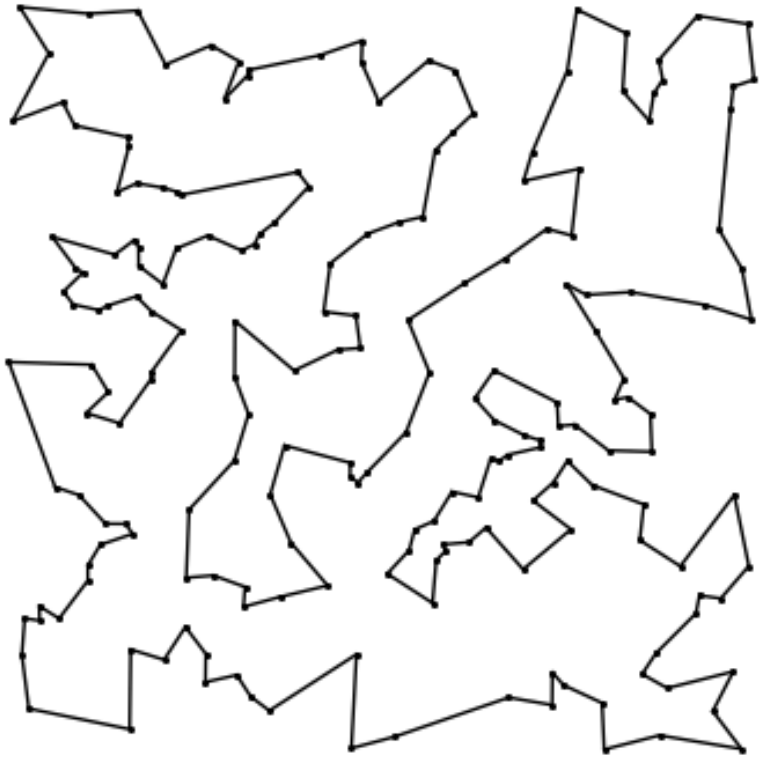
Lecture Notes
Introduction to Integer Programming



2024 Winter Semester

Lan Peng

Management School, Shanghai University

Contents

1 Simplex Method **1**

1.1 Preliminaries . . . 1
1.1.1 Linear Algebra . . . 1
1.1.2 Polyhedral Sets . . . 2
1.1.3 E.P. and B.F.S. . . . 4
1.2 Simplex Method . . . 5
1.2.1 Search Algorithm . . . 5
1.2.2 Find Initial Solution . . . 6
1.2.3 Degeneracy and Cycling . . . 7
1.3 Revised Simplex Method . . . 8
1.3.1 Key to Revised Simplex Method . . . 8
1.3.2 Comparison between Simplex and Revised Simplex . . . 9
1.3.3 Decomposition of B inverse . . . 9
1.4 Duality . . . 11
1.4.1 Dual Formulation . . . 11
1.4.2 Mixed Forms of Duality . . . 11
1.4.3 Primal-Dual Relationships . . . 12
1.4.4 Shadow Price . . . 14
1.4.5 Dual Simplex Method . . . 14

2 Computational Complexity **16**

2.1 Preliminaries . . . 16
2.1.1 How difficult is to solve an integer program? . . . 16
2.1.2 Asymptotic Notation . . . 16
2.2 P v.s. NP . . . 17
2.2.1 P and NP . . . 17
2.2.2 Polynomial Time Reduction . . . 19
2.2.3 NP-Completeness . . . 19
2.3 Algorithm Analysis and Design . . . 20
2.3.1 Three programming paradigms . . . 20
2.3.2 Master Theorem . . . 21
2.4 Some examples . . . 22
2.4.1 Minimum Spanning Tree Problem . . . 22
2.4.2 Knapsack Problem . . . 23
2.4.3 Fibonacci Numbers . . . 24

2.4.4 Multiplications . . . 25

3 Branch and Bound **29**
3.1 Preliminary . . . 29
3.1.1 Why Rounding Can be Bad? . . . 29
3.1.2 Relaxation . . . 30
3.2 Branch and Bound . . . 32
3.2.1 Algorithm overview . . . 32
3.2.2 Idea of Divide and Conquer . . . 32
3.3 Searching, Branching, and Pruning . . . 33
3.3.1 Strategies in B&B . . . 33
3.3.2 Search Strategy: Choose Node to Branch . . . 33
3.3.3 Branching Strategy: Choose Branching Variable . . . 34
3.3.4 Branching Strategy: Create Offspring . . . 35
3.3.5 Pruning Rules . . . 37
3.4 Use Branch-and-Bound to Solve Other Problems . . . 37

4 Special Topic: Formulation Tips **39**
4.1 Linear Program Formulation Tips . . . 39
4.2 Integer Program Formulation Tips . . . 41

5 Polyhedral Analysis and Cutting Plane Method **46**
5.1 Preliminaries . . . 46
5.1.1 Valid Inequalities and Faces . . . 46
5.1.2 Facet . . . 47
5.1.3 Proving Facet . . . 47
5.2 Some Examples . . . 47
5.2.1 Vertices Packing . . . 47
5.2.2 0-1 Knapsack Problem . . . 49
5.3 Generic Cutting Planes . . . 53
5.3.1 General Approach . . . 53
5.3.2 Generic Cutting Planes . . . 53
5.3.3 Gomory Cuts . . . 56
5.4 Branch and Cut . . . 57

6 Graph Algorithm **60**
6.1 Preliminaries . . . 60
6.1.1 Graphs and Subgraphs . . . 60
6.1.2 Walk, Path and Cycle . . . 63
6.1.3 Some warm-up algorithms . . . 65
6.2 Matching . . . 69
6.2.1 Matching . . . 69
6.2.2 Blossom Algorithm . . . 71
6.3 Maximum Flow Problem . . . 73
6.3.1 Maximum Flow Problem . . . 73

6.3.2 Prime and Dual of Maximum Network Flow Problem 75
6.3.3 Maximum Flow Minimum Cut Theorem 76
6.3.4 Ford-Fulkerson Method 77
6.4 Minimum Cost Flow Problem 78
6.4.1 Transshipment Problem 78
6.4.2 Network Simplex Method 78

7 Traveling Salesman Problem **85**
7.1 The Traveling Salesman Problem 85
7.1.1 Dantzig-Fulkerson-Johnson (DFJ) Formulation 85
7.1.2 Miller-Tucker-Zemlin (MTZ) Formulation 86
7.1.3 Flow Based Formulations 86
7.1.4 Shortest Path Formulation 88
7.1.5 Quadratic Formulation (QAP) 89
7.1.6 Numerical Comparisons 90
7.2 The Held and Karp Lower Bound 90
7.2.1 1-Tree 91
7.2.2 Held and Karp Lower Bound and Lagrangian Relaxation 92
7.2.3 Subgradient Descendant Method 93

8 Column Generation **94**
8.1 Fundamental Idea of the Column Generation 94
8.2 The Dantzig-Wolfe Reformulation 96
8.3 Cutting Stock Problem 98
8.3.1 (Restricted) master problem 98
8.3.2 Pricing subproblem 99
8.4 Vehicle Routing Problem with Time Windows 99
8.4.1 (Restricted) Master Problem 100
8.4.2 Pricing subproblem 101
8.5 Early Branching v.s. Branch-and-Price 101

9 Benders Decomposition **103**
9.1 Benders Decomposition 103
9.2 A Uncapacitated Facilities Location Problem 105
9.2.1 Formulation 105
9.2.2 Solution Approach 106
9.3 Pareto-optimality cut 108
9.3.1 Choosing procedure 108
9.3.2 Pareto Optimal Cuts for Facility Location Problem 110

10 Heuristic and Metaheuristic Methods **112**
10.1 Heuristic-Search Procedures 112
10.1.1 Hill-Climbing: An Irrevocable Strategy 112
10.1.2 Uninformed Search 113
10.1.3 Informed Search 115

10.2 Single-State Metaheuristic Methods 116
10.2.1 Simulated Annealing 116
10.2.2 Tabu Search 116
10.2.3 Iterated Local Search 117
10.2.4 Greedy Randomized Adaptive Search Procedures 118
10.3 Population-based Metaheuristic Methods 118
10.3.1 Evolution Strategies 118
10.3.2 Genetic Algorithm 120
10.3.3 Particle Swarm Optimization 120
10.3.4 Ant Colony Optimization 122
10.4 Tuning the evolutionary algorithms 122

Chapter 1

Simplex Method

"One step at a time."

1.1 Preliminaries

1.1.1 Linear Algebra

Linear combination A vector μ is said to be a linear combination of $v^1, v^2, \ldots, v^m$ if

$$\sum_{i=1}^{m} \lambda_i v^i = \mu$$

In addition, μ is a

- conic combination if $\lambda_i \geq 0$
- affine combination if $\sum_{i=1}^{m} \lambda_i = 1$
- convex combination if $\sum_{i=1}^{m} \lambda_i = 1$ and $\lambda_i \geq 0$

Linear independence and affinely independence A collection of vectors $v^1, v^2, \ldots, v^m$ of dimension n is called linearly independent if

$$\sum_{j=1}^{k} \lambda_j v^j = 0 \quad \Rightarrow \quad \lambda_j = 0, \forall j = 1, 2, \ldots, m$$

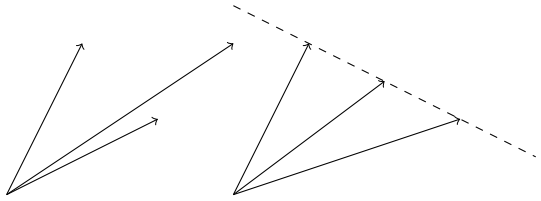
A collection of vectors $v^1, v^2, \ldots, v^m$ of dimension n is called affinely independent if

$$\sum_{j=1}^{k} \lambda_j v^j = 0 \text{ and } \sum_{j=1}^{k} \lambda_j = 0 \Rightarrow \lambda_j = 0, \forall j = 1, 2, \ldots, m$$

All the following statements are equivalent:

- $v^1, v^2, \ldots, v^m$ of dimension n are affinely independent
- $v^2 - v^1, v^3 - v^1, \ldots, v^m - v^1$ of dimension n are linearly independent
- $\begin{bmatrix} v^1 \\ 1 \end{bmatrix}, \begin{bmatrix} v^2 \\ 1 \end{bmatrix}, \ldots, \begin{bmatrix} v^m \\ 1 \end{bmatrix}$ are linearly independent

The difference between linearly independent and affinely independent is indicated in the following figure. For example, this figure is in 2-Dimension space



Linear Dependent
Affine Independent

Linear Dependent
Affine Dependent

Figure 1.1: Linearly Independent / Affinely Independent

Spanning set and basis A collection of vectors $v^1, v^2, \ldots, v^m$ of dimension n is said to span $\mathcal{R}^n$ if any vector in $\mathcal{R}^n$ can be represented as a linear combination of $v^1, v^2, \ldots, v^m$. $v^1, v^2, \ldots, v^m$ is said to form a basis of $\mathcal{R}^n$ if the following conditions holds.

- $v^1, v^2, \ldots, v^m$ span $\mathcal{R}^n$.
- If any of these vectors is deleted, the remaining collection of vector does not span $\mathcal{R}^n$.

Rank of a matrix The span of the columns of a matrix A is called the column space or the range, denoted $range(A)$. The span of the rows of a matrix A is called the row space. The dimension of the column space and row space are equal, which is denoted by $rank(A)$. $rank(A) \leq \min\{m, n\}$, if $rank(A) = \min\{m, n\}$, then A is said to have full rank and A is a basis of $\mathcal{R}^n$.

1.1.2 Polyhedral Sets

Convex sets A set $S \subseteq \mathcal{R}^n$ is convex if $\forall x, y \in S, \lambda \in [0, 1]$, we have $\lambda x + (1 - \lambda)y \in S$
Let $x^1, ..., x^k \in \mathcal{R}^n$ and $\lambda \in \mathcal{R}^k$ be given such that $\lambda^T e = 1$

- The vector $\sum_{i=1}^{k} \lambda_i x^i$ is said to be a convex combination of $x^1, ..., x^k$
- The convex hull of $x^1, ..., x^k$ is the set of all convex combinations of these vectors, denoted $conv(x^1, ..., x^k)$

Figure 1.2: A set is convex iff for any two points in the set, the line segment joining those two points lies entirely in the set

Polyhedral, hyperplanes and half-spaces

- A polyhedron is a set of the form $\{x \in \mathcal{R}^n | Ax \leq b\} = \{x \in \mathcal{R}^n | a^i x \leq b^i, \forall i \in M\}$, where $A \in \mathcal{R}^{m \times n}$ and $b \in \mathcal{R}^m$
- A polyhedron $P \subset \mathcal{R}^n$ is bounded if there exists a constant K such that $|x_i| < K, \forall x \in P, \forall i \in [1, n]$, in this case the polyhedron is call polytope.
- The lower-bound of K is called diagonal denoted by d
- A hyperplane is $\{x \in \mathcal{R}^n | a^T x = b\}$
- A half-space is $\{x \in \mathcal{R}^n | a^T x \leq b\}$

Open, close sets: boundary and interior

- Denote $N_\epsilon = \{y \in \mathcal{R}^n | \|y - x\| < \epsilon\}$ as the neighborhood of $x \in \mathcal{R}^n$
- Given $S \subseteq \mathcal{R}^n$, x belongs to the interior of S, denoted by $int(S)$ if there is $\epsilon > 0$ such that $N_\epsilon(x) \leq S$
- S is said to be an open set iff $S = int(S)$
- x belongs to the boundary ∂S if $\forall \epsilon > 0$, $N_\epsilon(x)$ contains at least one point in S and a point not in S
- $x \in S$ belongs to the closure of S, denoted $cl(s)$ if $\forall \epsilon > 0$, $N_\epsilon(x) \cap S = \emptyset$
- S is called closed iff $S = cl(S)$
- In IP, LP, MIP, etc. we always work with close set. No "<" or ">"

Dimension of polyhedral

- A polyhedron P is dimension k, denoted $dim(P) = k$, if the maximum number of affinely independent points in P is $k + 1$
- A polyhedron $P \subseteq \mathcal{R}^n$ is full-dimensional if $dim(P) = n$

- Let $M = \{1, 2, ..., m\}$, $M^{=} = \{i \in M | a_i x = b_i, \forall x \in P\}$, i.e. the equality set, $M^{\leq} = M \setminus M^{=}$, i.e. the inequality set. Then, Let $(A^{=}, b^{=})$, $(A^{\leq}, b^{\leq})$ be the corresponding rows of (A, b), If $P \subseteq \mathcal{R}^n$, then $dim(P) = n - rank(A^{=}, b^{=})$. To proof a constraint $(A^{=}, b^{=})$ is an equality constraint, we need to proof all point in the closure of P satisfied the constraint, to proof it is not an equality constraint, we need to find one point that is not in the hyperplane.
- $x \in P$ is called an inner point of P if $a^i x < b_i, \forall i \in M^{\leq}$
- $x \in P$ is called an interior point of P if $a^i x < b_i, \forall i \in M$
- Every nonempty polyhedron has at least one inner point
- A polyhedron has an interior point iff P is full-dimensional, i.e., there is no equality constraint

1.1.3 E.P. and B.F.S.

Extreme point A point x in a convex set X is called an extreme point iff x cannot be represented as a strict convex combination of two distinct points of X. In other words, if $x = \lambda x_1 + (1 - \lambda)x_2$, then $x_1 = x_2 = x$.

An alternative definition is as follows. Let the hyperplanes associated with the $(m + n)$ defining half-spaces of X be referred to as defining hyperplanes of X. Furthermore, note that a set of defining hyperplanes are linearly independent if the coefficient matrix associated with this set of equations has full row rank. Then a point x is said to be an extreme point if x lies on some n linearly independent defining hyperplanes of X. In addition, if more than n defining hyperplanes pass through an extreme point, then such extreme point is called a degenerated extreme point.

Basic feasible solution Consider the system $\{A_{m \times n} x = b_m, x \geq 0\}$, suppose $rank(A, b) = rank(A) = m$, we can arrange A and have a partition of A. Let $A = [B, N]$ where B is $m \times m$ invertible matrix, and N is a $m \times (n - m)$ matrix. The solution $x = \begin{bmatrix} x_B \\ x_N \end{bmatrix}$ to the equation $Ax = b$, where

$$x_B = B^{-1} b$$

and

$$x_N = 0$$

is called basic solution of system. If $x_B \geq 0$, it is called basic feasible solution. If $x_B > 0$ it is called non-degenerate basic feasible solution. For $x_B \geq 0$, if some $x_j = 0$, those components are called degenerated basic feasible solution. B is called the basic matrix, N is called nonbasic matrix.

Correspondence between B.F.S and E.P.

Theorem 1.1.1. *x is an E.P. iff x is a B.F.S.*

Proof. ($\Rightarrow$) If x is an extreme point, by definition, there are (at least) n linearly independent defining hyperplanes at x, since $Ax = b$ provides m linearly independent binding hyperplane, there must by some $p = n - m$ additional binding defining hyperplanes from the non-negativity constraints that together with $Ax = b$ provide n linearly independent defining hyperplanes binding at x. Denoting these p additional hyperplanes by $x_N = 0$, we therefore conclude that the system $Ax = b, x_N = 0$ has x as the unique solution. Now, let N represent the columns of the variables x_N in A, and let B be the remaining columns of A with x_B as the associated variables. Since $Ax = b$ can be written as $Bx_B + Nx_N = b$, this means that B is $m \times m$ and invertible, and moreover, $x_B = B^{-1}b \geq 0$, since $x = (x_B, x_N)$ is a feasible solution. Therefore, x is a basic feasible solution.

($\Leftarrow$) If x is a basic feasible solution, by definition, $x = (x_B, x_N)$ where correspondingly $A = (B, N)$ such that $x_B = B^{-1}b \geq 0$ and $x_N = 0$. This means that the n hyperplanes $Ax = b, x_N = 0$ are binding at x and are linearly independent. Thus, x is an extreme point. □

1.2 Simplex Method

1.2.1 Search Algorithm

Improving search algorithm A simplex method is a search algorithm, for each iteration it finds a not-worse solution, which can be represented as

$$x^t = x^{t-1} + \lambda_{t-1} d^{t-1}$$

Where x^t is the solution of the tth iteration, λ_t is the step length of tth iteration, and d^t is the direction of the tth iteration.

Optimality test

$$\begin{aligned} z &= cx \\ &= \begin{bmatrix} c_B & c_N \end{bmatrix} \begin{bmatrix} x_B \\ x_N \end{bmatrix} \\ &= c_B x_B + c_N x_N \end{aligned}$$

$$\begin{aligned} \text{and} \quad & Ax = b \\ \Rightarrow & Bx_B + Nx_N = b, x_B \geq 0, x_N \geq 0 \\ \Rightarrow & x_B = B^{-1}b - B^{-1}Nx_N \\ \Rightarrow & z = c_B B^{-1}b - c_B B^{-1}Nx_N + c_N x_N \end{aligned}$$

for current solution $\hat{x} = \begin{bmatrix} \hat{x_B} \\ 0 \end{bmatrix}$, $\hat{z} = c_B B^{-1} b$, then

$$z - \hat{z} = \begin{bmatrix} 0 & c_N - c_B B^{-1} N \end{bmatrix} \begin{bmatrix} x_B \\ x_N \end{bmatrix}$$

The $c_N - c_B B^{-1} N$ is the reduced cost, for a minimized problem, if $c_N - c_B B^{-1} N > 0$ means $z - \hat{z} \geq 0$, it reaches the optimality because we cannot find a solution less than $\hat{z}$.

Find direction Suppose we choose x_k as a candidate to pivot into Basis

$$x = \begin{bmatrix} B^{-1}b - B^{-1}a_k x_k \\ 0 + e_k x_k \end{bmatrix} = \begin{bmatrix} B^{-1}b \\ 0 \end{bmatrix} + \begin{bmatrix} -B^{-1}a_k \\ e_k \end{bmatrix} x_k$$

In this form, we can see: x is the result after tth iteration, $\begin{bmatrix} B^{-1}b \\ 0 \end{bmatrix}$ is the result after $(t-1)$th iteration. $\begin{bmatrix} -B^{-1}a_k \\ e_k \end{bmatrix}$ is the iteration direction, x_k is the step length. The only requirement of x_k is $r_k < 0$ where $r_k = c_k - z_k$ is reduce cost, which is the kth entry of $c_N - c_B B^{-1} N$. Generally speaking, we usually take $r_k = \min\{c_j - z_j\}$ (which in fact can not guarantee the efficient of the algorithm.)

Find the step length We need to guarantee the non-negativity, so for each iteration, we need to make sure $x \geq 0$. Which means

$$B^{-1}b - B^{-1}a_k x_k \geq 0$$

Denote $B^{-1}b$ as $\bar{b}$, denote $B^{-1}a_k$ as y_k. If $y_k \leq 0$, we can have x_k as large as infinite, which means unboundness. If $y_k > 0$ now we can use the minimum ratio to guarantee non-negativity.

$$\forall i \in B, \bar{b}_i - y_{ik} x_k \geq 0 \Rightarrow x_k = \min_{i \in B}\{\frac{\bar{b}_i}{y_{ik}}, y_{ik} > 0\}.$$

1.2.2 Find Initial Solution

Non-trivial case If some of the constraint is not in $\sum_{i=1}^{n} a_i x_i \leq 0$ form, we cannot add a positive slack variable. In this case, we add an artificial variable other than slack variable.

$$\sum_{i=1}^{n} a_i x_i \geq (or =) 0 \Rightarrow \sum_{j=1}^{n} a_i x_i + x_a = 0$$

Notice that in an optimal solution, $x_a = 0$, otherwise it is not valid.
Artificial variables are only a tool to get the simplex method started. A so-called Two-phase Method is describe in this section.

Two-Phase method

- Phase I: Solve the following program start with a basic feasible solution $x = 0, x_a = b$,

i.e., the artificial variable forms the basis.

$$\begin{aligned} \min \quad & 1x_a \\ \text{s.t.} \quad & Ax + x_a = b \\ & x \geq 0 \\ & x_a \geq 0 \end{aligned}$$

If the optimal $1x_a \neq 0$, infeasible, stop. Otherwise proceed Phase II.

- Phase II: Remove the columns of artificial variables, replace the objective function with the original objective function, proceed to solve simplex method.

Discussion

- Case A: $x_a \neq 0$. Infeasible.
- Case B.1: $x_a = 0$ and all artificial variables are out of the basis. At the end of Phase I, we derive

x_0	x_B	x_N	x_a	RHS
1	0	0	-1	0
0	I	$B^{-1}N$	B^{-1}	$B^{-1}b$

We can discard x_a columns, (or we can leave it because it keeps track of B^{-1}), and then we do the Phase II

z	x_B	x_N	RHS
1	0	$c_B B^{-1}N - c_N$	$c_B B^{-1}b$
0	I	$B^{-1}N$	$B^{-1}b$

- Case B.2: Some artificial variables are in the basis at zero values. This is because of degeneracy. We pivot on those artificial variables, once they leave the basis, eliminate them.

1.2.3 Degeneracy and Cycling

Degeneracy If the basic variable x_B is not strictly > 0, i.e. if some basic variable equals to 0, we call it degenerate.

Cycling In the degenerate case, pivoting by the simplex rule does not always give a strict decrease in the objective function value, because it may have $b_r = 0$. It is possible that the tableau may repeat if we use the simplex rule.Geometrically speaking, it means that at the same point - extreme point - it corresponds to more than one feasible solutions, so when we are pivoting, we stays at the same place. However, in computer algorithm, we rarely care about cycling because the data in computer is not precise, it is very hard to get into cycling.

Cycling prevent

- Lexicographic rule
  - For entering variable, same as simplex rule
  - For leaving variable, if there is a tie, choose the variable with the smallest $\frac{y_{r1}}{y_{rk}}$.
- Bland's rule
  - For entering variable, choose the variable with smallest index where $z_j - c_j \leq 0$
  - For leaving variable, if there is a tie, choose the variable with smallest index.
- Successive ratio rule
  - Select the pivot column as any column k where $z_k - c_k \leq 0$
  - Given k, select the pivot row r as the minimum successive ratio row associated with column k. In other words, for pivot columns where there is no tie in the usual minimum ratio, the successive ratio rule reduces to the simplex rule.

1.3 Revised Simplex Method

1.3.1 Key to Revised Simplex Method

The procedure of Simplex Method is (almost) exactly the same as original simplex method. However, notice that we don't need to use N so for the revised simplex method, we don't calculate any matrix related to N

The original matrix:

z	x_B	x_N	RHS
1	0	$c_B B^{-1} N - c_N$	$c_B B^{-1} b$
0	I	$B^{-1} N$	$B^{-1} b$

The revised matrix:

Basic Inverse	RHS
$w = c_B B^{-1}$	$c_B \bar{b} = c_B B^{-1} b$
B^{-1}	$\bar{b} = B^{-1} b$

For each pivot iteration, calculate $z_j - c_j = wa_j - c_j = c_B B^{-1} a_j - c_j, \forall j \in N$, pivot rules are the same as simplex method, each time find a variable x_k to enter basis

B^{-1}	RHS
w	$c_B \bar{b}$
B^{-1}	$\bar{b}$

x_k
$z_k - c_k$
y_k

Do the minimum ratio rule to find the variable x_r to leave the basis

B^{-1}	RHS	x_k
w	$c_B\bar{b}$	$z_k - c_k$
	$\bar{b}_1$	y_{1k}
	$\bar{b}_2$	y_{2k}
	$\cdots$	$\cdots$
B^{-1}	$\bar{b}_r$	y_{rk}(pivot at here)
	$\cdots$	$\cdots$
	$\bar{b}_m$	y_{mk}

1.3.2 Comparison between Simplex and Revised Simplex

Advantage of revised simplex

- Save storage memory
- Don't need to calculate N (including $B^{-1}N$ and $c_B B^{-1}N$)
- More accurate because round up errors will not be accumulated

Disadvantage of revised simplex Need to calculate wa_j for all $j \in N$ (in fact don't need to calculated it for the variable just left the basis)

Computation complexity

Method	Type	Operations
Simplex	$\times$	$(m+1)(n-m+1)$
	$+$	$m(n-m+1)$
Revised Simplex	$\times$	$(m+1)^2 + m(n-m)$
	$+$	$m(m+1) + m(n-m)$

When to use?

- When $m >> n$, do revise simplex method on the dual problem
- When $m \simeq n$, revise simplex method is not as good as simplex method
- When $m << n$ perfect for revise simplex method.

1.3.3 Decomposition of B inverse

Let $B = \{a_{B_1}, a_{B_2}, ..., a_{B_r}, ..., a_{B_m}\}$ and B^{-1} is known. If a_{B_r} is replaced by a_{B_k}, then B becomes $\bar{B}$. Which means a_{B_r} enters the basis and a_{B_k} leaves the basis.

Then $\bar{B}^{-1}$ can be represent by B^{-1}. Noting that $a_k = By_k$ and $a_{B_i} = Be_i$, then

$$\begin{aligned}\bar{B} &= (a_{B_1}, a_{B_2}, ..., a_{B_{r-1}}, a_k, a_{B_{r+1}}, a_m) \\ &= (Be_1, Be_2, ..., Be_{r-1}, By_k, Be_{r+1}, ..., Be_m) \\ &= BT\end{aligned}$$

where T is

$$T = \begin{bmatrix} 1 & 0 & ... & 0 & y_{1k} & 0 & ... & 0 \\ 0 & 1 & ... & 0 & y_{2k} & 0 & ... & 0 \\ \vdots & \vdots & & \vdots & \vdots & \vdots & & \vdots \\ 0 & 0 & ... & 1 & y_{r-1,k} & 0 & ... & 0 \\ 0 & 0 & ... & 0 & y_{rk} & 0 & ... & 0 \\ 0 & 0 & ... & 0 & y_{r+1,k} & 1 & ... & 0 \\ \vdots & \vdots & & \vdots & \vdots & \vdots & & \vdots \\ 0 & 0 & ... & 0 & y_{mk} & 0 & ... & 1 \end{bmatrix}$$

and

$$E = T^{-1} = \begin{bmatrix} 1 & 0 & ... & 0 & \frac{-y_{1k}}{y_{rk}} & 0 & ... & 0 \\ 0 & 1 & ... & 0 & \frac{-y_{2k}}{y_{rk}} & 0 & ... & 0 \\ \vdots & \vdots & & \vdots & \vdots & \vdots & & \vdots \\ 0 & 0 & ... & 1 & \frac{-y_{r-1,k}}{y_{rk}} & 0 & ... & 0 \\ 0 & 0 & ... & 0 & \frac{1}{y_{rk}} & 0 & ... & 0 \\ 0 & 0 & ... & 0 & \frac{-y_{r+1,k}}{y_{rk}} & 1 & ... & 0 \\ \vdots & \vdots & & \vdots & \vdots & \vdots & & \vdots \\ 0 & 0 & ... & 0 & \frac{-y_{mk}}{y_{rk}} & 0 & ... & 1 \end{bmatrix}$$

For each iteration, i.e. one variable enters the basis and one leaves the basis, $\bar{B}^{-1} = T^{-1}B^{-1} = EB^{-1}$. Given that the first iteration starts from slack variables, the first basis B_1 is I, then we have

$$B_t^{-1} = E_{t-1}E_{t-2}\cdots E_2E_1I$$

Using E in calculation can simplify the product of matrix where

$$\begin{aligned}cE &= c_1, c_2, ..., c_m \begin{bmatrix} 1 & 0 & ... & g_1 & ... & 0 \\ 0 & 1 & ... & g_2 & ... & 0 \\ \vdots & \vdots & & \vdots & & \vdots \\ 0 & 0 & ... & g_m & ... & 1 \end{bmatrix} \\ &= (c_1, c_2, ..., c_{r-1}, cg, c_{r+1}, ..., c_m)\end{aligned}$$

and

$$
\begin{aligned}
Ea &= \begin{bmatrix} 1 & 0 & ... & g_1 & ... & 0 \\ 0 & 1 & ... & g_2 & ... & 0 \\ \vdots & \vdots & & \vdots & & \vdots \\ 0 & 0 & ... & g_m & ... & 1 \end{bmatrix} \begin{bmatrix} a_1 \\ a_2 \\ \vdots \\ a_m \end{bmatrix} \\
&= \begin{bmatrix} a_1 \\ a_2 \\ \vdots \\ a_{r-1} \\ 0 \\ a_{r+1} \\ \vdots \\ a_m \end{bmatrix} + a_r \begin{bmatrix} g_1 \\ g_2 \\ \vdots \\ g_{r-1} \\ g_r \\ g_{r+1} \\ \vdots \\ g_m \end{bmatrix} \\
&= \bar{a} + a_r g
\end{aligned}
$$

Then we can calculate w, y_k and $\bar{b}$

$$
\begin{aligned}
w &= c_B B^{-1} = c_B E_{t-1} E_{t-2} ... E_2 E_1 \\
y_k &= B^{-1} a_k = E_{t-1} E_{t-2} ... E_2 E_1 a_k \\
\bar{b} &= B_{t+1}^{-1} b = E_t E_{t-1} E_{t-2} ... E_2 E_1 b
\end{aligned}
$$

1.4 Duality

1.4.1 Dual Formulation

For any prime problem

$$
\begin{aligned}
\min \quad & cx \\
\text{s.t.} \quad & Ax \geq b \\
& x \geq 0
\end{aligned}
$$

we can have a dual problem

$$
\begin{aligned}
\max \quad & wb \\
\text{s.t.} \quad & wA \leq c \\
& w \geq 0
\end{aligned}
$$

1.4.2 Mixed Forms of Duality

For the following prime problem

$$
\text{P(or D)} \quad \min \quad c_1 x_1 + c_2 x_2 + c_3 x_3
$$

$$
\begin{aligned}
\text{s.t.} \quad & A_{11}x_1 + A_{12}x_2 + A_{13}x_3 \geq b_1 \\
& A_{21}x_1 + A_{22}x_2 + A_{23}x_3 \leq b_2 \\
& A_{31}x_1 + A_{32}x_2 + A_{33}x_3 = b_3 \\
& x_1 \geq 0 \\
& x_2 \leq 0 \\
& x_3 \quad \text{unrestricted}
\end{aligned}
$$

The dual of the problem

$$
\begin{aligned}
\text{D(or P)} \quad \max \quad & w_1b_1 + w_2b_2 + w_3b||3 \\
\text{s.t.} \quad & w_1A_{11} + w_2A_{21} + w_3A_{31} \leq c_1 \\
& w_1A_{12} + w_2A_{22} + w_3A_{32} \geq c_2 \\
& w_1A_{13} + w_2A_{23} + w_3A_{33} = c_3 \\
& w_1 \geq 0 \\
& w_2 \leq 0 \\
& w_3 \quad \text{unrestricted}
\end{aligned}
$$

In sum, the relation between primal and dual problems are listed as following

	Minimization		Maximization	
Var	≥ 0	$\longleftrightarrow$	≤ 0	Cons
	≤ 0	$\longleftrightarrow$	≥ 0	
	Unrestricted	$\longleftrightarrow$	$=$	
Cons	≥ 0	$\longleftrightarrow$	≥ 0	Var
	≤ 0	$\longleftrightarrow$	≤ 0	
	$=$	$\longleftrightarrow$	Unrestricted	

1.4.3 Primal-Dual Relationships

Weak duality property Let x_0 be any feasible solution of a primal minimization problem,

$$Ax_0 \geq b, \quad x_0 \geq 0$$

Let x_0 be any feasible solution of a dual maximization problem,

$$w_0A \leq c, \quad w_0 \geq 0$$

Therefore, we have

$$cx_0 \geq w_0Ax_0 \geq w_0b$$

which is called the weak duality property. This property is for any feasible solution in the primal and dual problem.

Therefore, any feasible solution in the maximization problem gives the lower bound of its dual problem, which is a minimization problem, vice versa. We use this to give the bounds in using linear relaxation to solve IP problem.

Fundamental theorem of duality With regard to the primal and dual LP problems, one and only one of the following can be true.

- Both primal and dual has optimal solution x^* and w^*, where $cx^* = w^*b$
- One problem has an unbounded optimal objective value, the other problem must be infeasible
- Both problems are infeasible.

P		D
Optimal Degeneracy (or Multiple optimal)	$\Rightarrow$	Optimal Multiple optimal (or Degeneracy)
Infeasible	$\Rightarrow$	Unbounded or Infeasible
Unbounded	$\Rightarrow$	Infeasible

Strong duality property From KKT condition, we know that in order to make x^* the optimal solution, the following condition should be met.

- Primal Optimal: $Ax^* \geq b$, $x^* \geq 0$
- Dual Optimal: $w^*A \leq c$, $w^* \geq 0$
- Complementary Slackness:

$$\begin{cases} w^*(Ax^* - b) = 0 \\ (c - w^*A)x^* = 0 \end{cases}$$

The first condition means the primal has an optimal solution, the second condition means the dual has an optimal solution. The third condition means $cx^* = w^*b$, which is also called strong duality property.

Complementary slackness theorem Let $\boldsymbol{x^*}$ and $\boldsymbol{w^*}$ be any feasible solutions, they are optimal iff

$$(c_j - \boldsymbol{w^* a_j})x_j^* = 0, \quad j = 1, ..., n$$
$$w_i^*(\boldsymbol{a^i x^*} - b_i) = 0, \quad i = 1, ..., m$$

In particular

$$\begin{aligned} x_j^* > 0 &\Rightarrow \boldsymbol{w^* a_j} = c_j \\ \boldsymbol{w^* a_j} < c_j &\Rightarrow x_j^* = 0 \\ w_i^* > 0 &\Rightarrow \boldsymbol{a^i x^*} = b_i \\ \boldsymbol{a^i x^*} > b_i &\Rightarrow w_i^* = 0 \end{aligned}$$

It means, if in optimal solution a variable is positive (has to be in the basis), the correspond constraint in the other problem is tight. If the constraint in one problem is not tight, the correspond variable in the other problem is zero.

Use dual to solve the primal in the dual problem, we solved some w which is positive, we can know that the correspond constraint in primal is tight, furthermore we can solve the basic variables from those tight constraints, which becomes equality and we can solve it using Gaussian-Elimination.

1.4.4 Shadow Price

Shadow price under non-degeneracy Let B be an optimal basis for primal problem and the optimal solution x^* is non-degenerated.

$$z = c_B B^{-1} b - \sum_{j \in N} (z_j - c_j) x_j = w^* b - \sum_{j \in N} (z_j - c_j) x_j$$

therefore

$$\frac{\partial z^*}{\partial b_i} = c_B B_i^{-1} = w_i^*$$

w^* is the shadow prices for the right-hand-side vectors. We can also regard it as the incremental cost of producing one more unit of the ith product. Or w^* is the fair price we would pay to have an extra unit of the ith product.

Shadow price under degeneracy For shadow price under degeneracy, the w^* may not be the true shadow price, for it may not be the right basis. In this case, the partial differentiation may not be valid, for component b_i, if $x_i = 0$ and x_i is a basic variable, we can't find the differentiation.

1.4.5 Dual Simplex Method

Occasionally, an initial feasible solution for the Simplex Method might be difficult to acquired, for example, when $\mathbf{0}$ is not an initial solution, or, in some other cases, a basic infeasible solution is known in advance. To solve the LP, we can take advantage of the LP Strong Duality Theorem, and solve the dual problem to optimality instead of the prime problem. Such technique is called the Dual Simplex Method. The details of such method is as follows (for minimization problems)

1. Choose an initial basic solution x_B and corresponding basis matrix B so that $c_B B^{-1} A_j - c_j \geq 0$ for all $j \in J$, where J is the set of non-basic variables.

2. Construct a simplex tableau using this initial solution.

3. If $\bar{b} = B^{-1}b \geq 0$, then an optimal solution has been achieved; STOP. Otherwise, the dual problem is feasible, GOTO STEP 4.

4. Choose a leaving variable $x_{B_i} = \bar{b}_i$ so that $\bar{b}_i < 0$

5. Choose the index of the entering variable x_j using the following minimum ratio test:

$$\frac{z_j - c_j}{\bar{a}_{j_i}} = \min\{\frac{z_k - c_k}{\bar{a}_{k_i}} | k \in J, \bar{a}_{j_i} < 0\}$$

6. If no entering variable can be selected ($\bar{a}_{j_i} \geq 0, \forall k \in K$) then the dual problem is unbounded and the prime problem is infeasible. STOP

7. Using a standard simplex pivot, on element $\bar{a}_{j_i}$, thus causing w_{B_i} to become 0 (and thus feasible) and causing x_j to enter the basis. GOTO STEP 3.

In general, the Dual Simplex Method is (almost) the same as a Simplex Method. The Dual Simplex Method starts with an infeasible basic solution, during each iteration, it maintains dual feasibility and work towards primal feasibility. In Dual Simplex Method, the old Right Hand Side becomes new $z_j - c_j$, and the old $z_j - c_j$ become new Right Hand Side.

Chapter 2

Computational Complexity

```
``import numpy as np''
```

2.1 Preliminaries

2.1.1 How difficult is to solve an integer program?

- Hard!
- What is "hard"? $\Rightarrow$ measurement of difficulties.
- $\Rightarrow$ computational complexity $\Rightarrow$ the art of classifying computational problems according to their resource usage.
- What is "algorithm"? $\Rightarrow$ Given a string s, an algorithm takes the input s and within finite number of steps, returns an output string s'.
- What is "resource usage"? $\Rightarrow$ How much *time/space* it takes for an algorithm to process s?

2.1.2 Asymptotic Notation

O-Notation Let $f(n)$ and $g(n)$ be real functions. We say $f(n) = O(g(n))$ if there exist positive constants n_0 and c such that

$$0 \leq f(n) \leq cg(n), \quad \forall n \geq n_0$$

A common trick that finds $O(g(n))$ for $f(n)$ is to find function $g(n)$ such that

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = C \quad \text{(constant)}$$

The O-Notation is referred as the asymptotic upper bound. The O-Notation is more widely used, since it can provide a clear estimate of the worst case time/space complexity of an algorithm that exclusively considers the critical grow factors.

Ω-Notation Let $f(n)$ and $g(n)$ be real functions. We say $f(n) = \Omega(g(n))$ if there exist positive constants n_0 and c such that

$$0 \leq cg(n) \leq f(n), \quad \forall n \geq n_0$$

The Ω-Notation is referred as the asymptotic lower bound. In addition:

Theorem 2.1.1. $f(n) = O(g(n)) \Leftrightarrow g(n) = \Theta(f(n))$

Θ-Notation Let $f(n)$ and $g(n)$ be real functions. We say $f(n) = \Theta(g(n))$ if there exist positive constants n_0, c_1, and c_2 such that

$$c_1 g(n) \leq f(n) \leq c_2 g(n), \quad \forall n \geq n_0$$

The Θ-Notation is referred as the asymptotic tight bound. In addition:

Theorem 2.1.2. $f(n) = \Theta(g(n)) \Leftrightarrow f(n) = O(g(n))$ *and* $f(n) = \Omega(g(n))$

2.2 P v.s. NP

2.2.1 P and NP

Decision problem A decision problem X is the set of strings on which the output is yes, i.e., $s \in X$ iff the correct output for the input s is 1 (yes).

Example. The optimization version of the knapsack problem: Given a set of items N, each item $i \in N$ having a weight w_i and a value c_i, a capacity b and a threshold value k, find a collection of items $S \subseteq N$ of the maximum value whose total weight is less than or equal to b. The output is a set S.

The decision version of the knapsack problem: Given a set of items N, each item $i \in N$ having a weight w_i and a value c_i, a capacity b and a threshold value k, determine if there exists a collection of items $S \subseteq N$ whose total weight is less than or equal to b and its total value is at least k. The output is a boolean value, True/False.

If we have an algorithm to solve the optimization version of knapsack problem, we can immediately solve the decision version.

Class P

Definition 2.2.1 (polynomial running time)**.** Algorithm A has polynomial running time if there is a polynomial function $p(\dot{})$ such that for every string s, A terminates on s in at most $p(|s|)$ steps.

Definition 2.2.2 (Class P)**.** The complexity class P is the set of decision problems X that can be solved in polynomial time.

That is, there is a known algorithm that provides the solution of any instance of size n in time $n^{O(1)}$.

Example. The following problems are known as in class P:

- Shortest path problem
- Maximum flow problem
- Spanning tree problem
- Linear programming (but not the simplex method)
- Assignment problem

Class NP

Definition 2.2.3 (certifier, certificate)**.** B is an efficient certifier for a problem X if

- B is a polynomial-time algorithm that takes two input strings s and t
- There is a polynomial function p such that $s \in X$ if and only if there is a string t such that $|t| \leq p(|s|)$ and $B(s,t) = 1$

The string t such that $B(s,t) = 1$ is called a certificate.

Example. Q: Given a graph $G = (V, E)$ with a Hamiltonian cycle, how can one convince another there exists a Hamiltonian cycle?

A: One can send a string (certificate) to another, the other will run an algorithm (certifier) to check if the string represents a Hamiltonian cycle.

Definition 2.2.4 (class NP)**.** The complexity class NP is the set of all problems for which there exists an polynomial-time certifier.

P v.s. NP

- Is $P = NP$? This is the most famous and fundamental open problem in computer science.
- Most people believe $P \neq NP$.
- If $P = NP$, that means if one can check a solution in polynomial time, one can solve it in polynomial time.
- We are sure that $P \subseteq NP$

2.2.2 Polynomial Time Reduction

Polynomial-time reducible Given a black box algorithm A that solves a problem X, if any instance of a problem Y can be solved using a polynomial number of standard computational steps, plus a polynomial number of calls to A, then we say Y is polynomial-time reducible to X, denoted as $Y \leq_P X$.

Suppose $Y \leq_P X$:

- If X can be solved in polynomial time, then Y can be solved in polynomial time.
- If Y cannot be solved in polynomial time, then X cannot be solved in polynomial time.

X is at least as hard as Y.

2.2.3 NP-Completeness

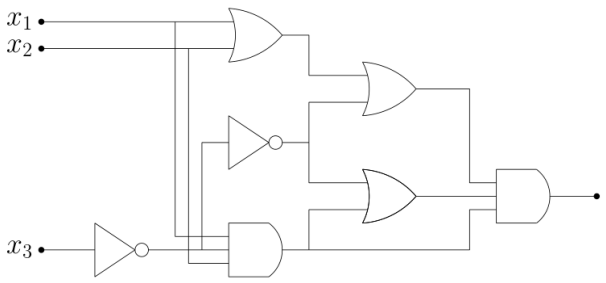
NP-Completeness A problem X is called NP-complete if

- $X \in NP$, and
- $Y \leq_P X$, for every $Y \in NP$

If any NP-complete problem can be solved in polynomial time, then $P = NP$. Unless $P = NP$, a NP-complete problem cannot be solved in polynomial time.

Circuit-Sat The Circuit Satisfiability problem is the first NP-complete problem.

Theorem 2.2.1 (Cook's Theorem)**.** *SAT is NP-Complete*



- key fact: any algorithm that takes n bits as input and outputs 0/1 with running time $T(n)$ can be converted into a circuit of size $p(T(n))$ for some polynomial function $p(\cdot)$.
- Then, we can show that any problem $Y \in NP$ can be reduced to Circuit-Sat

Reductions of NP-Complete problems We will show a SAT $\leq_P$ 3-SAT

Definition 2.2.5 (3-CNF)**.** 3-CNF is a special case of formula:

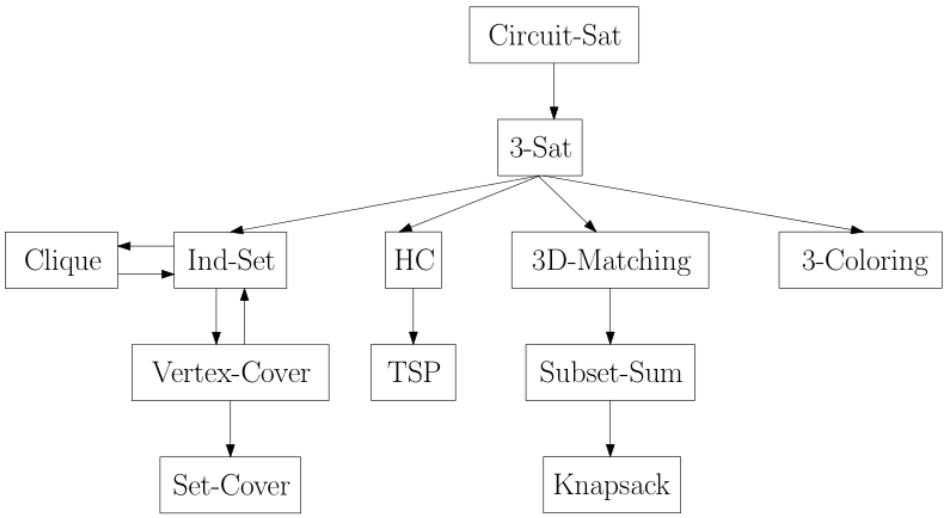
- Boolean variables: $x_1, x_2, \cdots, x_n$
- Literals: x_i or $\bar{x_i}$
- Clause: disjunction ("or") of at most 3 literals
- 3-CNF formula: conjunctino ("and") of clauses.

Example. This is a 3-CNF: $(x_1 \vee x_2 \vee x_3) \wedge (x_2 \vee \bar{x_3} \vee x_4)$

To satisfy a 3-CNF, we need to satisfy all clauses. To satisfy a clause, we need to satisfy at least 1 literal. Associate every wire with a new variable, the circuit will be equivalent to a formula. Each formula can be transformed into a 3-CNF.

The SAT is satisfiable iff the 3-CNF is satifiable, and the size of the 3-CNF formula is polynomial in the size of the circuit. Thus SAT $\leq_P$ 3-SAT.

For other problems, here is a polynomial-reducible relation graph for reference.



2.3 Algorithm Analysis and Design

2.3.1 Three programming paradigms

Greedy Algorithm

- Make a greedy choice
- At each step, make an irrevocable decision using a "reasonable" strategy
- Prove that the greedy choice is safe

- Show that the remaining task after applying the strategy is to solve a/many **smaller instance**(s) of the same problem
- Usually for optimization problems.

Dynamic Programming

- Break up a problem into many **overlapping** sub-problems
- Build solutions for larger and larger sub-problems
- Use a table/dictionary to store solutions for sub-problems for reuse

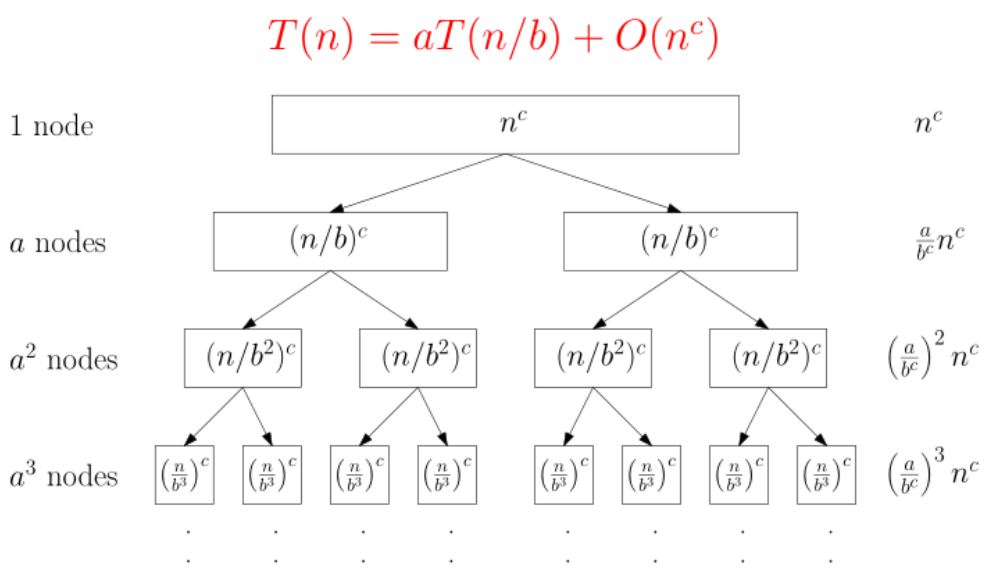
Divide and Concur

- Break a problem into many **independent** sub-problems
- Solve each sub-problem separately
- Combine solutions for sub-problems to form a solution for the original one
- Usually used to design more efficient algorithms

2.3.2 Master Theorem

Theorem 2.3.1. *$T(n) = aT(n/b) + O(n^c)$, where $a \geq 1, b > 1, c \geq 0$ are constants. Then,*

$$T(n) = \begin{cases} O(n^{\log_b a}) & \text{if } c < \log_b a \\ O(n^c \log n) & \text{if } c = \log_b a \\ O(n^c) & \text{if } c > \log_b a \end{cases}$$



- $c < \log_b a$: bottom-level dominates, $(\frac{a}{b^c})^{\log_b n} n^c = O(n^{\log_b a})$
- $c = \log_b a$: all levels have same time, $n^c \log_b n = O(n^c \log n)$
- $c > \log_b a$: top-level dominates, $O(n^c)$

2.4 Some examples

2.4.1 Minimum Spanning Tree Problem

Basic Concepts

Example. A company wants to build a communication network for their offices. For a link between office v and office w, there is a cost c_{vw}. If an office is connected to another office, then they are connected to with all its neighbors. Company wants to minimize the cost of communication networks.

Definition 2.4.1 (Minimum spanning tree problem)**.** Given a connected graph graph G, and a cost $C_e, \forall e \in E$, find a minimum cost spanning tree of G

Kruskal's Algorithm Let's start with a greedy algorithm

Algorithm 1 MST-Greedy(G, w)

```
1: F = ∅
2: Sort edges in E in non-decreasing order of weight w
3: for e = (u, v) do
4:     if u and v are not connected by a path of edge in F then
5:         F = F ∪ {(u, v)}
6: return F
```

The Kruskal's algorithm is as follows

Algorithm 2 Kruskal's Algorithm

```
F ← ∅
S ← {{v} : v ∈ V}
Sort edges in E in non-decreasing order of weight w
for e = (u, v) do
    S_u ← the set in S containing u
    S_v ← the set in S containing v
    if S_u ≠ S_v then
        F ← F ∪ {(u, v)}
        S ← S \ {S_u} \ {S_v} ∪ {S_u ∪ S_v}
return F
```

Prim's Algorithm Let's start with another greedy algorithm

Algorithm 3 MST-Greedy(G, w)

$S \leftarrow \{s\}$, where s is arbitary vertex in V
$F \leftarrow \emptyset$
while $S \neq V$ **do**
  $(u, v) \leftarrow$ the lightest edge between S and $V \setminus S$, where $u \in S$ and $v \in V \setminus S$
  $S \leftarrow S \cup \{v\}$
  $F \leftarrow F \cup \{(u, v)\}$

The Prim's algorithm is as follows

Algorithm 4 Prim's Algorithm

$s \leftarrow$ arbitary vertex in G
$S \leftarrow \emptyset$, $d(s) \leftarrow 0$ and $d(v) \leftarrow \infty$ for every $v \in V \setminus \{s\}$
while $S \neq V$ **do**
  $u \leftarrow$ vertex in $V \setminus S$ with minimum $d(u)$
  $S \leftarrow S \cup \{u\}$
  for Each $v \in V \setminus S$ such that $(u, v) \in E$ **do**
    if $w(u, v) < d(v)$ **then**
      $d(v) \leftarrow w(u, v)$
      $\pi(v) \leftarrow u$
return $\{(u, \pi(u)) | u \in V \setminus \{s\}\}$

2.4.2 Knapsack Problem

Integer program model

$$\max \quad \sum_{i \in S} v_i \tag{2.1}$$

$$\text{s.t.} \quad \sum_{i \in S} w_i \leq W \tag{2.2}$$

$$w_i \geq 0, \forall i \in S \tag{2.3}$$

Solve knapsack problem by dynamic programming

- Let $opt[i, W']$ be the optimum value when budget is W' and the items are $\{1, 2, 3, \cdots, i\}$.
- If $i = 0, opt[i, W'] = 0$ for every $W' = 0, 1, \cdots, W$. Then
- $$opt[i, W'] = \begin{cases} 0, & i = 0 \\ opt[i-1, W'], & i > 0, w_i > W' \\ \max\{opt[i-1, W'], opt[i-1, W' - w_i] + v_i\}, & i > 0, w_i \leq W' \end{cases}$$

2.4.3 Fibonacci Numbers

Fibonacci sequence

$$0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, \cdots \tag{2.4}$$

Naive algorithm Use recursion (in a stupid way)

Algorithm 5 Fib(n)

```
1: if n = 0 then
2:     return 0
3: if n = 1 then
4:     return 1
5: return Fib(n − 1) + Fib(n − 2)
```

The runtime is exponential $O(2^n)$. This is stupid.

Reasonable algorithm Use dynamic programming

Algorithm 6 Fib(n)

```
1: F(0) ← 0
2: F(1) ← 1
3: for i ← 2 to n do
4:     F(i) ← F(i − 1) + F(i − 2)
5: return F(n)
```

The runtime is $O(n)$.

Even better algorithm Notice that

$$\begin{bmatrix} F_n \\ F_{n-1} \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} F_{n-1} \\ F_{n-2} \end{bmatrix}$$
$$\begin{bmatrix} F_n \\ F_{n-1} \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^2 \begin{bmatrix} F_{n-2} \\ F_{n-3} \end{bmatrix}$$
$$\cdots$$
$$\begin{bmatrix} F_n \\ F_{n-1} \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^{n-1} \begin{bmatrix} F_1 \\ F_0 \end{bmatrix}$$

First define power(n)

Algorithm 7 power(n)

1: **if** $n = 0$ **then**
2: **return** $\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$
3: $R \leftarrow \text{power}(n/2.\text{floor}())$
4: $R \leftarrow R \times R$
5: **if** n is odd number **then**
6: $R \leftarrow R \times \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}$
7: **return** R

Then the Fibonacci sequence is calculated by

Algorithm 8 Fib(n)

1: **if** $n = 0$ **then**
2: **return** 0
3: $M \leftarrow \text{power}(n-1)$
4: **return** M[1][1]

The time complexity is $T(n) = T(n/2) + O(1) = O(\log n)$

2.4.4 Multiplications

Polynomial Multiplication Given two polynomials of degree $n-1$, we need to find the product of two polynomials.

Example. Let $A = (4, -5, 2, 3)$ and $B = (-5, 6, -3, 2)$ as input. The expected output $C = (-20, 49, -52, 20, 2, -5, 6)$, since

$$\begin{aligned} & (3x^3 + 2x^2 - 5x + 4) \times (2x^3 - 3x^2 + 6x - 5) \\ =& 6x^6 - 9x^5 + 18x^4 - 15x^3 + 4x^5 - 6x^4 + 12x^3 - 10x^2 \\ & - 10x^4 + 15x^3 - 30x^2 + 25x + 8x^3 - 12x^2 + 24x - 20 \\ =& 6x^6 - 5x^5 + 2x^4 + 20x^3 - 52x^2 + 49x - 20 \end{aligned}$$

A naive algorithm for polynomial multiplication is as follows

Algorithm 9 naivePolyMultiply(A, B, n)

1: Let $C[k] = 0$ for every $k = 0, 1, \cdots, 2n-2$
2: **for** $i \leftarrow 0$ to $n-1$ **do**
3: **for** $j \leftarrow 0$ to $n-1$ **do**
4: $C[i+j] \leftarrow C[i+j] + A[i] \times B[j]$
5: **return** C

Obviously, the running time is $O(n^2)$. Now we try to use divide-and-conquer to improve. For a polynomial $p(x)$ with degree of $n-1$, denote

$$p(x) = p_H(x)x^{\frac{n}{2}} + p_L(x)$$

then, $p_H(x)$ and $p_L(x)$ are polynomials of degree $n/2-1$. Consider

$$pq = (p_H x^{\frac{n}{2}} + p_L)(q_H x^{\frac{n}{2}} + q_L) = p_H q_H x^n + (p_H q_L + p_L q_H)x^{\frac{n}{2}} + p_L q_L$$

The recurrence time is

$$T(n) = 4T(n/2) + O(n) = O(n^2)$$

still not good... Consider

$$\begin{aligned} pq &= (p_H x^{\frac{n}{2}} + p_L)(q_H x^{\frac{n}{2}} + q_L) \\ &= p_H q_H x^n + ((p_H + p_L)(q_H + q_L) - p_H q_H - p_L q_L)x^{\frac{n}{2}} + p_L q_L \end{aligned}$$

The new recurrence time is

$$T(n) = 3T(n/2) + O(n) = O(n^{\log_2 3}) = O(n^{1.585})$$

Better!!! The algorithm is given as follows

Algorithm 10 polyMultiply(A, B, n)

```
1: A_L ← A[0..n/2 − 1], A_H ← A[n/2..n − 1]
2: B_L ← B[0..n/2 − 1], B_H ← B[n/2..n − 1]
3: C_L ← polyMultiply(A_L, B_L, n/2)
4: C_H ← polyMultiply(A_H, B_H, n/2)
5: C_M ← polyMultiply(A_L + A_H, B_L + B_H, n/2)
6: Initialize C ← [0..0] (2n − 1) 0s
7: for i ← 0 to n − 2 do
8:     C[i] ← C[i] + C_L[i]
9:     C[i + n] ← C[i + n] + C_H[i]
10:    C[i + n/2] ← C[i + n/2] + C_M[i] − C_L[i] − C_H[i]
11: return C
```

Notice that the name of Algorithm 10 appears inside itself (line 3, 4, 5). This is referred as *recursion*. The key to recursion functions is to break complex problems into smaller instances of the **same** problems, so we need to "think beyond timescape".

Matrix multiplication Given two matrices $\mathbf{A} \in \mathbb{R}^{n\times}$ and $\mathbf{B} \in \mathbb{R}^{n\times}$, we need to find the production, $\mathbf{C} = \mathbf{AB} \in \mathbb{R}^{n\times}$.

A naive algorithm for matrix multiplication is as follows

Algorithm 11 naiveMatMultiply(A, B, n)

```
1: for i ← 1 to n do
2:     for j ← 1 to n do
3:         C[i, j] ← 0
4:         for k ← 1 to n do
5:             C[i, j] ← C[i, j] + A[i, k] × B[k, j]
6: return C
```

The running time is $O(n^3)$. Try divided-and-conquer. Let

$$\mathbf{A} = \begin{bmatrix} \mathbf{A}_{11} & \mathbf{A}_{12} \\ \mathbf{A}_{21} & \mathbf{A}_{22} \end{bmatrix}, \quad \mathbf{B} = \begin{bmatrix} \mathbf{B}_{11} & \mathbf{B}_{12} \\ \mathbf{B}_{21} & \mathbf{B}_{22} \end{bmatrix}$$

each sub-matrix is $n/2 \times n/2$. Then

$$\mathbf{C} = \begin{bmatrix} \mathbf{A}_{11}\mathbf{B}_{11} + \mathbf{A}_{12}\mathbf{B}_{21} & \mathbf{A}_{11}\mathbf{B}_{12} + \mathbf{A}_{12}\mathbf{B}_{22} \\ \mathbf{A}_{21}\mathbf{B}_{11} + \mathbf{A}_{22}\mathbf{B}_{21} & \mathbf{A}_{21}\mathbf{B}_{12} + \mathbf{A}_{22}\mathbf{B}_{22} \end{bmatrix}$$

Still not good enough because $T(n) = 8T(n/2) + O(n)$.

Strassen's Algorithm In 1969, the first matrix multiplication algorithm with time complexity lower than $O(n^3)$ was introduced by Volker Strassen, the famous Strassen's Algorithm.

For $\mathbf{A}, \mathbf{B}, \mathbf{C}$, let

$$\begin{aligned}
\mathbf{M}_1 &= (\mathbf{A}_{11} + \mathbf{A}_{22})(\mathbf{B}_{11} + \mathbf{B}_{22}) \\
\mathbf{M}_2 &= (\mathbf{A}_{21} + \mathbf{A}_{22})\mathbf{B}_{11} \\
\mathbf{M}_3 &= \mathbf{A}_{11}(\mathbf{B}_{12} - \mathbf{B}_{22}) \\
\mathbf{M}_4 &= \mathbf{A}_{22}(\mathbf{B}_{21} - \mathbf{B}_{11}) \\
\mathbf{M}_5 &= (\mathbf{A}_{11} + \mathbf{A}_{12})\mathbf{B}_{22} \\
\mathbf{M}_6 &= (\mathbf{A}_{21} - \mathbf{A}_{11})(\mathbf{B}_{11} + \mathbf{B}_{12}) \\
\mathbf{M}_7 &= (\mathbf{A}_{12} - \mathbf{A}_{22})(\mathbf{B}_{21} + \mathbf{B}_{22})
\end{aligned}$$

Then,

$$\begin{aligned}
\mathbf{C}_{11} &= \mathbf{M}_1 + \mathbf{M}_4 - \mathbf{M}_5 + \mathbf{M}_7 \\
\mathbf{C}_{12} &= \mathbf{M}_3 + \mathbf{M}_5 \\
\mathbf{C}_{21} &= \mathbf{M}_2 + \mathbf{M}_4 \\
\mathbf{C}_{22} &= \mathbf{M}_1 - \mathbf{M}_2 + \mathbf{M}_3 + \mathbf{M}_6
\end{aligned}$$

In total, $T(n) = 7T(n/2) + O(n) = O(n^{\log_2 7})$.

Why is this important? Matrix multiplication has been widely used in machine learning such as calculating convolutions. For $n > 300$, the Strassen's algorithm will run significantly faster than the naive algorithm.

There are some algorithms even faster than Strassen, e.g., the Coppersmith-Winograd matrix multiplication algorithm.

Chapter 3

Branch and Bound

"The time to relax is when you don't have time for it."

3.1 Preliminary

3.1.1 Why Rounding Can be Bad?

IP example Rounding can be bad because the optimal of IP can be far away from optimal of LP. For example,

$$\begin{aligned} \max \quad & z = x_1 + 0.64x_2 \\ \text{s.t.} \quad & 50x_1 + 31x_2 \leq 250 \\ & 3x_1 - 2x_2 \geq -4 \\ & x_1, x_2 \geq 0 \quad \text{(for LP)} \\ & x_1, x_2 \in Z^+ \quad \text{(for IP)} \end{aligned}$$

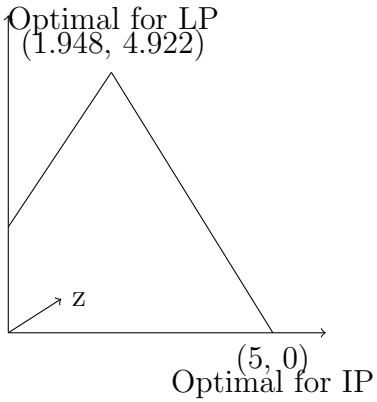


Figure 3.1: Optimal solution for LP / IP

QAP example Rounding can make the LP useless. For example, for QAP problem, the IP model is

$$\begin{aligned}
\min \quad & z = \sum_{i\in D}\sum_{s\in O} c_{is}x_{is} + \sum_{i\in D}\sum_{j\in D}\sum_{s\in O}\sum_{t\in O} w_{ij}^{st}y_{ij}^{st} \\
\text{s.t.} \quad & \sum_{i\in D} x_{is} = 1, \quad s \in O \\
& \sum_{s\in O} x_{is} = 1, \quad i \in D \\
& x_{is} \in \{0,1\}, \quad i \in D, s \in O \\
& y_{ij}^{st} \geq x_{is} + x_{jt} - 1, \quad i \in D, j \in D, s \in O, t \in O \\
& y_{ij}^{st} \geq 0, \quad i \in D, j \in D, s \in O, t \in O \\
& y_{ij}^{st} \leq x_{is}, \quad i \in D, j \in D, s \in O, t \in O \\
& y_{ij}^{st} \leq x_{jt}, \quad i \in D, j \in D, s \in O, t \in O
\end{aligned}$$

We can get the optimal solution for LP supposing $\forall i, s \quad x_{is} \in [0,1]$

$$\begin{aligned}
& x_{is} = \frac{1}{|D|}, \quad i \in D, s \in O \\
& y_{ij}^{st} = 0, \quad i \in D, j \in D, s \in O, t \in O
\end{aligned}$$

3.1.2 Relaxation

Local Optimal v.s. Global Optimal Let

$$\begin{aligned}
Z_s &= \min\{f(x) : x \in S\} \\
Z_t &= \min\{f(x) : x \in T\} \\
S &\subset T
\end{aligned}$$

then

$$Z_t \leq Z_s$$

Notice that if $x_T^* \in S$ then $x_S^* = x_T^*$, to generalized it, We have

$$\begin{cases} x_T^* \in \arg\min\{f(x) : x \in T\} \\ x_T^* \in S \end{cases} \\ \Rightarrow x_T^* \in \arg\min\{f(x) : x \in S\}$$

Especially for IP, we can take the LP relaxation as T and the original feasible region of IP as S, therefore, if we find an optimal solution from LP relaxation T which is also a feasible solution of S, then it is the optimal solution for IP (S)

LP Relaxation To perform the LP relaxation, we expand the feasible region

$$x \in \{0,1\} \to x \in [0,1]$$
$$y \in Z^+ \to y \geq 0$$

If we have $Z_{LP}(s) = conv(s)$ then

$$LP(s) : x \in R_+^n : Ax \leq b$$

The closer $LP(s)$ is to $conv(s)$ the better. Interestingly, we only need to know the convex in the direction of c.

For IP problem

$$\begin{aligned} Z_{IP} \quad \max \quad & z = cx \\ \text{s.t.} & Ax \leq b \\ & x \in Z^n \end{aligned}$$

In feasible region $S = \{x \in Z^n, Ax \leq b\}$, the optimal solution $Z_{IP} = \max\{cx : x \in S\}$. Denote $conv(S)$ as the convex hull of S then

$$Z_{IP}(S) = Z_{IP}(conv(S))$$

Relation Between LP Relaxation and IP Let

$$\begin{aligned} Z_{IP} &= \max_{x \in S} cx, \quad \text{where } s \text{ is a set of integer solutions} \\ Z_{LP} &= \max cx, \quad \text{the LP relaxation of IP} \end{aligned}$$

then

$$\begin{aligned} Z_{IP} &= \max_{1 \leq i \leq k} \{\max_{x \in S_i} cx\} \\ \text{iff} \quad S &= \bigcup_{1 \leq i \leq k} S_i \end{aligned}$$

Notice that S_i don't need to be disjointed.

LP feasibility and IP(or MIP) feasibility Solve the LP relaxation, one of the following things can happen

- LP relaxation is infeasible $\to$ MIP is infeasible
- LP relaxation is unbounded $\to$ MIP is infeasible or unbounded
- LP relaxation has optimal solution $\hat{x}$ and $\hat{x}$ are integer ($\hat{x} \in S$), $\to Z_{IP} = Z_{LP}$

- LP relaxation has optimal solution $\hat{x}$ and $\hat{x}$ are not integer ($\hat{x} \notin S$), now defines a new upper bound, $Z_{LP} \geq Z_{IP}$

If the first three happens, stop, if the fourth happens, we branch and recursively solve the sub-problems.

3.2 Branch and Bound

3.2.1 Algorithm overview

Algorithm 12 Branch and Bound (For maximization problem)

find a feasible solution as the initial Lower bound L
put the original LP relaxation in candidate list S
while $S \neq \emptyset$ **do**
    select a problem $\hat{S}$ from S
    solve the LP relaxation of $\hat{S}$ to obtain $u(\hat{S})$
    if LP is infeasible **then**
        $\hat{S}$ pruned by infeasibility
    else if LP is unbounded **then**
        return Unbounded
    else if LP $u(\hat{S}) \leq L$ **then**
        $\hat{S}$ pruned by bound
    else if LP $u(\hat{S}) > L$ **then**
        if $\hat{x} \in S$ **then**
            $u(\hat{S})$ becomes new L, $L = u(\hat{S})$
        else if $\hat{x} \notin S$ **then**
            branch and add the new sub-problems to S
            if LP $u(\hat{S})$ is at current best upper bound **then**
                set $U = u(\hat{S})$
if Lower bound exists **then**
    find the optimal at L
else
    Infeasible

3.2.2 Idea of Divide and Conquer

For each iteration, divide the feasible region of LP into two parts (and an infeasible part), solve the LP in those parts.

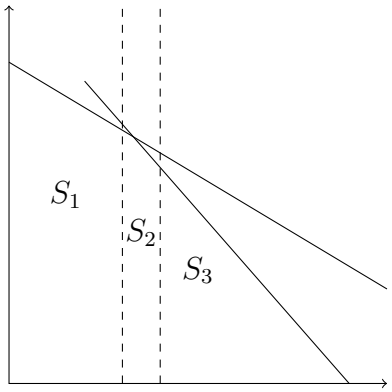


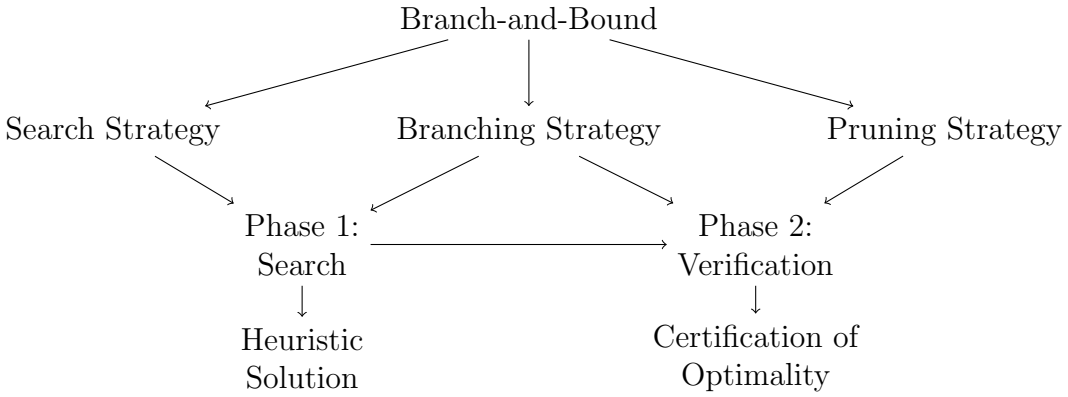
Figure 3.2: Divide and Conquer

In this iteration, the original feasible region have been partition into three parts, where S_2 is infeasible for IP because there is not integer point in it. We continue the iteration for S_1 and S_2. Each partition is suppose to give a new upper bound / lower bound and reduce the infeasible space.

If the temp optimal integer in S_1 is larger than the LP relaxation in S_3, we can cut S_3.

3.3 Searching, Branching, and Pruning

3.3.1 Strategies in B&B



3.3.2 Search Strategy: Choose Node to Branch

Depth-first search It can be implemented by maintaining the list of unexplored subproblems L as a stack. The algorithm removes the top item from the stack to choose the next subproblem to explore, and when children are generated as a result of branching, they are inserted on the top of L. Thus, the next subproblem that is explored is the most recently generated subproblem.

- DFS does not need to store the entire list of unexplored subproblems (only stores the path from the root of T to the current subproblem)

- If no unexplored children remain, the algorithm backtracks to the closest ancestor node with unexplored children.
- Use the LP relaxations as lower bounds. The LP solver can often reuse information from the parent LP solution as a starting point for the child LP solution
- Naive implementations of DFS do not use any information about problem structure or bounds
- Search tree is extremely unbalanced

Breadth-first search BFS explores all subproblems that are at a fixed distance from the root before exploring any deeper subproblems.

- Finding an optimal solution that is closest to the root of the tree, thus operating well on unbalanced search trees
- Generally unable to exploit pruning rules that compare against the current incumbent solution $\Rightarrow$ high memory
- Best bound search $\Rightarrow$ improve dual bound
- Best estimate search $\Rightarrow$ improve primal bound

Best-first search, A* search makes use of a heuristic measure-of-best function to compute every subproblem by a admissible index μ.

3.3.3 Branching Strategy: Choose Branching Variable

The Most Violated Integrality constraint Pick the j of which $x_j - \lfloor \hat{x_j} \rfloor$ is the closest to 0.5

Strong Branching Select a few candidates (K), create sub-problems for each of these variables, run a few dual simplex iterations to see the improved bounds, select the variable with the best bounds, and then selects for branching the variable that induces the most change in the objective.

- Might fix variable, when one side is infeasible
- Detect infeasibility, when both side are infeasible
- Can be speed up by limit the number of iterations, or stop when improvement is found.
- Domain propagation

Pseudo-cost Branching Pseudo-cost attempts to predict the per-unit change in the objective function for each candidate branching variable, based on past experience in the tree. The basic idea of this strategy is to keep track for each variable x_i the change in the objective function when this variable was previously chosen as the variable to branch on. By branching on a variable that is expected to produce a significant change in the objective function, it is more likely that the generated subproblems can be pruned. One difficulty with pseudocost branching is that no information about past branching behavior is available at the beginning of the algorithm, so the pseudocosts for each variable must be initialized in some way.

For variable $x_j \in K$,

$$\begin{cases} P_j^+, & \text{bound reduction if rounded up} \\ P_j^-, & \text{bound reduction if rounded down} \end{cases}$$

define $f_j = x_j - \lfloor x_j \rfloor$

$$\begin{cases} D_j^+ = P_j^+(1 - f_j) \\ D_j^- = P_j^- f_j \end{cases}$$

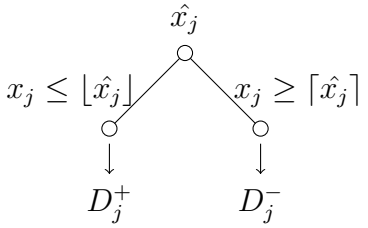


Figure 3.3: Strong Branching

For those variables in K find the

- $\max\{\min\{D_j^+, D_j^-\}\}$, or
- $\max\{\max\{D_j^+, D_j^-\}\}$, or
- $\max\{\dfrac{D_j^+ + D_j^-}{2}\}$, or
- $\max\{\alpha_1 \min\{D_j^+, D_j^-\} + \alpha_2 \max\{D_j^+, D_j^-\}\}$

to branch.

3.3.4 Branching Strategy: Create Offspring

Traditional Branching For $\hat{x} \notin S$, $\exists j \in N$ such that

$$\hat{x_j} - \lfloor \hat{x_j} \rfloor > 0$$

Create two sub-problems

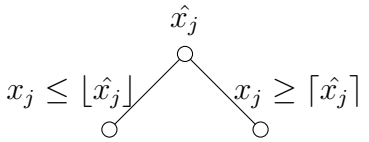


Figure 3.4: Traditional Branching

Constraint Branching Use parallel constraints to branch, e.g.

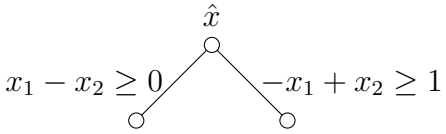


Figure 3.5: Constraint Branching

Special Ordered Sets of type 1 (SOS1 or S1) are a set of variables, at most one of which can take a non-zero value, all others being at 0.

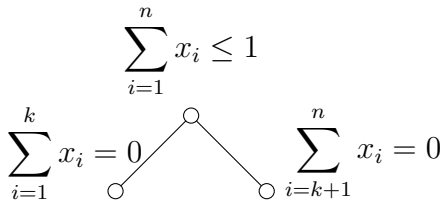


Figure 3.6: SOS1 Branching

Special Ordered Sets of type 2 (SOS2 or S2) : an ordered set of non-negative variables, of which at most two can be non-zero, and if two are non-zero these must be consecutive in their ordering.

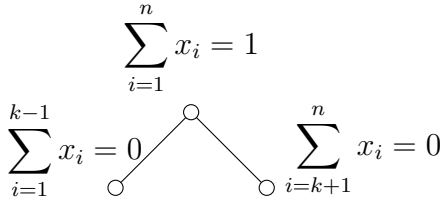


Figure 3.7: SOS2 Branching

Ryan-Foster Ryan-Foster is for Set covering problem. The typical model is

$$\begin{aligned} \min \quad & \sum_{i \in C} x_i \\ \text{s.t.} \quad & \sum_{i \in C} a_{ij} x_i \geq 1, \quad \forall j \in U \\ & x_i \in \{0,1\}, \quad \forall i \in C \end{aligned}$$

For any fractional solution, there are at least two elements (i, j) so that i and j are both partially covered by the same set S, but there is another set T that only covers i

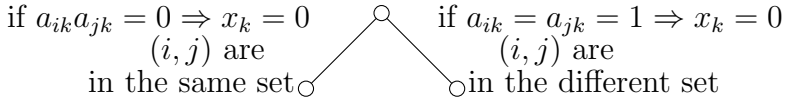


Figure 3.8: Ryan Foster Branching

3.3.5 Pruning Rules

Lower bound The most common way to prune is to produce a lower bound on the objective function value at each subproblem, and use this to prune subproblems whose lower bound is no better than the incumbent's solution value. Lower bounds are computed by relaxing various aspects of the problem.

- Many different lower bound can be computed
- Attempt to prune using the easy lower bounds first, then move on to more complex
- One of the "easy lower bound" can be LP relaxation
- Another "easy lower bound" can be Lagrangian relaxation for IP.

Dominance relations In contrast to lower bounding rules, dominance relations allow subproblems to be pruned if they can be shown to be dominated by some other subproblem. In other words, if subproblem S_1 dominates subproblem S_2 , this means that for any solution that is a descendant of S_2 , there exists a complete solution descending from S_1 that is at least as good.

3.4 Use Branch-and-Bound to Solve Other Problems

Close-enough Traveling Salesman Problem Given a set of vertices $V = \{0, 1, \cdots, n\}$ in a 2-dimensional space with coordinates $(\bar{x}, \bar{y}), i = 0, 1, \cdots, n$. Each vertex i is in the center of a convex region bounded by a circle D_i with radius r_i. Assume that $(\bar{x}_i, \bar{y}_i) \neq (\bar{x}_j, \bar{y}_j), \forall i, j \in V, i \neq j$. The problem lies in determining the value of the coordinates of the hitting points $(\bar{x}_i, \bar{y}_i) \in \mathbb{R}^2$ and a sequence $L = (k_0, k_1, \cdots, k_n), k_i \in V$ representing the order in which the vertices are covered, such that the tour over the hitting points forms a Hamiltonian cycle of minimum length and $(\bar{x}_i, \bar{y}_i) \in D_i, i \in V$.

Coutinho et al. 2016 Each branch-and-bound node is associated with an optimal partial tour that needs to visit only a given subset of vertices in a particular order. At the root node, the algorithm chooses three vertices to generate an initial sequence of nodes that need to be visited. Because there are only three vertices involved in this sequence and costs are symmetric, their order will not affect the solution. Therefore, the problem of finding an optimal tour that visits these three vertices in the given order is a valid relaxation of the main problem regardless of the choice of the initial sequence. If the associated solution is feasible, i.e., all customers are covered, then this solution is optimal and the problem is solved. Otherwise, for this root node, the algorithm branches into three subproblems; in each of them, a vertex that does not belong to the tour is inserted in a different position. A node is discarded if its cost is greater than or equal to the best known upper bound or if its associated solution is feasible. Otherwise, a branching is performed over this node using the same rationale applied in the root node. Note that the number of child nodes (subproblems) for every node in the tree is equal to the number of vertices in the partial sequence of the parent node. (**?**)

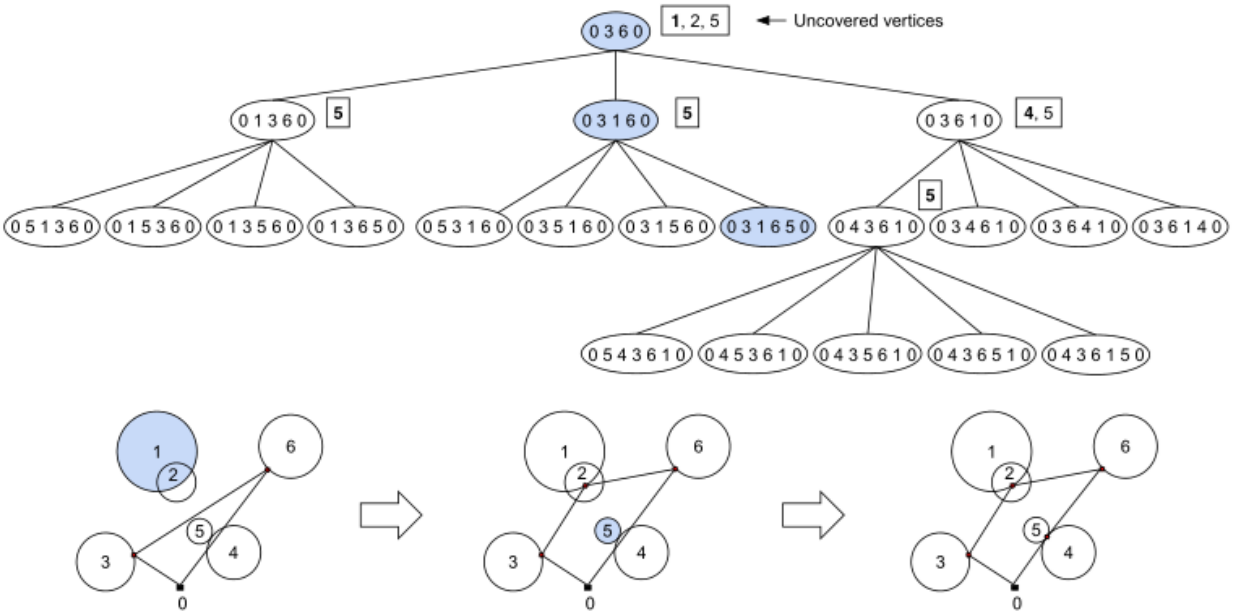


Figure 3.9: An example of the B&B algorithm for an instance with 7 vertices

Chapter 4

Special Topic: Formulation Tips

"Trick or Treat."

4.1 Linear Program Formulation Tips

Absolute Value Consider the following model statement:

$$\begin{aligned} \min \quad & \sum_{j \in J} c_j |x_j|, \quad c_j > 0 \\ \text{s.t.} \quad & \sum_{j \in J} a_{ij} x_j \gtreqless b_i, \quad \forall i \in I \\ & x_j \quad \text{unrestricted}, \quad \forall j \in J \end{aligned}$$

Equivalent model:

$$\begin{aligned} \min \quad & \sum_{j \in J} c_j (x_j^+ + x_j^-), \quad c_j > 0 \\ \text{s.t.} \quad & \sum_{j \in J} a_{ij} (x_j^+ - x_j^-) \gtreqless b_i, \quad \forall i \in I \\ & x_j^+, x_j^- \geq 0, \quad \forall j \in J \end{aligned}$$

Minimax Objective Consider the following model statement:

$$\begin{aligned} \min \quad & \max_{k \in K} \sum_{j \in J} c_{kj} x_j \\ \text{s.t.} \quad & \sum_{j \in J} a_{ij} x_j \gtreqless b_i, \quad \forall i \in I \\ & x_j \geq 0, \quad \forall j \in J \end{aligned}$$

Equivalent model:

$$\begin{aligned}
\min \quad & z \\
\text{s.t.} \quad & \sum_{j \in J} a_{ij} x_j \gtreqless b_i, \quad \forall i \in I \\
& \sum_{j \in J} c_{kj} x_j \leq z, \quad \forall k \in K \\
& x_j \geq 0, \quad \forall j \in J
\end{aligned}$$

Fractional Objective Consider the following model statement:

$$\begin{aligned}
\min \quad & \frac{\sum_{j \in J} c_j x_j + \alpha}{\sum_{j \in J} d_j x_j + \beta} \\
\text{s.t.} \quad & \sum_{j \in J} a_{ij} x_j \gtreqless b_i, \quad \forall i \in I \\
& x_j \geq 0, \quad \forall j \in J
\end{aligned}$$

Equivalent model:

$$\begin{aligned}
\min \quad & \sum_{j \in J} c_j x_j t + \alpha t \\
\text{s.t.} \quad & \sum_{j \in J} a_{ij} x_j \gtreqless b_i, \quad \forall i \in I \\
& \sum_{j \in J} d_j x_j t + \beta t = 1 \\
& t > 0 \\
& x_j \geq 0, \quad \forall j \in J \\
& (t = \frac{1}{\sum_{j \in J} d_j x_j + \beta})
\end{aligned}$$

Range Constraint Consider the following model statement:

$$\begin{aligned}
\min \quad & \sum_{j \in J} c_j x_j \\
\text{s.t.} \quad & d_i \leq \sum_{j \in J} a_{ij} x_j \leq e_i, \quad \forall i \in I \\
& x_j \geq 0, \quad \forall j \in J
\end{aligned}$$

Equivalent model:

$$\begin{aligned}
\min \quad & \sum_{j \in J} c_j x_j, \quad c_j > 0
\end{aligned}$$

$$\begin{aligned} \text{s.t.} \quad & u_i + \sum_{j \in J} a_{ij} x_j = e_i, \quad \forall i \in I \\ & x_j \geq 0, \quad \forall j \in J \\ & 0 \leq u_i \leq e_i - d_i, \quad \forall i \in I \end{aligned}$$

4.2 Integer Program Formulation Tips

A Variable Taking Discontinuous Values In algebraic notation:

$$x = 0, \quad \text{or} \quad l \leq x \leq u$$

Equivalent model:

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

where

$$y = \begin{cases} 0, & \text{if } x = 0 \\ 1, & \text{if } l \leq x \leq u \end{cases}$$

Fixed Costs In algebraic notation:

$$C(x) = \begin{cases} 0 & \text{for } x = 0 \\ k + cx & \text{for } x > 0 \end{cases}$$

Equivalent model:

$$\begin{aligned} & C^*(x, y) = ky + cx \\ & x \leq My \\ & x \geq 0 \\ & y \in \{0, 1\} \end{aligned}$$

where

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

Either-or Constraints In algebraic notation:

$$\sum_{j \in J} a_{1j} x_j \leq b_1 \text{ or } \sum_{j \in J} a_{2j} x_j \leq b_2$$

Equivalent model:

$$\sum_{j\in J} a_{1j}x_j \le b_1 + M_1 y$$
$$\sum_{j\in J} a_{2j}x_j \le b_2 + M_1(1-y)$$
$$y \in \{0,1\}$$

where

$$y = \begin{cases} 0, & \text{if } \sum_{j\in J} a_{1j}x_j \le b_1 \\ 1, & \text{if } \sum_{j\in J} a_{2j}x_j \le b_2 \end{cases}$$

Notice that the sign before M is determined by the inequality $\ge$ or $\le$, if it is "$\ge$", use "$-$", if it "$\le$", use "$+$".

Conditional Constraints If constraint A is satisfied, then constraint B must also be satisfied

$$\text{If} \quad \sum_{j\in J} a_{1j}x_j \le b_1 \text{ then } \sum_{j\in J} a_{2j}x_j \le b_2$$

The key part is to find the opposite of the first condition. We are using $A \Rightarrow B \Leftrightarrow \neg B \Rightarrow \neg A$ Therefore it is equivalent to

$$\sum_{j\in J} a_{1j}x_j > b_1 \text{ or } \sum_{j\in J} a_{2j}x_j \le b_2$$

Furthermore, it is equivalent to

$$\sum_{j\in J} a_{1j}x_j \ge b_1 + \epsilon \text{ or } \sum_{j\in J} a_{2j}x_j \le b_2$$

Where ϵ is a very small positive number.

Equivalent model:

$$\sum_{j\in J} a_{1j}x_j \ge b_1 + \epsilon - M_2 y$$
$$\sum_{j\in J} a_{2j}x_j \le b_2 + M_2(1-y)$$
$$y \in \{0,1\}$$

Special Ordered Sets SOS1 Description: Out of a set of yes-no decisions, at most one decision variable can be yes.

$$x_1 = 1, x_2 = x_3 = \cdots = x_n = 0$$
$$\text{or}$$
$$x_2 = 1, x_1 = x_3 = \cdots = x_n = 0$$
$$\text{or ...}$$

Equivalent model:

$$\sum_i x_i = 1, \quad i \in N$$

SOS2 Description: Out of a set of binary variables, at most two variables can be nonzero. In addition, the two variables must be adjacent to each other in a fixed order list.

Equivalent model: If $x_1, x_2, ..., x_n$ is a SOS2, then

$$\sum_{i=1}^{n} x_i \leq 2$$
$$x_i + x_j \leq 1, \forall i \in \{1, 2, ..., n\}, j \in \{i+2, i+3, ..., n\}$$
$$x_i \in \{0, 1\}$$

Piecewise Linear Formulations The objective function is a sequence of line segments, e.g. $y = f(x)$, consists $k-1$ linear segments going through k given points $(x_1, y_1), (x_2, y_2), ..., (x_k, y_k)$. Denote

$$d_i = \begin{cases} 1, & x \in (x_i, x_{i+1}) \\ 0, & \text{otherwise} \end{cases}$$

Then the objective function is

$$\sum_{i \in \{1,2,...,k-1\}} y = d_i f_i(x)$$

Equivalent model: Given that objective function as a piecewise linear formulation, we can have these constraints

$$\sum_{i \in \{1,2,...,k-1\}} d_i = 1$$
$$d_i \in \{0, 1\}, i \in \{1, 2, ..., k-1\}$$
$$x = \sum_{i \in \{1,2,...,k\}} w_i x_i$$
$$y = \sum_{i \in \{1,2,...,k\}} w_i y_i$$

$$w_1 \leq d_1$$
$$w_i \leq d_{i-1} + di, i \in \{2, 3, ..., k-1\}$$
$$w_k \leq d_{k-1}$$

In this case, $w_i \in SOS2$ (second definition)

Conditional Binary Variables Choose at most n binary variable to be 1 out of $x_1, x_2, ...x_m, m \geq n$
If $n = 1$ then it is SOS1.

Equivalent model:

$$\sum_{k \in \{1,2,...,m\}} x_k \leq n$$

Choose exactly n binary variable to be 1 out of $x_1, x_2, ...x_m, m \geq n$

Equivalent model:

$$\sum_{k \in \{1,2,...,m\}} x_k = n$$

Choose x_j only if $x_k = 1$

Equivalent model:

$$x_j = x_k$$

"and" condition, iff $x_1, x_2, ..., x_m = 1$ then $y = 1$

Equivalent model:

$$y \leq x_i, i \in \{1, 2, ..., m\}$$
$$y \geq \sum_{i \in \{1,2,...,m\}} x_i - (m-1)$$

Elimination of Products of Variables For variables x_1 and x_2,

$$y = x_1 x_2$$

Equivalent model: If x_1, x_2 are binary, it is the same as "and" condition of binary variables.
If x_1 is binary, while x_2 is continuous and $0 \leq x_2 \leq u$, then

$$y \leq ux_1$$
$$y \leq x_2$$
$$y \geq x_2 - u(1 - x_1)$$
$$y \geq 0$$

If both x_1 and x_2 are continuous, it is non-linear, we can use Piecewise linear formulation to simulate.

Chapter 5

Polyhedral Analysis and Cutting Plane Method

"Life is a Crystal."

5.1 Preliminaries

5.1.1 Valid Inequalities and Faces

The inequality denoted by (π, π_o) is called a valid inequality for P if $\pi x \leq \pi_0, \forall x \in P$. Note that (π, π_0) is a valid inequality iff P lies in the half-space $\{x \in \mathbb{R}^n | Ax \leq b\}$

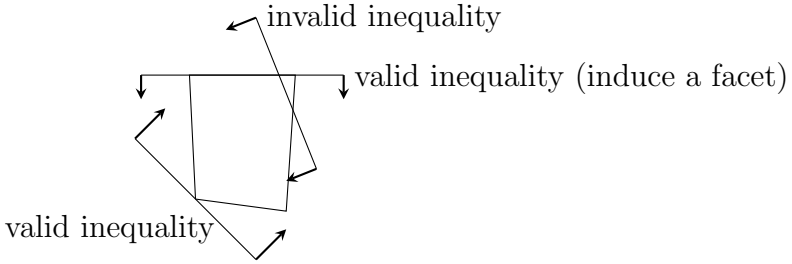


Figure 5.1: Example of valid/invalid inequality

- If (π, π_0) is a valid inequality for P and $F = \{x \in P | \pi x = x_0\}$, F is called a facet of P and we say that (π, π_0) represents or defines F
- A face is said to be proper if $F \neq \emptyset$ and $F \neq P$
- The face represented by (π, π_0) is nonempty iff $\max\{\pi x | x \in P\} = \pi_0$
- If the face F is nonempty, we say it supports P
- Let P be a polyhedron with equality set $M^=$. If

$$F = \{x \in P | \pi^T x = \pi_0\}$$

is not empty, then F is a polyhedron. Let

$$M^{=} \subseteq M_F^{=}, M_F^{\leq} = M \setminus M_F^{=}$$

then

$$F = \{x | a_i^T x = b_i, \forall i \in M_F^{=}, a_i^T x \leq b_i, \forall i \in M_f^{\leq}\}$$

5.1.2 Facet

- A face F is said to be a facet of P if $dim(F) = dim(P) - 1$
- Facets are all we need to describe polyhedral
- If F is a facet of P, then in any description of P, there exists some inequality representing F
- Every inequality that represents a face that is not a facet is unnecessary in the description of P
- Every full-dimensional polyhedron P has a unique (up to scalar multiplication) representation that consists of one inequality representing each facet of P
- If $dim(P) = n - k$ with $k > 0$, then P is described by a maximal set of linearly independent rows of $(A^{=}, b^{=})$, as well as one inequality representing each facet of P

5.1.3 Proving Facet

To prove an inequality $\sum_i a_i x_i \leq b_i$ is facet inducing for a D dimensional polyhedral, we need to prove there are D affinely independent vectors in $\sum_i a_i x_i = b_i$

5.2 Some Examples

5.2.1 Vertices Packing

Vertices Packing Given a graph $G = (V, E)$, with $|V| = n$. A vertices packing solution is that no two neighboring vertices can be chosen at the same time.

$$PACK(G) = \{x \in \mathbb{B}^n | x_i + x_j \leq 1, \forall (i, j) \in E\}$$

The PACK of this graph is

$$PACK = conv\left(\begin{pmatrix}0\\0\\0\\0\end{pmatrix}, \begin{pmatrix}1\\0\\0\\0\end{pmatrix}, \begin{pmatrix}0\\1\\0\\0\end{pmatrix}, \begin{pmatrix}0\\0\\1\\0\end{pmatrix}, \begin{pmatrix}0\\0\\0\\1\end{pmatrix}, \begin{pmatrix}0\\1\\1\\0\end{pmatrix}\right)$$

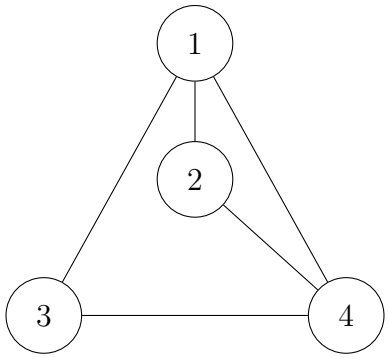


Figure 5.2: Example of vertices packing problem

The dimension of PACK, i.e. $dim(PACK(G))$ is (full-dimensional)

$$dim(PACK(G)) = |V|$$

To prove that $dim(PACK(G)) = |V|$, we need to find $|V| + 1$ affinely independent vectors.

Proof.

$$rank\left(\begin{bmatrix} 0 & I_{|V|} \\ 1 & 1 \end{bmatrix}\right) = |V| + 1$$

□

Therefore, in PACK, $rank(A^{=}, b^{=}) = 0$

Inequalities and Facets of conv(VP) - Nonnegative Constraints $x_i \geq 0$ induce facets.

Proof.

$$rank\left(\begin{bmatrix} 0 & 0 \\ 0 & I_{|V|} \end{bmatrix}\right) = |V| + 1$$

□

Inequalities and Facets of conv(VP) - Neighborhood Constraints $x_i + x_j \leq 1$ is a valid constraint, but it **DOES NOT** always induce facet.

Inequalities and Facets of conv(VP) - Odd Hole Consider a graph $G = (V, E)$, the covering problem is

$$\sum_{e \in \delta(i)} x_e \leq 1, i \in V, x_e \in \{0, 1\}, e \in E$$

For $T \subset V$, denote $\delta(i)$ as all edges induce to $i \in V$, denote $E(T) \subset E$ as all the edges linked between $(i, j), i \in T, j \in T$, therefore we have

$$\sum_{i \in T} \sum_{e \in \delta(i)} x_e \leq |T|$$

For edges linking $i \in T, j \in T$, count them twice, for edges linking $i \in T, j \notin T$, count them once.We can have a new constraint

$$2 \sum_{e \in E(T)} x_e + \sum_{e \in \delta(V \setminus T, T)} x_e \leq |T|$$

Perform the Gomory Cut, the following constraint is a valid:

$$\sum_{e \in E(T)} x_e \leq \lfloor \frac{|T|}{2} \rfloor$$

H is an odd hole if it contains circle of k nodes, such that k is odd and there is no cords. e.g. {1, 2, 5, 6, 3}. Then, the following inequality is valid,

$$\sum_{i \in H} x_i \leq \frac{|H| - 1}{2}$$

Odd Hole inequality **DOES NOT** always induce facets.

Inequalities and Facets of conv(VP) - Maximum Clique A **clique** is a subset of a graph that in the clique every two vertices linked with each other (complete sub-graph). A **maximum clique** is a clique that any other vertice can not form a clique with all the points in this clique.

C is a maximum clique, then the following inequality is valid and induce a facet,

$$\sum_{i \in C} x_i \leq 1$$

Proof. First, if $C = V$

$$rank\left([I]\right) = |C| = |V|$$

Second, if C is a subset of V, for each vertice in $V \setminus C$, there should be at least one vertice in C that is not linked with it. Therefore for each vertice in C we can find a packing. □

5.2.2 0-1 Knapsack Problem

0-1 Knapsack Problem Formulation Consider the knapsack set KNAP

$$conv(KNAP) = conv(\{x \in \mathbb{B}^n | \sum_{j \in N} a_j x_j \leq b\})$$

in where

- $N = \{1, 2, ..., n\}$
- With out lost of generality, assume that $a_j > 0, \forall j \in N$ and $a_j < b, \forall j \in N$

Valid Inequalities for a Relaxation For $P = \{x \in \mathbb{B}^n | Ax \leq b\}$, each row can be regard as a Knapsack problem, i.e. for row i

$$P_i = \{x \in \mathbb{B}^n | a_i^T x \leq b_i\}$$

is a relaxation of P, therefore,

$$P \subseteq P_i, \forall i = 1, 2, ..., m$$

$$P \subseteq \cap_{i=1}^m P_i$$

So any inequality valid for a relaxation of an IP is also valid for IP itself.

Cover and Extended Cover A set $C \subseteq N$ is a cover if $\sum_{j \in C} a_j > b$, a cover C is minimal cover if

$$C \subseteq N | \sum_{j \in C} a_j > b, \sum_{j \in C \setminus k} a_j \leq b, \forall k \in C$$

For a cover C, we can have the cover inequality

$$\sum_{j \in C} x_j \leq |C| - 1$$

The inequality is trivial considering the pigeonhole principle. $C \subseteq N$ is a minimal cover, then $E(C)$ is defined as following:

$$E(C) = C \cup \{j \in N | a_j \geq a_i, \forall i \in C\}$$

is called an extended cover. Then we have,

$$\sum_{i \in E(C)} x_i \leq |C| - 1 \text{ dominates } \sum_{i \in C} x_i \leq |C| - 1$$

and

$$\sum_{i \in E(C)} x_i \leq |C| - 1 \text{ dominates } \sum_{i \in E(C)} x_i \leq |E(C)| - 1$$

Hereby we need to prove that $\sum_{i \in E(C)} x_i \leq |C| - 1$ is valid, by contradiction.

Proof. Suppose $x^R \in KNAP$, R is a feasible solution, Where

$$x_j^R = \begin{cases} 1, & \text{if } j \in R \\ 0, & \text{otherwise} \end{cases}$$

Then

$$\sum_{j \in E(C)} x_j^R \geq |C| \Rightarrow |R \cap E(C)| \geq |C|$$

therefore

$$\sum_{j \in R} a_j \geq \sum_{j \in R \cap E(C)} a_j \geq \sum_{j \in C} a_j > b$$

which means R is a cover, it is contradict to $\sum_{j \in E(C)} x_j^R \geq |C|$ so $x^R \notin KNAP$ □

Dimension of KNAP $conv(KNAP)$ is full dimension, i.e. $dim(conv(KNAP)) = n$.

Proof. $0, e_j, \forall j \in N$ are $n+1$ affinely independent points in $conv(KNAP)$ □

Inequalities and Facets of conv(KNAP) - Lower Bound and Upper Bound Constraints

- $x_k \geq 0$ is a facet of $conv(KNAP)$

  Proof. $\mathbf{0}, \mathbf{e}_j, \forall j \in N \setminus k$ are n linearly independent vectors that satisfied $x_k = 0$ (thus they are affinely independent), i.e.,

  $$rank(\begin{bmatrix} \mathbf{e}_1 & \mathbf{e}_2 & \cdots & \mathbf{e}_{k-1} & \mathbf{0} & \mathbf{e}_{k+1} & \cdots & \mathbf{e}_n \end{bmatrix}) = |N|$$

  □

- $x_k \leq 1$ is a facet iff $a_j + a_k \leq b, \forall j \in N \setminus k$

  Proof. $\mathbf{e}_k, \mathbf{e}_j + \mathbf{e}_k, \forall j \in N \setminus k$ are n affinely independent points that satisfied $x_k = 1$, i.e.,

  $$rank(\begin{bmatrix} \mathbf{e}_{1,k} & \mathbf{e}_{2,k} & \cdots & \mathbf{e}_{k-1,k} & \mathbf{e}_k & \mathbf{e}_{k,k+1} & \cdots & \mathbf{e}_{k,n} \end{bmatrix}) = |N|$$

  □

Inequalities and Facets of conv(KNAP) - Extended Cover Order the variables so that $a_1 \geq a_2 \geq \cdots \geq a_n$, therefore $a_1 = a_{max}$. Let C be a cover with $\{j_1, j_2, \ldots, j_r\}$ where $j_1 < j_2 < \cdots < j_r$ so that $a_{j_1} \geq a_{j_2} \geq \cdots \geq a_{j_r}$, then

$$\sum_{j \in E(C)} x_j \leq |C| - 1$$

is a facet of $conv(KNAP)$ if

- $C = N$

  Proof.

$$rank(\begin{bmatrix} 0 & 1 & 1 & \cdots & 1 \\ 1 & 0 & 1 & \cdots & 1 \\ 1 & 1 & 0 & \cdots & 1 \\ \vdots & \vdots & \vdots & & \vdots \\ 1 & 1 & 1 & \cdots & 0 \end{bmatrix}) = |N|$$

  □

- $E(C) = N$ and $\sum_{j \in C \setminus \{j_1, j_2\}} a_j + a_{max} \leq b$

  Proof. (j_1, j_2 are two heaviest elements in C)

$$rank(\left[\begin{array}{cccc|ccc} 0 & 1 & \cdots & 1 & 0 & \cdots & 0 \\ 1 & 0 & \cdots & 1 & 0 & \cdots & 0 \\ \vdots & \vdots & & \vdots & \vdots & & \vdots \\ 1 & 1 & \cdots & 0 & 1 & \cdots & 1 \\ \hline 0 & 0 & \cdots & 0 & 1 & \cdots & 0 \\ \vdots & \vdots & & \vdots & \vdots & & \vdots \\ 0 & 0 & \cdots & 0 & 0 & \cdots & 1 \end{array}\right]) = |N|$$

  In $|C|$ rows, select $|C| - 1$ decision variables to be 1. In $|N| - |C|$ rows, select all decision variables in C except for the two with largest weight to be 1 and select one decision variable outside C to be 1. □

- $C = E(C)$ and $\sum_{j \in C \setminus j_1} a_j + a_p \leq b$, where $p = \max\{j | j \in N \setminus E(C)\}$

  Proof. (j_1 is the heaviest element in C, k is the lightest element outside extended cover)

$$rank(\left[\begin{array}{cccc|ccc} 0 & 1 & \cdots & 1 & 0 & \cdots & 0 \\ 1 & 0 & \cdots & 1 & 1 & \cdots & 1 \\ \vdots & \vdots & & \vdots & \vdots & & \vdots \\ 1 & 1 & \cdots & 0 & 1 & \cdots & 1 \\ \hline 0 & 0 & \cdots & 0 & 1 & \cdots & 0 \\ \vdots & \vdots & & \vdots & \vdots & & \vdots \\ 0 & 0 & \cdots & 0 & 0 & \cdots & 1 \end{array}\right]) = |N|$$

  In $|C|$ rows, select $|C| - 1$ decision variables to be 1. In $|N| - |C|$ rows, select all decision variables in C except for the one with largest weight to be 1 and select one decision variable outside C to be 1. □

- $C \subset E(C) \subset N$ and $\sum_{j \in C \setminus \{j_1, j_2\}} a_j + a_{max} \leq b$ and $\sum_{j \in C \setminus j_1} a_j + a_p \leq b$

  Proof.

$$rank\left(\left[\begin{array}{ccccc|ccc|ccc} 0 & 1 & \cdots & 1 & & 0 & \cdots & 0 & 0 & \cdots & 0 \\ 1 & 0 & \cdots & 1 & & 0 & \cdots & 0 & 1 & \cdots & 1 \\ \vdots & \vdots & & \vdots & & \vdots & & \vdots & \vdots & & \vdots \\ 1 & 1 & \cdots & 0 & & 1 & \cdots & 1 & 1 & \cdots & 1 \\ \hline 0 & 0 & \cdots & 0 & & 1 & \cdots & 0 & 0 & \cdots & 0 \\ \vdots & \vdots & & \vdots & & \vdots & & \vdots & \vdots & & \vdots \\ 0 & 0 & \cdots & 0 & & 0 & \cdots & 1 & 0 & \cdots & 0 \\ \hline 0 & 0 & \cdots & 0 & & 0 & \cdots & 0 & 1 & \cdots & 0 \\ \vdots & \vdots & & \vdots & & \vdots & & \vdots & \vdots & & \vdots \\ 0 & 0 & \cdots & 0 & & 0 & \cdots & 0 & 0 & \cdots & 1 \end{array}\right]\right) = |N|$$

  In the first $|C|$ rows, select $|C| - 1$ decision variables to be 1. In the next $|E(C)| - C$ rows, select all decision variables in C except for two with largest coefficient to be 1, select a decision variable that is in $E(C)$ but not in C to be 1. In the last $|N| - |E(C)|$ row, select all decision variables in C except for the one with largest coefficient to be 1, select a decision variable outside $E(C)$ to be 1. □

5.3 Generic Cutting Planes

5.3.1 General Approach

Cutting Planes

- Cutting planes are referred to inequalities valid for $conv(S)$, but which is violated by the solution obtained by solving the current LP relaxation.
- Cutting plane methods attempt to improve the bound produced by the LP relaxation by iteratively adding cutting planes to the initial LP relaxation.
- Adding such inequalities to the LP relaxation may improve the bound (this is not a guarantee).

Separation Problem Given a polygon $P \in \mathbb{R}^n$ and $\mathbf{x}^* \in \mathbb{R}^n$, determine whether $\mathbf{x}^* \in P$, and if not, determine (π, π_0), a valid inequality for P such that $\pi \mathbf{x}^* \geq \pi_0$

5.3.2 Generic Cutting Planes

Observation For integer program over a set of variables $x_1, x_2, \cdots, x_n \in \mathbb{Z}_+$

- If $\mathbf{ax} = \mathbf{b}$ is a constraint of P, then $\mathbf{ax} \leq \mathbf{b}$ is a valid constraint of P

- Suppose there are two valid constraints

$$a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n \leq b_1$$

and

$$a_{21}x_2 + a_{22}x_2 + \cdots + a_{2n}x_n \leq b_2$$

then

$$\begin{aligned} u_1(a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n) \\ +u_2(a_{21}x_2 + a_{22}x_2 + \cdots + a_{2n}x_n) \\ \leq u_1b_1 + u_2b_2 \end{aligned}$$

is a valid inequality for P

- If $a \leq b$ and a is an integer number, then $a \leq \lfloor b \rfloor$
- $x_i \in \mathbb{Z}_+ \Rightarrow -x_i \leq 0$
- WLOG, assume $\forall a_i$ in constraint $\sum_{i=1}^{n} a_i x_i \leq b$, a_i is a fractional number. Let $f_i = a_i - \lfloor a_i \rfloor$ be the fractional part, and $f_i \geq 0$. Then,

$$\sum_{i=1}^{n} a_i x_i - \sum_{i=1}^{n} f_i x_i \leq b - 0$$

$\Rightarrow$

$$\sum_{i=1}^{n} \lfloor a_i \rfloor \, x_i \leq \lfloor b \rfloor$$

is a valid inequality for P.

Chvátal-Gomory Rounding Method Let $X = P \cap \mathbb{Z}^n$ be the feasible set of an integer program, where

$$P = \{\mathbf{x} \in \mathbb{R}^n | \mathbf{Ax} \leq \mathbf{b}, \mathbf{x} \geq 0\}$$

Also let $\mathbf{a}_i$ be the ith column of $\mathbf{A}$, then

$$\sum_{i=1}^{n} \mathbf{u}\mathbf{a}_i x_i \leq \mathbf{ub}$$

is a valid inequality for P for $u \geq 0$, then

$$\sum_{i=1}^{n} \lfloor \mathbf{u}\mathbf{a}_i \rfloor \, x_i \leq \mathbf{ub}$$

is a valid inequality for P, then

$$\sum_{i=1}^{n} \lfloor \mathbf{u}\mathbf{a}_i \rfloor x_i \leq \lfloor \mathbf{u}\mathbf{b} \rfloor$$

is a valid inequality for P, since x_i is an integer. Such procedures are called Chvátal-Gomory rounding procedure, and such cuts are called Chvátal-Gomory Cuts.

Example 1 Let P be

$$\begin{aligned} 2x_1 + 3x_2 &\leq 5 \\ -x_1 + x_2 &\leq 2 \\ x_1, x_2 &\geq 0 \end{aligned}$$

Then

$$\mathbf{A} = \begin{bmatrix} 2 & 3 \\ -1 & 1 \end{bmatrix} \Rightarrow \mathbf{a}_1 = \begin{bmatrix} 2 \\ -1 \end{bmatrix}, \mathbf{a}_2 = \begin{bmatrix} 3 \\ 1 \end{bmatrix}$$

Let

$$\mathbf{u} = \begin{bmatrix} 1/2 & 1/3 \end{bmatrix}$$

, then

$$\begin{bmatrix} 1/2 & 1/3 \end{bmatrix} \begin{bmatrix} 2 \\ -1 \end{bmatrix} x_1 + \begin{bmatrix} 1/2 & 1/3 \end{bmatrix} \begin{bmatrix} 3 \\ 1 \end{bmatrix} x_2 \leq \begin{bmatrix} 1/2 & 1/3 \end{bmatrix} \begin{bmatrix} 5 \\ 2 \end{bmatrix}$$

which is

$$\frac{2}{3}x_1 + \frac{11}{6}x_2 \leq \frac{19}{6}$$

then,

$$\left\lfloor \frac{2}{3} \right\rfloor x_1 + \left\lfloor \frac{11}{6} \right\rfloor x_2 \leq \left\lfloor \frac{19}{6} \right\rfloor \Rightarrow x_2 \leq 3$$

Example 2 Let P be

$$\begin{aligned} 7x_1 - 2x_2 &\leq 14 \\ x_2 &\leq 3 \\ 2x_1 - 2x_2 &\leq 3 \\ x_1, x_2 &\geq 0 \end{aligned}$$

Let $u = \begin{bmatrix} \frac{2}{7} & \frac{37}{63} & 0 \end{bmatrix}$

Then $2x_1 + \frac{1}{63}x_2 \leq \frac{185}{21} \Rightarrow x_1 \leq 4$

Observation

- It is possible to generate $conv(X)$ using C-G procedures

- The crux lies on the choice of $\mathbf{u}$
- The number of iterations can be huge
- Nevertheless, the procedure has been proven to work well in practice in recent years to generate good inequalities
- One can try generating random multipliers and test the method

5.3.3 Gomory Cuts

Gomory cutting planes can also be derived directly from the tableau while solving an LP relaxation. Consider the set

$$P = \{(\mathbf{x}, \mathbf{s}) \in \mathbb{Z}_+^{n+m} | \mathbf{Ax} + \mathbf{Is} = \mathbf{b}\}$$

in which the LP relaxation of an ILP is put in standard form. Assume for A, all the coefficients are integers, so that the slack variables are integers. Clearly,

$$\mathbf{B^{-1}Ax} + \mathbf{B^{-1}Is} = \mathbf{B^{-1}b}$$

Let $\lambda = \mathbf{B^{-1}}_j$, then we obtain

$$\sum_{j=1}^{n} \lambda \mathbf{A_j} x_j + \sum_{i=1}^{m} \lambda_i s_i = \lambda \mathbf{b}$$

where $\mathbf{A}_j$ is the jth column of $\mathbf{A}$ and λ is a row if $\mathbf{B}^{-1}$. Then, the Gomory cut is define by

$$\sum_{j=1}^{n} (\lambda \mathbf{A}_j - \lfloor \lambda \mathbf{A}_j \rfloor) x_j + \sum_{i=1}^{m} (\lambda_i - \lfloor \lambda_i \rfloor) s_i \geq \lambda \mathbf{b} - \lfloor \lambda \mathbf{b} \rfloor$$

Gomory cut is a Chvátal-Gomory cut with weights $\mathbf{u}_i = \lambda_i - \lfloor \lambda_i \rfloor$.

Example For the following IP

$$\begin{aligned} \max \quad & 2x_1 + 5x_2 \\ \text{s.t.} \quad & 4x_1 + x_2 \leq 28 \\ & x_1 + 4x_2 \leq 27 \\ & x_1 - x_2 \leq 1 \\ & x_1, x_2 \in \mathbb{Z}_+ \end{aligned}$$

The optimal tableau of the LP relaxation is as follows

	x_1	x_2	s_1	s_2	s_3	RHS
x_2	0	1	-2/30	8/30	0	16/3
s_3	0	0	-1/3	1/3	1	2/3
x_1	1	0	8/30	-2/30	0	17/3

For the first row, the Gomory cut is

$$\frac{28}{30}s_1 + \frac{8}{30}s_2 \geq \frac{1}{3}$$

Replace the slack variables, we have

$$4x_1 + 2x_2 \leq 33$$

For the second row, the Gomory cut is

$$\frac{2}{3}s_1 + \frac{1}{3}s_2 \leq \frac{2}{3}$$

Replace the slack variables, we have

$$3x_1 + 2x_2 \leq 27$$

For the third row, the Gomory cut is

$$\frac{8}{30}s_1 + \frac{28}{30}s_2 \geq \frac{2}{3}$$

Replace the slack variables, we have

$$x_1 + 2x_2 \leq 16$$

This picture shows the effect of adding all Gomory cuts in the first round

5.4 Branch and Cut

Two types of valid inequalities The structural constraints are

- while this is not a standard term used in mathematical programming, we will use it in reference to the constraints that define a formulation
- these are constraints needs to define the problem. If removed, some infeasible integer solutions may become part of the solution space

The cutting planes are

- these are valid constraints that are not needed to define the problem but can be added to tighten the LP relaxation of a formulation
- in other words, they are used for trying to obtain a better LP bound

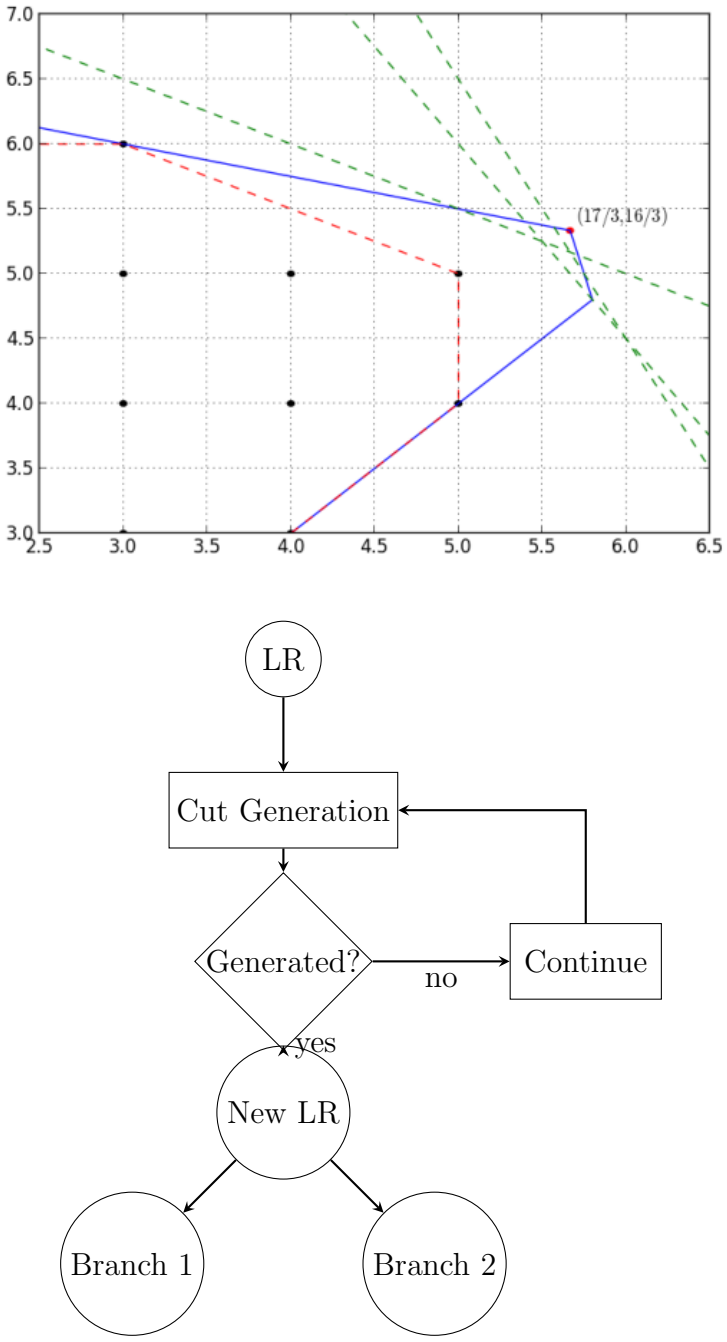


Figure 5.3: Branch and Cut for Optional Inequality

User cuts Consider the vertice packing problem, we have mentioned that the maximum cliques can be used for adding valid inequalities, this is how it works

- Given a fractional solution $\hat{x}$, we can find a clique for which $\sum_{i \in C} x_i \leq 1, C \in Clique(G)$ is violated

- Solve the following separation problem

$$\begin{aligned} \max \quad & \gamma = \sum_{i \in V} \hat{x}_i z_i \\ \text{s.t.} \quad & z_i + z_j \leq 1, \{i,j\} \notin E \\ & z_i \in \{0,1\}, i \in V \end{aligned}$$

- if $\gamma > 1$ add the clique cut associate with C.

Lazy cuts A typical example for lazy cuts is the DFJ sub-tour elimination for TSP. We will discuss later in this semester.

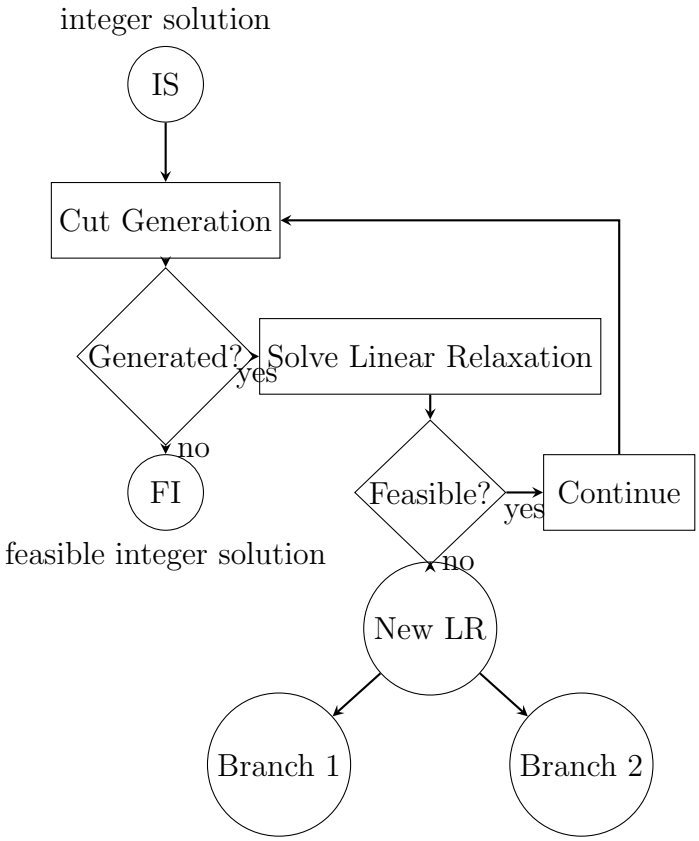


Figure 5.4: Branch and Cut for Essential Inequality

Chapter 6

Graph Algorithm

"You can't visit all seven bridges in Königsberg without repetition."

6.1 Preliminaries

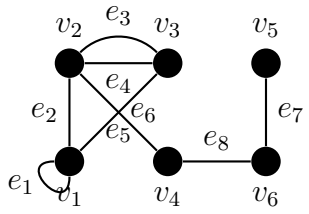
6.1.1 Graphs and Subgraphs

Graphs

Definition 6.1.1 (Graph)**.** A **graph** G consists of a finite set $V(G)$ on vertices, a finite set $E(G)$ on edges and an **incident relation** than associates with any edge $e \in E(G)$ an unordered pair of vertices not necessarily distinct called **ends**.

Example. The following graph can be represented as

$$V = V(G) = \{v_1, v_2, v_3, v_4, v_5, v_6\}$$
$$E = E(G) = \{e_1, e_2, e_3, e_4, e_5, e_6, e_7, e_8\}$$
$$e_1 = v_1v_2, \quad e_2 = v_2v_4, \quad \dots$$



Definition 6.1.2 (Adjacent)**.** Two edges of a graph are **adjacent** if they have a common end, two vertices are **adjacent** if they are jointed by an edge.

Saving a graph in computer program can be implemented in the following ways:

- Adjacency matrix: $m \times n$ matrix, for $A[u, v] = 1$ if $(u, v) \in E$ and $A[u, v] = 0$ otherwise
- Linked list: For every vertex v, there is a linked list containing all neighbors of v.

Assuming we are dealing with undirected graphs, n is the number of vertices, m is the number of edges, $n-1 \leq m \leq n(n-1)/2$, d_v is the number of neighbors of v then

	Matrix	Linked lists
memory usage	$O(n^2)$	$O(m)$
time to check $(u,v) \in E$	$O(1)$	$O(d_u)$
time to list all neighbors of v	$O(n)$	$O(d_v)$

Subgraphs

Definition 6.1.3 (Subgraph)**.** Given two graphs G and H, H is a **subgraph** of G if $V(H) \subseteq V(G)$, $E(H) \subseteq E(G)$ and an edge has the same ends in H as it does in G. Furthermore, if $E(H) \neq E(G)$ then H is a proper subgraph.

Definition 6.1.4 (Vertex-induced, Edge-induced)**.** For a subset $V' \subseteq V(G)$ we define an **vertex-induced** subgraph $G[V']$ to be the subgraph with vertices V' and those edges of G having both ends in V'. The **edge-induced** subgraph $G[E']$ has edges E' and those vertices of G that are ends to edges in E'.

If we combine node-induced or edge-induced subgraphs $G(V')$ and $G(V-V')$, we cannot always get the entire graph.

Definition 6.1.5 (Degree)**.** Let $v \in V(G)$, then the **degree** of $v \in V(G)$ denote by $d_G(v)$ is defines to be the number of edges incident of v. Loops counted twice.

Theorem 6.1.1. *For any graph $G = (V, E)$*

$$\sum_{v \in V} d(v) = 2|E|$$

Proof. $\forall$ edge $e = uv$ with $u \neq v$, e is counted once for u and once for v, a total of two altogether. If $e = uu$, a loop, then it is counted twice for u. □

Corollary 6.1.1.1. *Every graph has an even number of odd degree vertices.*

Proof.

$$V = V_E \cup V_O \Rightarrow \sum_{v \in V} d(v) = \sum_{v \in V_E} d(v) + \sum_{v \in V_O} d(v) = 2|E|$$

□

Special graphs

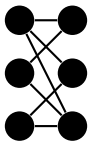
Definition 6.1.6 (Complete Graph)**.** A **complete** graph $K_n (n \geq 1)$ is a simple graph with n vertices and with exactly one edge between each pair of distinct vertices.

Definition 6.1.7 (Cycle Graph)**.** A **cycle** graph $C_n (n \geq 3)$ consists of n vertices $v_1, ...v_n$ and n edges $\{v_1, v_2\}, \{v_2, v_3\}, ...\{v_{n-1}, v_n\}$

Definition 6.1.8 (Wheel Graph)**.** A **wheel** graph $W_n (n \geq 3)$ is a simple graph obtains by adding one vertex to the cycle graph C_n, and connecting this new vertex to all vertices of C_n

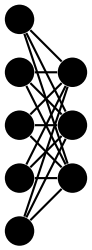
Definition 6.1.9 (Bipartite Graph)**.** A simple graph is said to be **bipartite** if the vertex set can be expressed as the union of two disjoint non-empty subsets V_1 and V_2 such that every edges has one end in V_1 and another end in V_2

Example. Here is an example for bipartite graph



Definition 6.1.10 (Complete Bipartite Graph)**.** The **complete bipartite graph** K_{mn} is the bipartite graph V_1 containing m vertices and V_2 containing n vertices such that each vertex in V_1 is adjacent to every vertex in V_2

Example. Here is an example for K_{53}



Theorem 6.1.2. *A graph G is bipartite iff every cycle is even.*

Proof. ($\Rightarrow$) If the graph G is bipartite, by definition, the vertices of graph can be partition into two groups, that within the group there is no connection between vertices. Therefore, for each cycle, the odd index of vertices and even index of vertices has to be choose alternatively from each groups. Therefore the cycle has to be even.

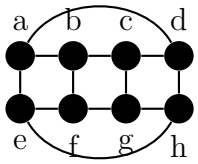
($\Leftarrow$) Prove by contradiction. A graph can be connected or not connected.

If G is connected and has at least two vertices, for an arbitrary vertex $v \in V(G)$, we can calculate the minimum number of edges between the other vertices v' and v (i.e., length, denoted by $l(v', v)$), for all the vertices that has odd length to v, assign them to set V_1, for the rest of vertices (and v), assign to set V_2. Assume that G is not bipartite, which means there are at least one edge between distinct vertices in set V_1 or set V_2, without lost of generality, assume that edge is uw, $u, w \in V_1$. For all vertices in V_1 there is an odd length of path between the vertex and v, therefore, there exists an odd $l(u, v)$, and an odd $l(w - v)$. The length of cycle $l(u, w, v) = 1 + l(u, v) + l(w, v)$, which is an odd number, it contradict with the prerequisite that all cycles are even, which means the assumption that G is not bipartite is incorrect, G should be bipartite.

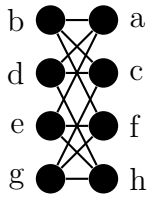
If G is not connected. Then G can be partition into a set of disjointed subgraphs which are connected with at least two vertices or contains only one vertex. For the component that

has more that one vertices, we already proved that it has to be bipartite. For the subgraph $G_i \subset G, i = 1, 2, ..., n$, the vertices can be partition into $V_{i1} \in V(G_i)$ and $V_{i2} \in V(G_i)$, where $V_{i1} \cap V_{i2} = \emptyset$, the union of those subgraphs are bipartite too because $V_1 = \cup_{i=1}^{n} V_{i1} \in V(G)$ and $V_2 = \cup_{i=1}^{n} V_{i2} \in V(G)$ satisfied the condition of bipartite. For the subgraph that has one one vertices, those vertices can be assigned into either V_1 or V_2. □

Example. The following graph is bipartite, it only contains even cycles.

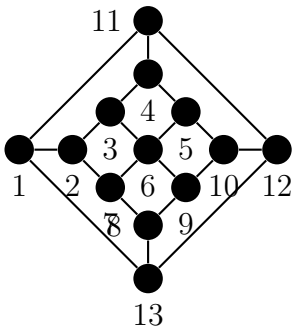


We can rearrange the graph to be more clear as following



The vertices of graph G can be partition into two sets, $\{a, c, f, h\}$ and $\{b, d, e, g\}$

Example. The following graph is not bipartite



The cycle $c = v_1 v_{11} v_4 v_3 v_2$ have odd number of vertices.

6.1.2 Walk, Path and Cycle

Walk, trail, path

Definition 6.1.11 (walk)**.** A **walk** in a graph G is a finite sequence $w = v_0 e_1 v_1 e_2 ... e_k v_k$, where for each $e_i = v_{i-1} v_i$ the edge and its ends exists in G. We say that walk v_0 to v_k on (v_0, v_k)-walk.

Example.

$$w = v_2e_4v_3e_4v_2e_5v_3$$

is a walk, or (v_2, v_3)-walk

Definition 6.1.12 (origin, terminal, internal, length)**.** For (v_0, v_k)-walk, The vertices v_0 and v_k are called the **origin** and the **terminal** of the walk w, $v_1..v_{k-1}$ are called **internal** vertices. The integer k is the **length** of the walk. Length of w equals to the number of edges.

We can create a reverse walk w^{-1} by reversing w.

$$w^{-1} = v_ke_kv_{k-1}e_{k-1}...e_2v_1$$

(The reverse walk is guaranteed to exist because it is an undirected graph)

Given two walks w and w' we can create a third walk denoted by ww' by concatenating w and w'. The new walk's origin is the same as terminal.

Definition 6.1.13 (trail)**.** A **trail** is a walk with no repeating edges. e.g., $v_3e_4v_2e_5v_3$

Definition 6.1.14 (path)**.** A **path** is a trail with no repeating vertices. e.g., $v_3e_4v_2$

Paths $\subseteq$ Trails $\subseteq$ Walks

Cycle

Definition 6.1.15 (closed, cycle)**.** A path is **closed** if it has positive length and its origin and terminal are the same. e.g., $v_1e_2v_2e_4v_3e_3v_1$. A closed trail where origin and internal vertices are distinct is called a **cycle** (The only time a vertex is repeated is the origin and terminal)

Definition 6.1.16 (even cycle, odd cycle)**.** A cycle is **even** if it has a even number of edges otherwise it is **odd**.

Problem 6.1.1. Prove that if C_1 and C_2 are cycles of a graph, then there exists cycles $K_1, K_2, ..., K_m$ such that $E(C_1)\Delta E(C_2) = E(K_1)\cup E(K_2)\cup...\cup E(K_m)$ and $E(K_i)\cap E(K_j) = \emptyset, \forall i \neq j$. (For set X and Y, $X\Delta Y = (X-Y)\cup(Y-X)$, and is called the symmetric difference of X and Y)

Proof. Proof by constructing $K_1, K_2, ...K_m$. Denote

$$C_1 = v_{11}e_{11}v_{12}e_{12}v_{13}e_{13}...v_{1n}e_{1n}v_{11}$$
$$C_2 = v_{21}e_{21}v_{22}e_{22}v_{23}e_{23}...v_{2k}e_{2k}v_{21}$$

Assume both cycle start at the same vertice, $v_{11} = v_{12}$. (If there is no intersected vertex for C_1 and C_2, just simply set $K_1 = C_1$ and $K_2 = C_2$)
The following algorithm can give us all $K_j, j = 1, 2, ..., m$ by constructing $E(C_1)\Delta E(C_2)$. Also, the complexity is $O(mn)$, which makes the proof doable.

Algorithm 13 Find $K_1, K_2, ...K_m$ by constructing $E(C_1)\Delta E(C_2)$

Require: Graph G, cycle C_1 and C_2
Ensure: $K_1, K_2, ...K_m$
Initial, $K \leftarrow \emptyset$, $j = 1$
Set temporary storage units, $v_o \leftarrow v_{11}$, $v_t \leftarrow \emptyset$
for $i = 1, 2, ..., n$ **do**
    if $e_{1i} \in C_2$ **then**
        if $v_o \neq v_{1i}$ **then**
            $v_t \leftarrow v_{1i}$
            concatenate (v_o, v_t)-path $\subset C_1$ and (v_o, v_t)-path $\subset C_2$ to create a new K_j
            Append K with K_j, $K \leftarrow K \cup K_j$
            Reset temporary storage unit. $v_o \leftarrow v_{1(i+1)}$ (or v_{11} if $i = n$), $v_t \leftarrow \emptyset$
        else
            $v_o \leftarrow v_{1(i+1)}$ (or v_{11} if $i = n$)

Now we prove that $K_i \cap K_j = \emptyset, \forall i \neq j$. For each K_j, it is defined by two (v_o, v_t)-paths in the algorithm. From the algorithm we know that all the edges in (v_o, v_t)-path in C_1 are not intersecting with C_2, because if the edge in C_1 is intersected with C_2, either we closed the cycle K_j before the edge, or we updated v_o after the edge (start a new K_j after that edge). By definition of cycle, all the (v_o, v_t)-path that are subset of C_1 are not intersecting with each other, as well as all the (v_o, v_t)-path that are subset of C_2. Therefore, $K_i \cap K_j = \emptyset, \forall i \neq j$. □

6.1.3 Some warm-up algorithms

Component

Definition 6.1.17 (connected vertices)**.** Two vertices u and v in a graph are said to be **connected** if there is a path between u and v.

Definition 6.1.18 (component)**.** Connectivity between vertices is an equivalence relation on $V(G)$, if $V_1, ...V_k$ are the corresponding equivalent classes then $G[V_1]...G[V_k]$ are **components** of G. If graph has only one component, then we say the graph is connected. A graph is connected iff every pair of vertices in G are connected, i.e., there exists a path between every pair of vertices.

The following algorithm finds components in a given graph $G = (V, E)$.

Algorithm 14 Find components

Let Q be the set of components
for $i \in V$ **do**
    $Q \leftarrow \{i\}$
for $e = (i, j) \in E$ **do**
    for $p \in Q$ **do**
        for $q \in Q$ **do**
            if $i \in p$ and $j \in q$ **then**
                $Q \leftarrow Q \setminus p \setminus q \cup (p \cup Q)$
return Q

We will revisit this procedure when we discuss the subtour elimination constraints for TSP. Such procedure can be improved by introducing disjoint-set forest.

Cycle detection The following algorithm is for connected graph, if the graph is not connected, run the algorithm for each component until cycle is detected or all the components have been calculated. Since the complexity for running in connected graph is $O(n + m)$, n as the number of vertices/nodes, and m as the number of edges/arcs, the running time of disconnected graph is the **summation** of running time in each component, where each component is connected. Therefore the complexity is the same in disconnected graph as in connected graph.

The main idea is starting with arbitrary vertex/node, using DFS or BFS to search on the graph try to revisit the vertex/node we start with. If succeed, a cycle is detected, otherwise if all the vertices/nodes has been visited, then no cycle exists. And in linked-list representation, the complexity is $O(|V| + |E|)$, i.e. $O(n + m)$. However, there is slightly different in undirected graph and directed graph, for undirected graph needs at least three vertices to form a cycle while directed graph needs at least two.

Here is the detail algorithm (DFS) for undirected graph:

Algorithm 15 Main algorithm

For all nodes, labeled as "unvisited"
Arbitrary choose a vertex v, add a dummy vertex w, add a dummy edge (w, v), label w as "visited"
run $DFS(w, v)$
Remove dummy vertex w and dummy edge (w, v)
if $DFS(w, v)$ returns "Cycle is found" **then**
    return "Cycle is found"
else
    return "No cycle detected"

Algorithm 16 DFS(w, v)

```
1: Label v as "visited"
2: if number of v 's neighbor is 1 then
3:     return null
4: else
5:     for all neighbor u in linked-list of v excepts w do
6:         if u is labeled as "visited" then
7:             return "Cycle is found"
8:         else
9:             run DFS(v, u)
```

Now check the complexity. Denote $v \in N$ as a node in graph G, total number of nodes denoted by n, denote d_v as number of neighbors of node v. The complexity of $DFS(w, v)$ is $O(d_v)$ for each node v it visited (it should be $O(v)$ because we need $O(1)$ to check if a node is w), each node can only be visited, by "visited" it means $DFS(w, v)$ is executed, at most once, which is controlled by the "visited" label. The total complexity is $O(n + \sum_{v \in N} d_v) = O(n + m)$

Test Bipartiteness We have proved that a bipartite graph only has even cycles, and the graph with only even cycles are bipartite graph, however, that is not very convenient to test if a graph is bipartite because it needs to enumerate all cycles.

The other idea to test bipartiteness is try to color the vertices of the graph, if it can be 2-colored, then the graph is bipartite, otherwise it is not.

The following is the algorithm (using BFS)

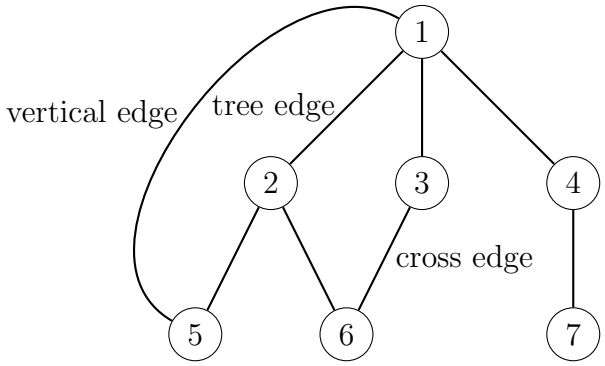
Algorithm 17 Test Bipartiteness

```
1: Initialize, head ← 1, tail ← 1, queue[1] ← s, mark all vertices as "unvisited"
2: Mark s as "visited"
3: color[s] ← 0
4: while head ≥ tail do
5:     v ← queue[tail], tail ← tail + 1
6:     for ∀u ∈ d_v do
7:         if u is "unvisited" then
8:             head ← head + 1, queue[head] = u
9:             Mark u as "visited"
10:            color[u] ← 1 − color[v]
11:        else
12:            if color[u] == color[v] then return False
   return True
```

Bridge

Definition 6.1.19 (Tree Edge, Cross Edge, Vertical Edge)**.** Given a graph $G = (V, E)$ and a rooted tree T in G, edge in G can be classified by three types:

- Tree edges: edge in T
- Cross edges: (u, v), u and v do not have an ancestor-descendant relation
- Vertical edges: (u, v), u is an ancestor of v, or v is an ancestor of u



In a BFS tree T of a graph G, there can not be vertical edges, there cannot be cross edges (u, v) with u and v 2 levels apart. (Cross edge at most 1 level apart)

In a DFS tree T of a graph G, there can not be cross edges, there can only be tree edges and vertical edges.

Definition 6.1.20 (bridge)**.** Given a connected graph $G = (V, E)$, an edge $e \in E$ is called a **bridge** if the graph $G = (V, E \setminus \{e\})$ is disconnected.

The idea to find bridge is through a DFS tree. Notice that there are only tree edges and vertical edges in DFS tree. Vertical edges are not bridges, a tree edge (u, v) is not a bridge if some vertical edge jumping from below u to above v. Other tree edges are bridges.

Define $level(v)$ as the level of vertex v in DFS tree. T_v as the sub tree rooted at v, $h(v)$ as the smallest level that can be reached using a vertical edge from vertices in T_v. $(parent(u), u)$ is a bridge if $h(u) \geq level(u)$. The algorithm is as following:

Algorithm 18 FindBridge(G)

```
1: Mark all vertices as "unvisited"
2: for v ∈ V do
3:     if v is "unvisited" then
4:         level(v) ← 0
5:         RecursiveDFS(v)
```

Algorithm 19 RecursiveDFS(v)

```
1: mark v as "visited"
2: h(v) ← ∞
3: for u ∈ d_v do
4:     if u is "unvisited" then
5:         level(u) ← level(v) + 1
6:         RecursiveDFS(u)
7:         if h(u) ≥ level(u) then
8:             (u, v) is a bridge
9:         if h(u) < h(v) then
10:            h(v) ← h(u)
11:    else
12:        if level(u) < level(v) − 1 and level(u) < h(v) then
13:            h(v) ← level(u)
```

6.2 Matching

6.2.1 Matching

Definition 6.2.1 (matching, M-saturated, M-unsaturated)**.** Let $G = (V, E)$ be a graph, a **matching** is a subset of edges $M \subseteq E$ such that no two elements of M are adjacent. The two ends of an edge in M are said to be **matched** under M. A matching M saturates a vertex v, and v is said to be **M-saturated**, if some edge of M is incident with v. Otherwise, v is **M-unsaturated**.

Definition 6.2.2 (perfect matching, maximum matching)**.** If every vertex of G is M-saturated, then the matching is said to be **perfect matching**, for perfect matching, we have $|M| = \dfrac{|V(G)|}{2}$. M is a **maximum matching** if G has no matching M' with $|M'| > |M|$. Every perfect matching is maximum. The maximum matching does not necessarily to be perfect. Perfect matching and maximum matching may not be unique.

Definition 6.2.3 (M-alternating)**.** An **M-alternating** path in G is a path whose edges are alternately in $E \setminus M$ and M.

Definition 6.2.4 (M-augmenting)**.** An **M-augmenting** path in G is an M-alternating path whose origin and terminus are M-unsaturated.

Lemma 6.2.1. *Every augmenting path P has property that let $M' = P\Delta M = (M \cup P) \setminus (M \cap P)$ then M' contains one more edge then M*

The following path is an M-augmenting path

The following path is $M' = P\Delta M(M \cup P) \setminus (M \cap P)$ and all the vertices are M-saturated.

Theorem 6.2.2 (Berge, 1957)**.** *A matching M in a graph G is maximum iff G has no M-augmenting path.*

Proof. ($\Rightarrow$) It is clear that if M is maximum, it has no augmenting paths since otherwise by problem claim we can increase by one.

($\Leftarrow$) Suppose M is not maximum and let M' be a bigger matching. Let $A = M \Delta M'$ now no vertex of G is incident to more than two members of A. For otherwise either two members of M or two members of M' would be adjacent. Contradict the definition of matching. It follows that every component of the edges incident subgraph $G[A]$ is either an even cycle with edge augmenting in $M \Delta M'$ or else A path with edges alternating between M and M'.

Since $|M'| \geq |M|$ then the even cycle cannot help because exchanging M and M' will have same cardinality.

The path case implies that p is alternating in M and since $|M'| > |M|$ the end arc exposed so that p is augmenting. □

Definition 6.2.5 (Vertex-cover)**.** The **vertex-cover** is a subset of vertices X such that every edge of G is incident to some member of X.

Lemma 6.2.3. *The cardinality of any matching is less than or equal to the cardinality of any vertex cover.*

Proof. Consider any matching. Any vertex cover must have nodes that at least incident to the edges in the matching. Since all the edges in the matching are disjointed, so for a single node can at most cover one edge in the matching. If the matching is not perfect, for the edges that not in the tree, they may or may not be possible to be covered by the nodes incident to the edges in the matching, with an easy triangle graph example, we can prove this lemma. □

Theorem 6.2.4 (König Theorem)**.** *If G is bipartite, the cardinality of the maximum matching is equal to the cardinality of the minimum vertex cover.*

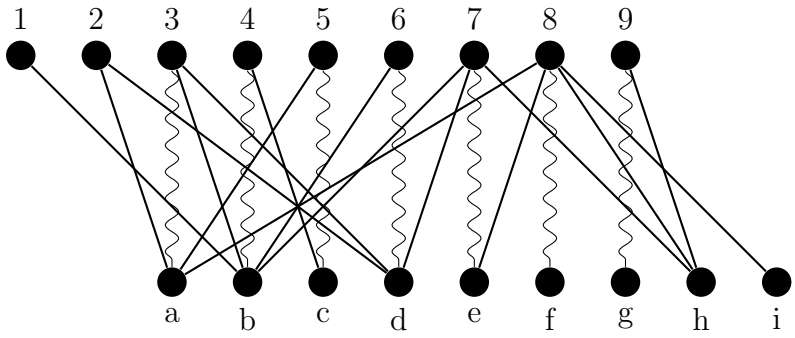
Proof. Let G be a bipartite graph, $G = (V, E)$ where $V = X \cup Y$ as X and Y are two disjointed sets of vertices. Let M be a maximum matching on G. For each edge in M, denoted by $e_i = a_i b_i$ where $e_i \in M$, $a_i \in A$ and $b_i \in B$ and $A = \{a_i : e_i \in M\} \subseteq X$, and $B = \{b_i : e_i \in M\} \subseteq Y$. Therefore, we can partition X by A and $U = X \setminus A$, partition Y by B and $W = Y \setminus B$.

We can further partition the matching M into M_1 and M_2. For all the edges in M_1 can be included into an M-alternating path starts from a vertex in U (which includes the edges directly linked to vertices in U), and $M_2 = M \setminus M_1$. For edges in M_1, we take the ends of edges in B in the vertex cover, denoted by B_1, take the ends of edges in A as a subset denoted by $A_1 \subseteq A$. For the edges in M_2, we take the ends of edges in A in the vertex cover, denoted by A_2, and the ends of edges in B as a subset denoted by $B_2 \subseteq B$.

We claim that all the vertices in U can only be connected to vertices in B_1 and vertices in W can only be connected to vertices in A_2.

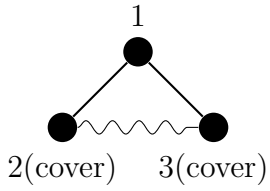
$U \subset X$ connects to vertices in B_1 by definition. If vertices in $W \subset Y$ is connected to vertices in A_1, then we will have M-augmenting path which is contradicted to the assumption that M is maximum matching. □

The following is an example. Where the edge in the matching that accessible from members of $U = \{1, 2\}$ in an M-alternating path is edge $3a, 4b, 5c, 6d$.



In which $U = \{1, 2\}$, $M_1 = \{3a, 4b, 5c, 6d\}$. $U = \{1, 2, \}, A_1 = \{3, 4, 5, 6\}, A_2 = \{7, 8, 9\}, W = \{h, i\}, B_1 = \{a, b, c, d\}, B_2 = \{e, f, g\}$. The vertex cover is $\{a, b, c, d, 7, 8, 9\}$.

The above theorem does not apply to non-bipartite graph. The following is an example



The maximum matching has one edge, where the minimum cover has two vertices.

6.2.2 Blossom Algorithm

First, for the bipartite graphs, there is no odd cycles in the graph, the maximum matching can be derived by repeatedly extend the existing augmenting paths. For the non-bipartite algorithm, we use the Blossom algorithm to find maximum matching.

Blossom A blossom is a set of nodes and edges starting at an unsaturated vertex, with a stem of even number of edges, and link to an odd cycle of edges.

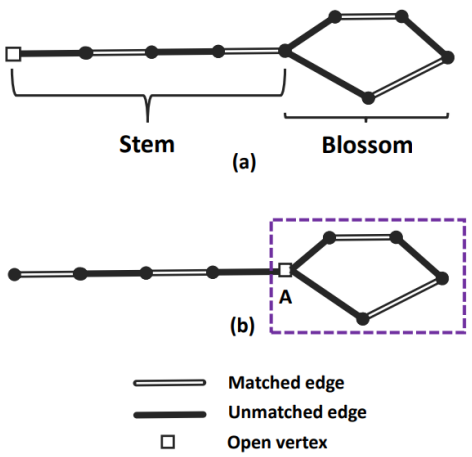


Figure 6.1: An example of blossom

Algorithm 20 Blossom algorithm

```
Let F be the set of unsaturated nodes
while F ≠ ∅ do
    Let r ∈ F be an unsaturated node
    queue.push(r)
    T ← ∅
    T.add(r)
    while queue ≠ ∅ do
        v ← queue.pop()
        for All neighbor w of v do
            if w ∉ T and w is saturated then
                T.add(w)
                T.add(mate(w)), mate(w) is another neighbor of w
                queue.push(mate(w))
            else if w ∈ T and odd cycle detected then
                Contract cycle
            else if w ∈ F then
                Expand all contract cycles
                Reconstruct augmenting path
                Augment path by inverting edges
```

Pseudo codes

6.3 Maximum Flow Problem

6.3.1 Maximum Flow Problem

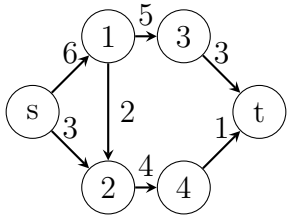
Let $D = (V, A)$ be a strict digraph with distinguished vertices s and t. We call s the source and t the sink, let $u = \{u_e : e \in A\}$ be a nonnegative integer-valued capacity function defined on the arcs of D. The maximum flow problem on (D, s, t, u) is the following Linear program.

$$\begin{aligned} \max \quad & v \\ \text{s.t.} \quad & \sum_{h(e)=i} x_e - \sum_{t(e)=i} x_e = \begin{cases} -v, & \text{if } i = s \\ v, & \text{if } i = t \\ 0, & \text{otherwise} \end{cases} \\ & 0 \le x_e \le u_e, \quad \forall e \in A \end{aligned}$$

We think of x_e as being the flow on arc e. Constraint says that for $i \neq s, t$ the flow into a vertex has to be equal to the flow out of vertex. That is, flow is conceded at vertex i for $i = s$ and for $i = t$ the net flow in the entire digraph must be equal to v. A $\mathbf{x_e}$ that satisfied the above constraints is an (s, t)-flow of value v. If in addition it satisfies the bounding constraints, then it is a feasible (s, t)-flow. A feasible (s, t)-flow that has maximum v is optimal on maximum.

Theorem 6.3.1. *For $S \subseteq V$ we define $(S, \bar{S})$ to be a (s, t)-cut if $s \in S$ and $t \in \bar{S} = V - S$, the capacity of the cut, denoted $u(S, \bar{S})$ as $\sum\{u_e : e \in \delta^-(S)\}$ where $\delta^-(S) = \{e \in A : t(e) \in S$ and $h(e) \in \bar{S}\}$*

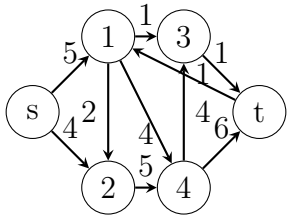
Example. For the following graph:



Let $S = \{1, 2, 3, s\}$, $\bar{S} = \{4, t\}$
then $\delta^-(S) = \{(2, 4), (3, t)\} \Rightarrow u(S, \bar{S}) = 7$

Definition 6.3.1. If $(S, \bar{S})$ has minimum capacity of all (s, t)-cuts, then it is called **minimum cut**.

Definition 6.3.2. Let $\delta^+(S) = \delta^-(V - S)$



Example. Let $S = \{s, 1, 2, 3\}$, $\bar{S} = \{4, t\}$, $u(S, \bar{S}) = u_{14} + u_{24} + u_{3t} = 10$, $\delta^-(S) = \{(1,4), (2,4), (3,t)\}$, $\delta^+(S) = \{(4,3), (t,1)\}$

Lemma 6.3.2. *If x is a (s,t) flow of value v and $(S, \bar{S})$ is a (s,t)-cut, then*

$$v = \sum_{e \in \delta^-(S)} x_e - \sum_{e \in \delta^+(S)} x_e$$

Proof. Summing the first set of constraints over the vertices of S,

$$\sum_{i \in S} (\sum_{h(e)=i} x_e - \sum_{t(e)=i} x_e) = -v$$

Now for an arc e with both ends in S, x_e will occur twice once with a positive and once with negative so they cancel and the above sum is reduced to

$$\sum_{e \in \delta^+(S)} x_e - \sum_{e \in \delta^-(S)} x_e = -v$$

□

Flow is the prime variable, capacity is the dual variable.

Corollary 6.3.2.1. *If x is a feasible flow of value v, and $(S, \bar{S})$ is an (s,t)-cut, then*

$$v \leq u(S, \bar{S}) \quad \textit{(Weak duality)}$$

Definition 6.3.3. Define an arc e to be **saturated** if $x_e = u_e$, and to be **flowless** if $x_e = 0$

Corollary 6.3.2.2. *Let x be a feasible flow and $(S, \bar{S})$ be a (s,t)-cut, if $\forall e \in \delta^-(S)$ is saturated, and $\forall e \in \delta^+(S)$ is flowless, then x is a maximum flow and $(S, \bar{S})$ is a minimum cut. (Strong duality)*

Proof. If every arc of $\delta^-(S)$ is saturated then

$$\sum_{e \in \delta^-(S)} x_e = \sum_{e \in \delta^-(S)} u_e$$

If every arc of $\delta^+(S)$ is flowless then

$$\sum_{e \in \delta^+(S)} x_e = 0$$

$\Rightarrow x$ is as large as it can get when as $u(S, \bar{S})$ is as small as it can get. $\square$

6.3.2 Prime and Dual of Maximum Network Flow Problem

The LP of maximum flow can be modeled as following, WLOG, we let $s = v_1 \in V, t = v_{|V|} \in V$.

$$\begin{aligned} \max \quad & f = \begin{bmatrix} 0 & 0 & \cdots & 0 & 1 \end{bmatrix} \begin{bmatrix} \mathbf{x} \\ f \end{bmatrix} \\ \text{s.t.} \quad & \begin{bmatrix} \mathbf{A} & \mathbf{F} \end{bmatrix} \begin{bmatrix} \mathbf{x} \\ f \end{bmatrix} = \mathbf{0} \\ & \mathbf{I}\mathbf{x} \leq \mathbf{u} \\ & \begin{bmatrix} \mathbf{x} \\ f \end{bmatrix} \geq 0 \end{aligned}$$

In which $\mathbf{A}$ is the vertex-arc incident matrix and $\mathbf{F}$ is a column vector where the first row is -1, last row is 1 and all other rows are 0s, which is because we denote the first vertex as source s and the last vertex as the sink t. $\mathbf{u}$ is the column vector of upper bound of each arcs.

$$\mathbf{A} = \mathbf{A}_{|E| \times |V|} = [a_{ij}], \text{ where } a_{ij} = \begin{cases} 1, & \text{if } v_i = h(e_j) \\ -1, & \text{if } v_i = t(e_j) \\ 0, & \text{otherwise} \end{cases}$$

$$\mathbf{F} = \begin{bmatrix} -1 & \cdots & 0 & \cdots & 1 \end{bmatrix}^\top$$

$$\mathbf{u} = \begin{bmatrix} u_1 & u_2 & \cdots & u_{|E|} \end{bmatrix}^\top$$

Then, we take the dual of LP

$$\begin{aligned} \min \quad & \mathbf{u}\mathbf{w}_\mathbf{E} \\ \text{s.t.} \quad & \begin{bmatrix} \mathbf{w}_\mathbf{V} & \mathbf{w}_\mathbf{E} \end{bmatrix} \begin{bmatrix} \mathbf{A} \\ \mathbf{I} \end{bmatrix} \geq 0 \\ & \begin{bmatrix} \mathbf{w}_\mathbf{V} & \mathbf{w}_\mathbf{E} \end{bmatrix} \begin{bmatrix} \mathbf{F} \\ \mathbf{0} \end{bmatrix} = 1 \\ & \mathbf{w}_\mathbf{V} \quad \text{unrestricted} \\ & \mathbf{w}_\mathbf{E} \geq \mathbf{0} \end{aligned}$$

In which $\mathbf{w}_\mathbf{V}$ is "whether or not" vertex v is in S where $(S, \bar{S})$ represents a cut, $\mathbf{w}_\mathbf{E}$ is "whether or not" an arc in in $\delta^+(S)$. $\mathbf{u}, \mathbf{E}, \mathbf{F}$ have the same meaning as in prime.

$$\mathbf{w}_\mathbf{V} = \begin{bmatrix} w_1 & w_2 & \cdots & w_{|V|} \end{bmatrix}^\top$$

$$\mathbf{w}_\mathbf{E} = \begin{bmatrix} w_{|V|+1} & w_{|V|+2} & \cdots & w_{|V|+|E|} \end{bmatrix}^\top$$

To make it more clear, it can be rewritten as following

$$\begin{aligned}
\min \quad & \sum_{e \in E} u_e w_e \\
\text{s.t.} \quad & w_i - w_j + w_{|V|+e} \geq 0, \forall e = (i,j) \in E \\
& - w_1 + w_{|V|} = 1 \\
& \mathbf{w_V} \quad \text{unrestricted} \\
& \mathbf{w_E \geq 0}
\end{aligned}$$

The meaning for the first set of constraint is to decide whether or not an arc is in $\delta^+(S)$ of a $(S, \bar{S})$, which is decided by w_V. The $w_1 - w_{|V|} = 1$, which is the second set of constraint means the source $s = v_1$ and the sink $t = v_{|V|}$ has to be in S and $\bar{S}$ respectively.

6.3.3 Maximum Flow Minimum Cut Theorem

Definition 6.3.4. Let P be a path, (not necessarily a dipath), P is called **unsaturated** if every **forward** arc is unsaturated ($x_e < u_e$) and every **reverse** arc has positive flow ($x_e > 0$). If in addition P is an (s,t)-path, then P is called an **x-augmenting path**

Theorem 6.3.3. *A feasible flow x in a digraph D is maximum iff D has no augmenting paths.*

Proof. (Prove by contradiction)

($\Rightarrow$) Let x be a maximum flow of value v and suppose D has an augmenting path. Define in P (augmenting path):

$$\begin{aligned}
D_1 &= \min\{u_e - x_e : e \text{ forward in } P\} \\
D_2 &= \min\{x_e : e \text{ backward in } P\} \\
D &= \min\{D_1, D_2\}
\end{aligned}$$

Since P is augmenting, then $D > 0$, let

$$\hat{x_e} = \begin{cases} x_e + D & \text{If } e \text{ is forward in } P \\ x_e - D & \text{If } e \text{ is backward in } P \\ x_e & otherwise \end{cases}$$

It is easy to see that $\hat{x}$ is feasible flow and that the value is $V + D$, a contradiction.

($\Leftarrow$) Suppose D admits no x-augmenting path, Let S be the set of vertices reachable from s by x-unsaturated path clearly $s \in S$ and $t \notin S$ (because otherwise there would be an augmenting path). Thus, $(S, \bar{S})$ is a (s,t)-cut.

Let $e \in \delta^-(S)$ then e must be saturated. For otherwise we could add the $h(e)$ to S

Let $e \in \delta^+(S)$ then e must be flow less. For otherwise we could add the $t(e)$ to S.

According to previous corollary, that x is maximum. □

Theorem 6.3.4. *(Max-flow = Minimum-cut) For any digraph, the value of a maximum (s,t)-flow is equal to the capacity of a minimum (s,t)-cut*

6.3.4 Ford-Fulkerson Method

Augmenting path An augmenting path is a path of edges in the residual graph with unused capacity greater than zero from the source s to target t. Every augmenting path has a bottleneck, which limits the flow that can go through this path.

Ford-Fulkerson method The Ford-Fulkerson method is a greedy algorithm with the following procedures:

- Iteratively find a path P from $s \to t$ with $c_f(e) = c(e) - f(e)$, where f is the current flow, and c_f is the so called residual capacity where

$$\forall (u,v) \in E : c_f(u,v) = c(u,v) - f(u,v), \quad \text{if } f(u,v) \geq 0 : c_f(v,u) = f(u,v) \tag{6.1}$$

- Then, define $\delta(P) = \min_{e \in P} c_f(e)$ and

$$f_P(e) = \begin{cases} \delta(P), & \text{if } e \in P \\ 0, & \text{otherwise} \end{cases} \tag{6.2}$$

  then we assign new values to the flow at each edge

$$f_{new} = f_{old} + f_P \tag{6.3}$$

  where both flows satisfy noon-negativity, capacity, and flow conservation constraints. And by the definition of the residual capacities, the original capacities constraints are never violated.

- The procedure terminates when no such path $s \to t$ can be found with $c_f(e) \geq 0$.

The Ford-Fulkerson method

Algorithm 21 Ford-Fulkerson method

```
Initialize, set f(e) = 0, ∀e ∈ E, G_f = G
while ∃P = s → t ∈ G_f such that ∀e ∈ E, c_f(e) ≥ 0 do
    δ(P) = min_{e∈P} c_f(e)
    for ∀(u, v) ∈ P do
        if f(u, v) > 0 then
            f(v, u) = f(v, u) − δ(P)
        else
            f(u, v) = f(u, v) + δ(P)
    Update G_f
```

Some implementations

- Edmonds-Karp algorithm. Runs in $O(VE^2)$. The augmenting path is the shortest path found by breadth-first search.
- Dinic's algorithm. Runs in $O(V^2E)$. The augmenting path is the shortest path found by combining BFS and DFS.
- Capacity scaling. Use heuristic to find the augmenting path that has the largest flow.

6.4 Minimum Cost Flow Problem

6.4.1 Transshipment Problem

Transshipment Problem (D, b, w) is a linear program of the form

$$\begin{aligned} \min \quad & wx \\ \text{s.t.} \quad & Nx = b \\ & x \geq 0 \end{aligned}$$

Where N is a vertex-arc incident matrix. For a feasible solution to LP to exist, the sum of all bs must be zero. Since the summation of rows of N is zero. The interpretation of the LP is as follows.

The variables are defined on the edges of the digraph and that x_e denote the amount of flow of some commodity from the tail of e to the head of e

Each constraints

$$\sum_{h(e)=i} x_e - \sum_{t(e)=i} x_e = b_i$$

represents consequential of flow of all edges into k vertex that have a demand of $b_i > 0$, or a supply of $b_i < 0$. If $b_i = 0$ we call that vertex a transshipment vertex.

6.4.2 Network Simplex Method

Lemma 6.4.1. *Let C_1 and C_2 be distinct cycles in a graph G and let $e \in C_1 \cup C_2$. Then $(C_1 \cup C_2) \setminus e$ contains a cycle.*

Proof. Case 1: $C_1 \cap C_2 = \emptyset$. Trivia.
Case 2: $C_1 \cap C_2 \neq \emptyset$. Let $e \in C_2$ and $f = uv \in C_1 \setminus C_2$. Starting at v traverse C_1 in the direction away from u until the first vertex of C_2, say x. Denote the (v, x)-path as P. Starting at u traverse C_1 in the direction away from v until the first vertex of C_2, say y. Denote the (u, y)-path as Q. C_2 is a cycle, there are two (x, y)-path in C_2. Denote the (x, y)-path without e as R. Then $vPxRyQ^{-1}uf$ is a cycle. □

Theorem 6.4.2. *Let T be a spanning tree of G. And let $e \in E \setminus T$ then $T + e$ contains a unique cycle C and for any edge $f \in C$, $T + e - f$ is a spanning tree of G*

Let (D, b, w) be a transshipment problem. A feasible solutions x is a **feasible tree solution** if there is a spanning tree T such that $||x|| = \{e \in A, x_e \neq 0\} \subseteq T$.

The strategy of network simplex algorithm is to generate negative cycles, if negative cycle exists, it means the solution can be improved.

For any tree T of D and for $e \in A \setminus T$, it follows from above theorem that $T + e$ contains a unique cycle. Denote that cycle $C(T, e)$ and orient it in the direction of e, define

$$\begin{aligned} w(T, e) = &\sum \{w_e : e \text{ forward in } C(T, e)\} \\ &- \sum \{w_e : e \text{ reverse in } C(T, e)\} \end{aligned} \tag{6.4}$$

We think of $w(T, e)$ as the weight of $C(T, e)$.

Network Simplex Method The following algorithm describes the procedure of using simplex method to solve the transshipment problem

Algorithm 22 Network Simplex Method Algorithm

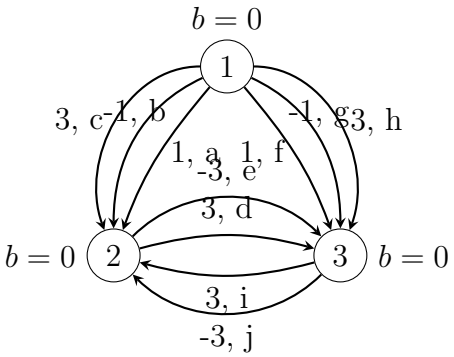
Ensure: An optimal solution or the conclusion that (D, b, w) is unbounded

Require: A transshipment problem (D, b, w) and a feasible tree solution x containing to a spanning tree T

```
while ∃e ∈ A \ T, w(T, e) < 0 do
    let e ∈ A \ T be such that w(T, e) < 0.
    if C(T, e) has no reverse arcs then
        Return unboundness
    else
        Set θ = min{x_f : f reverse in C(T, e)}
        Set f = {f ∈ C(T, e) : f reverse in C(T, e), x_f = θ}
        if f forward in C(T, e) then
            x_f ← x_f + θ
        else
            x_f ← x_f − θ
        Let f ∈ F and T ← T + e − f
Return x as optimal
```

Example for cycling Similar to Simplex Method in LP, there could be cycling problems.

The following is an example of cycling



Then for the following steps we can detect cycling:

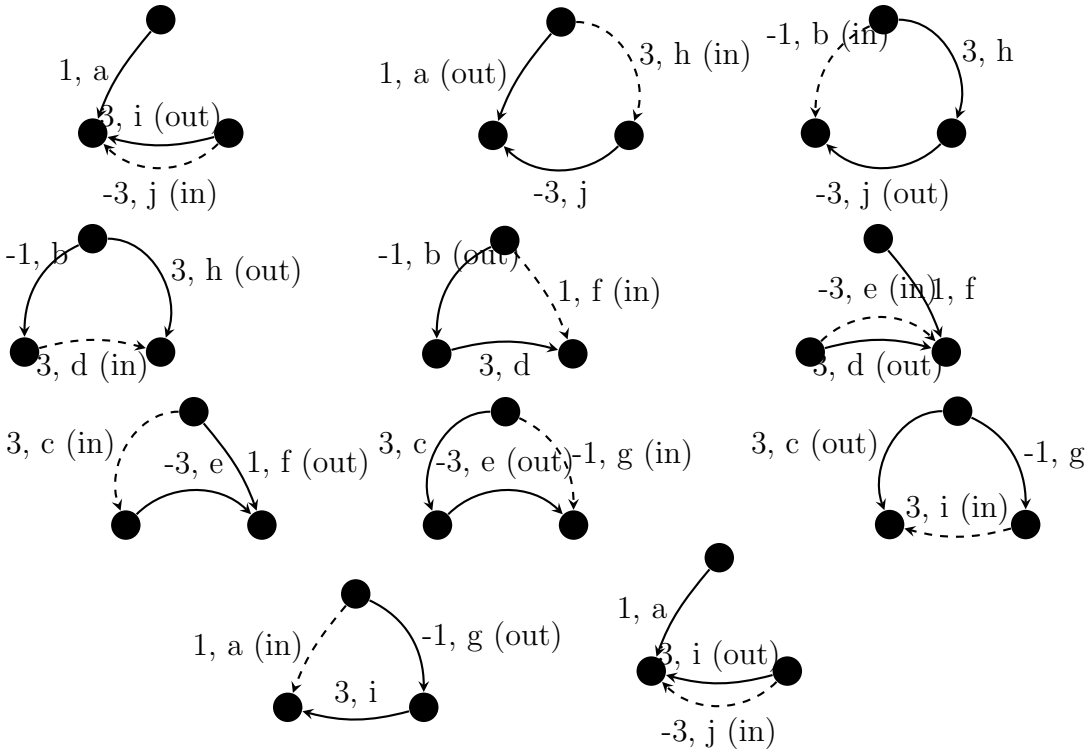
- $w(T, j) = w_j - w_i = -3 - 3 = -6$, therefore j in entering basis, i is leaving basis.
- $w(T, h) = w_h + w_j - w_a = 3 - 3 - 1 = -1$, therefore h is entering basis, a is leaving basis.
- $w(T, b) = w_b - w_j - w_h = -1 + 3 - 3 = -1$, therefore b is entering basis, j is leaving basis.
- $w(T, d) = w_d - w_h + w_b = 3 - 3 - 1 = -1$, therefore d is entering basis, h is leaving basis.
- $w(T, f) = w_f - w_d - w_b = 1 - 3 + 1 = -1$, therefore f is entering basis, b is leaving basis.
- $w(T, e) = w_e - w_d = -3 - 3 = -6$, therefore e is entering basis, d is leaving basis.
- $w(T, c) = w_c + w_e - w_f = 3 - 3 - 1 = -1$, therefore c is entering basis, f is leaving basis.
- $w(T, g) = w_g - w_e - w_c = -1 + 3 - 3 = -1$, therefore g is entering basis, e is leaving basis.
- $w(T, i) = w_i - w_c + w_g = 3 - 3 - 1 = -1$, therefore i is entering basis, c is leaving basis.
- $w(T, a) = w_a - w_i - w_g = 1 - 3 + 1 = -1$, therefore a is entering basis, g is leaving basis.

The last graph is the same as the first graph, i.e., cycling detected.

Cycling prevention To Avoid cycling we will introduce the Modified Network Simplex Method. Let T be a **rooted** spanning tree. Let f be an arc in T, we say f is **away** from the root r if $t(f)$ is the component of $T - f$. Otherwise we say f is **towards** r.

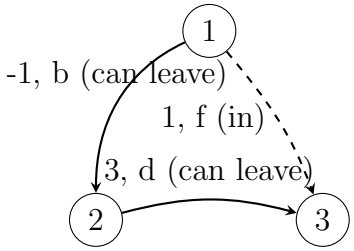
Let x be a feasible tree solution associated with T, then we say T is a **strong feasible tree** if for every arc $f \in T$ with $x_f = 0$ then f is away from $r \in T$.

Modification to NSM:

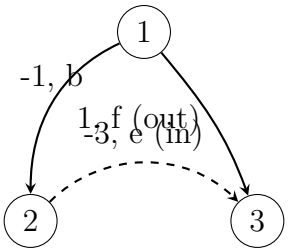


- The algorithm is initialed with a strong feasible tree.
- f in pivot phase is chosen to be the first reverse arc of $C(T, e)$ having $x_f = \theta$. By "first", we mean the first arc encountered in traversing $C(T, e)$ in the direction of e, starting at the vertex i of $C(T, e)$ that minimizes the number of arcs in the unique (r, i)-path in T.

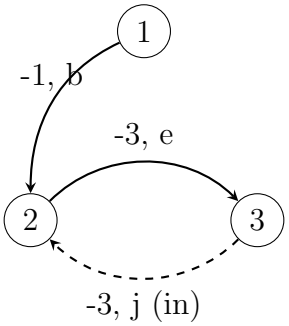
In the second rule above, r could also be in the cycle, in that case, i is r.
Continue the previous example. Now should how we can avoid cycling:
The first few (four) steps are the same as previous example, starting from



$w(T, f) = w_f - w_d - w_b = 1 - 3 + 1 = -1$. f is entering basis, both b and d can leave the basis, according to the modified pivot rule, we choose the "first" arc encountered in traversing $C(T, e)$, which is d to leave the basis, instead of b.



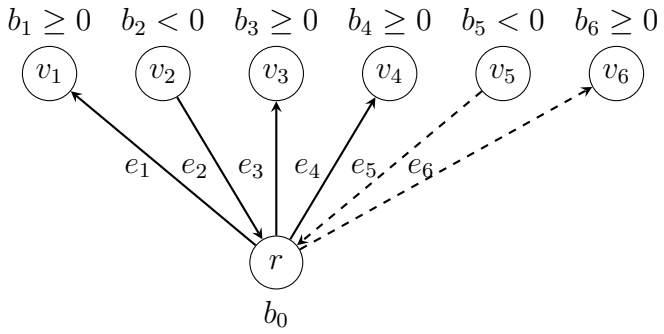
$w(T, e) = w_e - w_f + w_b = -5$, e is entering basis, f is leaving basis. Now the only arc to enter basis and maintain negative w is j.



$w(T, j) = w_j + w_e = -6$, but in $C(T, j)$ there is no reversing arc, therefore we detect unboundness.

Finding Initial Strong Feasible Tree Pick a vertex in D to be root r. The tree T has an arc e with the $t(e) = r$ and $h(e) = v$. For each $v \in V \setminus r$ with $b_v \geq 0$ and has an arc e with $h(e) = r$ and $t(e) = v$ for each $v \in V \setminus r$ for which $b_v < 0$. Wherever possible the arcs of T are chosen from A, where an appropriate arc doesn't exist. We create an **artificial arc** and give its weight $|V|(\max\{w_e : e \in A\}) + 1$. This is similar to Big-M method and if optimal solution contains artificial arcs ongoing arc problem is infeasible.

Here is an example after adding artificial arcs:



Where e_5 and e_6 are artificial arcs, the weight of those arcs are $|V|(\max\{w_e : e \in \mathcal{A}\}) + 1$. And the above tree is a basic feasible solution.

We need to prove that such artificial arc has sufficiently large weight to guarantee

- It will leave the basis, and
- It will not enter the basis again (for this, just delete the artificial arc after it leaves the basis, then it will never enter the basis again)

Proof. Now prove that such arcs will always leave the basis. Before the prove we give some notation.

- Define set E as the set of arcs which is not artificial arc, in the above example, $E = \{e_1, e_2, e_3, e_4\}$.
- Define set A as the set of arcs which are artificial arcs, in the above example, $A = \{e_5, e_6\}$. Noticed that $E \cap A = \emptyset$.
- Define set M as the vertices in the spanning tree that is reachable from r by E, in the above example, $M = \{v_1, v_2, v_3, v_4\}$.
- Define $M' = (V \setminus r) \setminus M$ in the tree that can only be reached from r by A, i.e., artificial arcs, in the above example, $M' = \{v_5, v_6\}$.

Then the initial basic feasible solution is a graph

$$G_0 =< M \cup M' \cup \{r\}, E \cup A >$$

Denote the origin graph

$$G =< V, \mathcal{A} >$$

Notice that with the artificial arcs, G_0 is not a subgraph of G.

Let $(M \cup \{r\}, M')$ be a cut in the origin graph G. For the vertices in M', one of the following cases will happen:

- case 1: $\sum_{v \in M'} b_v \geq 0$
- case 2: $\sum_{v \in M'} b_v < 0$

For case 1, we claim that at least one of the vertices $v_{M'} \in M'$ with $b_{v_{M'}} \geq 0$ linked by an arc, say f, such that $h(f) = v_{M'}$ and $t(f) = v_M \in M$. Otherwise the balance of flow cannot hold in the origin graph G. Furthermore, denote the artificial arc from r to $v_{M'}$ by $e_{rv_{M'}}$.

Notice that for v_M there is not necessarily be an arc between r and v_M, but there must exists an (r, v_M)-path denoted by P, for M is the set of vertices that reachable from r by arcs in E.

Take that arc f as entering arc to the basis. Then

$$C(T, f) = re_{rv_{M'}} v_{M'} f v_M P r$$

For

$$w(T, f) = w_f - w_{e_{rv_{M'}}} + \sum_{e \in P} d_e w_e$$

where $d_e = 1$ if w_e is forward in P and $d_e = -1$ otherwise.

Now that $w_{e_{rv_{M'}}} = |V|(\max\{w_e : e \in \mathcal{A}\}) + 1$, it guarantees that

$$\begin{aligned} w(T, f) &= w_f + \sum_{e \in P} d_e w_e - w_{e_{rv_{M'}}} \\ &\leq w_f + \sum_{e \in P} w_e - w_{e_{rv_{M'}}} \\ &\leq \sum_{e \in \mathcal{A}} w_e - w_{e_{rv_{M'}}} \\ &\leq |V|(\max\{w_e : e \in \mathcal{A}\}) - w_{e_{rv_{M'}}} \\ &\leq -1 < 0 \end{aligned}$$

So f can enter the basis, and the artificial variable $e_{rv_{M'}}$ will leave the basis, for it is the most violated reverse arc in the $C(T, f)$. When we put f into the basis, update G_0, such that $M \leftarrow M \cup \{v_{M'}\}$ and $M' \leftarrow M' \setminus \{v_{M'}\}$.

For case 2, it is similar. At least one of the vertices $v_{M'} \in M'$ with $b_{v_{M'}} < 0$ linked by an arc, say $f\prime$, such that $t(f') = v_{M'}$ and $h(f') = v_M \in M$. Otherwise the balance of flow cannot hold in the origin graph G. Furthermore, denote the artificial arc from $v_{M'}$ to r by $e_{v_{M'}r}$.

Similarly we can find a cycle $C(T, f') = rP'v_M f' v_{M'} e_{v_{M'}r} r$. $w(T, f') = w_{f'} - w_{e_{rv_{M'}}} + \sum_{e \in P'} d_e w_e$, where $d_e = 1$ if w_e is forward in $P\prime$ and $d_e = -1$. We can prove $w(T, f') \leq -1 < 0$. That that f' as entering arc to the basis, similarly move $v_{M'}$ form set M' to M.

The above case can be dealt with iteratively until set M' become $\emptyset$, at which stage there is no artificial arc in the basic feasible solution. Which means all the artificial variable can leave the basis. □

Chapter 7

Traveling Salesman Problem

"O never go back."

7.1 The Traveling Salesman Problem

In this section, we are going to compare between different formulations of the Traveling Salesman Problem (TSP). Generally speaking, let $G = (V, A)$ be a graph where V is a set of n vertices, and A is a set of arcs (or edges). Let $C = c_{ij}$ be a cost (distance) matrix associated with A. The TSP consists of determining a minimum cost (distance) Hamiltonian circle (or cycle) that visits each vertex once and only once. If for all $i, j \in V, c_{ij} = c_{ji}$, then the TSP is symmetrical, otherwise is asymmetrical.

Define the decision variable x_{ij} as the following

$$x_{ij} = \begin{cases} 1, & \text{if goes from } i \text{ to } j \\ 0, & \text{otherwise} \end{cases}, \quad (i,j) \in A \tag{7.1}$$

The objective function is

$$\min \quad \sum_{(i,j)\in A} c_{ij} x_{ij} \tag{7.2}$$

7.1.1 Dantzig-Fulkerson-Johnson (DFJ) Formulation

The first famous formulations for TSP is the **Dantzig-Fulkerson-Johnson (DFJ) formulation**:

$$\sum_{j\in V,(i,j)\in A} x_{ij} = 1, \quad \forall i \in V \tag{7.3}$$

$$\sum_{i\in V,(i,j)\in A} x_{ij} = 1, \quad \forall j \in V \tag{7.4}$$

$$\sum_{j\notin S, i\in S,(i,j)\in A} x_{ij} \geq 1, \quad \forall S \subset V, 2 \leq |S| \leq n-1 \tag{7.5}$$

In the formulation, constraints (7.3) and constraints (7.4) are degree constraints, which specify that every vertex is entered exactly once. Constraints (7.5) is the sub-tour constraints, they prohibit the formation of sub-tours. S is a non-empty subset of V, and has at least 2 vertices. (7.5) can be replaced by

$$\sum_{i,j \in S,(i,j) \in A} x_{ij} \leq |S| - 1, \quad \forall S \subset V, 2 \leq |S| \leq n-1 \tag{7.6}$$

If we list all sub-tour constraints in DFJ, there will be $O(2^n)$ constraints and $O(n^2)$ binary variables. The exponential number of constraints makes it impractical to solve directly. Instead, lazy constraints are usually implemented for the sub-tour elimination constraints (7.5) or (7.6).

7.1.2 Miller-Tucker-Zemlin (MTZ) Formulation

We can also formulate TSP using sequential formulations, namely, **Miller-Tucker-Zemlin (MTZ) formulation**. In the MTZ formulation, the degree constraints (7.3) and (7.4) are the same as in DFJ formulation.

Define a new set of integer decision variables u_i, u_i defined as the sequence in which node i is visited, $u_1 = 1$.

The sub-tour constraints (7.5) or (7.6) are replaced by the following:

$$u_i - u_j + (n-1)x_{ij} \leq n-2, \quad i,j = 2, \cdots, n \in V, (i,j) \in A \tag{7.7}$$

$$1 \leq u_i \leq n-1, \quad i \in 2, \cdots, n \in V \tag{7.8}$$

In MTZ formulation, there are $O(n^2)$ constraints, $O(n^2)$ binary variables, and $O(n)$ continuous variables.

7.1.3 Flow Based Formulations

In this section, flow based formulations are discussed, which includes **Single Commodity Flow**, **Two Commodity Flow** and **Multi-Commodity Flow**. In these formulations, continuous variables are introduced to represent the flow on the arcs.

In Single Commodity Flow formulation, define y_{ij} as the flow in an arc $(i,j) \in A$. Degree constraints (7.3) and (7.4) are retained. The following constraints are introduced:

$$y_{ij} \leq (n-1)x_{ij}, \quad \forall i,j \in V, (i,j) \in A \tag{7.9}$$

$$\sum_{j \in V,(1,j) \in A} y_{1j} = n-1 \tag{7.10}$$

$$\sum_{i \in V,(i,j) \in A} y_{ij} - \sum_{k \in V,(j,k) \in A} y_{jk} = 1, \quad \forall j \in V \setminus \{1\} \tag{7.11}$$

Constraints (7.9) can be tighten by the following:

$$y_{ij} \leq (n-1)x_{ij}, \quad i = 1, j \in V \setminus \{1\}, (i,j) \in A \tag{7.12}$$
$$y_{ij} \leq (n-2)x_{ij}, \quad i, j \in V \setminus \{1\}, (i,j) \in A \tag{7.13}$$

In SCM formulation, there are $O(n^2)$ constraints, $O(n^2)$ binary variables and $O(n^2)$ continuous variables.

In Two Commodity Flow formulation, define y_{ij} as the flow in an arc $(i,j) \in A$, for commodity type 1, and define z_{ij} as the flow in an arc $(i,j) \in A$, for commodity type 2.

Besides degree constraints, the other constraints are as following

$$y_{ij} + z_{ij} = (n-1)x_{ij}, \quad \forall i, j \in V, (i,j) \in A \tag{7.14}$$
$$\sum_{j \in V \setminus \{1\}} (y_{1j} - y_{j1}) = n - 1, \quad (1,j) \in A \tag{7.15}$$
$$\sum_{j \in V} (y_{ij} - y_{ji}) = 1, \quad \forall i \in V \setminus \{1\}, (i,j) \in A \tag{7.16}$$
$$\sum_{j \in V \setminus \{1\}} (z_{1j} - z_{j1}) = 1 - n, \quad (1,j) \in A \tag{7.17}$$
$$\sum_{j \in V} (z_{ij} - z_{ji}) = -1, \quad \forall i \in V \setminus \{1\}, (i,j) \in A \tag{7.18}$$
$$\sum_{j \in V} (y_{ij} + z_{ij}) = n - 1, \quad \forall i \in V \tag{7.19}$$

In TCM formulation, constraints (7.14) only allow flow in an arc if present. Constraints (7.15) and (7.16) forces $(n-1)$ units of commodity type 1 to flow in at node 1 and 1 unit to flow out at every other nodes. Constraints (7.17) and (7.18) are similar, those forces $(n-1)$ units of commodity type 2 to flow out at node 1 and 1 unit to flow in at every other nodes. Constraints (7.19) forces exactly $(n-1)$ units of combined commodity in each arc.

In TCM formulation, there are $O(n^2)$ constraints, $O(n^2)$ binary variables and $O(n^2)$ continuous variables.

The SCM and the TCM can be generalized into **Multi-Commodity Flow formulation**. As usual, degree constraints are retained. The following continuous variables are introduced. Define y_{ij}^k as the flow of commodity type k in arc $(i,j) \in A$.

The other constraints are

$$y_{ij}^k \leq x_{ij}, \quad \forall i, j, k \in N, k \neq 1 \tag{7.20}$$
$$\sum_{i \in V} y_{1i}^k = 1, \quad \forall k \in V \setminus \{1\} \tag{7.21}$$
$$\sum_{i \in V} y_{i1}^k = 0, \quad \forall k \in V \setminus \{1\} \tag{7.22}$$
$$\sum_{i \in V} y_{ik}^k = 1, \quad \forall k \in V \setminus \{1\} \tag{7.23}$$

$$\sum_{j \in V} y_{kj}^k = 0, \quad \forall k \in V \setminus \{1\} \tag{7.24}$$

$$\sum_{i \in V} y_{ij}^k - \sum_{i \in V} y_{ji}^k = 0, \quad \forall j, k \in V \setminus \{1\}, j \neq k \tag{7.25}$$

Constraints (7.20) only allow flow in an arc which is present. Constraints (7.21) forces exactly one unit of each type of commodity to flow in at node 1. Constraints (7.22) prevent any commodity flow out at node 1.Constraints (7.23), and Constraints (7.24), forces exactly one unit of type k commodity to flow out, and in, at every node except node 1. Constraints (7.25) forces balance of all types of commodities at every node except node 1.

This formulation has $O(n^3)$ constraints, $O(n^2)$ binary variables, and $O(n^3)$ continuous variables.

7.1.4 Shortest Path Formulation

In this section, we are going to introduce another form of formulation with different definition of decision variable and objective function.

Assuming for a completed graph $G = (V, A)$. Define x_{ij}^t as the following

$$x_{ij}^t = \begin{cases} 1, & \text{If path crosses arc } (i,t) \text{ and } (j,t+1) \\ 0, & \text{Otherwise} \end{cases}, \quad i \in V, j \in V \setminus \{i\}, t = 1, \cdots, n \tag{7.26}$$

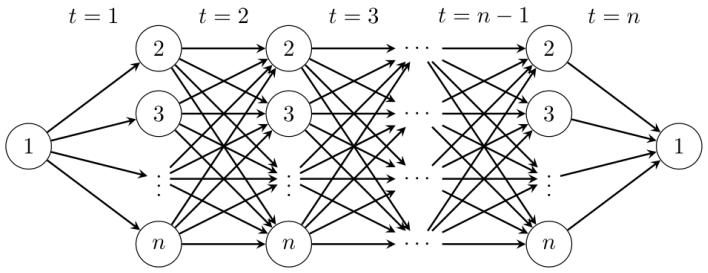


Figure 7.1: Time-staged graph

The objective function will be

$$\min \quad \sum_{i \in V} \sum_{j \in V \setminus \{i\}} c_{ij} \sum_{t=1}^{n} x_{ij}^t \tag{7.27}$$

The constraints are as following

$$\sum_{j \in V \setminus \{1\}} x_{1j}^1 = 1 \tag{7.28}$$

$$\sum_{j \in V \setminus \{1,i\}} x_{ij}^2 - x_{1i}^1 = 0, \quad \forall i \in V \setminus \{1\} \tag{7.29}$$

$$\sum_{j\in V\setminus\{1,i\}} x_{ij}^t - \sum_{j\in V\setminus\{1,i\}} x_{ji}^{t-1} = 0, \quad \forall i \in V\setminus\{1\}, t \in \{2,\dots,n-1\} \tag{7.30}$$

$$x_{i1}^n - \sum_{j\in V\setminus\{1,i\}} x_{ji}^{n-1} = 0, \quad \forall i \in V\setminus\{1\} \tag{7.31}$$

$$\sum_{i\in V\setminus\{1\}} x_{i1}^n = 1 \tag{7.32}$$

$$\sum_{t=2}^{n-1} \sum_{j\in V\setminus\{1,i\}} x_{ij}^t + x_{i1}^n \le 1, \quad \forall i \in V\setminus\{1\} \tag{7.33}$$

Notice that constraint (7.33) can be replaced by

$$x_{1i}^1 + \sum_{t=2}^{n-1} \sum_{j\in V\setminus\{1,i\}} x_{ji}^t \le 1, \quad \forall i \in V\setminus\{1\} \tag{7.34}$$

7.1.5 Quadratic Formulation (QAP)

In this section, we are going to go over a TSP formulation are super bad. However, it still has some value for further study.

The idea is to transform TSP into an assignment problem. Assuming we have n boxes, which represents n steps in the path. Define x_{ij} as

$$x_{ij} = \begin{cases} 1, & \text{Vertex } i \text{ is assigned to box } j \\ 0, & \text{Otherwise} \end{cases} \tag{7.35}$$

The constraints are simple as an assignment problem as following

$$\sum_{j=1}^{n} x_{ij} = 1, \quad \forall i \in V \tag{7.36}$$

$$\sum_{i\in V}^{n} x_{ij} = 1, \quad j = 1,\dots,n \tag{7.37}$$

However, the tricky part is in the objective function

$$\min \quad \sum_{i\in V} \sum_{j\in V\setminus\{i\}} \sum_{k=1}^{n-1} c_{ij} x_{ik} x_{j,k+1} + \sum_{i\in V} \sum_{j\in V\setminus\{i\}} c_{ij} x_{in} x_{j1} \tag{7.38}$$

Notice that the objective function is not linear function, with the multiplications of decision variables. Now we are going to linearize them. The linearized version is as following

$$\min \quad \sum_{i\in V} \sum_{j\in V\setminus\{i\}} \sum_{k=1}^{n-1} c_{ij} w_{ij}^k + \sum_{i\in V} \sum_{j\in V\setminus\{i\}} c_{ij} w_{ij}^n \tag{7.39}$$

$$\text{s.t.} \quad \sum_{j=1}^{n} x_{ij} = 1, \quad \forall i \in V \tag{7.40}$$

$$\sum_{i \in V}^{n} x_{ij} = 1, \quad j = 1, \ldots, n \tag{7.41}$$

$$w_{ij}^{k} \geq x_{ik} + x_{j,k+1} - 1, \quad i \in V, j \in V \setminus \{i\}, k = 1, \cdots, n-1 \tag{7.42}$$

$$w_{ij}^{k} \geq x_{ik} + x_{j1} - 1, \quad i \in V, j \in V \setminus \{i\}, k = n \tag{7.43}$$

$$w_{ij}^{k} \in \{0, 1\}, \quad i \in V, j \in V \setminus \{i\}, k = 1, \ldots, n \tag{7.44}$$

$$x_{ij} \in \{0, 1\}, \quad i \in V, j \in V \setminus \{i\} \tag{7.45}$$

We can prove that this is very very bad. The optimal solution of the LP Relaxation for the QAP formulation is as follows

$$x_{ij} = \frac{1}{n}, \quad \forall i, j \in V \tag{7.46}$$

$$w_{ij}^{k} = \frac{2}{n} - 1, \quad \forall i \in V, j \in V \setminus \{i\}, k = 1, 2, \ldots, n \tag{7.47}$$

The solution indicates that all decision variables are symmetric in the LP Relaxation, thus, it did not provide any information for branching. In fact, such formulation will search all $O(2^n)$ branches and the lower bound will be difficult to converge.

7.1.6 Numerical Comparisons

The following results are on Intel(R) Core(TM) i7-8750H CPU @ 2.20GHz, 2208 Mhz. with Gurobi version 9.3.0. Using this processor, DFJ, MTZ, multi-commodity flow methods can solve instances up to 70 nodes within 10 minutes. Shortest path formulation can solve to optimality for less than 50 nodes within 10 minutes. QAP has the worse performances, which cannot even solve TSP with 15 nodes to optimality within 10 minutes. Here are several observations

- In general DFJ formulation is the best formulation between DFJ, MTZ, multi-commodity flow, shortest path, and QAP formulations. Both plain loop and lazy cut methods can solve instances that we are studying within 1 second. Which is significantly better than others.
- As per run time, we can sort the performance of formulations as following: DFJ (Lazy cut) < DFJ (Plain Loop) < MTZ < multi-commodity flow < shortest path < QAP.

7.2 The Held and Karp Lower Bound

In this section, we will solve the Dantzig-Fulkerson-Johnson formulation using Lagrangian Relaxation. Before finally converge, the LR finds an infeasible solution as lower bound. The bound found by this method is also known as Held & Karp Bound.

7.2.1 1-Tree

We first introduce an intuitive lower bound for TSP, which is the Minimum Spanning Tree Lower Bound. As we known, an optimal solution for the TSP is a Hamilton Cycle which enumerated all vertices on the graph. The length of such Hamilton cycle is denoted as TSP^*. If we randomly remove one of the edges, e.g., $\{vw\}$, then, the cycle becomes a path, denoted by $TSP - \{vw\}$. In this case, the degree of vertices v and w reduce to 1 while all the other vertices remains the same as 2. By definition, such path $TSP - \{vw\}$ is a spanning tree on graph G. Therefore, the minimum spanning tree of graph G defines a lower bound.

$$MST \leq \text{spanning tree} = TSP - \{vw\} < TSP^*$$

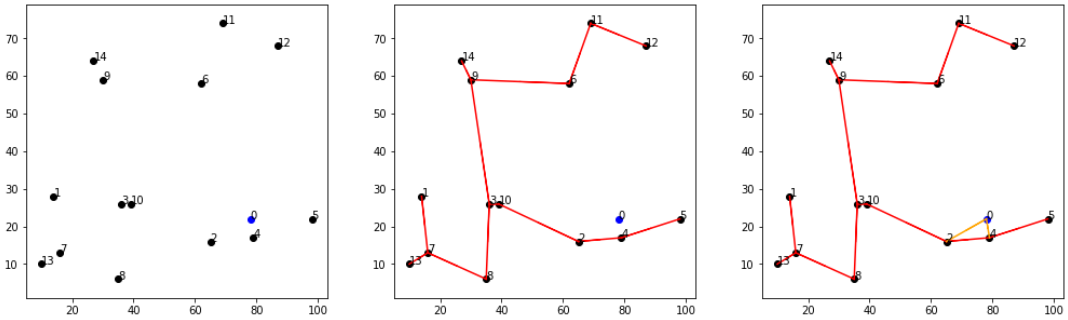
We can further improve the lower bound by introducing 1-tree. A spanning tree is a tree with no cycle, if the graph has 1 cycle, it is called 1-tree. Notice that a Hamilton Cycle has only 1 cycle, thus, the Hamilton Cycle is an 1-tree as well. Naturally, the length of all edges of any 1-tree is larger than the length of all edges of the minimum spanning tree. The lower bound of TSP is further improved as follows

$$MST < M1T \leq 1 - tree \leq TSP^*$$

The following algorithm defines an 1-Tree on graph G corresponding to vertex v.

Algorithm 23 1-Tree

Require: A connected graph
Ensure: A minimum 1-Tree
Remove v from the graph
Use Kroskal's algorithm or Prim's Algorithm to find the minimum spanning tree of $G \setminus \{v\}$
Find the shortest edge induced by v to the rest of graph $G \setminus \{v\}$, denoted by vu and vw, add them to the MST
return 1-Tree



(a) Remove v (b) Find MST (c) Add back v

Figure 7.2: Steps in finding minimum 1-Tree

A M1T is a good lower bound, however, we can still further improve the lower bound by finding 1-trees.

7.2.2 Held and Karp Lower Bound and Lagrangian Relaxation

Notice that in the minimum 1-tree example, vertices have different degrees, some are of degree 1, 2, 3, or more. However, the "optimal" 1-tree that we look for, is an 1-tree such that all the vertices are of degree 2. By intuition, if an 1-tree has fewer "branches", it's closer to be "optimal". That is actually the principle of the Held and Karp Lower Bound.

We first look at the DFJ formulation for the TSP.

$$\min \quad \sum_{e \in E} \tau_e x_e \tag{7.48}$$

$$\text{s.t.} \quad \sum_{e \in \delta(i)} x_e = 2, \quad \forall i \in 1, 2, \ldots, n \tag{7.49}$$

$$\sum_{e \in E(S)} x_e \leq |S| - 1, \quad \forall S \subset V, 2 \leq |S| \leq n - 1 \tag{7.50}$$

$$x_e \in \{0, 1\}, \quad \forall e \in E \tag{7.51}$$

In the DFJ model, the objective is to sum up all the cost of the edges that are chosen on the TSP path. Constraints (7.49) is a flow balancing constraint, the index $e \in \delta(i)$ represents all the edges that are induced by vertex i. Constraints (7.50) is the sub-tour constraint.

Replace the Constraints (7.49) by the following

$$\sum_{e \in \delta(i)} x_e = 2, \quad \forall i \in 1, 2, \ldots, n - 1 \tag{7.52}$$

$$\sum_{e \in E} x_e = n \tag{7.53}$$

We can reformulate the DFJ formulation as follows:

$$\min \quad \sum_{e \in E} \tau_e x_e \tag{7.54}$$

$$\text{s.t.} \quad \sum_{e \in \delta(i)} x_e = 2, \quad \forall i \in 1, 2, \ldots, n - 1 \tag{7.55}$$

$$\sum_{e \in E} x_e = n \tag{7.56}$$

$$\sum_{e \in E(S)} x_e \leq |S| - 1, \quad \forall S \subset V, 2 \leq |S| \leq n - 1 \tag{7.57}$$

$$x_e \in \{0, 1\}, \quad \forall e \in E \tag{7.58}$$

Look closely, Constraints (7.56) means, the number of edges in the subgraph should be the same as the number of vertices, so the subgraph has 1 cycle. The Constraints (7.57) guarantee that the graph is connected. Which means, Constraints (7.56) and (7.57) finds an 1-tree.

Until now, we can move the Constraints (7.55) to the objective function, and use the Lagrangian Relaxation to solve the problem

$$z(\mathbf{u}) = \min \quad \sum_{e \in E} \tau_e x_e + \sum_{i=1}^{n-1} u_i (2 - \sum_{e \in \delta(i)} x_e) \tag{7.59}$$

$$\text{s.t.} \quad \mathbf{x} \text{ defines an 1-tree with vertice } n \tag{7.60}$$

For the vertex i, define u_i, the formulation can be further rewritten as

$$z(\mathbf{u}) = \min \quad 2\sum_{i=1}^{n-1} u_i + \sum_{e \in \delta(i)} (\tau_e - u_{e^+} - u_{e^-}) x_e \tag{7.61}$$

$$\text{s.t.} \quad \mathbf{x} \text{ defines an 1-tree with vertice } n \tag{7.62}$$

7.2.3 Subgradient Descendant Method

Notice that in the previous section, we derived a function $z(\mathbf{u})$ of Lagrangian scalar u_i as the lower bound of the TSP. The solution space of $\mathbf{u}$ is a convex space, in this section, we will search the $\max_{\mathbf{u}} z(\mathbf{u})$ by subgradient descendant method.

Algorithm 24 Subgradient descendant method for Held-Karp Lower Bound

Require: A connected graph
Ensure: A lower bound of TSP
Initialize, for each vertex i, let $u_i = 0$, $d_i = \emptyset$. Define lower bound $L \leftarrow 0$, $L' \leftarrow 0$
while do$|L - L'| \geq \epsilon$
Update weights of edges $\tau_{ij} = \tau_{ij} - u_i - u_j$
Find the M1T on the graph G with edges updated. Let D be the summation of the length of all edges, let d_i be the degree of vertex i on the M1T.
Update $u_i \leftarrow u_i + \lambda_i \frac{UB - D}{\sum_i (d_i - 2)^2}$
Update $L' \leftarrow L$
Update $L \leftarrow D + 2\sum_i u_i$
return L

In the algorithm, different descendant function may be applied, for example, an easier way for implementation is to update

$$u_i \leftarrow u_i + (d_i - 2)\rho^k$$

Chapter 8

Column Generation

"Stepping bravely into the unknown."

8.1 Fundamental Idea of the Column Generation

We first introduce an intuitive approach based on the Revised Simplex Method. Recall the Minkowski-Weyl Theorem

Theorem 8.1.1 (Minkowski-Weyl Theorem)**.** *Let $P = \{x \in \mathbb{R}^n | Ax \leq b\}$ be a polyhedron with extreme points $V(P) = \{\mathbf{v}_1, \mathbf{v}_2, \cdots, \mathbf{v}_{|V|}\}$ and extreme directions $R(P) = \{\mathbf{r}_1, \mathbf{r}_2, \cdots, \mathbf{r}_{|R|}\}$, then*

$$\begin{aligned} P = \{x \in \mathbb{R}^n | x = & \sum_{j=1}^{|V|} \lambda_j \mathbf{v}_j + \sum_{j=1}^{|R|} \mu_j \mathbf{r}_j \\ & \sum_{j=1}^{|V|} \lambda_j = 1, \lambda_j \geq 0, \quad \forall j = 1, 2, \cdots, |V| \\ & \mu_j \geq 0, \quad \forall j = 1, 2, \cdots, |R|\} \end{aligned}$$

With out lost of generality, a (bounded) linear problem model can be formulated as follows

$$\begin{aligned} \text{(MP)} \quad z^*_{MP} = \min \quad & \sum_{j \in J} c_j \lambda_j \\ \text{s.t.} \quad & \sum_{j \in J} \mathbf{a}_j \lambda_j \geq \mathbf{b} \quad [\pi] \\ & \lambda_j \geq 0, \quad \forall j \in J \end{aligned}$$

Where each λ_j represents an extreme point of the polyhedron. We call this formulation the master problem (MP). The Revise Simplex Method allows us to solve the LP with a **large** $|J|$, which is, in each iteration add one more nonbasic variable with negative reduce

cost into the formulation. Since we do not need to consider all variables in the beginning, we can start with solving a restricted version of the master problem with only a subset of J (in fact, we can start the iteration with a B^{-1}, which is any basic feasible solution). That is

$$\begin{aligned} \text{(RMP)} \quad z^*_{RMP} = \min \quad & \sum_{j \in J'} c_j \lambda_j \\ \text{s.t.} \quad & \sum_{j \in J'} \mathbf{a}_j \lambda_j \geq \mathbf{b} \quad [\pi] \\ & \lambda_j \geq 0, \quad \forall j \in J' \subset J \end{aligned}$$

We call this formulation the restricted master problem (RMP). In the RMP, we have less choices than the MP, therefore, as a minimization problem,

$$z^*_{RMP} \geq z^*_{MP}$$

provides an upper bound for MP. In each iteration, a new variable - column - is introduced into the formulation, thus, this approach is called the Column Generation.

Algorithm 25 Column generation with explicit pricing

Initialize with a set $J' \subseteq J$ of variables
repeat
  Solve RMP to optimality, obtain λ and π
  Calculate $\bar{c}_j = z_j - c_j = \pi^\top \mathbf{a_j} - c_j, \forall j \in J$
  if $\exists \lambda_{j^*}$ with $z_{j^*} - c_{j^*} < 0$ **then**
    $J' \leftarrow J' \cup \{j^*\}$
until $\forall \lambda_j, j \in J, z_j - c_j \geq 0$

Notice that if $|J|$ is **huge**, it will be just too costly to calculate the reduce cost for *every* nonbasic variable, sometimes (in fact, most of the times) we are not even able to enumerate all nonbasic variables, as the size of J is usually exponentially large. Thus, an optimization problem is introduced to find the least-reduce-cost nonbasic variable, to replace the explicit pricing of all nonbasic variables, we call it the *pricing subproblem*. The updated Column Generation algorithm is as follows

Algorithm 26 Column generation with pricing subproblem

Initialize with a set $J' \subseteq J$ of variables
repeat
  Solve RMP to optimality, obtain λ and π
  Solve $\bar{c}_{j^*} = \min\{z_j - c_j = \pi^\top \mathbf{a_j} - c_j, \forall j \in J\}$
  if $\bar{c}_{j^*} < 0$ **then**
    $J' \leftarrow J' \cup \{j^*\}$
until $\bar{c}_{j^*} \geq 0$

Takeaway: explicit enumeration of all patterns is totally out of the question!! The difficulty lies in defining the pricing problem.

8.2 The Dantzig-Wolfe Reformulation

The previous section does not distinguish the "easy" part and the "difficult" part of the constraints, however, in real problems, some constraints might be easier to dealt with, while others are more complicated. We can equivalently reformulate the LP model as

$$\begin{aligned}
(LP) \quad \min \quad & \mathbf{c}^\top \mathbf{x} \\
\text{s.t.} \quad & \mathbf{A}\mathbf{x} \geq \mathbf{b} \quad (\text{"difficult" constraints}) \\
& \mathbf{D}\mathbf{x} \geq \mathbf{d} \quad (\text{"easy" constraints}) \\
& \mathbf{x} \geq \mathbf{0}
\end{aligned}$$

In which, $\mathbf{D}\mathbf{x} \geq \mathbf{d}$ are the constraints we know how to deal with, and $\mathbf{A}\mathbf{x} \geq \mathbf{b}$ are the rest of constraints (that we do not like). Recall the Minkowski-Weyl Theorem again. For those "easy" constraints, i.e., the polyhedron $X = \{\mathbf{x} \geq \mathbf{0} | \mathbf{D}\mathbf{x} \geq \mathbf{d}\}$, can be represented by a set of extreme points $V = \{\mathbf{v_1}, \mathbf{v_2}, \cdots, \mathbf{v}_{|\mathbf{V}|}\}$ and a set of extreme rays $R = \{\mathbf{r_1}, \mathbf{r_2}, \cdots, \mathbf{r}_{|\mathbf{R}|}\}$. Replace the $\mathbf{D}\mathbf{x} \geq \mathbf{d}$ constraints with V and R, the origin (LP) model can be reformulated by substitution of $\mathbf{D}\mathbf{x} \geq \mathbf{d}$, as follows

$$\begin{aligned}
(LP) \quad \min \quad & \mathbf{c}^\top (\sum_{\mathbf{v} \in \mathbf{V}} \lambda_{\mathbf{v}} \mathbf{x}_{\mathbf{v}} + \sum_{\mathbf{r} \in \mathbf{R}} \lambda_{\mathbf{r}} \mathbf{x}_{\mathbf{r}}) \\
\text{s.t.} \quad & \mathbf{A}(\sum_{v \in V} \lambda_v \mathbf{x}_{\mathbf{v}} + \sum_{r \in R} \lambda_r \mathbf{x}_{\mathbf{r}}) \geq \mathbf{b} \\
& \sum_{v \in V} \lambda_v = 1 \\
& \lambda_v \geq 0 \quad \forall v \in V \\
& \lambda_r \geq 0 \quad \forall r \in R
\end{aligned}$$

For each x_v and x_r, define

$$\begin{aligned}
c_v &= \mathbf{c}^\top \mathbf{x}_{\mathbf{v}}, \forall v \in V \\
c_r &= \mathbf{c}^\top \mathbf{x}_{\mathbf{r}}, \forall r \in R \\
a_v &= \mathbf{A}\mathbf{x}_{\mathbf{v}}, \forall v \in V \\
a_r &= \mathbf{A}\mathbf{x}_{\mathbf{r}}, \forall r \in R
\end{aligned}$$

The (LP) model is reformulated as the master problem (MP)

$$
\begin{aligned}
\text{(MP)} \quad z^*_{MP} = \min \quad & \sum_{v\in V} c_v\lambda_v + \sum_{r\in R} c_r\lambda_r \\
\text{s.t.} \quad & \sum_{v\in V} a_v\lambda_v + \sum_{r\in R} a_r\lambda_r \geq \mathbf{b} \quad [\pi] \\
& \sum_{v\in V} \lambda_v = 1 \quad [\pi_0] \\
& \lambda_v \geq 0 \quad \forall v \in V \\
& \lambda_r \geq 0 \quad \forall r \in R
\end{aligned}
$$

There are two cases for reduce cost calculation:

- for $\lambda_v, v \in V$

$$\bar{c}_v = c_v - \begin{bmatrix}\pi^\top & \pi_0\end{bmatrix} \begin{bmatrix}\mathbf{a_v} \\ 1\end{bmatrix} = c_v - \pi^\top \mathbf{a_v} - \pi_0$$

- for $\lambda_r, r \in R$

$$\bar{c}_r = c_r - \begin{bmatrix}\pi^\top & \pi_0\end{bmatrix} \begin{bmatrix}\mathbf{a_r} \\ 0\end{bmatrix} = c_r - \pi^\top \mathbf{a}_r$$

The smallest reduce cost is derived by $\bar{c}^* = \min\{\min_{v\in V}\{\bar{c}_v\}, \min_{r\in R}\{\bar{c}_r\}\}$, which is the so called Dantzig-Wolfe pricing problem.

$$z^*_{PP} = \min_{j\in V\cup R} c_j - \pi^\top \mathbf{a}_j$$

In this formulation, both extreme points and extreme directions are considered. Since π_0 is a constant, we can ignore it as it is in the objective function.

Replace the decision variables back to the origin ones in the Dantzig-Wolfe pricing problem.

$$
\begin{aligned}
z^*_{PP} &= \min_{j\in V\cup R} c_j - \pi^\top \mathbf{a}_j \\
&= \min_{j\in V\cup R} \mathbf{c}^\top \mathbf{x}_j - \pi^\top \mathbf{A}\mathbf{x}_j
\end{aligned}
$$

which is

$$
\begin{aligned}
\min \quad & (\mathbf{c}^\top - \pi^\top \mathbf{A})\mathbf{x} \quad \text{(dual of "difficult" constraints)} \\
\text{s.t.} \quad & \mathbf{D}\mathbf{x} \geq \mathbf{d} \quad \text{("easy" constraints)} \\
& \mathbf{x} \geq \mathbf{0}
\end{aligned}
$$

Each time when a DW pricing problem is solved, one of the following three cases will

happen

- $z^*_{PP} = -\infty$. An extreme ray is identified, add λ_{r^*} to the RMP with cost $\mathbf{c}^\top \mathbf{x}_{\mathbf{r}^*}$ and column coefficient $\begin{bmatrix} \mathbf{A}\mathbf{x}_{\mathbf{r}^*} \\ 0 \end{bmatrix}$
- $-\infty < z^*_{PP} - \pi_0 < 0$. An extreme point is identified, add λ_{v^*} to the RMP with cost $\mathbf{c}^\top \mathbf{x}_{\mathbf{v}^*}$ and column coefficient $\begin{bmatrix} \mathbf{A}\mathbf{x}_{\mathbf{v}^*} \\ 1 \end{bmatrix}$
- $0 \leq z^*_{PP} - \pi_0$. STOP, $V \cup R = \emptyset$

8.3 Cutting Stock Problem

There are a stock of rods with length W, we need a set of items, J, each item $j \in J$ has a length of w_j for b_j copies. The objective is to obtain the demanded number of copies of each item by cutting the minimum possible number of stocks. Assuming there are at most M rods, which is sufficient to satisfy all demands. Let x_{ij} be the number of item j cut from stock i, and y_i be binary variables that indicates where stock i is used. The cutting stock problem is then defined as follows

$$
\begin{aligned}
\min \quad & z = \sum_{i=1}^{M} y_i \\
\text{s.t.} \quad & \sum_{i=1}^{M} x_{ij} \geq b_j, \quad j \in J \\
& \sum_{j \in J} w_j x_{ij} \leq W y_i, \quad i \in \{1, 2, \cdots, M\} \\
& x_{ij} \in \mathbb{Z}_+^{M \times |J|}, \quad i \in \{1, 2, \cdots, M\}, j \in JElementaryShortestPathProblemwithTimeWindowsandC \\
& y_i \in \{0, 1\}, \quad i \in \{1, 2, \cdots, M\}
\end{aligned}
$$

8.3.1 (Restricted) master problem

A cutting pattern is a possible way of cutting a stock, defined by the number of copies of each items obtained from that stock.

$$
\begin{aligned}
\min \quad & z = \sum_{q=1}^{Q} \lambda_q \\
\text{s.t.} \quad & \sum_{q=1}^{Q} p_q \lambda_q \geq b_j, \quad j \in J \quad (\pi) \\
& \lambda_q \in \mathbb{Z}, \quad q = 1, 2, \cdots, Q
\end{aligned}
$$

In which, there are exponential number of λ variables, one for each cutting pattern, from 1 to Q. Let p_{qj} indicate how many copies of item j are obtained in the qth cutting pattern.

8.3.2 Pricing subproblem

At each iteration, the following Integer Knapsack Problem is solved

$$\begin{aligned} \min \quad & \bar{c} = 1 - \sum_{j \in J} \pi_j x_j \\ \text{s.t.} \quad & \sum_{j \in J} w_j x_j \leq W \\ x_j \in \mathbb{Z}_+^{|J|}, \quad & \forall j \in J \end{aligned}$$

Each solution of this model is a cutting pattern.

8.4 Vehicle Routing Problem with Time Windows

Consider a fleet of vehicles V, a set of customers C, and a directed graph $G = (C \cup \{0\} \cup \{|C|+1\}, N)$. (0 as the starting depot and $|C|+1$ as a duplicate of depot for vehicles to return to). The cost of each arc is c_{ij} and the travel time is t_{ij}. Each vehicle has a capacity of q and each customer i has a demand of d_i. Each customer is associated with a time window $[a_i, b_i]$.

Let x_{ijk} be a binary decision variable, where

$$x_{ijk} = \begin{cases} 1, & \text{if vehicle } k \text{ drives directly from vertex } i \text{ to } j \\ 0, & \text{Otherwise} \end{cases}$$

and the decision variable s_{ik} is defined for each vertex i and vehicle k, it denotes the time vehicle k starts to serve customer i. The MILP formulation for VRPTW is described as follows

$$\begin{aligned} \min \quad & \sum_{k=1}^{|V|} \sum_{i=0}^{|C|} \sum_{j=1}^{|C|+1} c_{ij} x_{ijk} \\ \text{s.t.} \quad & \sum_{k=1}^{|V|} \sum_{j=1}^{|C|+1} x_{ijk} = 1, \quad \forall i \in C \quad \text{(Coupling constraints, no } k \in V\text{)} \\ & \sum_{i=1}^{|C|} d_i \sum_{j=1}^{|C|+1} x_{ijk} \leq q, \quad \forall k \in V \\ & \sum_{j=1}^{|C|+1} x_{0jk} = 1, \quad \forall k \in V \end{aligned}$$

$$\sum_{i=0}^{|C|} x_{ihk} = \sum_{j=1}^{|C|+1} x_{hjk} \quad \forall h \in C, k \in V$$
$$\sum_{i=0}^{|C|} x_{i,|C|+1,k} = 1, \quad \forall k \in V$$
$$s_{ik} + t_{ij} - M_{ij}(1 - x_{ijk}) \leq s_{jk}, \quad \forall i \in C \cup \{0\}, j \in C \cup \{|C|+1\}, k \in V$$
$$a_i \leq s_{ik} \leq b_i, \quad \forall i \in C \cup \{0\} \cup \{|C|+1\}, k \in V$$
$$x_{ijk} \in \{0,1\}, \quad \forall i \in C \cup \{0\} \cup \{|C|+1\}, k \in V$$

Now, the question is, who are the "difficult" constraints? Notice that there is only one set of coupling constraints ($\sum_{k=1}^{|V|} \sum_{j=1}^{|C|+1} x_{ijk} = 1$), and the remaining constraints are dealing with each vehicle separately. (They all have $\forall k \in V$), therefore, for sure, that coupling constraints should remain in the master problem. One of the approaches to is to keep the coupling constraints in the master problem.

8.4.1 (Restricted) Master Problem

The master problem is a set partitioning problem and the subproblem becomes a Shortest Path Problem with Time Windows and Capacity Constraints (SPPTWCC) or more specifically, the Elementary Shortest Path Problem with Time Windows and Capacity Constraints (ESPPTWCC).

The restricted master problem is defined as follows

$$\begin{aligned}
\text{(RMP)} \quad \min \quad & \sum_{r \in R'} c_r y_r \\
\text{s.t.} \quad & \sum_{r \in R'} a_{ir} y_r = 1, \quad \forall i \in C \quad [\pi] \\
& -\sum_{r \in R'} y_r \geq -K \quad [\pi_0] \\
& y_r \geq 0, \quad \forall r \in R' \quad \text{(exponential number of } r\text{s)}
\end{aligned}$$

For each feasible route r (an extreme point), c_r is the length of route r, and a_{ir} is the 0-1 parameter that take 1 if vertex i is visited by route r. Let w_{ij} be binary variables which takes 1 if the sub-path go direct from vertex i to j. Then,

$$c_r = \sum_{(i,j) \in E} c_{ij} w_{ij}$$
$$a_{ir} = \sum_{(i,j) \in E} w_{ij}$$

8.4.2 Pricing subproblem

The pricing subproblem is defined as follows,

$$
\begin{aligned}
\text{(PP)} \quad \min \quad & \sum_{(i,j)\in E} (c_{ij} - \pi_i) w_{ij} \\
\text{s.t.} \quad & \sum_{j:(i,j)\in E} w_{ij} - \sum_{j:(j,i)\in E} w_{ji} = 0, \quad \forall i \in C \\
& \sum_{j:(0,j)\in E} w_{0j} = 1 \\
& \sum_{j:(j,0)\in E} w_{j0} = 1 \\
& \sum_{(i,j)\in E} w_{ij} d_i \leq q \\
& t_j \geq t_i + d_{ij} - (1 - w_{ij})M, \quad \forall i \in C, j \in C \\
& a_i \leq t_i \leq b_i, \quad \forall i \in C
\end{aligned}
$$

Notice that we would prefer to have the subproblem "easy to solve". Although the ESPPTWCC is strongly NP-hard, we can still use efficient (compare to IP) methods such as labeling algorithms to speed up the subproblem solving.

8.5 Early Branching v.s. Branch-and-Price

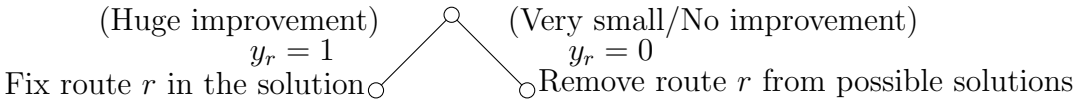
For MILP problems, the master problem in the Column Generation is an LP relaxation, thus it does not always produce integral solutions (if an integral solution is found, we reach the optimal solution). To derive the optimal solution, in general there are two approaches: early branching heuristic and the Branch-and-Price.

The early branching heuristic is a two-step approach. In the first step, do CG, until sufficient number of columns/routes are generated (with $y_r \geq 0$). Then, switch the definition of y_rs from $\mathbb{R}$ to $\{0, 1\}$, solve the IP. It is called early branching because it branches "early" with partial information. The downsides are, first, the subset of the routes might not always create a feasible solution for VRPTW, and second, the heuristic solution might be far away from the optimal solution.

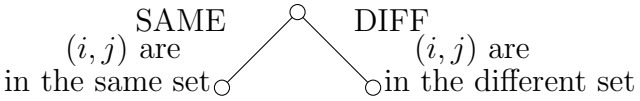
The other method is to incorporate the classic Branch-and-Bound strategy in the integer solution searching. The general idea is to replace the LP solving Simplex Methods in each iteration with a Column Generation subroutine.

There are several critical problems/challenges with Branch-and-Price for VRPTW (which is why my professor who taught me B&P don't like it, but somehow many researchers are fascinated with it... for some reasons). First, due to 0-1 branching, the B&P will create a **very** unbalance search tree. Take the following search tree as an example.

The right sub-tree will continue to be searched very inefficiently, due to the unbalance of the tree. The good news is, there are possible improvements to this issue by applying



different branching strategies. One of them is the Ryan-Foster Branching Rule, which is designed for the Set Covering Problem. For any fractional solution, there are at least two elements (i, j) so that i and j are both partially covered by the same set S, but there is another set T that only covers i



For the search go through the SAME branch, the following constraint will be added into the follow-up subproblems.

$$\sum_{l:(i,l)\in E} w_{il} = \sum_{l:(j,l\in E)} w_{jl}$$

For the DIFF branch, the corresponded constraints will be

$$\sum_{l:(i,l)\in E} w_{il} + \sum_{l:(j,l\in E)} w_{jl} \leq 1$$

Second, there is a terrible symmetric issue/degeneracy issue in the subproblem - the subproblem will produce duplicate routes. One of the another solutions is for every route r in $y_r = 0$ branches, add the following constraints

$$\sum_{(i,j)\in E, w_{ij}^r=0, \forall r\in R'} w_{ij} + \sum_{(i,j)\in E, w_{ij}^r=1, \forall r\in R'} (1 - w_{ij}) \geq 1$$

The idea is to use the edges that have not yet been used more, and use the edges that have appeared in other edges less. (But it will still be bad...)

Chapter 9

Benders Decomposition

"Learning from one's mistakes."

9.1 Benders Decomposition

Benders Decomposition is usually applied for those problems that are easy to solve after fixing some of the variables. The general idea is to divide the problem into a master problem containing all complicated variables and subproblems containing simple variables. For example, given an MILP model as follows

$$\begin{aligned} \text{(MILP)} \quad \min \quad & f^\top y + c^\top x \\ \text{s.t.} \quad & Ay \geq b \\ & By + Dx \geq d \\ & x \geq 0 \\ & \mathbf{y} \in \mathbb{Y} \subseteq \mathbb{Z}_+ \end{aligned}$$

In the formulation, decision variables x are separable, thus, for a given $\bar{y} \in \mathbb{Y}$, a subproblem is defined a follows

$$\begin{aligned} \text{(Sub)} \quad \min \quad & f^\top \bar{y} + c^\top x \\ \text{s.t.} \quad & B\bar{y} + Dx \geq d \\ & x \geq 0 \end{aligned}$$

In which $f^\top \bar{y}$ is a constant for given $\bar{y}$, rewrite (Sub) as follows

$$\text{(Sub)} \quad \eta(\bar{y}) = \min_x \{c^\top x | Dx \geq d - B\bar{y}, x \geq 0\}$$

Notice that $\bar{y}$ remains in the constraints of (Sub), every time when a new $\bar{y}$ is given, the feasible region changes, which makes it difficult to keep track of the model. To facilitate our analysis, we take the dual of (Sub), which maintains the same feasible region every time when a new $\bar{y}$ is given. The dual of the subproblem is defined as follows

$$\text{(Dual-Sub)} \quad \eta(\bar{y}) = \max_{\pi}\{\pi^\top(d - B\bar{y}) | \pi^\top D \leq c, \pi \geq 0\}$$

Then, the equivalent formulation of the original problem is

$$\begin{aligned} \text{(Master)} \quad \min \quad & f^\top y + \eta \\ & Ay \geq b \\ & \eta \geq \max_{\pi}\{\pi^\top(d - By) | \pi^\top D \leq c, \pi \geq 0\} \\ & \mathbf{y} \in \mathbb{Y} \subseteq \mathbb{Z}_+ \end{aligned}$$

Apply the Minkowski-Weyl Theorem to the (Dual-Sub), the feasible region $\{\pi | \pi^\top D \leq c, \pi \geq 0\}$ can be represented by a set of extreme points (e.p.) and a set of extreme directions (e.d.). Thus, let

- $\pi_e \in V$ be the set of all extreme points, and
- $\pi_d \in D$ be the set of all extreme directions

The original problem can be further rewritten as follows

$$\begin{aligned} \text{(Master)} \quad \min \quad & f^\top y + \eta \\ & Ay \geq b \\ & \eta \geq \pi_e^\top(d - By), \quad \forall \pi_e \in V \\ & 0 \geq \pi_d^\top(d - By), \quad \forall \pi_d \in D \\ & \eta \geq 0 \\ & \mathbf{y} \in \mathbb{Y} \subseteq \mathbb{Z}_+ \end{aligned}$$

In which, $\eta \geq \pi_e^\top(d - By)$ are referred as optimality cuts, and $0 \geq \pi_d^\top(d - By)$ are referred as feasibility cuts. However, the set V and D are exponentially large, it is intractable if we have all feasibility cuts and optimality cuts added into the model. To reduce the computational burden, a Branch-and-Cut scheme algorithm framework is more widely implemented by researchers.

Define the initial restricted master problem which only contains the integer variables $\mathbf{y}$ as follows

$$\begin{aligned} \text{(RMP)} \quad \min \quad & f^\top y + \eta \\ & Ay \geq b \\ & \eta \geq 0 \\ & \mathbf{y} \in \mathbb{Y} \subseteq \mathbb{Z}_+ \end{aligned}$$

Each iteration, an optimized $\bar{y}$ for the (RMP) will be given to the subproblem, which will create a (or multiple) optimality cut(s) or feasibility cut(s). Those cuts will be added

into the model in a lazy manner. Each time when an optimality cut is found and added, the lower bound of the master problem will be lifted, otherwise, if a feasibility cut is found and added, the upper bound will be updated. The iteration terminates when the lower bound meet with the upper bound within an acceptable error.

9.2 A Uncapacitated Facilities Location Problem

9.2.1 Formulation

Consider the following facility location problem, where m is the number of potential facilities, n is the number of customers, and $\mathbb{Y}$ is the set of feasible plans of facility location plans, where $\mathbf{y} \in \mathbb{Y} \subseteq \{0,1\}^m$. c_{ij} is the cost for customer i to be assigned to facility j, d_j is the cost for opening facility j. The formulation is as following

Table 9.1: Sets and Parameters

Notations	Description
m	Number of potential facilities
n	Number of customers
$F = \{1, 2, \ldots, m\}$	Set of potential facilities
$C = \{1, 2, \ldots, n\}$	Set of customers
d_j	The cost of constructing facility j, where $j \in$.
c_{ij}	The cost for customer i to receive service from facility j, where $i \in C$ and $j \in F$.

Table 9.2: Decision variables

Notations	Description
y_j	Binary variables, takes 1 if facility j is decided to be constructed, where $j \in F$.
x_{ij}	Continuous variables, the percentage of demand for customer i to be fulfilled by facilities j, where $i \in C$ and $j \in F$.

$$
\begin{aligned}
\text{(FLP)} \quad \min \quad & \sum_{i=1}^{n}\sum_{j=1}^{m} c_{ij}x_{ij} + \sum_{j=1}^{m} d_j y_j \\
\text{s.t.} \quad & \sum_{j=1}^{m} x_{ij} \geq 1, \quad \forall i \in C && (9.1)\\
& x_{ij} \leq y_j, \quad \forall i \in C, j \in F && (9.2)\\
& x_{ij} \geq 0, \quad \forall i \in C, j \in F && (9.3)\\
& y_j \in \{0,1\}, \quad \forall j \in F && (9.4)
\end{aligned}
$$

In the formulation (FLP),

- Constraints (9.1) indicate that for each customer i, its request needs to be fulfilled.
- Constraints (9.2) indicate that a facility j can only be able to serve customer i if the facility is opened.
- Constraints (9.3) is the nonnegative constraints for $\mathbf{x}$.
- Constraints (9.4) defines $\mathbf{y}$ as binary variables.

9.2.2 Solution Approach

If y are fixed, i.e., $y = \bar{y} \in \mathbb{Y}$, the rest of the formulation will become an LP model with x_{ij} as the nonnegative decision variables.

$$\begin{aligned}
\text{(Sub)} \quad \min \quad & \sum_{i=1}^{n}\sum_{j=1}^{m} c_{ij}x_{ij} + \sum_{j=1}^{m} d_j\bar{y}_j \\
\text{s.t.} \quad & \sum_{j=1}^{m} x_{ij} \geq 1, \quad \forall i \in C \\
& x_{ij} \leq \bar{y}_j, \quad \forall i \in C, j \in F \\
& x_{ij} \geq 0, \quad \forall i \in C, j \in F
\end{aligned}$$

If we take the dual of this new LP, we get

$$\begin{aligned}
\text{(Dual-Sub)} \quad \max \quad & \sum_{i=1}^{n}(\lambda_i - \sum_{j=1}^{m} \bar{y}_j\pi_{ij}) + \sum_{j=1}^{m} d_j\bar{y}_j \\
\text{s.t.} \quad & \lambda_i - \pi_{ij} \leq c_{ij} \quad \forall i \in C, j \in F \\
& \lambda_i \geq 0 \quad \forall i \in C \\
& \pi_{ij} \geq 0 \quad \forall i \in C, j \in F
\end{aligned}$$

Now get rid of the constant in the objective function of (Sub) and (Dual-Sub), which is $\sum_{j=1}^{m} d_j\bar{\mathbf{y}}$, we can then define

$$\begin{aligned}
\eta(\bar{\mathbf{y}}) &= \min_{\mathbf{x}}\{\sum_{i=1}^{n}\sum_{j=1}^{m} c_{ij}x_{ij} | \sum_{j=1}^{m} x_{ij} \geq 1, x_{ij} \leq \bar{y}_j, x_{ij} \geq 0, i \in C, j \in F\} \\
&= \max_{\pi}\{\sum_{i=1}^{n}(\lambda_i - \sum_{j=1}^{m} \bar{y}_j\pi_{ij}) | \lambda_i - \pi_{ij} \leq c_{ij}, \lambda_i \geq 0, \pi_{ij} \geq 0 i \in C, j \in F\}
\end{aligned}$$

The (FLP) model can be equivalently rewritten as follows

$$
\begin{aligned}
\text{(FLP)} \quad &= \min_{\bar{y}\in\mathbb{Y}}\{\sum_{j=1}^{m} d_j\bar{y}_j + \eta(\bar{\mathbf{y}})\} \\
&= \min_{\bar{y}\in\mathbb{Y}}\{\sum_{j=1}^{m} d_j\bar{y}_j + \max_{\pi}\{\sum_{i=1}^{n}(\lambda_i - \sum_{j=1}^{m}\bar{y}_j\pi_{ij}) | \lambda_i - \pi_{ij} \le c_{ij}, \lambda_i \ge 0, \pi_{ij} \ge 0 i \in C, j \in F\}\}
\end{aligned}
$$

Notice that for index i, the customer, there is no constraint among them, which is logically to be true since in reality customers are not related to each other.

$$
\begin{aligned}
\text{(Dual-Sub)} \quad v(\hat{\mathbf{y}}) = \max \quad & (\lambda_1 - \sum_{j=1}^{m}\hat{y}_j\pi_{1j}) + (\lambda_2 - \sum_{j=1}^{m}\hat{y}_j\pi_{2j}) + \ldots + (\lambda_n - \sum_{j=1}^{m}\hat{y}_j\pi_{nj}) + \sum_{j=1}^{m} d_j\hat{y}_j \\
\text{s.t.} \quad & \lambda_1 - \pi_{1j} \le c_{1j} && \forall j \in \{1,2,\ldots,m\} \\
& \lambda_2 - \pi_{2j} \le c_{2j} && \forall j \in \{1,2,\ldots,m\} \\
& \cdots \\
& \lambda_n - \pi_{nj} \le c_{nj} && \forall j \in \{1,2,\ldots,m\} \\
& \lambda_1 \ge 0 \\
& \lambda_i \ge 0 \\
& \cdots \\
& \lambda_n \ge 0 \\
& \pi_{1j} \ge 0 && j \in \{1,2,\ldots,m\} \\
& \pi_{2j} \ge 0 && j \in \{1,2,\ldots,m\} \\
& \cdots \\
& \pi_{nj} \ge 0 && j \in \{1,2,\ldots,m\}
\end{aligned}
$$

Therefore, the due of subproblem can be further decomposed by i, for each i,

$$
\begin{aligned}
\text{(Dual-Sub-}i\text{)} \quad \eta_i(\bar{\mathbf{y}}) = \max \quad & \lambda_i - \sum_{j=1}^{m}\bar{y}_j\pi_{ij} \\
\text{s.t.} \quad & \lambda_i - \pi_{ij} \le c_{ij}, \quad \forall j \in F \\
& \lambda_i \ge 0, \quad \forall i \in C \\
& \pi_{ij} \ge 0, \quad \forall i \in C, j \in F
\end{aligned}
$$

The (Dual-Sub-i) can easily be solved as following

$$
\bar{\lambda_i} = \min\{c_{ij}, j \in \mathbf{O}\}
$$

$$\begin{cases} \bar{\pi_{ij}} = 0, & j \in \mathbf{O} \\ \bar{\pi_{ij}} = \max\{0, \bar{\lambda_i} - c_{ij}\}, & j \in \mathbf{C} \end{cases}$$

In which $\mathbf{O}$ is the set of indices j where $\bar{y_j} = 1$ and $\mathbf{C}$ is the set of indices j where $\bar{y_j} = 0$. The optimal value of dual variables can be interpreted in terms of facilities location problem. $\bar{\lambda_i}$ is the cost of serving customer i when $y = \bar{y}$ and $\bar{\pi_{ij}}$ is the reduction in the cost of serving customer i when facility j is opened and $y_i = \bar{y_i}$.

After solving the (Dual-Sub-i), we can find the objective function value as well as the dual variable values for (Dual-Sub).

The restricted master problem initialized with no additional constraints, as follows

$$\begin{aligned} \text{(RMP)} \quad \min \quad & \sum_{j=1}^{m} d_j y_j \\ \text{s.t.} \quad & \mathbf{y} \in \mathbb{Y} \end{aligned}$$

There is only one case where the (Dual-Sub) problem will be unbounded, which would be when $\bar{\mathbf{y}} = \mathbf{0}$. In that case, a feasibility cut will be added into the master problem, as follows

$$0 \geq \sum_{i=1}^{n} (\bar{\lambda_i} - \sum_{j=1}^{m} y_j \bar{\pi_{ij}}) \tag{9.5}$$

For other cases, as long as there is at least one facility opened, the Subproblem is always feasible, therefore, a feasibility cut will be added into the master problem, as follows

$$\eta \geq \sum_{i=1}^{n} (\bar{\lambda_i} - \sum_{j=1}^{m} y_j \bar{\pi_{ij}})$$

The iteration continues until no more cut can be added into the master problem and we derive the optimal solutions.

9.3 Pareto-optimality cut

9.3.1 Choosing procedure

Definition 9.3.1 (dominance, pareto-optimal)**.** We say cut

$$z \geq hy + u^1(b - Dy) \succ z \geq hy + u^2(b - Dy) \tag{9.6}$$

if

$$hy + u^1(b - Dy) \geq hy + u^2(b - Dy), \quad \forall y \tag{9.7}$$

with at least one strict inequality. Further, we call a cut **pareto optimal** if it is not dominated by any other cut.

Notice that a Benders cut is determined by the vector u, so we want to find which of

the produced multiple solutions to the subproblem are not dominated by others (i.e. which vectors produce pareto optimal cuts).

Define a **core point** of $\mathbb{Y}$, $y^0 \in ri(\mathbb{Y}^c)$, which can be any point of the relative interior of convex hull of $\mathbb{Y}$.

Assume that we have $\hat{y}$, the current solution to the master problem, and $v(\hat{y})$ is the optimal value of the subproblem. To find a pareto optimal cut among the multiple solutions u of the dual of the subproblem $f(\hat{y})$, we solve the following linear program:

$$\begin{aligned} \max \quad & h^T y^0 + u(b - Dy^0) && (9.8) \\ \text{s.t.} \quad & u(b - D\hat{y}) = v(\hat{y}) - h^T \hat{y} && (9.9) \\ & uA \leq c^T && (9.10) \end{aligned}$$

Constraints (9.9) and (9.10) ensure that we are searching only through alternative optimal solutions for the current subproblem.

Note that y^0 can be any arbitrary core point of $\mathbb{Y}$. In other words, choosing different core point and solving that problem we obtain different pareto optimal cut, among which we cannot tell which one is better.

Let us see why is that true. Suppose that y^0 is a core point of $\mathbb{Y}$ and $U(\hat{y})$ is the set of optimal solutions to the subproblem:

$$\max\{h^T \hat{y} + u(b - D\hat{y})\} \tag{9.11}$$

Let u^0 solve the problem:

$$\max\{h^T y^0 + u(b - Dy^0)\} \tag{9.12}$$

Now we can prove by contradiction that u^0 is not dominated by any other cut from $U(\hat{y})$. Suppose there exists $\bar{u}$, that dominates u^0:

$$h^T y + \bar{u}(b - Dy) \geq h^T y + u^0(b - Dy) \quad \forall y \in \mathbb{Y} \tag{9.13}$$

taking any point w of the convex hull of $\mathbb{Y}^c$ (which can be represented as a convex combination of a finite number of points from $\mathbb{Y}$) we have:

$$h^T w + \bar{u}(b - Dw) \geq h^T w + u^0(b - Dw) \quad \forall w \in \mathbb{Y}^c \tag{9.14}$$

note that with $y = \hat{y}$ in 9.13, $\bar{u}$ must be an optimal solution to 9.11. But then from 9.13 and (9.12) we have:

$$h^T y + \bar{u}(b - Dy) = h^T y + u^0(b - Dy) \quad \forall y \in \mathbb{Y} \tag{9.15}$$

and since $\bar{u}$ dominated u^0, there exists at least one $\bar{y}$ such that:

$$h^T \bar{y} + \bar{u}(b - D\bar{y}) > h^T \bar{y} + u^0(b - D\bar{y}) \tag{9.16}$$

9.3.2 Pareto Optimal Cuts for Facility Location Problem

Consider the previous facility location problem. If y are fixed, i.e., $y = \hat{y} \in \mathbb{Y}$, the rest of the formulation will become a LP model with x_{ij} as the nonnegative decision variables which can further be separated as a set of subproblems for each i,

$$\begin{aligned} \text{(Dual-Sub-}i\text{)} \quad \max \quad & \lambda_i - \sum_{j=i}^{m} \hat{y}_j \pi_{ij} \\ \text{s.t.} \quad & \lambda_i - \pi_{ij} \le c_{ij} \quad \forall j \in \{1, 2, \ldots, m\} \\ & \lambda_i \ge 0 \\ & \pi_{ij} \ge 0 \qquad\qquad j \in \{1, 2, \ldots, m\} \end{aligned}$$

and each subproblem can be solved as following

$$\begin{aligned} & \hat{\lambda}_i = \min\{c_{ij}, j \in \mathbf{O}\} \\ & \hat{\pi}_{ij} = 0 \qquad\qquad\qquad \text{if } j \in \mathbf{O} \\ & \hat{\pi}_{ij} = \max\{0, \hat{\lambda}_i - c_{ij}\} \quad \text{if } j \in \mathbf{C} \end{aligned}$$

then, the optimal objective function for the ith separated subproblem is $v_i(\hat{\mathbf{y}}) = \hat{\lambda}_i$. From our previous discussion, for each i, we can solve the following problem (Pareto-Sub-i) to get (part of) the Pareto-optimal cut $\mathbf{u}_i = (\lambda_i, \pi_{i1}, \pi_{i2}, \cdots, \pi_{im})$. (The Pareto-optimal cut is $\mathbf{u} = (\mathbf{u}_1, \mathbf{u}_2, \cdots, \mathbf{u}_n)$)

$$\begin{aligned} \text{(Pareto-Sub-}i\text{)} \quad \max \quad & \lambda_i - \sum_{j=i}^{m} y_j^0 \pi_{ij} \\ \text{s.t.} \quad & \lambda_i - \sum_{j=i}^{m} \hat{y}_j \pi_{ij} = \hat{\lambda}_i \\ & \lambda_i - \pi_{ij} \le c_{ij} \qquad \forall j \in \{1, 2, \ldots, m\} \\ & \lambda_i \ge 0 \\ & \pi_{ij} \ge 0 \qquad\qquad j \in \{1, 2, \ldots, m\} \end{aligned}$$

where, as mentioned, $\mathbf{y}^0 \in ri(\mathbb{Y}^c)$. By observation, this problem can be solved as following. First, replace the objective function with the first equality constraint, then the formulation can be rewritten as

$$\text{(Pareto-Sub-}i\text{)} \quad \max \quad \hat{\lambda}_i + \sum_{j=i}^{m} (\hat{y}_j - y_j^0) \pi_{ij}$$

$$
\begin{aligned}
\text{s.t.} \quad & \lambda_i - \pi_{ij} \le c_{ij} && \forall j \in \{1, 2, \ldots, m\} \\
& \lambda_i \ge 0 \\
& \pi_{ij} \ge 0 && j \in \{1, 2, \ldots, m\}
\end{aligned}
$$

For those $j \in \mathbf{O}$, $\hat{y}_j = 1$, $\hat{y}_j - y_j^0 > 0$, for those π_{ij} we want to maximize them, however, we have $\lambda_i - \hat{\lambda}_i = \sum_{j \in \mathbf{O}} \pi_{ij}$, i.e., summation of all such π_{ij} is constrained, to maximize the objective function, we can choose the one where $\hat{y}_j - y_j^0$ is the most.

For those $j \in \mathbf{C}$, $\hat{y}_j = 0$, $\hat{y}_j - y_j^0 < 0$, for those π_{ij} we want to minimize them, therefore $\pi_{ij} = \max\{0, \lambda_i - c_{ij}\}$.

Therefore, the objective function value will be a function of λ_i, or $f(\lambda_i)$ as

$$
f(\lambda_i) = \hat{\lambda}_i + \max_j \{\hat{y}_j - y_j^0\}(\lambda_i - \hat{\lambda}_i) + \sum_{j \in \mathbf{C}} (\hat{y}_j - y_j^0) \max\{0, \lambda_i - c_{ij}\} \tag{9.17}
$$

This becomes a piece-wise linear function for λ_i, we can find the optimal value of λ_i easily, and once it is found, π_{ij} can be derived by $\max\{0, \lambda_i - c_{ij}\}$, therefore for a given i we can get $\mathbf{u}_i = (\lambda_i, \pi_{i1}, \pi_{i2}, \cdots, \pi_{im})$. Repeat this for n times, we can get the Pareto-optimal cut $\mathbf{u}$.

Chapter 10

Heuristic and Metaheuristic Methods

"Puzzle."

10.1 Heuristic-Search Procedures

The word "heuristic" stands for "I found it" in Greek. The heuristic algorithm are the procedures that we use some information available about the problem to

- reduce the search space
- or to speed up the search
- but not guarantee to find the optimum.

though there are some shortcomings about heuristic, still, we use heuristic because

- we need to avoid combinatorial explosion, because we need the solution FAST.
- we don't need the "optimal" solution, good approximation will be sufficient.
- The approximations may not be very good in the worst case, but in reality worst cases occur very rarely.
- Trying to understand why a heuristic works/does not work leads to a better understanding of the problem.

10.1.1 Hill-Climbing: An Irrevocable Strategy

Hill-climbing is a local search algorithm used for optimization problems. It starts with an initial solution and iteratively makes small improvements by selecting a neighboring solution that improves the objective function. The process continues until no better neighboring solutions can be found, often resulting in a local optimum. It is simple and efficient but can get stuck in suboptimal solutions due to its greedy nature.

The generic algorithm is as follows

Algorithm 27 Hill-climbing

```
1: S ← an initial solution
2: while not stopFlag do
3:     R ← Neighbor(S.clone())
4:     if OFV(R) is better than OFV(S) then
5:         S ← R
6: return S
```

We can make this algorithm a little more aggressive: create n neighbors to a candidate solution all at one time, and then possibly adopt the best one. This modified algorithm is called Steepest Ascent Hill-Climbing , because by sampling all around the original candidate solution and then picking the best, we're essentially sampling the gradient and marching straight up it.

The generic algorithm is as follows

Algorithm 28 Steep Ascent Hill-climbing

```
1: n ← number of neighbors
2: S ← an initial solution
3: while not stopFlag do
4:     R ← Neighbor(S.clone())
5:     for n − 1 times do
6:         W ← Neighbor(S.clone())
7:         if OFV(W) is better than OFV(R) then
8:             R ← W
9:     if OFV(R) is better than OFV(S) then
10:        S ← R
11: return S
```

There are several weakness of the Hill-climbing algorithm:

- They usually terminate at solutions that are only locally optimal.
- There is no information as to the amount by which the discovered local optimum deviates from the global optimum, or perhaps even other local optima.
- The optimum that's obtained depends on the initial configuration.
- In general, it is not possible to provide an upper bound for the computation time.

10.1.2 Uninformed Search

Uninformed search, also known as blind search, refers to a class of search algorithms that do not use any domain-specific knowledge or heuristic information to guide the search process. These algorithms explore the search space systematically without any additional information about the goal or the structure of the problem beyond the problem definition itself.

Depth-First and Backtracking: LIFO Search Strategies

Depth-First Search (DFS) starts at a selected node (often the root in trees or any node in graphs) and explores as far as possible along each branch before backtracking.

DFS uses a stack (either explicitly or through recursion) to keep track of nodes to visit. It is useful for tasks like solving mazes, topological sorting, and detecting cycles in graphs. However, it may not find the shortest path in unweighted graphs and can get stuck in infinite loops in graphs with cycles if not properly managed.

Breadth-First: A FIFO Search Strategy

Breadth-First Search (BFS) explores all nodes at the present depth level before moving on to nodes at the next depth level.

BFS is useful for finding the shortest path in unweighted graphs, level-order traversal in trees, and solving puzzles like the shortest path in a maze. It ensures that all nodes at a particular depth are explored before moving deeper, making it effective for shortest-path problems. However, it can be memory-intensive due to the queue storing all nodes at the current depth level.

Iterative Deepening Search

Iterative-Deepening Search combines BFS and DFS in an interesting way. IDS starts with a depth limit of 0 and increment it as long as it has not found a goal node. For every depth limit, IDS performs DFS up to the depth limit. This means that, if the current node is at the depth limit, IDS will not generate and add its successors to the frontier — essentially, IDS backtracks when it reaches the depth limit. Every time we increase the depth limit, IDS starts the depth-first search all over again.

IDS is an interesting hybrid of BFS and DFS. It is similar to DFS since it performs DFS for each depth limit. IDS is similar to BFS since it explores the search graph level by level by increasing the depth limit by one every time.

For space complexity, IDS performs DFS for every depth limit. Since IDS increases the depth limit by 1 each time, it will terminate at depth d, the depth of the shallowest goal node. The maximum length of the current path is d and each node on the path has at most b siblings.

Thus, the space complexity is $O(bd)$. This is linear in d, the depth of the shallowest goal node. The space complexity is similar to DFS.

In the worst case, IDS will visit all the nodes in the top d levels. Thus, IDS's time complexity is similar to that of BFS. The number of nodes up to depth d is dominated by the number of nodes at depth d, which is b^d. Thus, the time complexity is $O(b^d)$. This is exponential in d, the depth of the shallowest goal node.

Some pros about IDS:

- Similar to DFS, IDS requires linear space only.
- Similar to BFS, IDS is complete and is guaranteed to find the shallowest goal node.

- IDS also has the same time complexity as BFS, although the exact number of nodes visited by IDS is larger than that of BFS.

10.1.3 Informed Search

Informed search, also known as heuristic search, refers to a class of search algorithms that use problem-specific knowledge or heuristic information to guide the search process. Unlike uninformed search algorithms, which explore the search space blindly, informed search algorithms leverage additional information (such as an estimate of the cost to reach the goal) to make more intelligent decisions about which paths to explore.

For each node, we define the following notations for current node n.

- $h(n)$ as the estimated cost from the current node n to the destination.
- $g(n)$ as the known (exact) actual cost from the start node to current node n
- $f(n) = g(n) + h(n)$ is the heuristic knowledge. Think of $f(n)$ as an estimate of the cost of the cheapest path from the start state to a goal state through the current state n.

A* search

A* search explores the most promising next node, whereas depth first search goes as deep as possible in an arbitrary pattern and breadth first search explores all the nodes on one level before moving to the next. A* search uses a heuristic that provides a merit value for each node, whereas depth-first search and breadth-first search do not.

Algorithm 29 A* Search

```
1: Initialize priority queue Open with s
2: Initialize set Close as empty
3: Initialize f(s) = 0
4: while Open ≠ ∅ do
5:     cur ← node in Open with the lowest f(n)
6:     if cur == goal then
7:         return path found from s to cur
8:     Open ← Open \ {cur}
9:     Close ← Close ∪ {cur}
10:    for each neighbor n of cur do
11:        if n not in Close then
12:            g'(n) ← g(cur) + cost(cur, n)
13:            if n not in Open or g'(n) < g(n) then
14:                g(n) ← g'(n)
15:                f(n) ← g(n) + h(n)
16:                if n not in Open then
17:                    Add n to Open
```

Uniform Cost Search

A special case of the A* search, where $h(n) = 0$, use the cost only. Technically this is not an informed search.

Best-first search

Another special case of the A* search, where $g(n) = 0$, use the heuristic only.

10.2 Single-State Metaheuristic Methods

10.2.1 Simulated Annealing

Simulated Annealing is a probabilistic optimization technique inspired by the annealing process in metallurgy, where a material is heated and then slowly cooled to reduce defects. SA explores the solution space by accepting worse solutions with a certain probability, allowing it to escape local optima.

Simulated Annealing varies from Hill-Climbing in its decision of when to replace the original solution S with a newly found neighborhood R. Specially, if R is better than S, we always replace S with R as usual. But if R is worse than S, we may still replace S with R with a certain probability $P(t, R, S)$:

$$P(t, R, S) = e^{\frac{\texttt{OFV(R)} - \texttt{OFV(S)}}{t}}, t > 0 \tag{10.1}$$

The generic algorithm for Simulated Annealing is as follows

Algorithm 30 Simulated Annealing

```
1: t ← initial temperature
2: S ← initial candidate solution
3: Best ← S
4: while not stopFlag do
5:     R ← Neighbor(S.clone())
6:     if OFV(R) better than OFV(S) then
7:         S ← R
8:     else if Rnd() < e^((OFV(R) − OFV(S))/t) then
9:         S ← R
10:    Decrease t
11:    if OFV(S) better than OFV(Best) then
12:        Best ← S
13: return Best
```

10.2.2 Tabu Search

Tabu Search is a way of using memory in exchange of runtime. It keeps around a history of "recently" considered candidate solutions (known as a tabu list) and refuses to return to

those candidates until they are sufficiently far in the past. Thus if we wander up a hill, we have no choice but to wander back down the other side because we are not permitted to stay at or return to the top of the hill. The key to Tabu Search is to maintain the tabu list with some maximum length.

The generic algorithm for Tabu Search is as follows

Algorithm 31 Tabu Search

```
1: l ← desired maximum tabu list length
2: n ← number of new neighbor generate from current solution
3: S ← an initial solution
4: Best ← S
5: L ← {} as a tabu list of length l
6: L.enqueue(S)
7: while not stopFlag do
8:     while L.length > l do
9:         L.pop()
10:    R ← Neighbor(S.clone())
11:    for n times do
12:        W ← Neighbor(S.clone())
13:        if W ∉ L and OFV(W) better than OFV(R) or R ∈ L then
14:            R ← W
15:    if R ∉ L then
16:        S ← R
17:        L.enqueue(R)
18:    if S better than Best then
19:        Best ← S
20: return Best
```

10.2.3 Iterated Local Search

Iterated Local Search (ILS) tries to search through the space of local optima in a more intelligent fashion: it tries to stochastically hill-climb in the space of local optima. That is, ILS finds a local optimum, then looks for a "nearby" local optimum and possibly adopts that one instead, then finds a new "nearby" local optimum, and continue. The heuristic here is that you can often find a better local optima near to the current one, and walking from local optimum to local optimum in this way often outperforms just trying new locations entirely random.

There are two tricks in ILS, first, ILS does not pick a new restart location entirely at random, it maintains a "home base" local optimal and selects new restart locations that are near to the "home base" but not too close. We want the restart to be far away enough from current home base to wind up in a new local optimum, but not too far to be total random restart.

Second, when ILS discovers a new local optimum, it decides whether to retain the current "home base" local optimum, or to adopt the new local optimum as the "home base". There

are two extremes on the spectrum, if we always accept the new local optimum no matter what, we are doing a random walk, if we always accept the local optimum when it is better than our current one, we are doing hill-climbing. ILS is something in-between those two extremes.

The generic algorithm for ILS is as follows

Algorithm 32 Iterated Local Search with Random Restarts

```
1: T ← possible time intervals
2: S ← an initial solution
3: Update "home base" H ← S
4: Best ← S
5: while not stopFlag do
6:     t ← random time in the near future, sample from T
7:     while not run out to time t do
8:         R ← Neighbor(S.clone())
9:         if OFV(R) better than OFV(S) then
10:            S ← R
11:        if OFV(S) better than OFV(Best) then
12:            Best ← S
13:        H ← New "home base" according to S
14:        S ← Perturb(H)
15: return Best
```

10.2.4 Greedy Randomized Adaptive Search Procedures

The Greedy Randomized Adaptive Search Procedures (GRASP) is a component-oriented method, designed for certain kinds of spaces consist of combinations of components drawn from a typically fixed set. It's the presence of this fixed set that we can take advantage of in a greedy, local fashion by maintaining historical "quality" values of individual components rather than complete solutions.

GRASP is built on the notions of constructing and searching neighbors for feasible solutions, but it does not use any notion of component-level "historical quality". The overall algorithm is as follows: we create a feasible solution by constructing from among highest value (lowest cost) components and the do hill-climbing on the solution.

10.3 Population-based Metaheuristic Methods

10.3.1 Evolution Strategies

The family of evolution algorithms are based on a simple procedure for selecting individuals called truncation selection, and only use mutation as neighborhood searching operator. The simplest implementation is the (μ, λ) algorithm. We begin with a population of λ number of randomly generated individuals. Each iteration we access the fitness of each individual and

Algorithm 33 Greedy Randomized Adaptive Search Procedures (GRASP)

$C \leftarrow \{C_1, \cdots, C_n\}$ as a set of components
$p \leftarrow$ percentage of components to include each iteration
$m \leftarrow$ length of time to do hill-climbing
$Best \leftarrow$`None`
while not `stopFlag` **do**
    $S \leftarrow \{\}$ as candidate solution
    while S is not a feasible solution **do**
        $C' \leftarrow C_i \in C \setminus S$ which could be added to S without being infeasible
        if $C' = \emptyset$ **then**
            $S \leftarrow \{\}$
        else
            $C'' \leftarrow$ the $p\%$ highest value components in C'
            $S \leftarrow S \cup \{$component chosen uniformly at random from $C''\}$
    for m times **do**
        $R \leftarrow$ `Neighbor(S.clone())` ▷ R need to remain feasible
        if `OFV(`R`)` better than `OFV(`S`)` **then**
            $S \leftarrow R$
    if $B ==$`None` or `OFV(`S`)` better than `OFV(`$Best$`)` **then**
        $Best \leftarrow S$
return $Best$

remove all but the μ fittest ones. Each of the μ fittest individuals get to produce descendants to repopulate the group until the number of individuals recover to λ number.

Algorithm 34 The (μ, λ) Evolution Strategy

$\mu \leftarrow$ number of parents selected
$\lambda \leftarrow$ number of children generated by the parents
$P \leftarrow \{\}$
while $P.size < \lambda$ **do**
    $P \leftarrow P \cup \{$new random individual$\}$
$Best \leftarrow$ the best individual among P
while not `stopFlag` **do**
    for Each $P_i \in P$ **do**
        if `OFV(P`$_i$`)` better than `OFV(Best)` **then**
            $Best \leftarrow P_i$
    $Q \leftarrow \mu$ individuals in P whose $OFV()$ are better
    $P \leftarrow \{\}$ ▷ For $(\mu + \lambda)$, change it to $P \leftarrow Q$
    for Each $Q_i \in Q$ **do**
        for $\frac{\lambda}{\mu}$ times **do**
            $P \leftarrow P \cup \{$ `Mutate(Q.clone())` $\}$
return $Best$

The (μ, λ) algorithm has three knobs with which we may adjust exploration versus exploitation.

- The size of λ. This essentially controls the sample size for each population, and is basically the same thing as the n variable in Steepest-Ascent Hill Climbing With Replacement. At the extreme, as λ approaches ∞, the algorithm approaches exploration (random search).
- The size of μ. This controls how selective the algorithm is; low values of μ with respect to λ push the algorithm more towards exploitative search as only the best individuals survive.
- The degree to which Mutation is performed. If Mutate has a lot of noise, then new children fall far from the tree and are fairly random regardless of the selectivity of μ

10.3.2 Genetic Algorithm

The Genetic Algorithm is similar to the (μ, λ) Evolution Strategy in many respects: it iterates through fitness assessment, selection and breeding, and population reassembly. The primary difference is in how selection and breeding take place: whereas Evolution Strategies select all the parents and then create all the children, the Genetic Algorithm little-by-little selects a few parents and generates a few children until enough children have been created.To breed, we begin with an empty population of children. We then select two parents from the original population, copy them, cross them over with one another, and mutate the results. This forms two children, which we then add to the child population. We repeat this process until the child population is entirely filled.

10.3.3 Particle Swarm Optimization

Particle Swarm Optimization (PSO) is a stochastic optimization technique somewhat similar to evolutionary algorithms but different in an important way. It's modeled not after evolution per se, but after swarming and flocking behaviors in animals. Unlike other population-based methods, PSO does not resample populations to produce new ones: it has no selection of any kind. Instead, PSO maintains a single static population whose members are Tweaked in response to new discoveries about the space. The method is essentially a form of directed mutation.

The key differential between the GA and PSO are threefold:

- first, in GA, each individual moves in discrete space, each represents a discrete solution, while in PSO, each particle moves in continue space, the location of each particle represents a continuum option.
- second, in GA, individuals will be removed due to selection, but in PSO, the particles never die since there is no selection
- third, GA improves the population mainly by crossover, however, PSO improves the population by mutation.

Algorithm 35 Genetic Algorithm

$p \leftarrow$ size of population
$P \leftarrow \{\}$
while $P.size < p$ **do**
    $P \leftarrow P \cup \{\text{new random individual}\}$
$Best \leftarrow$ the best individual among P
while not `stopFlag` **do**
    for Each $P_i \in P$ **do**
        if `OFV(`P_i`)` better than `OFV(Best)` **then**
            $Best \leftarrow P_i$
    $Q \leftarrow \{\}$
    for $p/2$ times **do**
        Select a Parent $P_a \leftarrow$ `Select(P)`
        Select another parent $P_b \leftarrow$ `Select(P)`
        Breed children $C_a, C_b \leftarrow$ `Crossover(`P_a`,` P_b`)`
        $Q \leftarrow Q \cup \{C_a, C_b\}$
    $P \leftarrow Q$
return $Best$

Algorithm 36 Particle Swarm Optimization

$s \leftarrow$ size of swarm
$\alpha \leftarrow$ proportion of velocity to be retained
$\beta \leftarrow$ proportion of personal best to be retained
$\delta \leftarrow$ proportion of global best to be retained
$\epsilon \leftarrow$ jump size of a particle
$P \leftarrow \{\}$
while $P.size < s$ **do**
    $P \leftarrow P \cup \{\text{new random individual } (\mathbf{x}, \mathbf{v})\}$
$Best \leftarrow$ the best individual among P
while not `stopFlag` **do**
    for Each $P_i \in P$ **do**
        if `OFV(`P_i`)` better than `OFV(Best)` **then**
            $Best \leftarrow P_i$
    for $\mathbf{x_i} \in P$ with velocity $\mathbf{v_i}$ **do**
        $\mathbf{x}^* \leftarrow$ location of previous best solution
        $\mathbf{x_i}^+ \leftarrow$ location of i's previous best location
        Update speed $\mathbf{x_i} \leftarrow \mathbf{x_i} + \epsilon(\alpha \mathbf{v_i} + \beta(\mathbf{x}^* - \mathbf{x_i}) + \delta(\mathbf{x}^+{}_i - \mathbf{x}_i))$
return $Best$

10.3.4 Ant Colony Optimization

ACO is population-oriented. But there are two different kinds of "populations" in ACO. First, there is the set of components that make up a candidate solutions to the problem. In the Knapsack problem, this set would consist of all the blocks. In the TSP, it would consist of all the edges. The set of components never changes: but we will adjust the "fitness" (called the pheromone) of the various components in the population as time goes on.

Each generation we build one or more candidate solutions, called ant trails in ACO parlance, by selecting components one by one based, in part, on their pheromones. This constitutes the second "population" in ACO: the collection of trails. Then we assess the fitness of each trail. For each trail, each of the components in that trail is then updated based on that fitness: a bit of the trail's fitness is rolled into each component's pheromone.

Algorithm 37 Ant Colony Optimization (ACO)

$C \leftarrow \{C_1, \cdots, C_n\}$ as a set of components
$p \leftarrow$ size of population
$\mathbf{p} \leftarrow (p_1, p_2, \cdots, p)$ as the pheromones of the components, initially 0
$Best \leftarrow$ `None`
while not `stopFlag` **do**
    $P \leftarrow p$ trails built by iteratively selecting components based on pheromones and costs or values
    for $P_i \in P$ **do**
        $P_i \leftarrow$ `HillClimb(`P_i`)`
        if $B ==$`None` or `OFV(`S`)` better than `OFV(`$Best$`)` **then**
            $Best \leftarrow S$
    Update $\mathbf{p}$ the pheromones for each $P_i \in P$
return $Best$

10.4 Tuning the evolutionary algorithms

Almost every practical heuristic search algorithm is controlled by some set of parameters. E.g.,

- Genetic algorithm: population size, cross over rate, mutation rate, runtime limit, etc.
- Tabu search: tabu list length, tabu age, number of neighborhoods, runtime limit, etc.
- Simulated annealing: initial temperature, length staying at a temperature, accept rate, cooling rate, runtime limit, etc.
- $\cdots$

These parameters are applied to

- the representation

- the evaluation function
- the neighborhood searching operators
- the population size
- the stopping criteria
- $\cdots$

Since there are too many combinations of deciding the parameters, the trial-and-error method becomes very tedious and time-consuming, we need a theory to guide how to set some of the parameters of evolutionary algorithms.

In general, there are two types of parameter settings:

- Parameter tuning: offline parameter search - before the GA is running
  - Parameter sweep (try everything)
  - Meta-GA ontop of GA
  - Racing strategy to find best parameters
- Parameter control: online parameter search - during the GA is running
  - Deterministic genetic operators
  - Adaptive probabilities for mutation and crossover
  - Self-adaptive population size