, which allows us to improve run-time complexity by solving a related problem that prunes the search space.

47.1 The threshold theorem

Definition 47.2. *The graph G_α is constructed as follows. Let the weight at node i be the derivative of w_i evaluated at α: $w_i'(\alpha) \approx \lim_{h \to 0} \frac{w_i(\alpha+h) - w_i(\alpha)}{h}$.*

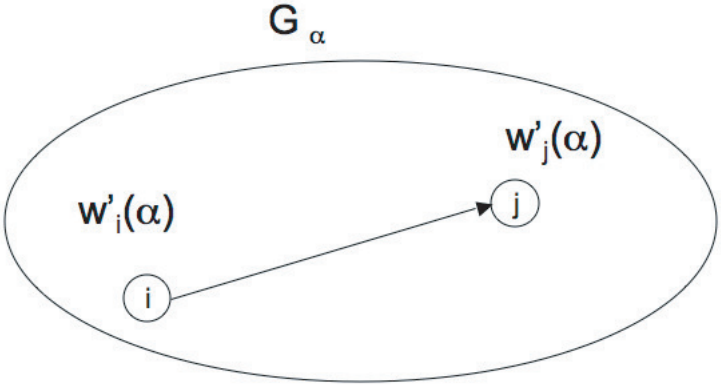


Figure 52: An illustration of G_α

Definition 47.3. *Let a minimal minimum closed set on a graph be a minimum closed set that does not contain any other minimum closed set.*

We now present the threshold theorem as follows.

Theorem 47.1. *An optimal solution to (ccc) on G, x^*, satisfies*

$$x_i^* \geq \alpha \quad \text{if } i \in S_\alpha^*$$
$$x_i^* < \alpha \quad \text{if } i \in \bar{S}_\alpha^*$$

where S_α^* is a minimal minimum weight closed set in G_α.

Proof. For sake of contradiction, let S_α^* be a minimal minimum weight closed set on G_α, and let there be a subset $S_\alpha^0 \subseteq S_\alpha^*$ such that at an optimal solution $x_j^* < \alpha$ for all $j \in S_\alpha^0$. Subsequently, the optimal value for every node $i \in S_\alpha^* \backslash S^0$ has weight $\geq \alpha$. (See Figure 47.1.)

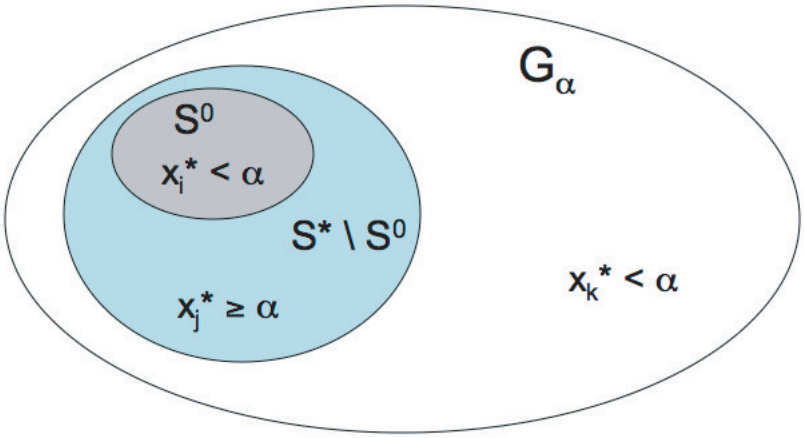


Figure 53: An illustration of the threshold theorem (1)

Recall that the problem requires that $x_i \leq x_j$ for all $(i,j) \in A$. As a consequence, there cannot be a node in S_α^0 that is a successor of a node in $S_\alpha^* \backslash S_\alpha^0$, otherwise the constraint will be violated. Since S_α^* is a closed set and there are no nodes in $S_\alpha^* \backslash S_\alpha^0$ that have successors in S_α^0, $S_\alpha^* \backslash S_\alpha^0$ must be a closed set.

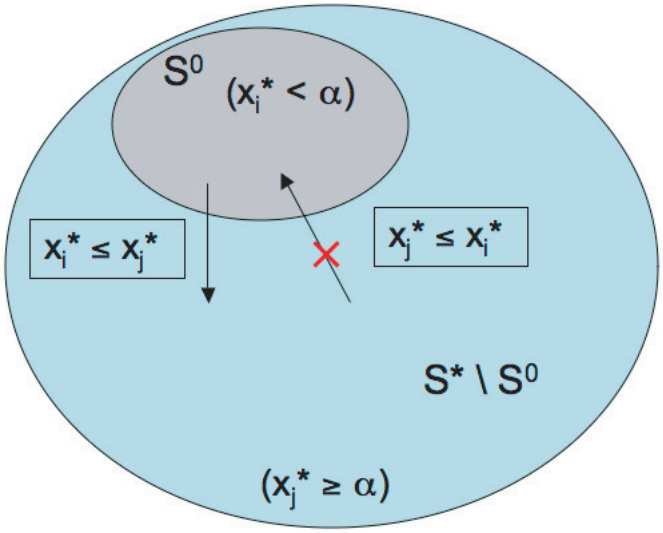


Figure 54: An illustration of the threshold theorem (2)

But this $S^*_\alpha \backslash S^0_\alpha$ cannot be a minimum closed set, otherwise we would violate the assumption that S^*_α is a minimal minimum closed set (since $S^*_\alpha \backslash S^0_\alpha \subset S^*_\alpha$). Therefore, it must be the case that

$$\begin{aligned} \sum_{j \in S^*_\alpha \backslash S^0_\alpha} w'_j(\alpha) &> \sum_{j \in S^*_\alpha} w'_j(\alpha) \\ &= \sum_{j \in S^0_\alpha} w'_j(\alpha) + \sum_{j \in S^*_\alpha \backslash S^0_\alpha} w'_j(\alpha) \end{aligned}$$

which implies that

$$\sum_{j \in S^0_\alpha} w'_j(\alpha) < 0 \, .$$

Next we observe that increasing the values x^*_i to α for all $i \in S^0$ does not violate the constraint that $x^*_i \leq x^*_j$, since by construction $x^*_i \leq \alpha < x^*_j$ for all $j \in S^*$ and $(i,j) \in A$.

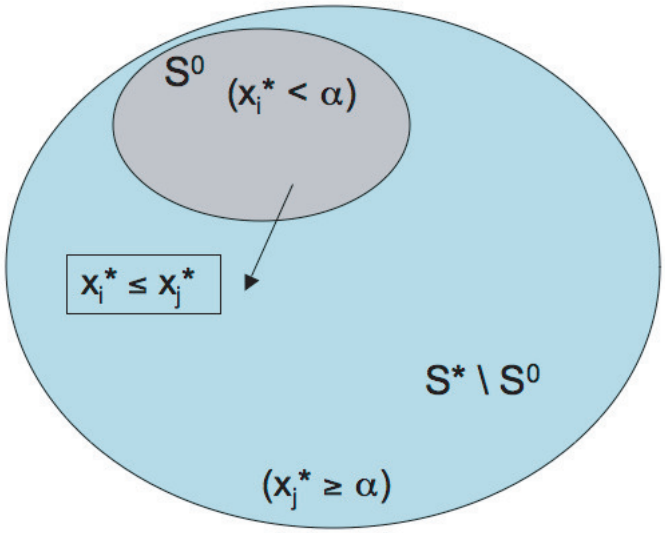


Figure 55: An illustration of the threshold theorem (3)

Since the node weights $w()$ are convex functions, their sum W is also a convex function. Further, we note that the derivative a convex function is monotone nondecreasing, and for any $\epsilon > 0$, if $W'(\alpha) < 0$ then $W(\alpha - \epsilon) > W(\alpha)$. (See Figure 56.)

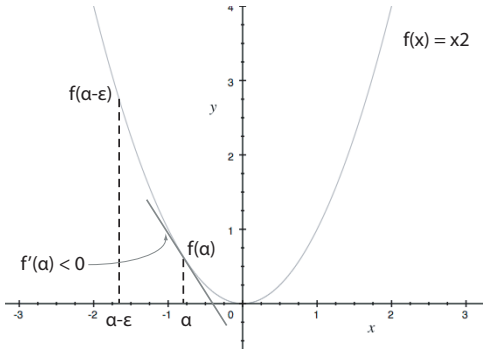


Figure 56: Properties of convex function

Therefore, if for all $i \in S^0_\alpha$ we increase the value of x^*_i to α, we will strictly reduce the weight of the closure. This is a contradiction on the assumption that x^* is optimal and S^*_α is a minimum closed set. □

47.2 Naive algorithm for solving (ccc)

The *Threshold Theorem* can be used to solve the convex cost closure problem by testing all values of $\alpha \in [l, u]$. The optimal value for node $i \in V$ is $x^*_i = \alpha$ such that $i \in S^*_{\alpha-1}$ and $i \in \bar{S}^*_\alpha$. The running time for this algorithm is $(u - l)\ O(\min\ s\text{-}t\ \text{cut})$. Since $u = 2^{\log_2 u}$ and $l = 2^{\log_2 l}$, we note that this is an exponential-time algorithm.

47.3 Solving (ccc) in polynomial time using binary search

We can improve the running time by using a binary search as follows. Pick a node $v \in V$. Solve for S^*_α, choosing α to be the median of u and l. If $v \in S^*_\alpha$, then by the Threshold Theorem we know that $x^*_v > \alpha$, and we set α to the median of $\frac{u-l}{2}$ and u. Otherwise, let α be the median of l and $\frac{u-l}{2}$. We repeat the process, performing binary search on α until we find the optimal solution. Algorithm 1 (below) describes the procedure in greater detail.
The binary search for every variable takes $O(\log_2 (u - l))$ time to compute, and we must do this a total of n times (once for each node in V). Hence, this method reduces our running time to $n \log_2 (u - l)\ O(\min\ s\text{-}t\ \text{cut})$, which is polynomial.

47.4 Solving (ccc) using parametric minimum cut

Hochbaum and Queyranne provide further run-time improvements for optimally solving (ccc) in [12]. This section summarizes their result.
From G we construct a parametric graph G_λ by introducing a source node s and sink node t. We add an arc from s to every node $v \in V$ with capacity $\max\{0,\ w'_v(\lambda)\}$, and an arc from v to t with capacity $-min\{0,\ w'_v(\lambda)\}$. We make the following observations.

- Every arc in G_λ has nonnegative capacity.
- In the residual graph of G_λ, every node in V is connected to either the source or the sink (not both).
- Arcs adjacent to the source have capacities that are monotone nondecreasing in λ, and arcs adjacent to the sink have capacities that are monotone nonincreasing as a function of λ.

Algorithm 1 Binary search algorithm for (ccc)

Require: $G = (V, A)$ is a closure graph, where node weights $w()$ are convex functions

procedure BINARY SEARCH(G, l_0, u_0)
  for $v = 1, \ldots, n$ **do**
    $l \leftarrow l_0, \quad u \leftarrow u_0$
    while $u - l > 0$ **do**
      $\alpha \leftarrow l + \lceil \frac{u-l}{2} \rceil$ ▷ Set α to the median of u and l
      $S^*_\alpha \leftarrow$ source set of minimum $s - t$ cut on G_α
      if $v \in S^*_\alpha$ **then**
        $l \leftarrow l + \lceil \frac{u-l}{2} \rceil$ ▷ α is the new lower bound on x^*_v
      else
        $u \leftarrow l + \lceil \frac{u-l}{2} \rceil$ ▷ α is the new upper bound on x^*_v
      end if
    end while
    $x^*_v \leftarrow \alpha$
  end for
  return $x^* = [x^*_1, \ldots, x^*_n]$
end procedure

Let S_λ be the source set of a minimum cut in G_λ; equivalently, S_λ is a minimum closed set on G_λ. As we increase λ in the interval $[l, u]$, we find that the size of the source set S_λ increases, and S_λ is a subset of source sets corresponding to higher values of λ. More formallly,

$$S_\lambda \subseteq S_{\lambda+1} \qquad \text{for all } \lambda.$$

Definition 47.4. *For any node $v \in V$, the node shifting breakpoint is defined as the largest value $\underline{\lambda}$ such that v is in the source set $S_{\underline{\lambda}}$, and the smallest value $\bar{\lambda}$ such that v is not in the minimum closed set ($S_{\bar{\lambda}}$). Note that $\bar{\lambda} = \underline{\lambda} + 1$. Further,*

$$x^*_v = \min_{x \in [\underline{\lambda}, \bar{\lambda}]} w_v(x)$$

If we can find the node shifting breakpoint for each node $v \in V$, then we can construct the optimal solution to (ccc) as shown in Algorithm 2. Hochbaum and Queyranne show that this problem reduces to solving the parametric minimum cut problem on G [12]. Conveniently, Gallo, Grigoriadis, and Tarjan provide a method that solves this problem in the same running time as a single minimum closure (or maximum flow) problem [17].

The first step of the algorithm requires solving the parametric cut problem on G to find the set of node shifting breakpoints. Following the method outlined in [17], this requires $O\left(mn \log \frac{n^2}{m} + n \log(u - l)\right)$ time to compute, where the second term corresponds to finding the minimum for each node weight function using binary search. Since determining the optimal solution x^* takes time linear in n, the total running time of the algorithm is

$$O\left(mn \log \frac{n^2}{m} + n \log(u - l)\right) .$$

If the node weight functions are quadratic we can find the minimum of each function in constant time, and the complexity of the algorithm reduces to $O\left(mn \log \frac{n^2}{m}\right)$; this is equivalent to solving maximum flow (or minimum $s - t$ cut) on a graph!

Algorithm 2 Breakpoint method for solving (ccc)

Require: $G = (V, A)$ is a closure graph, where node weights $w()$ are convex functions

procedure BREAKPOINT(G, w_v for $v = 1, \ldots, n$)
  Call (the modified) PARAMETRIC-CUT to find a set of up to n breakpoints $\lambda_1, \ldots, \lambda_n$
  for $v = 1, \ldots, n$ **do**
    $\min_v = \arg\min_{x \in [l,u]} w_v(x)$
    Let the optimal value of x_v fall in the interval $(\lambda_{v_i - 1}, \lambda_{v_i}]$
    Then x_v^* is determined by

$$x_v^* = \begin{cases} \lambda_{v_i - 1} + 1 & \text{if } \min_v \leq \lambda_{v_i - 1} \\ \lambda_{v_i} & \text{if } \min_v \geq \lambda_{v_i} \\ \min_v & \text{if } \lambda_{v_i - 1} \leq \min_v \leq \lambda_{v_i} \end{cases}$$

  end for
  return $x^* = [x_1^*, \ldots, x_n^*]$
end procedure

48 The s-excess problem

The s-excess problem is another generalization of the closure problem. Here we introduce edge weights e_{ij} and relax the closure requirement. However, the objective receives a penalty for any pair of nodes that do not satisfy the requirement, which is proportional to the amount of the violation. Let z_{ij} be 1 if $i \in S^*$ and $j \in \bar{S}^*$, and 0 otherwise. Then the convex s-excess problem is to find a subset of nodes $S \subseteq V$ that minimizes

$$\sum_{i \in S} w_i + \sum_{i \in S, j \in \bar{S}} e_{ij} \ ,$$

which can be represented by the following integer linear program.

$$\begin{array}{lll} \min & \sum_{i \in V} w_i x_i \ + \sum_{(i,j) \in A} e_{ij} z_{ij} & \\ & & \\ \text{subject to} & x_i - x_j \leq z_{ij} & (i,j) \in A \\ & 0 \leq x_i \leq 1 & i \in V \\ & 0 \leq z_{ij} \leq 1 & (i,j) \in A \\ & x_i, \ z_{ij} \text{ integer} & \end{array}$$

[1] and [10] show that this problem is equivalent to solving the minimum $s - t$ cut problem on a modified graph, G_{st}, defined as follows. We add nodes s and t to the graph, with an arc from s to every negative weight node i (with capacity $u_{si} = -w_i$), and an arc from every positive weight node j to t (with capacity $u_{jt} = w_j$).

Lemma 48.1. *S^* is a set of minimum s-excess capacity in the original graph G if and only if S^* is the source set of a minimum cut in G_{st}.*

Proof. As before, let $V^+ \equiv \{i \in V | w_i > 0\}$, and let $V^- \equiv \{j \in V | w_j < 0\}$. Let $(s \cup S, t \cup T)$ define an $s - t$ cut on G_{st}. Then the capacity of this cut is given by

$$\begin{aligned} C(s\cup S, t\cup T) &= \sum_{(s,i)\in A_{st},\ i\in T} u_{s,i} \ + \sum_{(j,t)\in A_{st},\ j\in S} u_{j,t} + \sum_{i\in S,\ j\in T} e_{ij} \\ &= \sum_{i\in T\cap V^-} -w_i + \sum_{j\in S\cap V^+} w_j + \sum_{i\in S,\ j\in T} e_{ij} \\ &= \sum_{i\in V^-} -w_i - \sum_{j\in S\cap V^-} -w_i + \sum_{j\in S\cap V^+} w_j + \sum_{i\in S,\ j\in T} e_{ij} \\ &= W^- + \sum_{j\in S} w_j + \sum_{i\in S,\ j\in T} e_{ij} \end{aligned}$$

Where W^- is the sum of all negative weights in G, which is a constant. Therefore, minimizing $C(s\cup S, t\cup T)$ is equivalent to minimizing $\sum_{j\in S} w_j + \sum_{i\in S,\ j\in T} e_{ij}$, and we conclude that the source set of a minimum $s-t$ cut on G_{st} is also a minimum s-excess set of G.

□

48.1 The convex s-excess problem

The convex s-excess problem is a generalized version of the s-excess problem. In this problem the weight for every node $i\in V$ is given by a convex function $w_i(x_i)$, where x_i is restricted to be within the range $[l,u]$ and integral.

$$\min \quad \sum_{i\in V} w_i(x_i) \ + \sum_{(i,j)\in A} e_{ij} z_{ij}$$

$$\begin{array}{lcl} \text{subject to} & x_i - x_j \le z_{ij} & (i,j)\in A \\ & l \le x_i \le u & i\in V \\ & z_{ij} \ge 0 & (i,j)\in A \\ & x_i,\ z_{ij} \text{ integer} & \end{array}$$

Observe that if any of the arc weights are negative, then the problem is unbounded. Hence, the only problems of interest are when the arc weights are positive. Further, because the arc weights are linear, it is sufficient to find the optimal values of x; that is, z_{ij} can be uniquely determined by x as follows: $z_{ij} = \max\{x_i - x_j, 0\}$.

48.2 Threshold theorem for linear edge weights

Similar to the convex cost closure problem, we define the convex s-excess problem on a graph G_α, where α is an integer scalar. The weight for every node i is the subgradient of w_i evaluated at α, $w'(\alpha)$, and the arc weights for every arc $(i,j)\in A$ are e_{ij}. The following threshold theorem for the convex s-excess problem with linear edge weights was proven by Hochbaum in [8].

Theorem 48.1. *Let S^*_α be the maximal minimum s-excess set in G_α. Then an optimal solution x^* to the corresponding convex s-excess problem (with linear e_{ij}'s) satisfies the following property*

$$\begin{array}{ll} x^*_i \ge \alpha & \text{if } i\in S^*_\alpha \\ x^*_i < \alpha & \text{if } i\in \bar{S}^*_\alpha \end{array}$$

Proof. Let S^*_α be a maximal minimum s-excess set in G_α that violates the theorem. Then for an optimal solution x^* there is either

- A subset $S^0 \subseteq S^*_\alpha$ such that for all $j \in S^0$, $x^*_j < \alpha$. Or,
- A subset $S^1 \subseteq \bar{S}^*_\alpha$ such that for all $j \in S^1$, $x^*_j \geq \alpha$.

Now suppose there exists a subset S^0. The amount contributed to the objective by adding S^0 to $S^*_\alpha \backslash S^0$ is given by

$$\Delta^0 = \sum_{j \in S^0} w_j + \mathrm{C}\left(S^0, \bar{S}^*_\alpha\right) - \mathrm{C}\left(S^*_\alpha \backslash S^0, S^0\right)$$

where $\mathrm{C}(A, B)$ is the capacity of a cut from set A to set B. Since S^*_α is a minimum s-excess set, this amount must be negative (otherwise $S^*_\alpha \backslash S^0$ would be a smaller s-excess set).
We now consider the following solution x' for some $\epsilon > 0$:

$$x'_i = \begin{cases} x^*_i & \text{if } i \notin S^0 \\ x^*_i + \epsilon & \text{if } i \in S^0 \end{cases}$$

Then the problem objective P evaluated at x' and x^* must satisfy the relation

$$P\left(x'\right) \leq P\left(x^*\right) + \epsilon\Delta^0 < P\left(x^*\right) \ ,$$

which is a contradiction on the optimality of x^*. Therefore, in every optimal solution x^*, we conclude that $x^*_i \geq \alpha$ for $i \in S^*_\alpha$.
We now assume that there exists a subset S^1 of $\bar{S}^*_\alpha$ as defined above, and we are given an optimal solution x^*. The amount contributed to the objective by adding S^1 to S^*_α is

$$\Delta^1 = \sum_{j \in S^1} w_j + \mathrm{C}\left(S^1, S^*_\alpha \backslash S^1\right) - \mathrm{C}\left(S^*_\alpha, S^1\right)$$

Let $\delta = \min_{j \in S^1}\left(x^*_j - \alpha\right) + \epsilon > 0$, and define solution x'' as follows.

$$x''_i = \begin{cases} x^*_i & \text{if } i \notin S^1 \\ x^*_i - \delta & \text{if } i \in S^1 \end{cases}$$

By construction, all arcs $(i, j) \in \left(S^1, \bar{S}^*_\alpha \backslash S^1\right)$ in the solution x'' must have $x''_i > x''_j$; as a result, the corresponding z_{ij}'s must be positive. The problem objective evaluated at x'' and $x*$ must therefore satisfy

$$P\left(x''\right) \leq P\left(x^*\right) + \delta\Delta^1 < P\left(x^*\right) \ ,$$

which contradicts the optimality of x^*. □

48.3 Variants / special cases

- The threshold theorem has not (yet!) been proven to hold for problems with nonlinear (convex) edge weights.
- For general functions, the s-excess problem is known as the Markov random fields problem.

Lec12

49 Forest Clearing

The Forest Clearing problem is a real, practical problem. We are given a forest which needs to be selectively cut. The forest is divided into squares where every square has different kind of trees. Therefore, the value (or benefit from cutting) is different for each such square. The goal is to find the most beneficial way of cutting the forest while preserving the regulatory constraints.

Version 1. One of the constraints in forest clearing is preservation of deer habitat. For example, deer like to eat in big open areas, but they like to have trees around the clearing so they can run for cover if a predator comes along. If they don't feel protected, they won't go into the clearing to eat. So, the constraint in this version of Forest Clearing is the following: no two adjacent squares can be cut. Adjacent squares are two squares which share the same side, i.e. adjacent either vertically or horizontally.

Solution. Make a *grid-graph* $G = (V, E)$. For every square of forest make a node v. Assign every node v, weight w, which is equal to the amount of benefit you get when cut the corresponding square. Connect two nodes if their corresponding squares are adjacent. The resulting graph is bipartite. We can color it in a chess-board fashion in two colors, such that no two adjacent nodes are of the same color. Now, the above Forest Clearing problem is equivalent to finding a max-weight independent set in the grid-graph G. It is well known that Maximum Independent Set Problem is equivalent to Minimum Vertex Cover Problem and we will show it in the next section. Therefore, we can solve our problem by solving weighted vertex cover problem on G by finding Min-Cut in the corresponding network. This problem is solved in polynomial time.

Version 2. Suppose, deer can see if the diagonal squares are cut. Then the new constraint is the following: no two adjacent vertically, horizontally or diagonally squares can be cut. We can build a grid-graph in a similar fashion as in *Version 1.* However, it will not be bipartite anymore, since it will have odd cycles. So, the above solution would not be applicable in this case. Moreover, the problem of finding Max Weight Independent Set on such a grid-graph with diagonals is proven to be *NP*-complete.

Another variation of Forest Clearing Problem is when the forest itself is not of a rectangular shape. Also, it might have "holes" in it, such as lakes. In this case the problem is also *NP*-complete. An example is given in Figure 57.

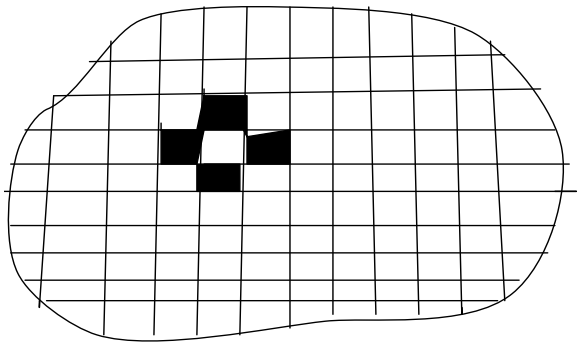


Figure 57: An example of Forest Clearing Problem with a non-rectangular shape forest

Because there are no odd cycles in the above grid graph of version 1, it is bipartite and the aforementioned approach works. Nevertheless, notice that this method breaks down for graphs where cells are neighbors also if they are adjacent diagonally. An example of such a graph is given in Figure 58.

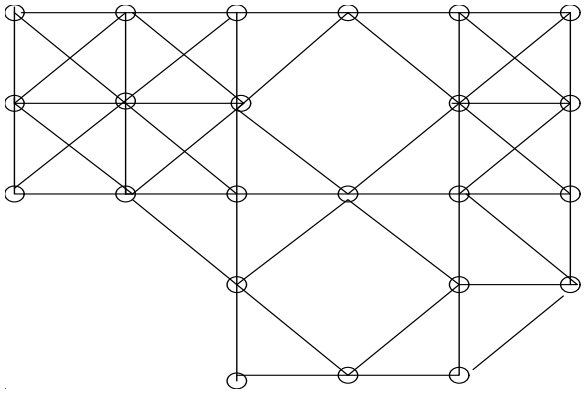


Figure 58: An example of Forest Clearing Version 2

Such graphs are no longer bipartite. In this case, the problem has indeed been proved *NP*-complete in the context of producing memory chips (next application).

50 Producing memory chips (VLSI layout)

We are given a silicon wafer of 1G RAM VLSI chips arranged in a grid-like fashion. Some chips are damaged, those will be marked by a little dot on the top and cannot be used. The objective is to make as many 4G chips as possible out of good 1G chips. A valid 4G chip is the one where 4 little chips of 1G form a square, (i.e. they have to be adjacent to each other in a square fashion.) Create a grid-graph $G = (V, E)$ in the following manner: place a node in the middle of every possible 4G chip. (The adjacent 256K chips will overlap.) Similar to *Version 2* of Forest Clearing Problem put an edge between every two adjacent vertically, horizontally or diagonally nodes. Now, to solve the problem we need to find a maximum weighted independent set on G with the nodes which cover the defective chip has zero weight. This problem is proven to be *NP*-complete.

51 Independent set problem

An independent set or stable set is a set of vertices in a graph, no two of which are adjacent. That is, given an undirected graph $G = (V, E)$, a set of the vertices, $S \subset V$, is said to be an independent set, if for every two vertices in S, there is no edge connected them. The maximum independent set problem is to find an independent set of G with the maximum number of vertices. In the weighted version, each node $i \in S$ has an associated weight $w_i \geq 0$ and we want to maximum the total weight of the set S. We can formulate the maximum weighted independent set problem as follows.

$$\text{Let } x_i = \begin{cases} 1 & \text{if } i \in V \text{ is in the set } S \\ 0 & \text{otherwise} \end{cases}$$

$$\text{maximum } \sum_i w_i x_i \tag{41}$$

$$\text{subject to } x_i + x_j \leq 1, \ \forall (i,j) \in E \tag{42}$$

$$x_i \in \{0, 1\}, \ \forall i \in V \tag{43}$$

51.1 Independent Set v.s. Vertex Cover

Recall that the vertex cover problem is to find the smallest collection of vertices S in an undirected graph G such that every edge in the graph is incident to some vertex in S. (Every edge in G has an endpoint in S).
Notation: If V is the set of all vertices in G, and $S \subset V$ is a subset of the vertices, then $V \setminus S$ is the set of vertices in G not in S.

Lemma 51.1. *If S is an independent set, then $V \setminus S$ is a vertex cover.*

Proof. Suppose $V \setminus S$ is not a vertex cover. Then there exists an edge whose two endpoints are not in $V \setminus S$. Then both endpoints must be in S. But, then S cannot be an independent set! (Recall the definition of an independent set – no edge has both endpoints in the set.) So therefore, we proved the lemma by contradiction. □

The proof will work the other way, too:
Result: S is an independent set if and only if $V - S$ is a vertex cover.
This leads to the result that the sum of a vertex cover and its corresponding independent set is a fixed number (namely $|V|$).

51.2 Independent set on bipartite graphs

The maximum independent set on a bipartite graph $G = (V_1 \cup V_2, E)$ can be transformed to the following s,t-Graph, and a finite capacity cut in the s,t-Graph correspond to an independent set and a vertex cover solution. To construct the following s,t-Graph, first we set the capacity of all edges in E equal to ∞. Then we add a source s, a sink t, set A_s of arcs from s to all nodes $i \in V_1$ with capacity $u_{s,i} = w_i$, and set A_t of arcs from all nodes $j \in V_2$ to t with capacity $u_{j,t} = w_j$.
It is easy to see that for a finite $s-t$ cut, $(s \cup S, t \cup T)$, on $G_{s,t}$, $(S \cap V_1 \cup T \cap V_2)$ is an independent set. The total weight of this independent set is

$$w(S \cap V_1 \cup T \cap V_2) = \sum_{i \in S \cap V_1} w_i + \sum_{j \in T \cap V_2} w_j$$

Moreover, the capacity of the $s-t$ cut is

$$C(s \cup S, t \cup T) = \sum_{i \in V_1 \cap T} C(s,i) + \sum_{j \in V_2 \cap S} C(j,t) \tag{44}$$

$$= \sum_{i \in V_1 \cap T} w_i + \sum_{j \in V_2 \cap S} w_j \tag{45}$$

$$= W - [\sum_{i \in S \cap V_1} w_i + \sum_{j \in T \cap V_2} w_j] \tag{46}$$

where

$$W = \sum_{i \in V_1} w_i + \sum_{j \in V_2} w_j$$

Therefore, we can observe that the minimum cut of the $G_{s,t}$ is equivalent to the maximum weighted independent set.

52 Maximum Density Subgraph

Given a graph $G = (V, E)$ and a subset $H \subseteq V$ of nodes, we denote as $E(H) \subseteq E$ the set of edges with both endpoints in H. That is,

$$E(H) = \{(i, j) | (i, j) \in E \text{ and } i, j \in H\}$$

The *density* of a graph is the ratio of the number of edges to the number of nodes in the graph. Given a graph $G = (V, E)$ the maximum density subgraph problem, is to identify the subgraph in G with maximum density; that is, we want to find a nonempty subset of nodes $H \subseteq V$ such that $\frac{|E(H)|}{|H|}$ is maximized. To simplify the notation, let $n = |V|$, the number of nodes in G, and let $m = |E|$, the number of edges in G.

52.1 Linearizing ratio problems

A general approach for maximizing a fractional (or as it is sometimes called, geometric) objective function over a feasible region $\mathcal{F}$, $\min_{x \in \mathcal{F}} \frac{f(x)}{g(x)}$, is to reduce it to a sequence of calls to an oracle that provides the yes/no answer to the λ-*question*:

$$\text{Is there a feasible subset } x \in \mathcal{F} \text{ such that } (f(x) - \lambda g(x) < 0)?$$

If the answer to the λ-question is *yes* then the optimal solution to the original problem has a value smaller than λ. Otherwise, the optimal value is greater than or equal to λ. A standard approach is then to utilize a binary search procedure that calls for the λ-question $O(\log \frac{U}{\ell})$ times in order to solve the problem, where U is an upper bound on the value of the numerator and ℓ an lower bound on the value of the denominator.
Therefore, if the linearized version of the problem, that is the λ-question, is solved in polynomial time, then so is the ratio problem. Note that the number of calls to the linear optimization is not strongly polynomial but rather, if binary search is employed, depends on the logarithm of the magnitude of the numbers in the input. In some cases however there is a more efficient procedure.
It is important to note that *not all* ratio problems are solvable in polynomial time. One prominent example is the *ratio-cuts*. For that problem, the linearized version is NP-hard by reduction from maximum cut.
It is also important to note that linearizing is not always the right approach to use for a ratio problem. For example, the ratio problem of finding a partition of a graph to k components minimizing the k-cut between components for $k \geq 2$ divided by the number of components k, always has an optimal solution with $k = 2$ which is attained by a, polynomial time, minimum 2-cut algorithm. On the other hand, the linearized problem is NP-hard to solve, since it is equivalent to solving the minimum k-cut problem. (However, note that we can solve a minimum k-cut problem for fixed k.)

52.2 Solving the maximum density subgraph problem

For formulating the maximum density problem, let x_i and y_{ij} be:

$$x_i = \begin{cases} 1 & \text{if } i \in S \\ 0 & \text{if } i \in \bar{S}. \end{cases}$$

and

$$y_{ij} = \begin{cases} 1 & \text{if } i, j \in S \\ 0 & \text{otherwise.} \end{cases}$$

Then, the linearized maximum density problem corresponding λ question is

$$\begin{array}{lll} (\lambda\text{-MD}) & \max & \sum_{[i,j]\in E} w_{ij}y_{ij} - \sum_{j\in V} \lambda x_j \\ \text{subject to} & & y_{ij} \leq x_j \quad \text{for all } [i,j] \in E \\ & & y_{ij} \leq x_j \quad \text{for all } [i,j] \in E \\ & & x_j \text{ binary } j \in V \\ & & y_{ij} \text{ binary } i,j \in V. \end{array}$$

The λ-question is presented as a minimum s,t-cut problem on an unbalanced bipartite graph G^b. As illustrated in figure 59, the graph G^b is constructed so nodes representing the *edges* y_{ij} of the graph G are on one side of the bipartition and nodes representing the *nodes*, x_i, of G are on the other. Each node on G^b representing an edge of G is connected with infinite capacities arcs to the nodes representing its end nodes on G. e.g the corresponding node on G^b to the edge y_{23} on G is connected to the nodes in G^b that correspond to the nodes x_2 and x_3 on G. That bipartite graph has $m+n$ nodes, and $m' = O(m)$ arcs. This is actually a selection problem where the edges y_{ij} are the set of selections and the nodes x_i are the elements. If the solution of the selection problem is empty set, the answer of the λ-question is "No." The complexity of a single minimum s,t-cut in such graph is therefore $O(m^2 \log m)$. This complexity however can be improved to $O(mn \log(\frac{n^2}{m} + 2))$ Once we know the answer of a λ-question, we know we can increase or decrease the value of λ to find the maximum density subgraph. By implementing a polynomial-time search, say, binary search, on the value of λ, we can solve the maximum density subgraph problem in strongly polynomial time.

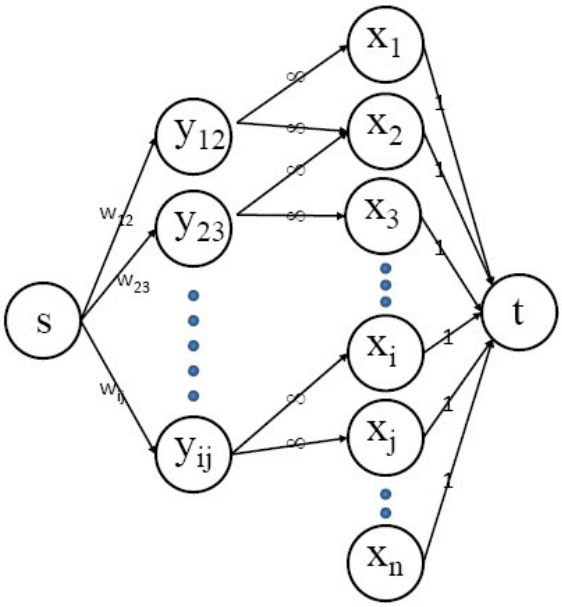


Figure 59: The graph for solving the densest Region Problem

53 Parametric cut/flow problem

Given a general graph $G = (V, E)$, as illustrated in Figure 60, we can create a graph with parametric capacities $G_\lambda = (V \cup \{s,t\}, A)$ where each node $j \in V$ has an incoming arc from s with weight $f_j(\lambda)$, and an outgoing arc to the sink t with weight $g_j(\lambda)$. If $f_j(\lambda)$ is monotone nondecreasing in λ and $g_j(\lambda)$ is monotone nonincreasing in λ, or $f_j(\lambda)$ is monotone nonincreasing in λ and $g_j(\lambda)$ is monotone nondecreasing in λ, the min-cut (or max-flow) problem of a such graph is a "parametric

cut(or flow) problem." Note that in Figure 60, we do not need to have infinity capacity on the edges in graph G.

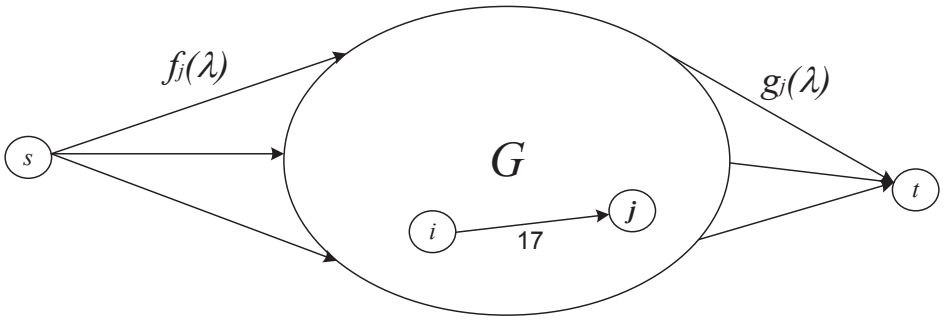


Figure 60: Parametric capacity graph

To solve this problem, we can take certain min-cut algorithm and apply it to all possible values of λ to solve the problem. The detail of the algorithm is shown in [11]. The complexity of the algorithm is $O(T(m,n) + n \log U)$, where $T(m,n)$ is the complexity of solving the min-cut problem, $n = |V|$, $m = |E|$, and $U = u - l$, where $\lambda \in [l, u]$.

53.1 The parametric cut/flow problem with convex function (tentative title)

Consider a convex function $f(x)$ and the parametric cut/flow problem where each node $j \in V$ has an incoming arc from s with capacity $\max\{0, f_j'(\lambda)\}$, and an outgoing arc to the sink t with capacity $-\min\{0, f_j'(\lambda)\}$. The capacities of the arcs adjacent to the source in this graph are monotone nondecreasing as a function of λ, and the arcs adjacent to the sink are all with capacities that are monotone nonincreasing as a function of λ. Note that each node is connected with a positive capacity arc, either to source, or to sink, but not to both. Denote the source set of a minimum cut in the graph G_λ by S_λ.
Restating the threshold theorem in terms of the corresponding minimum cut for the graph G_λ associated with the closure graph, any optimal solution $\mathbf{x}$ satisfies that $x_j > \lambda$ for $j \in \bar{S}_\lambda$ and $x_j \leq \lambda$ for $j \in S_\lambda$, where S_λ is the maximal source set of a minimum cut.
Let ℓ be the lowest lower bound on any of the variables and u be the largest upper bound. Consider varying the value of λ in the interval $[\ell, u]$. As the value of λ increases, the sink set becomes smaller and contained in the previous sink sets corresponding to smaller values of λ. Specifically, for some $\lambda \leq \ell$ $S_\lambda = \{s\}$, and for some $\lambda \geq u$ $S_\lambda = V \cup \{s\}$. We call each value of λ where S_λ strictly increases – a *node shifting breakpoint*. For $\lambda_1 < \ldots < \lambda_\ell$ the set of all node shifting breakpoints we get a corresponding nested collection of source sets:

$$\{s\} = S_{\lambda_1} \subset S_{\lambda_2} \subset \ldots \subset S_{\lambda_\ell} = \{s\} \cup V.$$

The number of S_λ's is at most the number of nodes. Hence, the number of breakpoints cannot be greater than n. Therefore, although λ can be any value between ℓ and u, the number of intervals is small. This problem cannot be solved in strongly polynomial time since the complexity of any algorithm solving this problem depends on the range of possible values of λ. Gallo, Grigoriadis and Tarjan [17] claimed that the running time of their algorithm is $O(T(m,n))$ but they are wrong, since in their algorithm, they claim taking intersection of two function is constant time, which is incorrect.

Note that although a parametric cut/flow problem cannot be solved in strongly polynomial time, the output of this problem is compact. The output is just those breakpoint with the nodes added into the current subset, S_λ.

54 Average k-cut problem

Given a graph $G = (V, E)$, a k-cut is finding a set of edges whose removal would partition the graph to k connected components, denoted as $V_1, V_2, \ldots, V_k$. The cost of a k-cut is the summation of the costs (or weight) of those edges. Let $C(V_i, V_j)$ be the total cost of the edges connecting V_i and V_j. The objective function of the average k-cut problem is given below.

$$\min \frac{\sum_{i=1}^{k} C(V_i, V - V_i)}{2|k|}$$

Another problem which is similar to the average k-cut problem is the strength problem with the following objective function.

$$\min \frac{\sum_{i=1}^{k} C(V_i, V - V_i)}{2(|k| - 1)}$$

If we linearize the above two problem, the problem is actually k-cut problem. For k-cut problem with fixed k, it is polynomial time solvable. However, in general, a k-cut problem is NP-hard. However, the average k-cut problem is easy, since we have

$$\min \frac{\sum_{i=1}^{k} C(V_i, V - V_i)}{2|k|} \geq \frac{|k| C(S^*, T^*)}{2|k|} = \frac{C(S^*, T^*)}{2}$$

where $C(S^*, T^*)$ is a minimum 2-cut of the graph G with the optimal source node and sink node in graph G. Note that to find the optimal source node and sink node, we have to check all possible pairs of $(s, t) \in V$ and find the minimum $s - t$ cut solution of each pair of (s, t) and then compare the values to find the best pair. Hence, the min 2-cut is the solution of this problem, which is easy to solve.
Hence, the average k-cut problem can be solved in polynomial time for general k, but the strength problem is NP-complete.

55 Image segmentation problem

A graph theoretical framework is suitable for representing image segmentation and grouping problems. The image segmentation problem is presented on an undirected graph $G = (V, E)$, where V is the set of pixels and E are the pairs of adjacent pixels for which similarity information is available. Typically one considers a planar image with pixels arranged along a grid. The 4-neighbors set up is a commonly used adjacency rule with each pixel having 4 neighbors – two along the vertical axis and two along the horizontal axis. This set up forms a planar grid graph. The 8-neighbors arrangement is also used, but then the planar structure is no longer preserved, and complexity of various algorithms increases, sometimes significantly. Planarity is also not satisfied for 3-dimensional images, and in general clustering problems there is no grid structure and thus the respective graphs are not planar.
The edges in the graph representing the image carry *similarity* weights. The similarity is inversely increasing with the difference in attributes between the pixels. In terms of the graph, each edge $[i, j]$ is assigned a similarity weight w_{ij} that increases as the two pixels i and j are perceived to

be more similar. Low values of w_{ij} are interpreted as dissimilarity. However, in some contexts one might want to generate *dissimilarity* weights independently. In that case each edge has two weights, w_{ij} for similarity, and $\hat{w}_{ij}$ for dissimilarity.
The goal of then *"normalized cut variant"* problem is to minimize the ratio of the similarity between the set of objects and its complement and the similarity within the set of objects. That is, given a graph $G = (V, E)$, we would like to find a subset $S \subset V$, such that $C(S, \overline{S})$ is very small and $C(S, S)$ is very large. Hence, the objective function is

$$\min_{\emptyset \neq S \subset V} \frac{C(S, \overline{S})}{C(S, S)} \tag{47}$$

Note that we also need to have $|S| \geq 2$ and $|E(S)| \geq 1$, otherwise, the objective is undefined.
Shi and Malik addressed in their work on segmentation [19] an alternative criterion to replace minimum cut procedures. This is because the minimum cut in a graph with edge similarity weights creates a bipartition that tends to have one side very small in size. To correct for this unbalanced partition they proposed several types of objective functions, one of which is the *normalized cut*, which is a bipartition of V, $(S, \bar{S})$, minimizing:

$$\min_{S \subset V} \frac{C(S, \bar{S})}{d(S)} + \frac{C(S, \bar{S})}{d(\bar{S})}. \tag{48}$$

where $d_i = \sum_{[i,j] \in E} w_{ij}$ denote the sum of edge weights adjacent to node i. The weight of a subset of nodes $B \subseteq V$ is denoted by $d(B) = \sum_{j \in B} d_j$ referred to as the *volume* of B. Note that with the notation above $d(B) = C(B, V)$.
In such an objective function, the one ratio with the smaller value of $d()$ will dominate the objective value - it will be always at least $\frac{1}{2}$ of the objective value. Therefore, this type of objective function drives the segment S and its complement to be approximately of equal size. Indeed, like the balanced cut problem the problem was shown to be NP-hard, [19], by reduction from set partitioning.
Back to the original objective function, that is, equation 47. This objective function is equivalent to minimizing one term in (48). To see this, note that:

$$\begin{aligned} \frac{C(S, \bar{S})}{C(S, S)} &= \frac{C(S, \bar{S})}{d(S) - C(S, \bar{S})} \\ &= \frac{1}{\frac{d(S)}{C(S, \bar{S})} - 1}. \end{aligned}$$

Therefore, minimizing this ratio is equivalent to maximizing $\frac{d(S)}{C(S,\bar{S})}$ which in turn is equivalent to minimizing the reciprocal quantity $\frac{C(S,\bar{S})}{d(S)}$, which is the first term in equation 48. The optimal solution in the bipartition S will be the one set for which the value of the similarity within, $C(S, S)$, is the greater between the set and its complement.

55.1 Solving the normalized cut variant problem

A formulation for the problem is provided first, $\min_{S \subset V} \frac{C_1(S,\bar{S})}{C_2(S,\bar{S})}$. This is a slight generalization of normalized cut variant problem in permitting different similarity weights for the numerator, w_{ij}, and denominator, w'_{ij}.
We begin with an integer programming formulation of the problem. Let

$$x_i = \begin{cases} 1 & \text{if } i \in S \\ 0 & \text{if } i \in \bar{S}. \end{cases}$$

We define two additional sets of binary variables: $z_{ij} = 1$ if exactly one of i or j is in S; $y_{ij} = 1$ if both i or j are in S. Thus,

$$z_{ij} = \begin{cases} 1 & \text{if } i \in S, j \in \bar{S}, \text{ or } i \in \bar{S}, j \in S \\ 0 & \text{if } i, j \in S \text{ or } i, j \in \bar{S}, \end{cases}$$

,

$$y_{ij} = \begin{cases} 1 & \text{if } i, j \in S \\ 0 & \text{otherwise.} \end{cases}$$

With these variables the following is a valid formulation (NC) of the normalized cut variant problem:

$$\text{(NC)} \quad \min \frac{\sum w_{ij} z_{ij}}{\sum w'_{ij} y_{ij}} \tag{49}$$

$$\text{subject to } x_i - x_j \leq z_{ij} \quad \text{for all } [i,j] \in E \tag{50}$$

$$x_j - x_i \leq z_{ji} \quad \text{for all } [i,j] \in E \tag{51}$$

$$y_{ij} \leq x_i \quad \text{for all } [i,j] \in E \tag{52}$$

$$y_{ij} \leq x_j \tag{53}$$

$$1 \leq \sum_{[i,j] \in E} y_{ij} \leq |E| - 1 \tag{54}$$

$$x_j \text{ binary } j \in V \tag{55}$$

$$z_{ij} \text{ binary } \quad [i,j] \in E \tag{56}$$

$$y_{ij} \text{ binary } i, j \in V. \tag{57}$$

To verify the validity of the formulation notice that the objective function drives the values of z_{ij} to be as small as possible, and the values of y_{ij} to be as large as possible. Constraint (50) and (51) ensure z_{ij} cannot be 0 unless both endpoints i and j are in the same set. On the other hand, constraint (52) and (53) ensure that y_{ij} cannot be equal to 1 unless both endpoints i and j are in S.

Constraint (54) ensures that at least one edge is in the segment S and at least one edge is in the complement - the background. Otherwise the ratio is undefined in the first case, and the optimal solution is to choose the trivial solution $S = V$ in the second.

Excluding the sum constraint, the problem formulation (NC) is easily recognized as a monotone integer programming with up to three variables per inequality according to the definition provided in Hochbaum's [9]. That is, a problem with constraints of the form $ax - by \leq c + z$, where a and b are nonnegative and the variable z appears only in that constraint. Such linear optimization problems were shown there to be solvable as a minimum cut problem on a certain associated graph. We can then linearizing the objective function and solve the following λ-question which asks whether $\min_{S \subset V} \frac{C_1(S,\bar{S})}{C_2(S,\bar{S})} < \lambda$. The λ-*question* for the normalized cut' problem can be stated as the following linear optimization question:

$$\text{Is there a feasible subset } V' \subset V \text{ such that } \sum_{[i,j] \in E} w_{ij} z_{ij} - \lambda \sum_{[i,j] \in E} w'_{ij} y_{ij} < 0?$$

We note that the λ-question is the following *monotone* optimization problem,

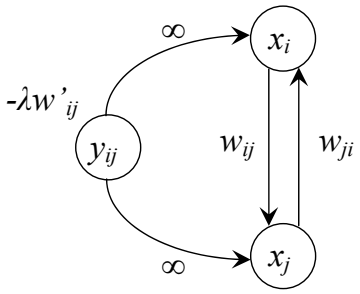


Figure 61: The basic gadget in the graph representation.

$$
\begin{array}{lll}
(\lambda\text{-NC}) & \min & \sum_{[i,j]\in E} w_{ij} z_{ij} - \lambda \sum_{[i,j]\in E} w'_{ij} y_{ij} \\
\text{subject to} & & x_i - x_j \le z_{ij} \quad \text{for all } [i,j] \in E \\
& & x_j - x_i \le z_{ji} \quad \text{for all } [i,j] \in E \\
& & y_{ij} \le x_i \quad \text{for all } [i,j] \in E \\
& & y_{ij} \le x_j \\
& & y_{i^*j^*} = 1 \text{ and } y_{i'j'} = 0 \\
& & x_j \text{ binary } j \in V \\
& & z_{ij} \text{ binary } \quad [i,j] \in E \\
& & y_{ij} \text{ binary } i,j \in V.
\end{array}
$$

If the optimal value of this problem is negative then the answer to the λ-question is yes, otherwise the answer is no. This problem (λ-NC) is an integer optimization problem on a totally unimodular constraint matrix. That means that we can solve the linear programming relaxation of this problem and get a basic optimal solution that is integer. Instead we will use a much more efficient algorithm described in [9] which relies on the monotone property of the constraints.

55.2 Solving the λ-question with a minimum cut procedure

We construct a directed graph $G' = (V', A')$ with a set of nodes V' that has a node for each variable x_i and a node for each variable y_{ij}. The nodes y_{ij} carry a negative weight of $-\lambda w_{ij}$. The arc from x_i to x_j has capacity w'_{ij} and so does the arc from x_j to x_i as in our problem $w_{ij} = w_{ji}$. The two arcs from each edge-node y_{ij} to the endpoint nodes x_i and x_j have infinite capacity. Figure 61 shows the basic gadget in the graph G' for each edge $[i,j] \in E$.

We claim that any finite cut in this graph, that has $y_{i^*j^*}$ on one side of the bipartition and $y_{i'j'}$ on the other, corresponds to a feasible solution to the problem λ-NC. Let the cut (S,T), where $T = V' \setminus S$, be of finite capacity $C(S,T)$. We set the value of the variable x_i or y_{ij} to be equal to 1 if the corresponding node is in S, and 0 otherwise. Because the cut is finite, then $y_{ij} = 1$ implies that $x_i = 1$ and $x_j = 1$.

Next we claim that for any finite cut the sum of the weights of the y_{ij} nodes in the source set and the capacity of the cut is equal to the objective value of problem λ-NC. Notice that if $x_i = 1$ and $x_j = 0$ then the arc from the node x_i to node x_j is in the cut and therefore the value of z_{ij} is equal to 1.

We next create a source node s and connect all y_{ij} nodes to the source with arcs of capacity $\lambda w'_{ij}$. The node $y_{i^*j^*}$ is then shrunk with a source node s and therefore also its endpoints nodes are

effectively shrunk with s. The node $y_{i'j'}$ and its endpoints nodes are analogously shrunk with the sink t. We denote this graph illustrated in Figure 62, G'_{st}.

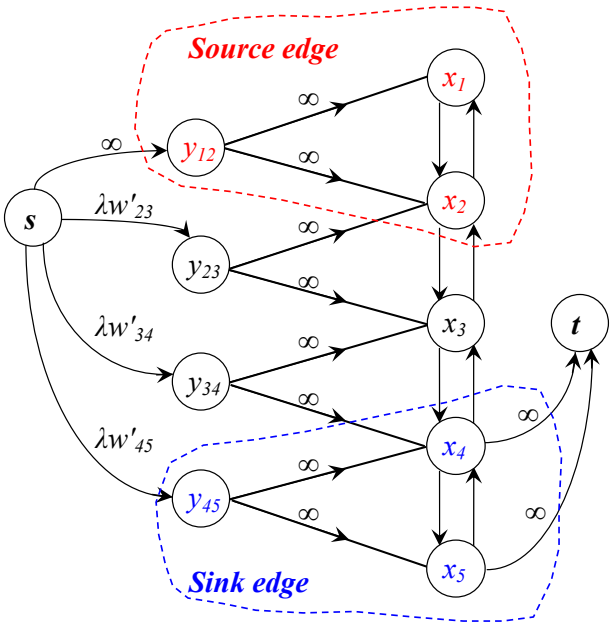


Figure 62: The graph G'_{st} with edge $[1,2]$ as source seed and edge $[4,5]$ as sink seed.

Theorem 55.1. *A minimum s,t-cut in the graph G'_{st}, (S,T), corresponds to an optimal solution to λ-NC by setting all the variables whose nodes belong to S to 1 and zero otherwise.*

Proof: Note that whenever a node y_{ij} is in the sink set T the arc connecting it to the source is included in the cut. Let the set of x variable nodes be denoted by V_x and the set of y variable nodes, excluding $y_{i^*j^*}$, be denoted by V_y. Let (S,T) be any finite cut in G'_{st} with $s \in S$ and $t \in T$ and capacity $C(S,T)$.

$$C(S,T) = \sum_{y_{ij} \in T \cap V_y} \lambda w'_{ij} + \sum_{i \in V_x \cap S, j \in V_x \cap T} w_{ij}$$

$$= \sum_{v \in V_y} \lambda w'_v - \sum_{y_{ij} \in S \cap V_y} \lambda w'_{ij} + \sum_{x_i \in V_x \cap S, x_j \in V_x \cap T} w_{ij}$$

$$= \lambda W' + [\sum_{i \in V_x \cap S, j \in V_x \cap T} w_{ij} - \sum_{y_{ij} \in S \cap V_y} \lambda w'_{ij}].$$

This proves that for a fixed constant $W' = \sum_{v \in V_y} w'_v$ the capacity of a cut is equal to a constant $W'\lambda$ plus the objective value corresponding to the feasible solution. Hence the partition (S,T) minimizing the capacity of the cut minimizes also the objective function of λ-NC. $\square$ Lec13

56 Duality of Max-Flow and MCNF Problems

56.1 Duality of Max-Flow Problem: Minimum Cut

Given $G = (V, E)$, capacities u_{ij} for all $(i, j) \in E$ and two special nodes $s, t \in V$, consider the formulation of the maximum flow problem. Let x_{ij} be the variable denoting the flow on arc (i, j).

$$\begin{array}{ll} \max & x_{ts} \\ \text{subject to} & \sum_i x_{ki} - \sum_j x_{jk} = 0 \quad \forall k \in V \\ & 0 \le x_{ij} \le u_{ij} \quad \forall (i,j) \in E. \end{array}$$

For dual variables, let $\{z_{ij}\}$ be the nonnegative dual variables associated with the capacity upper bounds constraints, and $\{\lambda_i\}$ be the variables associated with the flow balance constraints. The dual of the above formulation is given as follows:

$$\begin{array}{ll} \min & \sum_{(i,j)\in E} u_{ij} z_{ij} \\ \text{subject to} & z_{ij} - \lambda_i + \lambda_j \ge 0 \quad \forall (i,j) \in E \\ & \lambda_s - \lambda_t \ge 1 \\ & z_{ij} \ge 0 \qquad \forall (i,j) \in E. \end{array}$$

The dual problem has an infinite number of solutions: if (λ^*, z^*) is an optimal solution, then so is $(\lambda^* + \delta, z^*)$ for any constant δ. To avoid that we set $\lambda_t = 0$ (or to any other arbitrary value). Observe now that with this assignment there is an optimal solution with $\lambda_s = 1$ and a partition of the nodes into two sets: $S = \{i \in V | \lambda_i = 1\}$ and $\bar{S} = \{i \in V | \lambda_i = 0\}$.
The complementary slackness condition states that the primal and dual optimal solutions x^*, λ^*, z^* satisfy,

$$x^*_{ij} \cdot [z^*_{ij} - \lambda^*_i + \lambda^*_j] = 0$$

$$[u_{ij} - x^*_{ij}] \cdot z^*_{ij} = 0.$$

In an optimal solution $z^*_{ij} - \lambda^*_i + \lambda^*_j = 0$ so the first set of complementary slackness conditions do not provide any information on the primal variables $\{x_{ij}\}$. As for the second set, $z^*_{ij} = 0$ on all arcs other than the arcs in the cut $(S, \bar{S})$. So we can conclude that the cut arcs are saturated, but derive no further information on the flow on other arcs.
The only method known to date for solving the minimum cut problem requires finding a maximum flow first, and then recovering the cut partition by finding the set of nodes reachable from the source in the residual graph (or reachable from the sink in the reverse residual graph). That set is the source set of the cut, and the recovery can be done in linear time in the number of arcs, $O(m)$. On the other hand, if we are given a minimum cut, there is no efficient way of recovering the flow values on each arc other than essentially solving the max-flow problem from scratch. The only information given by the minimum cut, is the value of the maximum flow and the fact that the arcs on the cut are saturated. Beyond that, the flows have to be calculated with the same complexity as would be required without the knowledge of the minimum cut.

56.2 Duality of MCNF problem

Consider the following formulation of the Minimum Cost Network Flow Problem. Given a network $G = (V, E)$ with a cost c_{ij} and an upper bound u_{ij} associated with each directed arc (i, j), Also, b_i represent the supplies for every node in the network, where $b_i < 0$ implies a demand node. The

problem can be formulated as an LP in the following way:

$$\begin{array}{ll} \max & \sum_{j=1}^{n}\sum_{i=1}^{n} c_{ij}x_{ij} \\ \text{subject to} & \sum_{k\in V} x_{ik} - \sum_{k\in N} x_{ki} = b_i \quad \forall i \in V \\ & 0 \le x_{ij} \le u_{ij} \quad \forall (i,j) \in E \end{array}$$

Once again, let the dual variables be $\{z_{ij}\}$ as the nonnegative dual variables associated with the capacity upper bounds constraints, and $\{\lambda_i\}$ as the free variables associated with the flow balance constraints.

Hence the dual formulation of MCNF would be as follows:

$$\begin{array}{ll} \min & \sum_{(i,j)\in E} u_{ij}z_{ij} + \sum_{i\in V} \lambda_i b_i \\ \text{subject to} & z_{ij} - \lambda_i + \lambda_j \ge c_{ij} \quad \forall (i,j) \in E \\ & z_{ij} \ge 0 \quad \forall (i,j) \in E \end{array}$$

With the addition of the costs c_{ij} for every edge $i, j \in E$, the resulting monotone inequality in the dual fomulation now includes an extra constant term, c_{ij}.
Hence, the presence of a constant term in a monotone inequality in three variables (λ_i, λ_j and z_{ij}) is an indication of the formulation being a dual of a minimum cost network flow problem, as opposed to a maximum flow problem. This implies that the problem can no longer be solved as a minimum cut problem (as was the case with Max-Flow), but is still solvable in polynomial time because the constraint matrix is total unimodular (TUM).

57 Variant of Normalized Cut Problem

Consider a graph, $G = (V, E)$, with a weight w_{ij} assicated with each edge.
Let S be a subset of V and $C(S, \bar{S})$ be the capacity of the cut between S and the rest of the graph ($\bar{S}$). Also let C(S,S) be the sum of the weights of all the edges contained in the subgraph composed of S, and C(S,V) be the sum of edges with atleast one endpoint in S.
The variant of the normalized cut problem (NC') is given as follows:

$$\min_{\emptyset \neq S \subset V} \quad \frac{C(S,\bar{S})}{C(S,S)} \qquad (NC')$$

Let q_i be defined as the sum of weights of all adjacent edges to $i \in V$. Hence, based on the above definitions, the following relationships exist:

$$\begin{array}{l} q_i = \sum_{j|(i,j)\in E} w_{ij} \\ C(S,V) = C(S,S) + C(S,\bar{S}) \\ \sum_{i\in S} q_i = \sum_{i\in S}\sum_{j|(i,j)\in E} w_{ij} = C(S,S) + C(S,V) \end{array}$$

The first equality reiterates the definition of q_i, while the second equality shows that the sum of the weights of the edges with one node in S is equal to the sum of the edge weights with both nodes in S and exactly one node in S. Combining the first two relationships, we obtain the third equality since C(S,S) gets repeated twice (for both $i, j \in S$).
Based on the above-defined relationships, one can now observe the following equivalences:

$$\min_{\emptyset \neq S \subset V} \frac{C(S,\bar{S})}{\sum_{i\in S} q_i} \quad \Leftrightarrow \quad \frac{C(S,\bar{S})}{2C(S,S)+C(S,\bar{S})} \quad \Leftrightarrow \quad \frac{1}{2\frac{C(S,S)}{C(S,\bar{S})}+1}$$

Since,

$$\frac{C(S,\bar{S})}{C(S,S)} \quad \propto \quad \frac{1}{2\frac{C(S,S)}{C(S,\bar{S})}+1}$$

this leads to an alternate formulation for NC':

$$\min_{\emptyset \neq S \subset V} \quad \frac{C(S,\bar{S})}{\sum_{i \in S} q_i}$$

In order to linearize the ratio form of NC', the $\lambda - question$ can be asked as follows:

$$\min_{\emptyset \neq S \subset V} \quad C(S,\bar{S}) - \lambda \sum_{i \in S} q_i < 0$$

57.1 Problem Setup

Consider the following set of decision variables:

$$x_i = \begin{cases} 1 & \text{if } i \in \text{S} \\ 0 & \text{otherwise} \end{cases}$$

$$z_{ij} = \begin{cases} 1 & \text{if } i \in \text{S, j} \in \bar{S} \\ 0 & \text{otherwise} \end{cases}$$

Therefore, the variant of the normalized cut problem can be formulated as follows:

$$\begin{array}{ll} \min & \sum_{i,j \in E} w_{ij} z_{ij} - \lambda \sum_{j \in V} q_j x_j \\ \text{subject to} & x_i - x_j \leq z_{ij} \quad \forall (i,j) \in E \\ & 0 \leq x_i \leq 1 \quad \forall i \in V \\ & 0 \leq z_{ij} \qquad \forall (i,j) \in E \end{array}$$

Observe the following points about the formulation:

- Since the constraint matrix is totally unimodular, x_j need not be restricted as an integer, while z_{ij} only needs to be constrained with a lower bound. Hence, the problem can be solved as an LP.
- In general, the parameters of the node weights, q_j and edge weights, w_{ij} need not be related to each other, i.e. "$q_i = \sum_{j|(i,j) \in E} w_{ij}$" is not required for the problem to be solved. However, while q_j can be negative or positive, negative w_{ij} makes the problem *NP*-hard.
- In a *closure problem*, the *closure* constraint is of the following form:

  $$x_i \leq x_j \quad \Leftrightarrow \quad x_i - x_j \leq 0$$

  However, in our formulation, we relax the closure constraint. Hence, for any pair of of vertices (i,j) which violates the closure property, there is an associated penalty cost, w_{ij}, introduced through the means of the variable z_{ij}:

  $$x_i - x_j \leq z_{ij}$$

- The above-mentioned formulation, with a transformation of *minimization* to *maximization*, can be seen as an instance of the maximum s-excess problem (see section 57.3).

Before discussing the solution methodology for the variant of the normalized cut problem, let us review the concepts of the maximum closure and the s-excess problems, in order to appreciate the relationship between them.

57.2 Review of Maximum Closure Problem

Definition 57.1. *Given a directed graph $G = (V, E)$, a subset of the nodes $D \subseteq V$ is* closed*, if for every node in D, its successors are also in D.*

Consider a directed graph $G = (V, E)$ where every node $i \in V$ has a corresponding weight w_i. The *maximum closure problem* is to find a closed set $V' \subseteq V$ with maximum total weight. That is, the maximum closure problem is:

Problem Name: *Maximum closure*

Instance: *Given a directed graph $G = (V, E)$, and node weights (positive or negative) w_i for all $i \in V$.*

Optimization Problem: *find a closed subset of nodes $V' \subseteq V$ such that $\sum_{i \in V'} w_i$ is maximum.*

We can formulate the maximum closure problem as an integer linear program (ILP) as follows.

$$\begin{aligned} \max \quad & \sum_{i \in V} w_i x_i \\ \text{s.t.} \quad & x_i \leq x_j \quad \forall (i,j) \in E \\ & x_i \in \{0,1\} \quad \forall i \in V \end{aligned}$$

where x_i is a binary variable that takes the value 1 if node i is in the maximum closure, and 0 otherwise. The first set of constraints imposes the requirement that for every node i included in the set, its successor is also in the set. Observe that since every row has exactly one 1 and one -1, the constraint matrix is totally unimodular (TUM). Therefore, its linear relaxation formulation results in integer solutions. Specifically, this structure also indicates that the problem is the dual of a flow problem.
Let $V^+ \equiv \{j \in V | w_i > 0\}$, and $V^- \equiv \{i \in V | w_i \leq 0\}$. We construct an s,t-graph G_{st} as follows. Given the graph $G = (V, E)$ we set the capacity of all arcs in E equal to ∞. We add a source s, a sink t, set E_s of arcs from s to all nodes $i \in V^+$ (with capacity $u_{s,i} = w_i$), and set E_t of arcs from all nodes $j \in V^-$ to t (with capacity $u_{j,t} = |w_j| = -w_j$). The graph $G_{st} = \{V \cup \{s,t\}, E \cup E_s \cup E_t\}$ is a *closure graph* (a closure graph is a graph with a source, a sink, and with all finite capacity arcs adjacent only to either the source or the sink.) This construction is illustrated in Figure 63.

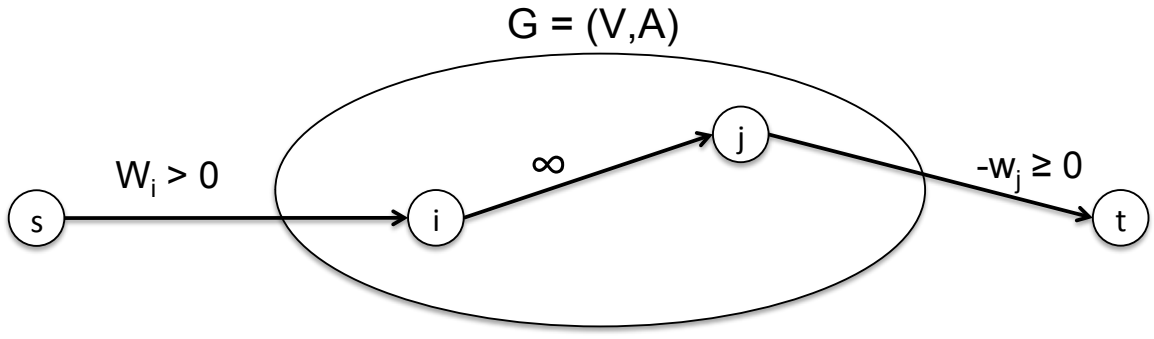


Figure 63: Visual representation of G_{st}.

Claim 57.2. *If $(s \cup S, t \cup T)$ is a finite $s - t$ cut on $G_{s,t}$, then S is a closed set on G.*

Proof. Assume by contradiction that S is not closed. This means that there must be an arc $(i,j) \in E$ such that $i \in S$ and $j \in T$. This arc must be on the cut (S,T), and by construction $u_{i,j} = \infty$, which is a contradiction on the cut being finite. □

Theorem 57.3. *If $(s \cup S, t \cup T)$ is an optimal solution to the minimum $s-t$ cut problem on $G_{s,t}$, then S is a maximum closed set on G.*

Proof.

$$\begin{aligned} C(s \cup S, t \cup T) &= \sum_{(s,i)\in E_{st},\ i\in T} u_{s,i} + \sum_{(j,t)\in E_{st},\ j\in S} u_{j,t} \\ &= \sum_{i \in T \cap V^+} w_i + \sum_{j \in S \cap V^-} -w_j \\ &= \sum_{i\in V^+} w_i - \sum_{i\in S\cap V^+} w_i - \sum_{j\in S\cap V^-} w_j \\ &= W^+ - \sum_{i\in S} w_i \end{aligned}$$

(Where $W^+ = \sum_{i\in V^+} w_i$, which is a constant.) This implies that minimizing $C(s \cup S, t \cup T)$ is equivalent to minimizing $W^+ - \sum_{i\in S} w_i$, which is in turn equivalent to $\max_{S\subseteq V} \sum_{i\in S} w_i$. Therefore, any source set S that minimizes the cut capacity also maximizes the sum of the weights of the nodes in S. Since by Claim 57.2 any source set of an $s-t$ cut in $G_{s,t}$ is closed, we conclude that S is a maximum closed set on G. □

57.3 Review of Maximum s-Excess Problem

Problem Name: *Maximum s-Excess*

Instance: *Given a directed graph $G = (V,E)$, node weights (positive or negative) w_i for all $i \in V$, and nonnegative arc weights c_{ij} for all $(i,j) \in E$.*

Optimization Problem: *Find a subset of nodes $S \subseteq V$ such that $\sum_{i\in S} w_i - \sum_{i\in S, j\in \bar{S}} c_{ij}$ is maximum.*

The s-excess problem is closely related to the maximum flow and minimum cut problems as shown next.

Lemma 57.4. *For $S \subseteq V$, $\{s\} \cup S$ is the source set of a minimum cut in G_{st} if and only if S is a set of maximum s-excess capacity $C(\{s\}, S) - C(S, \bar{S} \cup \{t\})$ in the graph G.*

Proof: We rewrite the objective function in the maximum s-excess problem:

$$\begin{aligned} \max_{S\subseteq V}[C(\{s\},S) - C(S,\bar{S}\cup\{t\})] &= \max_{S\subseteq V}[C(\{s\},V) - C(\{s\},\bar{S}) - C(S,\bar{S}\cup\{t\})] \\ &= C(\{s\},V) - \min_{S\subseteq V}[C(\{s\},\bar{S}) + C(S,\bar{S}\cup\{t\})]. \end{aligned}$$

In the last expression the term $C(\{s\},V)$ is a constant from which the minimum cut value is subtracted. Thus the set S maximizing the s-excess is also the source set of a minimum cut and, vice versa – the source set of a minimum cut also maximizes the s-excess. □

57.4 Relationship between s-excess and maximum closure problems

The s-excess problem is a natural extension of the maximum closure problem. The maximum closure problem is to find in a node weighted graph a subset of nodes that forms a closed set, (i.e. with each node in a closed set all its successors are in the set as well), so that the total weight of the nodes in the closed subset is maximized. The s-excess problem may be viewed as a maximum closure problem with a *relaxation* of the closure requirement: Nodes that are successors of other nodes in S (i.e. that have arcs originating from node of S to these nodes) may be excluded from the set, but at a penalty that is equal to the capacity of those arcs.
With the discussion in the previous section we conclude that any maximum flow or minimum cut algorithm solves the maximum s-excess problem: The source set of a minimum cut is a maximum s-excess set.

57.5 Solution Approach for Variant of the Normalized Cut problem

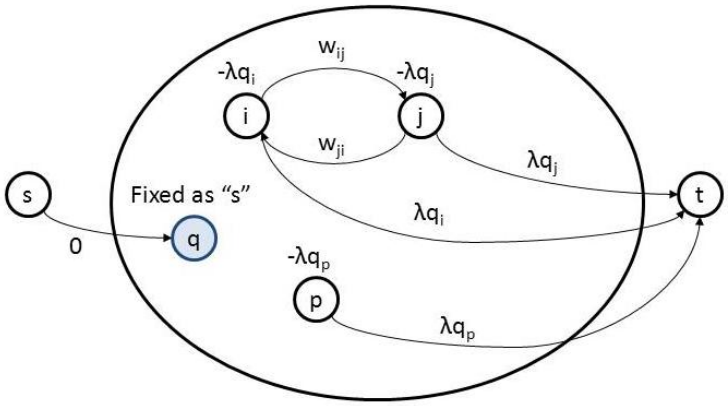


Figure 64: $s-t$ graph for the Variant of the Normalized Cut.

Given the graph $G=(V,E)$, in order to construct a $s-t$ graph, a source node has to be generated by randomly selecting a node $q \in V$, since all the node weights are negative, i.e. $-\lambda q_j < 0 \ \forall j \in V$. Let $V^+ \equiv \{q\}$, and $V^- \equiv \{j \ \forall \ V \backslash \{q\}\}$. We construct an $s-t$ graph G_{st} as follows: we set the capacity of all arcs in E equal to w_{ij}. We add a source s, a sink t, set E_s of arcs from s to q (with capacity $u_{s,q}=0$), and set E_t of arcs from all nodes $j \in V^-$ to t (with capacity $u_{j,t}=-\lambda q_j$). Hence figure 64 represents a generalization of the closure graph (figure 63), wherein the edge weights can have finite costs, w_{ij}, as opposed to ∞ in (figure 63).

With the modified graph G_{st}, the variant of the normalized cut problem can be solved as follows:

1. Select a source node, $q \in V$ and solve the parametric cut problem: the partition contains at least q in the source set.

2. Since q is chosen at random, iterate on different values of q to find different solutions to the $\lambda - question$.

3. The optimal value would be the minimum amongst the different solutions.

The parametric cut problem is solved for all breakpoints of λ (which are bounded by the number of nodes, n) with the same complexity as a single cut. Since the capacities are linear functions of λ, the parametric cut problem, and hence the normalized cut problem, is strongly polynomial solvable with the complexity of $O(mincut(\|V\|, \|E\|))$.
Lec14)

58 Markov Random Fields

Markov Random Fields can be used for image reconstruction or segmentation. In image reconstruction, the objective is to assign a color to each pixel such that the assignment approximates the noisy observations, but is similar to neighboring pixels. The reconstructed image is a smoothed version of the observed image.
A graph $G = (V, A)$ represents the image being considered. The vertices of the graph represent pixels in the image. The arcs in the graph represent adjacency relations between pixels; if pixels i and j are adjacent, then G will have directed arcs (i, j) and (j, i). Graph G allows image reconstruction to be cast as an MRF problem (Eq. 58). Let each vertex i have an observed color g_i and an assigned color x_i. Function G_i is the deviation cost, and penalizes the color assignment of each pixel according to its deviation from the observed value of the same pixel. Function F_{ij} is the separation cost, and penalizes the difference in color assignments between neighboring pixels.

$$\begin{aligned} \min \quad & \sum_{i \in V} G_i(g_i, x_i) + \sum_{(i,j) \in A} F_{ij}(x_i - x_j) \\ \textit{subject to} \quad & x_i \in [1, k] \quad \forall i \in V \end{aligned} \tag{58}$$

Let $z_{ij} = x_i - x_j$. When the separation cost F_{ij} is monotone increasing, this equality ($z_{ij} = x_i - x_j$) constraint is equivalent to an inequality constraint ($x_i - x_j \leq z_{ij}$). Eq. 58 can then be stated as the image segmentation problem [10] of Eq. 59.

$$\begin{aligned} \min \quad & \sum_{i \in V} G_i(x_i) + \sum_{(i,j) \in A} F_{ij}(z_{ij}) \\ \textit{subject to} \quad & l_i \leq x_i \leq u_i \qquad \forall i \in V \\ & x_i - x_j \leq z_{ij} \qquad \forall (i,j) \in A \\ & z_{ij} \geq 0 \qquad \forall (i,j) \in A \end{aligned}$$

1. *G_i convex F_{ij} linear :* There is no known strongly polynomial algorithm for solving this case. One solution to this case consists of generating all n integer node shifting breakpoints using a parametric cut, and then finding all the integer minima of the convex deviation costs for each range between breakpoints [10].

   Let $T(m, n)$ denote the cost of solving minimum s,t cut on a graph with n nodes and m arcs; for example $T(m, n) = O(mn \log n^2/m)$ if using the push-relabel algorithm.

   Let $U = \max_{\forall i,j \in V}(u_i - l_j)$. The convex deviation costs can be a function of any value in U. Using a binary search over range U, n convex deviation costs must be evaluated at each step of the search.

The time complexity is given by:

$$O\left(T\left(m,n\right)+n\log U\right) \tag{59}$$

2. *G_i convex F_{ij} convex :* In this case, the objective is the sum of separable convex cost functions (deviation cost and separation cost). This problem can be solved as the convex cost integer dual network flow problem [13, 4, 5]. With the separation cost coefficients given by c_{ij}, the complexity is:

$$\begin{aligned} &O\left(T\left(m,n\right)\log(nC)\right) \\ &C=\max_{(i,j)\in A}\left(c_{ij},k\right) \end{aligned} \tag{60}$$

3. *G_i nonlinear F_{ij} convex :* Consider a special case of MRF problem, where $F_{ij}=\overset{\infty}{M}z_{ij}$, with $\overset{\infty}{M}$ being a very large number. In this special case, all z_{ij} variables will be optimally assigned zero, and the problem reduces to Eq. 61. With nonlinear G_i, even this special case is *NP*-hard since it is necessary to evaluate $G_i(x_i)$ for all x_i. In this special case, if G_i were convex, the problem would be convex cost closure, and could be solved using minimum cut [22, 11].

$$\begin{aligned} \min \quad & \sum_{i\in V} G_i(x_i) \\ \textit{subject to} \quad & x_i \le x_j \\ & l_i \le x_i \le u_i \end{aligned} \tag{61}$$

4. *G_i linear, F_{ij} nonlinear:* With nonlinear separation costs, the problem is *NP*-hard, as is stated in [10]. One particular discontinuous separation function that leads to *NP*-hardness is given in [2].

59 Examples of 2 vs. 3 in Combinatorial Optimization

Many problems in combinatorial optimization change from polynomial complexity to being *NP*-hard when some aspect of the input goes from size 2 to size 3. The following subsections present problem instances where this occurs.

59.1 Edge Packing vs. Vertex Packing

Edge packing is the same as the maximum matching problem. Given a graph $G=(V,E)$, the goal of edge packing is to find a maximum subset of edges $S\subseteq E$ such that each node is incident to no more than one. Edge packing is given by Eq. 62. Note that each x_{ij} appears in exactly two constraints, so the constraint matrix is totally unimodular. Edge packing is polynomial time solvable; for example, Micali and Vazirani present an algorithm for solving edge packing in $O(\sqrt{V}E)$ [20].

$$x_{ij}=\begin{cases}1 & (ij)\in S\\ 0 & otherwise\end{cases}.$$

$$\begin{aligned} \max \quad & \sum_{(i,j)\in E} x_{ij} \\ \textit{subject to} \quad & \sum_{ij\in E} x_{ij} \leq 1 \quad \forall j \in V \\ & x_{ij} \in \{0,1\} \end{aligned} \tag{62}$$

Vertex packing is another name for independent set. Given a graph $G = (V, E)$, the goal of vertex packing is to choose a maximum subset $S \subseteq V$ of vertices such that no two share an edge. More formally, vertex packing is given by Eq. 63. Vertex packing is *NP*-hard if the degree of G is 3 or greater, and has polynomial complexity on graphs of degree 2.

$$x_i = \begin{cases} 1 & if i \in S \\ 0 & otherwise \end{cases}.$$

$$\begin{aligned} \max \quad & \sum_{i\in V} x_i \\ \textit{subject to} \quad & x_i + x_j \leq 1 \quad \forall (i,j) \in E \\ & x_i \in \{0,1\} \end{aligned} \tag{63}$$

59.2 Chinese Postman Problem vs. Traveling Salesman Problem

The Chinese Postman Problem (CPP) is given by Eq. 64. Both the undirected and directed cases of CPP can be solved in polynomial time.

$$x_{ij} = \begin{cases} 1 & (i,j) \in optimaltour \\ 0 & otherwise \end{cases}.$$

$$\begin{aligned} \min \quad & \sum_{i,j\in A} c_{ij} x_{ij} \\ \textit{subject to} \quad & x_{ij} \geq 1 \quad \forall (i,j) \in A \\ & \sum_{\substack{(i,j)\in A \\ i\in V}} x_{ij} - \sum_{\substack{(j,k)\in A \\ k\in V}} x_{jk} = 0 \quad \forall j \in V \\ & x_i \in \{0,1\} \end{aligned} \tag{64}$$

CPP on an undirected graph G can be solved via formulation as a matching problem. Note that at each vertex of odd degree, some edges must be traversed multiple times. Create fully connected graph $G^{ODD} = (V, E)$, where V is the set of all odd-degree vertices in G. The edge weights on E are all pairs shortest paths. Note that all-pairs shortest path is found in $O(n^3)$ using dynamic programming [6, 25]. Find a minimum matching on G^{ODD}; the corresponding edges from original graph G are the edges to be traversed twice. An Eulerian tour of G completes the solution to the undirected CPP problem.

CPP on a directed graph can be solved via formulation as a transportation problem. Created a subgraph $\hat{G} = (V, A)$ consisting of all vertices from G for which the indegree and outdegree differ. The demand of each vertex in $\hat{G}$ is equal to its $outdegree - indegree$. The arcs are the shortest paths from each negative demand node to each positive demand node. The solution to the transportation problem is then the set of arcs that must be traversed twice in the optimal CPP tour. The remainder of the solution is an Eulerian path in G.
Traveling Salesman Problem (TSP) is *NP*-hard. To create directed graph $G = (V, A)$, replace each undirected edge (i, j) with 2 directed arcs (i, j) and (j, i). The solution to TSP is a minimum cost vertex tour of G, with constraints on indegree and outdegree of each vertex, and constraints for subtour elimination (Eq. 65). The *NP*-hard problem of finding a Hamiltonian circuit [7] reduces to TSP.

$$x_{ij} = \begin{cases} 1 & (i,j) \in optimaltour \\ 0 & otherwise \end{cases}.$$

$$\begin{aligned} \min \quad & \sum_{i,j \in A} c_{ij} x_{ij} \\ subject\ to \quad & \sum_{\substack{(i,j)\in A \\ i \in V}} x_{ij} = 1 \quad \forall j \in V \\ & \sum_{\substack{(i,j)\in A \\ j \in V}} x_{ij} = 1 \quad \forall i \in V \\ & \sum_{\substack{i \in S \\ j \in V/S}} x_{ij} \geq 1 \quad \forall S \subset V \\ & x_i \in \{0,1\} \end{aligned} \tag{65}$$

59.3 2SAT vs. 3SAT

A 2SAT clause is of the form $x_i \vee x_j$. In a satisfying assignment, at least one of the two literals in the clause must be true, so the implications $\neg x_i \implies x_j$ and $\neg x_j \implies x_i$ must hold. An implication graph $G = (V, A)$ can thus be constructed, with an arc leading from each literal to its implication. A 2SAT problem is satisfiable if and only if no literal implies its negation. Stated differently, a 2SAT problem is satisfiable if and only if no variable appears as both a positive and negative literal in the same strongly connected component[15] of the graph [18]. Strongly connected components can be found in linear time using Tarjan's algorithm [24]. Therefore, 2SAT can be solved in $O(m)$, where m is the number of clauses [18].
3SAT is well known to be *NP*-Hard [3], and this can be shown by reducing satisfiability[16] to 3SAT [15]. Unlike 2SAT, a 3SAT problem cannot be translated into an implication graph, since an unsatisfied literal only implies that one or more of the two other literals in the clause are satisfied.

59.4 Even Multivertex Cover vs. Vertex Cover

An even multivertex cover of a graph $G = (V, E)$ is a minimum cost assignment of integers to vertices such that the assignments to the two vertices incident on each edge sum to an even number

[15] A strongly connected component is a subgraph in which there is a path from each vertex to every other vertex
[16] with unrestricted clause size

(Eq. 66). Let x_i represent the assignment to vertex i. Recall that the LP relaxation of vertex cover yields a half-integral solution. The following steps outline how the half-integral solution to vertex cover yields a integral solution to even multivertex cover. Since the LP relaxation can be solved in polynomial time, then even multivertex cover can be solved in polynomial time.

1. Convert an even multivertex cover to vertex cover by changing the 2 to 1 on the right hand side of the first constraint.
2. Solve the LP relaxation of vertex cover, yielding a half-integral solution.
3. Multiply the solution by 2, thus changing it from half-integral to integral. Each edge that was covered once is now covered twice.

$$\begin{aligned} \min \quad & \sum_{i \in V} w_i x_i \\ \textit{subject to} \quad & x_i + x_j \geq 2 \quad \forall (i,j) \in E \\ & x_i \geq 0 \quad \forall i \in V \\ & x_i \in \mathbb{Z} \end{aligned} \tag{66}$$

The *NP*-hard vertex cover problem on graph $G = (V, E)$ is to find a minimum cost subset of vertices $S \subseteq V$ that cover each edge (Eq. 67). While the LP relaxation can be solved in polynomial time to produce a half-integral solution, there is no known polynomial time algorithm for optimally rounding it to an integer solution. This is why vertex cover is more difficult than edge cover. To show that vertex cover is *NP*-hard for cubic and higher degree graphs, Karp uses a reduction from clique [15].

$$x_i = \begin{cases} 1 & v_i \in S \\ 0 & otherwise \end{cases}.$$

$$\begin{aligned} \min \quad & \sum_{i \in V} w_i x_i \\ \textit{subject to} \quad & x_i + x_j \geq 1 \quad \forall (i,j) \in E \\ & x_i \in \{0,1\} \end{aligned} \tag{67}$$

59.5 Edge Cover vs. Vertex Cover

Edge cover is polynomial time solvable. Given a graph $G = (V, E)$, a minimum edge cover is a minimum subset $S \subseteq E$, such that all vertices in the graph are incident on one or more edges in S. Unweighted minimum edge cover is then given by Eq. 68.

$$x_{ij} = \begin{cases} 1 & (i,j) \in S \\ 0 & otherwise \end{cases}.$$

$$\begin{aligned} \min \quad & \sum_{(i,j) \in E} x_{ij} \\ \textit{subject to} \quad & \sum_{(i,j) \in E} x_{ij} \geq 1 \quad \forall i \in V \\ & x_{ij} \in \{0,1\} \end{aligned} \tag{68}$$

Edge cover can be solved in polynomial time by first finding a maximum matching (Eq. 69). Let subset $M \subset E$ be a maximum matching of G. If M happens to be a perfect matching, then it is also a minimum edge cover. Otherwise, M contains the maximum possible number of edges such that each one covers two unique vertices. The minimum edge cover can be created by adding to M, one additional edge to cover each vertex not covered by M. Since maximum matching can be solved in $O(\sqrt{V}E)$ [20], edge cover can also be solved with this same complexity. Weighted edge cover is also polynomial time solvable, but extra steps are required; this was not covered in lecture.

$$x_{ij} = \begin{cases} 1 & (i,j) \in M \\ 0 & otherwise \end{cases}.$$

$$\begin{aligned} \max \quad & \sum_{(i,j)\in E} x_{ij} \\ \textit{subject to} \quad & \sum_{(i,j)\in E} x_{ij} \leq 1 \quad \forall i \in V \\ & x_{ij} \in \{0,1\} \end{aligned} \tag{69}$$

Recall from Sec. 59.4 that vertex cover is *NP*-hard for cubic and higher degree graphs. As previously mentioned, this can be shown using reduction from clique [15].

59.6 IP Feasibility: 2 vs. 3 Variables per Inequality

IP feasibility with 2 variables per inequality can be solved in pseudo-polynomial time by binarizing. Initially, each constraint is of the form:

$$a_{ij}x_i + a_{kj}x_k \leq b_k$$

Each integer variable x_i is replaced by U binary variables $x_{il}(l = 1, \ldots U)$. Let x_{il} have the following meaning: x_{ij} is 1 if and only if $x_i \geq l$. Naturally, the constraint of Eq. 70 exists.

$$x_{il} \leq x_{i,l+1} \quad \forall l \in 0, 1, \ldots U-1 \tag{70}$$

Let α_{kl} be defined as follows:

$$\alpha_{kl} = \lfloor \frac{b_k - la_{ij}}{b_{ij}} \rfloor - 1 \quad \forall l \in 0, 1 \ldots U$$

An assignment is feasible if and only if one of the following inequalities hold for each value of $l(l = 1, \ldots U)$:

$$x_i \leq l - 1 \quad or \quad x_j \leq \alpha_{kl} + 1$$

Using the binary representation of x_i, the above inequalities are equivalent to:

$$x_{il} = 0 \quad or \quad x_{j,\alpha_{kl}} = 0$$

The above disjunction can be rewritten as:

$$x_{il} + x_{j,\alpha_{kl}} \leq 1 \tag{71}$$

In total, the problem now contains only 2 binary variables per constraint (Eq. 70 and Eq. 71), and all coefficients are either 1 or -1. Finding an IP feasible solution is equivalent to solving 2SAT, since each constraint from Eq. 70 and Eq. 71 is equivalent to a SAT clause with two binary variables. Thus, IP feasibility problem reduces to 2SAT in polynomial time. As discussed in Sec. 59.3, 2SAT can be solved in time that is linear to the number of clauses. However, since the number of 2SAT clauses is a function of U, the problem is only solvable in pseudo-polynomial time $O(mU)$.
With three variables per inequality, the IP feasibility problem cannot be reduced to 2SAT. This problem is a general IP and is *NP*-hard.

References

[1] D. Greig B. Porteous, A. Seheult. Exact maximum a posteriori estimation for binary images. *Journal of the Royal Statistical Society, Series B*, 51(2):271–279, 1989.

[2] Y. Boykov, O. Veksler, and R. Zabih. A new algorithm for energy minimization with discontinuities. In *Energy Minimization Methods in Computer Vision and Pattern Recognition*, pages 730–730. Springer, 1999.

[3] S. A. Cook. The complexity of theorem-proving procedures. In *Proceedings of the third annual ACM symposium on Theory of computing*, page 158. ACM, 1971.

[4] R. K. Ahuja D. S. Hochbaum, J. B. Orlin. Solving the convex cost integer dual network flow problem. *Management Science*, 49(7):950–964, 2003.

[5] R. K. Ahuja D. S. Hochbaum, J. B. Orlin. A cut based algorithm for the convex dual of the minimum cost network flow problem. *Algorithmica*, 39(3):189–208, 2004.

[6] R.W. Floyd. Algorithm 97: shortest path. *Communications of the ACM*, 5(6):345, 1962.

[7] M. R. Garey and D. S. Johnson. *Computers and intractability*. Freeman San Francisco, 1979.

[8] D. S. Hochbaum. An efficient algorithm for image segmentation, markov random fields and related problems. *Journal of the ACM (JACM)*, 48(4):686–701, 2001.

[9] D. S. Hochbaum. Solving integer programs over monotone inequalities in three variables: A framework for half integrality and good approximations. *European Journal of Operational Research*, 140(2):291–321, 2002.

[10] D. S. Hochbaum. The pseudoflow algorithm: A new algorithm for the maximum flow problem. *Operations Research*, 58(4):992–1009, July-Aug 2008.

[11] D. S. Hochbaum. Polynomial time algorithms for ratio regions and a variant of normalized cut. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 32(5):889–898, 2010.

[12] D. S. Hochbaum and M. Queyranne. Minimizing a convex cost closure set. *SIAM Journal of Discrete Math*, 16(2):192–207, 2003.

[13] D. S. Hochbaum and J. G. Shanthikumar. Convex separable optimization is not much harder than linear optimization. *Journal of the ACM (JACM)*, 37(4):843–862, 1990.

[14] T. B. Johnson. *Optimum Open Pit Mine Production Technology*. PhD thesis, Operations Research Center, University of California, Berkeley, 1968.

[15] R. M. Karp. Reducibility among combinatorial problems. *50 Years of Integer Programming 1958-2008*, pages 219–241, 1972.

[16] Joseph B. Kruskal. On the shortest spanning subtree of a graph and the traveling salesman problem. *Proceedings of the American Mathematical Society*, 7(1):48–50, February 1956.

[17] G. Gallo M. D. Grigoriadis, R. E. Tarjan. A fast parametric maximum flow algorithm and applications. *SIAM Journal on Computing*, 18(30), 1989.

[18] B. Aspvall M. Plass, R. Tarjan. A linear-time algorithm for testing the truth of certain quantified boolean formulas. *Information Processing Letters*, 8:121–123, 1979.

[19] J. Malik and J. Shi. Normalized cuts and image segmentation. *IEEE Trans. Pattern Anal. Mach. Intell.*, 22(8):888–905, 2000.

[20] S. Micali and V.V. Vazirani. An O (4 |V| |E|) algorithm for finding maximum matching in general graphs. In *Foundations of Computer Science, 1980., 21st Annual Symposium on*, pages 17–27, 1980.

[21] Christos H. Papadimitriou and Kenneth Steiglitz. *Combinatorial Optimization: Algorithms and Complexity*. Prentice-Hall.

[22] J. C. Picard. Maximal closure of a graph and applications to combinatorial problems. *Management Science*, 22:1268–1272, 1976.

[23] R. C. Prim. Shortest connection networks and some generalizations. *Bell Systems Technical Journal*, pages 1389–1401, November 1957.

[24] R. Tarjan. Depth-first search and linear graph algorithms. In *Switching and Automata Theory, 1971., 12th Annual Symposium on*, pages 114–121, 1971.

[25] S. Warshall. A theorem on boolean matrices. *Journal of the ACM (JACM)*, 9(1):11–12, 1962.