

# ISEN 621: Heuristic Optimization

## Lectures

Sergiy Butenko

Industrial and Systems Engineering  
Texas A&M University

Spring 2015

## Course description

The course focusses on approximation algorithms and heuristic optimization methods that search beyond local optima. Procedures to be covered include neighborhood search methods and advanced search strategies such as genetic algorithms, simulated annealing, neural networks, tabu search, and greedy randomized adaptive search procedures.

**Text:** Emile Aarts and Jan Karel Lenstra (editors). *Local Search in Combinatorial Optimization*, John Wiley & Sons, Chichester.

**Prerequisites:** ISEN 421 and ISEN 622 or instructor's permission.

# Course description

**Announcements:** The course website will be maintained through <http://ecampus.tamu.edu/>. This site will contain announcements and other information concerning the course. In addition, it will be used to distribute homework assignments, class notes, or other material as required during the course. Please check the web site daily – any information posted on it will be as valid as if it was mentioned in class. You will need to use your TAMU NetId to log in.

# Course description

**Computer & Software Requirements:** Some of the methods will be implemented in MATLAB. Handouts on MATLAB programming will be distributed in class. You are not required to purchase any software.

**Computer Accounts:** The Industrial and Systems Engineering Department maintains an undergraduate computer lab that has virtual 24 hours access, 7 days a week. In order to use this facility, students are expected to establish their accounts within the first week of the course. The lab help desk is located at room 3005A.

# Outline of topics

1. Introduction
2. Computational complexity
3. Approximation algorithms
4. Construction and local search methods
5. Simulated annealing
6. Tabu search
7. Genetic algorithms
8. Neural networks
9. Greedy Randomized Adaptive Search Procedures (GRASP)
10. Overview of recent developments
11. Project presentations

# Course description

## Grading:

- ◇ Homework – 33%
- ◇ Exams (one in-class and one take-home) – 17% each
- ◇ Projects (individual or in teams of up to 3 students) – 33%

**Final grades** will be assigned as follows: 90-100% A, 80-90% B, 70-80% C, 60-70% D, < 60% F.

**Homeworks, projects, and due dates:** Homework assignments and projects must be handed in at the start of class on the day they are due. The grade for a late submission will be reduced by 20% of the maximum possible grade for each day overdue.

**Missed tests:** There will be no make-up exams unless you have a serious reason, and, in such cases, you must notify the instructor as early as possible before the exam.

## Course description

**The Americans with Disabilities Act (ADA)** is a federal anti-discrimination statute that provides comprehensive civil rights protection for persons with disabilities. Among other things, this legislation requires that all students with disabilities be guaranteed a learning environment that provides for reasonable accommodation of their disabilities. If you believe you have a disability requiring an accommodation, please contact the Department of Student Life, Services for Students with Disabilities in Room 126 of the Koldus Building, or call 845-1637.

**Academic Integrity:** “Aggies do not lie, cheat, or steal, nor do they tolerate those who do.” It is the responsibility of students and instructors to help maintain scholastic integrity at the university by refusing to participate in or tolerate scholastic dishonesty. (<http://www.tamu.edu/aggiehonor>)

# Introduction

- ◇ A combinatorial optimization problem is given by a set of problem instances and a minimization (or maximization) objective.
- ◇ An instance of a combinatorial optimization problem is a pair  $(S, f)$ , where the solution set  $S$  is the set of feasible solutions and the cost function  $f$  is a mapping  $f : S \rightarrow \mathbb{R}$ .
- ◇ The objective is to find a globally optimal solution, i.e., a feasible solution  $s^* \in S$  such that  $f(s^*) \leq f(s)$  for all  $s \in S$ .
- ◇ Denote by  $f^* = f(s^*) = \min_{s \in S} f(s)$  the optimal cost, and by  $S^* = \{s \in S : f(s) = f^*\} = \arg \min_{s \in S} f(s)$  - the set of optimal solutions.
- ◇ The instance  $(S, f)$  is usually given by a compact data representation rather than by listing all elements of  $S$  explicitly.



# Introduction

## Example (The Traveling Salesman Problem (TSP))

- ◇ Given a set of  $n$  cities  $C = \{1, 2, \dots, n\}$  and their pairwise distances  $D = \{d(i, j) : i, j \in C\}$ , a TSP instance is given by all possible tours of the cities in  $C$ .
- ◇ A tour  $T$  of the  $n$  cities in  $C$  is defined as an ordered sequence  $T = \langle c_1 - c_2 - \dots - c_n - c_{n+1} \rangle$  of the cities, which starts and ends with city 1 (i.e.,  $c_1 = c_{n+1} = 1$ ) and contains each of the remaining cities exactly once (i.e.,  $\{c_1, \dots, c_n\} = C$ ).
- ◇ The length  $\ell(T)$  of the tour  $T$  is given by

$$\ell(T) = \sum_{i=1}^n d(c_i, c_{i+1}).$$

- ◇ The objective is to find a tour  $T^*$  of the minimum possible length for the given instance.

# Neighborhood

## Definition

For an instance  $(S, f)$  of a combinatorial optimization problem, a **neighborhood function** is a mapping  $N : S \rightarrow 2^S$ .

- ◇ For each feasible solution  $s \in S$ ,  $N(s) \subseteq S$  is a set of solutions that are in some sense close to  $s$ .
- ◇  $N(s)$  is the neighborhood of solution  $s$ ;
- ◇ each  $i \in N(s)$  is a neighbor of  $s$ .
- ◇ A neighborhood for a problem  $\mathcal{P}$  is defined by a rule that can be used to find the neighborhood of any feasible solution for any instance of  $\mathcal{P}$ .

# Local and global optima

## Definition

For an instance  $(S, f)$  of a combinatorial optimization problem and a neighborhood  $N$ , a solution  $\hat{s}$  is locally minimal with respect to  $N$  if

$$f(\hat{s}) \leq f(i) \text{ for all } i \in N(\hat{s}).$$

◇ Let  $\hat{S}$  denote the set of all local optimal solutions of  $(S, f)$ .

## Definition

A neighborhood  $N$  defined for a problem  $\mathcal{P}$  is called exact for  $\mathcal{P}$  if for any instance  $(S, f)$  of  $\mathcal{P}$  we have  $\hat{S} \subseteq S^*$ , i.e., any local optimal solution is global optimal.

# Optimization algorithms

Different types of algorithms:

## 1. Exact algorithms:

- ◇ guaranteed to find a global optimal solution
- ◇ in most cases, the run times are unreasonably high for large-scale problems

## 2. Approximate algorithms:

- ◇ guaranteed to find an approximate solution within a known approximation ratio
- ◇ can find an approximate solution for large-scale instances fast
- ◇ For many hard problems, it can be shown that, unless some very unlikely statement is true, there cannot be an efficient approximation algorithm with a constant approximation ratio

## 3. Heuristic algorithms (from Greek “εὕρισκω” - “to find”):

- ◇ find a “good” solution “fast”
- ◇ no guarantee on the quality of the solution in general



# Heuristic algorithms

## 1. Construction heuristics

- ◇ construct a feasible solution from scratch
- ◇ random construction; greedy construction

## 2. Local search

- ◇ given a feasible solution, we iteratively search for a better neighbor, until we achieve a local optimum
- ◇ first improvement; best improvement

## 3. Metaheuristics (from the Greek “ $\mu\epsilon\tau\alpha$ ” - “beyond”)

- ◇ search beyond local optimum
- ◇ use various (often randomized) strategies to escape poor-quality local optima in a hope to find a global optimum.

## 4. Hybrid algorithms

- ◇ combine several metaheuristic strategies together
- ◇ expected to be at least as good as any of the metaheuristics combined in terms of quality of the solutions

# What is an algorithm?

An **algorithm**  $\mathcal{A}$  is a clearly defined sequence of instructions aiming to solve a clearly defined computational problem  $\mathcal{P}$ .

## Example

Given integers  $a$  and  $n$ , find  $a^n$

1.  $power \leftarrow 1$ ;
2. for  $i \leftarrow 1..n$   
     $power \leftarrow power * a$ ;  
end

The **run time** of an algorithm is defined as the number of elementary operations (such as addition, multiplication, logical instructions, ...) performed by the algorithm before termination. We assume that each elementary operation takes one unit of time.

# Asymptotic notations

Let  $f(n), g(n)$  be functions from the positive integers to the positive reals.

1.  $f(n) = O(g(n))$  if there exists a constant  $c > 0$  such that, for large enough  $n_0$ ,  $f(n) \leq cg(n)$  if  $n \geq n_0$ .
2.  $f(n) = \Omega(g(n))$  if there exists a constant  $c > 0$  such that, for large enough  $n_0$ ,  $f(n) \geq cg(n)$  if  $n \geq n_0$ .
3.  $f(n) = \Theta(g(n))$  if there exists constants  $c, c' > 0$  such that, for large enough  $n_0$ ,  $cg(n) \leq f(n) \leq c'g(n)$  if  $n \geq n_0$ .

Note that  $\Theta$  induces an equivalence relation on the set of functions. The equivalence class of  $f(n)$  is the set of all functions  $g(n)$  such that  $f(n) = \Theta(g(n))$ . We call this class the **rate of growth** of  $f(n)$ .



# Asymptotic notations

For any polynomial  $f(n)$  and  $g(n)$  of degree  $k$ , we have

$$f(n) = \Theta(g(n)) = \Theta(n^k).$$

So,  $n^k$  is a natural representative element for the class of polynomials of degree  $k$ .

If an algorithm runs in time  $O(n^k)$ , where  $n$  is the problem input size (to be defined next), then we say that this is a **polynomial-time** or **efficient** algorithm.

# Size of a problem

Run time is usually expressed in terms of the problem's size.

The problem's size is the size of input provided to the algorithm for this problem.

The input of an algorithm is represented as a sequence (string) of symbols.

Then the **input size** is defined as the length of this string, i.e., the number of bits required to store the problem's input.

## Example

Assume that the input is given by a single integer. The number of symbols required to represent an integer  $n$  in a base- $\beta$  arithmetic system is  $\lceil \log_{\beta} n \rceil$ , where  $\beta \geq 2$ . So, the size of the representation of  $n$  is  $\Theta(\log n)$ .

# Size of a problem

## Example

Consider a linear programming (LP) problem

$$\max c^T x, \quad \text{s.t. } Ax \leq b, x \geq 0,$$

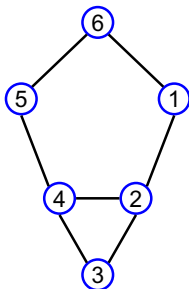
where  $A$  is an  $m \times n$  matrix,  $c$  is an  $n$ -vector,  $b$  is an  $m$ -vector.  
Then the size of this  $m \times n$  LP is

$$\Theta(mn + \lceil \log |P| \rceil),$$

where  $P$  is the product of all nonzero coefficients.

# Graph theory

A **simple, undirected graph** is a pair  $G = (V, E)$ , where  $V$  is a finite set of vertices and  $E \subseteq V \times V$  is a set of edges, with each edge defined on a pair of vertices.



$$V = \{1, 2, 3, 4, 5, 6\}$$

$$E = \{(1, 2), (1, 6), (2, 3), (2, 4), (3, 4), (4, 5), (5, 6)\}$$

# Graph theory

- ◇ If  $G = (V, E)$  is a graph and  $e = (u, v) \in E$ , we say that  $u$  is **adjacent** to  $v$  and vice-versa. We also say that  $u$  and  $v$  are **neighbors**.
- ◇ Neighborhood  $N(v)$  of a vertex  $v$  is the set of all its neighbors in  $G$ :  $N(v) = \{j : (i, j) \in E\}$ .
- ◇ If  $G = (V, E)$  is a graph and  $e = (u, v) \in E$ , we say that  $e$  is **incident** to  $u$  and  $v$ .
- ◇ The **degree** of a vertex  $v$  is the number of its incident edges.

# Graph theory

A graph can be represented in several ways.

- ◇ We can represent a graph  $G = (V, E)$  by its  $|V| \times |V|$  **adjacency matrix**  $A_G = [a_{ij}]$ , such that

$$a_{ij} = \begin{cases} 1, & \text{if } (v_i, v_j) \in E, \\ 0, & \text{otherwise.} \end{cases}$$

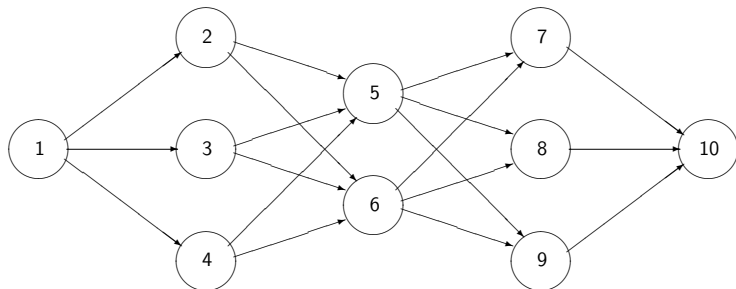
- ◇ Another way of representing a graph is by its **adjacency lists**, where for each vertex  $v \in V$  we record the set of vertices adjacent to it.

We have  $2|E|$  elements, each requiring  $\Theta(\log |V|)$  bits  $\Rightarrow$   
 $\Theta(|E| \log |V|)$  space is required.

Since modern computers usually treat all integers in their range the same,  $O(|E|)$  space is a reasonable approximation of the size of a graph.

# Graph theory

- ◇ A **multigraph** is a graph with repeated edges.
- ◇ A **directed graph**, or **digraph**, is a graph with directions assigned to its edges.



# Graph theory

- ◇ A **subgraph** of  $G = (V, E)$  is a graph  $G' = (V', E')$  such that

$$V' \subseteq V \text{ and } E' \subseteq E.$$

- ◇ For a subset  $S$  of vertices, the **subgraph induced by  $S$**  is given by

$$G[S] = (S, (S \times S) \cap E),$$

where  $S \times S = \{(i, j) : i, j \in S\}$ .



# Graph theory

- ◇ A **walk** in  $G$  is a sequence of vertices  $w = (v_1, v_2, \dots, v_k)$ ,  $k \geq 1$ , such that  $(v_j, v_{j+1}) \in E$  for  $j = 1, \dots, k - 1$ . The walk is closed if  $k > 1$  and  $v_1 = v_k$ .
- ◇ A walk without any repeated vertices in it is called a **path**.
- ◇ A closed walk with no repeated vertices other than the first and last one is called a **circuit** or a **cycle**.
- ◇ The **length** of the path  $(v_1, \dots, v_k)$  is  $k - 1$ ; so is the length of the cycle  $(v_1, \dots, v_k = v_1)$ .
- ◇ A path or cycle is **Hamiltonian** (or **spanning**) if it uses all vertices exactly once.

# Sorting

## Problem

*Given  $n$  integers  $a_1, a_2, \dots, a_n$ , sort them in non-decreasing order.*

**Input:**  $a_1, a_2, \dots, a_n$

**Output:**  $q_1, q_2, \dots, q_n$ , where  $q_1 \leq q_2 \leq \dots \leq q_n$  and  
 $\{q_1, q_2, \dots, q_n\} = \{a_1, a_2, \dots, a_n\}$

# Quicksort

Quicksort (Hoare, 1960) uses a divide and conquer strategy to divide a list into two sub-lists.

1. Pick an element, called a pivot, from the list. We can always pick the first element in the list as the pivot.
2. Partition Operation: Reorder the list so that all elements which are less than the pivot come before the pivot and so that all elements greater than the pivot come after it (equal values can go either way). After this partitioning, the pivot is in its final position.
3. Recursively sort the sub-list of lesser elements and the sub-list of greater elements.

# Quicksort

At step  $k$  of Quicksort we make at most  $n - k$  comparisons (the worst case is when the list is already sorted). The worst-case run time is given by

$$T(n) = (n - 1) + (n - 2) + (n - 3) + \dots + 1 = \frac{n(n - 1)}{2} = \Theta(n^2).$$

Thus, Quicksort is an  $\Omega(n^2)$  algorithm.

# Mergesort

Mergesort (von Neumann, 1945) proceeds as follows:

1. If the list is of length 0 or 1, then do nothing (list is already sorted). Otherwise:
2. Split the unsorted list into two equal parts.
3. Sort each sublist recursively by re-applying Mergesort.
4. Merge the two sublists back into one sorted list.

# Mergesort

Note that the merge operation involves  $n - 1$  comparisons. Let  $T(n)$  denote the run time of Mergesort on the list of  $n$  numbers. Then

$$\begin{aligned}T(n) &\leq 2T(n/2) + n - 1 \\&\leq 2(2T(n/4) + n/2 - 1) + n - 1 \\&= 4T(n/4) + (n - 2) + (n - 1) \\&\dots \\&\leq 2^k T(n/2^k) + \sum_{i=0}^{k-1} (n - 2^i) \\&= 2^k T(n/2^k) + kn - (2^k - 1) \\T(1) &= 0\end{aligned}$$

Using  $k = \log_2 n$ , we obtain

$$T(n) \leq n \log n - n + 1 = \Theta(n \log n).$$

# Sorting

We discussed two sorting algorithms with the following run times:

- ◇ Quicksort:  $\Omega(n^2)$ ;
- ◇ Mergesort:  $O(n \log n)$ .

Mergesort is better in terms of worst-case run time.

How about the “average” run time?

# Average run time

## Definition

Assume that all possible inputs of a given problem of size  $n$  are equally probable. Then the **average run time**  $A(n)$  of an algorithm for solving this problem is defined as the expected run time over all possible outcomes.

## Example

For sorting, different inputs correspond to different orders of numbers in the list.



## Average run time of Quicksort

How many comparisons do we make on average in Quicksort?

Denote by  $x_{ij}$  the random variable representing the number of comparisons between  $q_i$  and  $q_j$  during a Quicksort run, where  $q_1 \leq q_2 \leq \dots \leq q_n$  is the sorted version of the input  $a_1, a_2, \dots, a_n$ .

$$x_{ij} = \begin{cases} 1, & \text{if } q_i \text{ and } q_j \text{ are compared (with probability } p_{ij}) \\ 0, & \text{if } q_i \text{ and } q_j \text{ are not compared (with probability } 1 - p_{ij}) \end{cases}$$

Then

$$\begin{aligned} A(n) &= E \left( \sum_{i=1}^n \sum_{j=i+1}^n x_{ij} \right) \\ &= \sum_{i=1}^n \sum_{j=i+1}^n E(x_{ij}) \\ &= \sum_{i=1}^n \sum_{j=i+1}^n (1 \cdot p_{ij} + 0 \cdot (1 - p_{ij})) \\ &= \sum_{i=1}^n \sum_{j=i+1}^n p_{ij}. \end{aligned}$$

## Average run time of Quicksort

Note that  $q_i$  and  $q_j$  are compared iff none of the numbers  $q_{i+1}, q_{i+2}, \dots, q_{j-1}$  appear before both  $q_i$  and  $q_j$  in the list  $a_1, \dots, a_n$ . The probability of this happening is

$$p_{ij} = \frac{2}{j-i+1}.$$

Hence,

$$\begin{aligned} A(n) &= \sum_{i=1}^n \sum_{j=i+1}^n p_{ij} = \sum_{i=1}^n \sum_{j=i+1}^n \frac{2}{j-i+1} \\ &= \sum_{i=1}^n \left( \frac{2}{2} + \frac{2}{3} + \frac{2}{4} + \dots + \frac{2}{n-i+1} \right) \\ &< 2 \sum_{i=1}^n \left( \frac{1}{2} + \frac{1}{3} + \frac{1}{4} + \dots + \frac{1}{n} \right) \\ &= 2n \sum_{i=2}^n \frac{1}{i} \leq 2n \ln n = O(n \log n). \end{aligned}$$

So, the average run time of Quicksort is  $O(n \log n)$ .

# Randomized algorithms

A **randomized algorithm** is an algorithm in which some decisions depend on the outcome of a coin flip (that can be either 0 or 1).

- ◇ **Las-Vegas Algorithms:** The answer is always correct, but run time is random.
- ◇ **Monte-Carlo Algorithms:** The run time is predefined, and the answer is correct with “high probability”.

# What is “high” probability?

## Definition

For a problem of size  $n$ , we say that

$$p = 1 - \frac{1}{n^\alpha} = 1 - n^{-\alpha},$$

where  $\alpha > 1$ , is a **high probability**.

## Definition

We say that a randomized algorithm takes  $\tilde{O}(f(n))$  of a resource (such as space or time) if there exist numbers  $C, n_0 > 0$  such that for any  $n \geq n_0$  the amount of resource used is

$$\leq C\alpha f(n) \text{ with probability } \geq 1 - n^{-\alpha}, \alpha > 1.$$

# Randomized algorithms

## Problem (Repeated element identification)

*Given  $n$  numbers,  $n/2$  of which are the same number and the remaining  $n/2$  numbers are all different, find the repeated number.*

- ◇ A deterministic solution: sort the numbers, scan the sorted list until two consecutive numbers are equal. Time complexity:  $O(n \log n)$ .
- ◇ Las-Vegas Algorithm:

```
Input:  $a_1, a_2, \dots, a_n$ 
repeat forever
    randomly pick two elements  $a_i$  and  $a_j$ ;
    if  $(i \neq j)$  and  $(a_i = a_j)$ 
        return  $a_i$ ;
    end if
end repeat
```

## Repeated element identification

After how many steps do we get the correct answer with high probability?

- ◇ Probability of success in a single try:

$$P_s = \frac{\frac{n}{2}(\frac{n}{2} - 1)}{n^2} = \frac{\frac{n^2}{4} - \frac{n}{2}}{n^2} = \frac{1}{4} - \frac{1}{2n} > \frac{1}{8} \text{ if } n > 4.$$

- ◇ Probability of failure in a single try:

$$P_f = 1 - P_s < 7/8.$$

- ◇ Probability of success in  $k$  tries is  $1 - (P_f)^k > 1 - (7/8)^k$ .
- ◇ We want the probability of success to be high ( $1 - n^{-\alpha}$ ):

$$1 - (7/8)^k = 1 - n^{-\alpha} \Leftrightarrow (7/8)^k = n^{-\alpha} \Leftrightarrow k = \frac{\alpha \log n}{\log(8/7)}.$$

- ◇ Let  $c = 1/\log(8/7)$ . If  $k = c\alpha \log n$ , then we have a high probability of success in  $k$  tries.
- ◇ The run time of the algorithm is  $\tilde{O}(\log n)$ .

# Basics of the complexity theory

- ◇ Given a combinatorial optimization problem, a natural question is:  
is this problem “easy” or “hard”?
- ◇ Designing heuristics for easy problems makes little sense.
- ◇ How do we distinguish between “easy” and “hard” problems?

# “Easy” problems

- ◇ By easy or **tractable** problems we mean the problems that can be solved in time polynomial with respect to their size.
- ◇ We also call such problems **polynomially solvable** and denote the class of polynomially solvable problems by  $\mathcal{P}$ .
  - ◇ Sorting
  - ◇ Minimum weight spanning tree
  - ◇ Linear programming



# Defining “hard” problems

- ◇ How do we define “hard” problems?
- ◇ How about defining hard problems as all problems that are not easy, i.e., not in  $\mathcal{P}$ ?
- ◇ Then some of the problems in such a class could be TOO hard – we cannot even hope to be able to solve them.
- ◇ We want to define a class of hard problems that we may be able to solve, if we are lucky (say, we may be able to guess the solution and check that it is indeed correct).

# Defining “hard” problems

- ◇ Let us take the traveling salesman problem (TSP) as an example and try to guess a solution.
- ◇ We can pick some random tour, compute its length, but how do we know if this tour is optimal?
- ◇ Indeed, this is as difficult to verify as to solve the TSP – it is unlikely that we can count on luck in this case...
- ◇ How about transforming our optimization problem to such a form, where instead of finding an optimal solution the question would be “is there a solution with objective value  $\leq L$ , where  $L$  is a given integer?

# Three versions of optimization problems

Consider a problem

$$\min f(x) \text{ subject to } x \in X.$$

- ▶ **Optimization version:** find  $x$  from  $X$  that minimizes  $f(x)$ ;  
Answer:  $x^*$  minimizes  $f(x)$
- ▶ **Evaluation version:** find the smallest possible  $f(x)$ ;  
Answer: the smallest possible value for  $f(x)$  is  $f^*$
- ▶ **Recognition version:** Given  $f^*$ , does there exist an  $x$  such that  $f(x) \leq f^*$ ?  
Answer: “yes” or “no”

# Recognition problems

- ◇ Recognition problems can still be undecidable.

*Halting problem: Given a computer program with its input, will it ever halt?*

- ◇ Recall the “luck factor”: if we pick a random feasible solution and it happens to give “yes” answer, then we solved the problem in polynomial time.

*TSP: Randomly pick a solution (a TSP tour). If its length is  $\leq L$  (which we can verify in polynomial time), then obviously the answer is “yes”. This tour can be viewed as a certificate proving that this is indeed a yes instance of TSP.*

# Class $\mathcal{NP}$

- ◇ We only consider problems for which any **yes** instance there exists a **concise** (polynomial-size) **certificate** that can be verified in polynomial time.
- ◇ We call this class of problems **nondeterministic polynomial** and denote it by  $\mathcal{NP}$ .

# $\mathcal{P}$ vs $\mathcal{NP}$

- ◇ Note that any problem from  $\mathcal{P}$  is also in  $\mathcal{NP}$  (i.e.,  $\mathcal{P} \subseteq \mathcal{NP}$ ), so there are easy problems in  $\mathcal{NP}$ .
- ◇ So, are there “hard” problems in  $\mathcal{NP}$ , and if there are, how do we define them?
- ◇ We don’t know if  $P = NP$ , but “most” people believe that  $P \neq NP$ .
- ◇ An “easy” way to make \$1,000,000!  
[http://www.claymath.org/millennium/P\\_vs\\_NP/](http://www.claymath.org/millennium/P_vs_NP/)
- ◇ We can call a problem hard if the fact that we can solve this problem would mean that we can solve any other problem in comparable amount of time.

# Polynomial reducibility

- ◇ Reduce  $\pi_1$  to  $\pi_2$ : if we can solve  $\pi_2$  fast, then we can solve  $\pi_1$  fast, given that the reduction is “easy”.
- ◇ Polynomial reduction from  $\pi_1$  to  $\pi_2$  requires existence of polynomial-time algorithms
  1.  $A_1$  converts an input for  $\pi_1$  into an input for  $\pi_2$ ;
  2.  $A_2$  converts an output for  $\pi_2$  into output for  $\pi_1$ .
- ◇ **Transitivity**: If  $\pi_1$  is polynomially reducible to  $\pi_2$  and  $\pi_2$  is polynomially reducible to  $\pi_3$  then  $\pi_1$  is polynomially reducible to  $\pi_3$ .

# NP-complete problems

- ◇ A problem  $\pi$  is called **NP-complete** if
  1.  $\pi \in NP$ ;
  2. Any problem from  $NP$  can be reduced to  $\pi$  in polynomial time.
- ◇ A problem  $\pi$  is called **NP-hard** if any problem from  $NP$  can be reduced to  $\pi$  in polynomial time. (no  $\pi \in NP$  requirement)
- ◇ Due to transitivity of polynomial reducibility, in order to show that a problem  $\pi$  is  $\mathcal{NP}$ -complete, it is sufficient to show that
  1.  $\pi \in NP$ ;
  2. There is an  $NP$ -complete problem  $\pi'$  that can be reduced to  $\pi$  in polynomial time.

To use this observation, we need to know at least one  $\mathcal{NP}$ -complete problem...



# Satisfiability (SAT) problem

- ▶ A Boolean variable  $x$  is a variable that can assume only the values **true** and **false**.
- ▶ Boolean variables can be combined to form Boolean formulas using the following logical operations:
  1. Logical AND ( $\wedge$  or  $\cdot$ ) -conjunction
  2. Logical OR ( $\vee$  or  $+$ ) - disjunction
  3. Logical NOT ( $\bar{x}$ )
- ▶ A clause is  $C_j = \bigvee_{p=1}^{k_j} y_{j_p}$ , where a *literal*  $y_{j_p}$  is  $x_r$  or  $\bar{x}_r$  for some  $r$ .
- ▶ Conjunctive normal form (CNF):

$$F = \bigwedge_{j=1}^m C_j,$$

where  $C_j$  is a clause.

# Satisfiability problem

- ▶ A CNF  $F$  is called satisfiable if there is an assignment of variables such that  $F = 1$  (*TRUE*).
- ▶ **Satisfiability (SAT)** problem: Given  $m$  clauses  $C_1, \dots, C_m$  involving the variables  $x_1, \dots, x_n$ , is the CNF

$$F = \bigwedge_{j=1}^m C_j,$$

satisfiable?

**Theorem (Cook, 1971)**

*SAT is NP-complete.*

# Satisfiability problem

- ▶ **3-Satisfiability (3-SAT)** problem: each clause is restricted to consist of just 3 literals.

## Theorem

*3-SAT is NP-complete.*

### Proof:

1. Obviously, 3-SAT is in  $\mathcal{NP}$ .
2. We will reduce SAT to 3-SAT. Given  $F = \bigwedge_{j=1}^m C_j$ , we will construct a new formula  $F'$  with 3 literals per clause, such that  $F'$  is satisfiable iff  $F$  is satisfiable. We modify each clause  $C_j$  as follows:

## Satisfiability problem

1. If  $C_j$  has three literals, do nothing.
2. If  $C_j$  has  $k > 3$  literals,  $C_j = \bigvee_{p=1}^k \lambda_p$ , we replace  $C_j$  by  $k - 2$  clauses

$$(\lambda_1 \vee \lambda_2 \vee x_1) \wedge (\bar{x}_1 \vee \lambda_3 \vee x_2) \wedge (\bar{x}_2 \vee \lambda_4 \vee x_3) \wedge \cdots \wedge (\bar{x}_{k-3} \vee \lambda_{k-1} \vee \lambda_k),$$

where  $x_1, \dots, x_{k-3}$  are new variables. The new clauses are satisfiable iff  $C_j$  is.

3. If  $C_j = \lambda$ , we replace  $C_j$  by  $\lambda \vee y \vee z$ , and if  $C_j = \lambda \vee \lambda'$ , we replace it by  $\lambda \vee \lambda' \vee y$ . We add to the formula the clauses

$$\begin{aligned} &(\bar{z} \vee \alpha \vee \beta) \wedge (\bar{z} \vee \bar{\alpha} \vee \beta) \wedge (\bar{z} \vee \alpha \vee \bar{\beta}) \wedge (\bar{z} \vee \bar{\alpha} \vee \bar{\beta}) \wedge \\ &(\bar{y} \vee \alpha \vee \beta) \wedge (\bar{y} \vee \bar{\alpha} \vee \beta) \wedge (\bar{y} \vee \alpha \vee \bar{\beta}) \wedge (\bar{y} \vee \bar{\alpha} \vee \bar{\beta}), \end{aligned}$$

where  $y, z, \alpha, \beta$  are new variables. This makes sure that  $z$  and  $y$  are false in any truth assignment satisfying  $F'$ , so that the clauses  $C_j = \lambda$  and  $C_j = \lambda \vee \lambda'$  are equivalent to their replacements.

# Complement graph and induced subgraph

- ▶  $G = (V, E)$  is a simple undirected graph,  $V = \{1, 2, \dots, n\}$ .
- ▶  $\overline{G} = (V, \overline{E})$ , is the *complement graph* of  $G = (V, E)$ , where  $\overline{E} = \{(i, j) \mid i, j \in V, i \neq j \text{ and } (i, j) \notin E\}$ .
- ▶ For  $S \subseteq V$ ,  $G(S) = (S, E \cap S \times S)$  the *subgraph induced by*  $S$ .

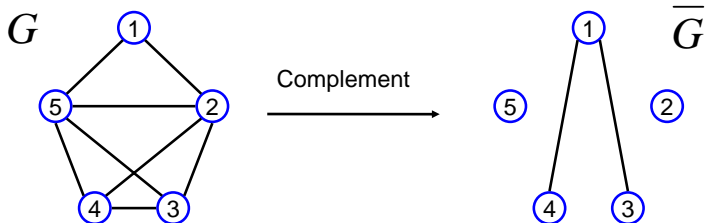
# Cliques and independent sets

- ▶ A subset of vertices  $C \subseteq V$  is called a *clique* if  $G(C)$  is a complete graph.
- ▶ A subset  $I \subseteq V$  is called an *independent set* (stable set, vertex packing) if  $G(I)$  has no edges.
- ▶  $C$  is a clique in  $G$  if and only if  $C$  is an independent set in  $\bar{G}$ .

# Cliques and independent sets

- ▶ A clique (independent set) is said to be
  - *maximal*, if it is not a subset of any larger clique (independent set);
  - *maximum*, if there is no larger clique (independent set) in the graph.
- ▶  $\omega(G)$  – the *clique number* of  $G$ .
- ▶  $\alpha(G)$  – the *independence (stability) number* of  $G$ .

# Cliques and independent sets



$\{1,2,5\}$  : maximal clique

$\{1,4\}$  : maximal  
independent set

$\{2,3,4,5\}$  : maximum clique

$\{1,2,5\}$  : maximal  
independent set

$\{1,4\}$  : maximal clique

$\{2,3,4,5\}$  : maximum  
independent set



# CLIQUE problem

The recognition version: CLIQUE problem.

*CLIQUE: Given a simple undirected graph  $G = (V, E)$  and an integer  $k$ , does there exist a clique of size  $\geq k$  in  $G$ ?*

## Theorem

*CLIQUE is NP-complete.*

# CLIQUE problem

## Proof:

1.  $\text{CLIQUE} \in NP$ . Certificate: a clique of size  $k$ , can be verified in  $O(k^2)$  time.
2. 3-SAT is polynomially reducible to CLIQUE.

Given  $F = \bigwedge_{i=1}^m C_i$ , we build a graph  $G = (V, E)$  as follows:

- ◇ The vertices correspond to pairs  $(C_i, x_{ij})$ , where  $x_{ij}$  is a literal in  $C_i (i = 1, \dots, m; j = 1, 2, 3)$ .
- ◇ Two vertices  $(C_i, x_{ij})$  and  $(C_p, x_{pq})$  are connected by an edge iff

$$i \neq p \text{ and } x_{ij} \neq \bar{x}_{pq}.$$

Then  $F$  is satisfiable iff  $G$  has a clique of size  $m$ . □

# Vertex cover

Given a simple undirected graph  $G = (V, E)$ , a subset  $C \subseteq V$  of vertices is called a **vertex cover** if each edge in  $E$  has at least one endpoint in  $C$ .

$I$  is a maximum independent set of  $G$



$I$  is a maximum clique of  $\bar{G}$



$V \setminus I$  is a minimum vertex cover of  $G$ .

## Corollary

*Maximum clique, maximum independent set and minimum vertex cover problems are NP-hard.*

# Matchings

- ▶ Given a simple undirected graph  $G = (V, E)$ , a set  $M \subseteq E$  of independent edges is called a **matching**.
- ▶ The maximum matching problem can be solved in polynomial time (e.g., Edmond's algorithm)

## Theorem (König)

*The maximum cardinality of a matching in a bipartite graph is equal to the minimum cardinality of a vertex cover.*

## Corollary

*Maximum independent set and minimum vertex cover problems are polynomially solvable in bipartite graphs.*

# Construction heuristics

construct a feasible solution from scratch

- ◇ random construction;
- ◇ greedy construction;
- ◇ other.

# Greedy algorithms

- ▶ Usually a feasible solution of a combinatorial optimization problem consists of parts, such as nodes, edges, cities visited, etc.
- ▶ The construction of a feasible solution can be performed by adding one such part at a time, i.e., at each step we select one of the elements from a candidate set, from which a solution is created.
- ▶ **Greedy algorithms** select such an element according to a greedy selection function.
- ▶ The greedy selection function is such that it makes locally best feasible contribution to the constructed solution with respect to the problem's objective function.

# Greedy algorithms

In some cases, greedy algorithms produce globally optimal solutions:

- ◇ Minimum weight spanning tree
- ◇ Real knapsack

However, this is usually not the case for NP-hard problems.

Example ( “Nearest neighbor” strategy for TSP)

Each time visit the closest city that have not yet been visited.

# A greedy algorithm for MIS

0. Set  $I = \emptyset$ .
1. If  $G$  has no vertices then stop; otherwise choose a vertex  $v$  with the smallest degree  $d$  in the current graph.
2. Add  $v$  to  $I$ , delete  $v$  and all its neighbors from  $G$ , and return to step 1.

## Theorem (Erdős)

*Upon the termination of the above algorithm we have  $|I| \geq \frac{n}{\delta+1}$ , where  $n = |V|$  and  $\delta = \frac{2|E|}{n}$  (i.e., the average degree of a vertex in  $G$ ).*



# A greedy algorithm for MIS

Proof:

- ▶ Note that when we remove a vertex of degree  $d_i$  and its neighbors from the graph, we eliminate a total of  $d_i + 1$  vertices with the total sum of degrees at least  $d_i(d_i + 1)$ .
- ▶ Let  $|I| = q$  upon termination, then

$$\sum_{i=1}^q d_i(d_i + 1) \leq n\delta \text{ and } \sum_{i=1}^q (d_i + 1) = n$$

- ▶ By adding these two equations together, we obtain

$$n(\delta + 1) \geq \sum_{i=1}^q (d_i + 1)^2.$$

## A greedy algorithm for MIS

- ▶ Recall the Cauchy-Schwarz inequality in  $\mathbb{R}^n$ :

$$\left(\sum_{i=1}^n x_i y_i\right)^2 \leq \left(\sum_{i=1}^n x_i^2\right) \left(\sum_{i=1}^n y_i^2\right)$$

- ▶ In particular, if  $y_i = 1, i = 1, \dots, n$  then we have:

$$\left(\sum_{i=1}^n x_i\right)^2 \leq n \sum_{i=1}^n x_i^2$$

- ▶ So, for  $\sum_{i=1}^q (d_i + 1)^2$  we obtain

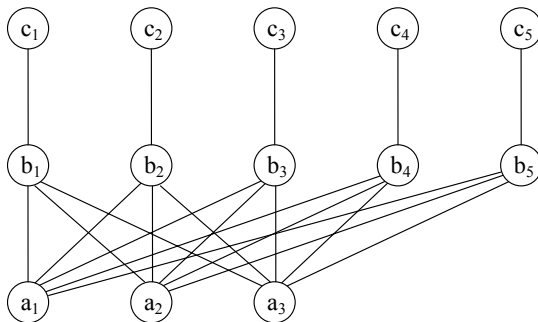
$$n(\delta + 1) \geq \sum_{i=1}^q (d_i + 1)^2 \geq \frac{\left(\sum_{i=1}^q (d_i + 1)\right)^2}{q} = \frac{n^2}{q}$$



# A greedy algorithm for minimum vertex cover

0. Set  $C = \emptyset$ .
1. If  $G$  has no edges then stop; otherwise choose a vertex  $v$  with the largest degree  $d$  in the current graph.
2. Add  $v$  to  $C$ , delete  $v$  from  $G$ , and return to step 1.

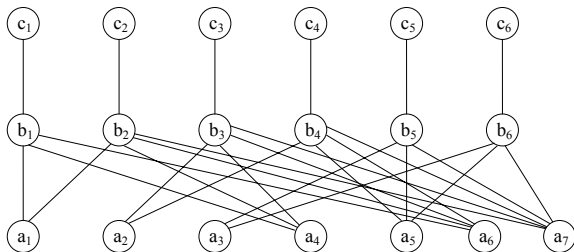
## A greedy algorithm for minimum vertex cover



- ◇ Using the greedy algorithm:  $C = \{a_1, a_2, a_3, c_1, c_2, c_3, c_4, c_5\}$ .
- ◇ Minimum vertex cover:  $C^* = \{b_1, b_2, b_3, b_4, b_5\}$ .
- ◇ If we had  $n$   $a_i$ 's and  $n + 2$   $b_i$ 's and  $c_i$ 's then

$$\frac{|C|}{|C^*|} = \frac{2n + 2}{n + 2} \rightarrow 2, \quad n \rightarrow \infty.$$

# A greedy algorithm for minimum vertex cover



- ◇ Using the greedy algorithm:  
 $C = \{a_1, a_2, a_3, a_4, a_6, a_7, c_1, c_2, c_3, c_4, c_5, c_6\}$ ;  $|C| = 13$ .
- ◇ Minimum vertex cover:  $C^* = \{b_1, b_2, b_3, b_4, b_5, b_6\}$ ;  $|C^*| = 6$ .
- ◇ So,  $\frac{|C|}{|C^*|} = \frac{13}{6} > 2$ .
- ◇ Similarly, we can construct graphs with any large  $\frac{|C|}{|C^*|}$  ratio.

## Another construction for minimum vertex cover

0. Set  $C = \emptyset$ .

1. While  $E \neq \emptyset$  do

    Choose any edge  $(u, v) \in E$ ;

$G = G \setminus \{u, v\}$ ;

$C = C \cup \{u, v\}$ .

- ◇ Note that the selected edges form a matching of size  $|C|/2$ , so no two of them can be covered by the same vertex.
- ◇ Thus, any vertex cover must be of size at least  $|C|/2$ , i.e.,

$$|C| \leq 2|C^*|.$$

# MAX-CUT problem

Given a complete undirected graph

$$G = (V, E), V = \{1, 2, \dots, n\}, E = V \times V$$

with weights  $w_{ij}, i, j = 1, \dots, n$  on the edges, find a vertex partition

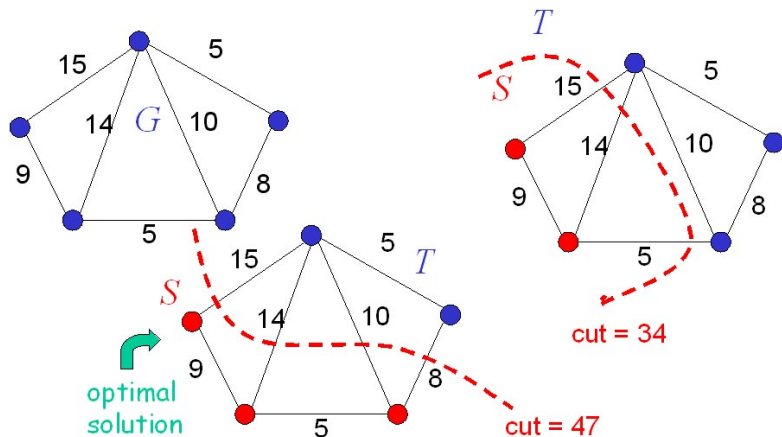
$$S, \bar{S} \subseteq V, S \cup \bar{S} = V, S \cap \bar{S} = \emptyset$$

such that the sum of the weights in the cut  $(S, \bar{S})$ ,

$$\sum_{i \in S, j \in \bar{S}} w_{ij}$$

is maximized.

# MAX-CUT problem: an example





# A greedy algorithm for MAX-CUT

0. Set  $V_1 = \emptyset$ ,  $V_2 = \emptyset$ .
1. Pick an edge  $(i, j)$  of maximum weight in  $G$  and set

$$V_1 = \{i\}, V_2 = \{j\}, V = V \setminus \{i, j\}.$$

2. While  $V \neq \emptyset$  repeat
  - (a) Pick a vertex  $k \in V$ ; let

$$w_1 = \sum_{i \in V_1} w_{ki}, w_2 = \sum_{j \in V_2} w_{kj}$$

- (b) If  $w_1 < w_2$  then  $V_1 = V_1 \cup \{k\}$ , else  $V_2 = V_2 \cup \{k\}$ .
- (c)  $V = V \setminus \{k\}$ .

# A greedy algorithm for MAX-CUT

## Lemma

*Let  $W^*$  be the optimal cut weight and  $W_{cut}$  be the weight of the cut obtained using our greedy algorithm. Then*

$$W_{cut} \geq \frac{1}{2} W^*.$$

## Proof.

- ◇ At each step, the weight added to the cut is  $\leq$  the weight added “inside”  $V_1$  or  $V_2$ .
- ◇ Thus, at least half of the total weight  $W$  of all edges will be in the cut.
- ◇ We have:

$$W_{cut} \geq \frac{1}{2} W \geq \frac{1}{2} W^*.$$



## Edge contraction heuristic for MAX-CUT

Given a complete graph  $G(V, E)$  with edge weights  $w_{ij}, \forall i, j \in V, i \neq j$ , find a cut  $(S_1, S_2)$  and the cut value  $cut(S_1, S_2)$ .

1. For  $j = 1 : |V|$   
     $ContractionList(j) = \{j\}$

2. For  $j = 1 : |V| - 2$

    Find a minimum weight edge  $(x, y)$  in  $G$

$v = contract(x, y)$

$V = V \cup \{v\} \setminus \{x, y\}$

    For  $i \in V \setminus \{v\}$

$w_{vi} = w_{xi} + w_{yi}$

$ContractionList(v) = ContractionList(x) \cup ContractionList(y)$

3. Denote by  $x$  and  $y$  the only 2 vertices in  $V$

$S_1 = ContractionList(x);$

$S_2 = ContractionList(y);$

$cut(S_1, S_2) = w_{xy}$

# Edge contraction heuristic for MAX-CUT

The following lemma will be used to prove an approximation ratio result for the contraction algorithm for the MAX-CUT problem.

## Lemma

*Let  $W_k$  denote the total weight of the first  $k$  edges contracted by the edge contraction heuristic and let  $W$  be the total weight of all edges in the graph. Then for any  $1 \leq k \leq n - 2$*

$$W_k \leq \frac{2kW}{(n-1)(n-k+1)}$$

*where  $n$  is the number of vertices in the input graph.*

# Edge contraction heuristic for MAX-CUT

## Proof.

- ◇ The proof is based on the fact that the weight of a contracted edge is no greater than the average edge weight in the current graph at each iteration.
- ◇ This is true since the procedure always contracts an edge of the smallest weight. So, the weight of the first contracted edge satisfies

$$W_1 \leq \frac{W}{\binom{n}{2}} = \frac{2W}{n(n-1)}.$$

- ◇ We will use induction on  $k$ . We have already shown that the lemma is valid for  $k = 1$ . Assume it is correct for all integer  $k \leq \kappa$ . We need to derive the inequality in lemma for  $k = \kappa + 1$ .

## Edge contraction heuristic for MAX-CUT

Expressing the upper bound on  $W_{\kappa+1}$  through  $W_{\kappa}$  and  $W$ , and using the induction assumption for  $W_{\kappa}$ , we obtain:

$$\begin{aligned}W_{\kappa+1} &\leq W_{\kappa} + \frac{W - W_{\kappa}}{\binom{n-\kappa}{2}} \\&= \frac{(n-\kappa)(n-\kappa-1)-2}{(n-\kappa)(n-\kappa-1)} W_{\kappa} + \frac{2W}{(n-\kappa)(n-\kappa-1)} \\&\leq \frac{2W}{(n-\kappa)(n-\kappa-1)} \left( \frac{((n-\kappa)(n-\kappa-1)-2)\kappa}{(n-1)(n-\kappa+1)} + 1 \right) \\&= \frac{2(\kappa+1)W(n-\kappa+1)(n-\kappa-1)}{(n-\kappa)(n-\kappa-1)(n-1)(n-\kappa+1)} \\&= \frac{2(\kappa+1)W}{(n-1)(n-\kappa)}.\end{aligned}$$

Thus, by induction the lemma is correct. □

# Edge contraction heuristic for MAX-CUT

## Theorem

Denote by  $W_{cut}$  the value of the cut obtained using the edge contraction heuristic and by  $W^*$  the weight of an optimal cut. Then

$$W_{cut} \geq \frac{1}{3} \left( 1 + \frac{2}{n-1} \right) W$$

and, in particular,

$$W_{cut} > \frac{1}{3} W^*.$$

Here, as before,  $W$  denotes the total weight of all edges in the graph.

**Proof.**  $W_{cut} = W - W_{n-2}$ , thus from the above lemma

$$W_{cut} \geq W - \frac{2(n-2)W}{(n-1)(n-(n-2)+1)} = \frac{1}{3} \left( 1 + \frac{2}{n-1} \right) W,$$

and since  $W \geq W^*$ , we obtain  $W_{cut} > \frac{1}{3} W^*$ .



# A better approximation algorithm for MAX-CUT

- ▶ Good quality bounds are also crucial in estimating quality of the solutions obtained using heuristics or approximation algorithms.
- ▶ Bounds are typically obtained using convex relaxations.
- ▶ Next we discuss an example of an approximation algorithm for the classical MAX-CUT problem based on a [semidefinite programming relaxation](#).



# Binary formulation for MAX-CUT

We can formulate the MAX-CUT problem as the following **strict binary quadratic program** (*strict* since no monomials of degree 1 are involved):

$$\begin{array}{ll} \max & \frac{1}{2} \sum_{1 \leq i < j \leq n} w_{ij} (1 - x_i x_j) \\ \text{s.t.} & x_i^2 = 1, \ i \in V. \end{array}$$

We will first relax this program to a vector program and then will show that our relaxation is equivalent to a certain semidefinite program.

# Vector programs

A **vector program** is the problem of optimizing a linear function of the inner products  $v_i^T v_j$ ,  $1 \leq i \leq j \leq n$ , where  $v_1, \dots, v_n$  are vector variables in  $\mathbb{R}^n$ , subject to linear constraints of these inner products.

We can obtain a vector program corresponding to a strict binary quadratic program as follows:

- ▶ Define  $n$  vector variables corresponding to the  $n$  binary variables in the quadratic program (e.g.,  $x_i \rightarrow v_i$ );
- ▶ Replace each degree 2 term with the corresponding inner product (i.e.,  $x_i x_j \rightarrow v_i^T v_j$ ).

# Vector program for MAX-CUT

For the MAX-CUT formulation,

$$\begin{array}{ll}\max & \frac{1}{2} \sum_{1 \leq i \leq j \leq n} w_{ij} (1 - x_i x_j) \\ \text{s.t.} & x_i^2 = 1, \quad i \in V,\end{array}$$

we obtain the following vector program:

$$\begin{array}{ll}\max & \frac{1}{2} \sum_{1 \leq i \leq j \leq n} w_{ij} (1 - v_i^T v_j) \\ \text{s.t.} & v_i^T v_i = 1, \quad i \in V.\end{array}$$

Note that for any feasible solution of the original formulation, the vector  $v_i = (x_i, 0, \dots, 0)^T \in \mathbb{R}^n$  has the same objective function value in the vector program. Therefore, the vector program is a relaxation of the binary quadratic program.

# Semidefinite optimization

A semidefinite program (SDP) is formulated as follows:

$$\begin{aligned} & \min \langle C, X \rangle_F, \\ \text{s.t. } & \langle A_i, X \rangle_F = b_i, \quad i = 1, \dots, m, \\ & X \in \mathcal{P}_n, \end{aligned} \tag{1}$$

where  $C$  and  $A_i$  are some matrices from  $S^{n \times n}$  and the Frobenius inner product on  $S^{n \times n}$  is defined by

$$\langle X, Y \rangle_F = \sum_{i=1}^n \sum_{j=1}^n X^{(i,j)} Y^{(i,j)} = \text{trace}(XY),$$

for any  $X, Y \in S^{n \times n}$ . Here  $S^{n \times n}$  is the linear space of symmetric  $n \times n$ -matrices and  $\mathcal{P}_n \subset S^{n \times n}$  is the cone of positive semidefinite  $n \times n$ -matrices.

SDP is polynomially solvable

# SDP relaxation for MAX-CUT

Next we show that vector programs are equivalent to SDP.

Let  $(V)$  be a vector program on  $n$   $n$ -dimensional vector variables  $v_1, \dots, v_n$ .

We define the corresponding SDP  $(S)$  as follows:

- ▶ replace each inner product  $v_i^T v_j$  in  $(V)$  with the variable  $y_{ij}$
- ▶ Require that the matrix  $Y = (y_{ij})_{i,j=1}^n$  is symmetric and positive semidefinite.

## Lemma

*The vector program  $(V)$  and the SDP  $(S)$  are equivalent.*

## Proof.

Correspondence between solutions of  $(V)$  and  $(S)$ :

- ▶  $w_1, \dots, w_n$  - a feasible solution of  $(V)$ ;
- ▶  $W$  - the matrix with columns  $w_1, \dots, w_n$ ;
- ▶  $Y = W^T W$  is a feasible solution of  $(S)$ .

# SDP relaxation for MAX-CUT

For the vector program relaxing the MAX-CUT formulation,

$$\begin{array}{ll}\max & \frac{1}{2} \sum_{1 \leq i \leq j \leq n} w_{ij} (1 - v_i^T v_j) \\ \text{s.t.} & v_i^T v_i = 1, \quad i \in V.\end{array}$$

we obtain the following equivalent SDP:

$$\begin{array}{ll}\max & \frac{1}{2} \sum_{1 \leq i \leq j \leq n} w_{ij} (1 - y_{ij}) \\ \text{s.t.} & y_{ii} = 1, \quad i \in V, \\ & Y \in \mathcal{P}_n.\end{array}$$

# Randomized rounding algorithm

Assume that we have an optimal solution  $v_1^*, \dots, v_n^*$  of our vector program for MAX-CUT with  $OPT_V$  being the corresponding objective function value.

Note that vectors  $v_1^*, \dots, v_n^*$  all lie on the  $n$ -dimensional unit sphere.

Our goal is to obtain a cut  $(S, \bar{S})$  whose weight is a large fraction of  $OPT_V$ .

Let  $\theta_{ij}$  denote the angle between  $v_i^*$  and  $v_j^*$ . Since  $v_i^{*T} v_j^* = \cos \theta_{ij}$ , the contribution of the pair  $v_i^*, v_j^*$  to  $OPT_V$  is given by

$$\frac{w_{ij}}{2}(1 - \cos \theta_{ij}).$$

We want to separate  $v_i^*$  and  $v_j^*$  if  $\theta_{ij}$  is large.

# Randomized rounding algorithm

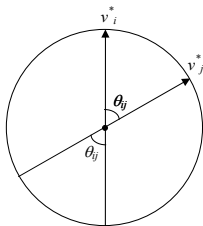
**Randomized rounding:** Let  $r$  be a uniformly distributed vector on the  $n$ -dimensional unit sphere. Set

$$S = \{i : v_i^{*T} r \geq 0\}.$$

Lemma

$$\text{Prob}(i \text{ and } j \text{ are separated}) = \frac{\theta_{ij}}{\pi}.$$

Proof.



Project  $r$  onto the plane defined by  $v_i^*$  and  $v_j^*$ . The vertices  $i$  and  $j$  will be separated if and only if the projection lies on one of the two arcs of angle  $\theta_{ij}$ . Since  $r$  has been picked from the uniform distribution over the sphere, its projection is a random direction on the plane.





# Randomized rounding algorithm

## Lemma

Let  $x_1, \dots, x_n$  be independent random variables with the standard normal distribution  $N(0, 1)$ . Then

$$\frac{1}{\sqrt{x_1^2 + \dots + x_n^2}}(x_1, \dots, x_n)^T$$

is a random vector on the unit sphere in  $\mathbb{R}^n$ .

## Proof.

Let  $r = (x_1, \dots, x_n)^T$ . Then the distribution function for  $r$  has density

$$f(y_1, \dots, y_n) = \prod_{i=1}^n \frac{1}{\sqrt{2\pi}} e^{-y_i^2/2} = \frac{1}{(2\pi)^{n/2}} e^{-\frac{1}{2} \sum_i y_i^2}.$$

Since it depends only on the distance of  $(y_1, \dots, y_n)$  from the origin, the distribution of  $r$  is spherically symmetric.



# Randomized rounding algorithm

## Algorithm:

1. Find an optimal solution  $v_1^*, \dots, v_n^*$  of the vector program for MAX-CUT.
2. Pick a vector  $r$  uniformly distributed over the  $n$ -dimensional unit sphere.
3. Set  $S = \{i : v_i^{*T} r \geq 0\}$ .

Denote by  $W_A$  the random variable representing the weight of edges in the cut obtained from the algorithm.

We want to find  $\alpha$  for which

$$E[W_A] \geq \alpha \cdot OPT_v.$$

# Randomized rounding algorithm

Note that

$$\begin{aligned} E[W_A] &= \sum_{1 \leq i < j \leq n} w_{ij} \text{Prob}(i \text{ and } j \text{ are separated}) \\ &= \sum_{1 \leq i < j \leq n} w_{ij} \frac{\theta_{ij}}{\pi} \end{aligned}$$

and

$$OPT_v = \sum_{1 \leq i < j \leq n} \frac{w_{ij}}{2} (1 - \cos \theta_{ij}).$$

Hence, if we select  $\alpha$  so that

$$\frac{\theta_{ij}}{\pi} \geq \alpha \frac{1}{2} (1 - \cos \theta_{ij})$$

for all  $i, j \in V$  then  $E[W_A] \geq \alpha \cdot OPT_v$ . We can take

$$\alpha = \min_{0 \leq \theta \leq \pi} \frac{2}{\pi} \frac{\theta}{(1 - \cos \theta)}.$$

# Randomized rounding algorithm

Thus, we have shown that the following statement is correct:

## Lemma

$$E[W_A] \geq \alpha \cdot OPT_v,$$

where

$$\alpha = \min_{0 \leq \theta \leq \pi} \frac{2}{\pi} \frac{\theta}{(1 - \cos \theta)}.$$

It can be shown that  $\alpha > 0.87856$ .

## Theorem

*There is a randomized approximation algorithm for MAX-CUT with an approximation ratio of 0.87856.*

# Randomized rounding algorithm

## Proof.

Recall the following information concerning the randomized algorithms.

In Monte-Carlo algorithms, the answer has to be correct with “high probability” in a predefined run time.

For a problem of size  $n$ , we say that  $p = 1 - \frac{1}{n^\gamma}$ , where  $\gamma > 1$ , is a high probability.

We say that a randomized algorithm takes  $\tilde{O}(f(n))$  of a resource (such as space or time), if there exist some  $C, n_0 > 0$  such that for any  $n \geq n_0$  the amount of resource used is  $\leq C\gamma f(n)$  with probability  $\geq 1 - \frac{1}{n^\gamma}, \gamma > 1$ .

We need to obtain a “high probability” result using the bound  $E[W_A] \geq \alpha \cdot OPT_v$ .

## Randomized rounding algorithm

Denote by  $W$  the sum of all edge weights in the graph, and let  $a, p$  be such that

$$E[W_A] = aW; \quad p = \text{Prob}(W_A < (1 - \epsilon)aW),$$

where  $\epsilon$  is a constant.  $W_A$  cannot be greater than  $W$ , hence

$$aW \leq p(1 - \epsilon)aW + (1 - p)W \quad \Rightarrow \quad p \leq \frac{1 - a}{1 - a + a\epsilon}.$$

Since  $OPT \geq W/2$  (exercise), we have

$$W \geq E[W_A] = aW \geq \alpha \cdot OPT_v \geq \alpha \cdot OPT \geq \frac{\alpha W}{2},$$

and thus  $\alpha/2 \leq a \leq 1$ . Therefore, we get

$$p \leq \frac{1 - a}{1 - a + a\epsilon} \leq 1 - \frac{\epsilon\alpha/2}{1 + \epsilon\alpha/2 - \alpha/2} = 1 - c,$$

where  $c = \frac{\epsilon\alpha/2}{1 + \epsilon\alpha/2 - \alpha/2}$ .

## Randomized rounding algorithm

Thus, for  $p = \text{Prob}(W_A < (1 - \epsilon)aW)$  we have  $p \leq 1 - c$ , so after  $k$  iterations we obtain

$$\text{Prob}(W_A \geq (1 - \epsilon)aW) \geq 1 - (1 - c)^k.$$

To achieve high probability, we need to have

$$1 - (1 - c)^k \geq 1 - n^{-\gamma} \Leftrightarrow k \geq -\frac{\gamma}{\ln(1 - c)} \ln n.$$

Since  $aW \geq \alpha \cdot \text{OPT} \geq 0.87856\text{OPT}$ , we can pick a value of  $\epsilon > 0$  such that  $(1 - \epsilon)aW \geq 0.87856\text{OPT}$ . Therefore, denoting by  $C = -\frac{1}{\ln(1 - c)}$ , after at most  $C\gamma \ln n$  iteration we have

$$\text{Prob}(W_A \geq 0.87856\text{OPT}) \geq 1 - n^{-\gamma}.$$

Thus, the algorithm approximates the MAX-CUT problem with the approximation ratio of 0.87856 in  $\tilde{O}(\ln n)$  time. □

# Minimizing makespan for identical parallel machines

- ▶  $n$  jobs  $J_1, \dots, J_n$
- ▶  $m$  identical machines  $M_1, \dots, M_m$
- ▶  $p_j$  is the processing time for job  $J_j$
- ▶ Each machine can process at most one job at a time
- ▶ The objective is to minimize the makespan (the time of completion of the last job)



# List scheduling (Graham)

**List Scheduling (LS):** Given a list of jobs in an arbitrary order, assign the next job in the list whenever a machine becomes available.

To analyze this algorithm, we introduce the following notations:

- ▶  $s_i$  - the start time for job  $J_i$
- ▶  $c_i$  - the completion time for  $J_i$
- ▶  $c_{max}^{LS}$  - the makespan obtained using the LS algorithm
- ▶  $c_{max}^*$  - the optimal makespan

# List scheduling (Graham)

- ▶ Let  $J_k$  be the last job assigned by LS algorithm
- ▶ There cannot be an idle machine prior to  $s_k$
- ▶ Job  $J_k$  has to be processed, so

$$c_{max}^* \geq p_k$$

- ▶ The sum of  $n$  processing times is split between  $m$  machines, so

$$c_{max}^* \geq \frac{1}{m} \sum_{j=1}^n p_j$$

(with equality when all machines complete processing simultaneously)

# List scheduling (Graham)

$$\begin{aligned}c_{max}^* &\geq p_k; \\c_{max}^* &\geq \frac{1}{m} \sum_{j=1}^n p_j\end{aligned}$$

$$\begin{aligned}c_{max}^{LS} &= c_k = s_k + p_k \\&\leq \frac{1}{m} \sum_{j \neq k}^n p_j + p_k \\&= \frac{1}{m} \sum_{j=1}^n p_j + \left(1 - \frac{1}{m}\right) p_k \\&\leq c_{max}^* + \left(1 - \frac{1}{m}\right) c_{max}^* \\&= \left(2 - \frac{1}{m}\right) c_{max}^*\end{aligned}$$

# List scheduling (Graham)

A worst case example:

- ▶  $m(m - 1) + 1$  jobs
- ▶ The first  $m(m - 1)$  jobs in the list have processing time 1
- ▶ The last job has processing time  $m$
- ▶ We have

$$c_{max}^{LS} = m - 1 + m = 2m - 1$$

$$c_{max}^* = m$$

$$\frac{c_{max}^{LS}}{c_{max}^*} = \frac{2m - 1}{m} = 2 - \frac{1}{m}$$

# The LPT rule

LPT (longest processing time) rule: apply list scheduling, ordering the job list by non-increasing processing times.

- ▶ Intuitively this should help, since the worst-case example for list scheduling schedules a very long job last.
- ▶  $c_{max}^{LPT}$  - the makespan obtained using the LPT algorithm
- ▶  $c_{max}^*$  - the optimal makespan

## Theorem (Graham)

$$c_{max}^{LPT} \leq \left( \frac{4}{3} - \frac{1}{3m} \right) c_{max}^*.$$

# The LPT rule

Proof:

- ▶ Assume that the jobs are indexed so that

$$p_1 \geq p_2 \geq \cdots \geq p_n.$$

- ▶ Let  $J_k$  be the job that completes last using LPT rule.
- ▶ We consider two cases:
  1.  $p_k \leq C_{max}^*/3$ ;
  2.  $p_k > C_{max}^*/3$

# The LPT rule

Case 1:

$$\begin{aligned}\frac{1}{3}c_{\max}^* &\geq p_k; \\ c_{\max}^* &\geq \frac{1}{m} \sum_{j=1}^n p_j\end{aligned}$$

$$\begin{aligned}c_{\max}^{LPT} &= c_k = s_k + p_k \\ &\leq \frac{1}{m} \sum_{j \neq k}^n p_j + p_k \\ &= \frac{1}{m} \sum_{j=1}^n p_j + \left(1 - \frac{1}{m}\right) p_k \\ &\leq c_{\max}^* + \frac{1}{3} \left(1 - \frac{1}{m}\right) c_{\max}^* \\ &= \left(\frac{4}{3} - \frac{1}{3m}\right) c_{\max}^*\end{aligned}$$

# The LPT rule

Case 2:  $p_k > C_{max}^*/3$ .

- ▶ Without loss of generality we can assume that  $J_k$  was the last job on the list.
  - ▶ If it was not, we can ignore the jobs after - the LPT will perform at least as badly with respect to the optimal schedule in the resulting instance.
- ▶ For such an instance, at most two jobs can be processed on any machine. Suppose that the instance has  $2m - h$  jobs total.
- ▶ Optimal schedule:
  - ▶ schedule jobs  $J_1, \dots, J_h$  alone
  - ▶ pair up the jobs  $\{J_{h+1}, J_n\}, \{J_{h+2}, J_{n-1}\}, \dots$
  - ▶ this schedule corresponds to the LPT rule  $\Rightarrow$  LPT rule is optimal in this case



# Scheduling with release dates and delivery times

- ▶  $n$  jobs  $J_1, \dots, J_n$
- ▶ 1 machine
- ▶  $r_j$  - the release date for job  $J_j$
- ▶  $p_j$  - the processing time for job  $J_j$
- ▶  $q_j$  - the delivery time for job  $J_j$
- ▶ The objective is to schedule the jobs so that the lateness (the time of the last delivery) is minimized

# Scheduling with release dates and delivery times

**Jackson's rule:** Whenever the machine becomes available, schedule the first **available** (i.e., released) job on the list.

- ▶  $s_j$  - starting time for job  $J_j$
- ▶  $c_j$  - completion time for job  $J_j$
- ▶ Let  $L_{max}^J$  be the lateness obtained using the Jackson's rule
- ▶ Let  $J_k$  be the last job delivered using Jackson's algorithm
- ▶ Let  $P = \sum_{j=1}^n p_j$

$$\begin{aligned} L_{max}^J &= c_k + q_k = s_k + p_k + q_k \leq (r_k + P) + p_k + q_k \\ &\leq (r_k + p_k + q_k) + P \leq 2L_{max}^* \end{aligned}$$

# Scheduling with release dates and delivery times

## Example

- ▶ 2 jobs:

$$r_1 = q_1 = 0$$

$$r_2 = p_2 = 1$$

$$p_1 = q_2 = M$$

- ▶ Jackson's rule:  $J_1, J_2 \Rightarrow L_{max}^J = M + 1 + M = 2M + 1$
- ▶ Optimal:  $J_2, J_1 \Rightarrow L_{max}^* = 1 + 1 + M = M + 2$

$$\frac{2M + 1}{M + 2} \rightarrow 2, \quad M \rightarrow \infty$$

# Scheduling with precedence constraints, release dates and delivery times

- ▶  $n$  jobs  $J_1, \dots, J_n$
- ▶  $M$  machines
- ▶  $J_{j_1} \prec J_{j_2}$  means that job  $j_1$  has to be completed before job  $j_2$  is started
- ▶  $r_j$  - the release date for job  $J_j$
- ▶  $p_j$  - the processing time for job  $J_j$
- ▶  $q_j$  - the delivery time for job  $J_j$
- ▶ The objective is to schedule the jobs so that the lateness (the time of the last delivery) is minimized

# Scheduling with precedence constraints, release dates and delivery times

- ▶  $s_j$  - starting time for job  $J_j$
- ▶  $c_j$  - completion time for job  $J_j$
- ▶ The lateness is given by  $L_{max} = \max_j \{c_j + q_j\}$
- ▶ The job  $J_j$  cannot be started before release time  $r_j$
- ▶ The job  $J_j$  cannot be started if one of the preceding jobs has not been completed
- ▶ A job  $J_j$  is **available** if the current time is  $\geq r_j$  and all preceding jobs have been completed

# Scheduling with precedence constraints, release dates and delivery times

**Construction heuristic** (a variant of list scheduling): List all jobs in an arbitrary order. Schedule the first **available** job from the list whenever there is an idle machine.

- ▶ Let  $J_{k_1}$  be the last job delivered:

$$s_{k_1} + p_{k_1} + q_{k_1} = L_{max}$$

- ▶ Denote by  $t_{k_1}$  the latest time before  $s_{k_1}$  when there was an idle machine
- ▶ Job  $J_{k_1}$  was not scheduled at  $t_{k_1}$ , it was not available for one of the two reasons:
  1. one of its predecessors  $J_{k_2}$  had not been processed
  2. it had not been released

# Scheduling with precedence constraints, release dates and delivery times

- ▶ Repeating this argument, we eventually construct a sequence of jobs

$$J_{k_l} \prec J_{k_{l-1}} \prec \cdots \prec J_{k_2} \prec J_{k_1}$$

such that  $J_{k_l}$  is not available at time  $t_{k_l}$  because it has not been released

- ▶ Consider the union of intervals

$$I_1 = [0, r_{k_l}) \bigcup_{i=1}^{l-1} [s_{k_i}, c_{k_i}) \bigcup [c_{k_1}, c_{k_1} + q_{k_1}]$$

- ▶ The total length of  $I_1$  is the minimum time required to process jobs  $J_{k_1}, \dots, J_{k_l}$ . So,

$$\text{length}(I_1) \leq L_{\max}^*$$

# Scheduling with precedence constraints, release dates and delivery times

- ▶ Let

$$I_2 = [0, L_{\max}] \setminus I_1.$$

- ▶ All machines are busy during any time in  $I_2$ , so

$$\text{length}(I_2) \leq L_{\max}^*$$

- ▶ Thus,

$$L_{\max} = \text{length}([0, L_{\max}]) = \text{length}(I_1) + \text{length}(I_2) \leq 2L_{\max}^*$$



# Minimizing makespan for unrelated parallel machines

- ▶  $n$  jobs  $J_1, \dots, J_n$
- ▶  $m$  unrelated machines  $M_1, \dots, M_m$
- ▶  $p_{ij}$  is the processing time for job  $J_j$  on machine  $M_i$
- ▶ Each machine can process at most one job at a time
- ▶ The objective is to minimize the makespan (the time of completion of the last job)

# Integer programming formulation

Minimize  $t$

$$\begin{aligned} \text{subject to} \quad & \sum_{i=1}^m x_{ij} = 1, & j = 1, \dots, n; \\ & \sum_{j=1}^n p_{ij} x_{ij} \leq t, & i = 1, \dots, m; \\ & x_{ij} \in \{0, 1\}, & i = 1, \dots, m, j = 1, \dots, n. \end{aligned}$$

- ◇ Consider the LP relaxation of this IP
- ◇ A **basic** optimal solution to this LP has at most  $m + n$  positive variables (since the number of constraints is  $m + n$ )
- ◇ Since  $t$  is positive, at most  $m + n - 1$  of  $x_{ij}$ 's are positive
- ◇ Every job has at least one positive variable associated with it
- ◇ Thus, at most  $m - 1$  jobs are split onto two or more machines

# Constructing a feasible schedule

Two stages:

1. Jobs with integer assignments in the LP solution are assigned according to the LP solution
  - ▶ the length of this piece of schedule is  $\leq C_{max}^{LP} \leq C_{max}^*$
2. For the remaining set of at most  $m - 1$  jobs, we will construct an optimal assignment.
  - ▶ the length of this piece of schedule is  $\leq C_{max}^*$

By concatenating the two schedules we will obtain a schedule of length at most  $2C_{max}^*$

Complete enumeration can be performed in time at most  $O(m^{m-1})$  – exponential in the number of machines.

# Can we do better?

- ◇ Observe that since we need to assign  $m - 1$  jobs (at most) to  $m$  machines, on average less than one job is assigned to each machine.
- ◇ Can we assign at most one job to each machine so that each machine would get a job with a reasonably small processing time?

## 2-approximation LP-based algorithm

Lenstra, Shmoys, and Tardos proposed a 2-relaxed decision procedure:

Given a “target” length of  $t$ , the procedure will perform one of the following two actions:

- ▶ correctly deduce that no schedule of length  $t$  exists
- ▶ construct a schedule with makespan at most  $2t$  (possibly even if no schedule of length  $t$  exists)

This procedure can be converted into a 2-approximation algorithm using binary search.

# Schedule with a target length

Given the target schedule length  $t$ , we consider the following sets of indices:

- ◇ for each job  $J_j$ :

$$\mathcal{M}_t(j) = \{i : p_{ij} \leq t\}$$

- ◇ for each machine  $M_i$ :

$$\mathcal{J}_t(i) = \{j : p_{ij} \leq t\}$$

In other words,

- ◇  $\mathcal{M}_t(j)$  is the set of all machines that  $J_j$  could possibly be scheduled on in a schedule of length  $\leq t$ ;
- ◇  $\mathcal{J}_t(i)$  is the set of all jobs that could be processed on  $M_i$  in a schedule of length at most  $t$ .

# A rounding theorem

## Theorem

For the matrix  $P = [p_{ij}]$ , some vector  $d$  and a positive integer  $t$ , consider the following linear feasibility system, which we denote by  $LP(P, d, t)$ :

$$\begin{aligned}\sum_{i \in \mathcal{M}_t(j)} x_{ij} &= 1, & j = 1, \dots, n; \\ \sum_{j \in \mathcal{J}_t(i)} p_{ij} x_{ij} &\leq d_i, & i = 1, \dots, m; \\ x_{ij} &\geq 0, & i = 1, \dots, m, j = 1, \dots, n.\end{aligned}$$

If  $LP(P, d, t)$  has a feasible solution, then any vertex  $\tilde{x}$  of the polytope defined by  $LP(P, d, t)$  can be rounded to a feasible solution  $\bar{x}$  of the following integer program  $IP(P, d, t)$ :

$$\begin{aligned}\sum_{i \in \mathcal{M}_t(j)} x_{ij} &= 1, & j = 1, \dots, n; \\ \sum_{j \in \mathcal{J}_t(i)} p_{ij} x_{ij} &\leq d_i + t, & i = 1, \dots, m; \\ x_{ij} &\in \{0, 1\} & i = 1, \dots, m, j = 1, \dots, n.\end{aligned}$$

# A rounding theorem

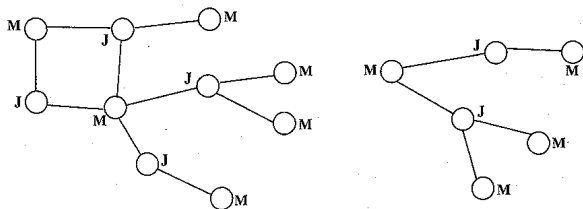
$$\begin{aligned}\sum_{i \in \mathcal{M}_t(j)} x_{ij} &= 1, & j = 1, \dots, n; \\ \sum_{j \in \mathcal{J}_t(i)} p_{ij} x_{ij} &\leq d_i, & i = 1, \dots, m; \\ x_{ij} &\geq 0, & i = 1, \dots, m, j = 1, \dots, n.\end{aligned}$$

- ◇ The polyhedron described by  $LP(P, d, t)$  is defined by  $v + m + n$  constraints and is contained in the unit hypercube.
- ◇ At most  $n + m$  variables are positive in a basic feasible solution.
- ◇ For any feasible solution  $x$ , define a bipartite graph  $G(x) = (V, E)$  with  $V = M \cup J$ , where  $M = \{1, \dots, m\}$  and  $J = \{1, \dots, n\}$  correspond to the sets of machines and jobs, respectively, and  $E = \{(i, j) : x_{ij} > 0\}$ .



# A rounding theorem

- ◇ Let  $\tilde{x}$  be a vertex of the polytope.
- ◇ The number of edges in  $G(\tilde{x})$  is at most the number of nodes  $(n + m)$ .
- ◇ We will show that, in fact, each connected component of  $G(\tilde{x})$  has this property; i.e.,  $G(\tilde{x})$  is a **pseudoforest**.



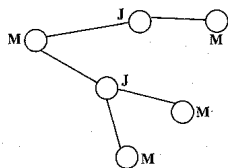
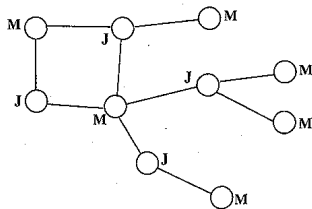
(figure source: D. Hochbaum (editor), *Approximation algorithms for NP-hard problems*)

## A rounding theorem

- ◇ Let  $C$  be a connected component of  $G(\tilde{x})$  with the vertex set given by  $M_C \cup J_C$ .
- ◇ Let  $\tilde{x}_C$  denote the restriction of  $\tilde{x}$  to components  $\tilde{x}_{ij}$  such that  $i \in M_C, j \in J_C$ , and let  $\tilde{x}_{\bar{C}}$  denote the remaining components of  $\tilde{x}$ .
- ◇ We can reorder the components so that  $\tilde{x} = (\tilde{x}_C, \tilde{x}_{\bar{C}})$ .
- ◇ Denote by  $P_C$  the restriction of  $P$  to the rows corresponding to machines in  $M_C$  and the columns corresponding to jobs in  $J_C$ .
- ◇ Denote by  $d_C$  the restriction of  $d$  to machines in  $M_C$ .
- ◇ Then  $\tilde{x}_C$  is an extreme point of  $LP(P_C, d_C, t)$ .
  - ▶ Suppose not; then there exist feasible  $y_1, y_2$ :  $\tilde{x}_C = \frac{1}{2}(y_1 + y_2)$ . But then  $\tilde{x} = \frac{1}{2}((y_1, \tilde{x}_{\bar{C}}) + (y_2, \tilde{x}_{\bar{C}}))$ , contradicting the choice of  $\tilde{x}$ .

# A rounding theorem

- ◇ Since  $\tilde{x}_C$  is an extreme point of  $LP(P_C, d_C, t)$ , it follows that  $G(\tilde{x}_C)$  has no more edges than nodes.
- ◇ Since  $C = G(\tilde{x}_C)$  is connected,  $C$  is either a tree or a tree plus one additional edge, so  $G(\tilde{x})$  is a pseudoforest.



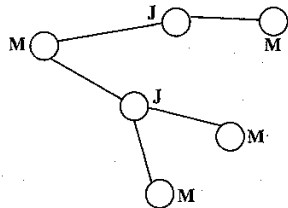
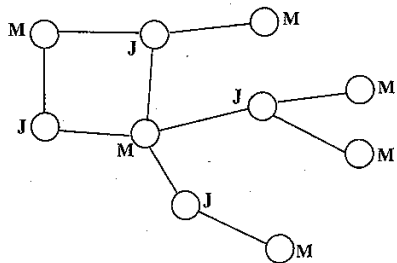
# Rounding the solution

- ◇ Set  $\bar{x}_{ij} = 1$  if  $\tilde{x}_{ij} = 1$ .
- ◇ These jobs correspond to the job nodes of degree 1, so their removal will result in a pseudoforest  $G'(\tilde{x})$  with each job node having degree at least 2.
- ◇ We show that  $G'(\tilde{x})$  has a matching that covers all of the job nodes.
- ◇ For each component that is a tree, let us root the tree at any node and match each job node with any one of its children.
  - ▶ Each job node must have at least one child.
  - ▶ Since each node has at most one parent, no machine is matched with more than one job.

# Rounding the solution

- ◇ For each component that contains a cycle, take alternate edges of the cycle in the matching.
  - ▶ There can be at most one cycle.
  - ▶ The cycle must be of even length since the graph is bipartite.
- ◇ If we delete the edges of the cycle, we get a collection of trees.
  - ▶ Root each tree in the node that had been contained in the cycle.
  - ▶ For each job node not already matched, pair it with one of its children.
- ◇ If  $(i, j)$  is in the matching, set  $\bar{x}_{ij} = 1$ , otherwise set  $\bar{x}_{ij} = 0$ .

## Rounding the solution



# Rounding the solution

Let us verify that  $\bar{x}$  is a feasible solution to  $IP(P, d, t)$ .

- ◇ Each job has been scheduled on exactly one machine, so

$$\sum_{i \in M_j(t)} \bar{x}_{ij} = 1, \quad j = 1, \dots, n.$$

- ◇ For each machine  $M_i$ , there is at most one job  $J_j$  such that  $\tilde{x}_{ij} < \bar{x}_{ij} = 1$ ; since  $p_{ij} \leq t$ , we have

$$\sum_{j \in J_i(t)} p_{ij} \bar{x}_{ij} \leq \sum_{j \in J_i(t)} p_{ij} \tilde{x}_{ij} + t \leq d_i + t, \quad i = 1, \dots, m.$$

This completes the proof of the Theorem. □

## $\rho$ -relaxed decision procedure

- ◇ Consider the following decision version of our problem: given a matrix  $P$  of processing times and a deadline  $d$ , does there exist a schedule with makespan at most  $d$ ?
- ◇ An ordinary decision procedure would output “yes” or “no”.
- ◇ A  $\rho$ -relaxed decision procedure outputs “no” or “almost”, namely:
  1. it either outputs “no” or produces a schedule with makespan at most  $\rho d$ , and
  2. if the output is “no”, then there is no schedule with makespan at most  $d$ .



# $\rho$ -relaxed decision procedure

## Lemma

*If there is a polynomial-time  $\rho$ -relaxed decision procedure for the minimum makespan problem on unrelated parallel machines, then there is a polynomial-time  $\rho$ -approximation algorithm for this problem.*

## Proof:

- ◇ Construct a greedy schedule by assigning each job to the machine on which it runs fastest.
- ◇ If the corresponding makespan is  $t$ , then

$$t/m \leq c_{max}^* \leq t$$

- ◇ Using these initial bounds we run a binary search procedure.

# Binary search procedure

- ◇ If  $u$  and  $l$  are the current upper and lower bounds, set

$$d = \left\lfloor \frac{1}{2}(u + l) \right\rfloor$$

and apply the  $\rho$ -relaxed decision procedure to  $(P, d)$ .

- ◇ If the answer is “almost”, then reset  $u$  to  $d$ ;
  - ◇ Otherwise, reset  $l$  to  $d + 1$ .
  - ◇ Store the best solution found so far.
- ◇ When the upper and lower bounds are equal, output the best solution found.
    - ◇  $l$  is always a lower bound on the optimum makespan.
    - ◇ The best solution found has makespan at most  $\rho u$ .
    - ◇  $u = l$  at termination.



# Limits to approximation

## Theorem

*For every  $\rho < \frac{4}{3}$ , there does not exist a polynomial-time  $\rho$ -approximation algorithm for the minimum makespan problem on unrelated parallel machines, unless  $P = NP$ .*



J. K. Lenstra, D. B. Shmoys and E. Tardos. Approximation algorithms for scheduling unrelated parallel machines. *Mathematical Programming*, 46: 259-271, 1990.

# Dominating sets

- ▶ Given a simple undirected graph  $G = (V, E)$ , a subset  $D \subseteq V$  is a **dominating set** if each vertex in  $V$  is either in  $D$  or has at least one neighbor in  $D$ .
- ▶ A **connected dominating set** is a dominating set that induces a connected subgraph.
- ▶ The minimum dominating set problem and the minimum connected dominating set problem are NP-hard.

## $k$ -center

- ▶ Consider a simple undirected graph  $G = (V, E)$ .
- ▶ Let  $d_{ij}$  denote the shortest path distance between  $i$  and  $j$ .
- ▶ Define a complete graph with edge weights given by  $d_{ij}$ .
- ▶ The  $k$ -center problem is to find a subset of vertices  $S$  with  $|S| = k$ , so that the longest distance of a vertex from the nearest vertex in  $S$  is minimized.
- ▶ I.e., the objective is to identify  $S \subseteq V, |S| = k$  that minimizes

$$\text{cost}(S) = \max_{i \in V} \min_{j \in S} d_{ij}.$$

# $k$ -center

## Theorem

*The problem of approximating the  $k$ -center problem with triangle inequality satisfied within a factor of  $(2 - \epsilon)$  is NP-complete for any  $\epsilon > 0$ .*

## Proof:

- ▶ We use a reduction from the dominating set problem.
- ▶ Given an instance  $G = (V, E)$  of the dominating set problem, we construct a complete graph with the following edge weights:

$$w(e) = \begin{cases} 1, & \text{if } e \in E \\ 2, & \text{if } e \notin E \end{cases}$$

- ▶ The existence of a  $k$ -center of cost  $< 2$  in the complete graph is equivalent to existence of a dominating set of size  $\leq k$  in  $G$ .
- ▶ Moreover, if there is a  $k$ -center of cost  $< 2$  then the optimal cost is equal to 1, thus a  $(2 - \epsilon)$ -approximation algorithm would be exact.



## A 2-approximate algorithm for $k$ -center

**Observation:** The optimal objective value always equals to the weight of one of the edges for the  $k$ -center problem.

- ◇ Sort all edges in nondecreasing order of their weights:

$$c_{e_1} \leq c_{e_2} \leq \cdots \leq c_{e_m}, \quad m = \binom{n}{2}$$

- ◇ Define the **bottleneck graph** as follows:

$$G_{e_i} = (V, E_i), \text{ where } E_i = \{e_j : c_{e_j} \leq c_{e_i}\}.$$

- ◇  $G_{e_i}$  has a dominating set of size  $\leq k$  iff  $G$  has a  $k$ -center  $S$  of cost  $\text{cost}(S) = \max_{i \in V} \min_{j \in S} c_{ij} \leq c_{e_i}$ .

# $t$ -th power of a graph

## Definition

Let  $t$  be a positive integer. Given a simple undirected graph  $G = (V, E)$  (not complete), its  $t$ -th power  $G^t$  is given by  $G^t = (V, E^t)$ , where  $E^t$  connects pairs of vertices, such that there is a path consisting of at most  $t$  edges between them:

$$E^t = \{(i, j) : d_G(i, j) \leq t\}.$$

- ◇ In  $G_{e_i}^t$ , the weights of its edges are given by the weights of these edges in the original complete graph  $G$ .
- ◇ Assume that the weights of edges of the original complete graph satisfy the triangle inequalities.
- ◇ Let  $c(G_{e_i})$  denote the largest weight of an edge in  $G$ .
- ◇ Then  $c(G_{e_i}^2) \leq 2c(G_{e_i})$  (due to the triangle inequality).



# Independent sets in $G_{e_i}^2$

- ◇ Consider a **maximal** independent set  $I_2(e_i)$  in  $G_{e_i}^2$ .
- ◇ Note that any two vertices  $i, j \in I_2(e_i)$  cannot be dominated by the same vertex in  $G_{e_i}$ 
  - ▶ if they were, they would be connected in  $G_{e_i}^2$
- ◇ Thus, we need at least  $|I_2(e_i)|$  vertices to dominate all vertices in  $G$ .
- ◇ Thus, for any dominating set  $D$  in  $G_{e_i}$ , we have

$$|D| \geq |I_2(e_i)|.$$

# The 2-approximation algorithm

INPUT:  $G = (V, E)$  with edges sorted in nondecreasing order of weights,  $c_{e_1} \leq c_{e_2} \leq \dots \leq c_{e_m}$ .

0. Set  $i \leftarrow 1$ .
1. Let  $G_{e_i} = \text{Bottleneck}[G(e_i)]$   
Compute a maximal independent set  $I_2(e_i)$  in  $G_{e_i}^2$ .
2. If  $|I_2(e_i)| > k$  then  $G_{e_i}$  cannot have a dominating set of size  $\leq k$ . Thus, the cost of optimal  $k$ -center  $> c_{e_i}$ .  
 $i \leftarrow i + 1$ ; go to step 1.
3. If  $|I_2(e_i)| \leq k$  then, since  $I_2(e_i)$  is a maximal independent set in  $G_{e_i}^2$ , it is a  $k$ -center in  $G_{e_i}^2$  of cost  $\leq c(G_{e_i}^2) \leq 2c(G_{e_i})$ .
4. Output  $I_2(e_i)$ .

# The 2-approximation algorithm

## Theorem

$$\text{cost}(I_2(e_i)) \leq 2\text{cost}(S^*),$$

where  $S^*$  is an optimal  $k$ -center.

## Proof:

- ◇ Assume that our algorithm terminates after  $i$  iterations with output  $I_2(e_i)$ .
- ◇ Since iteration  $i - 1$  resulted in an independent set of size  $> k$  in  $G_{e_{i-1}}^2$ , any dominating set in  $G_{e_{i-1}}$  is of size  $> k$ .
- ◇ Thus,

$$\text{cost}(S^*) > c_{e_{i-1}} \implies \text{cost}(S^*) \geq c_{e_i}.$$

- ◇ Finally, we have

$$\text{cost}(I_2(e_i)) \leq 2c_{e_i} \leq 2\text{cost}(S^*).$$

# Bottleneck problems

Given a complete graph with nonnegative edge weights, find a subgraph of  $G$ , which satisfies certain structural properties, and such that the weight of a heaviest edge in the subgraph is minimized.

## Example

1. In the  *$k$ -center problem*, the subgraph sought for has to consist of  $k$  stars that span all vertices.
2. In the *min-max  $k$ -clustering* problem, the subgraph has to be a disjoint union of  $k$  cliques that covers all vertices.

The procedure considered for  $k$ -center can be generalized to a class of bottleneck problems.

# Bottleneck problems

- ◇ We assume that the edge weights satisfy the triangle inequality.
- ◇ We sort all edges in the nondecreasing order of weights:

$$c_{e_1} \leq c_{e_2} \leq \cdots \leq c_{e_m}, m = \binom{n}{2}.$$

- ◇ For a considered bottleneck problem, let  $\mathcal{H}$  denote the set of all *feasible* subgraphs of  $G$ .
- ◇ We need to minimize  $\max(H)$  for  $H \in \mathcal{H}$ , where

$$\max(H) = \max_{e \in H} \{c_e\}.$$

# Bottleneck problems

---

PROCEDURE BOTTLENECK( $G, \mathcal{H}, t$ )

---

$i \leftarrow 0$ ;

until test\_result is not “failure” do

begin

$i \leftarrow i + 1$ ;

$G_i \leftarrow G_{e_i}$ ;

test\_result  $\leftarrow$  TEST( $G_i, \mathcal{H}, t$ );

end

output test\_result

---

TEST( $G_i, \mathcal{H}, t$ ) has the following possible outputs:

1. “failure”: proof that there is no subgraph of  $G_i$  that belongs to  $\mathcal{H}$ .
2. a graph  $H \in \mathcal{H}$  that is a subgraph of  $G_i^t$ .

# Bottleneck problems

## Theorem

Let  $\mathcal{H}$  be the set of all feasible subgraphs of  $G$ . Let  $H^* \in \mathcal{H}$  be an optimal subgraph in  $\mathcal{H}$ . Let  $H_A$  be the graph output by the procedure  $BOTTLENECK(G, \mathcal{H}, t)$ . Then

$$\max(H_A) \leq t \max(H^*).$$

## Proof:

- ◇ Assume that the procedure terminates at iteration  $i$ .
- ◇ This means that at previous iterations  $\text{TEST}(G_i, \mathcal{H}, t)$  returned “failure”.
- ◇ Thus,  $\max(H^*) \geq c_{e_i}$ .
- ◇ Since  $H_A$  is a subgraph of  $G_i^t$ , we have

$$\max(H_A) \leq \max(G_i^t) \leq t \max(G_i) = t c_{e_i} \leq t \max(H^*).$$

# A network design problem

- ◇ Let  $\mathcal{F}$  be a family of graphs  $\{F_i\}_{i=1,2,\dots}$ , where  $F_i$  has  $i$  vertices.
- ◇ This family represents the allowed network structure.
- ◇ Let  $part(\mathcal{F}, H)$  be the minimum number of parts, into which the network  $H$  can be partitioned, so that each part of  $i$  vertices has  $F_i$  as a subgraph.
- ◇ Given a complete edge-weighted graph  $G$ , find the minimum value of  $c$  such that for  $H = bottleneck[G(c)]$  we have  $part(\mathcal{F}, H) \leq k$ .
- ◇ This problem is called the  $(k, \mathcal{F})$ -partition problem.



# A network design problem

## Example

1.  $\mathcal{F} = \{K_{i,1}\}_{i=1,2,\dots}$ . Then  $part(\mathcal{F}, H)$  is the domination number of  $H$ . The corresponding  $(k, \mathcal{F})$ -partition problem is the  $k$ -center problem.
2.  $\mathcal{F} = \{K_i\}_{i=1,2,\dots}$ . Then  $part(\mathcal{F}, H)$  is the clique partition number of  $H$ . The corresponding  $(k, \mathcal{F})$ -partition problem is min-max  $k$ -clustering problem.

Additional assumption:  $diam(F_i) \leq d$  for any  $i$ .

## A network design problem

---

### PROCEDURE TEST( $H, \mathcal{F}, 2d$ )

---

Find a maximal independent set  $S$  in  $H^d$ .

if  $|S| > k$

    return “failure” ( $H$  cannot be partitioned into  $k$  parts of diameter  $\leq d$ )

else

    if  $|S| < k$ , add  $k - |S|$  vertices to  $S$  arbitrarily:  $S = \{v_1, \dots, v_k$   
    partition  $V$  into subsets  $V_1, \dots, V_k$ , where  
     $v_i \in V_i, i = 1, \dots, k$ , and all vertices in  $V_i$  are neighbors of  $v_i$   
    output the partition  $V_1, \dots, V_k$ .

---

- ◇  $V_1, \dots, V_k$  are cliques in  $H^{2d}$ .
- ◇ Any graph is a subgraph of the clique on the same vertices, so no matter what  $F_i$ 's are,  $H^{2d}$  is a feasible solution.
- ◇ Thus, we have a  $2d$ -approximation.

# Construction heuristics for TSP

- ◇ The **triangle inequality** (**metric**) **TSP** (abbreviated  $\Delta$ TSP) if the TSP is restricted to cost matrices satisfying the triangle inequality.
- ◇ If the TSP is restricted to Euclidean distance matrices, we have the **Euclidean TSP**.
- ◇ The recognition version of  $\Delta$ TSP is NP-complete.
- ◇ We will discuss the following construction algorithms for TSP:
  1. Nearest neighbor
  2. Greedy (multifragment) heuristic
  3. Tree algorithm
  4. Christofides algorithm

# Eulerian graphs

- ◇ A **Eulerian walk** in a graph is a closed walk, in which each node appears at least once and each edge appears exactly once.
- ◇ A multigraph (a graph with repetitions of edges allowed) is called Eulerian if it has a Eulerian walk.

## Theorem

*A multigraph  $G = (V, E)$  is Eulerian if and only if*

- 1.  $G$  is connected and*
- 2. All nodes in  $V$  have even degree.*

- ◇ A Eulerian walk can be computed in  $O(|E|)$  time.

# Eulerian graphs

- ◇ Let  $[d_{ij}]_{i,j=1}^n$  be a distance matrix satisfying the triangle inequality.
- ◇ An **Eulerian spanning graph** is an Eulerian multigraph  $G = (V, E)$  with  $V = \{1, 2, \dots, n\}$ .
- ◇ The cost of  $G$  is  $c(G) = \sum_{(i,j) \in E} d_{ij}$ .

## Theorem

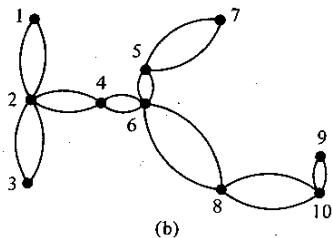
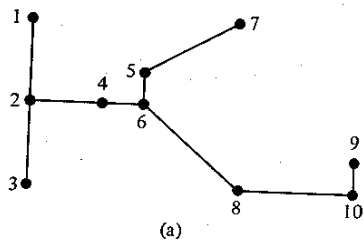
*If  $G = (V, E)$  is an Eulerian spanning graph then we can find a tour  $\tau$  of  $V$  with  $c(\tau) \leq c(G)$  in  $O(|E|)$  time.*

- ◇ Consider an arbitrary Eulerian walk  $w$ :  
 $w = [\alpha_0 v_1 \alpha_2 v_2 \cdots \alpha_n v_n]$ , where  $\tau = (v_1, \dots, v_n)$  is a tour and  $\alpha_0, \dots, \alpha_n$  are sequences (possibly empty) of vertices from  $\{1, \dots, n\}$ .
- ◇ The triangle inequality implies that  
 $d_{ik} \leq d_{j_1 j_2} + d_{j_2 j_3} + \cdots + d_{j_{m-1} j_m} + d_{j_m k}$
- ◇ Thus,  $c(\tau) = d_{v_1 v_2} + d_{v_2 v_3} + \cdots + d_{v_n v_1} \leq c(w) = c(G)$ .

# The tree algorithm

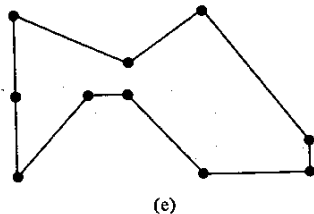
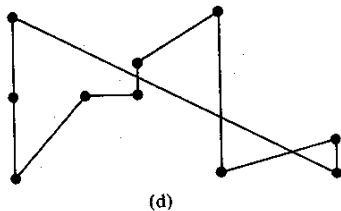
1. Find a minimum spanning tree  $T$  for  $[d_{ij}]$ ;
2. Create a multigraph  $G$  by using two copies of each edge of  $T$ ;
3. Find an Eulerian walk of  $G$  and an embedded tour  $\tau$ .

# The tree algorithm: example



[1, 2, 3, 2, 4, 6, 5, 7, 5, 6, 8, 10, 9, 10, 8, 6, 4, 2, 11]

(c)



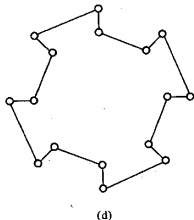
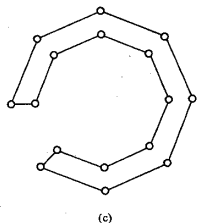
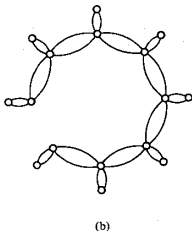
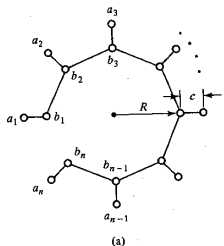
# The tree algorithm

Next, we show that this is a 2-approximation algorithm.

- ◇ Since all degrees in  $G$  are even,  $G$  is Eulerian.
- ◇ From the previous theorem, we have  $c(\tau) \leq c(G)$ .
- ◇  $c(G) = 2c(T)$ , where  $c(T)$  is the cost of the minimum spanning tree.
- ◇  $c(T) \leq c^*$ , where  $c^*$  is the cost of a shortest tour.
- ◇ Thus,  $c(\tau) \leq 2c^*$ .



# The tree algorithm: worst-case example



(a) minimum spanning tree

(b) multigraph  $G$

(c) the tour  $\tau$

(d) optimal tour  $\tau^*$

We have:

$$c(\tau) = 2(n-1)(2R+c) \sin \frac{\pi}{n} + 2c$$

$$c(\tau^*) = n(2R+c) \sin \frac{\pi}{n} + nc$$

With  $R = 1, c = 1/n^2$ , we have

$$\lim_{n \rightarrow \infty} c(\tau) = 4\pi$$

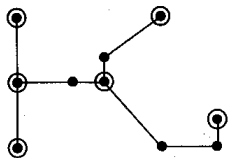
$$\lim_{n \rightarrow \infty} c(\tau^*) = 2\pi$$

# Christofides' algorithm

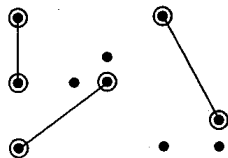
1. Find a minimum spanning tree  $T$  for  $[d_{ij}]$ ;
2. Let  $U$  be the subset of odd-degree vertices in  $T$ . Find a minimum weight perfect matching  $M$  in  $G[U]$ ;
  - ▶ perfect matching matches all vertices in pairs
  - ▶  $U$  has an even number of vertices, so a perfect matching exists
3. Create a multigraph  $G'$  by combining edges from  $T$  and  $M$ ;
4. Find an Eulerian walk of  $G'$  and an embedded tour  $\tau$ .

We will show that this is a  $\frac{3}{2}$ -approximation algorithm.

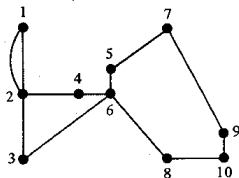
# Christofides' algorithm: example



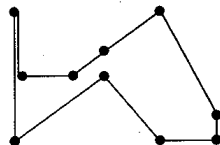
(a)



(b)



(c)



(d)

[1, 2, 3, 6, 8, 10, 9, 7, 5, 6, 4, 2, 1]

(e)

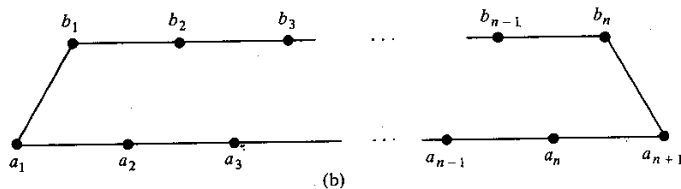
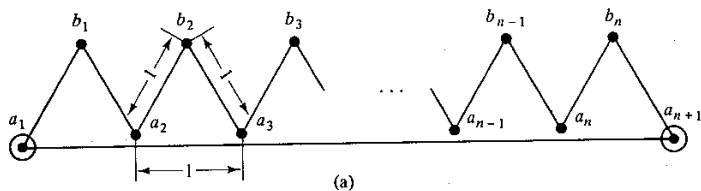
## Christofides' algorithm

- ◇ Since all degrees in  $G'$  are even,  $G'$  is Eulerian.
- ◇ Again, we have  $c(\tau) \leq c(G')$ .
- ◇  $c(G') = c(T) + c(M)$ , where  $c(T)$  is the cost of the minimum spanning tree and  $c(M)$  is the cost of the minimum weight perfect matching for  $G[U]$ .
- ◇  $c(T) \leq c^*$ , where  $c^*$  is the cost of a shortest tour in  $G$ .
- ◇ Let  $\tau'$  be an optimal tour for  $U$ . Then due to the triangle inequality  $c(\tau') \leq c^*$ .
- ◇ Since  $|U|$  is even, the edges of  $\tau'$  can be separated into two perfect matchings  $M_1$  and  $M_2$  in  $G[U]$ .
- ◇ We have  $c(M) \leq c(M_1)$  and  $c(M) \leq c(M_2)$ , so

$$c(M) \leq \frac{c(M_1) + c(M_2)}{2} = c(\tau')/2 \leq c^*/2.$$

- ◇ Thus,  $c(\tau) \leq c(G') = c(T) + c(M) \leq c^* + c^*/2 = \frac{3}{2}c^*$ .

# Christofides' algorithm: worst-case example



$$c(\tau) = 3n, \quad c(\tau^*) = 2n + 1$$

# Vehicle routing problem (VRP)

- ◇  $m$  vehicles
- ◇  $n$  cities, distance matrix (may not be symmetric)
- ◇  $v_0$  is the depot
- ◇ all vehicles are at the depot in the beginning
- ◇ construct  $m$  tours of minimum total length, so that each tour starts and ends at  $v_0$  and each city is visited by exactly one vehicle

# Vehicle routing problem (VRP)

Additional constraints:

1. **Capacitated VRP**: a weight  $q_i$  is associated with city  $i$  and  $\sum_{i \in T} q_i \leq W$  for any tour  $T$
2. **distance/time constrained VRP**: the total distance traveled by any vehicle is  $\leq L$  (in the time constrained version there is also time  $t_i$  associated with city  $i$ , which is added to the length of the tour passing through  $i$ )
3. **VRP with time windows**: city  $i$  has to be visited during a time within  $[a_i, b_i]$

# Clarke-Wright savings algorithm for VRP

- ◇ When two routes  $\tau_i = (v_0, \dots, i, v_0)$  and  $\tau_j = (v_0, j, \dots, v_0)$  can be merged into a single route  $(v_0, \dots, i, j, \dots, v_0)$ , a distance saving

$$s_{ij} = c_{i0} + c_{0j} - c_{ij}$$

is generated.

- ◇ Starting with  $n$  tours  $(v_0, i, v_0)$  for each city  $i$ , we combine two tours corresponding to largest savings
- ◇ Continue recursively until  $m$  tours are obtained



# Local search

- ◇ Two major steps of heuristic algorithms are
  - ▶ Construction
  - ▶ Local search
- ◇ The choice of **neighborhood** is critical for local search performance.
- ◇ Let for a given problem  $P$   $S$  denotes the set of all feasible solutions.
- ◇ By  $2^S$  we denote the set of all subsets of  $S$ .
- ◇ A neighborhood is defined on  $2^S$ .

# Neighborhood

## Definition

For an instance  $(S, f)$  of a combinatorial optimization problem, a **neighborhood function** is a mapping  $N : S \rightarrow 2^S$ .

- ◇ For each feasible solution  $s \in S$ ,  $N(s) \subseteq S$  is a set of solutions that are in some sense close to  $s$ .
- ◇  $N(s)$  is the neighborhood of solution  $s$ ;
- ◇ each  $i \in N(s)$  is a neighbor of  $s$ .
- ◇ A neighborhood for a problem  $\mathcal{P}$  is defined by a rule that can be used to find the neighborhood of any feasible solution for any instance of  $\mathcal{P}$ .

# Local and global optima

## Definition

For an instance  $(S, f)$  of a combinatorial optimization problem and a neighborhood  $N$ , a solution  $\hat{s}$  is locally minimal with respect to  $N$  if

$$f(\hat{s}) \leq f(i) \text{ for all } i \in N(\hat{s}).$$

- ◇ Let  $\hat{S}$  denote the set of all local optimal solutions of  $(S, f)$ .

## Definition

A neighborhood  $N$  defined for a problem  $\mathcal{P}$  is called exact for  $\mathcal{P}$  if for any instance  $(S, f)$  of  $\mathcal{P}$  we have  $\hat{S} \subseteq S^*$ , i.e., any local optimal solution is global optimal.

# Local search

- ◇ Given a feasible solution, we iteratively search for a better neighbor, until we achieve a local optimum.
- ◇ Two popular strategies:
  - ▶ **first improvement**: move to the first better solution found in the neighborhood;
  - ▶ **best improvement**: move to the best solution in the neighborhood.

# Local search

## Example (Sorting)

- ◇  $a_1, \dots, a_n$  need to be sorted in nondecreasing order.
- ◇ By a feasible solution we will mean a permutation  $\pi$  on  $n$  numbers, i.e., a bijection

$$\pi : \{1, \dots, n\} \rightarrow \{1, \dots, n\}.$$

- ◇ Here  $\pi(i)$  is the original order of the number on the  $i$ -th position in the list corresponding to  $\pi$ .
- ◇ For example, if we are given a list of the following numbers, 1, 0, 7, 8, 5, then the order corresponding to

$$\pi = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 \\ 2 & 3 & 4 & 1 & 5 \end{pmatrix} = (2, 3, 4, 1, 5)$$

is given by 0, 7, 8, 1, 5.

# Local search

## Example (Continued)

- ◇ For a feasible solution (permutation)  $\pi$ , we define the objective function as

$$f(\pi) = \sum_{i=1}^n ia_{\pi(i)}.$$

- ◇ For example, for the above case (list 0, 7, 8, 1, 5), we have

$$f(\pi) = 0 + 2 \cdot 7 + 3 \cdot 8 + 4 \cdot 1 + 5 \cdot 5 = 67.$$

- ◇ If we have  $ia_{\pi(i)} + ja_{\pi(j)}$  with  $i < j$  and  $a_{\pi(i)} > a_{\pi(j)}$  as a part of the objective sum, then swapping  $a_{\pi(i)}$  and  $a_{\pi(j)}$  will increase the objective value.
- ◇ Thus,  $\max_{\pi} f(\pi)$  corresponds to the sorted list.
- ◇ We can define a **neighborhood**  $N(\pi)$  as follows: given  $\pi$ , exchange two numbers in the corresponding list to obtain a neighbor of  $\pi$ .

# Local search

## Example (Continued)

- ◇ For example, if  $\pi = (3, 1, 2)$  then

$$N(\pi) = \{\pi_1 = (1, 3, 2), \pi_2 = (2, 1, 3), \pi_3 = (3, 2, 1)\}.$$

- ◇ For a list of  $n$  numbers, we have  $\binom{n}{2} = \frac{n(n-1)}{2}$  neighbors for any feasible solution (permutation).
- ◇ Applying local search for this example, we obtain:

$$\begin{aligned} f(\pi) &= 1 \cdot 3 + 2 \cdot 1 + 3 \cdot 2 = 11 \\ f(\pi_1) &= 1 \cdot 1 + 2 \cdot 3 + 3 \cdot 2 = 13 \\ f(\pi_2) &= 1 \cdot 2 + 2 \cdot 1 + 3 \cdot 3 = 13 \\ f(\pi_3) &= 1 \cdot 3 + 2 \cdot 2 + 3 \cdot 1 = 10 \end{aligned}$$

so, a possible move is  $\pi \rightarrow \pi_1$ .

$$N(\pi) = \{\pi_4 = (3, 1, 2), \pi_5 = (2, 3, 1), \pi_6 = (1, 2, 3)\}.$$

$\pi_6$  is the optimal solution.

# Local search

- ◇ The above defined neighborhood is exact, even if it is restricted to pairs of consecutive elements in the list (“bubble sort”).



## $k$ -exchange neighborhoods

- ◇ For many problems, their feasible solution can be represented as a sequence or a partition.
- ◇ A  $k$ -exchange neighbor is obtained from the current feasible solution by taking  $k$  pairs of elements in the corresponding sequence (partition) and exchanging the two elements in each pair.

## $k$ -exchange neighborhoods

### Example (Max-bisection)

Given a graph  $G = (V, E)$  with nonnegative edge weights  $w_{ij}$ ,  $(i, j) \in E$  and  $|V| = 2n$ , partition  $V$  into two disjoint subsets  $V_1$  and  $V_2$  such that  $|V_1| = |V_2| = n$  and the total sum of weights of edges with endpoints in different parts is maximized.

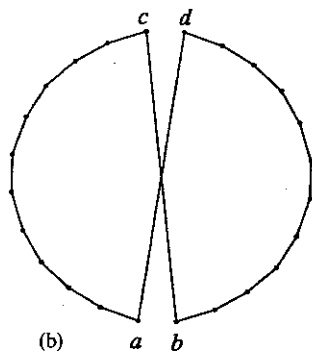
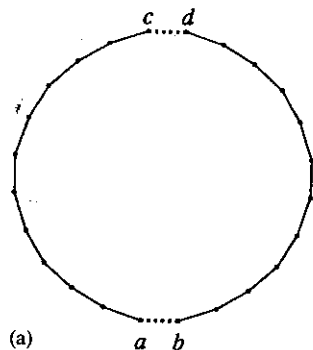
- ◇ objective:  $f(V_1, V_2) = \sum_{(i,j) \in (V_1, V_2)} w_{ij} \rightarrow \max;$
- ◇ neighborhood: exchange  $k$  vertices from  $V_1$  with  $k$  vertices from  $V_2$ .

# $k$ -exchange neighborhoods

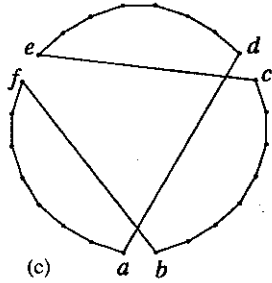
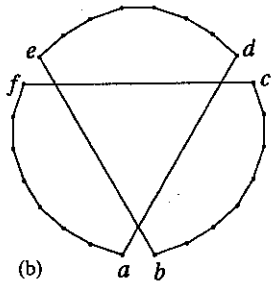
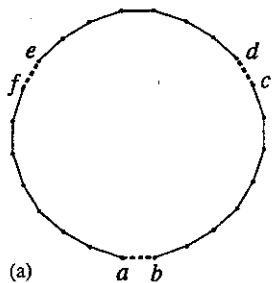
## Example (TSP)

- ◇ A feasible solution, which is a tour, can be viewed as a sequence of edges used in the tour.
- ◇  $k$ -exchange: remove  $k$  edges from the tour, add  $k$  new edges.
- ◇ The  $k$ -exchange neighborhood for TSP is usually referred to as  $k$ -OPT neighborhood
  - ▶ 2-OPT
  - ▶ 3-OPT
- ◇ There are  $O(n^k)$  ways of choosing  $k$  elements to exchange.
- ◇ There may be exponential (with respect to  $k$ ) number of ways to complete the exchange (the subtours can be ordered differently to form a feasible tour).

## 2-OPT neighborhood for TSP



## 3-OPT neighborhood for TSP

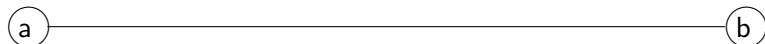


# The Lin-Kernighan algorithm

- ▶ Lin & Kernighan (1973) proposed an algorithm generalizing 3-Opt.
- ▶ In practice, the resulting algorithm yields significantly better results than 3-Opt with only a modest increase in running time.

# The basic idea of LKA

- ▶ Given a tour, we can remove an edge  $(a, b)$  to obtain a Hamilton path with endpoints  $a$  and  $b$ . Let  $a$  be fixed and  $b$  - variable.



- ▶ If we add an edge  $(b, c)$ , a cycle is formed; removal of  $(c, d)$  produces a new Hamilton path with a new variable endpoint  $d$ .



- ▶ This operation is called a *rotation*.

# The basic idea of LKA

- ▶ A move of the LKA from one tour to a neighbor consists of
  - ▶ removing an edge to form a Hamilton path;
  - ▶ performing a sequence of greedy rotations [The Lin-Kernighan inner loop];
  - ▶ reconnecting the two endpoints to form a tour.



# The Lin-Kernighan inner loop



- ▶ The new neighbor  $c$  of  $b$  must be such that the total length of edges in the graph obtained from the current path by adding the edge  $(b, c)$  is less than the length of the best tour found so far.
- ▶ Two lists are maintained:
  - ▶ added edges (like  $(b, c)$ );
  - ▶ deleted edges (like  $(c, d)$ ).

A move is not allowed if it adds an edge on the *deleted* list or deletes an edge on the *added* list.

# The Lin-Kernighan inner loop

- ▶ Once a particular edge has been added to (deleted from) the path, it can no longer be deleted (added).
- ▶ This implies that at most  $n$  moves can be made before the search terminates (it stops when there are no qualifying nontabu moves for the current path).
- ▶ For each qualifying, nontabu move, if the resulting tour is better than the best found so far, it is saved as the best current solution.
- ▶ The qualifying, nontabu move that yields the shortest new path is chosen as the current solution.
- ▶ This path may still be longer than the previous one, and its corresponding tour may be longer than the corresponding tour for the previous path.

# Variable-depth search

- ◇ Introduced by Kernighan and Lin

## Example (min-bisection (uniform graph partitioning))

Given a graph  $G = (V, E)$  with nonnegative edge weights  $w_{ij}$ ,  $(i, j) \in E$  and  $|V| = 2n$ , partition  $V$  into two disjoint subsets  $V_1$  and  $V_2$  such that  $|V_1| = |V_2| = n$  and the total sum of weights of edges with endpoints in different parts is minimized.

- ◇ objective:  $f(V_1, V_2) = \sum_{(i,j) \in (V_1, V_2)} w_{ij} \rightarrow \min;$
- ◇ Given a feasible solution with  $a \in V_1, b \in V_2$ , denote by  $g(a, b)$  the “gain function” given by

$$g(a, b) = \sum_{\substack{(a, v) \in E \\ v \in V_2 \setminus \{b\}}} w_{av} - \sum_{\substack{(a, u) \in E \\ u \in V_1}} w_{au} + \sum_{\substack{(b, u) \in E \\ u \in V_1 \setminus \{a\}}} w_{bu} - \sum_{\substack{(b, v) \in E \\ v \in V_2}} w_{bv}$$

## Variable-depth search

- Step 1. Choose two nodes  $a \in V_1$  and  $b \in V_2$  such that  $g(a, b)$  is maximal, even if that maximum is nonpositive.
- Step 2. Perform a *tentative* exchange of  $a$  and  $b$ .
- Step 3. Repeat Steps 1 and 2  $n$  times, where a node cannot be chosen to be exchanged if it has been exchanged in one of the previous iterations of Steps 1 and 2. The sequence of 2-exchanges obtained in this way defines a sequence  $g_1, g_2, \dots, g_n$  of gains  $g_i$ , where  $g_i$  corresponds to the exchange of the nodes  $a_i \in V_1$  and  $b_i \in V_2$  in the  $i$ th iteration. The total gain after  $k$  exchanges equals

$$G(k) = \sum_{i=1}^k g_i, \quad 0 \leq k \leq n.$$

Note that, for  $k=n$ ,  $V_1$  and  $V_2$  have been entirely interchanged, resulting in a total gain  $G(n) = 0$ .

- Step 4. Choose the value of  $k$  for which  $G(k)$  is maximal. If  $G(k) > 0$ , the neighboring solution is given by the partition that is obtained from  $(V_1, V_2)$  by effectuating the *definite* exchange of  $\{a_1, a_2, \dots, a_k\}$  and  $\{b_1, b_2, \dots, b_k\}$ . If  $G(k) \leq 0$  then  $(V_1, V_2)$  has no neighboring solutions.

# Complexity of Local Search

- ▶ Search Problems in Combinatorial Optimization
- ▶ Local Optima
- ▶ Motivation
- ▶ The Class PLS
- ▶ PLS Reducibility
- ▶ First PLS-Complete Problem
- ▶ Other PLS-Complete Problems

# Search Problems

- ▶ Each instance is associated with a finite set of feasible solutions
- ▶ Each feasible solution has a cost
- ▶ Objective is to find a solution of minimum (maximum) cost
- ▶ Define a **neighborhood** for each solution
- ▶ A solution is said to be locally optimal if there is no solution in its neighborhood with better cost

# Motivation

- ▶ Local search algorithms have been observed to be efficient in practice
- ▶ The common assumption is that local optima are easy to obtain
- ▶ How easy is it to find a local optimum?

# The Class PLS - Polynomial-time Local Search

A PLS problem  $\Pi$  can be a maximization or minimization problem:

- ▶  $\Pi$  has a set  $D_\Pi$  of instances
- ▶ For each instance  $x \in D_\Pi$  we have a finite set  $F_\Pi(x)$  of solutions, all with the same polynomially bounded length
- ▶ For each solution  $s \in F_\Pi(x)$ , we have a non-negative integer *cost*  $c_\Pi(s, x)$  and also a subset  $N(s, x) \subseteq F_\Pi(x)$  called the *neighborhood* of  $s$
- ▶ And the following algorithms must exist



# The Class PLS -Contd.

Polynomial-time algorithms  $A_\Pi$ ,  $B_\Pi$  &  $C_\Pi$  such that:

- ▶ Given an instance  $x \in D_\Pi$ ,  $A_\Pi$  produces a particular standard solution  $A_\Pi(x) \in F_\Pi(x)$ .
- ▶ Given an instance  $x$  and a string  $s$ ,  $B_\Pi$  checks if  $s \in F_\Pi(x)$  and if so, computes that cost  $c_\Pi(s, x)$ .
- ▶ Given an instance  $x$  and a solution  $s$ ,  $C_\Pi$  identifies a solution  $s' \in N(s, x)$  with better cost if it exists OR reports that  $s$  is locally optimal.

# Standard Local Search Algorithm

Inherent in the definition of a PLS problem, is the following algorithm:

1. Given  $x$ , use  $A_{\Pi}$  to produce a starting solution  $s = A_{\Pi}(x)$ .
2. Repeat until locally optimal:  
    Apply algorithm  $C_{\Pi}$  to  $x$  and  $s$ .  
    If  $C_{\Pi}$  yields a better cost neighbor  $s'$  of  $s$ , set  $s = s'$ .

# Standard Algorithm Problem

Given  $x$ , find the local optimum  $s$  that would be output by the *standard local search algorithm* for  $\Pi$  on input  $x$ .

## Lemma

*There is a PLS problem  $\Pi$  whose standard algorithm problem is NP-hard.*

Hence, general polynomial time algorithms for the standard algorithm problems, seems unlikely.

# What about *some* local optimum ?

## Lemma

*If a PLS problem is NP-Hard, then  $NP=co-NP$ .*

Finding *some* local optimum for a PLS problem  $\Pi$  is an “easier” task than finding the local optimum that is output by the standard algorithm for  $\Pi$ .

# PLS reducibility

A problem  $\Pi$  in PLS is PLS-reducible to another,  $\Pi'$ , if there are polynomial-time computable functions  $f$  and  $g$  such that

1.  $f$  maps instances  $x$  of  $\Pi$  to instances  $f(x)$  of  $\Pi'$ .
2.  $g$  maps (solution of  $f(x)$ ,  $x$ ) pairs to solutions of  $x$ .
3. For all instances  $x$  of  $\Pi$ , if  $s$  is a local optimum for instance  $f(x)$  of  $\Pi'$ , then  $g(s, x)$  is a local optimum for  $x$ .

$$\begin{array}{ccc} \Pi & \leq_{pls} & \Pi' \\ & f & \\ x & \longrightarrow & f(x) \\ & g & \\ (g(s), x) & \longleftarrow & (s, f(x)) \end{array}$$

# PLS reducibility

## Lemma

*If  $\Pi_1, \Pi_2, \Pi_3$  are problems in PLS such that  $\Pi_1 \leq_{pls} \Pi_2$  and  $\Pi_2 \leq_{pls} \Pi_3$ , then  $\Pi_1 \leq_{pls} \Pi_3$ .*

## Lemma

*If  $\Pi_1, \Pi_2$  are problems in PLS such that  $\Pi_1 \leq_{pls} \Pi_2$  and if there is polynomial-time algorithm for finding local optima for  $\Pi_2$ , then there is also a polynomial-time algorithm for finding local optima for  $\Pi_1$ .*

A problem  $\Pi$  in PLS is said to be **PLS-Complete** if every problem in PLS is PLS-reducible to  $\Pi$ .

# Problem Definition: FLIP

## Definition (FLIP)

- ▶ Given a circuit  $x$  with  $m$  inputs and  $n$  outputs, a solution in  $F_{FLIP}(x)$  is any bit vector  $s$  with  $m$  components.
- ▶ It has  $m$  bit vectors of length  $m$  with hamming distance one from  $s$  as neighbors.
- ▶ The cost of solution  $s$  is defined as  $\sum_{j=1}^n 2^j y_j$ , where  $y_j$  is the  $j^{th}$  output of the circuit  $x$  with input  $s$ .

We can have either minimization or maximization objective.

- ◇ Algorithm  $A_{FLIP}$  returns the all-1 vector
- ◇  $B_{FLIP}$  (cost-computation) is straight-forward from above
- ◇  $C_{FLIP}$  returns the best of the  $m$  neighbors of  $s$  (ties broken lexicographically) if  $s$  is not locally optimal.

# First PLS-Complete Problem

## Theorem

*FLIP is PLS-Complete.*

## Corollary

- (a) *The standard algorithm problem for FLIP is NP-hard.*
- (b) *There are instances of FLIP for which the standard algorithm requires exponential time.*



# Other PLS-Complete Problems

- ▶ Graph partitioning under Kernighan-Lin neighborhood.
- ▶ Graph partitioning under the swap neighborhood.
- ▶ Graph partitioning under Fiduccia-Mattheyses (FM) neighborhood (similar to LK, but the vertices are moved to the other part one at a time).
- ▶ MAXCUT under the flip neighborhood.
- ▶ TSP under the  $k$ -opt neighborhood (for some  $k$ ).
- ▶ TSP under the Lin-Kernighan neighborhood.

# Greedy Randomized Adaptive Search Procedures (GRASP)

# What is GRASP?

Multi-start procedure where each iteration consists of:

1. Construction of a greedy randomized feasible solution
2. Local search: starting from the constructed solution, finds a locally optimal solution

# Generic GRASP (Maximization)

## **procedure** GRASP

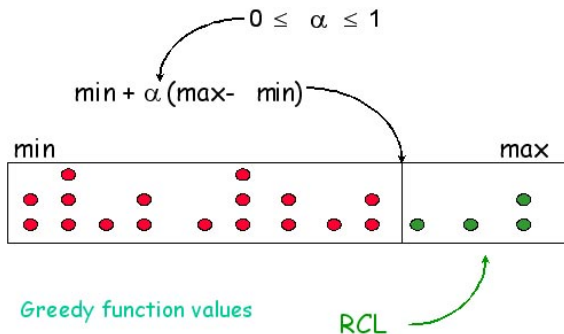
```
1  BestSolutionFound= $\emptyset$ ;  
2   $f(\text{BestSolutionFound}) = -\infty$ ;  
3  for  $k = 1, \dots, \text{MaxIter} \rightarrow$   
4       $x = \text{GreedyRandomizedConstruction}(\text{RandomSeed})$ ;  
5       $x' = \text{LocalSearch}(x)$ ;  
6      if  $f(x') > f(\text{BestSolutionFound}) \rightarrow$   
7           $\text{BestSolutionFound} = x'$ ;  
8      end if;  
9  end for;  
10 return BestSolutionFound  
end GRASP;
```

# GRASP Construction Phase

**procedure** GreedyRandomizedConstruction

```
1  SolutionFound= $\emptyset$ ;  
2  Rank all candidates using a greedy function;  
3  while solution is not complete  $\rightarrow$   
4      Build the Restricted Candidate List (RCL);  
5      Select element  $s$  from RCL at random;  
6      SolutionFound = SolutionFound  $\cup \{s\}$ ;  
7  end while;  
8  return SolutionFound  
end GreedyRandomizedConstruction;
```

## RCL based on minmax $\alpha$ percentage



$\alpha = 0$ : completely random

$\alpha = 1$ : completely greedy

# Local Search

Consider a maximization problem

$$\max_{x \in S} f(x).$$

Given an initial solution  $x_0 \in S$ , a neighborhood  $N(\cdot) : S \rightarrow 2^S$ , a generic local search works as follows:

$x = x_0$ ;

**while**  $\exists y \in N(x) : f(y) > f(x)$

$x = y$ ;

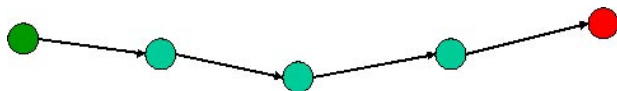
**end while**

**return**  $x$

The output  $x$  is a local maximum.

# Path Relinking

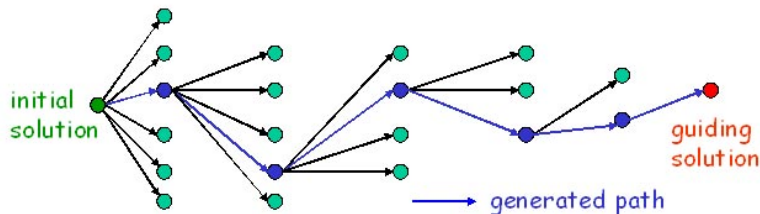
- ▶ Introduced by Glover (1996); Laguna & Marti (1999)
- ▶ Approach to integrate intensification and diversification in search.
- ▶ Consists in exploring trajectories that connect high quality solutions.





# Path Relinking

- ▶ Path is generated by selecting moves that introduce in the initial solution attributes of the guiding solution.
- ▶ At each step, all moves in the neighborhood of the current solution that incorporate attributes of the guiding solution are analyzed and best move is taken.



# GRASP with Path Relinking

- ▶ Maintain an elite set of solutions found using GRASP iterations.
- ▶ After each GRASP iteration (construction & local search):
  - ▶ Select an elite solution at random: guiding solution
  - ▶ Use the GRASP solution as initial solution
  - ▶ Do path relinking from initial solution to guiding solution

# Elite set membership rule

(Fleurent & Glover, 1999)

A feasible solution  $x$  is accepted into the elite set if:

- ▶  $x$  is better than the best elite set solution
- ▶  $x$  is better than the worst elite set solution and sufficiently different from all elite solutions

If  $x$  is accepted, it replaces the worst elite set member.

# Path relinking: Intensification & Post-optimization

Elite set intensification:

- ▶ Apply path relinking between all pairs of elite set solutions.
- ▶ Update elite set, if necessary, and repeat until no change occurs.

Post-optimization:

- ▶ Apply local search to each elite set solution.
- ▶ Repeat if necessary.

# MAX-CUT Problem

Given a complete undirected graph

$$G = (V, E), V = \{1, 2, \dots, n\}, E = V \times V$$

with weights  $w_{ij}, i, j = 1, \dots, n$  on the edges, find a vertex partition

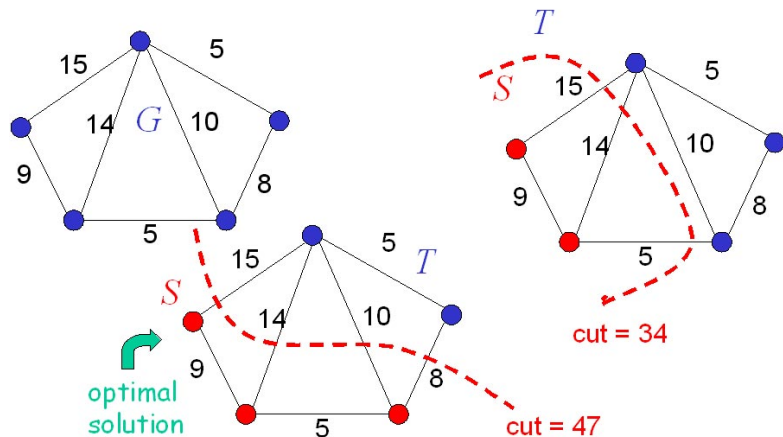
$$S, \bar{S} \subseteq V, S \cup \bar{S} = V, S \cap \bar{S} = \emptyset$$

such that the sum of the weights in the cut  $(S, \bar{S})$ ,

$$\sum_{i \in S, j \in \bar{S}} w_{ij}$$

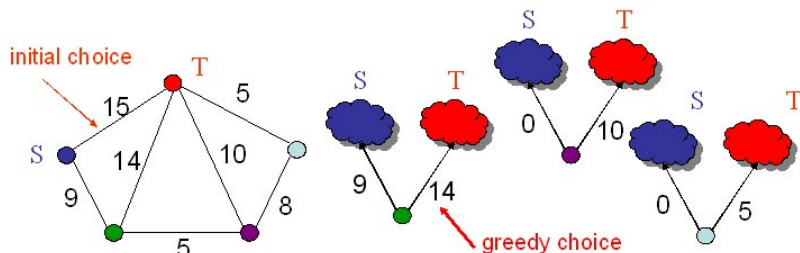
is maximized.

# MAX-CUT Problem: An Example



# Greedy Construction for MAX-CUT

- ▶ Initial partition is based on the edge of maximum weight.
- ▶ Vertices are added greedily, one at a time, based on the sum of weights of its edges incident to the constructed partition.



# Greedy Randomized Construction

Denote by

- ▶  $\sigma(v, S)$  ( $\sigma(v, T)$ ) is the sum of edge weights between vertex  $v$  and partition  $S$  ( $T$ );
- ▶  $\sigma^+ = \max\{\sigma(v, S), \sigma(v, T) : v \notin S \cup T\}$ ;
- ▶  $\sigma^- = \min\{\sigma(v, S), \sigma(v, T) : v \notin S \cup T\}$ .

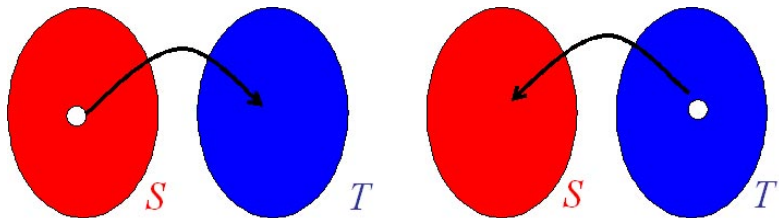
Then

$$\text{RCL} = \{v \notin S \cup T : \sigma(v, S) \text{ or } \sigma(v, T) \geq \sigma^- + \alpha(\sigma^+ - \sigma^-)\}$$

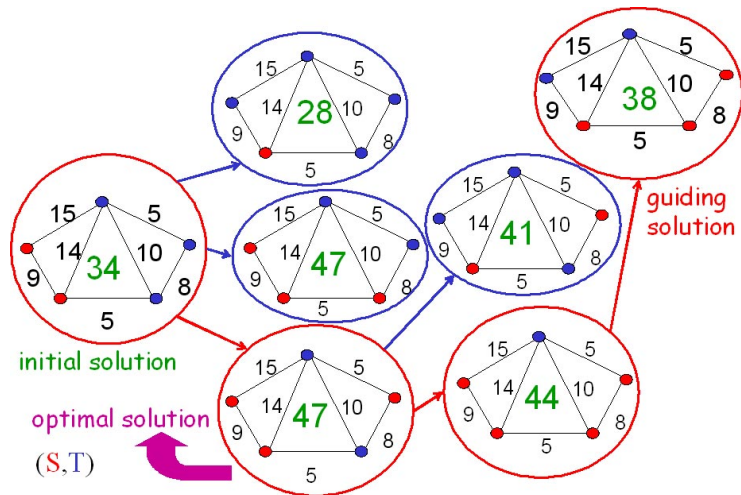


# Local Search for MAX-CUT

Neighborhood: 1-flip



# Path Relinking for MAX-CUT



# References



T. A. Feo and M. G. C. Resende.

Greedy randomized adaptive search procedures.

*Journal of Global Optimization*, 6:109–133, 1995.



P. Festa and M. G. C. Resende, An annotated bibliography of GRASP, *European Journal of Operational Research*, 2004



Mauricio Resende's homepage:

<http://www.research.att.com/~mgcr>

# Threshold algorithms

---

Generate a starting solution  $i$ ;

$k = 0$ ;

**repeat**

    take a solution  $j$  from  $N(i)$  (neighb. of  $i$ );

    if  $f(j) - f(i) < t_k$  then

$i = j$ ;

$k = k + 1$ ;

**until** a stopping criterion is satisfied

---

# Threshold algorithms

1. Iterative improvement:  $t_k = 0, \forall k$ .
2. Threshold accepting:

$$t_k \geq 0, t_k \geq t_{k+1}, \lim_{k \rightarrow \infty} t_k = 0.$$

3. Simulated annealing (SA):  $t_k$  is a nonnegative random variable with a positive expected value.

Each neighboring solution can be chosen with a positive probability.

The smaller the decrease in objective function (for minimization problem), the smaller is the probability of being accepted.

## SA: The original version

- Kirkpatrick et al. (1983):

$$P_{c_k}\{\text{accept } j\} = \begin{cases} 1 & \text{if } f(j) \leq f(i), \\ \exp\left(\frac{f(i)-f(j)}{c_k}\right) & \text{if } f(j) > f(i). \end{cases}$$

$c_k$  is a control parameter.

# The physics analogy

- ▶ In condensed matter physics, *annealing* is a thermal process for obtaining low-energy states of a solid in a heat bath.
- ▶ Consists of the following two steps:
  1. The temperature of the heat bath is increased to melt the solid;
  2. The temperature is gradually reduced to obtain a highly structured lattice corresponding to the minimum energy of the system.
- ▶ Computer simulations of the annealing process are based on Monte Carlo techniques.

# Metropolis algorithm (1953)

- ▶ Given a state  $i$  of the solid with energy  $E_i$ , the next state  $j$  (with energy  $E_j$ ) is generated by applying a permutation, such as a displacement of a single particle.
- ▶ If  $E_j \leq E_i$ ,  $j$  is accepted as the current state.
- ▶ If  $E_j > E_i$ ,  $j$  is accepted with probability

$$\exp\left(\frac{E_i - E_j}{k_B T}\right), \quad [\text{Metropolis criterion}]$$

where  $T$  is the temperature of the heat bath and  $k_B$  is the Boltzmann constant.



# Boltzmann distribution

- ▶ If the temperature is decreased slowly, at each temperature the solid can reach thermal equilibrium characterized by the Boltzmann distribution:

$$P_T\{X = i\} = \frac{\exp(-E_i/k_B T)}{\sum_j \exp(-E_j/k_B T)},$$

where  $X$  is a random variable denoting the current state of the solid.

# Combinatorial optimization

- Analogy between a physical system and a combinatorial optimization problem:

feasible solution  $\leftrightarrow$  state of a system

objective function value  $\leftrightarrow$  energy of the state

control parameter  $c_k \leftrightarrow$  temperature ( $k_B T$ )

# Role of the control parameter

$c_k$  is often called the *temperature parameter*, and the method for choosing  $c_k$  is called the *cooling schedule*.

- ▶ For large values of  $c_k$ , large increases in cost are accepted with high probability.
- ▶ As  $c_k$  decreases, only smaller increases are accepted.
- ▶ As  $c_k$  approaches 0, no increases are accepted at all.

[Minimization objective]

# Markov chains

- ▶ Let  $S$  be a finite set of possible states of a stochastic process.
- ▶ A *Markov chain* is a sequence of trials such that the probability of outcome of a given trial depends only on the outcome of the previous trial.
- ▶ For two states  $i, j \in S$ , the transition probability at time  $k$  is

$$P_{ij}(k) = P\{X(k) = j | X(k-1) = i\}.$$

# Markov chains

- ▶  $P(k) = [P_{ij}(k)]_{i,j \in S}$  - the transition matrix.
- ▶ A Markov chain is called homogeneous if  $P(k)$  is the same for all  $k$ .
- ▶ Let  $a_i(k) = P\{X(k) = i\}$ . Then

$$a_i(k) = \sum_{j \in S} a_j(k-1)P_{ji}(k).$$

# Markov chains

- ▶ An  $n$ -vector  $x = [x_i]$  is called stochastic if

$$x_i \geq 0, \quad i = 1, \dots, n, \quad \text{and} \quad \sum_{i=1}^n x_i = 1$$

- ▶ An  $n \times m$  matrix  $X = [X_{ij}]$  is called stochastic if

$$X_{ij} \geq 0, \quad \forall i, j \quad \text{and} \quad \sum_{j=1}^m X_{ij} = 1, \quad i = 1, \dots, n.$$

# Transition probability

Consider a combinatorial optimization problem with objective function  $f$  and neighborhood function  $N$ .

- For a pair of feasible solutions  $i, j$ , the transition probability is

$$P_{ij}(k) = \begin{cases} G_{ij}(c_k)A_{ij}(c_k), & i \neq j \\ 1 - \sum_{s \in S, s \neq i} G_{is}(c_k)A_{is}(c_k), & i = j, \end{cases}$$

where  $G_{ij}(c_k)$  is the probability of generating solution  $j$  from solution  $i$  [“generation probability”], and  $A_{ij}(c_k)$  is the probability of accepting a solution  $j$  generated from solution  $i$  [“acceptance probability”].

# Transition probability

- ▶ These probabilities are conditional:

$$G_{ij}(c_k) = P_{c_k} \{\text{generate } j|i\};$$

$$A_{ij}(c_k) = P_{c_k} \{\text{accept } j|i, j\}.$$

- ▶ Denote by  $\chi_A(i)$  the characteristic function, i.e.,

$$\chi_A(i) = \begin{cases} 1, & i \in A, \\ 0, & i \notin A. \end{cases}$$

- ▶ Assume that  $\theta = |N(i)| = |N(j)|$  for any  $i, j \in S$ .



# Transition probability

- ▶ The generation probability is

$$G_{ij}(c_k) = G_{ij} = \frac{1}{\theta} \chi_{N(i)}(j), \quad \forall i, j \in S.$$

- ▶ The acceptance probability is

$$A_{ij}(c_k) = \exp \left( -\frac{(f(j) - f(i))^+}{c_k} \right).$$

# SA Convergence

- ▶ Let  $S^*$  denote the set of all global optima.
- ▶ Next we will show that under certain assumptions the SA algorithm asymptotically converges to the set of optimal solutions:

$$\lim_{k \rightarrow \infty} P\{X(k) \in S^*\} = 1.$$

# Markov chains

- ▶ For a finite homogeneous Markov chain with transition matrix  $P$  and the set of states  $S$ , a stationary distribution is a stochastic vector of length  $|S|$  such that the  $i$ -th component of  $q$  is

$$q_i = \lim_{k \rightarrow \infty} P\{X(k) = i | X(0) = j\}, \quad \forall j \in S.$$

- ▶ If such a stationary distribution exists, then

$$\lim_{k \rightarrow \infty} a_i(k) = q_i \quad \text{and} \quad q^T = q^T P.$$

- ▶ A Markov chain is called irreducible if for any  $i, j \in S$  there exists  $n > 0$  such that  $(P^n)_{ij} > 0$ .

# Markov chains

- ▶ A Markov chain with transition matrix  $P$  is called aperiodic if for each outcome  $i \in S$  the greatest common divisor of the set of all integers  $n > 0$  with  $(P^n)_{ii} > 0$  is 1.
- ▶ For an irreducible Markov chain to be aperiodic, it is sufficient that there exists  $j \in S : P_{jj} > 0$ .

# Markov chains

- ▶ **Theorem:** Let a finite homogeneous Markov chain with transition matrix  $P$  be irreducible and aperiodic. Then there exists a unique stationary distribution  $q$  with the  $i$ -th component  $q_i$  defined by

$$\sum_{j \in S} q_j P_{ji} = q_i$$

- ▶ Any probability distribution  $q$  associated with a finite, irreducible and aperiodic homogeneous Markov chain that satisfies the *detailed balance equations*

$$q_i P_{ij} = q_j P_{ji}, \quad \forall i, j \in S,$$

is the unique stationary distribution from the theorem.

# SA Convergence

**Theorem:** Assume that for our SA alg.  $c_k = c \forall k$  and

$$\forall i, j \in S \exists p \geq 1, \exists s_0, s_1, s_2, \dots, s_p \in S$$

with

$$s_0 = i, s_p = j \text{ and } G_{s_k s_{k+1}} > 0, k = 0, \dots, p-1.$$

Then the associated homogeneous Markov chain has the stationary distribution  $q(c)$  given by

$$q_i(c) = \frac{\exp(-f(i)/c)}{\sum_{j \in S} \exp(-f(j)/c)}, \forall i \in S$$

and

$$\lim_{c \rightarrow 0+} q_i(c) = \frac{1}{|S^*|} \chi_{S^*}(i),$$

where  $S^*$  is the set of global optima.

# SA Convergence proof

- (1) The Markov chain corresponding to the SA alg. is irreducible due to the first assumption (existence of  $s_0, \dots$ ).
- (2) The Markov chain is aperiodic since there exists  $j : P_{jj} > 0$ .

Hence, there exists a unique stationary distribution that can be obtained from the detailed balance equations as follows...

# SA Convergence proof

- ▶  $\forall i, j \in S, i \neq j :$

$$q_i P_{ij} = q_j P_{ji} \iff q_i A_{ij}(c) = q_j A_{ji}(c),$$

which is equivalent to

$$q_i \exp\left(-\frac{(f(j) - f(i))^+}{c}\right) = q_j \exp\left(-\frac{(f(i) - f(j))^+}{c}\right),$$

which is true for  $q_i = \exp(-f(i)/c)$ .

- ▶ Dividing by  $\sum_{j \in S} q_j$  (to get a stochastic vec.):

$$q_i = \frac{\exp(-f(i)/c)}{\sum_{j \in S} \exp(-f(j)/c)} \quad [\text{Boltzmann distr.}]$$



# SA Convergence proof

- Finally,

$$\begin{aligned}\lim_{c \rightarrow 0+} q_i &= \lim_{c \rightarrow 0+} \frac{\exp(-f(i)/c)}{\sum_{j \in S} \exp(-f(j)/c)} \\ &= \lim_{c \rightarrow 0+} \frac{1}{\sum_{j \in S} \exp((f(i)-f(j))/c)} \\ &= \frac{1}{|S^*|} \chi_{S^*}(i). \quad \square\end{aligned}$$

- Note that since  $q$  represents the stationary distribution,

$$q_i = \lim_{k \rightarrow \infty} P\{X(k) = i\},$$

thus

$$\lim_{c \rightarrow 0+} \lim_{k \rightarrow \infty} P\{X(k) = i\} = \frac{1}{|S^*|} \chi_{S^*}(i)$$

and

$$\lim_{c \rightarrow 0+} \lim_{k \rightarrow \infty} P\{X(k) \in S^*\} = 1.$$

# Cooling Schedules

A cooling schedule is given by

- ▶ an initial value of the control parameter  $c_0$ ;
- ▶ a decrement function for lowering the value of the control parameter;
- ▶ a final value of the control parameter;
- ▶ a finite number of steps for each value of the control parameter (length of each homogeneous Markov chain).

# Quasi-equilibrium

Notations:

- ▶  $L_k$  the length of the  $k$ -th Markov chain;
- ▶  $c_k$  - the corresponding value of the control parameter;
- ▶  $a(L_k, c_k)$  - the probability distribution of the solutions after  $L_k$  trials of the  $k$ -th Markov chain;
- ▶  $q(c_k)$  - the stationary distribution at  $c_k$ .

## Definition

The *quasi-equilibrium* is achieved if for some specified positive  $\epsilon$ :

$$\|a(L_k, c_k) - q(c_k)\|_1 < \epsilon.$$

# Quasi-equilibrium

- ▶ To guarantee a quasi-equilibrium, an exponential number of transitions is needed.
- ▶ Hence, we adopt the following approach:
  - ▶ Since  $\lim_{c \rightarrow \infty} q_i(c) = 1/|S|$ , for sufficiently large values of  $c_k$  quasi-equilibrium is obtained by definition (all solutions occur with equal probability).
  - ▶ Choose the decrement function and the Markov chain lengths so that quasi-equilibrium is restored at the end of each Markov chain.
  - ▶ As  $c_k \rightarrow 0+$ , arrive at the uniform distribution of the set of optimal solutions.

# Quasi-equilibrium

- ▶ Intuitively, large decrements in  $c_k$  require longer Markov chains in order to restore quasi-equilibrium at  $c_{k+1}$ .
- ▶ Trade-off between large decrements of the control parameter and small Markov chain lengths:
  - ▶ choose small decrements to avoid very long chains
  - ▶ use large values for  $L_k$  to be able to make large decrements in the control parameter.

# Cooling schedules

Two broad classes:

- ▶ Static schedules
  - ▶ parameters are fixed and cannot be changed during execution of the algorithm.
- ▶ Dynamic schedules
  - ▶ parameters are changed during execution of the algorithm.

# Static cooling schedules

Geometric cooling schedule:

- Initialization:

$$c_0 = \Delta f_{\max},$$

where  $\Delta f_{\max}$  is an estimate of the maximal difference in cost between any two neighboring solutions.

- Decrement function:

$$c_{k+1} = \alpha c_k, k = 0, 1, \dots,$$

where  $\alpha \in [0.8, 0.99]$  (typically).

# Static cooling schedules

Geometric cooling schedule:

- ▶ Final value is some small number (may be related to the smallest difference in cost between two neighbors).
- ▶ Markov chain length is some number that may be related to the size of the neighborhoods.



# Dynamic cooling schedules

- ▶ A sufficiently large **initial value of the control parameter**  $c_0$  can be obtained by requiring the initial acceptance ratio (the ratio of accepted transitions at  $c_0$ ) to be close to 1.
  - ▶ start at a small positive value  $c_0$ ;
  - ▶ multiply it by a constant factor  $> 1$  until the acceptance ratio is close to 1 (at least 0.9).
- ▶ A **final value of the control parameter** may be the value of  $c_k$  for which the value of the cost function of the solution obtained in the last trial of a Markov chain remains unchanged for a number of consecutive chains.

# Dynamic cooling schedules

- ▶ The **length of a Markov chain** may be determined by posing a lower bound on the number of transitions accepted at each value  $c_k$ .
- ▶ Since transitions are accepted with decreasing probability,  $L_k$  is usually bounded by some constant  $L_{max}$  to avoid very long chains for small values of  $c_k$ .

# Dynamic cooling schedules

- ▶ Parameters may be estimated based on a statistical analysis of the SA process.
- ▶ For the transition probabilities used in SA, the statistical analysis yields a model for the cost distribution resembling an exponential distribution for low  $c$  values and a normal distribution for high  $c$  values.
- ▶ The first two moments of this distribution are given by...

## Dynamic cooling schedules

$$E_c(f) = E_\infty(f) - \frac{\sigma_\infty^2(f)}{c} \left( \frac{\gamma c}{\gamma c + 1} \right),$$

$$\sigma_c^2(f) = \sigma_\infty^2(f) \left( \frac{\gamma c}{\gamma c + 1} \right)^2,$$

where

$$\gamma = \frac{E_\infty(f) - f^*}{\sigma_\infty^2(f)}.$$

Values of  $E_\infty(f)$  and  $\sigma_\infty^2(f)$  can be approximated as the average cost value of the solutions and the corresponding standard deviation.

# Dynamic cooling schedules

These formulae can be used to derive adaptive parameter estimates, e.g., as follows.

- ▶  $c_0 = K\sigma_{\infty}^2(f)$ , where  $K$  is a constant  
(since  $E_c(f) \approx E_{\infty}(f)$  if  $c \gg \sigma_{\infty}^2(f)$ ).  
Typically,  $K$  ranges from 5 to 10.
- ▶  $c_{k+1} = c_k \exp\left(-\frac{\epsilon c_k}{\sigma_{c_k}^2(f)}\right)$ , where  $\sigma_{c_k}^2(f)$  is approximated by the measured deviation.
- ▶ Final value of  $c_k$  is determined as follows....

# Dynamic cooling schedules

- ▶ Terminate if at the end of a Markov chain  $f'_{max} - f'_{min} = \Delta f'_{max}$ , where  $f'_{max}$  and  $f'_{min}$  are max and min costs obtained on that chain, respectively;  $\Delta f'_{max}$  is the maximum cost difference of the solutions accepted during the generation of that chain.
- ▶ set  $c = 0$ ;
- ▶ apply a local search to ensure local optimality of the output solution.

# Dynamic cooling schedules

- ▶ Statistical analysis reveals that in equilibrium, assuming a normal distribution of the cost values, the fraction of solutions generated with cost values within  $(E_c(f) - \epsilon, E_c(f) + \epsilon)$  reaches a stationary value

$$\kappa = -\text{erf}(\epsilon/\sigma_c(f)),$$

where  $\text{erf} = \frac{2}{\sqrt{\pi}} \int_0^x e^{-t^2} dt$  is the error function.

- ▶ The Markov chain length is determined by the number of trials  $L_k$  such that

$$L_k^a = pk,$$

where  $L_k^a$  is the number of accepted solutions within interval  $(E_c(f) - \epsilon, E_c(f) + \epsilon)$ ;  $p$  is a parameter characterizing the problem instance size.

# Neighborhood search

- ◇ Descent method
  - we may end up with a locally optimal solution of poor quality
- ◇ Descent-ascent methods
  - cycling may occur
- ◇ **Tabu search** introduces a memory in order to avoid some of the previously visited solutions



# Tabu search (TS)

- ◇ Let  $S$  be the feasible set.
- ◇  $N : S \rightarrow 2^S$  is a neighborhood.
- ◇ In tabu search, instead of  $N(s), s \in S$ , we will use a “reduced” neighborhood:

$$N(s, k) = N(s) - T(s, k),$$

where

- ▶  $k$  is the number of iteration;
  - ▶  $T(s, k)$  is a “tabu” list (solutions in the neighborhood that are not considered at iteration  $k$ ).
- ◇  $N(s, k)$  is a **variable neighborhood**, since the structure of this neighborhood depends on the iteration number,  $k$ .

# Tabu search (TS)

- ◇ If  $T = T(s, k)$  contains the list of  $|T|$  most recently visited solutions, then we cannot have cycles of length  $\leq |T|$ .
- ◇ For a solution  $s \in S$ , let  $\mathcal{M}(s)$  be the set of moves  $m$  that can be applied to  $s$  in order to obtain a solution  $j \in N(s)$ .
- ◇ We will use the following notation to state that  $j$  is obtained from  $i$  using move  $m$ :

$$j = i \oplus m$$

- ◇ Then  $N(s) = \{j : \exists m \in \mathcal{M}(s) \text{ s.t. } j = s \oplus m\}$ .
- ◇  $m^{-1}$  will be used to denote the move reverse to  $m$ :

$$(s \oplus m) \oplus m^{-1} = s.$$

# Tabu lists

- ◇ In some cases, instead of keeping the list of last  $|T|$  solutions visited, we may keep a list of last  $|T|$  moves (or last  $|T|$  reverse moves) performed.
- ◇ A tabu move can still be allowed if some aspiration criterion is satisfied.
  - ▶ For example, if such a move leads to a solution with the best objective value among the solutions found so far.
- ◇ Several lists  $T_r, r = 1, \dots, t$  at a time can be used to enforce tabu conditions:

$$t_r(s, m) \in T_r, r = 1, \dots, t.$$

- ◇ A move  $m$  cannot be applied to solution  $s$  if all these conditions are satisfied.

# Aspiration criteria

- ◇ We can use several aspiration level criteria as well.
- ◇ A move from tabu list is accepted if it has an aspiration level  $a(s, m)$  that is better than a threshold value  $A(s, m)$ .
- ◇ Conditions of aspiration can be written as

$$a_r(s, m) \in A_r(s, m), r = 1, \dots, a.$$

- ◇ If one of these conditions is satisfied, the move  $m$  is accepted.

# Intensification and diversification

- ◇ In some cases, the objective function  $f$  can be replaced by another function  $\bar{f}$  that can be used instead of  $f$ :

$$\bar{f} = f + \textit{intensification} + \textit{diversification}.$$

- ◇ **Intensification**: penalize solutions that are very different from detected good quality solutions.
- ◇ **Diversification**: penalize solutions that are close to the solutions that have already been analyzed.

# Tabu Search (TS)

1. Choose an initial solution  $i \in S$ . Set  $i^* = i, k = 0$ .
2. Set  $k = k + 1$  and generate a subset  $V^*$  of solutions in  $N(i, k)$  such that either one of the tabu conditions  $t_r(i, m) \in T_r$  is violated ( $r = 1, \dots, t$ ) or at least one of the aspiration conditions  $a_r(i, m) \in A_r(i, m)$  holds ( $r = 1, \dots, a$ ).
3. Choose best  $j = i \oplus m \in V^*$  with respect to  $f$  or some modified function  $\tilde{f}$ , and set  $i = j$ .
4. If  $f(i) < f(i^*)$ , then set  $i^* = i$ .
5. Update the tabu and aspiration conditions.
6. If a stopping criterion is not satisfied then go to Step 2.

# Tabu Search (TS)

Possible stopping criteria:

- ▶  $N(i, k) = \emptyset$ .
- ▶  $k$  exceeds the maximum allowable number of iterations.
- ▶ More than  $N$  iterations without an improvement.
- ▶ The obtained solution is known to be optimal.

# Quadratic Assignment Problem

Given:

- ▶  $n$  items (e.g., facilities)
- ▶ flows  $a_{ij}$  from item  $i$  to item  $j$ ,  $i, j = 1, \dots, n$  (e.g., the flow of people from facility  $i$  to facility  $j$ )
- ▶  $n$  locations (e.g., sites where the facilities may be located)
- ▶ distances  $b_{uv}$  between locations  $u$  and  $v$ ,  $u, v = 1, \dots, n$

Assign one location to each item so that the sum of the products of the flows and the distances is minimized.



# Quadratic Assignment Problem

We need to determine a permutation  $\pi^*$  that minimizes

$$f(\pi) = \sum_{i=1}^n \sum_{j=1}^n a_{ij} b_{\pi(i)\pi(j)},$$

where  $\pi(i)$  is the  $i$ -th component of the permutation  $\pi$  (the location of facility  $i$ ).

# Solutions and neighborhood

- ▶ The set  $S$  of feasible solutions is the set of all permutations of  $n$  items.
- ▶ The neighborhood  $N(\pi)$  of a solution  $\pi$  is the set of permutation that can be obtained from  $\pi$  by exchanging two elements of  $\pi$ , i.e.,  $\pi' \in N(\pi)$  if there exist  $i, j \in \{1, \dots, n\}$  such that

$$\pi'(i) = \pi(j), \pi'(j) = \pi(i)$$

and

$$\pi'(k) = \pi(k) \quad \forall k \neq i, j.$$

## An exchange move

$$\pi = \left( \begin{array}{ccccccc} 1 & \dots & \boxed{i} & \dots & \boxed{j} & \dots & n \\ \pi(1) & \dots & \pi(i) & \dots & \pi(j) & \dots & \pi(n) \end{array} \right)$$
$$\pi' = \left( \begin{array}{ccccccc} 1 & \dots & \boxed{i} & \dots & \boxed{j} & \dots & n \\ \pi(1) & \dots & \pi(j) & \dots & \pi(i) & \dots & \pi(n) \end{array} \right)$$

- ▶ A move  $m$  is defined by both locations  $\pi(i)$  and  $\pi(j)$  whose items are exchanged.
- ▶ The set of moves does not depend on the current solution!

# Tabu and aspiration conditions

Tabu rule:

- ▶ Forbid the moves placing both facilities back in locations they have already occupied in the last  $|T|$  iterations.

Aspiration conditions:

1. Perform a tabu move if it improves the best solution  $\pi^*$  found so far.
2. If a move has not been chosen during a large number  $A$  of iterations, it is performed regardless of the value of the solution it leads to [diversification].

# Aspiration conditions

- ▶  $A_1(\pi, m) = \{x \in \mathbb{R} : x < f(\pi^*)\},$
- ▶  $A_2(\pi, m) = \{m' \in M(\pi) : m' \text{ has not been chosen in the last } A \text{ iterations}\},$
- ▶  $a_1(\pi, m) = f(\pi \oplus m),$
- ▶  $a_2(\pi, m) = m.$

The best solution  $\pi' \in V^*$  at a step will be chosen using the modified function

$$\tilde{f}(\pi \oplus m) = \begin{cases} f(\pi \oplus m), & \text{if } m \notin A_2(\pi, m), \\ -\infty, & \text{if } m \in A_2(\pi, m). \end{cases}$$

# Implementation

- ▶ Use  $n \times n$  matrix  $Z$  with elements  $Z_{ip}$  corresponding to the iteration number at which facility  $i$  has been moved to location  $p$ .
- ▶ Let  $m_{ij}$  be the move exchanging locations of facilities  $i$  and  $j$ , and let  $k$  be the current iteration number. Then a move is tabu if

$$Z_{i\pi(j)} + |T| \geq k \text{ and } Z_{j\pi(i)} + |T| \geq k,$$

# Implementation

- ▶ The tabu status of a move  $m_{ij}$  is cancelled if the following aspiration condition is satisfied:

$$f(\pi \oplus m_{ij}) \leq f(\pi^*) \text{ or } \max\{Z_{i\pi(j)} + A, Z_{j\pi(i)} + A\} \leq k.$$

- ▶ Hence, using matrix  $Z$  we can check if a solution belongs to  $V^*$  in constant time.
- ▶ Note that the memory used for matrix  $Z$  is  $O(n^2)$ , which is the same as the memory needed to store the problem data.

# Implementation

Denote by  $\Delta(\pi, i, j)$  the value of the move  $m_{ij}$  from  $\pi$ :

$$\Delta(\pi, i, j) = f(\pi \bigoplus m_{ij}) - f(\pi).$$

$$\begin{aligned} \Delta(\pi, i, j) = & \sum_{p=1}^n \left[ a_{pi}(b_{\pi(p)\pi(j)} - b_{\pi(p)\pi(i)}) + a_{pj}(b_{\pi(p)\pi(i)} - b_{\pi(p)\pi(j)}) \right. \\ & \left. + a_{ip}(b_{\pi(j)\pi(p)} - b_{\pi(i)\pi(p)}) + a_{jp}(b_{\pi(i)\pi(p)} - b_{\pi(j)\pi(p)}) \right] \\ & + \left[ a_{ii}(b_{\pi(j)\pi(j)} - b_{\pi(i)\pi(i)}) + a_{ij}(b_{\pi(j)\pi(i)} - b_{\pi(i)\pi(j)}) \right. \\ & \left. + a_{ji}(b_{\pi(i)\pi(j)} - b_{\pi(j)\pi(i)}) + a_{jj}(b_{\pi(i)\pi(i)} - b_{\pi(j)\pi(j)}) \right]. \end{aligned}$$



# Implementation

If  $\pi' = \pi \oplus m_{uv}$ , then for  $i \neq u, v$  and  $j \neq u, v$ :

$$\begin{aligned}\Delta(\pi', i, j) &= \Delta(\pi, i, j) \\ &+ (a_{ui} - a_{uj} + a_{vj} - a_{vi})(b_{\pi'(v)\pi'(i)} - b_{\pi'(v)\pi'(j)} + b_{\pi'(u)\pi'(j)} - b_{\pi'(u)\pi'(i)}) \\ &+ (a_{iu} - a_{ju} + a_{jv} - a_{iv})(b_{\pi'(i)\pi'(v)} - b_{\pi'(j)\pi'(v)} + b_{\pi'(j)\pi'(u)} - b_{\pi'(i)\pi'(u)}).\end{aligned}$$

- ▶ By storing the values  $\Delta(\pi, i, j)$  of all possible moves  $m_{ij}$  from  $\pi$ , we perform each iteration of TS in  $O(n^2)$  time.
- ▶ Suggested values of TS parameters:
  - ▶  $|T| \in [\lfloor 0.9n \rfloor, \lceil 1.1n + 4 \rceil]$ ;
  - ▶  $A = 5n^2$ .

# TS for the MIS problem

- ▶ Given a graph  $G = (V, E)$ , determine a maximum subset  $I$  of  $V$  such that  $E(I) = \emptyset$ , where  $E(I)$  is a subset of edges having both endpoints in  $I$ .
- ▶ We use TS in an attempt to determine an independent set of size  $k$  in  $G$ .
- ▶ A solution is any subset  $I \subset V$  of size  $k$ , and the objective function is  $|E(I)|$ .
- ▶ The neighborhood consists of exchanging a vertex  $v \in I$  with a vertex  $w \in V \setminus I$ .
- ▶  $|N(I)| = k(n - k) = O(nk)$ , where  $n = |V|$ .

# TS for the MIS problem

- ▶ For a vertex  $v \in V$ , denote by  $\Gamma_I(v)$  the set of vertices  $w \in I$  that are adjacent to  $v$ :

$$\Gamma_I(v) = \{w \in I : (v, w) \in E\}.$$

- ▶ The vertices  $v \in I$  are sorted according to nonincreasing values of  $|\Gamma_I(v)|$ , and the vertices  $w \in V \setminus I$  are sorted according to nondecreasing values of  $|\Gamma_I(w)|$ .
- ▶ Keeping these two lists sorted requires  $O(n)$  time.

# TS for the MIS problem

Three tabu lists are used:

1.  $T_1$  stores the last visited solutions;
2.  $T_2$  contains the last vertices introduced to  $I$ ;
3.  $T_3$  contains the last vertices removed from  $I$ ;

# TS for the graph coloring problem

- ▶ A proper  $k$ -coloring of the vertices of a graph  $G = (V, E)$  is a partition of the vertex set  $V$  into  $k$  independent sets  $I_1, I_2, \dots, I_k$ .
- ▶ We relax the independence constraint and assume that a feasible solution is any partition  $(I_1, I_2, \dots, I_k)$  of  $V$  into  $k$  sets.
- ▶ By defining  $E(I_j)$  as the set of edges having both endpoints in  $I_j$ , the objective is to minimize  $\sum_{j=1}^k |E(I_j)|$ .
- ▶ A solution of value 0 corresponds to a proper  $k$ -coloring of  $G$ .

# TS for the graph coloring problem

- ▶ Neighborhood  $N(i)$  of a solution  $i$  consists of all solutions generated from  $i$  by the following modification:  
*move a vertex from  $I_j$  that is adjacent to another vertex in  $I_j$  to a set  $I_p, p \neq j$ .*
- ▶ When a vertex  $v$  is moved from  $I_j$  to  $I_p$ , the pair  $(v, j)$  is introduced in the tabu list  $T$ , meaning it is forbidden to move  $v$  to  $I_j$  during  $|T|$  iterations.

# TS for the VRP

- ▶ Let  $V = \{V_1, \dots, V_n\}$  be a set of customers and  $V_0$  a depot at which  $m$  identical vehicles of limited capacity are based.
- ▶ In VRP we need to design a set of least-cost vehicle routes in such a way that each route starts and ends at the depot, and each customer is visited exactly once by exactly one vehicle.
- ▶ Every client has an associated demand, and the total demand of any vehicle route may not exceed the vehicle capacity.
- ▶ The total duration of any route (travel plus service time) may not exceed a preset bound.

# TS for the VRP

- ▶ Most approaches use a neighborhood function where customers are moved between routes.
- ▶ With a short-memory TS, it appears that the customers located close to the depot are moved more frequently.
- ▶ To diversify the search, a penalty is added to the value of a move that is proportional to the frequency of the use of the move.



# TS for the VRP

- ▶ We can relax the capacity or total route length constraints and add components proportional to the violation of these constraints to the objective function.
- ▶ The proportionality factor can be automatically updated by increasing it when the recently visited solutions are all violating the constraints, and by decreasing it when they are feasible.

# TS for the TSP

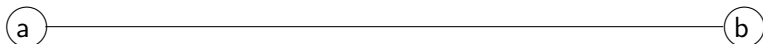
- ▶ Basic moves: 2-Opt exchange
- ▶ Glover (1986):
  - ▶ Add the shortest of the two deleted edges to the tabu list. A move is tabu if it tries to reinsert an edge from the tabu list.
  - ▶ The aspiration level  $A(L)$  is the length of the shortest tour that has ever been converted to one of length  $L$  in a single move.
- ▶ Knox (1994):
  - ▶ Tabu list - pairs of edges deleted in a previous move. A move is tabu if it reinserts such a pair.
  - ▶ A tabu move is accepted if it produces a tour better than the tour which existed when the corresponding pair of edges was most recently deleted.

# The Lin-Kernighan algorithm

- ▶ Recall the Lin-Kernighan algorithm for TSP.
- ▶ The LKA's inner loop can be viewed as a variant of tabu search over the 2-Opt neighborhood structure.
- ▶ The inner loop in turn is embedded in an overall procedure that can be viewed as embodying the tabu search idea of *intensification*.
- ▶ In practice, the resulting algorithm yields significantly better results than 3-Opt with only a modest increase in running time.

# The basic idea of LKA

- ▶ Given a tour, we can remove an edge  $(a, b)$  to obtain a Hamilton path with endpoints  $a$  and  $b$ . Let  $a$  be fixed and  $b$  - variable.



- ▶ If we add an edge  $(b, c)$ , a cycle is formed; removal of  $(c, d)$  produces a new Hamilton path with a new variable endpoint  $d$ .



- ▶ This operation is called a *rotation*.

# The basic idea of LKA

- ▶ A move of the LKA from one tour to a neighbor consists of
  - ▶ removing an edge to form a Hamilton path;
  - ▶ performing a sequence of greedy rotations [The Lin-Kernighan inner loop];
  - ▶ reconnecting the two endpoints to form a tour.

# The Lin-Kernighan inner loop



- ▶ The new neighbor  $c$  of  $b$  must be such that the total length of edges in the graph obtained from the current path by adding the edge  $(b, c)$  is less than the length of the best tour found so far.
- ▶ Tabu lists:
  - ▶ added edges (like  $(b, c)$ );
  - ▶ deleted edges (like  $(c, d)$ ).

A move is tabu if it adds an edge on the *deleted* list or deletes an edge on the *added* list.

# The Lin-Kernighan inner loop

- ▶ The LK search places no bound on the length of the tabu list and makes no use of aspiration levels.
- ▶ Thus, once a particular edge has been added to (deleted from) the path, it can no longer be deleted (added).
- ▶ This implies that at most  $n$  moves can be made before the search terminates (it stops when there are no qualifying nontabu moves for the current path).

# The Lin-Kernighan inner loop

The moves are made as follows:

- ▶ For each qualifying, nontabu move, if the resulting tour is better than the best found so far, it is saved as the best current solution.
- ▶ However, the qualifying, nontabu move that yields the shortest new path is chosen as the current solution.
- ▶ This path may still be longer than the previous one, and its corresponding tour may be longer than the corresponding tour for the previous path.



# Key differences between TS and LKA

- (1) LKA has unbounded tabu lists with no aspiration levels;
  - ▶ Restricts us to at most  $n$  steps per LK search.
  - ▶ This number is significantly lower in practice.
- (2) In LKA, the set of moves that are considered at each step is restricted.
  - ▶ The success of LKA can be explained by the way starting tours are obtained and by the large number of LK searches performed [The principle of **intensification**].

# Genetic Algorithms (GA)

# Evolutionary Algorithms

- ▶ Evolutionary algorithms (EAs) try to model natural evolution process.
- ▶ EAs have been first proposed in 1960s.
- ▶ Genetic algorithm and the evolution strategies have been proposed in 1970s.
- ▶ Evolution strategies driven mainly by selection and mutation (variation) have been developed by Rechenberg (1973) and Schwefel (1981) for continuous optimization.
- ▶ Next we review the most famous such strategy, the  $(\mu + \lambda)$  ES.

## $(\mu + \lambda)$ Evolution strategy (ES)

1. Create an initial population of size  $\lambda$ .
  2. Compute the fitness  $F(x_i)$ ,  $i = 1, \dots, \lambda$ .
  3. Select the  $\mu \leq \lambda$  best individuals.
  4. Create  $\lambda/\mu$  offsprings of each of the  $\mu$  individuals by small variation.
  5. If not stop, return to Step 2.
- Variation: randomly generate a number from a normal distribution with zero mean and add it to the value of the continuous variable. The amount of variation can be adopted by changing the variance of the normal distribution. The most popular choice of parameters is  $\mu = \lambda = 1$ .

# Evolution strategies vs genetic algorithms

- ▶ In biological terms, evolution strategies model natural evolution by asexual reproduction with mutation and selection.
- ▶ In contrast, genetic algorithms (GAs) are search algorithms that model sexual reproduction.
- ▶ In sexual reproduction, two parent strings are recombined into an offspring. This process is called crossover.

# Genetic Algorithms

- ▶ Introduced by John Holland in 1975.
- ▶ GAs attempt to simulate the process of evolution, which can be viewed as an optimization process.

## Features of GAs:

- ▶ Feasible solutions are encoded as chromosomes.
- ▶ At each iteration, a GA works with a “population” of solutions instead of just one solution.
- ▶ The population is constructed using a *selection* operation.

# Features of GAs

- ▶ Fitter individuals are combined in pairs of parents to produce children (1 or 2) using a *crossover* operator, which combines chromosomes of the two parents in order to obtain children's chromosomes.
- ▶ To introduce diversity to the search, a *mutation* operator is used.

# A general scheme

1. Generate an initial population  $S = \{s_1, \dots, s_k\}$ .
2. Apply a local search alg.  $A$  to each sol. in  $S$  and replace each sol. in  $S$  with the corresponding local optimum.
3. While stopping criterion is not satisfied
  - (a) Select  $k'$  distinct subsets of size 1 or 2 as parents [the mating strategy]
  - (b) For each 1-element subset perform a mutation operation to obtain a new solution.
  - (c) For each 2-element subset, perform a crossover operation.
  - (d) Apply local search alg.  $A$  to each of the  $k'$  new solutions obtained in (b) and (c), resulting in set  $S'$  of solutions.
  - (e) Choose  $k$  survivors from  $S \cup S'$  using a selection strategy to obtain new  $S$ .
4. Return the best solution found.



# Genetic representation

- ▶ A feasible solution is represented as a bitstring of length  $n$ , the *chromosome*.
- ▶ The positions of the strings are called *loci* of the chromosome.
- ▶ The variable at a locus is called a *gene*, its value is an *allele*.
- ▶ The set of chromosomes is called the *genotype*.
- ▶ The genotype defines a *phenotype* (the individual) with a certain fitness.

# Selection procedures

Selection procedures are usually based on individual fitness.

- ▶ **Breeder selection:**

- ▶ Order individuals in nonincreasing order of their fitness;
- ▶ Select  $T\%$  of the top individuals.

- ▶ **Tournament selection:**

- ▶ Randomly pick pairs (groups) of individuals;
- ▶ Select the best individual in each group.

- ▶ **Roulette wheel selection...**

# Roulette wheel selection

- ▶ The probability  $p_i$  of selecting individual  $i$  is proportional to the individual's fitness:

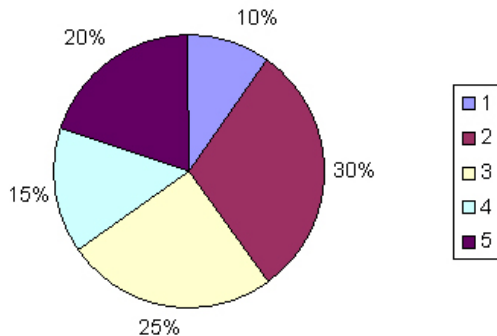
$$p_i = \frac{f_i}{\sum_{j=1}^n f_j},$$

where

- ▶  $f_i$  is the fitness of individual  $i$ ;
- ▶  $n$  is the total number of individuals.

# Roulette wheel selection

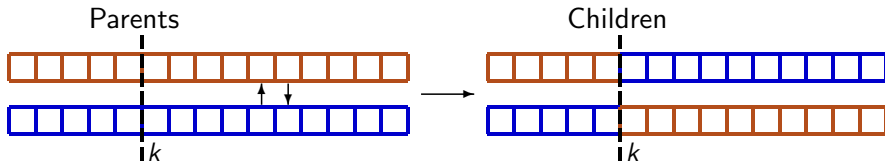
Example: 5 individuals with fitness 10, 30, 25, 15, and 20.



# Crossover operation

## ► One-point crossover:

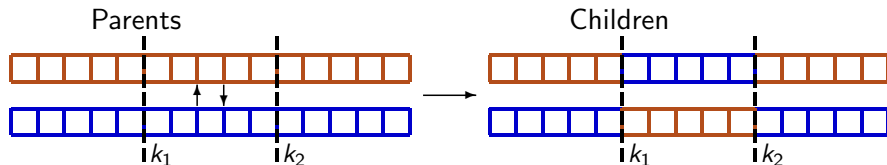
- randomly pick the position  $k$ ;
- exchange the parts of the two strings after position  $k$ .



# Crossover operation

## ► Two-point crossover:

- randomly pick two positions  $k_1$  and  $k_2$ ;
- exchange the parts of the two strings between positions  $k_1$  and  $k_2$ .



# Crossover operation

- **Uniform crossover:**

Given two parent strings

$$x = (x_1, x_2, \dots, x_n) \text{ and } y = (y_1, y_2, \dots, y_n),$$

their child string  $z = (z_1, z_2, \dots, z_n)$  is obtained as follows:

$$z_i = \begin{cases} x_i, & \text{with probability } p_i, \\ y_i, & \text{with probability } 1 - p_i, \end{cases} \quad i = 1, \dots, n.$$

Normally,  $p_i = 0.5, i = 1, \dots, n$ .

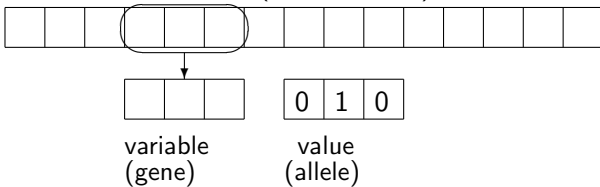
# Mutation operation

Common mutation operators:

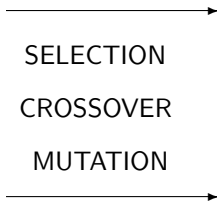
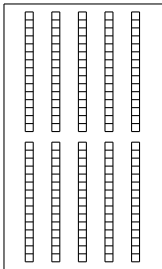
- ▶ Change the value of each bit with probability  $p$ .
- ▶ Pick randomly  $r\%$  of bits to be flipped.



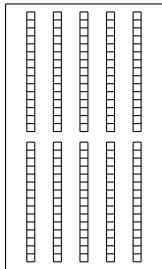
## A feasible solution (chromosome)



Current population



New population



## Example: designing a can



- ▶ A cylindrical can is defined by two parameters:
  - ▶ diameter  $d$ ;
  - ▶ height  $h$ .
- ▶ The can needs to have a volume of at least 300 ml.
- ▶ The objective of the design is to minimize the cost of the can material.

# A nonlinear programming formulation

$$\text{Minimize} \quad f(d, h) = c \left( \frac{\pi d^2}{2} + \pi dh \right),$$

$$\text{subject to} \quad g(d, h) = \frac{\pi d^2 h}{4} \geq 300,$$

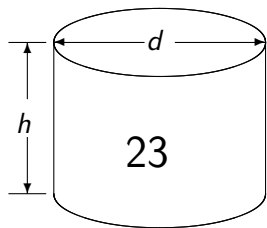
$$d_{\min} \leq d \leq d_{\max},$$

$$h_{\min} \leq h \leq h_{\max}.$$

- ▶  $c$  is the cost (per square cm) of the can material;
- ▶  $d_{\min}$  and  $d_{\max}$  are the smallest and the largest allowable diameter, respectively;
- ▶  $h_{\min}$  and  $h_{\max}$  are the smallest and the largest allowable height, respectively.

## Representing a solution

- ▶ We use five bits to code each of the two decision variables.
- ▶ Hence the overall string length is 10.
- ▶ Assume  $c = 0.065$ .



$$(d, h) = (8, 10) \text{ cm}$$

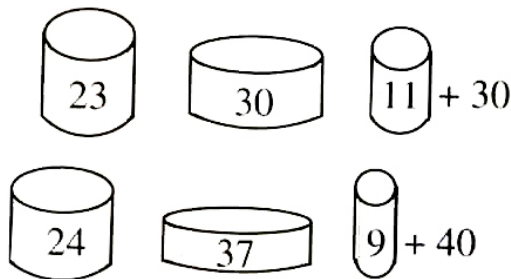
Chromosome:



$$f(8, 10) = 0.065 \left( \frac{\pi 8^2}{2} + \pi \times 8 \times 10 \right) \approx 23.$$

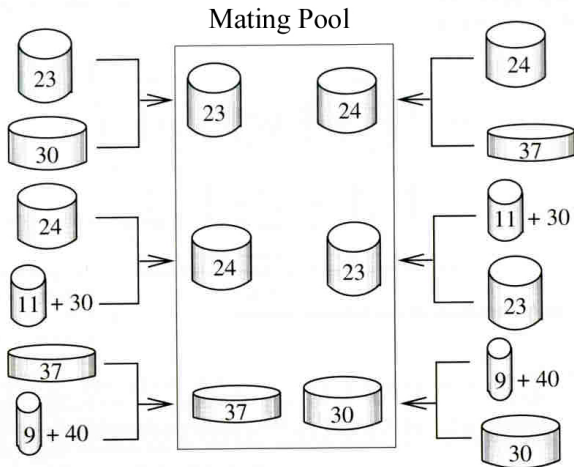
# Fitness of a solution

Phenotypes of a random population of six cans and their fitness.



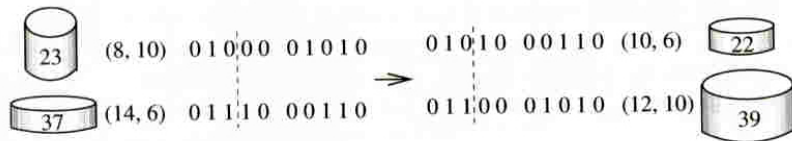
- ▶ Two infeasible solutions (volume less than 300 ml) are penalized by adding extra artificial cost.

# Tournament selection



# Crossover and mutation

The single point crossover operation:



The bit-wise mutation operation:



(the fourth bit is mutated)

# Non-binary encoding: TSP

We will represent a feasible solution of TSP as a cyclic permutation.

## Example:

- ▶ Consider an instance of TSP with 8 cities.
- ▶ Assume that the current solution is given by the tour  $1 - 8 - 6 - 2 - 5 - 7 - 4 - 3 - 1$ .
- ▶ The cyclic permutation representation of this tour:

$$\pi = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \\ 8 & 5 & 1 & 3 & 7 & 2 & 4 & 6 \end{pmatrix}$$



# One-point crossover

- ▶ Select  $k \in \{1, \dots, n\}$  randomly.
- ▶ Visit the first  $k$  cities in the order they were visited in the first parent tour, and the remaining cities in the order they were visited in the second parent tour.
- ▶ **Example:**  $k = 3$ 
  - ▶ parent 1: **1** – **8** – **6** – 2 – 5 – 7 – 4 – 3 – 1
  - ▶ parent 2: 1 – **5** – **4** – **3** – **7** – 8 – **2** – 6 – 1
  - ▶ child:     **1** – **8** – **6** – **5** – **4** – **3** – **7** – **2** – **1**

# Two-point crossover

- ▶ Select  $k_1, k_2 \in \{1, \dots, n\}$  randomly.
- ▶ Visit cities  $1, \dots, k_1$  and  $k_2, \dots, n$  in the order they were visited in the first parent tour, and the remaining cities in the order they were visited in the second parent tour.
- ▶ **Example:**  $k_1 = 3, k_2 = 7$ 
  - ▶ parent 1: **1** – **8** – **6** – 2 – 5 – 7 – **4** – **3** – 1
  - ▶ parent 2: 1 – **5** – 4 – 3 – **7** – 8 – **2** – 6 – 1
  - ▶ child:     **1** – **8** – **6** – **5** – **7** – **2** – **4** – **3** – **1**

# Crossover with random keys

(analog of the uniform crossover)

- ▶ Generate  $n$  random keys for each tour.
- ▶ Sort the keys for each tour in nondecreasing order.
- ▶ Assign keys to cities based on the order they were visited by the tour.
- ▶ For each city, create a new key, which is equal to its key in the first/second tour with probab. 0.5.
- ▶ Create a new tour that visits cities in the order corresponding to the nondecreasing order of the new keys.

# Crossover with random keys

Example:

▶ parent 1: 1 – 8 – 6 – 2 – 5 – 7 – 4 – 3 – 1

random keys:    1    8    6    2    5    7    4    3  
                  5    7    36   48   52   67   88   93

▶ parent 2: 1 – 5 – 4 – 3 – 7 – 8 – 2 – 6 – 1

random keys:    1    5    4    3    7    8    2    6  
                  7    9    11   48   62   78   83   99

▶ A child:    1    2    3    4    5    6    7    8  
              7   48   48   88   52   99   67   78

new tour: 1 – 2 – 3 – 5 – 7 – 8 – 4 – 6 – 1

# Crossover based on random insertion

- ▶ Random keys for the child are generated as in the random key representation technique.
- ▶ Create a 3-city tour visiting the cities with lowest keys.
- ▶ Insert the next city with smallest key in the current tour in an optimal way (so that the resulting tour has minimum cost among tours obtained from all possible insertions of that city).

# Distance-preserving crossover

- ▶ Distance between two tours is defined as the number of different edges (present in one tour, absent in the other).
- ▶ i.e., if two tours on  $n$  cities have  $k$  edges in common, then the distance between them is  $n - k$ .
- ▶ **Crossover** is performed as follows:
  - ▶ fix the edges that coincide;
  - ▶ connect the fixed parts without using edges from any of the two tours (can be done greedily).
- ▶ **Mutation**: randomly exchange 2 cities.

# Population genetics

- ▶ Population genetics analyzes the evolution of genetic populations.
- ▶ Theoretical analysis of evolution tries to model natural selection.

*The preservation of favourable variations and the rejection of injurious variations, we call Natural Selection*

Darwin, ‘‘On the Origins of Species by Means of Natural Selection’’.

# Population genetics

- ▶ The behavior of genetic populations is very difficult to analyze mathematically.
- ▶ Population genetics proposed several approaches investigating specific aspects of genetic populations:
  - ▶ phenotypic (biometricians): use correlation and regression to quantify relation between offspring and parent
  - ▶ genotypic (Mendelians): use genetic chance model to compute the change of the gene frequencies in the population
  - ▶ statistical (breeders): predict the outcome of selection experiments with the equation for the response to selection



# Population genetics

- ▶ The population genetics ideas can be used to analyze GAs theoretically.
- ▶ To describe the initial state and the evolution of an artificial population of a GA, the following parameters are needed:
  - ▶ population size  $N$ ,
  - ▶ initial frequency of the alleles  $p_0$ ,
  - ▶ number of loci  $n$ ,
  - ▶ mutation rate  $m$ ,
  - ▶ intensity of selection  $I$ .
- ▶ Some of the parameters are usually fixed to simplify the analysis.

# Gene pool recombination (GPR)

- ▶ In gene pool recombination (GPR), the two parent alleles are independently chosen for each locus out of the alleles for that locus in the gene pool defined by the selected parent population.
- ▶ After that the offspring allele is computed by any of the standard two-parent recombination (TPR) schemes.
- ▶ GPR is convenient to use in theoretical analysis since it creates a binomial distribution.

# Response to selection

- ▶ Response to selection,  $R$ , is the change produced by selection, which is defined as the difference between the population mean fitness of generation  $t + 1$  and generation  $t$ :

$$R(t) = \bar{f}(t + 1) - \bar{f}(t).$$

- ▶ Breeders measure the selection with the selection differential,  $S(t)$ , which is defined as the difference between the mean fitness of the selected parents  $\bar{f}_S(t)$  and the mean fitness of the population:

$$S(t) = \bar{f}_S(t) - \bar{f}(t).$$

# Truncation selection

- ▶ Breeders try to predict  $R(t)$  from  $S(t)$ .
- ▶ Truncation selection (mass selection): select *Trunc* best individuals as parents (10% to 50%).
- ▶ The response to selection can be approximated as

$$R(t) = b \cdot S(t).$$

- ▶ We are interested in predicting the cumulative response  $R_s$  for  $s$  generations of the breeding scheme:

$$R_s = \sum_{t=0}^s R(t) = \bar{f}(s+1) - \bar{f}(0).$$

# Selection intensity

- ▶ Breeders use the quantity

$$I = S(t)/\sigma(t)$$

to measure the strength of selection.

- ▶  $\sigma(t)$  is the standard deviation of the fitness of the individuals;  
 $I$  is called the *selection intensity*.
- ▶  $I$  can be computed analytically if the fitness values are normally distributed.
- ▶ For arbitrary distributions, there is an estimate:

$$I \leq \sqrt{(100 - Trunc)/Trunc}.$$

# Response to selection

- Using

$$R(t) = b \cdot S(t),$$

$$I = S(t)/\sigma(t),$$

we obtain the following equation for response to selection:

$$R(t) = b \cdot I \cdot \sigma(t).$$

- The science of artificial selection consists in estimating  $b$  and  $\sigma(t)$  depending on the specific traits to be improved.

# Response to selection and GAs

The response to selection equation can be used in analysis of GAs in several ways:

- ▶ Given a class of selection methods with the same selection intensity, the preferred method generates parent populations with the highest standard deviation.
  - ▶ Blickle & Thiele (1995): tournament selection creates a parent population with a higher standard deviation than truncation selection with the same selection intensity.
- ▶ To maximize the response, we need to design a crossover operator that maximizes  $b \cdot I \cdot \sigma(t)$  [difficult to fulfill].

# Response to selection and GA

- ▶ We will use the response to selection equation to predict the behavior of a GA with *uniform crossover*.
- ▶ We will use the binary ONEMAX function of size  $n$ , with the fitness given by the number of 1's in the binary string.
- ▶ To estimate  $b$ , we make a regression of the midparent fitness value (the average of the fitness of the two parents) to the offspring. The average fitness of the offspring is the same as the average of the midparents, so  $b = 1$ .
- ▶ Assuming that uniform crossover is a random process creating a binomial fitness distribution with probability  $p(t)$  (the probability that there is 1 at a locus), the standard deviation is given by  $\sigma(t) = \sqrt{np(t)(1 - p(t))}$ .



# Response to selection and GA

## Theorem

*If the population is large enough to converge to the optimum and if the selection intensity  $I$  is greater than 0, the response to selection for the ONEMAX function is approximately given by*

$$R(t) = \frac{I}{\sqrt{n}} \sqrt{p(t)(1 - p(t))}.$$

*The number of generations needed until equilibrium is approximately*

$$GEN_e = \left[ \frac{\pi}{2} - \arcsin(2p_0 - 1) \right] \frac{\sqrt{n}}{I}.$$

# Response to selection and GA

**Proof:** Noting that  $R(t) = n(p(t+1) - p(t))$ , we have

$$p(t+1) - p(t) = \frac{I}{\sqrt{n}} \sqrt{p(t)(1-p(t))},$$

which can be approximated by the differential equation

$$\frac{dp(t)}{dt} = \frac{I}{\sqrt{n}} \sqrt{p(t)(1-p(t))} \text{ with } p(0) = p_0.$$

The solution of this equation is given by

$$p(t) = 0.5 \left( 1 + \sin \left[ \frac{I}{\sqrt{n}} t + \arcsin(2p_0 - 1) \right] \right).$$

Setting  $p(GEN_e) = 1$ , we obtain the value of  $GEN_e$ . □

# Response to selection and GA

**Remark:** From simulation, the variance is slightly less than that of the binomial distribution. Multiplying by a factor  $\pi/4.3$ , the resulting equations fit simulation results very well:

$$R'(t) = \frac{\pi}{4.3} \frac{l}{\sqrt{n}} \sqrt{p(t)(1-p(t))}.$$

$$GEN'_e = \frac{\pi}{4.3} \left[ \frac{\pi}{2} - \arcsin(2p_0 - 1) \right] \frac{\sqrt{n}}{l}.$$

- ▶ Note: the equations are only valid if the population is large enough to converge to the optimum.
- ▶ The minimum acceptable population size  $N^*$  is determined based on  $n$ ,  $l$  and  $p_0$ .

# Natural selection

- ▶ In quantitative genetics, natural selection is modeled by proportionate selection.
- ▶ Denote by  $q_i(t) \in [0, 1]$  the frequency of genotype  $i$  in a population of size  $N$  at generation  $t$ ;  $f_i$  - its fitness.
- ▶ Then the genotype frequency  $q_{i,S}$  of the selected parents is given by

$$q_{i,S}(t) = q_i(t) \frac{f_i}{\bar{f}(t)},$$

where  $\bar{f}(t) = \sum q_i(t)f_i$  is the average fitness of the population.

# Natural selection

## Theorem

*In proportionate selection the selection differential  $S(t) = \bar{f}_S(t) - \bar{f}(t)$  is given by*

$$S(t) = \frac{\sigma^2(t)}{\bar{f}(t)}.$$

*Assuming ONEMAX( $n$ ) fitness function and binomially distributed genotypes, we have  $\sigma^2(t) \approx np(t)(1 - p(t))$ . Then the response to selection is*

$$R(t) = 1 - p(t).$$

*If the population is large enough, the number of generations until  $p(t) = 1 - \epsilon$  for large  $n$  is given by*

$$GEN_{1-\epsilon} \approx n \ln \frac{1 - p_0}{\epsilon}.$$

# Truncation vs natural selection

- ▶ With truncation (breeder) selection, the population will converge in  $O(\sqrt{n})$  generations.
- ▶ With natural (proportionate, roulette wheel) selection, we have the convergence in  $O(n \log n)$  generations (assuming  $\epsilon = 1/n$ ).
- ▶ Hence, breeder selection seems to be more effective method of optimization than proportionate selection.

# Mutation

Empirical observation:

- ▶ For  $ONEMAX(n)$ , a truncation threshold of  $T = 50\%$ , a mutation rate of  $m = 1/n$ , and  $n \gg 1$ , the response to selection of a large population changing by mutation only can be approximated by

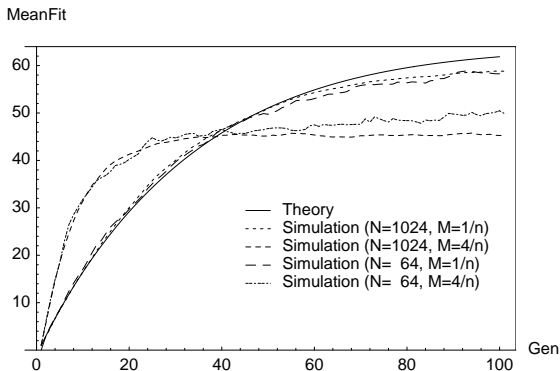
$$R(t) = 2 - 2p(t).$$

- ▶ The number of generations to reach  $p(t) = 1 - \epsilon$  is given by

$$GEN_{1-\epsilon} \approx \frac{n}{2} \ln \frac{1 - p_0}{\epsilon}.$$

# Mutation

- Theory and simulation results for various population sizes  $N$  and mutation probabilities  $M$ :



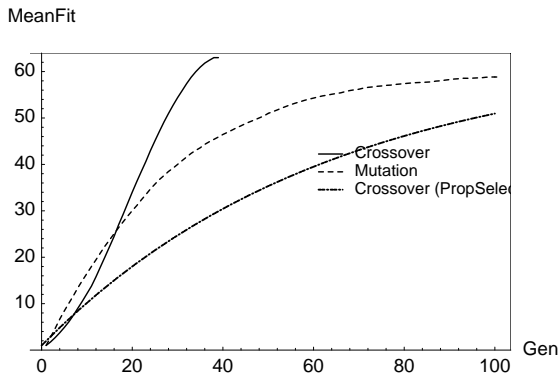


# Mutation vs crossover

- ▶ Using either recombination or mutation, optimum is obtained.
- ▶ If  $p_0 = 1 - 1/n$  (only one bit is wrong on average):
  - ▶ Mutation will need  $O(n)$  trials to change the incorrect bit.
  - ▶ Uniform crossover of two strings, each with one wrong bit, will generate the optimal string with probability  $1/4$ , independent of  $n$ .
- ▶ If  $p_0 = 1/n$ , mutation is much more efficient than crossover.

# Mutation vs crossover

- Comparison of crossover and mutation for *ONEMAX*(64) with  $p_0 = 1/64$ . Crossover with truncation selection is performed using  $T = 0.5$ .



# Summary of results

- ▶ Any finite genetic population of size  $N$  will converge to a single genotype, even if selection is not applied (*genetic drift* phenomenon).
- ▶ The expected number of generations until convergence  $GEN_e$  is surprisingly low:

$$E(GEN_e) \approx 1.4N(0.5 \ln n + 1),$$

where  $n$  is the number of genes.

- ▶ This equation is for random mating with recombination but without selection and mutation.
  - ▶ linear in  $N$  and logarithmic in  $n$ .

# Summary of results

- ▶ Truncation selection:

$$GEN_e \approx \left( \frac{\pi}{2} - \arcsin(2p_0 - 1) \right) \frac{\sqrt{n}}{l}; \quad N^* \approx \sqrt{n} \ln(n) f_1(p_0) f_2(l).$$

- ▶ Proportionate selection:

$$GEN_{1-1/n} \approx n \ln[n(1 - p_0)]$$

- ▶ Mutation only: The mutation rate  $m = 1/n$  is a recommended choice. Using the (1+1)-strategy (one parent, one offspring; the better of the two survives) with truncation  $T = 0.5$ , for large population size we have

$$GEN_{1-1/n} \approx \frac{n}{2} \ln[n(1 - p_0)].$$

# Replicator Dynamics

## Selection and the fundamental theorem

- ▶ Consider a single chromosomal locus with  $n$  alleles  $A_1, \dots, A_n$ .
- ▶ Let  $p_1, p_2, \dots, p_n$  be the gene frequencies at the mating stage of the parental generation.
- ▶ We assume random mating
  - ▶ The probability that a zygote<sup>1</sup> carries the gene pair  $(A_i, A_j)$  is  $p_i p_j$ .

---

<sup>1</sup>a cell that is the result of fertilization

## Selection and the fundamental theorem

- ▶ Denote by  $w_{ij}$  the probability that an individual defined by pair  $(A_i, A_j)$  survives to adult age [*selective values*]
- ▶ Then  $w_{ij} \geq 0$  and  $w_{ij} = w_{ji}$  since the gene pairs  $(A_i, A_j)$  and  $(A_j, A_i)$  belong to the same genotype.
- ▶ The selection matrix

$$W = \begin{bmatrix} w_{11} & \cdots & w_{1n} \\ \cdots & \ddots & \cdots \\ w_{n1} & \cdots & w_{nn} \end{bmatrix}$$

is therefore symmetric.

## Selection and the fundamental theorem

- ▶ Let  $N$  be the number of zygotes in the new generation.
- ▶  $p_i p_j N$  of them carry the gene pair  $A_i, A_j$ .
- ▶  $w_{ij} p_i p_j N$  of them survive to adulthood.
- ▶ The total number of individuals reaching the mating stage is

$$\sum_{r,s=1}^n w_{rs} p_r p_s N \neq 0.$$

(assumption)



# Selection and the fundamental theorem

## Assumptions:

- ▶ Separate generations;
- ▶ Very large population size;
- ▶ Random union of gametes<sup>2</sup>
- ▶ Selection acting solely upon one chromosomal locus, through different survival probabilities. No other effects like mutation, migration, etc.

---

<sup>2</sup>specialized germ cell that fuses with another gamete during fertilization (conception) in organisms that reproduce sexually

## Selection and the fundamental theorem

► Denote by

- $p'_{ij}$  the frequency of the pair  $(A_i, A_j)$  ;
- $p'_i$  - the frequency of allele  $A_i$

in the adult stage of the new generation.

- Then the following equations describe the evolution of gene frequencies from one generation to the next:

$$p'_{ij} = \frac{w_{ij} p_i p_j N}{\sum_{r,s=1}^n w_{rs} p_r p_s N} = p'_{ji} \quad (2)$$

$$p'_i = \frac{1}{2} \sum_j p'_{ij} + \frac{1}{2} \sum_j p'_{ji} = p_i \frac{\sum_j w_{ij} p_j}{\sum_{r,s} w_{rs} p_r p_s}. \quad (3)$$

## Selection and the fundamental theorem

- ▶ The state of the gene pool of the parental generation is given by the vector  $p = (p_1, \dots, p_n)$  of gene frequencies.
- ▶  $p' = (p'_1, \dots, p'_n)$  - the state of the next generation.
- ▶ Denote by

$$S_n = \left\{ x \in \mathbb{R}^n : \sum_{i=1}^n x_i = 1, x_i \geq 0, i = 1, \dots, n \right\}$$

- ▶ Then (3) describes a dynamical system on  $S_n$ .

## Selection and the fundamental theorem

- ▶ Orbit:

$$(p_1, \dots, p_n) \rightarrow (p'_1, \dots, p'_n) \rightarrow \dots \rightarrow (p_1^{(k)}, \dots, p_n^{(k)}) \rightarrow$$

( $k$  is the number of time intervals elapsed)

- ▶ Recall that  $w_{ij}$  is the selective value and  $p_i p_j$  is the frequency for pair  $(A_i, A_j)$ .
- ▶  $\bar{\omega}(p) = p^T W p$  - the average fitness (average selective value) of the population.
- ▶ What happens on a long run?

## Selection and the fundamental theorem

- ▶ **The fundamental theorem of natural selection (FTNS):**  
*The average fitness of population increases from generation to generation.*
- ▶ From (3) we have

$$p'_i = p_i \frac{\sum_j w_{ij} p_j}{\sum_{r,s} w_{rs} p_r p_s} = p_i \frac{(Wp)_i}{p^T Wp}. \quad (4)$$

- ▶  $\bar{\omega}(p) = p^T Wp$  increases along every orbit, i.e.

$$\bar{\omega}(p') \geq \bar{\omega}(p).$$

# Motzkin-Strauss theorem (1965)

- ▶ Given a graph  $G = (V, E)$ , let  $A_G = (a_{ij})_{n \times n}$  be the adjacency matrix of  $G$ .
- ▶ Consider the indefinite quadratic programming problem

$$\begin{aligned} \max \quad & f_G(x) = \sum_{(i,j) \in E} x_i x_j = \frac{1}{2} x^T A_G x \\ \text{s.t.} \quad & x \in S = \{x = (x_1, \dots, x_n)^T : \sum_{i=1}^n x_i = 1, \\ & x_i \geq 0 \quad (i = 1, \dots, n)\}. \end{aligned}$$

- ▶ **Proposition.** If  $\alpha = \max\{f_G(x) : x \in S\}$ , then  $G$  has a maximum clique  $C$  of size  $k = 1/(1 - 2\alpha)$ . This maximum can be attained by setting  $x_i = 1/k$  if  $i \in C$  and  $x_i = 0$  if  $i \notin C$ .

# Regularized Motzkin-Straus QP

- Consider the following regularization of the Motzkin-Straus quadratic program (QP):

$$\begin{array}{ll}\text{maximize} & f(x) = x^T (A_G + \frac{1}{2}I_n) x \\ \text{subject to} & e^T x = 1, \\ & x \geq 0,\end{array}$$

where  $A_G$  is the adjacency matrix of some graph  $G = (V, E)$  with  $n$  vertices,  $e = [1, \dots, 1]^T \in \mathbb{R}^n$  is the unit vector, and  $I_n$  is the  $n \times n$  identity matrix.

# Regularized Motzkin-Straus QP

- Denote by

$$S = \{x \in \mathbb{R}^n : e^T x = 1, x \geq 0\}$$



the standard simplex in  $\mathbb{R}^n$ , which is the feasible region of the above QP. For a subset  $C \subseteq V$  of vertices, a vector  $x^C \in S$  is called the characteristic vector of  $C$  if

$$x_i^C = \begin{cases} 1/|C|, & \text{if } i \in C \\ 0, & \text{otherwise.} \end{cases}$$

- Bomze (1997) has shown that  $x^* \in S$  is a local maximizer of  $f(x)$  over  $S$  if and only if  $x^*$  is a characteristic vector of a maximal clique in  $G$ . Therefore, the problem of finding a maximal (maximum) clique of  $G$  is reduced to the problem of finding a local (global) maximizer of the above QP.



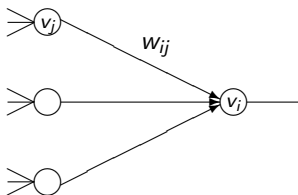
# References

-  I. M. Bomze. Evolution towards the maximum clique. *Journal of Global Optimization*, 10: 143-164, 1997.
-  I. M. Bomze, M. Budinich, M. Pelillo, and C. Rossi. Annealed replication: a new heuristic for the maximum clique problem. *Discrete Applied Mathematics*, 121: pp. 27 - 49, 2002.

# Artificial Neural Networks

# Neurons in a human brain

- ▶ The number of neurons in a human brain is  $\sim 10^{12}$ .
- ▶ Neurons are connected with each other through synapses.
- ▶ Each neuron is connected with 1 to  $10^5$  other neurons.
- ▶ Assuming that an average number of connections is  $10^3$  per neuron, we have  $\sim 10^{15}$  synapses.
- ▶ Considering only two states of synapses (excitatory and inhibitory) we obtain  $\sim 2^{10^{15}}$  states of the brain!



- ▶ Input for  $v_i$ :  $\sum_j w_{ij} v_j$ , where  $w_{ij}$  are weights (synapses).
  - ▶  $w_{ij}$  are nonzero only for neurons  $v_j$  that feed to  $v_i$ .
- ▶ Output for  $v_i$  (local updating rule):

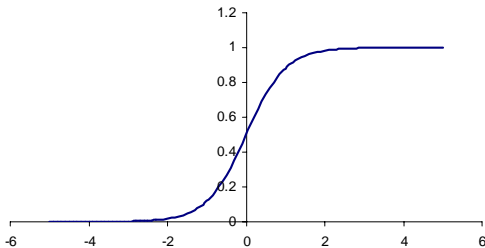
$$v_i = g\left(\sum_j w_{ij} v_j - \theta_i\right),$$

where  $\theta_i$  is a threshold (“membrane potential”)

- ▶ The nonlinear transfer function  $g : \mathbb{R} \rightarrow [0, 1]$  is typically of the form:

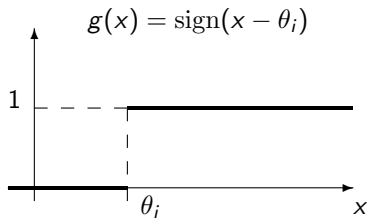
$$g(x) = \frac{1}{2}(\tanh(x/c) + 1),$$

where  $\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$ .



$c \rightarrow 0 : g(x) \rightarrow \text{sign}(x)$  [discrete case].

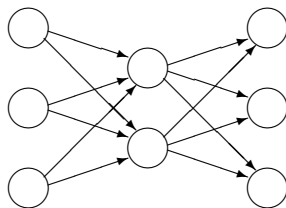
- Discrete case:  $v_j \in \{0, 1\}$  ( $v_j \in \{-1, 1\}$ ).



# Artificial neural networks

- ▶ The artificial neuron imitates the main features of real neurons: linear additivity for input and strong nonlinearity for the resulting output.
- ▶ If the integrated input signal exceeds a certain threshold  $\theta_i$ , the neuron will fire.
- ▶ There are two major types of architecture in neural network modeling:
  - ▶ Feedforward networks;
  - ▶ Feedback networks.

# Feedforward networks



- ▶ “Levels” of neurons with connections only between the levels and only in one direction.
- ▶ Feedforward neural networks are often used for classification and function approximation.
- ▶ The process of neural network training is used.



## Classification with feedforward networks

- ▶ In classification (feature recognition) the input patterns  $x \in \mathbb{R}^{N_i}$  need to be categorized in terms of different features  $o_1, \dots, o_{N_f}$ .
- ▶ An input pattern  $x = [x_1, \dots, x_{N_i}]$  is fed into the input layer (receptors), while the output nodes represent the features.
- ▶ The transfer function:

$$o_i = \frac{1}{2} \left( \tanh\left(\sum_j w_{ij} x_j - \theta_i\right) + 1 \right).$$

# Classification with feedforward networks

- ▶ The process of training is used to determine optimal values for  $w_{ij}$  as follows.
- ▶ Assume that we have historical data for inputs and outputs.
- ▶ We define an error function

$$Error(w) = \sum_{i=1}^n \left( g\left(\sum_j w_{ij} \bar{x}_j - \theta_i\right) - \bar{v}_i \right)^2.$$

- ▶ We use nonlinear optimization techniques to find the optimal values of  $w_{ij}$ .
- ▶ The transfer function with the obtained values of  $w_{ij}$  is then used to classify the future data.

# Feedback networks

- ▶ In feedback neural networks, the pattern of connections is arbitrary.
- ▶ The activation in feedback network continues until a steady state has been reached.
- ▶ Feedback networks are used in combinatorial optimization.
- ▶ Simple models for magnetic systems have many commonalities with feedback networks, so they have provided a source of ideas for research in artificial neural networks.

# Magnetic systems

- ▶ Consider a magnetic system of  $N$  atoms positioned on a lattice.
- ▶ The Ising model describes this system in terms of a set of binary spins  $s_i \in \{-1, 1\}$ ,  $i = 1, \dots, N$ , which are effective variables for the individual atoms.
- ▶ The two spin states represent the possible magnetization directions.

# Magnetic systems

- ▶ The model is governed by an energy function given by

$$E(s) = -\frac{J}{2} \sum_{i,j \in N(i)} s_i s_j,$$

where  $N(j)$  denotes the set of sites neighboring with  $j$ .

- ▶ The neighboring spins interact via a constant attractive coupling of strength  $J > 0$ .

# Magnetic systems

- ▶ The lowest energy state is reached by iterative updating of spins according to

$$s_i = \text{sign} \left( J \sum_{j \in N(i)} s_j \right).$$

- ▶ Eventually, a state is reached where all spins point in one of the two possible directions.

# Magnetic systems

- ▶ An interesting generalization of the Ising model is a *spin glass* system, which allows for:
  - ▶ nonlocal interactions (can have  $s_i s_j$  terms in  $E$  for all  $i \neq j$ );
  - ▶ symmetric pair-dependent couplings  $w_{ij} = w_{ji}$  between  $s_i$  and  $s_j$ .
- ▶ We obtain the energy function in the form

$$E(s) = -\frac{1}{2} \sum_{i,j=1}^n w_{ij} s_i s_j.$$

# The Hopfield model

- ▶ A dynamics that locally minimizes the energy function

$$E(s) = -\frac{1}{2} \sum_{i,j=1}^n w_{ij} s_i s_j.$$

over  $s_i \in \{-1, 1\}$  is given by

$$s_i = \text{sign} \left( \sum_{j \neq i} w_{ij} s_j \right).$$



# The Hopfield model

- ▶ Given a set of  $N_p$  patterns  $s^{(p)} = [s_1^{(p)}, \dots, s_N^{(p)}]$ ,  $p = 1, \dots, N_p$ , with  $s_i^{(p)} \in \{-1, 1\}$ , the Hebb rule

$$w_{ij} = \frac{1}{N_p} \sum_{p=1}^{N_p} s_i^{(p)} s_j^{(p)}$$

is used for learning.

- ▶ With this choice of  $w_{ij}$  it can be shown that under certain conditions and with certain value  $s_i^{(0)}$ , the updating rule used yields a local minimum of  $E$ .

# ANN for combinatorial optimization

Two families of ANN algorithms for optimization:

- ▶ The 'pure' neural approach is based on a system of either binary or multistate neurons with mean field equations for the dynamics.
- ▶ Hybrid approaches supplement the neural variables with problem-specific variables.

# Pure ANN Approach

- ▶ We minimize an energy function

$$E(x) = -\frac{1}{2} \sum_{i,j=1}^n w_{ij} x_i x_j = -\frac{1}{2} x^T W x,$$

where  $W = [w_{ij}]_{i,j=1}^n$ ,  $x_i \in [-1, 1]$  ( $x_i \in \{-1, 1\}$ ).

- ▶ Transfer equations (*Mean Field Equations*):

$$x_i = \tanh \left( \sum_{j \neq i}^n w_{ij} x_j / T \right), i = 1, \dots, n.$$

- ▶ Here  $T$  is the temperature parameter.
- ▶ Under the transformation defined by Mean Field Equations, the energy function is decreasing.

# NN Algorithms

- ▶ To solve a combinatorial optimization problem using the neural network algorithm, we need to encode the objective function as an energy function.
- ▶ A penalty for constraint violation may be added in constrained problems.
- ▶ Energy function needs to be in the form of a binary quadratic ( $x_i \in \{-1, 1\}$ ).

## Example: graph bisection problem

- ▶ Given a graph  $G = (V, E)$ ,  $|V| = 2n$ , partition the vertices into two equal parts so that the number of edges between parts is minimized.
- ▶ Let  $A_G$  be the adjacency matrix of  $G$ .
- ▶ A feasible solution  $(V_1, V_2)$ , where  $V = V_1 \cup V_2$ ,  $|V_1| = |V_2| = n$  can be encoded as a binary  $n$ -vector  $x$  in which  $x_i = -1$  if  $i \in V_1$  and  $x_i = 1$  if  $i \in V_2$ .
- ▶ Then the objective can be written as

$$f(x) = -\frac{1}{2}x^T A_G x = -\frac{1}{2} \sum_{i,j=1}^{2n} a_{ij} x_i x_j.$$

## Example: graph bisection problem

- ▶ We need to make sure that  $\sum_{i=1}^{2n} x_i = 0$ .
- ▶ This could be done by adding  $\left(\sum_{i=1}^{2n} x_i\right)^2$  as penalty, however, this would create a nonzero diagonal in the quadratic's matrix.
- ▶ Thus, we will subtract  $\sum_{i=1}^{2n} x_i^2 = 2n$ , which is a constant that does not change the optimization problem.
- ▶ So, the total penalty is  $\left(\sum_{i=1}^{2n} x_i\right)^2 - \sum_{i=1}^{2n} x_i^2$ .

## Example: graph bisection problem

- ▶ The final energy function is

$$E(x) = -\frac{1}{2}x^T A_G x + \frac{1}{2}\alpha \left( \left( \sum_{i=1}^{2n} x_i \right)^2 - \sum_{i=1}^{2n} x_i^2 \right).$$

- ▶ The transfer equations:

$$x_i = \tanh \left( \frac{1}{T} \sum_{i \neq j=1}^{2n} (a_{ij} - 2\alpha)x_j \right), i = 1, \dots, 2n.$$

## Example: TSP

- ▶ We introduce the following decision variables:

$$x_{ij} = \begin{cases} 1, & \text{if city } i \text{ was visited in } j\text{-th order,} \\ 0, & \text{otherwise.} \end{cases}$$

- ▶ Objective:  $\sum_{i=1}^n \sum_{j=1}^n \sum_{k=1}^n d_{ik} x_{ij} x_{k(j+1)}.$
- ▶ Finding a feasible solution using NN approach appears to be challenging even for small instances.



## Example: maximum clique

- ▶ For a graph  $G = (V, E)$ , let  $\bar{G}$  be its complement with the adjacency matrix  $A_{\bar{G}}$ . Then the maximum clique problem can be formulated as follows:

$$\omega(G) = \max_{x \in [0,1]^n} e^T x - x^T A_{\bar{G}} x,$$

where  $e$  is the  $n$ -vector of all ones.

- ▶ Equivalently,

$$\omega(G) = \max_{x \in \{0,1\}^n} e^T x - x^T A_{\bar{G}} x,$$

# Ant Colony Optimization (ACO)

# Swarm intelligence

- ▶ Swarm intelligence is an approach to problem solving inspired by the social behavior of insects/animals.
- ▶ The ant colony optimization (ACO) attempts to imitate the foraging behavior of some ant species.
  - ▶ Ants deposit pheromone on the ground in order to mark some favorable path for other members of the colony to follow.
  - ▶ Similar mechanisms can be exploited for solving optimization problems.

# Biological Inspiration

- ▶ In 1940's the French entomologist Pierre-Paul Grassé observed that some species of termites react to what he called “significant stimuli”.
- ▶ He used the term *stigmergy* to describe the type of communication in which the “workers are stimulated by the performance they have achieved”.
  - ▶ Insects exchange information by modifying their environment;
  - ▶ Stigmergic information is local: it can only be accessed by those insects that visit the locus in which it was released (or its immediate neighborhood).

# Stigmergy in ant colonies

- ▶ In many ant species, ants walking to and from a food source deposit on the ground a substance called pheromone.
- ▶ Other ants perceive the presence of pheromone and tend to follow paths where pheromone concentration is higher.
- ▶ Through this mechanism, ants are able to transport food to their nest in a remarkably effective way.

# Double bridge experiment

Deneubourg et al (1990):

- ▶ The nest of a colony of Argentine ants was connected to a food source by two bridges of equal lengths.
- ▶ Ants start to explore the surroundings of the nest and eventually reach the food source.
- ▶ Along their path between food source and nest, Argentine ants deposit pheromone.
- ▶ Initially, each ant randomly chooses one of the two bridges. Due to random fluctuations, after some time one of the two bridges presents a higher concentration of pheromone than the other and attracts more ants.
- ▶ This eventually results in the whole colony converging toward the use of the same bridge.

# Double bridge experiment

Goss et al. (1989):

- ▶ One bridge is significantly longer than the other.
- ▶ The stochastic fluctuations in the initial choice of a bridge are much reduced: the ants choosing by chance the short bridge are the first to reach the nest.
- ▶ The short bridge receives, therefore, pheromone earlier than the long one and this fact increases the probability that further ants select it rather than the long one.
- ▶ This phenomenon can be modeled as follows...

# Double bridge experiment

- ▶ Assuming that at a given time moment  $m_1$  ants have used the first bridge and  $m_2$  - the second, the probability  $p_1$  of an ant choosing the first bridge is

$$p_1 = \frac{(m_1 + k)^h}{(m_1 + k)^h + (m_2 + k)^h},$$

where parameters  $k$  and  $h$  are fitted to the experimental data.

- ▶ Monte Carlo simulations showed a very good fit for  $k \approx 20$  and  $h \approx 2$ .



# The ACO technique

- ▶ Based on the Goss' model.
- ▶ In ACO, a number of artificial ants build solutions to an optimization problem.
- ▶ They exchange information on the quality of these solutions via a scheme imitating the one adopted by real ants.
- ▶ The first ACO algorithm known as Ant System was proposed by Dorigo et al. (1991).

# The ACO Metaheuristic

- 
- ▶ Set parameters, initialize pheromone trails
  - ▶ **while** stopping criterion not met **do**
    - ▶ construct ant solutions
    - ▶ apply local search (optional)
    - ▶ update pheromone values
  - ▶ **end while**
-

# Constructing ant solutions

- ▶  $m$  artificial ants construct solutions from elements of a finite set of available solution components  $C$ .
- ▶ A solution construction starts from an empty partial solution  $s_p = \emptyset$ .
- ▶ At each construction step, the partial solution  $s_p$  is extended by adding a feasible solution component from the set  $N(s_p) \subseteq C$ , which is defined as the set of components that can be added to the current partial solution  $s_p$  without violating any of the constraints.

# Constructing ant solutions

- ▶ The choice of a solution component from  $N(s_p)$  is guided by a stochastic mechanism, which is biased by the pheromone associated with each of the elements of  $N(s_p)$ .
- ▶ The rule for the stochastic choice of solution components vary across different ACO algorithms but, in all of them, it is inspired by the model of the behavior of real ants as modeled by Goss et al.

# Local search

- ▶ Once solutions have been constructed, and before updating the pheromone, it is common to improve the solutions obtained by the ants through a local search.
- ▶ This phase, which is highly problem-specific, is optional although it is usually included in state-of-the-art ACO algorithms.

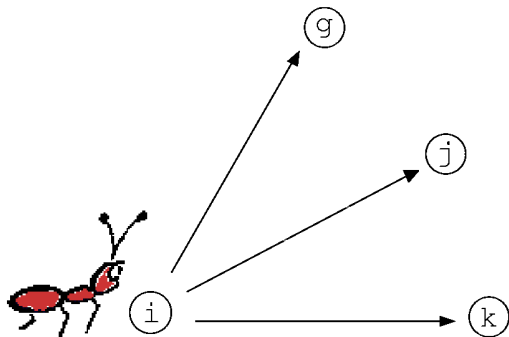
# Updating pheromone values

- ▶ The aim of the pheromone update is to increase the pheromone values associated with good or promising solutions, and to decrease those that are associated with bad ones.
- ▶ Usually, this is achieved
  - (i) by decreasing all the pheromone values through pheromone evaporation;
  - (ii) by increasing the pheromone levels associated with a chosen set of good solutions.

# ACO for TSP

- ▶ A number of artificial ants move on the graph representing the TSP instance.
- ▶ A variable called pheromone is associated with each edge and can be read and modified by ants.
- ▶ At each iteration, a number of artificial ants are considered.
- ▶ Each of them builds a solution by walking from vertex to vertex on the graph with the constraint of not visiting any vertex that has already been visited.

## ACO for TSP



- If  $j$  has not been previously visited, it is selected with probability proportional to the pheromone associated with edge  $(i, j)$ .



# ACO for TSP

- ▶ When ant  $k$  is in city  $i$  and has so far constructed a partial solution  $s_p$ , the probability of going to city  $j$  is given by

$$p_{ij}^k = \begin{cases} \frac{\tau_{ij}^\alpha \eta_{ij}^\beta}{\sum_{(i,l) \in N(s_p)} \tau_{il}^\alpha \eta_{il}^\beta}, & \text{if } c_{ij} \in N(s_p), \\ 0, & \text{otherwise.} \end{cases}$$

- ▶ The parameters  $\alpha$  and  $\beta$  control the relative importance of the pheromone  $\tau_{ij}$  versus a heuristic parameter  $\eta_{ij} = 1/d_{ij}$ .

# ACO for TSP

- ▶ At the end of an iteration, the pheromone values are modified based of the quality of the solutions constructed by the ants.
- ▶ This is done in order to bias ants in future iterations to construct solutions similar to the best ones previously constructed.
- ▶ In Ant System (the original ACO algorithm), at each iteration, the pheromone values are updated by all the  $m$  ants that have built a solution in this iteration.

# ACO for TSP

- Denoting by  $\tau_{ij}$  the pheromone associated with the edge  $(i, j)$ , the update equations:

$$\begin{aligned}\tau_{ij} &= (1 - \rho)\tau_{ij} + \sum_{k=1}^m \Delta\tau_{ij}^k, \\ \Delta\tau_{ij}^k &= \begin{cases} Q/L_k, & \text{if ant } k \text{ used edge } (i, j), \\ 0, & \text{otherwise,} \end{cases}\end{aligned}$$

where  $\rho$  is the evaporation rate and  $\Delta\tau_{ij}^k$  is the quantity of pheromone laid on edge  $(i, j)$  by ant  $k$ ,  $Q$  is a constant, and  $L_k$  is the length of the tour constructed by ant  $k$ .

# ACO References



The official website of the ACO metaheuristic:  
<http://www.aco-metaheuristic.org/>.

# Global Equilibrium Search (GES)

# Boltzmann distribution

- ▶ Consider the problem

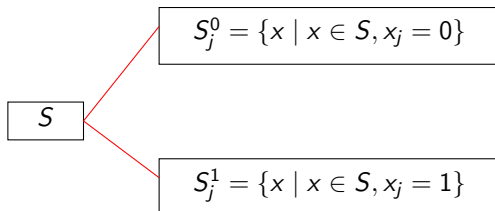
$$\min\{f(x) : x \in S : S \subseteq \{0, 1\}^n\}$$

- ▶ Let  $\xi(\mu)$ , where  $\mu$  is a temperature parameter ( $\geq 0$ ), be a random vector with the following distribution:

$$P\{\xi(\mu) = x\} = \frac{\exp(-\mu f(x))}{\sum_{x \in S} \exp(-\mu f(x))}$$

- ▶ This is the stationary distribution of the Markov chain associated with SA method.

# Boltzmann distribution



$$p_j(\mu) \equiv P\{\xi_j = 1\} = \frac{\sum_{x \in S_j^1} \exp(-\mu f(x))}{\sum_{x \in S} \exp(-\mu f(x))}$$

# Global Equilibrium Search

- ▶ Instead of  $S$  use a set  $S'$  of known solutions.
- ▶ Generate new solutions randomly using the following probability distribution:

$$\tilde{p}_j(\mu) = \frac{\sum_{x \in S'_j} \exp(-\mu f(x))}{\sum_{x \in S'} \exp(-\mu f(x))}$$

- ▶ If the sample  $S'$  is not very large, there is a chance of zero probabilities.
- ▶ We will approximate probabilities by solving a differential equation relating  $p_j$  to  $\mu$ ...



# Global Equilibrium Search

- Denote by

$$Z_j^u(\mu) = \sum_{x \in S_j^u} \exp(-\mu f(x)), \quad u \in \{0, 1\};$$

$$Z(\mu) = \sum_{x \in S} \exp(-\mu f(x)).$$

Then

$$p_j(\mu) = Z_j^1(\mu)/Z(\mu), \quad j = 1, \dots, n.$$

# Global Equilibrium Search

- Consider the derivative

$$\begin{aligned}\frac{\partial p_j}{\partial \mu} &= \frac{\partial}{\partial \mu} \frac{Z_j^1(\mu)}{Z(\mu)} = \frac{Z(\mu) \frac{\partial Z_j^1(\mu)}{\partial \mu} - Z_j^1(\mu) \frac{\partial Z(\mu)}{\partial \mu}}{Z(\mu)^2} \\ &= p_j(\mu)(\langle f \rangle_\mu - \langle f \rangle_\mu^{x_j=1}),\end{aligned}$$

where

$$\langle f \rangle_\mu^{x_j=u} = \frac{\sum_{x \in S_j^u} f(x) \exp(-\mu f(x))}{\sum_{x \in S_j^u} \exp(-\mu f(x))}, \quad u \in \{0, 1\}$$

# Global Equilibrium Search

- Note that

$$\langle f \rangle_\mu = p_j(\mu) \langle f \rangle_\mu^{x_j=1} + (1 - p_j(\mu)) \langle f \rangle_\mu^{x_j=0}, \quad j = 1, \dots, n,$$

so

$$\frac{\partial p_j}{\partial \mu} = p_j(\mu)(1 - p_j(\mu))(\langle f \rangle_\mu^{x_j=0} - \langle f \rangle_\mu^{x_j=1}).$$

- Solution of this equation is given by:

$$p_j(\mu) = \frac{1}{1 + \frac{1-p_j(0)}{p_j(0)} \exp(-\int_0^\mu (\langle f \rangle_t^{x_j=0} - \langle f \rangle_t^{x_j=1}) dt)}.$$

# Global Equilibrium Search

By the trapezoid rule:

$$\tilde{\rho}_j(\mu_i) = \frac{1}{1 + \frac{1 - \tilde{\rho}_j(0)}{\tilde{\rho}_j(0)} \exp(-\frac{1}{2} \sum_{k=0}^{i-1} (E_{kj}^0 - E_{kj}^1 + E_{k+1j}^0 - E_{k+1j}^1))}$$

where

$$E_{ij}^u = \frac{\sum_{x \in S_j'^u} f(x) \exp(-\mu_i f(x))}{\sum_{x \in S_j'^u} \exp(-\mu_i f(x))}, \quad u \in \{0, 1\}$$

# Global Equilibrium Search

- ▶ Define # of temperature stages  $K$
- ▶ Define temperatures  $\mu_1, \mu_2, \dots, \mu_K$
- ▶ Matrix for storing  $E_{ij}^0$  and  $E_{ij}^1$

# Generation procedure

## Solution

-	-	-	-	-	-	-	-
---	---	---	---	---	---	---	---

- ▶ Generation cycle...
  - ▶ Generate solution using  $\tilde{p}_j(\mu)$
  - ▶ Local search
  - ▶ Update  $E_{ij}^0$  and  $E_{ij}^1$
  - ▶ Generate...
  - ▶ Local search
  - ▶ Update...
- ▶ Change the temperature  $\mu$
- ▶ Update probabilities

# Generation procedure

## Solution

1	0	1	0	1	0	1	0
---	---	---	---	---	---	---	---

- ▶ Generation cycle...
  - ▶ Generate solution using  $\tilde{p}_j(\mu)$
  - ▶ Local search
  - ▶ Update  $E_{ij}^0$  and  $E_{ij}^1$
  - ▶ Generate...
  - ▶ Local search
  - ▶ Update...
- ▶ Change the temperature  $\mu$
- ▶ Update probabilities

# Generation procedure

## Solution

1	0	1	0	1	0	1	0
---	---	---	---	---	---	---	---

- ▶ Generation cycle...
  - ▶ Generate solution using  $\tilde{p}_j(\mu)$
  - ▶ Local search
  - ▶ Update  $E_{ij}^0$  and  $E_{ij}^1$
  - ▶ Generate...
  - ▶ Local search
  - ▶ Update...
- ▶ Change the temperature  $\mu$
- ▶ Update probabilities



# Generation procedure

## Solution

1	0	1	0	1	1	0	1
---	---	---	---	---	---	---	---

- ▶ Generation cycle...
  - ▶ Generate solution using  $\tilde{p}_j(\mu)$
  - ▶ Local search
  - ▶ Update  $E_{ij}^0$  and  $E_{ij}^1$
  - ▶ Generate...
  - ▶ Local search
  - ▶ Update...
- ▶ Change the temperature  $\mu$
- ▶ Update probabilities

# Diversification vs Intensification

- ▶ restarting the search
- ▶ what information to save?  
(elite solutions)
- ▶ how to avoid the same areas?  
(tabu rules)

# Applications of GES

Successful implementations have been developed for the following problems:

- ▶ 0-1 Quadratic Programming
- ▶ Weighted MAX-SAT
- ▶ Job Shop Scheduling Problem
- ▶ Knapsack Problem
- ▶ ...



V. Shylo. The Global Equilibrium Search Method, *Cybernetics and Systems Analysis*, 35: 68-74, 1999.

# Variable Neighborhood Search

# Variable Neighborhood Search (VNS)

- ▶ Proposed by Hansen and Mladenovic (1997).
- ▶ Based on a systematic change of neighborhood within the search.
- ▶ Consider an optimization problem in the form

$$\min\{f(x) : x \in X\}.$$

- ▶ Let  $N_k$  denote the  $k$ -th neighborhood structure and  $N_k(x)$  be the set of all neighbors of  $x$  in this neighborhood structure ( $k = 1, \dots, K$ ).

# Variable Neighborhood Search (VNS)

VNS is based on the following three facts:

1. A local minimum with respect to the neighborhood  $N_{k_1}$  may not be a local minimum with respect to another neighborhood  $N_{k_2}$ .
2. A global minimum is a local minimum with respect to any neighborhood structure.
3. For many problems local minima with respect to one or several neighborhood structures are relatively close to each other.

# Variable Neighborhood Descent (VND)

Initialization. Select the set of neighborhood structures  $N'_k$ , for  $k = 1, \dots, K'$ , that will be used in the descent. Find an initial solution  $x$ .

**Repeat until** no improvement is obtained:

- (1) Set  $k \leftarrow 1$ ;
- (2) **Repeat until**  $k = K'$ :
  - (a) Find the best neighbor  $x' \in N'_k(x)$  of  $x$ ;
  - (b) **If**  $f(x') < f(x)$  **set**  $x \leftarrow x'$  and  $k \leftarrow 1$ ;  
**else set**  $k \leftarrow k + 1$ ;

- changes the neighborhood deterministically.

## Variable Neighborhood Descent (VND)

- ▶ The output solution is a local minimum with respect to each  $N_k, k = 1, \dots, K'$ .
- ▶ In addition to the above *sequential* order of neighborhood structures, one can develop a *nested* strategy
- ▶ Example:  $K' = 3$ ; then a possible nested strategy is to perform VND for the first two neighborhoods in each point  $x'$  that belongs to the third ( $x' \in N_3(x)$ ).



# Reduced VNS (RVNS)

Initialization. Select the set of neighborhood structures  $N'_k$ , for  $k = 1, \dots, K'$ , that will be used in the search. Find an initial solution  $x$ . Choose a stopping condition.

**Repeat until** the stopping condition is met:

- (1) Set  $k \leftarrow 1$ ;
- (2) **Repeat until**  $k = K'$ :
  - (a) Generate a point  $x' \in N_k(x)$  at random (“shaking”);
  - (b) **If**  $f(x') < f(x)$  set  $x \leftarrow x'$  and  $k \leftarrow 1$ ;  
    **else** set  $k \leftarrow k + 1$ ;

# Reduced VNS (RVNS)

- ▶ RVNS is useful for very large instances for which local search is expensive.
- ▶ Observation:  $K' = 2$  works well in many cases.
- ▶ The maximum number of iterations between two improvements is usually used as stopping condition.
- ▶ RVNS is similar to a Monte Carlo method, but is more systematic.

Initialization. Select the set of neighborhood structures  $N'_k$ , for  $k = 1, \dots, K'$ , that will be used in the search. Find an initial solution  $x$ . Choose a stopping condition.

**Repeat until** the stopping condition is met:

- (1) Set  $k \leftarrow 1$ ;
- (2) **Repeat until**  $k = K'$ :
  - (a) Generate a point  $x' \in N_k(x)$  at random (“shaking”);
  - (b) Apply some local search starting from  $x'$  to obtain a local optimum  $x''$ ;
  - (c) **If**  $f(x'') < f(x)$  **set**  $x \leftarrow x''$  **and**  $k \leftarrow 1$ ;  
**else set**  $k \leftarrow k + 1$ ;

# Basic VNS

- ▶ Stopping conditions: maximum CPU time allowed, maximum number of iterations, maximum number of iterations between two improvements.
- ▶ Point  $x'$  is generated at random to avoid cycling, which might occur if a deterministic rule was used.
- ▶ The local search step may be replaced by VND – some of the most successful implementations use this strategy.

## Some modifications of basic VNS

- ▶ The basic VNS is a descent first improvement method with randomization.
- ▶ To obtain a descent-ascent method: allow moving to a worse solution (with some probability) in Step 2c.
- ▶ To obtain a best improvement method: make a move to the best neighborhood  $k^*$ .
- ▶ Find solution  $x'$  at Step 2a as the best among  $b$  generated solutions from the  $k$ -th neighborhood.

## Var. Neigh. Decomposition Search (VNDS)

Initialization. Select the set of neighborhood structures  $N'_k$ , for  $k = 1, \dots, K'$ , that will be used in the search. Find an initial solution  $x$ . Choose a stopping condition.

**Repeat until** the stopping condition is met:

- (1) Set  $k \leftarrow 1$ ;
- (2) **Repeat until**  $k = K'$ :
  - (a) Generate a point  $x' \in N_k(x)$  at random (“shaking”);
  - (b) Apply some local search with respect to attributes of  $x'$  that are not present in  $x$  to obtain  $x''$ ;
  - (c) **If**  $f(x'') < f(x)$  set  $x \leftarrow x''$  and  $k \leftarrow 1$ ;  
**else** set  $k \leftarrow k + 1$ ;

# The Skewed VNS (SVNS)

Initialization. Select the set of neighborhood structures  $N'_k$ , for  $k = 1, \dots, K'$ . Find an initial solution  $x$ , set  $x_{opt} \leftarrow x$ ,  $f_{opt} \leftarrow f(x)$ . Choose a stopping condition and a parameter value  $\alpha$ .

**Repeat until** the stopping condition is met:

- (1) Set  $k \leftarrow 1$ ;
- (2) **Repeat until**  $k = K'$ :
  - (a) Generate a point  $x' \in N_k(x)$  at random ("shaking");
  - (b) Apply some local search to obtain a local optimum  $x''$ ;
  - (c) **If**  $f(x'') < f_{opt}$  **set**  $x_{opt} \leftarrow x''$  and  $f_{opt} \leftarrow f(x'')$ ;
  - (d) **If**  $f(x'') - \alpha \rho(x, x'') < f(x)$  **set**  $x \leftarrow x''$  and  $k \leftarrow 1$ ;  
**else set**  $k \leftarrow k + 1$ ;

# The Skewed VNS (SVNS)

- ▶ SVNS makes use of a function  $\rho(x, x'')$  to measure distance between  $x$  and  $x''$ .
- ▶ The parameter  $\alpha$  must be chosen in order to accept exploring regions far from  $x$  when  $f(x'')$  is larger than  $f(x)$ , but not too much larger (otherwise one will always leave  $x$ ).
- ▶ The parameter values are chosen experimentally.