Contents

1 The Linear Programming Problem

- The variables whose values are to be decided in some optimal fashion are referred to as **decision variables** and usually denoted as

$$x_j,\ j = 1, 2, \dots, n \tag{1}$$

- In linear programming the objective is to maximize of minimize some linear function of the decision variables

$$\varsigma = c_1x_1 + c_2x_2 + \cdots + c_nx_n \tag{2}$$

- This function is called the **objective function**
- The constrains of the problem are on the following form

$$a_1x_1 + a_2x_2 + \cdots + a_nx_n \left\{ \begin{array}{c} \leq \\ = \\ \geq \end{array} \right\} b \tag{3}$$

- One can convert constraints from one form to another
  - An inequality constraint can be converted to an equality constraint by adding a slack variable $w \geq 0$ or $w \leq 0$

- An equality constraint can be converted to two inequality constraints
- It is preferred to pose the inequality as less-thans to stipulate all the decision variables be nonnegative

- A linear programming problem can be formulated as follows

$$\begin{array}{llllll}
\text{maximize} & c_1x_1 + & c_2x_2 + \cdots + & c_nx_n & & \\
\text{subject to} & a_{11}x_1 + & a_{12}x_2 + \cdots + & a_{1n}x_n & \leq & b_1 \\
& a_{21}x_1 + & a_{22}x_2 + \cdots + & a_{2n}x_n & \leq & b_2 \\
& & \vdots & & & \\
& a_{m1}x_1 + & a_{m2}x_2 + \cdots + & a_{mn}x_n & \leq & b_m \\
& & & x_1,\ x_2,\ \ldots\ x_n & \geq & 0.
\end{array}$$

- Linear programs formulated this way is refereed to as linear programs in standard form
  - m is used to denote the number of constrains
  - n is used to denote the number of decision variables
- A proposal of specific values for the decision variables is called a **solution**
- A solution $(x_1, x_2, \ldots, x_n)$ is called **feasible** if it satisfies all of the constraints
  - It is called **optimal** if it also attains the desired maximum
  - Problems are **infeasible** when there exists no feasible solution
  - Problems is **unbounded** if it has feasible solution with arbitrarily large objective values

2 The Simplex Method

2.1 Problem

- Given a general linear programming problem presented in standard form:

$$\begin{array}{lll} \text{maximize} & \sum_{j=1}^{n} c_j x_j & \\ \text{subject to} & \sum_{j=1}^{n} a_{ij} x_j \leq b_i & i = 1, 2, \ldots, m \\ & x_j \geq 0 & j = 1, 2, \ldots, n. \end{array}$$

- The first task is to introduce slack variables and a name for the objective function value

$$\begin{aligned} \zeta &= \sum_{j=1}^{n} c_j x_j \\ w_i &= b_i - \sum_{j=1}^{n} a_{ij} x_j \qquad i = 1, 2, \ldots, m. \end{aligned}$$

2.2 Basic and nonbasic variables

- The list of slack variables are added to the list of x variables

$$(x_1, \ldots, x_n, w_1, \ldots, w_m) = (x_1, \ldots, x_n, x_{n+1}, \ldots, x_{n+m}) \tag{4}$$

- With this notation the problem can be rewritten as

$$\begin{aligned} \zeta &= \sum_{j=1}^{n} c_j x_j \\ x_{n+i} &= b_i - \sum_{j=1}^{n} a_{ij} x_j \qquad i = 1, 2, \ldots, m. \end{aligned}$$

- This is the **starting dictionary**
- As the simplex method progresses it moves from one dictionary to another in search for an optimal solution
  - As dictionary has m basic variables and n nonbasic variables
  - Let $\mathcal{B}$ denote the collection of indices from $\{1, 2, \ldots, n + m\}$ corresponding to the basic variables
    - Initially $\mathcal{N} = \{n + 1, n + 2, \ldots, n + m\}$

- Let $\mathcal{N}$ denote the indices corresponding to the nonbasic variables
  - Initially $\mathcal{B} = \{1, 2, \ldots, n\}$
- In an iteration the current dictionary will look like this

$$\begin{aligned} \zeta &= \bar{\zeta} + \sum_{j \in \mathcal{N}} \bar{c}_j x_j \\ x_i &= \bar{b}_i - \sum_{j \in \mathcal{N}} \bar{a}_{ij} x_j \qquad i \in \mathcal{B}. \end{aligned}$$

2.3 Choosing the variables

- In each iteration of the simplex method exactly one variable goes from nonbasic to basic and exactly one goes from basic to nonbasic
  - The variable that goes from nonbasic to basic is called the **entering variable**
    - It is chosen with the aim of increasing ζ
    - One whose coefficient is positive
    - Pick k from $\{j \in \mathcal{N} \mid \bar{c}_j > 0\}$
      - If the set is empty the current solution is optimal
    - The k that is usually picked is the one with the largest coefficient
  - The variable that goes from basic to nonbasic is called the **leaving variable**
    - It is chosen to preserve non-negativity of the current basic variables
    - Once we have decided that x_k will be the entering variable its value will be increased from 0 to a positive value
      - The increase will change the values of the basic variables: $x_i = \bar{b}_i - \bar{a}_{ik} x_k \quad i \in \mathcal{B}$
    - To ensure that each of the variables remains nonnegative we must require that
      - $\bar{b}_i - \bar{a}_{ik} x_k \geq 0 \quad i \in \mathcal{B}$
    - Since the only ones that can go negative as x_k increases are those for which $\bar{a}_{ik}$ is positive
      - The value x_k at which the expression becomes zero is $x_k = \bar{b}_i / \bar{a}_{ik}$

      · x_k can only be raised to the smallest of these values $x_k = \min_{i \in \mathcal{B}: \hat{a}_{ik} > 0} \hat{b}_i / \hat{a}_{ik}$

    * The rule for selecting the leaving variable is pick l from $\{i \in \mathcal{B} \mid \hat{a}_{ik} > 0 \text{ and } \hat{b}_i / \hat{a}_{ik} \text{ is minimal } \}$

- Once the leaving-basic and entering basic variables have been selected the move from the current dictionary to a new dictionary involves appropriate row operations
  - This step is called a **pivot**
- Rules that make the choice of leaving variables unambiguous are called **pivot rules**

2.4 Initialization

- The solution associated with the initial dictionary is obtained by setting each x_j to zero and setting each w_i equal to the corresponding b_i
  - This solution is only feasible if and only if all the right-hand sides are nonnegative
- If the right hand side is not nonnegative and auxiliary problem is defined for which
  1. A feasible dictionary is easy to find
  2. The optimal dictionary provides a feasible dictionary for the original problem
- The auxiliary problem is

$$\begin{array}{lll} \text{maximize} & -x_0 & \\ \text{subject to} & \displaystyle\sum_{j=1}^{n} a_{ij}x_j - x_0 \leq b_i & i = 1, 2, \ldots, m \\ & x_j \geq 0 & j = 0, 1, \ldots, n. \end{array}$$

- It is easy to give a feasible solution to the auxilary problem
  - Simply set $x_j = 0$ for $j = 1, \ldots, n$ and then pick x_0 to be sufficiently large
  - Often referred to as **Phase I**

2.5 Unboundedness

- If none of the ratios are positive the problem is unbounded

3 Degeneracy

3.1 Definition

- A dictionary is **degenerate** if $\bar{b}_i$ vanishes for some $i \in \mathcal{B}$
  - A degenerate dictionary could cause difficulties for the simplex algorithm but it might not
  - Problems arise when a degenerate dictionary produces degenerate pivots
    - A pivot is degenerate is the calculation of the leaving variable is $+\infty$
    - If the numerate is positive and the denominator vanishes
  - If might happen that the simplex algorithm will make a sequence of degenerate pivots and eventually return to a dictionary that has appeared before
    - This is called **cycling**
    - It is typical for a pivot to "break away" from the degeneracy
  - Under certain pivoting rules cycling is possible this could e.g. be
    - Choose the entering variable as the one with the largest coefficient in the ς row of the dictionary
    - When two or more variables compete for leaving the basis, pick an x variable over a slack variable
      - If there is a choice use the variable with the smallest subscript
      - This means reading left to right, pick the first leaving candidate from the list: $x_1, x_2, \ldots, x_n, w_1, w_2, \ldots, w_m$
    - It is hard to find examples of cycling in which m and n are small
      - It has been shown that if a problem has an optimal solution but cycles off-optimum the problem must involve dictionary with at least four non-slack variables and two constraints

- **Theorem 3.1.** If the simplex method fails to terminate, then it must cycle
  - **Proof intuition:** there is a finite amount of dictionaries if it does not terminate it must cycle

3.2 The Perturbation/Lexicographic Method

- The simplex method is a whole family of related algorithms from which we can pick a specific instance by specifying what we have been referring to as pivoting rules
- There are pivoting rules for the simplex algorithm in which one either reach an optimal solution or prove that no such solution exists
- One of the methods is based on the observation that degeneracy is sort of an accident
  - A dictionary is degenerate if one or more of the $\bar{b}_i$ vanish
  - The right hand side could be any real number
    - The probability of the occurrence of any specific number is quite unlikely
    - Permute the problem by adding small random perturbation independently
      - The probability of exact cancellation is zero if they are chosen independently
  - Small positive numbers $\epsilon_1, \ldots, \epsilon_m$ are introduced for each constraint where the perturbation is getting much smaller on each succeeding constraint
    - It is written as \$0<ϵ_{m} « … « ϵ_2 « ϵ_1 « \$ all other data
    - The idea is that each ϵ_i acts on an entirely different scale from all the other ϵ_i's
      - No linear combination of the ϵ_i using coefficients that might arise will ever produce a number whose size is the same scale as the data in the problem
      - Instead of using specific values they are simply treat them as abstract symbols having these scale properties which is called the lexicographic method
- **Theorem** The simplex method always terminates provided that the leaving variable is selected by the lexicographic rule

3.3 Bland's Rule

- Bland's rule stipulates that both the entering and the leaving variable be elected from their respective sets of choices by choosing the variable x_k with the smallest index k
- **Theorem** The simplex method always terminates provided that both the entering and leaving variable are chosen according to Bland's rule

3.4 Fundamental Theorem of Linear Programming

- **Theorem** For an arbitrary linear program in standard form the following statements are true
  1. If there is no optimal solution, then the problem is either infeasible or unbounded
  2. If a feasible solution exists, then a basic feasible solution exists
  3. If an optimal solution exists then a basic optimal solution exists

4 Efficiency of the Simplex Method

4.1 Worst-Case Analysis

- For noncycling variants of the simplex method the simplex method moves from one basic feasible solution to another
  - This is without returning to a previously visited solution
  - The upper bound for the number iterations is simply the number of basic feasible solution which there can be at most $\binom{n+m}{m}$
  - The bound is maximized when $n = m$
    * It is bounded by 2^{2n}
    * This is very huge even when n is very small
- The following is a linear problem proposed in 1972 by V.Klee and G.J. Minty which requires $2^n - 1$ iterations to solve

$$\begin{array}{lll} \text{maximize} & \sum_{j=1}^{n} 10^{n-j} x_j & \\ \text{subject to} & 2\sum_{j=1}^{i-1} 10^{i-j} x_j + x_i \leq 100^{i-1} & i = 1, 2, \ldots, n \\ & x_j \geq 0 & j = 1, 2, \ldots, n. \end{array}$$

- The simplex method with the largest coefficient rule will start at one of these vertices and visit every vertex before finding the optimal solution
  - The idea is that the RHS of the conditions have the following conditions

$$1 = b_1 << b2 << \cdots << b_n \tag{5}$$

- No one has found a rule that is better than the largest coefficient rule for Simplex

4.2 Empirical Average Performance of the Simplex Method

- The best measure is $min(n, m)$

5 Duality Theory

5.1 The Dual Problem

- Given a linear programming problem in standard form, which is called the **primal problem**

$$\begin{array}{lll} \text{maximize} & \sum_{j=1}^{n} c_j x_j & \\ \text{subject to} & \sum_{j=1}^{n} a_{ij} x_j \leq b_i & i = 1, 2, \ldots, m \\ & x_j \geq 0 & j = 1, 2, \ldots, n, \end{array}$$

- the associated **dual linear program** is given by

$$\begin{array}{lll} \text{minimize} & \sum_{i=1}^{m} b_i y_i & \\ \text{subject to} & \sum_{i=1}^{m} y_i a_{ij} \geq c_j & j = 1, 2, \ldots, n \\ & y_i \geq 0 & i = 1, 2, \ldots, m. \end{array}$$

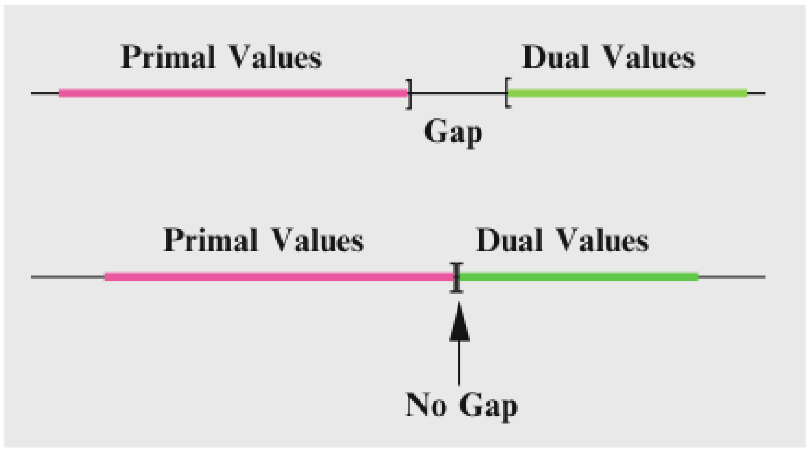
- Taking the dual of the dual returns us to the primal
- The dual problem provides upper bounds for the primal objective function value

5.2 The Weak Duality Theorem

- **Theorem** If $(x_1, x_2, \ldots, x_n)$ is feasible for the primal and $(y_1, y_2, \ldots, y_n)$ is feasible for the dual then

$$\sum_j c_j x_j \leq \sum_i b_i y_i \tag{6}$$

5.3 The Strong Duality Theorem



- **Theorem** If the primal problem has an optimal solution

$$x^* = (x_1^*, x_2^*, \ldots, x_n^*) \tag{7}$$

- then the dual has an optimal solution

$$y^* = (y_1^*, y_2^*, \ldots, y_n^*) \tag{8}$$

such that

$$\sum_j c_j x_j^* = \sum_i b_i y_i^* \tag{9}$$

- If the primal problem is unbounded the dual problem is infeasible
  - If the dual problem is unbounded then the primal problem must be infeasible
- Duality theory provides a certificate of optimality
  - One can check that the two given solutions for the dual and primal problem is feasible
  - One can check that the two solutions are equal
- Sometimes it is easy to apply the simplex method to the dual

5.4 Complementary Slackness

- **Theorem** Suppose that $x = (x_1, x_2, \ldots, x_n)$ is primal feasible and $y = (y_1, y_2, \ldots, y_m)$ is dual feasible. Let $(w_1, w_2, \ldots, \ldots, w_m)$ denote the corresponding primal slack variable and let $(z_1, z_2, \ldots, z_n)$ denote the corresponding dual slack variable. Then x and y are optimal for the respective problems if and only if

$$x_j z_h = 0, \quad \text{for } j = 1, 2, \ldots, n \tag{10}$$

$$w_i y_i = 0, \quad \text{for } i = 1, 2, \ldots, n \tag{11}$$

- This can be used to find the solution to the corresponding dual problem given an optimal feasible solution to the primal problem

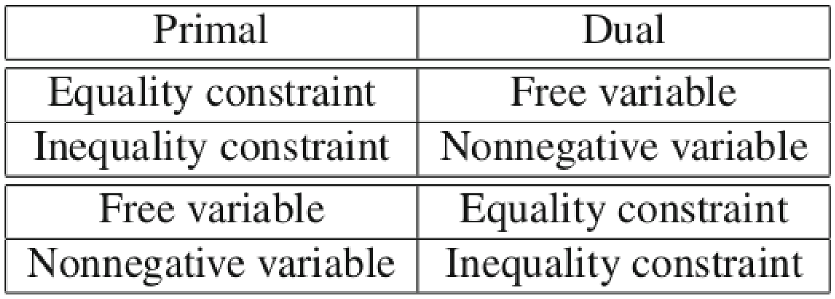
5.5 The Dual Simplex Method

- By using the simplex algorithm on the dual problem while keeping track of corresponding primal, this can be used to find feasible dictionary for the primal problem
  - This is done if the starting dictionary for the dual problem is feasible but the one for the primal is not

5.6 A Dual-Based Phase I Algorithm

- If both the starting dictionary for the dual and primal problem are infeasible one can change the primal problem to make the corresponding dual problem feasible
  - This can finding a feasible starting dictionary for the primal problem

5.7 The Dual of a Problem in General Form



Primal	Dual
Equality constraint	Free variable
Inequality constraint	Nonnegative variable
Free variable	Equality constraint
Nonnegative variable	Inequality constraint

TABLE 5.1. Rules for forming the dual.

- Free variables are unconstrained variables
- Given the following linear programming problem

$$\begin{array}{lll} \text{maximize} & \sum_{j=1}^{n} c_j x_j & \\ \text{subject to} & \sum_{j=1}^{n} a_{ij} x_j = b_i & i = 1, 2, \ldots, m \\ & x_j \geq 0 & j = 1, 2, \ldots, n. \end{array}$$

- The corresponding dual is

$$\begin{array}{ll} \text{minimize} & \sum_{i=1}^{m} b_i y_i \\ \text{subject to} & \sum_{i=1}^{m} y_i a_{ij} \geq c_j \qquad j = 1, 2, \ldots, n. \end{array}$$

6 Convex Analysis

6.1 Farkas' Lemma

- The system $Ax \leq b$ has no solutions if and only if there is a y such that

$$\begin{aligned} A^T y &= 0 \\ y &\geq 0 \\ b^T y &< 0 \end{aligned}$$

6.2 Strict Complementarity

- **Theorem** If both the primal and the dual have feasible solutions, then there exists a primal feasible solution $(\bar{x}, \bar{w})$ and a feasible solution $(\bar{y}, \bar{z})$ such that $\bar{x} + \bar{z} > 0$ and $(\bar{y} + \bar{w}) > 0$
- A variable x_j that must vanish in order for a linear programming problem to be feasible is called a **null variable**
- **Strict Complementarity Slackness Theorem:** If a linear programming problem has an optimal solution, then there is an optimal solution (x^*, w^*) and an optimal dual solution (y^*, z^*) such that $x^* + z^* > 0$ and $y^* + w^* > 0$

7 Game Theory

7.1 Matrix Games

- A **matrix game** is a two person game which is defined as follows
  - First each person selects an action from a finite set of choices

    * Done independently of the other
    * They will in general be confronted with different set of actions
  - Then Both reveal to each other their choice
    * i denote the first player's choice
    * j denote the second player's choice
    * The rules of the game stipulate that the first player will pay the second player a_{ij} dollars
  - The array of possible payments $A = [a_{ij}]$ is presumed to be known to both players before the game begins
    * If the payment is negative for some (i, j) the payment goes in the other direction
  - The first player is refereed to as the **row player** and the second as **column player**
  - Since the is a finite number of actions for each player i is a number selected from 1 to m and j is selected from 1 to n
  - Rock paper scissors can be described as a matrix game in the following way

$$\begin{bmatrix} 0 & 1 & -1 \\ -1 & 0 & 1 \\ 1 & -1 & 0 \end{bmatrix} \tag{12}$$

- **Randomized strategy** means that each play of the game appears from the other players point of view that the player is making the choices random according to some probability distribution
  - Let y_i denote the probability that the row player selects action i
    * The vector y composed of these probabilities is called a **stochastic vector**
  - Mathematically a vector is a stochastic vector if it has nonnegative components that sum to one
    * i.e. $y \geq 0$ and $e^T y = 1$ where e is a vector consisting of all ones
  - The column players must also adopt a randomized strategy
  - Let x_j denote the probability that the column player selects action j

    - Let x denote the stochastic vector composed of these probabilities
  - The expected payoff of the column player is computed by summing over all possible outcomes
    - The payoff associated becomes the outcome times the probability of that outcome
  - The set of the possible outcomes is simply the set of pairs (i, j)
    - i ranges over the row indices $(1, 2, \ldots, m)$
    - j ranges over the column indices $(1, 2, \ldots, n)$
  - The expected payoff to the column players is

$$\sum_{i,j} y_i a_{ij} x_j = y^T A x$$

7.2 Optimal Strategies

- If the column player adopts strategy x the row player's then the row player's best defense is to use the strategy y^* that achieves the following minimum

$$\begin{array}{ll} \text{minimize} & y^T A x \\ \text{subject to} & e^T y = 1 \\ & \quad y \geq 0. \end{array}$$

- Since for any given x the row player will adopt the strategy that achieves the minimum in the previous problem the column player should employ a strategy x^* that attains the following maximum

$$\max_x \ \min_y \ y^T A x$$

- where max and the min are over all stochastic vectors
  - This problem can be reformulated as a linear programming problem
  - The inner optimization can be taken over just the deterministic strategies

$$\min_y y^T A x = \min_i e_i^T A x \tag{13}$$

- The max-min problem can be rewritten as

$$\begin{array}{ll}\text{maximize} & \left(\min_i e_i^T Ax\right) \\ \text{subject to} & \displaystyle\sum_{j=1}^{n} x_j = 1 \\ & \quad x_j \geq 0 \qquad j = 1, 2, \ldots, n.\end{array}$$

- If one introduce a new variable v representing a lower bound on the $e_i^T Ax$ the problem can be recast as a linear program

$$\begin{array}{lll}\text{maximize} & v & \\ \text{subject to} & v \leq e_i^T Ax & i = 1, 2, \ldots, m \\ & \displaystyle\sum_{j=1}^{n} x_j = 1 & \\ & x_j \geq 0 & j = 1, 2, \ldots, n.\end{array}$$

- In vector notation it is written as

$$\begin{array}{lr}\text{maximize} & v \\ \text{subject to} & ve - Ax \leq 0 \\ & e^T x = 1 \\ & x \geq 0.\end{array}$$

- In block-matrix form one gets

$$\begin{array}{ll}\text{maximize} & \begin{bmatrix} 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ v \end{bmatrix} \\ \text{subject to} & \begin{bmatrix} -A & e \\ e^T & 0 \end{bmatrix} \begin{bmatrix} x \\ v \end{bmatrix} \begin{matrix} \leq \\ = \end{matrix} \begin{bmatrix} 0 \\ 1 \end{bmatrix} \\ & \qquad\qquad x \geq 0 \\ & \qquad\qquad v \ \text{free.}\end{array}$$

- By symmetry the row player seeks a strategy y^* that attains optimality in the following min-max problem:

$$\min_y \max_x y^T Ax$$

- which can be reformulated as the linear program

$$\begin{array}{ll} \text{minimize} & u \\ \text{subject to} & ue - A^T y \geq 0 \\ & e^T y = 1 \\ & y \geq 0. \end{array}$$

- Written in block-matrix form we get

$$\begin{array}{ll} \text{minimize} & \begin{bmatrix} 0 & 1 \end{bmatrix} \begin{bmatrix} y \\ u \end{bmatrix} \\ \text{subject to} & \begin{bmatrix} -A^T & e \\ e^T & 0 \end{bmatrix} \begin{bmatrix} y \\ u \end{bmatrix} \begin{matrix} \geq \\ = \end{matrix} \begin{bmatrix} 0 \\ 1 \end{bmatrix} \\ & y \geq 0 \\ & u \text{ free.} \end{array}$$

7.3 The Minimax Theorem

- **Minimax Theorem** There exist stochastic vectors x^* and y^* for which

$$\max_x {y^*}^T Ax = \min_y y^T Ax^* \tag{14}$$

- Proof using the strong duality theorem
- The common optimal value $v^* = u^*$ is called the **value** of the game
  - A game whose value is zero is a fair game
  - Games where the two players are interchangeable are fair
    - They are called **symmetric**
    - They are characterized by payoff matrices having the property that $a_{ij} = -a_{ji}$ for all i and j

8 The Simplex Method in Matrix Notation

8.1 Matrix Notation

- A standard-form linear programming problem

$$\begin{array}{lll} \text{maximize} & \sum_{j=1}^{n} c_j x_j & \\ \text{subject to} & \sum_{j=1}^{n} a_{ij} x_j \le b_i & i = 1, 2, \ldots, m \\ & x_j \ge 0 & j = 1, 2, \ldots, n. \end{array}$$

- It the be written in matrix for the following way with the slack variables denoted as x_{n+i} instead of w_i

$$\begin{array}{ll} \text{maximize} & c^T x \\ \text{subject to} & Ax = b \\ & x \ge 0, \end{array}$$

- where

$$A = \begin{bmatrix} a_{11} & a_{12} & \ldots & a_{1n} & 1 & & & \\ a_{21} & a_{22} & \ldots & a_{2n} & & 1 & & \\ \vdots & \vdots & & \vdots & & & \ddots & \\ a_{m1} & a_{m2} & \ldots & a_{mn} & & & & 1 \end{bmatrix},$$

- and

$$b = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_m \end{bmatrix}, \qquad c = \begin{bmatrix} c_1 \\ c_2 \\ \vdots \\ c_n \\ 0 \\ \vdots \\ 0 \end{bmatrix}, \quad \text{and} \quad x = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \\ x_{n+1} \\ \vdots \\ x_{n+m} \end{bmatrix}.$$

- In component notation the \$i\$th component of Ax can be broken up into a basic part and a nonbasic part

$$\sum_{j=1}^{n+m} a_{ij}x_j = \sum_{j\in\mathcal{B}} x_j + \sum_{j\in\mathcal{N}} x_j \tag{15}$$

- We let B denote an $m \times m$ matrix whose columns consists precisely of the m columns of A that are associated with the basic variables
  - N denote an $m \times n$ matrix whose columns are the n nonbasic columns of A
  - A is written in an partitioned matrix form as follows $A = [B \quad N]$
    - The matrix on the right does not equal the A matrix
    - It is the A matrix with its columns rearranged in such a manner that all the columns associated with basic variables are listed first followed by the nonbasic
  - Let rearrange the rows of x and write

$$x = \begin{bmatrix} x_{\mathcal{B}} \\ x_{\mathcal{N}} \end{bmatrix} \tag{16}$$

- Then the following separation of Ax into a sum of two terms is true and captures the separation of variables

$$Ax = [B \quad N] \begin{bmatrix} x_{\mathcal{B}} \\ x_{\mathcal{N}} \end{bmatrix} = Bx_{\mathcal{B}} + Nx_{\mathcal{N}} \tag{17}$$

- By similarly partitioning c one can write

$$c^T x = \begin{bmatrix} c_{\mathcal{B}} \\ c_{\mathcal{N}} \end{bmatrix}^T \begin{bmatrix} x_{\mathcal{B}} \\ x_{\mathcal{N}} \end{bmatrix} = c_{\mathcal{B}}^T x_{\mathcal{B}} + c_{\mathcal{N}}^T x_{\mathcal{N}}.$$

8.2 The Primal Simplex Method

- A dictionary has the property that the basic variables are written as functions of the nonbasic variables
  - The constrain equations $Ax = b$ can be written as $Bx_{\mathcal{B}}+Nx_{\mathcal{N}} = b$
  - The fact that basic variables $x_{\mathcal{B}}$ can be written as a function of the nonbasic variables is equivalent to the fact that the matrix B is invertible and therefore

$$x_{\mathcal{B}} = B^{-1}b - B^{-1}Nx_{\mathcal{N}}.$$

- The objective function can be written as

$$\begin{aligned}\zeta &= c_{\mathcal{B}}^T x_{\mathcal{B}} + c_{\mathcal{N}}^T x_{\mathcal{N}} \\ &= c_{\mathcal{B}}^T \left(B^{-1}b - B^{-1}Nx_{\mathcal{N}}\right) + c_{\mathcal{N}}^T x_{\mathcal{N}} \\ &= c_{\mathcal{B}}^T B^{-1}b - \left((B^{-1}N)^T c_{\mathcal{B}} - c_{\mathcal{N}}\right)^T x_{\mathcal{N}}.\end{aligned}$$

- The dictionary associated with basic $\mathcal{B}$ can be written as

$$\begin{aligned}\zeta &= c_{\mathcal{B}}^T B^{-1}b - \left((B^{-1}N)^T c_{\mathcal{B}} - c_{\mathcal{N}}\right)^T x_{\mathcal{N}} \\ x_{\mathcal{B}} &= \quad B^{-1}b - B^{-1}Nx_{\mathcal{N}}.\end{aligned}$$

- Based on the component-form notation the following identifications can be made

$$\begin{aligned}c_{\mathcal{B}}^T B^{-1}b &= \bar{\zeta} \\ c_{\mathcal{N}} - (B^{-1}N)^T c_{\mathcal{B}} &= [\bar{c}_j] \\ B^{-1}b &= \left[\bar{b}_i\right] \\ B^{-1}N &= [\bar{a}_{ij}],\end{aligned}$$

- where the bracketed expressions on the right denote vectors and matrices with the index i running over $\mathcal{B}$ and the index j running over $\mathcal{N}$

- The basic solution associated with the dictionary is obtained by settings $x_{\mathcal{N}}$ equal to zero

o

$$\begin{aligned}x_{\mathcal{N}}^* &= 0, \\ x_{\mathcal{B}}^* &= B^{-1}b.\end{aligned}$$

- For the dual slack variables we need to relabel the dual variables and append them to the dual slacks such that y_i becomes z_{n+i}
  - Using the relabeling of the dual variables the dual dictionary corresponding to a given primal dictionary is

$$\begin{aligned} -\xi &= \quad -c_{\mathcal{B}}^T B^{-1} b - (B^{-1}b)^T z_{\mathcal{B}} \\ z_{\mathcal{N}} &= (B^{-1}N)^T c_{\mathcal{B}} - c_{\mathcal{N}} + (B^{-1}N)^T z_{\mathcal{B}}. \end{aligned}$$

- The dual solution associated with this dictionary is obtained by setting $z_{\mathcal{B}}$ equal to zero

$$\begin{aligned} z_{\mathcal{B}}^* &= 0, \\ z_{\mathcal{N}}^* &= (B^{-1}N)^T c_{\mathcal{B}} - c_{\mathcal{N}}. \end{aligned}$$

- The following shorthand can be introduced

$$\zeta^* = c_{\mathcal{B}}^T B^{-1} b,$$

- The primal dictionary can be written succinctly as

$$\begin{aligned} \zeta &= \zeta^* - {z_{\mathcal{N}}^*}^T x_{\mathcal{N}} \\ x_{\mathcal{B}} &= x_{\mathcal{B}}^* - B^{-1} N x_{\mathcal{N}}. \end{aligned}$$

- The associated dual dictionary has a very symmetric appearance

$$\begin{aligned} -\xi &= -\zeta^* - (x_{\mathcal{B}}^*)^T z_{\mathcal{B}} \\ z_{\mathcal{N}} &= \quad z_{\mathcal{N}}^* + (B^{-1}N)^T z_{\mathcal{B}}. \end{aligned}$$

- The primal simplex method can be described as follows.
  - The starting assumptions are that we are given
    1. A partition of the $n + m$ indices into a collection $\mathcal{B}$ of m basic indices and a collection $\mathcal{N}$ of n nonbasic ones with the property that the basic matrix B is invertible

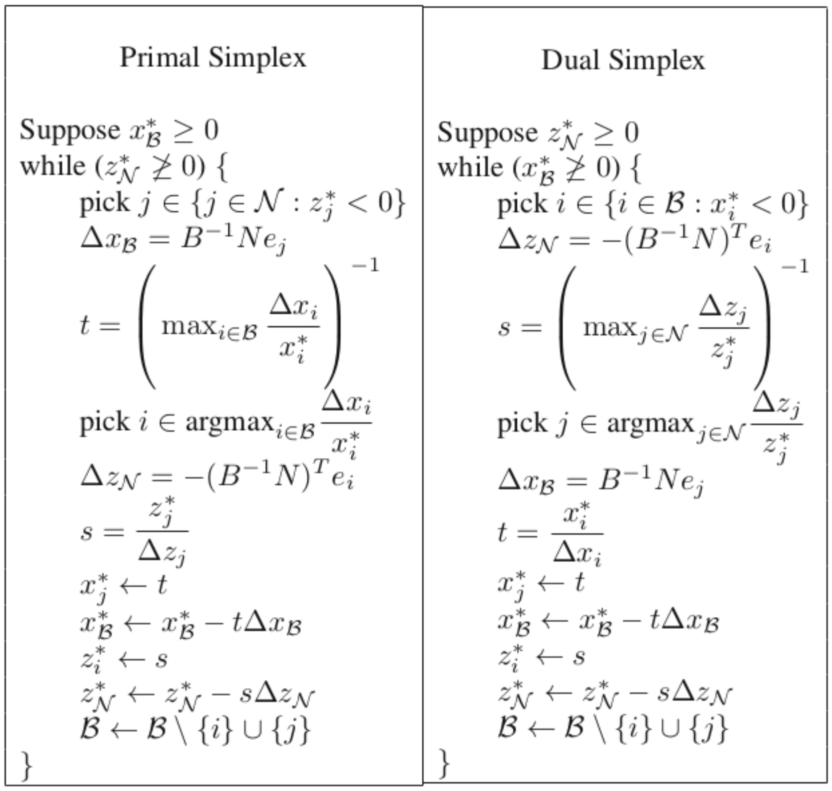
2. An associated current primal solution $x^*_{\mathcal{B}} \geq 0$ (and $x^*_{\mathcal{N}} = 0$)
3. An Associated current dual solution $z^*_{\mathcal{N}}$ (with $z_{\mathcal{B}} = 0$)

- The simplex algorithm then produces a sequence of steps to "adjacent" bases such that the current value ς^* of the objective function ς increases at each step updating $x^*_{\mathcal{B}}$ and $x^*_{\mathcal{N}}$ along the way
  * If the step size is positive
- Two bases are said to be adjacent to each other if they differ in only one index
  * i.e. given a basic $\mathcal{B}$ an adjacent basic is determined by removing one basic index and replacing it with a nonbasic index
  * The index that gets removed corresponds to the leaving variable
- One step of the simplex method is called an iteration
- An iteration runs as follows
  1. **Check for optimality**
     * If $z^*_{\mathcal{N}} \geq 0$ stop. The current situation is optimal
     * All that is required for optimality is dual feasibility and that is the case if an only if $z^*_{\mathcal{N}} \geq 0$
  2. **Select Entering Variable**
     * Pick an index $j \in \mathcal{N}$ for which $z^*_j < 0$
     * Variable x_j is the entering variable
  3. **Compute Primal Step Direction $\Delta x_{\mathcal{B}}$**
     * Having selected the entering variable we want to increase its value from zero and we let $x_{\mathcal{N}} = te_j$
     * Then we have that from the primal dictionary that $x_{\mathcal{B}} = x^*_{\mathcal{B}} - B^{-1}Nte_j$
     * The step direction $\Delta x_{\mathcal{B}}$ is given by $\Delta x_{\mathcal{B}} = B^{-1}Ne_j$
  4. **Compute Primal Step Length**
     * We want to pick the largest $t \geq 0$ for which every component of $x_{\mathcal{B}}$ remains nonnegative
       · i.e. we want to pick the largest t for which $x^*_{\mathcal{B}} \geq t\Delta x_{\mathcal{B}}$
     * The largest t for which all of the inequalities hold is given by $t = \left(\max_{i\in\mathcal{B}} \frac{\Delta x_i}{x^*_i}\right)^{-1}$
       · The correct convention for $0/0$ is to set such ratios equal to zero

  - · If the maximum is less than or equal to zero we can stop here the primal is unbounded

5. **Select Leaving Variable**
   - ∗ The leaving variable is chosen as any variable $x_i, i \in \mathcal{N}$ for which the maximum in the calculation of t is obtained
6. **Compute Dual Step Direction $\Delta z_{\mathcal{N}}$**
   - ∗ Since in that dictionary z_i is the entering variable we see that
     - · $\Delta z_{\mathcal{N}} = -(B^{-1}N)^T e_i$
7. **Compute Dual Step Length**
   - ∗ Since we known that z_j is the leaving variable in the dual dictionary the step length for the dual variables is
     - · $s = \frac{z_j^*}{\Delta z_j}$
8. **Update Current Primal and Dual solutions**
   - ∗ Now everything has been obtained to update the data in the dictionary
     - · $x_j^* \leftarrow t$
     - · $x_{\mathcal{B}}^* \leftarrow x_{\mathcal{B}}^* - t\Delta x_{\mathcal{B}}$
   - ∗ and
     - · $z_i^* \leftarrow s$
     - · $z_{\mathcal{N}}^* \leftarrow z_{\mathcal{N}}^* - s\Delta z_{\mathcal{N}}$
9. **Update Basis**
   - ∗ The basis is updated $\mathcal{B} \leftarrow B \backslash \{i\} \cup \{j\}$

8.3 The Dual Simplex Method



Primal Simplex

```
Suppose $x^*_{\mathcal{B}} \geq 0$
while ($z^*_{\mathcal{N}} \not\geq 0$) {
    pick $j \in \{j \in \mathcal{N} : z^*_j < 0\}$
    $\Delta x_{\mathcal{B}} = B^{-1} N e_j$
    $t = \left( \max_{i \in \mathcal{B}} \frac{\Delta x_i}{x^*_i} \right)^{-1}$
    pick $i \in \operatorname{argmax}_{i \in \mathcal{B}} \frac{\Delta x_i}{x^*_i}$
    $\Delta z_{\mathcal{N}} = -(B^{-1} N)^T e_i$
    $s = \frac{z^*_j}{\Delta z_j}$
    $x^*_j \leftarrow t$
    $x^*_{\mathcal{B}} \leftarrow x^*_{\mathcal{B}} - t \Delta x_{\mathcal{B}}$
    $z^*_i \leftarrow s$
    $z^*_{\mathcal{N}} \leftarrow z^*_{\mathcal{N}} - s \Delta z_{\mathcal{N}}$
    $\mathcal{B} \leftarrow \mathcal{B} \setminus \{i\} \cup \{j\}$
}
```

Dual Simplex

```
Suppose $z^*_{\mathcal{N}} \geq 0$
while ($x^*_{\mathcal{B}} \not\geq 0$) {
    pick $i \in \{i \in \mathcal{B} : x^*_i < 0\}$
    $\Delta z_{\mathcal{N}} = -(B^{-1} N)^T e_i$
    $s = \left( \max_{j \in \mathcal{N}} \frac{\Delta z_j}{z^*_j} \right)^{-1}$
    pick $j \in \operatorname{argmax}_{j \in \mathcal{N}} \frac{\Delta z_j}{z^*_j}$
    $\Delta x_{\mathcal{B}} = B^{-1} N e_j$
    $t = \frac{x^*_i}{\Delta x_i}$
    $x^*_j \leftarrow t$
    $x^*_{\mathcal{B}} \leftarrow x^*_{\mathcal{B}} - t \Delta x_{\mathcal{B}}$
    $z^*_i \leftarrow s$
    $z^*_{\mathcal{N}} \leftarrow z^*_{\mathcal{N}} - s \Delta z_{\mathcal{N}}$
    $\mathcal{B} \leftarrow \mathcal{B} \setminus \{i\} \cup \{j\}$
}
```

- Instead of assuming that the primal dictionary is feasible one can use that the dual dictionary is feasible to perform the analogous steps

9 Implementation Issues

9.1 Introduction

- The most time-consuming steps in the simplex method are the computations

$$\Delta x_{\mathcal{B}} = B^{-1} N e_j \qquad \text{and} \qquad \Delta z_{\mathcal{N}} = -(B^{-1} N)^T e_i,$$

- We dont compute the inverse of the basis matrix instead we calculate $\Delta x_{\mathcal{B}}$ by solving the following system of equations

$$B\Delta x_{\mathcal{B}} = a_j \tag{18}$$

where

$$a_j = Ne_j \tag{19}$$

is the column of N associated with nonbasic variable x_j

- The calculation of $\Delta z_{\mathcal{N}}$ is also broken into two steps

$$B^T v = e_i$$
$$\Delta z_{\mathcal{N}} = -N^T v$$

- Solving the two systems of equation is where most of the complexity of the simplex iteration ies

9.2 Solving Systems of Equations: LU Factorization

- The systems of equation will be on the form

$$Bx = b$$

where B is an invertible $m \times m$ matrix and b is an arbitrary m vector

- If when doing Gaussian Elimination on a matrix B one saves the row before doing the operation then one obtains a matrix D
  - One takes the matrix and splits it into three matrices
    - D_1 containing all elements on or below the diagonal
    - D_2 One containing only the diagonal elements
    - D_3 containing all elements on or above the diagonal
  - Then the following equation holds $B = D_1 D_2^{-1} D_3$
  - The following is denoted L

$$L = D_1 D_2^{-1} \tag{20}$$

- The following is denoted U

$$U = D_3 \tag{21}$$

- The resulting representation $B = LU$ is called an LU-factorization of B

  - Finding an LU factorization is equivalent to Gaussian elimination since multiplying B on the left by L^{-1} has the effect of applying row operations to B to put it into upper-triangular form U

- The value of an LU factorization can be used to solve systems of equations

- If one wanted to solve $B\Delta x_{\mathcal{B}} = a_j$

  - One first substitute LU for B so the system becomes $LU\Delta x_{\beta} = a_j$
  - If one lets $y = U\Delta x_{\mathcal{B}}$ one can solve $Ly = b$ for y
    * Since L is lower triangular solving this equation is easy
      · Successively solving for the elements of the vector y starting with the first and proceeding to the last is called **forward substitution**
      · This is an easy way to solve this
  - Once y is known one can solve $U\Delta x_{\mathcal{B}} = y$ for $\Delta x_{\mathcal{B}}$
    * Since U is upper triangular solving this is easy as well
    * Successively solving for the elements of the vector $\Delta x_{\mathcal{B}}$ starting with the last and back to the last is called **backward substitution**
      · This is an easy way to solve this

9.3 Exploiting Sparsity

- A matrix that contains zeroes is called a **sparse matrix**

- When a sparse matrix has lots of zeros two things happen

  1. The changes of being required to make row and/or column permutations is high
  2. Additional computation efficiency can be obtained by making further row and/or column permutation with the aim of keeping L and/or U as sparse as possible

- The problem with finding the "best" permutation is, in itself, harder than the linear programming problem

- There are simple heuristics that help preserver sparsity in L and U

  - One such heuristic called the **minimum-degree** ordering heuristic is as follows

Before eliminating the nonzeros below a diagonal "pivot" element, scan all uneliminated rows and select the sparsest row, i.e., that row having the fewest nonzeros in its uneliminated part (ties can be broken arbitrarily). Swap this row with the pivot row. Then scan the uneliminated nonzeros in this row and select that one whose column has the fewest nonzeros in its uneliminated part. Swap this column with the pivot column so that this nonzero becomes the pivot element. (Of course, provisions should be made to reject such a pivot element if its value is close to zero.)

- The number of non zeros in the uneliminated part of a row/column is called the **degree** of the row/column
- The fact that the rows and columns to get this factorization has been permuted has only a small impact on how one uses the factorization to solve the systems of equations
  - The first step is to permute the rows of the a_j so the fit the given permutation
  - Then one just does the same thing as before and in the end rewrite the solution to get the listing of elements in the original order
  - Since LU factorization is a m^3 algorithm one should perform as few LU-factorizations as possible

10 Integer Programming

10.1 Introduction

- Many real-world problems could be modeled as linear programs except some or all the variables are constrained to be integers
  - They are called **integer programming problems**
  - It is harder than linear programming

10.2 Scheduling Problems

- There are many problems that can be classified as scheduling problems

  - There are e.g. two related problems of this type the equipment scheduling and crew scheduling problem by large airlines

- The equipment scheduling problem
  - Airlines determine how to route their planes as follows
    1. A number of specific flight legs are defined based on market demand
       * A leg is by definition one flight taking off from somewhere at some time and landing somewhere else
       * They are defined by market demand
       * One wants to put this legs together in such a way that the available aircraft can cover all of them
         · For each airplane the airline must put together a route that it will fly
       * A route is a sequence of flight legs for which the destination of one leg is the origin or the next
         · The final destination must be the origin of the first leg
    2. Given potential routes one wants select a subset of them with the property that each leg is covered by exactly one route
       * If there are enough potential routes there might be multiple feasible solutions
       * The goal is to find the optimal one
       * To formulate the problem as an integer program let

$$x_j = \begin{cases} 1 & \text{if route } j \text{ is selected,} \\ 0 & \text{otherwise,} \end{cases}$$

$$a_{ij} = \begin{cases} 1 & \text{if leg } i \text{ is part of route } j, \\ 0 & \text{otherwise,} \end{cases}$$

- and

$$c_j = \text{cost of using route } j.$$

- Then the problem is

$$\begin{aligned}\text{minimize} \quad & \sum_{j=1}^{n} c_j x_j \\ \text{subject to} \quad & \sum_{j=1}^{n} a_{ij} x_j = 1 && i = 1, 2, \ldots, m, \\ & x_j \in \{0, 1\} && j = 1, 2, \ldots, n.\end{aligned}$$

- This model is often called a **set-partitioning problem**
- The **crew scheduling problem**
  - Flight crews do not necessarily follow the same aircraft around a route
  - The constraints differ from those for the aircraft
  - Flight crews might ride as passengers on some legs
  - The model is

$$\begin{aligned}\text{minimize} \quad & \sum_{j=1}^{n} c_j x_j \\ \text{subject to} \quad & \sum_{j=1}^{n} a_{ij} x_j \geq 1 && i = 1, 2, \ldots, m, \\ & x_j \in \{0, 1\} && j = 1, 2, \ldots, n.\end{aligned}$$

- Is often referred to as a **set-covering problem**

10.3 The Traveling Salesman Problem

- Consider a salesman who needs to visit each of n cities
  - They are enumerated as $0, 1, \ldots, n-1$
  - The goal is to start from his home city 0 and make a tour visiting each of the remaining cities once and only once and then returning to his home
  - The distance between each pair of cities c_{ij} is known
    - It could also be travel time or cost of travel
  - He wants to make the tour that minimizes the total distances

- The tour is determined by listing the cities in the order in which they will be visited
  - If one let s_i denote the ith city visited the
  - The tour can be described as $s_0 = 0, s_1, s_2, \ldots, s_{n-1}$
- One can permute the cities in $(n-1)!$ possible ways and therefore the problem cannot be solved by enumeration
- For each (i, j) a decision variable x_{ij} is introduced that will be equal to one if the tour vists city j immediately after visiting city i
  - Otherwise it will be equal to 0
  - The objective function can be written using these variables as minimize $\sum_i \sum_j c_{ij} x_{ij}$
- If the salesman visits city i he must go to one and only one city next
  - These constraints can be written as $\sum_j x_{ij} = 1 \quad i = 0, 1, \ldots, n-1$
  - They are called **go-to constraints**
- When the salesman visits a city he must have come from one and only one prior city i.e.
  - $\sum_i x_{ij} = 1 \quad j = 0, 1, \ldots, n-1$
  - They are called the **come-from constraints**
- Let t_i for $i = 0, 1, \ldots, n$ be defined as the number of the stop along the tour for which city i is visited
  - From this we see that $t_{s_i} = i \quad i = 0, 1, \ldots, n-1$
  - For a bonafide tour $t_j = t_i + 1$ if $x_{ij} = 1$
  - Each t_i is an integer between 0 and $n-1$ inclusive
  - t_j satisfies the following constraints

$$t_j \geq \begin{cases} t_i + 1 - n & \text{if } x_{ij} = 0, \\ t_i + 1 & \text{if } x_{ij} = 1. \end{cases}$$

- The constraints can be written succinctly as

$$t_j \geq t_i + 1 - n(1 - x_{ij}), \qquad i \geq 0, j \geq 1, i \neq j.$$

- The constraints forces a solution to be a bonafide tour
- The traveling salesman problem can be formulated as the following integer programming problem

$$
\begin{array}{lll}
\text{minimize} & \sum_{i,j} c_{ij} x_{ij} & \\
\text{subject to} & \sum_{j=1}^{n} x_{ij} = 1, & i = 0, 1, \ldots, n-1, \\
& \sum_{i=1}^{n} x_{ij} = 1, & j = 0, 1, \ldots, n-1, \\
& t_j \geq t_i + 1 - n(1 - x_{ij}), & i \geq 0, j \geq 1, i \neq j, \\
& t_0 = 0, & \\
& x_{ij} \in \{0, 1\}, & \\
& t_i \in \{0, 1, 2, \ldots\}. &
\end{array}
$$

10.4 Fixed Costs

- It is sometimes more realistic to assume that there is a fixed cost for engaging in the activity plus a linear variable cost
  - Such a term might have the form

$$c(x) = \begin{cases} 0 & \text{if } x = 0 \\ K + cx & \text{if } x > 0 \end{cases}$$

- If there is an upper bound on the size of x such a function can be equivalently modeled using strictly linear functions at the expense of introducing one integer-valued variable
  - Suppose u is an upper bound on the x variable
  - Let y denote a $\{0, 1\}$ valued variable which is one when and only when $x > 0$
  - Then $c(x) = Ky + cx$
  - The condition that y is one exactly when $x > 0$ can be guaranteed by introducing the following constraints

$$
\begin{aligned}
x &\leq uy \\
x &\geq 0 \\
y &\in \{0, 1\}
\end{aligned}
$$

10.5 Nonlinear Objective Functions

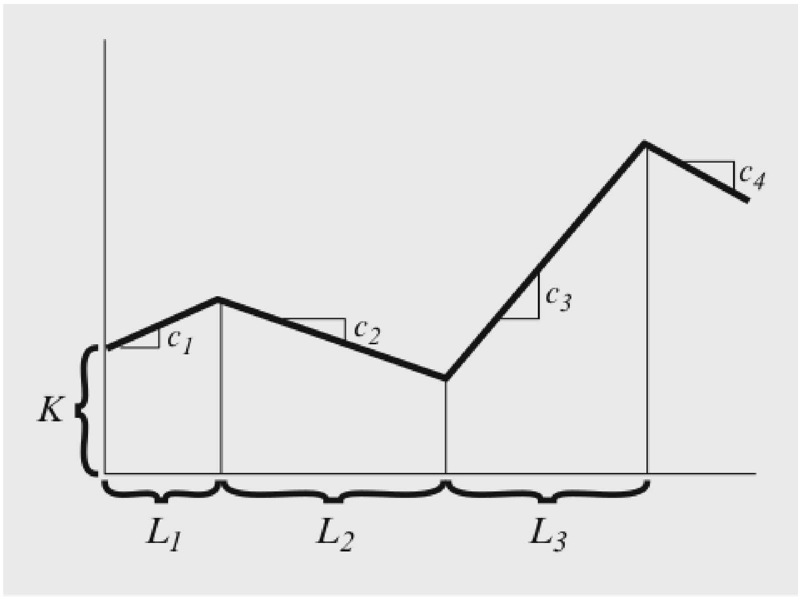


FIGURE 23.4. A piecewise linear function.

- Sometimes the terms in the objective function are not linear at all
  - Formulating an integer programming approximation to a general nonlinear term in the objective function is done in the following way
    1. Approximate the nonlinear function by a continuous piecewise linear function
    2. Introduce integer variables that allow us to represents the piecewise linear function using linear relations
       - ∗ We decompose the variable x into a sum: $x = x_1 + x_2 + \cdots + x_k$
       - ∗ x_i denotes how much of the interval $[0, x]$ is contained in the ith linear segment of the piecewise linear function
       - ∗ Constraints are needed to guarantee
         - · That the initial x_i's are equal to the length of their respective segments

  - That after the straddling segment the subsequent x_i's are all zero
  - The following constraints do the trick:

$$\begin{aligned}
L_j w_j \le x_j \le L_j w_{j-1} &\qquad j = 1, 2, \dots, k \\
w_0 = 1 & \\
w_j \in \{0, 1\} &\qquad j = 1, 2, \dots, k \\
x_j \ge 0 &\qquad j = 1, 2, \dots, k.
\end{aligned}$$

- It follows from the constrains that $w_j \le w_{j-1}$ for $j = 1, 2, \dots, k$
  - This inequality implies that once one of the w_j is zero all the subsequent once must be zero
  - With this decomposition we can write the piecewise linear function as

$$K + c_1 x_1 + c_2 x_2 + \cdots + c_k x_k \tag{22}$$

10.6 Branch-and-Bound

- The standard **integer programming problem** is defined as follows:

$$\begin{aligned}
\text{maximize} \quad & c^T x \\
\text{subject to} \quad & Ax \le b \\
& x \ge 0 \\
& x \text{ has integer components.}
\end{aligned}$$

- The algorithm called **branch-and-bound** solves the standard integer problem
  - It starts out with the following wishful approach
    1. First ignore the constraint that the components of x be integers
    2. Solve the resulting linear programming problem
    3. Hope that the solution vector has all integer components
  - Hopes are almost always unfulfilled so a backup strategy is needed

- The simplest strategy of rounding down the numbers do not always work
  - ∗ The numbers might not be feasible
- The linear programming problem obtained by dropping the integrality constraint is called the **LP-relaxation**
  - ∗ Since it has fewer constraints its optimal solution provides an upper bound ζ^0 on the optimal solution ζ^*

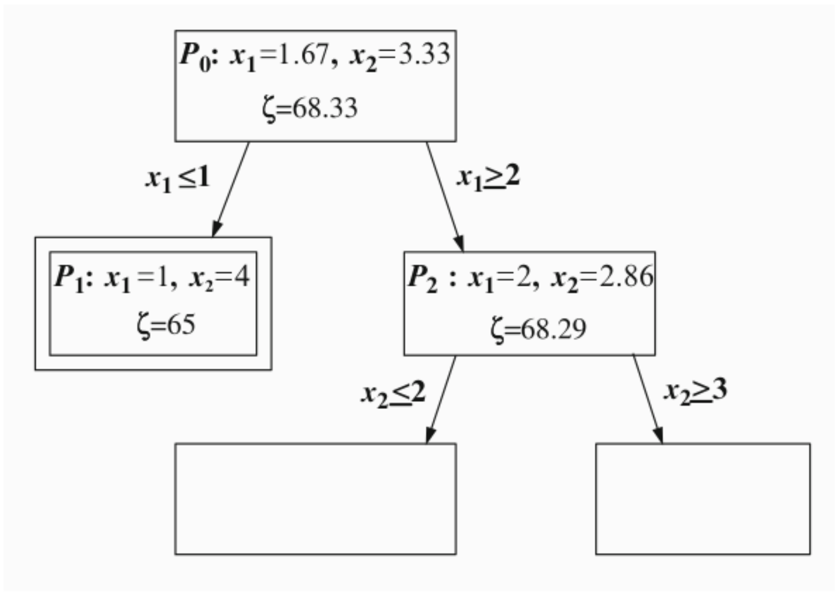


FIGURE 23.7. The beginnings of the enumeration tree.

- The algorithm works as follows
  - Solve the problem using the simplex algorithm
    1. If the solution is an integer solution check if it is greater than the current best solution
    2. Else if the value is greater than the current best solution split the problem into to for the first non-integer variable
       - ∗ e.g. if $x_i = 3.1$ we split into to problems one with $x_i \leq 3$ and one with $x_i \geq 4$
       - ∗ Solve the left problem first by going to step 1 and then the right problem

      * This is done using a enumeration tree

- Reasons for using depth first search

  1. Most integer solutions lie deep in the three there are two advantages to finding the integer feasible solutions early

     (a) It is better to have a feasible solution than nothing in case one wishes to abort the solution process early

     (b) Identifying a feasible integer solution can result in subsequent nodes of the enumaration tree begin made into leaves

        – This isbecause the optimal objective function associated with the nodes is lower than the best so-far integer solution

  2. The facts that it is very easy to code the algorithm as a recursive defined function

  3. As one moves deeper in the enumaration tree each subsequent linear problem is obtained from the preceding one by simply adding an upper/lower bound on one specific variable

11 Network Flows

11.1 Totally unimodular linear programs

11.1.1 The assignment problem

- There is given a $n \times n$ matrix of matrix of costs $C = (c_{ij})$

- The goal is to find a permutation π on $\{1, 2, \ldots, n\}$ minimizing $\sum_{i=1}^{n} c_{i,\pi(i)}$

- The following n^2 decision variables is chosen

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

- The problem can be formulated as an integer LP problem as follows:

$$
\begin{array}{llcll}
min & \sum_{i,j=1}^{n} c_{ij}x_{ij} & & & \\
s.t. & \sum_{j=1}^{n} x_{ij} & = & 1 & i = 1,\dots,n \\
 & \sum_{i=1}^{n} x_{ij} & = & 1 & j = 1,\dots,n \\
 & x_{ij} & \geq & 0 & i,j = 1,\dots,n \\
 & x_{ij} & \in & \mathbb{Z} &
\end{array}
$$

- Solving this problem by using the simplex algorithm without the integral constraint always gives us an integral solution

11.1.2 Cramer's rule and matrix determinants

- Let $A = (a_{ij})$ be a $m \times m$ matrix, $b \in \mathbb{R}^m$ and consider the linear system

$$Ax = b \tag{23}$$

- When the system has a unique solution it can be found using Cramer's rule

- **Theorem 1.** (Cramer's rule) The system has a unique solution if and only if $det(A) \neq 0$ and in that case it is given by

$$x_i = \frac{\det(A_i)}{\det A} \tag{24}$$

where A_i is obtained from A by replacing column i by the vector b

- **Observation 1.** Given that all entries of A and b are integers. Then $\det(A_i)$ is an integer for all i Now, a sufficient condition for x_i to be integer for all i is that $\det(A) \in -1, 1$

- **Proposition 1.** (Laplace's formula) Let $A = (a_{ij})$ be a $m \times m$ matrix. Then

$$\det(A) = \sum_{j=1}^{m} (-1)^{i+j} a_{ij} \det(A_{ij}) = \sum_{i=1}^{m} (-1)^{i+j} a_{ij} \det(A_{ij}) \ ,$$

where A_{ij} is the $(m-1) \times (m-1)$ matrix obtained from A by removing row i as well as column j

- **Proposition 2.** Let A be a $m \times m$ matrix. Then
  1. If B is obtained from A by exchanging two rows or two columns then $\det(B) = -\det(A)$
  2. If B is obtained from A my multiplying a row or a column by c, then $\det(B) = c \det(A)$
  3. If B is obtained from A by adding a multiple of a row to another row or by adding a multiple of a column to another column, then $\det(B) = \det(A)$

11.1.3 Totally unimodular matrices

- **Definition 1.** A matrix A is **totally unimodular** if every square submatrix of A has determininant either 0, 1 or -1
- **Theorem 2** (Hoffman and Kruskal's Theorem). Let A be an integer $m \times n$ matrix
  - A is totally unimodular if and only if for every integer vector $b \in \mathbb{Z}^m$ all the basic solutions of $F = \{Ax \leq b, x \geq 0\}$ are integer
- **Theorem 3.** Let $A = (a_{ij})$ be a totally unimodular $m \times n$ matrix and let $b \in \mathbb{Z}^m$. Then every basic solution of $F = \{A \leq b, x \geq 0\}$ is integer
- A linear program in standard form is called for totally unimodular if the coefficient matrix A is totally unimodular and the vector of constants b is integer valued
  - A desirable property since for many real life problems one is interested in integer solutions
- **Lemma 1.** If A is a matrix with entries from $\{-1, 0, 1\}$ with the property that each column contains at most two non-zero entries, at most one being 1 and at most one being -1 then A is totally unimodular

- **Lemma 2.** Let A be totally unimodular.
  - Then A^T is totally unimodular.
  - Let B be obtained from A by removing rows or columns, by exchanging rows, by exchanging columns, or by multiplying rows or columns by -1. Then B is also totally unimodular
- **Lemma 3.** Let A be totally unimodular. Then $[A \quad I]$ and $[A \quad A]$ are totally unimodular

11.2 Networks

11.2.1 General

- A **flow network** or simply a **network** is a directed graph $D = (\mathcal{N}, \mathcal{A})$
  - A **flow** in a network is an assignment of a real number to each arc, the **flow** on the arc
    * One may thus think of a flow as a function $x : \mathcal{A} \to \mathbb{R}$
    * One typically view the flow as an assignment to variables x_{ij} for each $ij \in A$
  - Various **constraints** can be imposed onto flow networks
    * There is always a nonnegative constraint, stating that $x_{ij} \geq 0$ for all $ij \in A$
    * Additional constraints that one can consider are **balance constraints** and arc **capacity constraints**
    * A flow that satisfies all given constraints is called a **feasible** flow
  - It is often built to model a real-life network
    * Much terminology is borrow from real life networks
  - For a node $i \in \mathcal{N}$ the **outgoing flow** from **i** is given by the summation of flow on all outgoing arcs from i, $\sum_{ij \in A} x_{ij}$
  - The **ingoing flow** from i is given by the summation of flow on all ingoing arcs to i, $\sum_{ji \in A} x_{ji}$
  - The **balance** of node i with respect to the flow x is given by the difference of the outgoing flow and the ingoing flow

$$b_i(x) = \sum_{ij \in a} x_{ij} - \sum_{ji \in a} x_{ji} \tag{25}$$

- Flow is supplied at a node i if $b_i(x) > 0$ we call i a **source** node
- Flow is consumed at a node i if $b_i(x) < 0$ we call i a **sink** node
- If a flow has no sources or sinks we call the flow a **circulation**

11.2.2 Node balance constraints

- A balance constraint is specified by balances b_i for each $i \in \mathcal{N}$ and states that

$$b_i(x) = b_i \tag{26}$$

for all $i \in \mathcal{N}$

- It is necessary to have $\sum_{i\in\mathcal{N}} b_i = 0$ in order to have feasible flows which always is an asumption when we have a balance constraint
- This means we always have

$$\sum_{i\in\mathcal{N}} b_i(x) = \sum_{i\in\mathcal{N}} \left(\sum_{ij\in\mathcal{A}} x_{ij} - \sum_{ji\in\mathcal{A}} x_{ji} \right) = \sum_{ij\in\mathcal{A}} x_{ij} - \sum_{ji\in\mathcal{A}} x_{ji} = 0 \ .$$

11.2.3 Arc constraints

- Arc constraint are specified by **upper bounds** u_{ij} (also called capacities) and lower bounds l_{ij} for each $ij \in \mathcal{A}$ and states that

$$l_{ij} \le x_{ij} \le u_{ij} \tag{27}$$

for all $ij \in \mathcal{A}$

- It is also necessary to have $0 \le l_{ij} \le u_{ij}$ for all $ij \in \mathcal{A}$ to have feasible flows
- When only upper bounds is specified we implicitly assume $l_{ij} = 0$ for all $ij \in \mathcal{A}$

11.3 The minimum cost flow problem

11.3.1 General

- The minimum cost flow problem is to minimize the **cost** of a flow in a network subject to certain given constraints
  - The most basic cost model is a linear cost model that assigns a real-valued cost c_{ij} to each arc $ij \in A$
    - The cost of an arc represents the cost per unit flow along that arc
    - It is also allowed to be a negative number
  - The cost of a given flow x is given by

$$c(x) = \sum_{ij \in \mathcal{A}} c_{ij} x_{ij} \tag{28}$$

- In the minimum cost flow problem we are always given balance constraints
- Capacity constraints are optional

11.3.2 Integrality theorem

- **Theorem 4.** Let $D = (\mathcal{N}, \mathcal{A})$ be a network
  - with costs given by $c_{ij} \in \mathbb{R}$
  - with balance constraints given by $b_i \in \mathbb{Z}$ for $i \in \mathcal{N}$
  - with lower and upper bounds given by $l_{ij}, u_{ij} \in \mathbb{Z}$ where $0 \leq l_{ij} \leq u_{ij}$, for $ij \in \mathcal{A}$
  - Then there is a minimum cost feasible flow x with an integer valued flow on every arc

11.4 The maximum (s, t) flow problem

11.4.1 General

- The (s, t) flow problem may be viewed as a important special case of the minimum cost flow problem
  - One is given a flow network $D = (\mathcal{N}, \mathcal{A})$ with specified upper bound, together with two nodes $s, t \in \mathcal{N}$ being singled out

  - The node s is thought of as a source node
  - The node t is thought of as a sink node
  - The **flow conservation** constraint states that $b_i(x) = 0$ for all $i \in \mathcal{N} \backslash \{s, t\}$
  - A flow is feasible if it satisfies
    * The non-negativity constraint
    * The flow conservation constraint
    * The arc constraint given by the upper bounds
  - For a feasible we define the **value** as $|x| = b_s(x)$
  - The maximum (s, t) flow problem is to find a feasible flow x maximizing the value $|x|$

- To model the maximum (s, t) flow problem as a minimum cost flow problem the following is done
  - Given a network $D = (\mathcal{N}, \mathcal{A})$ with a specified source s and sink t we add to $\mathcal{A}$ an arc from t to s obtaining the new network D'
  - We state the minimum cost flow problem on D' by given the new arc from t to s cost -1 and upper bound ∞
  - All other arcs are given cost 0
  - All nodes are given balance 0
  - Feasible flows x in D corresponds to feasible flows x' in D' where the cost of x' is the negative of the value of x

11.4.2 The max-flow min-cut theorem

- A reason for the importance of the maximum (s, t) flow problem is the celebrated max-flow min-cut theorem

- An (s, t) cut of D is a partition (S, T) of the nodes of $D, \mathcal{N} = S \cup T$ such that $s \in S$ and $t \in T$
  - The **capacity** $u(S, T)$ of a (s, t) cut (S, T) is given by

$$u(S, T) = \sum_{\substack{ij \in \mathcal{A} \\ i \in S, j \in T}} u_{ij} \ .$$

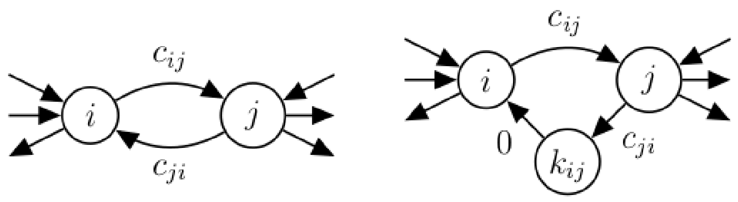
- The minimum capacity (s,t) cut problem is the associated minimization problem of finding a (s,t) cut of minimum capacity
- **Theorem 5** (Max-flow min-cut Theorem). Let x be a flow of maximum value and let (S,T) be a (s,t) cut of minimum capacity. Then

$$|x| = u(S,T) \tag{29}$$

11.5 Solving the minimum cost flow problem

11.5.1 Introduction

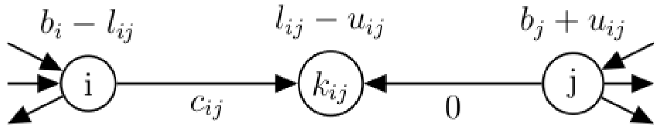
- The two basic assumptions placed on networks are the following
  1. The network is connected when viewed as an undirected graph
  2. The network does not contain any 2-cycles
     - i.e. if ij is an arc then the arc ji is not present in the network
- If the graph is not connected one can just solve the minimum cost flow problem for each graph and then combine them
- The two cycles assumption is only made for clarity of presentation
  - The algorithm considered work even in the presence of 2 cycles
  - One can eliminate 2 cycles by subdividing one of the arcs introducing an auxiliary node



11.5.2 Transformation to uncapacitated networks

- Transforming a network with arc-constraints to an uncapacitated network can be done by introducing an auxiliary node for each arc adjusting balances
  - An arc ij with lower bound l_{ij}, upper bound u_{ij} and cost c_{ij} with arcs from is replaced with arcs from node i and j to an auxilary node j_{ij} with the balance $l_{ij} - u_{ij}$

- The cost of the arc (i, l_{ij}) becomes c_{ij} and the cost of the arc (k_{ij}, i) becomes 0
- A flow x in the original network corresponds to the flow x' in the transformed network letting $x'_{i,k_{ij}} = x_{ij} - l_{ij}$ and $x'_{j,k_{ij}} = u_{ij} - x_{ij}$
- A flow x' in transformed network corresponds to to the flow x in the original network letting $x_{ij} = x'_{i,k_{ij}} + l_{ij}$
- The cost of the two flows are related by a constant difference $c(x') = c(x) - \sum_{ij \in \mathcal{A}} l_{ij} c_{ij}$



- This transformation can be used on the general network, so one can use the algorithm from the book

11.5.3 Klein's cycle cancelling algorithm

1. General
    - The general case of a network $D = (\mathcal{N}, \mathcal{A})$ is considered with balances b_i lower bounds l_{ij}, upper bounds u_{ij} and costs c_{ij}
    - The cycle cancellation algorithm abandons the restriction to tree solution and allows any cycle in the network to be a candidate for improving the cost the current flow
        - If no cycle can be found that improves the network the algorithms terminates and returns the current solution
        - A **cycle** C in D will be a simple cycle in D when viewed as an undirected graph together with a direction
            - C is just a sequence of nodes $i_1, i_2, \dots, i_\ell \in \mathcal{N}$ with $i_\ell = i_1$ butt all other i_k being different and such that either
            (a) $(i_k, i_{k+1}) \in \mathcal{A}$ for all $k = 1, \dots, \ell - 1$ which is called a **forward edge** of C
            (b) $(i_{k+1}, i_k) \in \mathcal{A}$ for all $k = 1, \dots, \ell - 1$ which is called a **backward edge** of C
        - Let C be a cycle in D

  * Let F be the set of forward edges of C and
  * Let B be the set of backward edges of C
  * The cost $c(C)$ of the cycle C is defined as

$$c(C) = \sum_{ij \in F} c_{ij} - \sum_{ij \in B} c_{ij} \tag{30}$$

- Given a real number δ, define the cycle flow γ_C^δ as

$$\gamma_C^\delta = \begin{cases} \delta & \text{if } ij \in F \\ -\delta & \text{if } ij \in B \\ 0 & \text{otherwise} \end{cases}$$

- **Observation 2.** Let C be a cycle in D
  (a) The flow γ_C^δ is a circulation. That is $b_i(\gamma_C^\delta) = 0$ for all $i \in \mathcal{N}$
  (b) The cost of γ_C^δ is given by $c(\gamma_C^\delta) = \delta \cdot c(C)$
- **Definition 2.** Let x be a feasible flow in D and let C be a cycle in D. Then C is an **augmenting cycle** relative to x if $c(C) < 0$ and there exists $\delta > 0$ such that $x + \gamma_C^\delta$ is a feasible flow in D
- When C is a cycle such that $c(C) < 0$. Then C is an augmenting cycle relative to x if and only if $x_{ij} < u_{ij}$ for all $ij \in F$ and $l_{ij} < x_{ij}$ for all $ij \in B$.
  - When C is an augmenting cycle, the maximum $\delta > 0$ for which $x + \gamma_C^\delta$ remains a feasible flow, which we will denote by $\delta(C)$ given by

$$\delta(C) = \min\left\{ \min_{ij \in F}(u_{ij} - x_{ij}), \min_{ij \in B}(x_{ij} - l_{ij}) \right\} .$$

┘

- Outline of Klein's cycle cancelling algorithm

```
1: Let $x$ be a feasible flow.
2: while exist augmenting cycle $C$ do
3:     $x := x + \gamma_C^{\delta(C)}$
4: end while
5: return $x$
```

2. Finding the first feasible flow

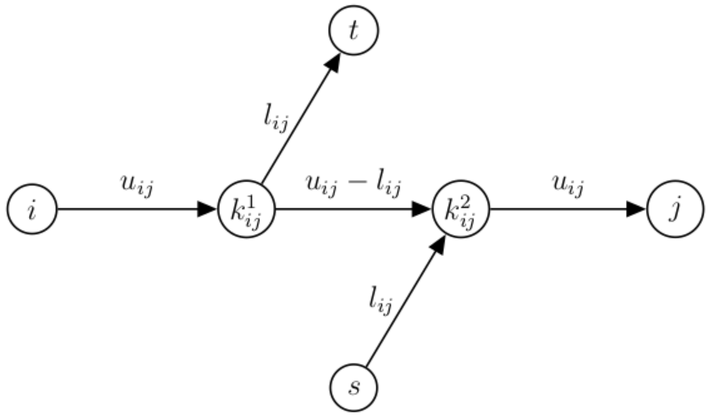


Figure 6: The part of D' corresponding to the arc ij of D.

- The problem is formulated as a maximum flow problem
  - It can be solved by Ford-Fulkerson or Edmonds-Karp algorithm
  - Given a flow network $D = (\mathcal{N}, \mathcal{A})$ a new flow network $D' = (\mathcal{N}', \mathcal{A}')$ is build
    * $\mathcal{N}' = \{s, t\} \cup \mathcal{N} \cup \{k_{ij}^{\ell} \mid ij \in \mathcal{A}, \ell = 1, 2\}$
    * The arcs $\mathcal{A}'$ of the network D' corresponds to the nodes and arcs of D
    * The arcs in D' corresponding to the nodes of D are as follows
      · From s we have an arc to each $i \in \mathcal{N}$ for which $b_i > 0$
      · To t we have an arc from each $i \in \mathcal{N}$ for which $b_i < 0$
    * The arcs in D' corresponding to arcs $ij \in \mathcal{A}$ are as follows
      · We have arcs forming a path from i to j through nodes k_{ij}^1 and k_{ij}^2
      · We have an arc from s to k_{ij}^2 and an arc from k_{ij}^1 to t
    * An arc from s to i is given upper bound b_i
    * An arc from i to t is given upper bound $-b_i$
    * The arcs from i to k_{ij}^1 and from k_{ij}^2 is given upper bounds $u_i j$

  * The arcs from s to k^2_{ij} and from k^1_{ij} to t are given upper bound l_{ij}
  * D has a feasible flow if and only if D' has a (s,t) flow x' that saturates all arcs from the source s

3. Finding an augmenting cycle

- The problem of finding an augmenting cycle is reduced to finding a negative weight directed cycle in a weighted directed graph
- Let $D = (\mathcal{N}, \mathcal{A})$ be a network with balances b_i, lower bounds l_{ij}, upper bounds u_{ij} and costs c_{ij}
- Let x be a feasible flow in D
- The residual network relative to x is the network $D_x = (\mathcal{N}, \mathcal{A}_x)$
  - It has the same set of nodes as D
  - The set of arcs indicates how the flow x may be changed maintaining feasibility
  - The arcs $\mathcal{A}_x$ of D_x are given by

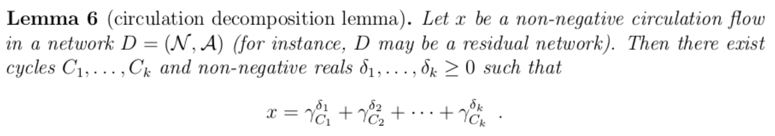
$$\mathcal{A}_x = \{ij \mid ij \in \mathcal{A} \;\&\; x_{ij} < u_{ij}\} \cup \{ji \mid ij \in \mathcal{A} \;\&\; l_{ij} < x_{ij}\}$$

- Each arc $ij \in \mathcal{A}$ of D can give rise to $0, 1$ or 2 arcs of D_x between the nodes i and j
- When $x_{ij} < u_{ij}$ the arc $ij \in A_x$ is given upper bound $(u_x)_{ij} = u_{ij} - x_{ij}$
  - This specifies how much the flow on arc ij can be increased
  - It is given cost $(c_x)_{ij} = c_{ij}$
- When $l_{ij} < x_{ij}$ the arc $ji \in \mathcal{A}_x$ is given upper bound $(u_x)_{ji} = x_{ij} - l_{ij}$
  - This specifies how much the flow on arc ij can be decreased
  - It is given cost $(c_x)_{ji} = -c_{ij}$
- These upper bounds is called **residual capacities**
- The balances of all nodes are set to 0
- All the arcs of D_x is given lower bound 0
- Let x and y be feasible flow in D
  - The flow $z = y - x$ in D is a circulation

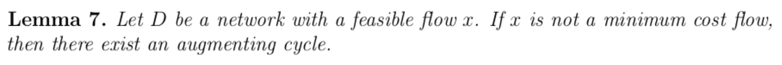
- The corresponding flow $\tilde{z}$ in D_x is defined by giving positive flow according to the following rules
  - If $z_{ij} > 0$ we let $\tilde{z}_{ij} = z_{ij}$ it also implies $x_{ij} < y_{ij} \leq u_{ij}$ which means that the arc ij is present in D_x
  - If $z_{ij} < 0$ we let $\tilde{z}_{ji} = -z_{ij}$ it also implies $x_{ij} > y_{ij} \geq l_{ij}$ which means that the arc ji is present in D_x
- Arcs of D_x not assigned flow by these rules are given flow 0
- **Lemma 4.** $\tilde{z}$ is a feasible flow in D_x and $c_x(\tilde{z}) = c(z) = c(y) - c(x)$

- Let x be a feasible flow in D, and let $\tilde{z}$ be a feasible flow in D_x
  - Define the flow z in D by the following rules specifying the flow z_{ij} on arc $ij \in \mathcal{A}$
    - If $ij \in \mathcal{A}_x$ and $ji \in \mathcal{A}_x$ let $z_{ij} = \tilde{z}_{ij} - \tilde{z}_{ji}$
    - If $ij \in \mathcal{A}_x$ and $ji \notin \mathcal{A}_x$ let $z_{ij} = \tilde{z}_{ij}$
    - If $ij \notin \mathcal{A}_x$ and $ji \in \mathcal{A}_x$ let $z_{ij} = -\tilde{z}_{ji}$
  - Let $y = x + z$
  - **Lemma 5.** y is a feasible flow in D and $c(y) = c(x) + c(z) = c(x) + c_x(\tilde{z})$
- The problem of finding an augmenting cycle in D relative to x is precisely the problem of finding a negative weight directed cycle in D_x when using the costs as weights

4. Partial correctness


Lemma 6 (circulation decomposition lemma). *Let x be a non-negative circulation flow in a network $D = (\mathcal{N}, \mathcal{A})$ (for instance, D may be a residual network). Then there exist cycles $C_1, \ldots, C_k$ and non-negative reals $\delta_1, \ldots, \delta_k \geq 0$ such that*

$$x = \gamma_{C_1}^{\delta_1} + \gamma_{C_2}^{\delta_2} + \cdots + \gamma_{C_k}^{\delta_k} \ .$$


Lemma 7. *Let D be a network with a feasible flow x. If x is not a minimum cost flow, then there exist an augmenting cycle.*

12 Network Flow Problems

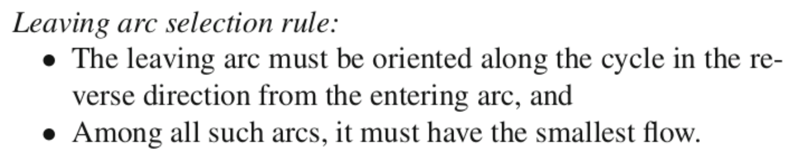
12.1 Spanning Trees and Bases

- An ordered list of nodes $(n_1, n_2, \ldots, n_k)$ is called a **path** in the network if each adjacent pair of nodes in the list is connected by an arc in the network

  - It is not assumed that the arcs point in any particular direction

- A network is called connected if there is a path connecting every pair of nodes
- For any arc (i, j) i is its **tail** and j is its **head**
- A **cycle** is a path in which the last node coincides with the first node
- A network is called **acyclic** if it does not contain cycles
- A network is a **tree** if it is connected and acyclic
- A network $(\tilde{\mathcal{N}}, \tilde{\mathcal{A}})$ if called a **subnetwork** of $(\mathcal{N}, \mathcal{A})$ if $\tilde{\mathcal{N}} \subset \mathcal{N}$ and $\tilde{\mathcal{A}} \subset \mathcal{A}$
  - It is a **spanning tree** if it is a tree and $\tilde{\mathcal{N}} = \mathcal{N}$
  - It is suffices to refer to a spanning tree by simply giving the arc set
- Given a network flow problem any selection of primal flow values that satisfies the balance equations at every node is called a **balanced flow**
  - If all flows are nonnegative the a it is called a **feasible flow**
- Given a spanning tree, a balance flow that assigns zero flow to every arc not on the spanning tree is called a **tree solution**
- It is assumed that the network is connected
  - **Theorem** A square submatrix of $\tilde{A}$ is a basis if and only if the arcs to which its columns correspond form a spanning tree

12.2 The Primal Network Simplex Method



Leaving arc selection rule:

- The leaving arc must be oriented along the cycle in the reverse direction from the entering arc, and
- Among all such arcs, it must have the smallest flow.

Primal flows update:

- Flows oriented in the same direction as the leaving arc are decreased by the amount of flow that was on the leaving arc whereas flows in the opposite direction are increased by this amount.

Dual variables update:

- If the entering arc crosses from the root-containing tree to the non-root-containing tree, then increase all dual variables on the non-root-containing tree by the dual slack of the entering arc.
- Otherwise, decrease these dual variables by this amount.

Dual slacks update:

- The dual slacks corresponding to those arcs that bridge in the same direction as the entering arc get decremented by the old dual slack on the entering arc, whereas those that correspond to arcs bridging in the opposite direction get incremented by this amount.

12.3 The Dual Network Simplex Method

Entering arc selection rule:

- The entering arc must bridge the two subtrees in the opposite direction from the leaving arc, and
- Among all such arcs, it must have the smallest dual slack.

12.4 Putting It All Together

- The **self-dual network simplex method**
  1. **Identify a spanning tree**
     - Any one will do
     - Also identify a root node
  2. **Compute initial primal flows** on the tree arcs by assuming that nontree arcs have zero flow and at each node must be balanced

  - For this calculation the computed primal flows may be negative
  - In this case the initial primal solution is not feasible
  - The calculation is performed working from leaf nodes inward
3. **Compute initial dual values** by working out from the root node along the tree arcs using the formula $y_j - y_i = c_{ij}$ which is valid on tree arcs since the dual slacs vanish on these arcs
4. Compute **Initial dual slacks** on each nontree arcs using the formula $z_{ij} = y_i + c_{ij} - y_j$
  - Some of the z_{ij} might be nonnegative
  - This is okay but it is important that the satisfy the equality
5. **Perturb** each primal flow and each dual slack that has a negative initial value by adding a positive scalar μ to each such value
6. **Identify a range** $\mu_{\text{MIN}} \leq \mu \leq \mu_{\text{MAX}}$ over which the current solution is optimal
  - On first iteration μ_{MAX} will be infinite
7. **Check the stopping rule**: if $\mu_{\text{MIN}} \leq 0$ then set $\mu = 0$ to recover an optimal solution
  - While not optimal perform each of the remaining steps and the return to recheck this condition
8. **Select an arc** associated with the inequality $\mu_{\text{MIN}} \leq \mu$ (if there are several pick one arbitrarily)
  (a) If the pivot is a primal pivot, the arc identified above is the **entering arc**
    - If the arc is a nontree arc then the current pivot is a primal pivot
    - Add the entering arc to the tree
    - The leaving arc should be chosen from the arcs on the cycle that go in the opposite direction
    - The leaving arc should have the smallest flow among all such arc (evaluated at $\mu = \mu_{\text{MIN}}$)
  (b) If the pivot is a dual pivot the arc identified above is the **leaving arc**
    - If the pivot it is a tree arc then the pivot is a dual pivot
    - Delete the leaving arc from the tree
    - The deletion splits the tree into two subtrees

    - The entering arc must bridge the trees in the opposite direction
    - It should be the one with smallest dual slack

9. **Update primal flow** as follows
   - Add the entering arc to the tree (creates a cycle containing both the entering and leaving arcs)
   - Adjust the flow on the leaving arc to zero
   - Adjust the flow of the other cycle arcs as necessary to maintain flow balance
10. **Update dual variables** as follows
    - Delete the leaving arc from the old tree
    - The deletion splits the old tree into to subtrees
    - Let $\mathcal{T}_u$ denote the subtree containing the tail of the entering variable
    - Let $\mathcal{T}_v$ denote the subtree containing the head
    - The dual variables for nodes in $\mathcal{T}_u$ remain unchanged
    - The dual variable for nodes in $\mathcal{T}_v$ get incremented by the old dual slack on the entering arc
11. **Update dual slacks** as follows
    - All dual slacks remain unchanged except for those associated with nontree arcs that bridge the two subtrees $\mathcal{T}_u$ and $\mathcal{T}_v$
    - The dual slacks corresponding to those arcs that bridge in the same direction as the entering arc get decremented by the old dual slack on the entering arc
    - Those that correspond to arcs bridging in the opposite direction get incremented by this amount

12.5 The Integrality Theorem

- Only network flow problem where all the supplies and demands are integers are considered
- **Integrality Theorem** For network flow problems with integer data, every basic solution and in particular every basic optimal solution assigns integer flow to every arc
- **König's Theorem** Suppose there are n girls and n boys that every girl knows exactly k boys and that every boy knows exactly k girls.

Then n marriages can be arranged with everybody knowing his or her spouse

13 The Ellipsoid Algorithm

13.1 LP, LI and LSI

- **Linear programming** (LP) in standard form is the following computational problem:
  - Given an integer $m \times n$ matrix A, m vector b and n vector c either
    1. Find a rational n vector x such that $x \geq 0$, $Ax = b$ and c is minimized subject to these conditions
    2. Report that there is no n vector such that $x \geq 0$ and $Ax = b$
    3. Report that the set $\{c'x | Ax = b, x \geq 0\}$ has no lower bound
- The problem of **linear inequalities** (LI) is defined as follows:
  - Given an integer $m \times n$ matrix A and m vector b is there an n vector x such that $Ax \leq b$
- It is assumed that $m \geq n$ in LI and LSI
  - This is not really restrictive
- LI is almost as hard as LP
- **Lemma 8.4** An integer x bet 1 and B can be determined by $O(\log(B))$ questions of the form "Is $x > a$"
- Given an $m \times n$ LP in standard form

$$\begin{aligned} \min\ & c'x \\ Ax &= b \\ x &\geq 0 \end{aligned}$$

- Its size is

$$L = mn + O(log(|P|)) \tag{31}$$

- where P is the product of the nonzero (integer) coefficients appearing in A, b and c

- **Lemma 8.5** The bfs's of a LP in standard form are n vectors of rational numbers, both the absolute value and the denominators of which are bounded by 2^L
- **Lemma 8.6** Suppose that two bfs's x_1, x_2 of a standard LP satisfy $K2^{-2K} < c'x_1, c'x_2 \leq (K+1)2^{-2L}$ for some integer K. Then $c'x_1 = c'x_2$
- **Theorem 8.2** There is a polynomial-time algorithm for LP if and only if there is a polynomial-time algorithm for LI
- The following is the problem of **linear strict inequalities**
  - Given an $m \times n$ integer matrix A and m vector b is there an n vector x such that $Ax < b$
- **Lemma 8.7** The system of inequalities

$$a_i'x \leq b_i, \quad i = 1, \dots, m \tag{32}$$

has a solution iff the system of linear strict inequalities

$$a_i'x < b_i + \epsilon, \quad i = 1, \dots, m \tag{33}$$

has a solution, where $\epsilon = 2^{-2L}$

- **Corollary** If there is a polynomial-time algorithm for LSI, then there is a polynomial-time algorithm for LI

13.2 Affine Transformations and Ellipsoids

- The Q be an $n \times n$ nonsingular matrix, and t an n vector
  - The transformation $T : \mathbb{R}^n \to \mathbb{R}^n$ defined as $T(x) = t + Q \cdot x$ for each $x \in \mathbb{R}^n$ is called an **affine transformation**
  - Since Q is nonsingular, T is a uniquely invertible transformation
  - The inverse of T is an affine transformation itself
  - The **unit sphere** is the set

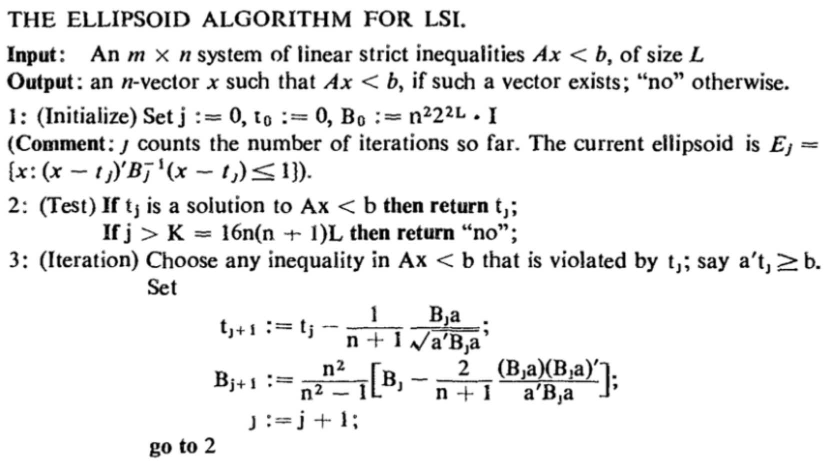
$$S_n = \{x \in \mathcal{R}^n \mid x'x \leq 1\} \tag{34}$$

- If T is an affine transformation, then $T(S_n)$ is called an **ellipsoid**
  - $T(S_n) = \{y \in \mathbb{R}^n \mid (y-t)'B^{-1}(y-t) \leq 1\}$ where $B = QQ^T$

  - A matrix such as B is positive definite i.e. $x'Bx > 0$ for all nonzero $x \in \mathcal{R}^n$
  - Affine transformation preserve set inclusion

- **Lemma 8.8** If $S \subseteq S' \subseteq \mathbb{R}^n$, then $T(S) \subseteq T(S')$
- **Lemma 8.9** Suppose that a subset S of $\mathbb{R}^n$ has volume V. Then $T(S)$ has volume $V \cdot |\det(Q)|$
- **Lemma 8.10** Let $a \in \mathbb{R}^n$ be a vector of length $||a||$. There is a rotation R such that $Ra = (||a||, 0, \ldots, 0)$
- Consider a convex polytope P in $\mathcal{R}^n$
  - It can be written as $P = \{x \in \mathcal{R}^n \mid Ax \leq b\}$ for some $m > n$, $m \times n$ matrix A and m vector b
  - Let the interior of P be defined as follows $\text{Int}(P) = \{x \in \mathbb{R}^n \mid Ax < b\}$
- **Lemma 8.11** If $\text{Int}(P) \neq \emptyset$ then there exist $n+1$ linearly independent vertices of P

13.3 Algorithm



```
THE ELLIPSOID ALGORITHM FOR LSI.
Input:  An $m \times n$ system of linear strict inequalities $Ax < b$, of size $L$
Output: an $n$-vector $x$ such that $Ax < b$, if such a vector exists; "no" otherwise.
1: (Initialize) Set $j := 0$, $t_0 := 0$, $B_0 := n^2 2^{2L} \cdot I$
(Comment: $j$ counts the number of iterations so far. The current ellipsoid is $E_j = \{x: (x - t_j)' B_j^{-1} (x - t_j) \leq 1\}$).
2: (Test) If $t_j$ is a solution to $Ax < b$ then return $t_j$;
          If $j > K = 16n(n+1)L$ then return "no";
3: (Iteration) Choose any inequality in $Ax < b$ that is violated by $t_j$; say $a' t_j \geq b$.
               Set
                    $t_{j+1} := t_j - \frac{1}{n+1} \frac{B_j a}{\sqrt{a' B_j a}}$;
                    $B_{j+1} := \frac{n^2}{n^2 - 1} \left[ B_j - \frac{2}{n+1} \frac{(B_j a)(B_j a)'}{a' B_j a} \right]$;
                    $j := j + 1$;
               go to 2
```

Figure 8–5

Theorem 8.3 *Let B_j be a positive definite matrix, let $t_j \in R^n$, and let a be any nonzero n-vector. Let B_{j+1} and t_{j+1} be as in Step 3 of the ellipsoid algorithm. Then the following hold.*

(a) *B_{j+1} is positive-definite (or, equivalently, $E_{j+1} = \{x \in R^n: (x - t_{j+1})'B_{j+1}^{-1}(x - t_{j+1}) \leq 1\}$ is an ellipsoid).*

(*b*) *The semiellipsoid*

$$\tfrac{1}{2}E_j[a] = \{x \in R^n: (x - t_j)'B_j^{-1}(x - t_j) \leq 1, \quad a'(x - t_j) \leq 0\}$$

is a subset of E_{j+1}.

(*c*) *The volumes of E_j and E_{j+1} satisfy*

$$\frac{\text{vol}\,(E_{j+1})}{\text{vol}\,(E_j)} < 2^{-1/2(n+1)}$$

Lemma 8.12 *Consider the sphere S_n and the set $E = \{x \in R^n: (x - t)'B^{-1}(x - t) \leq 1\}$, where $t = (-1/(n+1), 0, \ldots, 0)'$ and $B = \text{diag}\,(n^2/(n+1)^2, n^2/(n^2-1), \ldots, n^2/(n^2-1))$.*

(*a*) *B is positive-definite (and hence E is an ellipsoid).*

(*b*) *The hemisphere $\tfrac{1}{2}S_n = \{x \in R^n: x'x \leq 1$ and $x_1 \leq 0\}$ is a subset of E.*

(*c*) *The volumes of S_n and E satisfy*

$$\frac{\text{vol}\,(E)}{\text{vol}\,(S_n)} < 2^{-1/2(n+1)}$$

Lemma 8.13 *Let B_j be a positive-definite matrix, $t_j \in R^n$, and let a be any nonzero n-vector. Let t_{j+1} and B_{j+1} be obtained as in Step 3 of the ellipsoid algorithm. Let $\tfrac{1}{2}S_n$ and E be as in previous lemma. Then there exists an affine transformation T such that*

(*a*) $T(S_n) = \{x \in R^n: (x - t_j)'B_j^{-1}(x - t_j) \leq 1\}$;

(*b*) $T(E) = \{x \in R^n: (x - t_{j+1})'B_{j+1}^{-1}(x - t_{j+1}) \leq 1\}$;

(*c*) $T(\tfrac{1}{2}S_n) = \{x \in R^n: (x - t_j)'B_j^{-1}(x - t_j) \leq 1, a'(x - t_j) \leq 0\}$.

Lemma 8.14 *If an LSI system of size L has a solution, then the set of solutions within the sphere $||x|| \leq n2^L$ has volume at least $2^{-(n+2)L}$.*

Theorem 8.4 *The ellipsoid algorithm correctly decides whether a system of LSI's has a solution.*

14 The Central Path

14.1 General

- A lower case letter denotes a vector quantity and the upper-case form of the same letter denote the diagonal matrix whose diagonal entries are those of the corresponding vector e.g

$$x = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} \implies X = \begin{bmatrix} x_1 & & & \\ & x_2 & & \\ & & \ddots & \\ & & & x_n \end{bmatrix}.$$

14.2 The Barrier Problem

- The linear programming problem considered is the following

$$\begin{aligned} \text{maximize} \quad & c^T x \\ \text{subject to} \quad & Ax \le b \\ x \ge 0 & \end{aligned}$$

- The corresponding dual problem is

$$\begin{aligned} \text{minimize} \quad & b^T y \\ \text{subject to} \quad & A^T y \ge c \\ y \ge 0 & \end{aligned}$$

- Slack variables is used to convert both problem to equality form
- Given a constrained maximization problem where some of the constraints are inequalities one can consider replacing any inequality constraint with an extra term in the objective function
  - e.g. x_j is nonnegative can be removed by adding to the objective function a term that is negative infinity when x_j is negative and is zero otherwise
  - The problem with such a function is that one cannot use calculus to study it
  - One could replace it with another function that approaches negative infinity as x_j approaches zero
    * e.g. the logarithm function

- The following problem is called the **barrier problem** associated with the given function

$$\begin{array}{ll} \text{maximize} & c^T x + \mu \sum_j \log(x_j) + \mu \sum_i \log(w_i) \\ \text{subject to} & Ax + w = b \end{array}$$

- When the parameter μ gets small it, the function becomes a better and better standin for the original function
- It is a family of problems associated with the parameter μ
- Each of the problems is a nonlinear problem
- The nonlinear objective function is called a **barrier function** or **logarithmic barrier function**
- The set of optimal solutions to the barrier problems forms a path through the interoir of the polyhedron of feasible solutions
  - This path is called the **central path**

14.3 Lagrange Multipliers

- There is a single constraint equation so the problem can be formally stated as

$$\begin{array}{ll} \text{maximize} & f(x) \\ \text{subject to} & g(x) = 0 \end{array}$$

- The gradient of f, denoted ∇f is a vector that points in the direction of most rapid increase of f
- For unconstrained optimization one would solve it by setting the gradient equal to zero to determine the **critical points** of f
  - The maximum if it exists would part of this set
- The gradient must be orthogonal to the set of feasible solutions $\{x \mid g(x) = 0\}$
- At each point x in the feasible set $\nabla g(x)$ is a vector that is orthogonal to the feasible set at this point x

- The new requirement for a point x^* to be a critical point is that it is feasible and that $\nabla f(x^*)$ to be proportional to $\nabla g(x^*)$
  - Writing it out as a system of equations one has

$$\begin{aligned} g(x^*) &= 0 \\ \nabla f(x^*) &= y \nabla g(x^*) \end{aligned}$$

- y is a proportionally constant, which can be any real number
  - This proportionality constant is called a **Lagrange multiplier**
- We consider several constraints:

$$\begin{aligned} \text{maximize} \quad & f(x) \\ \text{subject to} \quad & g_1(x) = 0 \\ & g_2(x) = 0 \\ & \quad \cdots \\ & g_m(x) = 0 \end{aligned}$$

- The feasible region is the interaction of m hypersurfaces
  - The space orthogonal to the feasible set at a point x is a instead a higher dimensional space (typical m) given by the span of the gradients
- It is required that $\nabla f(x^*)$ lie in this span
- This yields the following set of equation for a critical point

$$\begin{aligned} g(x^*) &= 0 \\ \nabla f(x^*) &= \sum_{i=1}^{m} y_i \nabla g(x^*) \end{aligned}$$

- The derivation of these equation can be done using the **Lagrangian** function

$$L(x, y) = f(x) - \sum_{u} y_i g_i(x) \tag{35}$$

- and look at its critial points over both x and y

- The following matrix is called the **Hessian** of f at x

$$Hf(x) = \left[\frac{\partial^2 f}{\partial x_i \partial x_j}\right] \tag{36}$$

- **Theorem 17.1** If the constraints are linear, a critical point x^* is a local maximum if

$$\xi^T Hf(x^*)\xi < 0 \tag{37}$$

for each $\xi \neq 0$ satisfying

$$\xi^T \nabla g_i(x^*) = 0, \quad i = 1, 2, \ldots, m$$

14.4 Lagrange Multipliers Applied to the Barrier Problem

- The following is the **first-order optimality conditions**

$$\begin{aligned}
\frac{\partial L}{\partial x_j} &= c_j + \mu \frac{1}{x_j} - \sum_i y_i a_{ij} &&= 0, \quad j = 1, 2, \ldots, n, \\
\frac{\partial L}{\partial w_i} &= \mu \frac{1}{w_i} - y_i &&= 0, \quad i = 1, 2, \ldots, m. \\
\frac{\partial L}{\partial y_i} &= b_i - \sum_j a_{ij} x_j - w_i &&= 0, \quad i = 1, 2, \ldots, m.
\end{aligned}$$

- Writing the first order optimality conditions in matrix form one gets the following

$$\begin{aligned}
A^T y - \mu X^{-1} e &= c \\
y &= \mu W^{-1} e \\
Ax + w &= b.
\end{aligned}$$

- Using $z = \mu X^{-1} e$ one can rewrite the first order optimality conditions like this

$$\begin{aligned}
Ax + w &= b \\
A^T y - z &= c \\
XZe &= \mu e \\
YWe &= \mu e.
\end{aligned}$$

- The following is called the μ complementarity conditions

$$\begin{aligned} x_j z_j &= \mu \qquad j = 1, 2, \dots, n \\ y_i w_i &= \mu \qquad i = 1, 2, \dots, m, \end{aligned}$$

14.5 Second-Order Information

- There is at most one critical point to the barrier problem by theorem 17.1

14.6 Existence

- The problems which doesn't have a maximum is rare
- **Theorem 17.2.** There exists a solution to the barrier problem if and only if both the primal and the dual feasible regions have nonempty interior
- **Corollary 17.3.** If a primal feasible set (or dual) has a non-empty interior and is bounded, then for each $\mu > 0$ there exists a unique solution

$$(x_\mu, w_\mu, y_\mu, z_\mu) \tag{38}$$

- The path $\{(x_\mu, w_\mu, y_\mu, z_\mu) \mid \mu > 0$ is called the **primal-dual central path**

15 A Path-Following Method

15.1 Algorithm

```
initialize $(x, w, y, z) > 0$
while (not optimal) {
    $\rho = b - Ax - w$
    $\sigma = c - A^T y + z$
    $\gamma = z^T x + y^T w$
    $\mu = \delta \frac{\gamma}{n+m}$
    solve:
        $A\Delta x + \Delta w = \rho$
        $A^T \Delta y - \Delta z = \sigma$
        $Z\Delta x + X\Delta z = \mu e - XZe$
        $W\Delta y + Y\Delta w = \mu e - YWe$
    $\theta = r \left( \max_{ij} \left\{ -\frac{\Delta x_j}{x_j}, -\frac{\Delta w_i}{w_i}, -\frac{\Delta y_i}{y_i}, -\frac{\Delta z_j}{z_j} \right\} \right)^{-1} \wedge 1$
    $x \leftarrow x + \theta\Delta x$,        $w \leftarrow w + \theta\Delta w$
    $y \leftarrow y + \theta\Delta y$,        $z \leftarrow z + \theta\Delta z$
}
```

15.2 General

- The path-following method is a one-phase method
  - It can begin from a point that is neither primal nor dual feasible
- One starts with an arbitrary choice of strictly positive values for all primal and dual variables i.e. $(x, w, y, z) > 0$ and then update these values as follows
  1. Estimate an appropriate value for μ
     - i.e. smaller than the "current" value but not too small
  2. Compute step directions $(\nabla x, \nabla w, \nabla y, \nabla z)$ poiting approximately at the point $(x_\mu, w_\mu, y_\mu, z_\mu)$ on the central path
  3. Compute a step length parameter θ such that the new point continues to have strictly positive components

  - $\tilde{x} = x + \theta \nabla x$
  - $\tilde{y} = y + \theta \nabla y$
  - $\tilde{w} = w + \theta \nabla w$
  - $\tilde{z} = z + \theta \nabla z$
4. Replace (x, w, y, z) with the new solution $(\tilde{x}, \tilde{w}, \tilde{y}, \tilde{z})$

15.3 Newton's Method

- Given a function

$$F(\xi) = \begin{bmatrix} F_1(\xi) \\ F_2(\xi) \\ \vdots \\ F_N(\xi) \end{bmatrix}, \qquad \xi = \begin{bmatrix} \xi_1 \\ \xi_2 \\ \vdots \\ \xi_N \end{bmatrix},$$

- from $\mathbb{R}^n$ into $\mathbb{R}^n$, a common problem is to find a point $\xi^* \in \mathbb{R}^N$ for which $F(\xi^*) = 0$
  - Newton method is an iterative method for solving this problem
- One step of the method is defined as follows
  - Given any point $\xi \in \mathcal{R}^n$ the goal is to find a step direction $\Delta\xi$ for which $F(\xi + \delta\xi) = 0$
  - For a nonlinear F it is not possible to find such a step direction
  - The step direction is approximated by the first two terms of its Taylor's series expansion

$$F(\xi + \Delta\xi) \approx F(\xi) + F'(\xi)\Delta\xi,$$

- where

o

$$F'(\xi) = \begin{bmatrix} \frac{\partial F_1}{\partial \xi_1} & \frac{\partial F_1}{\partial \xi_2} & \cdots & \frac{\partial F_1}{\partial \xi_N} \\ \frac{\partial F_2}{\partial \xi_1} & \frac{\partial F_2}{\partial \xi_2} & \cdots & \frac{\partial F_2}{\partial \xi_N} \\ \vdots & \vdots & & \vdots \\ \frac{\partial F_N}{\partial \xi_1} & \frac{\partial F_N}{\partial \xi_2} & \cdots & \frac{\partial F_N}{\partial \xi_N} \end{bmatrix}.$$

- Equating it to zero gives a linear system to solve for the step direction

$$F'(\xi)\Delta\xi = -F(\xi).$$

- Given $\Delta\xi$ Newton's method updates the current solution ξ by replacing it with $\xi + \Delta\xi$
  - It continues until the current solution is approximatly a root

15.4 Estimating an Appropriate Value for the Barrier Parameter

- The following is the formula for the value of μ whenever it is on the central path

$$\mu = \frac{z^T x + y^T w}{n + m}.$$

- It is used to estimate μ even when the current solution (x, w, y, z) does not lie on the central path
- The algorithm takes this value and reduces it by a certain fraction

$$\mu = \delta \frac{z^T x + y^T w}{n + m},$$

- δ is a value between 0 and 1 but generally a value of $\frac{1}{10}$ works quite well

15.5 Convergence Analysis

- The following is called the **p-norm**

$$\|x\|_p = \left(\sum_j |x_j|^p \right)^{\frac{1}{p}}.$$

- The following is called the **sub-norm**

$$\|x\|_\infty = \max_j |x_j|.$$

$$\gamma = z^T x + y^T w.$$

- Stopping rule
  - Let $\epsilon > 0$ be a small positive tolerance and let $M < \infty$ be a large finite tolerance
  - If $||x||_\infty$ gets larger than M one stops and declares the problem primal unbounded
  - If $||y||_\infty$ gets larger than M one stops and declares the problem dual unbounded
  - If $||p||_1 < \epsilon, ||\sigma||_1 < \epsilon$ and $\gamma < \epsilon$ then we stop and declare the current solution to be optimal

16 The Homogeneous Self-Dual Method

```
initialize
    $(x, y, \phi, z, w, \psi) = (e, e, 1, e, e, 1)$
while (not optimal) {
    $\mu = (z^T x + w^T y + \psi\phi)/(n + m + 1)$
    $\delta = \begin{cases} 0, & \text{on odd iterations} \\ 1, & \text{on even iterations} \end{cases}$
    $\hat{\rho} = -(1 - \delta)(Ax - b\phi + w) + w - \delta\mu Y^{-1} e$
    $\hat{\sigma} = -(1 - \delta)(-A^T y + c\phi + z) + z - \delta\mu X^{-1} e$
    $\hat{\gamma} = -(1 - \delta)(b^T y - c^T x + \psi) + \psi - \delta\mu/\phi$
    solve the two $(n + m) \times (n + m)$ quasidefinite systems:
        $\begin{bmatrix} -Y^{-1}W & A \\ A^T & X^{-1}Z \end{bmatrix} \begin{bmatrix} f_y \\ f_x \end{bmatrix} = \begin{bmatrix} \hat{\rho} \\ -\hat{\sigma} \end{bmatrix}$
    and
        $\begin{bmatrix} -Y^{-1}W & A \\ A^T & X^{-1}Z \end{bmatrix} \begin{bmatrix} g_y \\ g_x \end{bmatrix} = \begin{bmatrix} -b \\ -c \end{bmatrix}$
    $\Delta\phi = \dfrac{c^T f_x - b^T f_y + \hat{\gamma}}{c^T g_x - b^T g_y - \psi/\phi}$
    $\begin{bmatrix} \Delta x \\ \Delta y \end{bmatrix} = \begin{bmatrix} f_x \\ f_y \end{bmatrix} - \begin{bmatrix} g_x \\ g_y \end{bmatrix} \Delta\phi$
    $\Delta z = -X^{-1} Z \Delta x + \delta\mu X^{-1} e - z$
    $\Delta w = -Y^{-1} W \Delta y + \delta\mu Y^{-1} e - w$
    $\Delta\psi = -\frac{\psi}{\phi}\Delta\phi + \delta\mu/\phi - \psi$
    $\theta = \begin{cases} \max\{t : (x(t), \ldots, \psi(t)) \in \mathcal{N}(1/2)\}, & \text{on odd iterations} \\ 1, & \text{on even iterations} \end{cases}$
    $x \leftarrow x + \theta\Delta x$,        $z \leftarrow z + \theta\Delta z$
    $y \leftarrow y + \theta\Delta y$,        $w \leftarrow w + \theta\Delta w$
    $\phi \leftarrow \phi + \theta\Delta\phi$,        $\psi \leftarrow \psi + \theta\Delta\psi$
}
```

16.1 From Standard Form to Self-Dual Form

- The linear programming problem is given in standard form

$$\begin{array}{ll} \text{maximize} & c^T x \\ \text{subject to} & Ax \leq b \\ & x \geq 0 \end{array}$$

and its dual

$$\begin{array}{ll} \text{minimize} & b^T y \\ \text{subject to} & A^T y \geq c \\ & y \geq 0. \end{array}$$

- The dual and the primal problem can be solved by solving the following problem

$$\begin{array}{lrcrcrcl} \text{maximize} & & & & & 0 & & \\ \text{subject to} & & - & A^T y & + & c\phi & \leq & 0, \\ & Ax & & & - & b\phi & \leq & 0, \\ & -c^T x & + & b^T y & & & \leq & 0, \\ & & & x,\ y,\ \phi & & & \geq & 0. \end{array}$$

- One new variable ϕ has been added and one new constraint have been added
  - The total number of variables is $n + m + 1$
  - The total number of variables is $n + m + 1$
  - Homogenous linear programming problems having a skew symmetric constraint matrix are called **self-dual**
  - A solution to this problem with $\gamma > 0$ can be converted into solutions for the primal and dual problem
  - For a solution $(\bar{x}, \bar{y}, \bar{\phi})$ to this problem a solution to the dual and primal (x^*, y^*) can be extracted by

$$x^* = \bar{x}/\bar{\phi} \text{ and } y^* = \bar{y}/\bar{\phi} \tag{39}$$

16.2 Homogenous Self-Dual Problems

16.2.1 General

- A linear programming problem is called **self-dual** if $m = n$, $A = -A^T$ and $b = -c$

- The dual of such a problem is the same as the primal
- A linear problem in which the right-hand side vanishes is called a **homogeneous** problem
- If a problem is homogeneous and self-dual then its objective function must vanish too

- **Theorem 22.1.** For a homogeneous self-dual problem, the following statements hold:
  1. It has feasible solutions and every feasible solution is optimal
  2. The set of feasible solution has empty interios
     - If (x, z) is feasible then $z^T x = 0$
     - This means that these types of problems do not have central paths

16.2.2 Step Directions

- The intermediate solution will be infeasible
- The infeasibility of a solution (x, z) is denoted as

$$\rho(x, z) = Ax + z \tag{40}$$

- The number $\mu(x, z)$ measures the degree of noncomplementarity between x and z

$$\mu(x, z) = \frac{1}{n} x^T z \tag{41}$$

- $\rho(x, z)$ is written as ρ and $\mu(x, z)$ is written as μ when x, z is clear from context
  - Step directions $(\Delta x, \Delta z)$ are chosen to reduce the infeasibility and noncomplementarity of the current solution by a given factor δ, $0 \leq \delta \leq 1$
  - The following linear system of equations is used for the step directions

$$\begin{aligned} A\Delta x + \Delta z &= -(1 - \delta)\rho(x, z), \\ Z\Delta x + X\Delta z &= \delta\mu(x, z)e - XZe. \end{aligned}$$

- With those step directions, we pick a step length θ and step to a new point

$$\bar{x} = x + \theta \Delta x, \qquad \bar{z} = z + \theta \Delta z.$$

- The new ρ vector is denoted as $\bar{\rho}$ and the new μ value is denoted as $\bar{\mu}$

THEOREM 22.2. *The following relations hold:*
(1) $\Delta z^T \Delta x = 0$.
(2) $\bar{\rho} = (1 - \theta + \theta\delta)\rho$.
(3) $\bar{\mu} = (1 - \theta + \theta\delta)\mu$.
(4) $\bar{X}\bar{Z}e - \bar{\mu}e = (1 - \theta)(XZe - \mu e) + \theta^2 \Delta X \Delta Z e$.

16.2.3 Predictor Corrector Algorithm

- For each $0 \leq \beta \leq 1$ let

$$\mathcal{N}(\beta) = \left\{(x, z) > 0 : \|XZe - \mu(x, z)e\| \leq \beta\mu(x, z)\right\}.$$

- $\beta < \beta'$ implies that $\mathcal{N}(\beta) \subsetneq \mathcal{N}(\beta')$
  - $\mathcal{N}(0)$ is the set of points for which XZe has all equal components
- The algorithm alternates between two types of steps
  - On the first iteration and on every other iteration the algorithm performs a **predictor step**
    - Before the predictor step, one assumes that $(x, z) \in \mathcal{N}(1/4)$
  - The step directions are computed using $\delta = 0$ and the step length is calculated so as not to go outside of $\mathcal{N}(1/2)$

$$\theta = \max\{t \mid (x + t\Delta x, z + t\Delta z) \in \mathcal{N}(1/2)\} \tag{42}$$

- On even iterations, the algorithm performs a **corrector step**
  - Before such a step one assumes that $(x, z) \in \mathcal{N}(1/2)$
- The step directions are computed using $\delta = 1$ and the step length parameter θ is set to 1

THEOREM 22.3. *The following statements are true:*

(1) *After a predictor step,* $(\bar{x}, \bar{z}) \in \mathcal{N}(1/2)$ *and* $\bar{\mu} = (1-\theta)\mu$.
(2) *After a corrector step,* $(\bar{x}, \bar{z}) \in \mathcal{N}(1/4)$ *and* $\bar{\mu} = \mu$.

- Let

$$\begin{aligned} p &= X^{-1/2}Z^{1/2}\Delta x, \\ q &= X^{1/2}Z^{-1/2}\Delta z, \\ r &= p + q \\ &= X^{-1/2}Z^{-1/2}(Z\Delta x + X\Delta z) \\ &= X^{-1/2}Z^{-1/2}(\delta\mu e - XZe). \end{aligned}$$

LEMMA 22.4. *The following statements are true:*

(1) $\|PQe\| \le \frac{1}{2}\|r\|^2$.
(2) *If* $\delta = 0$, *then* $\|r\|^2 = n\mu$.
(3) *If* $\delta = 1$ *and* $(x, z) \in \mathcal{N}(\beta)$, *then* $\|r\|^2 \le \beta^2\mu/(1-\beta)$.

17 Quadratic Programming

17.1 General

- Problems that would be linear except that the objective function is permitted to include terms involving products of pairs of variables are called **quadratic programming** problems
  - Such terms are called **quadratic terms**

17.2 The Markowitz Model

- Given a collection of potential investments, let R_j denote the return in the next time period on investment j, $j = 1, \dots, n$

- In general R_j is a random variable although some investments may be essentially deterministic
- A **portfolio** is determined by specifying what fraction of one's assets to put into each investment
  - It is a collection of nonnegative numbers x_j, $j = 1, \ldots, n$ that sum to one
  - The **return** (on each dollar) one would obtain by $R = \sum_j x_j R_j$
  - The **reward** associated with a portfolio is defined as the expected return $\mathbb{E}r = \sum_j x_j \mathbb{E}R_j$
  - The **risk** associated with an investment is the variance of the return $\mathrm{Var}(R) = \mathbb{E}\left(\sum_j x_j \tilde{R}_j \right)^2$
  - One would like to maximize the reqard while at the same time not incur excessive risk
  - In the Markowitz model one forms a linear combination of the mean and the variance and minimizes that

$$\begin{array}{ll}\text{minimize} & -\sum_j x_j \mathbb{E}R_j + \mu \mathbb{E}\left(\sum_j x_j \tilde{R}_j\right)^2 \\ \text{subject to} & \sum_j x_j = 1 \\ & \quad x_j \geq 0 \qquad j = 1, 2, \ldots, n.\end{array}$$

- μ is a positive parameter that represents the importance of risk relative to reward
  - High values of μ tend to minimize the risk at the expense of the reward
  - Low values of μ put more weight on reward
- The problem can be rewritten as

$$
\begin{aligned}
\text{minimize} \quad & -\sum_j r_j x_j + \mu \sum_i \sum_j x_i x_j C_{i,j} \\
\text{subject to} \quad & \sum_j x_j = 1 \\
& x_j \geq 0 \qquad j = 1, 2, \ldots, n,
\end{aligned}
$$

- where $C_{i,j} = \mathbb{E}(\tilde{R}_j, \tilde{R}_j)$ is the covariance matrix and $r_j = \mathbb{E}R_j$ for the mean return on investment j
- Solving this problem requires an estimate of the mean return for each of the investments as well as an estimate of the covariance matrix
  - These quantities are not know but instead must be estimated by looking at historical data
  - One way to estimate the mean $\mathbb{R}_j$ is to take the average of the historical returns
  - It is better to take a weighted average that puts more weight on the recent years

$$\mathbb{E}R_j = \frac{\sum_{t=1}^{T} p^{T-t} R_j(t)}{\sum_{t=1}^{T} p^{T-t}}.$$

- p is a discount factor which by setting to $p = 0.9$ gives more emphasis on recent years
- Logarithms are used in cancelling a return r and its reciprocal

$$\mathbb{E}R_j = \exp\left(\frac{\sum_{t=1}^{T} p^{T-t} \log R_j(t)}{\sum_{t=1}^{T} p^{T-t}} \right).$$

- Letting μ vary continuously generates a curve of optimal solutions
  - This is called the **efficient frontier**
  - One should only invest in portfolios that lie on the efficient frontier

17.3 The Dual

- Quadratic programming problems are usually formulated as minimizations
- Problems in the following form are considered

$$\begin{array}{ll} \text{minimize} & c^T x + \frac{1}{2} x^T Q x \\ \text{subject to} & Ax \geq b \\ & x \geq 0. \end{array}$$

- The matrix Q is assumed to be symmetric
- The following is the barrier problem associated with the previous problem

$$\begin{array}{ll} \text{minimize} & c^T x + \frac{1}{2} x^T Q x - \mu \sum_j \log x_j - \mu \sum_i \log w_i \\ \text{subject to} & Ax - w = b. \end{array}$$

- The following is the Lagrangian

$$\begin{aligned} f(x, w, y) = c^T x + \frac{1}{2} x^T Q x - \mu \sum_j \log x_j - \mu \sum_i \log w_i \\ + y^T (b - Ax + w). \end{aligned}$$

- The following is the first-order optimality conditions for the Lagrangian

$$\begin{aligned} A^T y + z - Qx &= c \\ Ax - w &= b \\ XZe &= \mu e \\ YWe &= \mu e. \end{aligned}$$

where z is defined as such

$$z = \mu X^{-1} e.$$

- The constraints for the dual problem is

$$\begin{aligned} A^T y + z - Qx &= c \\ y, z &\geq 0. \end{aligned}$$

- The dual problem can be stated as

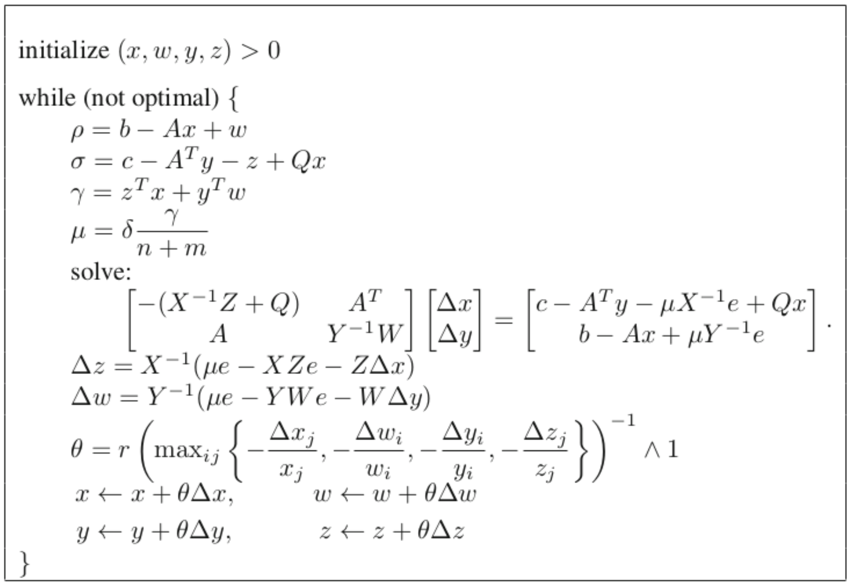
$$\begin{array}{ll} \text{maximize} & b^T y - \frac{1}{2} x^T Q x \\ \text{subject to} & A^T y + z - Qx = c \\ & \qquad\qquad y, z \geq 0. \end{array}$$

17.4 Convexity and Complexity

- A quadratic programming problem of the given form in which the matrix Q is positive semidefinite is called a **convex quadratic programming problem**

THEOREM 24.1. *For convex quadratic programming problems, given a solution* (x^*, w^*) *that is feasible for the primal and a solution* (x^*, y^*, z^*) *that is feasible for the dual, if these solutions make inequality* (24.4) *into an equality, then the primal solution is optimal for the primal problem and the dual solution is optimal for the dual problem.*

17.5 Solution via Interior-Point Methods



```
initialize $(x, w, y, z) > 0$
while (not optimal) {
    $\rho = b - Ax + w$
    $\sigma = c - A^T y - z + Qx$
    $\gamma = z^T x + y^T w$
    $\mu = \delta \frac{\gamma}{n+m}$
    solve:
        $\begin{bmatrix} -(X^{-1}Z + Q) & A^T \\ A & Y^{-1}W \end{bmatrix} \begin{bmatrix} \Delta x \\ \Delta y \end{bmatrix} = \begin{bmatrix} c - A^T y - \mu X^{-1} e + Qx \\ b - Ax + \mu Y^{-1} e \end{bmatrix}.$
    $\Delta z = X^{-1}(\mu e - XZe - Z\Delta x)$
    $\Delta w = Y^{-1}(\mu e - YWe - W\Delta y)$
    $\theta = r \left( \max_{ij} \left\{ -\frac{\Delta x_j}{x_j}, -\frac{\Delta w_i}{w_i}, -\frac{\Delta y_i}{y_i}, -\frac{\Delta z_j}{z_j} \right\} \right)^{-1} \wedge 1$
    $x \leftarrow x + \theta \Delta x$,        $w \leftarrow w + \theta \Delta w$
    $y \leftarrow y + \theta \Delta y$,        $z \leftarrow z + \theta \Delta z$
}
```

17.6 Practical Considerations

- A quadratic programming problem for which the matrix Q is diagonal is called a **separable quadratic programming problem**
  - Every non separable quadratic programming problem can be replaced by an equivalent separable version

18 Convex Programming

18.1 Differentiable Functions and Taylor Approximations

- All nonlinear functions will be assumed to be twice differentiable
  - Second derivatives will be assumed continuous
  - In first and second derivatives of a function in one dimension the gradient and Hessian are analogues of the first and second derivatives
  - They appear in the three term Taylor expansion of f about the point x

$$f(x+\Delta x) = f(x) + \nabla f(x)^T \Delta x + \frac{1}{2}\Delta x^T H f(x) \Delta x + r_x(\Delta x).$$

- The last term is called the remainder term which has the following property

$$\lim_{\Delta x \to 0} \frac{r_x(\Delta x)}{\|\Delta x\|^2} = 0.$$

18.2 Convex and Concave Functions

- A function is convex if its second derivative is nonnegative
- A real-valued function defined on a domain in $\mathbb{R}^n$ is **convex** if its Hessian is positive semidefinite everywhere in its domain
- A function is called **concave** if its negation is convex

18.3 Problem Formulation

- The convex optimization problems are posed in the following form

$$\begin{array}{ll} \text{minimize} & c(x) \\ \text{subject to} & a_i(x) \geq b_i, \qquad i = 1, 2, \ldots, m. \end{array}$$

- The real-valued function $c(\cdot)$ is assumed to be convex
- The m real-valued functions $a_i(x)$ are assumed to be concave
- The problem can be specified using vector notation as follows

$$\begin{array}{ll} \text{minimize} & c(x) \\ \text{subject to} & A(x) \geq b, \end{array}$$

- Where $A(\cdot)$ is a function from $\mathbb{R}^n$ into $\mathbb{R}^m$ and $b \in \mathbb{R}^m$
- w denote the slack variables that convert the inequality constraints to equalities

18.4 Solution via Interior-Point Methods

```
initialize $(x, w, y)$ so that $(w, y) > 0$

while (not optimal) {
    set up QP subproblem:
        $A = \nabla A(x)$
        $b = b - A(x) + \nabla A(x)x$
        $c = \nabla c(x) - Hc(x)x + \left(\sum_i y_i Ha_i(x)\right) x$
        $Q = Hc(x) - \sum_i y_i Ha_i(x)$
    $\rho = b - Ax + w$
    $\sigma = c - A^T y + Qx$
    $\gamma = y^T w$
    $\mu = \delta \frac{\gamma}{n+m}$
    solve:
        $\begin{bmatrix} -Q & A^T \\ A & Y^{-1}W \end{bmatrix} \begin{bmatrix} \Delta x \\ \Delta y \end{bmatrix} = \begin{bmatrix} c - A^T y + Qx \\ b - Ax - \mu Y^{-1} e \end{bmatrix}$
    $\Delta w = Y^{-1}(\mu e - YWe - W\Delta y)$
    $\theta = r \left( \max_{ij} \left\{ -\frac{\Delta x_j}{x_j}, -\frac{\Delta w_i}{w_i}, -\frac{\Delta y_i}{y_i} \right\} \right)^{-1} \wedge 1$
    do {
        $x^{\text{new}} = x + \theta \Delta x,$
        $w^{\text{new}} = w + \theta \Delta w$
        $y^{\text{new}} = y + \theta \Delta y$
        $\theta \leftarrow \theta/2$
    }while ( $\Psi(x^{\text{new}}, w^{\text{new}}) \geq \Psi(x, w)$ )
}
```

19 P, NP and NPC

19.1 Decision problems and language

- **Decision problems** are problems with should either output yes or no

- Problems where each instance is given by a string over $\{0,1\}$ can be described as a **language** i.e. a subset L of $\{0,1\}^*$
    - The members are the ones which results and "yes" and the non-members are the one which it is "no"
- Inputs is restricted to be strings over $\{0,1\}$
    - This only restricts string to be over some finite alphabet due to one being able to encode strings
    - This can also be used to represents things such as numbers or graphs
- One standard concrete representation is the **paring function** $\langle \cdot, \cdot \rangle$
    - If $x = x_1 x_2, \ldots, x_n$ and $y = y_1 y_2 \ldots y_m$ are strings over $\{0,1\}$, $\langle x, y \rangle$ denotes the string
        - $x_1 0 x_2 0 \ldots x_{n-1} x_n 011 y_1 0 y_2 0 \ldots y_{m-1} 0 y_m 0$
    - $\langle x, y \rangle$ is also a string over $\{0,1\}$
    - x and y can be reconstructed from $\langle x, y \rangle$
    - This is generalized to tuples and lists
- The inner products of vectors c and x is written as $c^T x$
- For some number $n \in \mathbb{N}$ we let $b(n) \in \{0,1\}^*$ denote its binary notation
- When representing strings a certain way some instances are malformed
    - They are just represented as no instances in the language
- If one wants to have a function f with a binary string as an output from a function one can represent it as the following decision problem L_f

$$L_f = \{\langle x, b(j), y \rangle | x \in \{0,1\}^*, j \in \mathbf{N}, y \in \{0,1\}, f(x)_j = y\}.$$

- The language L_f is used as a "stand in" for the function f
- Sometimes one is interested in solving problems which cannot obviously be expressed as computing a *function*

  - e.g. the simplex algorithm which can have multiple legal outputs for a given input

- A optimization problem OPT of the following form is given
  - OPT: "Given an input string defining a set of feasible solutions F and an objective function f, fin $x \in F$ maximizing $f(x)$"
  - Associated to OPT are the decision problem L_{OPT}
    - L_{OPT}: ""Given an input string defining F and f and a target value $v \in \mathbf{Q}$, decide if there is a solution $x \in F$ so that $f(x) \geq v$
  - L_{OPT} is used as a stand-in for OPT
    - It can be used to argue that OPT does not have an efficient algorithm

19.2 Turing machines and P

- A Turing machine consists of
  1. A (potentially infinite) bi-directional tape divided into cells each which is inscribed with a symbol from some finite **tape alphabet** Σ
     (a) It is assumed that the alphabet includes at least the symbols $0, 1$ and $\#$ which interpreted as blank
     (b) The position of a given cell is an integer indicating where the on the tape it is placed
  2. A read/write **head** that at any given point in time t is positioned at a particular cell
  3. A **finite control**, defined by a map

$$\delta : (Q - \{\texttt{accept}, \texttt{reject}\}) \times \Sigma \to Q \times (\Sigma \cup \{\texttt{left}, \texttt{right}\})$$

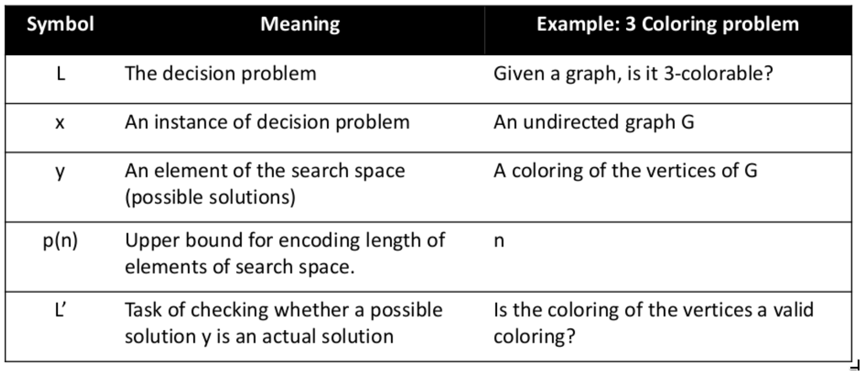
- There are tree distinguished states of Q
  - The start state `start`
  - The accepting state `state`
  - The rejecting state `reject`

- The finite control defines the operational semantics of the machine the following way
  - At any point in time $t \in \mathbf{Z}$ the finite control is exactly one of the states $q \in Q$
  - The tape cells are inscribed with a particular sequence
  - The head is positioned at a particular cell inscribed by some symbol $\sigma \in \Sigma$
  - Called the **configuration** of the machine
  - If q is either `accept` or `reject` the configuration is **terminal**
- **Church-Turing thesis** Any decision problem that can be solved by some mechanical procedure, can be solved by a Turing machine.
- Given a Turing Machine that decides some language, it decides the language in **polynomial time** if there is a fixed polynomial p, so that the number of steps taken on any input x is at most $p(|x|)$
- The **complexity class P** is defined as the class of language that are decided in polynomial time by some Turing machine
- **Polynomial Church-Turing thesis** A decision problem can be solved in polynomial time using a reasonable sequential model of computation and only if it can be solved in polynomial time by a Turing machine
- The following is a notion of polynomial time computable **map** $f : \{0,1\}^* \to \{0,1\}^*$
- We say f is polynomial time computable if the following two properties are true
  1. There is a polynomial p, so that $\forall x : |f(x)| \leq p(|x|)$
  2. $L_f \in \mathbf{P}$, where L_f is the decision problem associated with f that was defined in Section 1
- A representation π is **good** if $\pi(S)$ is in **P**
  - i.e. if it can be decided efficiently if a given string is a valid representation of an object
  - The representations π_1 and π_2 are **polynomially equivalent** if there are polynomial time computable maps r_1 and r_2 translating between the representations:

        * i.e. for all $x \in S$ $\pi_1(x) = r_1(\pi_2(x))$ and $\pi_2(x) = r_2(\pi_1(x))$

- **Proposition 1** If π_1 and π_2 are good representations that are polynomically equivalent, then $L_1 \in \mathbf{P} \Leftrightarrow L_2 \in \mathbf{P}$
- If L is in $\mathbf{P}$, then the problem is said to have a **pseudopolynomial time algorithm** when L is the language representing the problem using unary notation to represent integers

19.3 NP and the P vs. NP problem



Symbol	Meaning	Example: 3 Coloring problem
L	The decision problem	Given a graph, is it 3-colorable?
x	An instance of decision problem	An undirected graph G
y	An element of the search space (possible solutions)	A coloring of the vertices of G
p(n)	Upper bound for encoding length of elements of search space.	n
L'	Task of checking whether a possible solution y is an actual solution	Is the coloring of the vertices a valid coloring?

- **NP** is the class of languages L for which there exists a language $L' \in \mathbf{P}$ and a polynomial p, so that

$$\forall x : \quad x \in L \Leftrightarrow [\exists y \in \{0,1\}^* : |y| \leq p(|x|) \wedge \langle x, y \rangle \in L']$$

- This can be thought of as "simple" search verification problems in the following sense
    1. One requires that $L' \in \mathbf{P}$
        - i.e. one wants it to possible to efficiently check that a given solution y indeed is a valid solution for the instance x
    2. It is required that the potential solutions have length at most $p(|x|)$
        - i.e. length polynomial in the problem instance
- One could for a language L in **NP** decide is $x \in L$ by going through all $2^{p(|x|)+1} - 1$ possible values of y and check for each of them if $\langle x, y \rangle \in L'$

    - One do not want to perform this exhaustive search as it would take exponential time

- One clearly have that $\mathbf{P} \subseteq \mathbf{NP}$
    - If $L \in \mathbf{P}$ one can just define L' by $\langle x, y\rangle \in L'$ if and only if $x \in L$
        - i.e. simply ignoring the y part when deciding L
    - It is an open problem whether $P = NP$
    - Most people believe that there are problems in **NP** that are not in **P**
        - i.e. simple search verification problems for which there is no efficient alternative to searching through an exponentially big search space of possible solutions

- Set theory which is formalized by the formal system ZFC is generally accepted as a formal system strong enough to capture all mainstream mathematics
    - Theorems and proofs in ZFC are strings over some finite alphabet Σ
    - By encoding each symbol of Σ as a string over $\{0, 1\}$ one can think of theorems and proofs in ZFC as Boolean strings
    - Given an alleged theorem string t and an alleged proof string p we can decide in polynomial time in $|t| + |p|$ if p really is a proof of t

- **Proposition 2** If $\mathbf{P} = \mathbf{NP}$ then there is an algorithmic procedure that takes as input a formal statement t of ZFC and, if this statement has a proof in ZFC of length n, the procedure terminates in time polynomial in $|t| + n$ outputting the shortest proof of t

- It is believed that $\mathbf{P} \neq \mathbf{NP}$

19.4 Boolean Circuits

- A **Boolean circuit** with n input gates and m output gates is a directed, acyclic graph $G = (V, E)$
    - The vertices of V are called **gates**
    - Each gate has a **label** which indicates the type of gate

- A label is either taken from
    - The set of *function* symbols $\{AND, OR, NOT, COPY\}$
    - The set of *constant* symbols $\{0, 1\}$
    - The set of *variable* symbols $\{X_1, X_2, \ldots, X_n\}$
- For each j, at most one gate is labeled X_j
- The gates labeled with a variable symbol are called **input gates**
- The arcs in E are called *wires*
- The following constraints must be satisfied
    1. Each AND-gate and OR-gate has exactly two inputs
    2. Each NOT-gate and COPY gate has exactly one input
    3. The input gates are sources in G
- m of the gates are designated as **output gates**
    - They are called $o_1, o_2, \ldots, o_m$

• A Boolean circuit C defines or computes a Boolean function from $\{0,1\}^n$ to $\{0,1\}^m$ in the following way

- Given a vector $x \in \{0,1\}^n$ one can assign the value x_i to each gate labeled X_i
- The gates labeled 0 are assigned to Boolean values 0
- Values are iteratively assigned to the rest of the gates using the following rules
    1. Whenever the two inputs of an AND-gate have been assigned values, one assigns the Boolean AND of these two values to the gate
    2. Whenever the two inputs of an OR-gate have been assigned values, one assigns the Boolean OR of these two values to the gate
    3. Whenever the two inputs of an NOT-gate have been assigned a value, one assigns the Boolean negation of this value to this gate
    4. Whenever the two inputs of an COPY-gate have been assigned a value, one assign the same value to the gate
- Each gate in the circuit will eventually be assigned a value since the circuit is acyclic

  - The value of the function of input x is the vector $y \in \{0,1\}^m$ obtained by collecting the values assigned to the output gates $o_1, o_2, \ldots, o_m$
  - This process is referred to as **evaluating** the circuit on input x
    * Written as $C(x) = y$

- **Lemma 8** For any Boolean function $f : \{0,1\}^n \to \{0,1\}^m$, there is a circuit C, so that $\forall x \in \{0,1\}^n : C(x) = f(x)$

- Circuits can be regarded as completely general computational deices
  - Takes unlike Turing Machines inputs of a fixed input length only
  - The **size** of a circuit mean the number of gates in the circuit

- **Lemma 9** Let any Turing machine M running in time at most $p(n) \geq n$ on inputs of length n where p is a polynomial be given
  - Then given any fixed input length n, there is a circuit C_n of size at most $O(p(n)^2)$ so that for all $x \in \{0,1\}^n$, $C_n(x) = 1$ if and only if M accepts x
  - The function mapping 1^n to a description of C_n is polynomial time computable

19.5 Reductions and the complexity class NPC

- Given two languages L_1 and L_2 there is a reduction from L_1 to L_2 or L_1 reduces to L_2 if there is a polynomial time computable function r so that for all $x \in \{0,1\}^*$ we have $x \in L_1$ if and only if $r(x) \in L_2$
  - Written as $L_1 \leq L_2$
  - $L_1 \leq L_2$ and $L_2 \leq L_1$ does not imply $L_1 = L_2$

- **Proposition 3** If $L_1 \leq L_2$ and $L_2 \leq L_3$ then $L_1 \leq L_3$

- **Proposition 4** If $L_1 \leq L_2$ and $L_2 \in \mathbf{P}$ then $L_1 \in \mathbf{P}$

- A language L is $\mathbf{N}P$ hard if it has the property that for all $L' \in \mathbf{NP}$ L' reduces to L

- **Proposition 5** Let L be an **NP** hard language. If $\mathbf{P} \neq \mathbf{NP}$ then $L \notin P$

- **NP** hard languages do not necessarily lie in **NP**

- The class **NPC** is defined to be the class of **NP** complete language to be those languages in **NP** that are **NP** hard
- **Proposition 6** Let $L \in \mathbf{NPC}$. Then $\mathbf{P} = \mathbf{NP}$ is and only if L is in **P**
- **Lemma 7** If L_1 is **NP** hard and $L_1 \leq L_2$ then L_2 is **NP** hard

19.6 Cook's Theorem

- Given a set of symbols denoting Boolean variables
  - A **literal** is a variable or its negation
  - A **clause** is a junction of a number of literals
  - A **Conjunctive Normal Form** (CNF) formula is conjunction of a number of clauses
- The **Satisfiability Problem** or **SAT** is the following decision problem:
  - Given a CNF formula, decide if there is assignment of truth values to the variables of the formula so that the formula evaluates to true
  - This type of assignment is referred to as a **satisfying assignment** of the formula
- **Theorem 10** SAT $\in$ NPC
- **Circuit SAT** is the following decision problem:
  - Given a Boolean circuit C, the there a vector x so that $C(x) = 1$
- **Theorem 11** CIRCUIT SAT $\in$ **NPC**
- **Proposition 12** CIRCUIT SAT $\leq$ SAT

20 NP-Complete Problems

20.1 Variants of Satisfiability

- Any computational problem if generalized enough will become **NP** complete or worse
- Any problem has special cases that are in **P**

- Let k sat, where $k \geq 1$ is an integer, be the special case of SAT in which the formula is in conjunctive normal form, and all clauses have k literals
- **Proposition 9.2** 3SAT is **NP** complete
- **Proposition 9.3** 3SAT remains NP-complete even for expression in which each variable is restricted to appear at most three times, and each literal at most twice
- Let ϕ be an instance of 2SAT one can define a graph $G(\phi)$ as follows
  - The vertices of G are the variables of ϕ an their negation
  - There is an arc (α, β) if and only if there is a clause $(\neg\alpha \vee \beta)$ (or $\beta \vee \neg\alpha$) in ϕ
  - **Theorem 9.1** ϕ is unnoticeable if and only if there is a variable x such that there are path from x to $\neg x$ to x in $G(\phi)$
  - **Collary** 2SAT is in NL (and therefore in **P**)
    * NL is the complexity class which can be solved by a using a logarithmic amount of memory space.
- The problem MAX2SAT is finding the assignment which satisfies the most clauses
  - This is an optimization problem
  - This can turned into a yes-no problem by adding a goal K of clauses
  - **Theorem 9.2:** MAX2SAT is **NP** complete
- NAESAT is the problem where in no clause all three literals are equal
- **Theorem 9.3:** NAESAT is **NP** complete

20.2 Graph-Theoretic Problems

- An undirected graph is a pair $G = (V, E)$, V is a finite set of notes and E is a set of unordered pair of nodes in V called edges
  - An edge is denoted $[i, j]$ and have no direction
- For an undirected graph G let $I \subseteq V$.

  - I is said to be independent if whenever $i, j \in I$ there is no edge between i and j
  - All graph with nodes have non-empty independent sets

- The INDEPENDENT SET problem is the following:
  - Given an undirected graph $G = (V, E)$ and a goal K is there an independent set I with $|I| = K$
- **Theorem 9.4:** INDEPENDENT SET is **NP** complete
- Let k DEGREE INDEPENDENT SET be the special case of INDEPENDENT SET problem which all degrees are at most k
- **Corollary 1:** 4-DEGREE INDEPENDENT SET is **NP** complete
- In the CLIQUE problem one is given a graph G and a goal K and i asked whether there exists a set of K notes that form a clique by having all possible edges between them
  - Complement to the INDEPENDENT SET problem
- NODE COVER ask if there is a set C with B or fewer nodes such that each edge G has at least one of its endpoints in C
- **Corollary 2:** CLIQUE and NODE COVER are **NP** complete
- **Theorem 9.5:** MAX CUT is **NP** complete
- The problem MAX BISECTION is:
  - We want to find a cut S, $V - S$ of size K or more such that $|S| = |V - S|$
- **Lemma 9.1:** MAX BISECTION is **NP** complete
- The minimization version of the bisection problem is called BISECTION WIDTH
- **Theorem 9.6:** BISECTION WIDTH is **NP** complete
- The HAMILTON PATH problem is the following problem:
  - Given an undirected graph, does it have a Hamilton path i.e. a path visiting each node exactly once

- **Theorem 9.7:** HAMILTON PATH is **NP** complete
- **Corollary:** TSP (D) is **NP** complete
- **Theorem 9.8:** 3 COLORING is **NP** complete

20.3 Sets and Numbers

20.3.1 General

- The **TRIPARTITE MATCHING** problem is defined as follows:
  - We are given three sets B, G and H (boys, girls and homes), each containing n elements, and a ternary relation $T \subseteq B \times G \times H$
  - One should find a set of n triples in T where no two should have a component in common
    - i.e. each boy is matched to a different girl, and each couple has a home of its own
- **Theorem 9.9:** TRIPARTITE MATCHING is **NP** complete
- In **SET COVERING** one are given a family $F = \{S_1, \ldots, S_n\}$ of subsets of a finite set U, and a budget B
  - One wants a set of B sets in F whose union U
- In **SET PACKING** one are given a family of subsets of a set U, and a goal K
  - One are asked if there are K pairwise disjoint sets in the family
- In **EXACT COVER BY 3-SETS** one are given a family $F = \{S_1, \ldots, S_n\}$ of subset of a set U, such that $|U| = 3m$ for some integer m, and $|S_i| = 3$ for all i
  - One are asked if there are m sets in F that are disjoint and have U as their union
- **Corollary:** EXACT COVER BY 3-SETS, SET COVERING, and SET PACKING are **NP** complete
- The **knapsack** problem is a special case of INTEGER PROGRAMMING
  - One must select among a set of n items

  - Item i has value v_i, and weight w_i which are both positive integers
  - There is a limit W to the total weight of the items we can pick
  - We which to pick certain items without repetitions to maximize the total value subject to the constraint that the total weight is at most W
    * i.e. one is looking for a subset $S \subseteq \{1, \ldots, n\}$ such that $\sum_{i \in S} w_i \leq W$ and $\sum_{i \in S} v_i$ is as large as possible
  - In the recognition version called KNAPSACK, we are also given a goal K and we wish to find a subset $S \subseteq \{1, \ldots, n\}$ such that $\sum_{i \in S} w_i \leq W$ and $\sum_{i \in S} v_i \geq K$
- KNAPSACK is **NP** complete

20.3.2 Pseudopolynomial Algorithms and Strong NP-completeness

- **Proposition 9.4:** Any instance of KNAPSACK can be solved in $o(nW)$ time, where n is the number of items and W is the weight limit
  - Does not prove that **P** = **NP** since it is not polynomial in the input size
- A problem is **strongly NP-complete** if it remains **NP** complete even if any instance of length n is restricted to contain integers of size at most $p(n)$ (a polynomial)
  - All problems defined in this section are strongly NP-complete except the KNAPSACK problem
- The BIN PACKING problem is defined as follows:
  - We are given:
    * N positive integers $a_1, a_2, \ldots, a_N$ (the items)
    * C (the capacity)
    * B (the number of bins)
  - We are asked whether these numbers can be partitioned into B subsets, each of which has total sum at most C
- **Theorem 9.11:** BIN PACKING is **NP** complete

21 Approximation Algorithms

21.1 General

- There are at least three ways to get around NP-completeness
    1. If the inputs are small an exponential running time may be satisfactory
    2. One may be able to isolate special cases that can be solved in polynomial time
    3. There might be near optimal solutions in polynomial time (either in worst case or the expected case)
        - This is often good enough
        - Called an **approximation algorithm**
- An algorithm for a problem has an **approximation ratio** of $p(n)$ if for any input of size n, the cost C of the solution is within a factor of $p(n)$ of the cost C^* of an optimal solution:

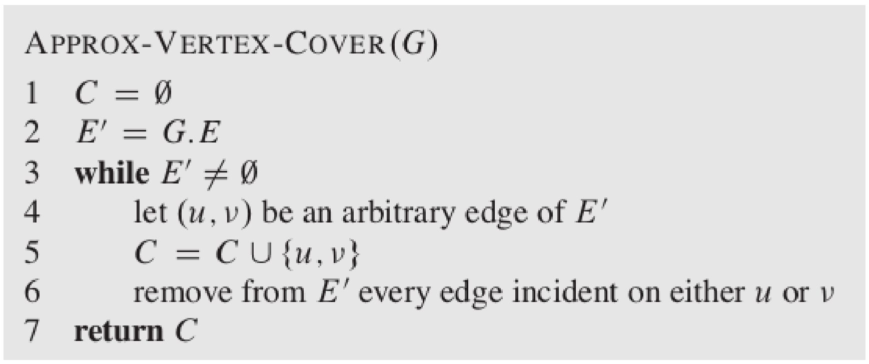
$$\max\left(\frac{C}{C^*}\frac{C^*}{C}\right) \leq p(n)$$

- If an algorithm achieves an approximation ration of $p(n)$ it is called a $p(n)$ **approximation algorithm**
- These definitions apply to both minimization and maximization algorithm
    - For a maximization problem, $0 < C^* \leq C$ and the ratio C/C^* gives the factor by which the cost of an optimal solution is larger than the solution of the approximate solution
    - For a minimization problem, $0 < C \leq C^*$, and the ration C^*/C gives the factor by which the cost of the approximate solution is larger than the cost of the optimal solution
- The approximation ratio of an approximation algorithm is never less than 1
    - A 1 approximation algorithm produces an optimal solution
    - An approximation algorithm with a large ratio may return a solution that is much worse tan optimal

- Some approximation algorithms can achieve better approximation ratios by using more an more computation time
- An **approximation scheme** for an optimization problem is an approximation algorithm that not only takes the instance of the problem as input but also a value $\epsilon > 0$ such that for any fixed ϵ the scheme is a $(1 + \epsilon)$ approximation scheme
  - It is a **polynomial-time approximation scheme** if for any fixed $\epsilon > 0$, the scheme runs in time polynomial in the size of n of its input instance
    - The running time of a polynomial-time approximation scheme can increase very rapidly as ϵ decreases
  - It is a **fully polynomial-time approximation scheme** if it is an approximation scheme and its running time is polynomial in both $1/\epsilon$ and the size n of the input instance

21.2 Vertex-cover problem

- A **vertex cover** of an undirected graph $G = (V, E)$ is a subset $V' \subseteq V$ such that if (u, v) is an edge of G, then either $u \in V'$ or V' (or both).
  - The size of a vertex cover is the number of vertices in it
- The **vertex-cover problem** is to find a vertex cover of minimum size in a given undirected graph
  - Such a vertex cover is called an **optimal vertex cover**
  - It is a NP complete problem
- The following is an approximation algorithm that takes as input an undirected graph G and returns a vertex cover whose size is guaranteed to be no more than twice the size of an optimal vertex cover



```
APPROX-VERTEX-COVER($G$)
$C = \emptyset$
$E' = G.E$
while $E' \neq \emptyset$
    let $(u, v)$ be an arbitrary edge of $E'$
    $C = C \cup \{u, v\}$
    remove from $E'$ every edge incident on either $u$ or $v$
return $C$
```

- **Theorem 35.1** APPROX-VERTEX-COVER is a polynomial-time 2-approximation algorithm

21.3 The traveling-salesman problem

21.3.1 General

- In the traveling-salesman one are given a complete undirected graph $G = (V, E)$ that has a nonnegative integer cost $c(u, v)$ associated with each edge $(u, v) \in E$ and a Hamilton cycle of G should be found with minimum cost
  - Let $c(A)$ denote the total cost of the edge in the subset $A \subseteq E$
  - The cost function C satisfies the **triangle inequality** if, for all vertices $u, v, w \in V$: $c(u, w) \leq c(u, v) + c(v, w)$
  - It is NP complete
  - An instance of the traveling-salesman problem where the triangle inequality holds is also NP complete

21.3.2 The TSP with the triangle inequality

- The following is an approximation of TSP where the resulting tours cost is no more than twice that of the minimum spanning trees weight as long as the triangle inequality is satisfies
  - It takes G a complete undirected graph and cost function c

```
APPROX-TSP-TOUR(G, c)
1  select a vertex r ∈ G.V to be a "root" vertex
2  compute a minimum spanning tree T for G from root r
       using MST-PRIM(G, c, r)
3  let H be a list of vertices, ordered according to when they are first visited
       in a preorder tree walk of T
4  return the hamiltonian cycle H
```

- It has running time $\Theta(V^2)$
- **Theorem 35.2** APPROX-TSP-TOUR is a polynomial-time 2-approximation algorithm for the traveling-salesman problem with the triangle inequality

21.3.3 The general TSP

- **Theorem 35.3** If $P \neq NP$, then for any constant $\rho \geq 1$, there is no polynomial-time approximation algorithm with approximation ratio ρ for the general TSP

21.4 Randomization and linear programming

- A randomized algorithm for a problem has an **approximation ratio** of $p(n)$ if, for any input of size n. the expected cost C of the solution produced by the randomized algorithm is within a factor of $p(n)$ of the cost C^* of an optimal solution:

$$\max\left(\frac{C}{C^*}, \frac{C^*}{C}\right) \leq p(n)$$

- A **randomized $p(n)$ approximation algorithm** is an randomized algorithm that achieves an approximation ratio of $p(n)$
- **MAX-3-CNF satisfiability** is the problem of finding the assignment that maximizes the amount of satisfied clauses

Theorem 35.6
Given an instance of MAX-3-CNF satisfiability with n variables $x_1, x_2, \ldots, x_n$ and m clauses, the randomized algorithm that independently sets each variable to 1 with probability 1/2 and to 0 with probability 1/2 is a randomized 8/7-approximation algorithm.

- In the **minimum-weight vertex-cover problem**, an undirected graph $G = (V, E)$ is given where each vertex $v \in V$ has an associated positive weight $w(v)$.
  - For any vertex cover $V' \subset V$, we define the weight of the vertex cover $w(V') = \sum_{v \in V'} w(v)$
  - The goal is to find a vertex cover of minimum weight
- The **minimum-weight vertex-cover problem** can be formulated as the following $0 - 1$ integer program where $x(v)$ defines whether v is in the vertex cover or not

$$\begin{array}{llll} \text{minimize} & \displaystyle\sum_{v \in V} w(v)\, x(v) & & (35.14) \\ \text{subject to} & & & \\ & x(u) + x(v) \geq 1 & \text{for each } (u, v) \in E & (35.15) \\ & x(v) \in \{0, 1\} & \text{for each } v \in V\ . & (35.16) \end{array}$$

- If all $w(v)$ is 1 it is just the vertex-cover problem
- If the constaint is removed that $x(v) \in \{0, 1\}$ the following **linear-programming relaxation** is obtained

$$\begin{array}{llll} \text{minimize} & \displaystyle\sum_{v \in V} w(v)\, x(v) & & (35.17) \\ \text{subject to} & & & \\ & x(u) + x(v) \geq 1 & \text{for each } (u, v) \in E & (35.18) \\ & x(v) \leq 1 & \text{for each } v \in V & (35.19) \\ & x(v) \geq 0 & \text{for each } v \in V\ . & (35.20) \end{array}$$

- This gives a lower bound on the value of an optimal solution to the 0-1 integer program
  - The following procedure uses this relaxation:

```
APPROX-MIN-WEIGHT-VC(G, w)
1  C = ∅
2  compute x̄, an optimal solution to the linear program in lines (35.17)–(35.20)
3  for each v ∈ V
4      if x̄(v) ≥ 1/2
5          C = C ∪ {v}
6  return C
```

Theorem 35.7
Algorithm APPROX-MIN-WEIGHT-VC is a polynomial-time 2-approximation algorithm for the minimum-weight vertex-cover problem.

21.5 The set covering problem

SET COVER

Instance: Universe X, $n = |X|$. Family $\mathcal{F}$ of subsets of X, such that $\cup_{S \in \mathcal{F}} S = X$.
Objective: Find $\mathcal{C} \subseteq \mathcal{F}$ that minimizes $|\mathcal{C}|$ and satisfies $\cup_{S \in \mathcal{C}} S = X$.

- The **SET COVER** problem is a generalization of the node covering problem:

NODE COVER

Instance: Graph $G = (V, E)$, $n = |V|$.
Objective: Find $\mathcal{C} \subseteq V$ that minimizes $|\mathcal{C}|$ and satisfies that for all $(u, v) \in E$ either $u \in \mathcal{C}$ or $v \in \mathcal{C}$.

Input: Universe X, Family $\mathcal{F}$ of subsets of X.
Output: Set cover $\mathcal{C}$.

```
1: $U \leftarrow X$
2: $\mathcal{C} \leftarrow \emptyset$
3: while $U \neq \emptyset$ do
4:     pick $S \in \mathcal{F}$ that maximizes $|S \cap U|$
5:     $U \leftarrow U \setminus S$
6:     $\mathcal{C} \leftarrow \mathcal{C} \cup \{S\}$
7: end while
8: return $\mathcal{C}$
```

Algorithm 1: Approximation algorithm for SET COVER.

- **Theorem 1** Algorithm 1 is a polynomial time approximation algorithm for SET COVER with approximation ratio $ln(n)$

Definition 2 *The kth Harmonic number H_k is defined as*

$$H_k = 1 + \frac{1}{2} + \cdots + \frac{1}{k} \ .$$

We remark that for all k we have $\ln(k) \leq H_k \leq \ln(k) + 1$, see Lemma 6.

Theorem 3 *Algorithm 1 is a polynomial time approximation algorithm for* SET COVER *with approximation ratio H_k, where $k = \max_{S \in \mathcal{F}} |S|$.*

Observation 4 *Specializing to* VERTEX COVER *gives*

$$\frac{|C|}{|C^*|} \le \max_{v \in V} H_{\deg(v)} \ .$$

When the degree of G is at most 3 we have an approximation algorithm with approximation ratio $H_3 = \frac{11}{6} < 2$.

21.6 The knapsack problem

KNAPSACK

Instance: Weights $w_1, \dots, w_n$, values $v_1, \dots, v_n$ and weight limit W.
Objective: Find $S \subseteq \{1, \dots, n\}$ that maximizes $\sum_{i \in S} v_i$ and satisfies $\sum_{i \in S} w_i \le W$.

Input: Weight $w_1, \dots, w_n$, values $v_1, \dots, v_n$, weight limit W and parameter $\epsilon > 0$.
Output: Selection of items, $S' \subseteq \{1, \dots, n\}$.

```
1: $V \leftarrow \max\{v_i \mid w_i \le W\}$
2: $B \leftarrow \epsilon V/n$
3: $v'_i \leftarrow \lfloor v_i/B \rfloor$
4: Compute optimal solution $S'$ using weights $w_1, \dots, w_n$, new values $v'_1, \dots, v'_n$ and weight
   limit $W$, by the dynamic programming algorithm.
5: return $S'$
```

Algorithm 2: Approximation algorithm for KNAPSACK.

Theorem 5 *Algorithm 2 computes in time $O(n^3/\epsilon)$ a solution that obtains a solution with value at least a $(1-\epsilon)$ fraction of the optimum value.*

21.7 Mathematical Preliminaries

We have the following simple bound on the Harmonic numbers.

Lemma 6

$$\ln(k) \le H_k \le \ln(k) + 1 \ .$$

Lemma 7 *For all $x \neq 0$ it holds that $1 + x < e^x$.*

22 TSP Problem (local optimization)

22.1 Introduction

- In the TSP
  - We are given
    - A set $\{c_1, c_2, \dots, c_n\}$ of cites
    - For each pair $\{c_i, c_j\}$ of distinct cities a distance $d(c_i, c_j)$

- The goal is to find an ordering π of the cities that minimizes the quantity

$$\sum_{i=1}^{N-1} d(c_{\pi(i),c_{\pi(i+1)}}) + d(c_{\pi(N)}, c_{\pi(i)})$$

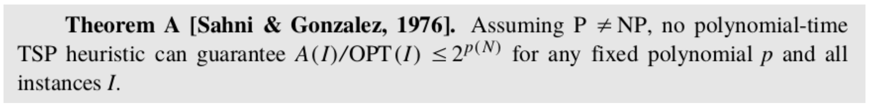
- The quantity is referred to as **tour length**
- The symmetric TSP in which the distance satisfy $d(c_i, c_j) = d(c_j, c_i)$ for $1 \leq i, j \leq N$

22.2 Tour Construction Heuristics

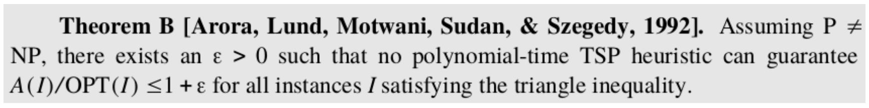
22.2.1 Introduction

- Every TSP heuristic can be evaluated in terms of two key parameters:
  - Its running time
  - The quality of the tours that it produces
- A heuristic is undominated if no competing heuristic both finds better tours and runs more quickly

22.2.2 Theoretical Results



Theorem A [Sahni & Gonzalez, 1976]. Assuming P $\neq$ NP, no polynomial-time TSP heuristic can guarantee $A(I)/\mathrm{OPT}(I) \leq 2^{p(N)}$ for any fixed polynomial p and all instances I.



Theorem B [Arora, Lund, Motwani, Sudan, & Szegedy, 1992]. Assuming P $\neq$ NP, there exists an $\varepsilon > 0$ such that no polynomial-time TSP heuristic can guarantee $A(I)/\mathrm{OPT}(I) \leq 1 + \varepsilon$ for all instances I satisfying the triangle inequality.

22.2.3 Four Important Tour Construction Algorithms

- **Nearest Neighbour**
  - It chooses the closest next city
  - It starts by chosen an arbitrary start city
  - Its running time is $\Theta(N^2)$
  - Its approximation ratio is $(0.5)(\log_2 N + 1)$
- **Greedy**

  - A tour is built up by continuously adding the smallest edge to the graph until there is a TSP
  - It has running time $\Theta(N^2 \log N)$
  - Its approximation ratio is $(0.5)(\log_2 N + 1)$

- **Clarke-Wright**
  - Starts with a tour where every path go through an arbitrarily chosen "hub" city
  - Contiguously removes an edge and chosen a new edge by taking the one which saves the most money
  - Stops when only two non-hub city exists
  - This algorithm can be implemented to run in time $\Theta(N^2 \log N)$
  - It has approximation ration $\log N + 1$

22.3 2-OPT, 3-OPT, and their variants

22.3.1 2-OPT and 3-OPT

- Local improvement algorithms for TSP is considered which is based on simple tour modification
- The **2-Opt** algorithm move
  - It deletes two edges and therefore breaks the tour into two paths
  - It reconnects the path in the other possible way
  - If the distance does not improve the move is not performed
- In the **3-opt**
  - The exchange replaces up to three edges of the current tour

22.3.2 Parallel 2- and 3-Opt

- To speed up TSP algorithm parallelism can be used
- **Geometric Partitioning**
  - It partitioning cities similar to that used in the construction of a $k - d$ tree data structure

- It is based on a recursive subdivision of the overall region containing the cities into rectangles with the set of cities in the corresponding rectangle
- A partitioning where each set contains no more that K cities is done as folllows
  - Let C_R be the set of cities assigned to rectangle R
  - Suppose $|C_R| > K$ one subdivides R into two subrectangles as follows
    - If the x coordinates have a larger range than the y coordinates find the city c with the median x value
    - Divide R into two subrectangles R_1 and R_2 by drawing a vertical line through c
    - Let R_1 be the rectangle be rectangle to the left of the line
    - Each city in C_R to the left of the line is assigned to R_1
    - Each city in C_R to the right of the line is assigned to R_2
    - Cities on the line are equal likely to be on the left or the right side
    - The city c is assigned to both R_1 and R_2
  - Once the cities have been partitioned in this way into subrectangles where each $|C_R| \leq K$, one can send each subrectangle to a processor
    - The instances can be converted to a tour for the whole graph using shortcuts as in the Christofides algorithm

- The alternative **tour-based partitioning** partitioning scheme bases its partition of the cities on the structure of the current tour and by performing more than one partioning phase
  - One begins by using a simple heuristic to generate an initial tour
  - Then the tour is broken into k segments of length N/k
    - Where k is greater than or equal to the number of processors available
  - Each segment is handed to a processor which
    - Converts the segment into a tour by adding an edge between its endpoints

    * Tries to improve the tour by local optimization with the constraint that the added edge cannot be deleted
  - The resulting tour can be converted into a segment again
    * Which then can be used to construct the overall tour

22.4 Tabu Search and the Lin-Kerninghan Algorithm

22.4.1 General

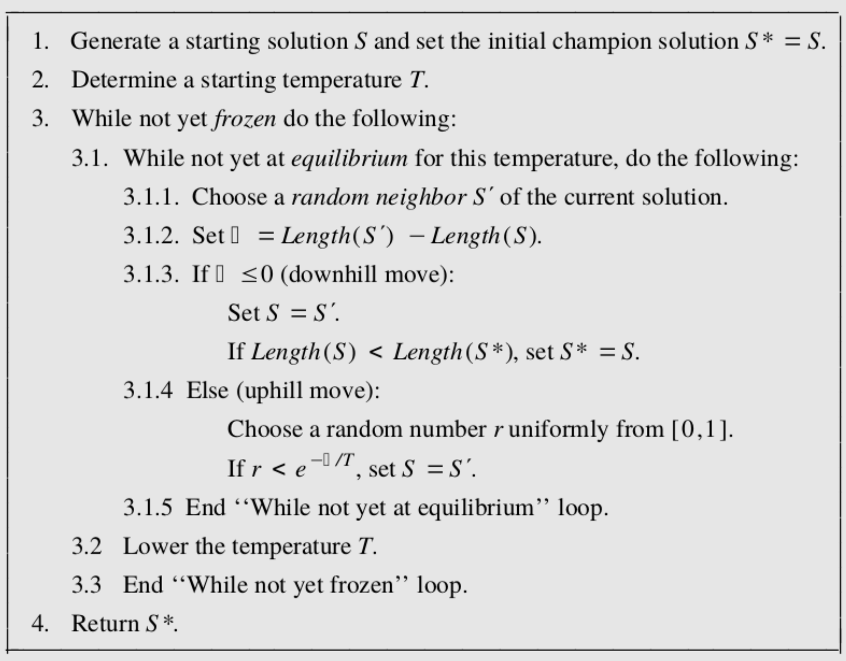
- The general strategy of **Tabu search** is looking for a local optimum and once one has been found identifying the best neighbouring solution which is then used as a starting points for a new local optimization phase
  - The moves used to find the best solution is kept in a **tabu list** and this is used to disqualify new moves that would undo the work of recent moves

22.4.2 The Lin-Kernighan Algorithm

- It is based on 2-Opt moves
- Given the current solution viewed as an anchored path P
  - The **anchor** of the path is fixed endpoint city t_1
  - Let t_{2i} denote the other endpoint of the path P_i that exists at the beginning of step i of the LK-search
    * The tour corresponding to path P_i can be obtained by adding the edge t_{2i}, t_1
    * At each step only 2-Opt moves are considered that flip some suffix of the path
      · i.e. ones in which one of the tour edges being broken is $\{t_1, t_{2i}\}$
    * The new neighbor t_{2i+1} of t_{2i} must be such that the length of the **one-tree** (spanning tree plus one edge) is less that the length of the best tour seen so far

- It keeps track of two lists a added list of added elements which cannot be deleted and a list of deleted elements which cannot be added again
    - Ensures that at most $O(N)$ be moves can be made
- The differences from standard Tabu Search thus
    1. unbounded tabu lists with no aspiration level
    2. restriction on the set of moves considered at each step
- The usefulness of the Lin-Kernighan Algorithm is by doing it multiple times

22.5 Simulated Annealing



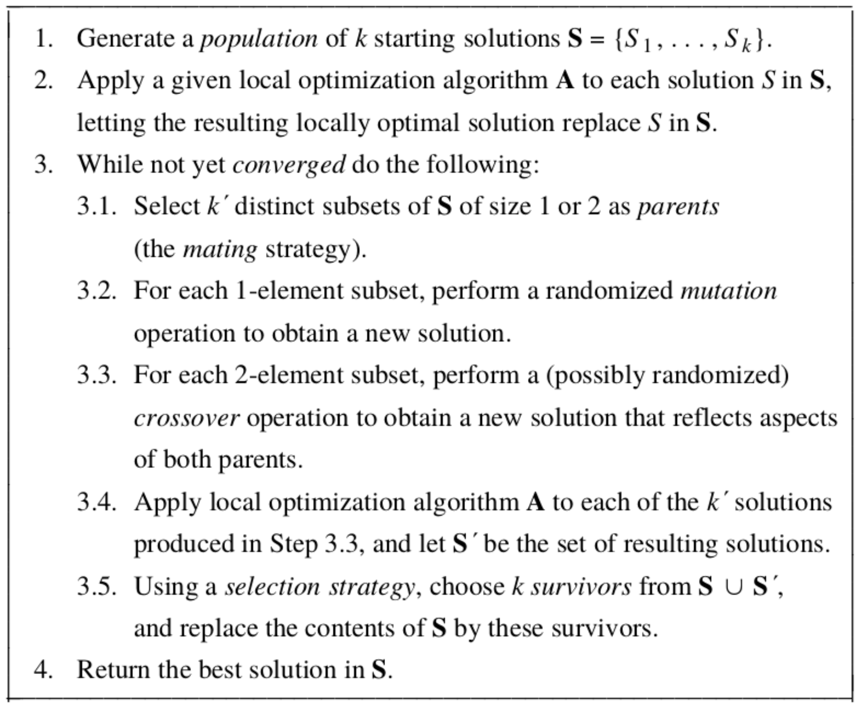
```
1. Generate a starting solution $S$ and set the initial champion solution $S* = S$.
2. Determine a starting temperature $T$.
3. While not yet frozen do the following:
   3.1. While not yet at equilibrium for this temperature, do the following:
        3.1.1. Choose a random neighbor $S'$ of the current solution.
        3.1.2. Set $\Delta = Length(S') - Length(S)$.
        3.1.3. If $\Delta \leq 0$ (downhill move):
                   Set $S = S'$.
                   If $Length(S) < Length(S*)$, set $S* = S$.
        3.1.4  Else (uphill move):
                   Choose a random number $r$ uniformly from $[0,1]$.
                   If $r < e^{-\Delta/T}$, set $S = S'$.
        3.1.5  End ``While not yet at equilibrium'' loop.
   3.2  Lower the temperature $T$.
   3.3  End ``While not yet frozen'' loop.
4. Return $S*$.
```

Figure 4. General schema for a simulated annealing algorithm.

- Simulated Annealing can unlike tabu search make uphill moves at any time
    - Is based on randomization

- It has been shown that dropping the temperature at the rate of $C/\log n$ where n is the number of steps finds the optimum
  - This is much slower than exhaustive search
  - Dropping the temperature more quickly at a rate of i.e. C^n where $C < 1$ is typically used in practice
    * Makes it an approximation algorithm

22.6 Genetic Algorithms and Iterated Lin-Kernighan



```
1. Generate a population of k starting solutions $\mathbf{S} = \{S_1, \ldots, S_k\}$.
2. Apply a given local optimization algorithm A to each solution S in S,
   letting the resulting locally optimal solution replace S in S.
3. While not yet converged do the following:
   3.1. Select $k'$ distinct subsets of S of size 1 or 2 as parents
        (the mating strategy).
   3.2. For each 1-element subset, perform a randomized mutation
        operation to obtain a new solution.
   3.3. For each 2-element subset, perform a (possibly randomized)
        crossover operation to obtain a new solution that reflects aspects
        of both parents.
   3.4. Apply local optimization algorithm A to each of the $k'$ solutions
        produced in Step 3.3, and let $\mathbf{S}'$ be the set of resulting solutions.
   3.5. Using a selection strategy, choose k survivors from $\mathbf{S} \cup \mathbf{S}'$,
        and replace the contents of S by these survivors.
4. Return the best solution in S.
```

Figure 6. General schema for a genetic optimization algorithm.

23 Exam

23.1 The linear programming model and LP algorithms

- LP model

  - Solutions
  - Standard form
  - Matrix form
  - Slack variables
- Simplex algorithm
  - Basis
  - Basic solution
  - Politobe
  - Dictionary
  - Pivot
  - Pivoting rules
  - Two phase Simplex Algorithm
    * Auxiliary Dictionary
    * Starting pivot operation
    * Dual Simplex Algorithm
  - Fundamental theorem of linear programming
  - Cycling
    * Degeneracy
    * Blands rule
    * Lexicographic method
  - Running time
    * Worst case
    * Expected
- Elipsoid algorithm
- Path following method
  - Also good in practise

23.2 LP Duality, Matrix Games, and Integer linear programming.

- Dual
  - Weak duality theorem
  - Strong duality theorem
  - The complementary slackness property and theorem
  - Linear programming in matrix form (with dual)
  - Rules for taking the dual
  - Farkas Lemma
  - Strict complementary slackness property and theorem
- Matrix games
  - Randomized strategy
  - Von Neuman Min-Max Theorem
  - Principle of indifference
  - Yaos principle
- Integer programming
  - Branch and bound
  - TSP

23.3 Network Flows and Minimum Cost Flow algorithms.

- Totally unimodular matrices
  - TUM integrality theorem
  - Cramers rule
  - Node arc adjacency matrix

- Flow networks
  - Minimum cost flow problem
  - Maximum (s,t)-flow problem
  - Max-flow Min-cut Theorem
- Algorithms for minimum cost flow
  - Network simplex algorithm
  - Klein cycle cancelling algorithm

23.4 P, NP, and Cook's theorem.

- Theory of NP completeness
  - Turing machines
  - Models of computation
  - Polynomial church turing thesis
  - Formulation of languages
  - Standin languages
  - Polynomial time maps
- Language complexity
  - P and NP
  - Finding witness
  - Reductions
  - NP hard and NP completeness
- Boolean formulas and circuits
  - SAT
  - Circuit SAT (Prove)
  - Cooks theorem
  - Circuit SAT to SAT

23.5 NP complete problems

- SAT and its relatives
  - SAT
  - Circuitsat
  - KSAT
    - 2SAT
    - 3SAT
  - NAESAT
  - MAX2SAT
- Graph problems
  - 3 coloring
  - MAXCUT
  - Independent SET
    - Clique
    - Vertex cover
  - Hamilton path
  - TSP
- Sets and numbers
  - Tripartite matching
  - Exact cover
  - SET cover
  - SET packing
  - Knapsack
  - Psuedo polynomial time algorithm
  - Bin packing problem

23.6 Approximation Algorithms and Local Search Heuristics.

- Definition of NP + more
- Approximation algorithms
  - Definition of approximation algorithms
  - Min vertex cover
  - TSP
    - Metric TSP
    - Christofides algorithm
    - Problem of general TSP
  - Min weight vertex cover
    - ILP, LP and rounding
  - MAXE3SAT
    - Randomized
  - Min Set Cover
  - Approximation schemes
    - Knapsack
- Local search heuristics
  - Christofides
  - Greedy heuristic
  - Nearest neighbour
  - Clack Wright
  - Natural neighbourhood for TSP
  - k OPT
  - Boosting local search

* Taboo search
* Simulated annealing
* Evolutionary algorithms
* Lin-Kernighan Search