A Parallel Tabu Search Alglorithm for the Quadratic Assignment Problem

Samuel Gabrielsson

September 2007

Abstract

A parallel version of the tabu search algorithm is implemented and used to optimize the solutions for a quadratic assignment problem (QAP). The instances are taken from the qaplib website[1] and we mainly concentrate on solving and optimizing the instances announced by Sergio Carvalho derived from the "Microarray Placement Problem"[2] where one wants to find an arrangement of the probes (small DNA fragments) on specific locations of a microarray chip.

We briefly explain combinatorics including graph theory and also the theory behind combinatorial optimization, heuristics and metaheuristcs. A description of some network optimization problems are also introduced before we apply our parallel tabu search algorithm to the quadratic assignment problem.

Different approaches like Boltzmann selection procedure and random restarts are used to optimize the solutions. Through our experiments, we show that our parallel version of tabu Search do indeed manage to further optimize and even find better solutions found so far in the litterature.

We try out a communication protocol based on sequentially generating graphs, where each node in the graph corresponds to a CPU or tabu search thread. One of the main goals is to find out if communication helps to further optimize the best known solution found so far for each instace.

[1]http://www.opt.math.tu-graz.ac.at/qaplib/
[2]http://gi.cebitec.uni-bielefeld.de/comet/chiplayout/qap

Acknowledgements

This thesis is the final part of the Master of Science programme in Computer Science and Engineering. It has been carried out during the spring semester of 2007 in the Toronto Intelligent Decision Engineering Lab (TIDEL), at the University of Toronto (UofT), Ontario, Canada.

I was one of the lucky few students from abroad to work for Prof. J. Christopher Beck in the department of Mechanical and Industrial Engineering at UofT as a research trainee. It was truly an honor and I most definitely had lots of fun learning and working with the different projects. All thanks to Prof Beck and the IAESTE organization in Luleå, Sweden for giving me that chance and exposure to the wonderful world of research.

I would also like to thank the people in TIDEL, especially Lei Duan for all his hard work, and Ivan Heckman for helping out on making the fundy cluster behave nicely. Finally, I would like to thank my thesis supervisor Inge Söderkvist in Luleå, Sweden for his work on helping me to improve this thesis.

The thesis is meant to be a spin off the parallel tabu search part from a published paper [1] by Lei Duan, the author of this thesis, and professor J. Christopher Beck at the university of Toronto[3].

[3] http://tidel.mie.utoronto.ca/publications.php

Contents

1 Introduction **3**

2 Combinatorics **5**
2.1 The Rule of Product . . . 5
2.2 Permutations - When Order Matters . . . 6
2.2.1 Permutations Without Repetition . . . 6
2.2.2 Permutations With Repetition . . . 7
2.3 Combinations - When Order do not Matter . . . 7
2.3.1 Combinations Without Repetition . . . 7
2.3.2 Combinations With Repetition . . . 8
2.4 Graph Theory . . . 8
2.4.1 Introducing Graphs . . . 9
2.4.2 Directed Graphs and Undirected Graphs . . . 9
2.4.3 Adjacency and Incidence . . . 10
2.4.4 Paths and Cycles . . . 11
2.4.5 Subgraphs . . . 13
2.4.6 Vertex Degrees . . . 15
2.4.7 Graph Representation . . . 16
2.4.8 Graph Applications . . . 17

3 Combinatorial Optimization **19**
3.1 Combinatorial Optimization Problems . . . 20
3.1.1 The Traveling Salesman Problem . . . 21
3.2 Computational Complexity . . . 23
3.2.1 The Class $\mathcal{P}$ . . . 24
3.2.2 The Class $\mathcal{NP}$ . . . 24
3.2.3 The Classes $\mathcal{NP}$-Hard and $\mathcal{NP}$-Complete . . . 25
3.3 Heuristics . . . 27
3.4 Heuristic Methods . . . 27
3.4.1 Local Search . . . 27
3.4.2 Hill Climbing . . . 30
3.4.3 Local Improvement Procedures . . . 31
3.5 Metaheuristics . . . 33
3.6 Metaheuristic Methods . . . 34
3.6.1 Greedy Algorithms and Greedy Satisfiability . . . 34
3.6.2 Simulated Annealing . . . 35
3.6.3 Hybrid Evolutionary Algorithm . . . 35

4 Network Optimization Problems **39**
4.1 Network Flow Problem Terminology . . . 39
4.2 The Minimum Cost Network Flow Problem . . . 40
4.3 The Transportation Problem . . . 40
4.4 The Assignment Problem . . . 41
4.5 The Quadratic Assignment Problem . . . 41
4.5.1 Mathematical Formulation of the QAP . . . 42
4.5.2 Location Theory . . . 43
4.5.3 A Quadratic Assignment Problem Library . . . 46

5 Parallel Tabu Search Algorithm **47**
5.1 Tabu Search for Combinatorial Optimization Problems . . . 47
5.1.1 Short Term Memory . . . 49
5.1.2 Long Term Memory . . . 49
5.2 Parallel Tabu Search . . . 50
5.2.1 Communication Procedure . . . 50
5.2.2 Communication Graph Topology . . . 50
5.2.3 Boltzmann Selection Procedure . . . 52

6 Simulation and Results **55**
6.1 Experiment Details . . . 55
6.2 Computing Environment . . . 56
6.3 Experimental Results . . . 56
6.4 Conclusion and Future Work . . . 59

List of Figures

2.1 The seven bridges of Könighsberg with four land areas interconnected by the seven bridges. 9
2.2 The graph representation of the seven bridges problem. 9
2.3 The difference between an undirected and a directed graph is indicated by drawing an arrow to show the direction. 10
2.4 The two edges (e, c) and (c, e) joining the same pair or vertices in Figure 2.4(a) is a graph with multiple edges. Figure 2.4(b) has a loop in node b and Figure 2.4(c) is just a simple graph with no loop or multiple edges. 11
2.5 A graph showing the relationship between adjacency and incidence. . . . 11
2.6 An example of a large graph. 12
2.7 A disconnected graph. 13
2.8 A graph G and one of its subgraphs G_1 with an isolated node d. 13
2.9 The complete graph with n vertices denoted by K_n 14
2.10 The null graph of $1, 2, 3, 4, 5$ and 6 vertices denoted N_n where n is the amount of vertices or nodes. 14
2.11 The graphs show two different ways to represent the weighted graph. . . 16

3.1 A small instance with size 4 of the traveling salesman problem. 22
3.2 The Venn diagram for $\mathcal{P}$, $\mathcal{NP}$, $\mathcal{NP}$-complete, and $\mathcal{NP}$-hard set of problems. 27
3.3 The local search move showing the solution s, its neighborhood $N(s)$, its set $L(N(s), s)$ of legal moves and the selected solution in bold thick circle. 29
3.4 The possible landscape of a search space and problems that may occur for the hill climbing algorithm. The algorithm can get stuck in plateaus or local optimum. 31

4.1 One possible solution to QAP with four facilities. 44

5.1 Communication graphs for four solvers with different densities. 51
5.2 Communication graph for eight solvers with different densities. 51
5.3 Communication graph for twelve solvers with different densities. 52

6.1 Speedup on the mean concurrent iterations of 4, 8, and 12 solvers. . . . 57
6.2 Comparing the mean concurrent iterations across different communication graphs for 12 solvers. The best solution found has an objective value of 168611971. The target ranges are 0%, 0.1%, and 0.25% away from this objective value. 58

List of Tables

2.1 The amount of students in each position. 6
2.2 The amount of n distinct objects in each position. 6

3.1 Values of several functions important for analysis of algorithms. Problems that grow too fast are left empty. 24

4.1 Components of some typical networks in todays modern society. 39

6.1 The best values are shown in bold. These values are even better than those found in the literature. 57

Chapter 1

Introduction

In *parallel computing*, a large number of computers, each with multiple processors, are interconnected so that the individual processors can work simultaneously to solve a smaller part of a complex problem that is too large for a single computer. To take advantage of parallel computing, a program must be written so that execution of the program occurs on more than one process, where a process can represent a single instance of a program or subprogram, executing autonomously on a physical processor. The primary objective in parallel programming is to gain an increase in the computational performance relative to the performance obtained by the serial execution of that same program.

This thesis present parallel cooperative solvers, in which solutions, or partial solutions, are communicated to provide heuristic guidance, i.e, communication is used to influence the search as one solver's search can be guided by anothers solution. The cooperative solvers can be completely independent or fully collaborative. Our hypothesis is that the best performance required a good balance between guidance by an outside solution and searching on one's own.

We experiment with Tabu Search for quadratic assignment problems. Experimental results demonstrate that adding more solvers improves performance and the performance gain relies on how solvers collaborate. Although the speedup and performance of our tabu search did not gain that much from cooperation, the Solution-Guided Multi-Point Consructive Search for quasigroup-with-holes did, as shown in our previous work [1]. The main contribution of this thesis is an initial investigation of using parallel cooperative solvers to solve hard optimization problems.

Chapter 2 gives a basic introductory theory of combinatorics and graphs so the reader can better understand the QAP and the different communication graphs used when communication with other CPUs. In chapter 3, we give an example of a special case of a QAP, i.e, the traveling salesman problem, and explain how common heuristc and metaheuristc algorithms are implemented. Chapter 4 describes different network optimization problems including the quadratic assignment problem and where we have obtained our instances. In chapter 5, we describe and explain our parallel tabu search algorithm with its communication protocol and its usage of the Boltzmann selection procedure to further improve solutions followed by chapter 6, where we apply our algorithm on a QAP called the microarray placement problem.

Chapter 2

Combinatorics

Combinatorics [2], which is a collectively name for the *fundamental principles of counting*, combinatorics and permutations, was first presented by Thomas Kirkman in a paper from 1857 to the Historic Society of Lancashire and Cheshire. Combinatorial methods are today very important in statistics and computer science. In many computer applications, determining the efficiency of algorithms requires some skills in counting.

In this chapter we will be counting choices or distributions which may be ordered or unordered and in which repetitions may or may not be allowed. Counting becomes a very important activity in combinatorial mathematics. Graph theory is also is also included because we are often concerned with counting the number of objects of a given type in particular graphs.

2.1 The Rule of Product

Let us start with an important rule called *the rule of product* also known as *the principal choice* which is one of the fundamental principals of counting.

Definition 1 (The Rule of Product)**.** If a procedure can be broken down into first and second stages, and if there are m possible outcomes for the first stage and if, for each of these outcomes, there are n possible outcomes for the second stage, then the total procedure can be carried out, in the designated order, in $m \times n$ ways.

Example 1. To choose one of $\{X, Y\}$ and one of $\{A, B, C\}$ is to choose one of $\{XA, XB, XC, YA, YB, YC\}$ according to the rule of product.

Example 2. The students' farce of Luleå University of Technology is holding tryouts for the spring play "Jakten på Dr. Livingstone". With two men and three women auditioning for the leading male and female roles. In how many ways can the director cast his leading couple?

Solution. By the rule of product, the director can cast his leading couple in $2 \times 3 = 6$ ways.

2.2 Permutations - When Order Matters

Counting linear arrangements of distinct objects are often called *permutations*. We give an example adopted from [3].

Example 3. A computer science class at LuleåUniversity of Technology consists of 15 students. Four are to be chosen and seated in a row for a picture during the start of the new school year. How many such linear arrangements are possible?

Solution. The most important word in this example is *arrangement*, which implies *order*. Let A, B, C, $\ldots$, N and O denote the 15 students, then $CAGO$, $EGBO$, and $OBGE$ are three different arrangements, even though the last two involve the same four students. Each of the 15 students can occupy the first position in the row. To fill the second position, we can only select one of the fourteen remaining students because repetitions are not allowed. Continuing in this way, we find only twelve students to select from in order to fill the fourth and final position as shown in Table 2.1.

1st position		2nd position		3rd position		4th position
15	×	14	×	13	×	12

Table 2.1: The amount of students in each position.

This gives us a total of 32760 possible arrangements of four students selected from the class of 15.

The following notation allow us to express our answers in a more convenient form.

Definition 2. For an integer $n \geq 0$, n factorial (denoted $n!$) is defined by

$$0! = 1, \tag{2.1}$$

$$n! = (n)(n-1)(n-2)\cdots(3)(2)(1), \qquad \text{for } n \geq 1. \tag{2.2}$$

Example 4. Calculate the 5 factorial or 5!.

Solution. $5! = 5 \times 4 \times 3 \times 2 \times 1 = 120$.

The values of $n!$ increase very fast. To better appreciate how fast $n!$ grows we calculate $10! = 3628800$ which happens to be the number of seconds in six weeks. In the same way 11! exceeds the number of seconds in one year, 12! in twelve years and 13! surpasses the number of seconds in a century.

2.2.1 Permutations Without Repetition

Definition 3. In general, given n distinct objects, denoted $a_1, a_2, \ldots, a_n$, and an integer r, where $1 \leq r \leq n$, then by the rule of product, the number of permutations of size r for the n objects as shown in Table 2.2 becomes

1st position		2nd position		3rd position				rth position
n	×	(n-1)	×	(n-2)	×	...	×	$(n-r+1)$

Table 2.2: The amount of n distinct objects in each position.

$$(n)(n-1)(n-2)\cdots(n-r+1)\times\frac{(n-r)(n-r-1)\cdots(3)(2)(1)}{(n-r)(n-r-1)\cdots(3)(2)(1)} \quad (2.3)$$

which in factorial notation results to

$$\frac{n!}{(n-r)!}. \quad (2.4)$$

We denote Equation 2.4 by $P(n,r)$. When $r=n$ we find that $P(n,n)=\frac{n!}{0!}=n!$.

2.2.2 Permutations With Repetition

If repetition is allowed then again by the rule of product there are

$$n^r \quad (2.5)$$

possible arrangements with $r \geq 0$.

Example 5. The letters in the word COMPUTER can be permuted in 8! different ways. If only five of the letters are used, the number of permutations of size 5 is $P(8,5)=\frac{8!}{(8-5)!}=\frac{8!}{3!}=6720$. If repetition of letters are allowed, the number of possible arrangements are $8^5=32768$.

We give a general principal for arrangements with repeated symbols which is common in the derivation of discrete and combinatorial formulas.

If there are n objects with n_1 of a first type, n_2 of a second type, ..., and n_r of rth type, where $n_1+n_2+\cdots+n_r=n$, then there are

$$\frac{n!}{n_1!n_2!\cdots n_r!} \quad (2.6)$$

linear arrangements of the given n objects.

Example 6. How many possible arrangements of all the letters in TORONTO are there?

Solution. There are $\frac{7!}{3!2!1!1!}=420$ possible arrangements.

For any counting problem, we should always ask ourselves about the importance of order in the problem. When dealing with a problem where order matters we have to think in terms of permutations, arrangements and the rule of product. But then order does not matter, we think in terms of combinations.

2.3 Combinations - When Order do not Matter

2.3.1 Combinations Without Repetition

Definition 4. If we start with n distinct objects, each *selection* or *combination* of r of these objects, with no reference to order, corresponds to $r!$ permutations of size r from the n objects. Thus the number of combinations of size r from a collection of size n, denoted $C(n,r)$, where $0 \leq r \leq n$ satisfies $(r!)\times C(n,r)=P(n,r)$ and

$$C(n,r)=\frac{P(n,r)}{r!}=\frac{n!}{r!(n-r)!}, \qquad 0 \leq r \leq n \quad (2.7)$$

The binomial coefficient symbol $\binom{n}{r}$ is also used instead of $C(n,r)$ and are sometimes read as "n choose r". Note that $C(n,0) = 1$ for all $n \geq 0$.

Example 7. Luleå Academic Computer Society (LUDD) is hosting the yearly taco party but there is only room for 40 members and 45 wants to get in. In how many ways can the chairman invite the lucky 40 members? The order is not important.

Solution. The chairman can invite the lucky 40 members in $C(45,40) = \binom{45}{40} = \frac{45!}{5!40!} = 1221759$ ways. However once the 40 members arrive, how the chairman arranges them around the table becomes an arrangement problem.

2.3.2 Combinations With Repetition

In general if there are $r + (n-1)$ positions, and we want to choose, with repetition, r of n distinct objects, then the number of combinations is

$$\frac{(n+r-1)!}{r!(n-1)!} = \binom{n+r-1}{r} = C(n+r-1,r) \tag{2.8}$$

Now we consider one example where we are concerned with how many of each item are purchased, not with the order in which they are purchased. The problem becomes a selection problem or combinations problem with repetition where each object can be chosen more than once.

Example 8. An ice cream shop offers five flavors of ice cream: vanilla, chocolate, strawberry, banana and lemon. You can only have three scoops. How many variations will there be?

Solution. There will be $\frac{(5+3-1)!}{3!(5-1)!} = \frac{7!}{3!4!} = 35$ variations.

2.4 Graph Theory

The theory of graphs was first introduced in a paper published in 1736 by the Swiss mathematician Leonhard Euler $(1707 - 1883)$. He developed some of the fundamental concepts for the theory of graphs. The idea behind this grew out of a, now popular, problem known as the seven bridges of Königberg [3], [4], [5], [6]. The town Königsberg, in eastern Prussia, contained a central island called Kneiphof, around which the river Pregel flowed before dividing into two. The four parts of the city (A,B,C,D) were interconnected by the seven bridges (a,b,c,d,e,f,g) as shown in Figure 2.1. The citizens of Königsberg entertained themselves by trying to find a route that crosses each bridge exactly once and returns to the starting point.

Euler showed[1] that such a journey was impossible, not only for the Konigsberg bridges, for any network of bridges. The reason is that for such a journey to be possible, each land mass should have an even number of bridges connected to it. If the journey would begin at one land mass and end at another, then exactly those two land masses could have an odd number of connecting bridges while all other land masses must have an even number of connecting bridges. All the land masses of Konigsberg have an odd number of connecting bridges and the journey that would take a traveler across all the bridges, one and only one time during the journey proves to be impossible.

[1] Original publications can be found in http://math.dartmouth.edu/∼euler

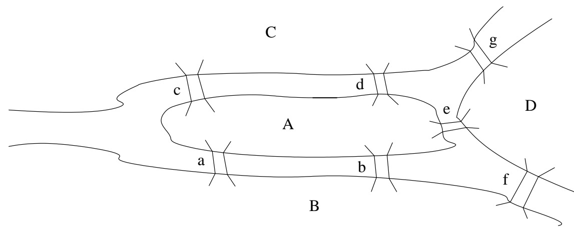


Figure 2.1: The seven bridges of Könighsberg with four land areas interconnected by the seven bridges.

2.4.1 Introducing Graphs

A graph is informally thought of as a collection of points in a plane called *vertices* or *nodes*. Some of the vertices are connected by line segments called *edges* or *arcs*. Note that the graphs we are going to study here are not functions plotted on an (x, y) coordinate system.

We can represent the seven bridges of Königsberg in Figure 2.1 by a graph as shown in Figure 2.2. Basically the part of the towns corresponds to vertices (A, B, C, D) and

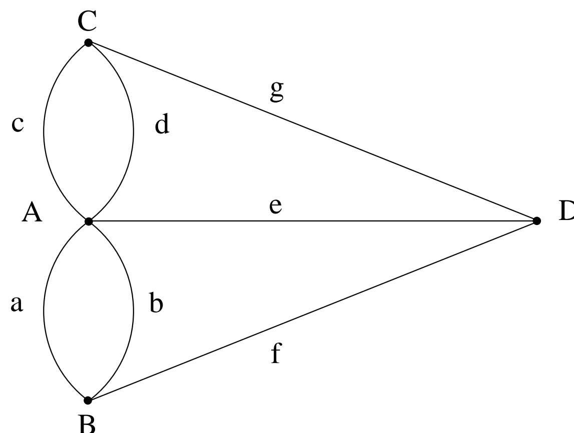


Figure 2.2: The graph representation of the seven bridges problem.

the bridges corresponds to edges (a, b, c, d, e, f, g).

2.4.2 Directed Graphs and Undirected Graphs

One can find *directed graphs* naturally in many applications of graph theory. For example, the street map of a city, abstract representation of computer programs and network flows, the study of sequential machines, and system analysis in control theory can be modeled by directed graphs rather than graphs.

When dealing with two distinct objects like towns and roads, we can define a relation. If V denotes the set of towns and E the set of roads, we define a relation $\Re$ on V by $(a \Re b)$ if we can travel from a to b on the roads in E. If the roads on E from

a to b are two-way roads, we also get the relation $(b \Re a)$. If all the roads are two-way, we get a symmetric relation.

Definition 5. Let V be a finite nonempty set, and let $E \subseteq V \times V$. The pair (V, E) is then called a directed graph on V, or *digraph* on V, where V is the set of vertices or nodes and E is its set of directed edges or arcs. We write the graph as $G = (V, E)$.

When there is no concern about the direction of any edge, we still write $G = (V, E)$. But now E is a set of undirected pairs of elements taken from V, and G is called an *undirected graph*. In general, if a graph G is not specified as directed or undirected, it is assumed to be undirected. Whether $G = (V, E)$ is directed or undirected, we often call V the *vertex set* of G and E the *edge set* of G.

(a) Undirected graph

(b) Directed graph or digraph

Figure 2.3: The difference between an undirected and a directed graph is indicated by drawing an arrow to show the direction.

The graph in Figure 2.3(a) has six vertices and seven edges:

$$V = \{a, b, c, d, e, f\} \tag{2.9}$$
$$E = \{(a, b), (a, d), (b, c), (b, e), (c, f), (d, e), (e, f)\} \tag{2.10}$$

The directed graph in figure 2.3(b) has six vertices and eight directed edges:

$$V = \{a, b, c, d, e, f\} \tag{2.11}$$
$$E = \{(a, b), (b, e), (e, b), (c, b), (c, f), (e, f), (d, e), (d, a)\} \tag{2.12}$$

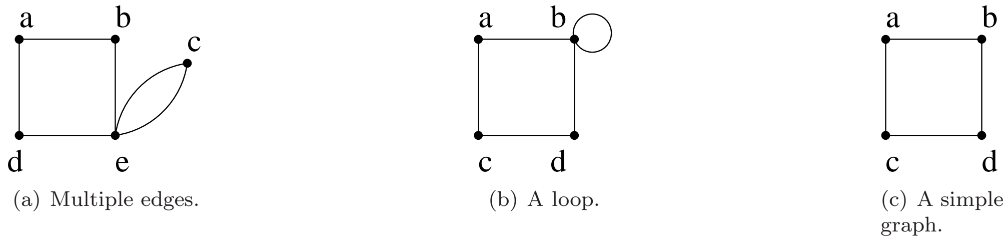
Definition 6. In a graph, two or more edges joining the same pair of vertices are *multiple edges*. An edge joining a vertex to itself as a *loop*, see Figure 2.4(b).

2.4.3 Adjacency and Incidence

Since graph theory is primarily concerned with relationships between objects, it is convenient to introduce some terminology that indicates when certain vertices and edges are next to each other in a graph.

Definition 7. The vertices a and b of a graph are *adjacent* vertices if they are joined by an edge e. The vertices a and b are *incident* with the edge e, and the edge e is incident with the vertices a and b.

Example 9. In the graph of Figure 2.5, the vertices a and d are adjacent, vertex e is incident with edges $3, 4, 5$ and 6. Edge 7 is incident with vertex d.



(a) Multiple edges.

(b) A loop.

(c) A simple graph.

Figure 2.4: The two edges (e, c) and (c, e) joining the same pair or vertices in Figure 2.4(a) is a graph with multiple edges. Figure 2.4(b) has a loop in node b and Figure 2.4(c) is just a simple graph with no loop or multiple edges.

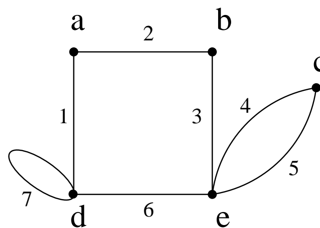


Figure 2.5: A graph showing the relationship between adjacency and incidence.

2.4.4 Paths and Cycles

There exist many applications of graphs which involve getting from one vertex to another. For example when finding the shortest route between different towns, the flow of current between two terminals of an electrical network and the tracing of a maze. We make this idea precise by defining a *walk* in a graph.

Definition 8. Given two vertices a and b in an undirected graph $G = (V, E)$ we define an a-b walk in G as a finite alternating sequence

$$a = a_0, e_1, a_1, e_2, a_2, e_3, \dots, e_{n-1}, e_n, a_n = b \tag{2.13}$$

of vertices and edges from G, beginning at vertex a and ending at vertex b such that the consecutive vertices and edges are incident. This involves the n edges $e_i = (a_{i-1}, a_i)$, where $1 \leq i \leq n$.

The number of edges in a walk is its *length.* Note that vertices and edges in a walk may be repeated. When $n = 0$, there are no edges and $a = b$. This is called a *trivial walk.* A a-b walk where $a = b$ and $n > 1$ is called a *closed walk*, otherwise it is an *open walk.* Note that a walk may repeat both vertices and edges.

There are of course special types of walk.

Definition 9. If no edge in the a-b walk is repeated, we call the walk a *a-b trail.* If the trail begins and ends at the same vertex, i.e, if the trail is *closed* then we call the a-b trail a *circuit.* A walk is called a *a-b path* when no vertex is repeated. An edge cannot be repeated if the two vertices of that edge aren't repeated, so a path is also a *trail.* When $a = b$, the term *cycle* is used to describe such a closed path.

We show an example from [4] to clarify definition 9 above.

Example 10. In the graph of Figure 2.6 (a) Find a walk that is not a trail. (b) Find a trail that is not a path. (c) Find five *b*-*d* paths. (d) Find the length for each path of part *c*. (e) Find a circuit that is not a cycle. (f) Find all distinct cycles that are present.

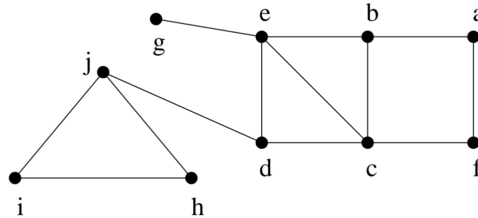


Figure 2.6: An example of a large graph.

Solution. (a) g, e, d, c, b, e, d is an example of a walk that is not a trail. It repeats the edge e, d. (b) g, e, d, c, b, e, c is a trail that is not a path. It repeats the vertices c and e, which is not allowed for a path. (c) b, c, d; b, e, d; b, c, e, d; b, a, f, c, d; and b, a, f, c, e, d. (d) Remember that the length is the number of edges in the path, not the number of vertices. The lengths of the given paths are $2, 2, 3, 3$ and 4. (e) c, b, a, f, c, e, d, c is a circuit in the graph. Note that it repeats vertices but does not repeat any edges. (f) This is done best by organizing the cycles by length. There are three cycles of length 3 : i, j, h, i; c, d, e, c; and b, c, e, b. There are two cycles of length 4 : b, c, d, e, b and a, b, c, f, a. There is one cycle of length 5 : a, b, e, c, f, a and one of length 6 : a, b, e, d, c, f, a

Definition 10. Let $G = (V, E)$ be an undirected graph. If there is a path from any point to any other point in the graph, i.e every pair of its vertices is connected by a path is called a *connected graph*. A graph that is not connected is said to be a *disconnected graph*.

Definition 11. Let $G = (V, E)$ be a directed graph. Its associated undirected graph is the graph obtained from G by ignoring the directions on the edges. If more than one undirected edge results for a pair of distinct vertices in G, then only one of these edges is drawn in the associated undirected graph. When this associated graph is connected, we consider G connected.

For example, Figure 2.7 is a disconnected graph because there is no path from a and c. However, the graph is composed of pieces with vertex sets $V_1 = \{a, b, d, e\}$, $V_2 = \{c, f\}$ and edge set $E_1 = \{\{a, b\}, \{a, d\}, \{b, e\}, \{d, e\}\}$, $E_2 = \{c, f\}$ that are connected. These pieces are called the (connected) components of the graph. Hence an undirected graph $G = (V, E)$ is disconnected if and only if V can be partitioned into at least two subsets V_1, V_2 such that there is no edge in E of the form $\{x, y\}$ where $x \in V_1$ and $y \in V_2$. A graph is connected if and only if it has only one component.

Definition 12. For any graph $G = (V, E)$, $\kappa(G)$ denotes the number of components of G.

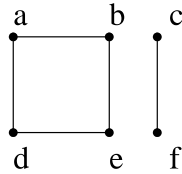


Figure 2.7: A disconnected graph.

So far we have allowed at most one edge between two vertices. We extend our concept of a graph by considering an extension.

Definition 13. Let V be a finite nonempty set. Then the pair (V, E) determines a *multigraph* G with vertex set V and edge set E if, for some $x, y \in V$, there are two or more edges in E of the form (a) (x, y) (for directed multigraph), or (b) $\{x, y\}$ (for an undirected multigraph). In both cases we write $G = (V, E)$ to designate the multigraph.

2.4.5 Subgraphs

We often want to solve complicated problems by looking at simpler objects of the same type. We do that in mathematics by sometimes studying subsets of sets, subgroups of groups and so on. In graph theory we define subgraphs of graphs.

Definition 14. Let $G = (V, E)$ be a directed or undirected graph, then $G_1 = (V_1, E_1)$ is called a *subgraph* of G if $\varnothing \neq V_1 \subseteq V$ and $E_1 \subseteq E$, where each edge in E_1 is incident with vertices in V_1.

Figure 2.8: A graph G and one of its subgraphs G_1 with an isolated node d.

Definition 15. A subgraph $G_1 = (V_1, E_1)$ of graph G is called a *spanning subgraph* of G if $V_1 = V$. In that case we say that G_1 spans G.

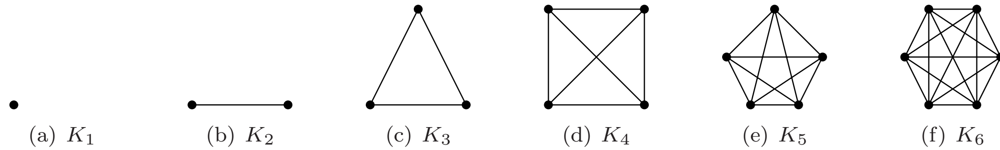
We form a spanning subgraph of a given graph by simply deleting edges. The subgraph in Figure 2.8(a) is also a spanning subgraph. It follows that a labeled graph[2] with e edges has 2^k spanning subgraphs.

[2]A graph whose vertices have labels attached to it.

Definition 16. Let $G = (V, E)$ be a directed or undirected graph. If $\varnothing \neq S \subseteq V$ then the *subgraph* induced by S, denoted $\langle S \rangle$ is the subgraph whose vertex set is S and which contains all edges form G.

Definition 17. Let V be the set of vertices. The *complete graph* on V, denoted K_n is a loop free undirected graph where for all $a, b \in V$, $a \neq b$, there is an edge $\{a, b\}$.

In a more readable form, the definition above means that a complete graph is a graph where each vertex is connected to each of the others by exactly one edge.



(a) K_1 (b) K_2 (c) K_3 (d) K_4 (e) K_5 (f) K_6

Figure 2.9: The complete graph with n vertices denoted by K_n

Since there are n vertices, this implies that the number of edges satisfies

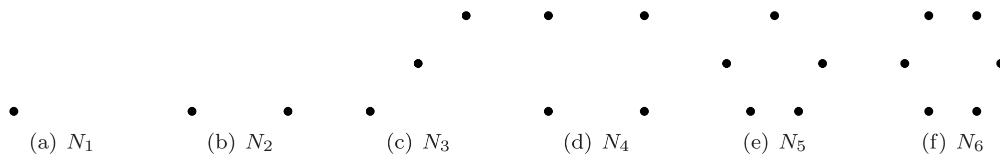
$$|E(K_n)| = \binom{n}{2} = \frac{n(n-1)}{2}. \tag{2.14}$$

It also follows that this number is an upper bound of the number of edges of any graph on n vertices defined as

$$|V(G)| = n \implies \quad |E(G)| \leq \frac{n(n-1)}{2} \tag{2.15}$$

Definition 18. Let G be a graph on n vertices. Then the *complement* $\overline{G}$ of G is the subgraph of K_n consisting of the n vertices in G and all the edges that are not in G.

It is clear that $\overline{G}$ is also a simple graph and that $\overline{(\overline{G})} = G$. If $G = K_n$ then $\overline{G}$ is a *null graph* consisting of n vertices and no edges. This means that the *singleton graph*[3] in Figure 2.10(a) is considered connected, while empty graphs on $n \geq 2$ nodes are disconnected.



(a) N_1 (b) N_2 (c) N_3 (d) N_4 (e) N_5 (f) N_6

Figure 2.10: The null graph of $1, 2, 3, 4, 5$ and 6 vertices denoted N_n where n is the amount of vertices or nodes.

[3] A single isolated node with no edges, i.e, the null graph on 1 node.

2.4.6 Vertex Degrees

It is convenient to define a term for the number of edges meeting at a vertex. For example, when we wish to specify the number of roads meeting at a particular intersection or the number of chemical bonds joining an atom to its neighbors.

Definition 19. Let G be an undirected graph or multigraph. The number of edges incident at vertex v in G is called the *degree* or *valence* of v in G, written $d_G(v)$ or simply $deg(v)$ when G requires no explicit reference.

A loop at v is to be counted twice in computing the degree of v. The minimum of the degrees of the vertices of a graph G is denoted $\delta(G)$ and $\Delta(G)$ for the maximum number of degrees. An undirected graph or multigraph where each vertex has the same degree is called a *regular graph*, if $deg(v) = k$ for all vertices v, then the graph is called k-regular. In particular, a vertex of degree 0 is an *isolated vertex* of G. A vertex of degree 1 is called a *pendant vertex*.

As mentioned before, the very first theorem of graph theory was due to Leonhard Euler.

Theorem 1 (Euler). *The sum of the degrees of the vertices of a graph is equal to twice the number of its edges*

$$\sum_{v \in V} deg(v) = 2|E|. \tag{2.16}$$

Proof. An edge $e = \{a, b\}$ of G, is counted once while counting the degrees of each of a and b, even when $a = b$. Consequently each edge contributes 2 to the sum of the degrees of the vertices $(2 \times \sum_{v \in V} deg(v))$. Thus $2|E|$ accounts for $deg(v)$, for all $v \in V$ and $\sum_{v \in V} deg(v) = 2|E|$. □

Corollary 1. For any graph G, the number of vertices of odd degree is even.

Proof. Let V_1 and V_2 be the subsets of vertices of G with odd and even degrees respectively. By Theorem 1

$$2|E| = \sum_{v \in V} deg(v) = \sum_{v \in V_1} deg(v) + \sum_{v \in V_2} deg(v) \tag{2.17}$$

The numbers $2|E|$, $\sum_{v \in V_2} deg(v)$ and $\sum_{v \in V_1} deg(v)$ are even. Then for each vertices $v \in V_1$, $deg(v)$ is odd, $|V_1|$ must be even. □

Definition 20. Let $G = (V, E)$ be an undirected graph or multigraph with no isolated vertices. If there is a circuit in G that traverses every edge of the graph exactly once then G has an *Euler circuit*. An open trail that traverses each edge in G exactly once is called an *Euler trail* or *Euler path*.

So far we have presented a lot of definitions. We can now finally conclude that the seven bridges problem actually requires us to find an Euler circuit. The questions remains, is there an easy way to find out if a graph G is an Euler circuit or an Euler trail without trying to traverse every single edge by hand?

Theorem 2. *Let $G = (V, E)$ be an undirected graph or multigraph with no isolated vertices. Then G has an Euler circuit if and only if G is connected and every vertex has even degree.*

Proof. Can be found in [4]. □

Corollary 2. If G is an undirected graph or multigraph with no isolated vertices, then the connected graph G has an Euler trail if and only if it has at most two vertices of odd degree.

Remark. An Euler trail in G must begin at one of the odd vertices and end at the other.

We return once again to the seven bridges problem. We observer from Figure 2.2 that each vertex has an odd number of edges. For example, $deg(B) = deg(C) = deg(D) = 3$ and $deg(A) = 5$. Therefor the citizens of Königsberg could not find a solution as each edge can be used only once and all the vertices are odd. It is impossible to re-enter any vertex again after leaving it and this makes the starting and ending at the same point impossible as conducted in the beginning of this chapter.

2.4.7 Graph Representation

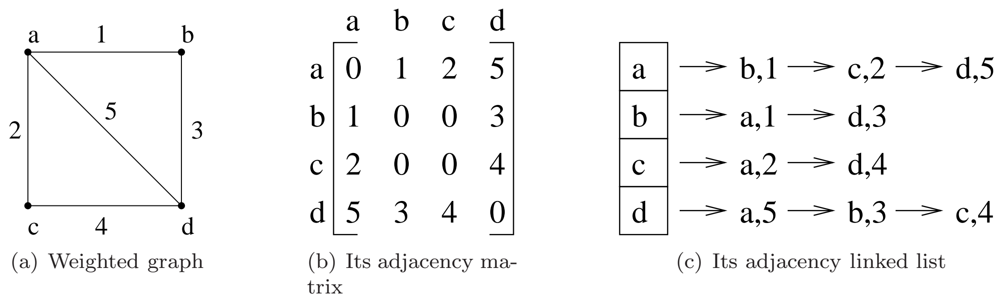
There are two important principal ways to represent graphs in an algorithm. One is the *adjacent matrix* and the second is the *adjacent list*.

Definition 21. Let G be an undirected graph with n vertices. The adjacent matrix $A(G)$ of G is the $n \times n$ boolean matrix with one row and one column for each of the graph's vertices, in which the entry in row i and column j is equal to 1 if there is an edge joining the ith vertex to the jth vertex and equal to 0 if there is no such edge.

Remark. The adjacency matrix of an undirected graph is always symmetric, i.e., $A[i, j] = A[j, i]$ for every $i \geq 0$ and $j \leq n - 1$.

When assigning numbers to a graphs edges we get a so called *weighted graph* or *weighted digraph*. These numbers are called *weights* or *costs*. If a weighted graph is represented by its adjacency matrix, then its element $A[i, j]$ will contain the weight of the edge from the ith to the jth vertex if such an edge exists and 0 or sometimes ∞ if not.

The following figures show the relationship between an graph and its adjacency matrix and its adjacency linked list as shown in Figure 2.11(c).



(a) Weighted graph (b) Its adjacency matrix (c) Its adjacency linked list

Figure 2.11: The graphs show two different ways to represent the weighted graph.

2.4.8 Graph Applications

A chemist named Cayley found a good use for graph theory. The earliest application to chemistry was found by him in 1857.A chemical molecule can be represented by a graph by mapping each atom of the molecule to a vertex of the graph and making the edges represent atomic bounds. The degree of each vertex gives the valence of the corresponding atom.

Graphs in computer science and parallel programming are crucial. When working on parallel computers one defines and models the communication protocol using graphs. The experiments in later chapters would be impossible without graph theory and combinatorics. Each CPU represents a node or vertex and each edge defines the intercommunication between the CPUs when sending and receiving solutions from and to each CPU. For example, two CPUs or processors, say p_1 and p_2 are able to communicate directly with one another. We draw the edge $\{p_1, p_2\}$ to represent this line of possible communication. Note that a graph with relatively few edges missing is called a *dense graph* and a graph with few edges relative to the number of its vertices is called a *sparse graph.* The running time of an algorithm is heavily dependent on whether we are dealing with a dense or a sparse graph. How to decide on a model for the communication, i.e., the graph to speed up the processing time becomes an optimization problem.

Chapter 3

Combinatorial Optimization

Optimization or *mathematical programming* is the study of problems where the main goal is to minimize or maximize a function by systematically choosing the values of real variables from an allowed set. The problem is represented in the following way.

Example 11. Given a function $f : A \to \mathbb{R}$ from a set A to the real numbers, find an element x_0 in A such that $f(x_0) \leq f(x)$ for all $x \in A$.

In an optimization problem, A is a *subset* of the Euclidean space $\mathbb{R}^n$, sometimes described by a set of *constraints*. The domain A of f is called the *search space*. The elements of A are called *feasible solutions* and the function f is called an *objective function* or a *cost function*. The solution becomes an optimal solution when a feasible solution minimizes of maximizes the objective function.

Definition 22. An *instance* of an optimization problem is a pair (A, f), where A is the domain of feasible points and f is the cost function with a mapping

$$f : A \to \mathbb{R}^1. \tag{3.1}$$

The problem is to find a solution $x_0 \in f$ for which

$$f(x_0) \leq f(x) \quad \text{for all } x \in A \tag{3.2}$$

Such a point is called a *globally optimum* solution to the given instance or simply an *optimal* solution.

Definition 23. An *optimization problem* is a set I of instances of an optimization problem.

Note the difference between a problem and an instance of a problem. In an instance we are given the "input data" and have enough information to obtain a solution. A problem is a collection of instances. For example, an instance of the *traveling salesman problem* (in Section 3.1.1) has a given distance matrix, but we speak in general of the traveling salesman problem as the collection of all the instances associated with all distance matrices.

Definition 24. A point x_0 is a *locally optimal* solutions to an instance I if

$$f(x_0) \leq f(x) \qquad \text{for all } x \in N(x_0) \tag{3.3}$$

where N is a *neighborhood* defined for each instance in the following way

Definition 25.

$$N_\varepsilon(x_0) = \{x : x \in A \text{ and } |x - x_0| \leq \varepsilon\}. \tag{3.4}$$

Over the past few decades major subfields of optimization has emerged, together with a corresponding collection of techniques for their solution. The first subfield is the *nonlinear programming problem* where the main goal is to

$$\min_{x \in \mathbb{R}^n} f(x) \tag{3.5}$$

subject to

$$g_i(x) \geq 0, \qquad i = 1, \ldots, m \tag{3.6}$$

$$h_j(x) = 0, \qquad j = 1, \ldots, n \tag{3.7}$$

where f is an objective function, g_i and h_j are general functions of $x \in \mathbb{R}^n$. If f is convex, g_i concave, and h_j linear we arrive to a new subfield in optimization where the problem is now called *convex programming problem.* If f, g_i and h_j are all linear, we come to another major subfield in optimization called the *linear programming problem.* A widely used algorithm called the *simplex algorithm* of G.B Dantzig finds an optimal solution to a linear programming problem in a finite number of steps. After thirty years of improvement, it now solves problems with hundreds of variables and thousands of constraints. When dealing with problems where the set of feasible solutions are finite and discrete, or can be reduced to a discrete one, we call the optimization problem *combinatorial.* Just to mention a few more, there exists a subfields like *quadratic programming problem* where the objective is a quadratic function of the decision variables, and constraints which are all linear functions of the variables.

Applying the structures of trees[1] and graphs, we mainly work with optimization techniques that arise in the area of *operations research.* These techniques can be applied to graphs and multigraphs with a positive integer weight associated to each edge of the graph or multigraph. The weights relate information such as the distance between the vertices that are endpoints of the edge or the amount of material that can be shipped from one vertex to another along an edge that represents a highway or air route.

3.1 Combinatorial Optimization Problems

Optimization problems are naturally divided into two categories. Those with continuous variables and those with discrete variables which are called *combinatorial.* When working with *continuous problems* we are generally looking for a set of real numbers or even a function. In the combinatorial problems, we are looking for an object from a finite or possibly infinite set, typically an integer set, permutation or graph. In our work, we focus mainly on discrete optimization or *combinatorial optimization problems* and the process of finding an optimal solutions in a well defined discrete space.

Similar to Equation 3.5, a combinatorial optimization problem, $\mathcal{P}$, assumes the form of

$$\begin{array}{cc} \min f(\overline{x}) & \text{subject to} \\ C_1(\overline{x}) & \\ \vdots & \\ C_n(\overline{x}) & \end{array} \tag{3.8}$$

[1] A tree is a connected acyclic graph.

where f is an objective function ($\mathbb{N}^n \rightarrow \mathbb{N}$) that associates a performance measure with a variable assignment, $\overline{x}$ is a vector of n discrete decision variables, and $C_1, \ldots, C_n$ are constraints defining the solution space.

Definition 26. A *solution* to $\mathcal{P}$ is an assignment of values to the variables in $\overline{x}$. The set of solutions to $\mathcal{P}$ is denoted by $\mathcal{L}_{\mathcal{P}}$.

Definition 27. A *feasible solution* to $\mathcal{P}$ is a solution $\hat{x}$ that satisfies all constraints $C_i(\hat{x})$ for $i = 1, \ldots, n$. The set of all feasible solutions in $\mathcal{P}$ is denoted by $\widetilde{\mathcal{L}}_{\mathcal{P}}$.

Definition 28. The set of *optimal solutions* to $\mathcal{P}$, denoted by $\mathcal{L}^*_{\mathcal{P}}$ is defined as

$$\mathcal{L}^*_{\mathcal{P}} = \{s \in \widetilde{\mathcal{L}}_{\mathcal{P}} \mid f(s) = \min_{k \in \widetilde{\mathcal{L}}_{\mathcal{P}}} f(k)\} \tag{3.9}$$

Some algorithms take into account only solutions that satisfy some of the constraints. The set of solutions over which the algorithm is defined is called the search space.

Definition 29. A *search space* of $\mathcal{P}$ is a set $\widehat{\mathcal{L}}_{\mathcal{P}}$ such that $\mathcal{L}_{\mathcal{P}} \subseteq \widehat{\mathcal{L}}_{\mathcal{P}} \subseteq \mathcal{N}^n$. Elements of the set $\widehat{\mathcal{L}}_{\mathcal{P}}$ often satisfy a subset of $\{C_1, \ldots, C_n\}$.

It is useful in many situations to define a set $N(s)$ of points that are "close" in some sense to the solution s.

Definition 30. A *neighborhood* is a pair $(\widehat{\mathcal{L}}_{\mathcal{P}}, N)$, where $\widehat{\mathcal{L}}_{\mathcal{P}}$ is a search space and N is a mapping in the following way.

$$N : \widehat{\mathcal{L}}_{\mathcal{P}} \rightarrow 2^{\widehat{\mathcal{L}}_{\mathcal{P}}} \tag{3.10}$$

that defines for each solution s, the set of adjacent solutions $N(s) \subseteq \widehat{\mathcal{L}}_{\mathcal{P}}$. If the relation $s_1 \in N(s_2) \Leftrightarrow s_2 \in N(s_1)$ holds, then the neighborhood is symmetric.

To find a globally optimal solution to an instance of $\mathcal{P}$ can be very difficult and requires in many cases a lot of computational time. But it is often possible to find a solution s which is best in the sense that there is nothing better in its neighborhood $N(s)$.

Definition 31. A solution s in $\mathcal{L}_{\mathcal{P}}$ is *locally optimal* with respect to N if

$$f(s) \leq \min_{x \in N(s)} f(x) \tag{3.11}$$

The set of locally optimal solutions with respect to N is denoted $\mathcal{L}^+_{\mathcal{P}}$.

3.1.1 The Traveling Salesman Problem

The traveling salesman problem (TSP) has kept researches busy for the last 100 years by its simple formulation, important applications and interesting connections to other combinatorial problems. An article related to the traveling salesman problem was treated by the Irish mathematician Sir William Rowan Hamilton in the 1800s and later an article was published by the British mathematician Thomas Penyngton Kirkman in 1855.

The salesman wishes to make a *tour* visiting each city exactly once and finishing at the city he starts from. A tour is a closed path that visits every city exactly once.

There is a integer cost c_{ij} to travel from city i to city j and the salesman wishes to make the tour with a minimal total cost. The total cost becomes the sum of the individual costs along the edges of the tour. The travel costs are symmetric in the sense that traveling from city i to city j costs just at much as traveling from city j to city i.

The problem can be modeled as a complete graph with n vertices. Each vertices represent a city and the edge weights specifying the distances. This is closely related to the Hamiltonian cycle problem and can be stated as the problem of finding the shortest *Hamiltonian circuit*[2] of the graph. If there are n cities to visit, the number of possible tours or paths is finite. To be precise it becomes $(n-1)!$. Hence an algorithm can easily be designed that systematically examines all tours in order to find the shortest tour. This is done by generating all the permutations of $n-1$ intermediate cities, computing the tour lengths and finding the shortest among them. Mathematically, the cost is represented as

$$c(\pi) = \sum_{j=1}^{n} d_{j\pi(j)}, \tag{3.12}$$

where a cyclic permutation π represents a tour if we interpret $\pi(j)$ to be the city visited after city j, $j = 1, \ldots, n$. Then the cost c maps π to the total sum of the costs. The objective is to miminimize the cost function in the following way.

$$\min_{\pi \in F(n)} c(\pi), \tag{3.13}$$

where $F = \{\text{all cyclic permutations } \pi \text{ on } n \text{ objects}\}$ and d_{ij} denotes the distance between city c_i and c_j. We are assuming that $d_{ii} = 0$ and $d_{ij} = d_{ji}$ for all i, j meaning that the graphs adjacency $n \times n$ $matrix[d_{ij}]$ is loop free and symmetric. Note that $d_{ij} \in \mathbb{Z}^+$.

Example 12. We show here a small instance with four cities. The objective is to find the optimal tour or the minimal cost and minimize the cost function in Equation 3.12.

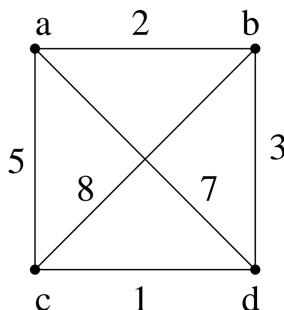


Figure 3.1: A small instance with size 4 of the traveling salesman problem.

Solution. The total number of cyclic permutations π on 4 objects becomes $(n-1)! = 3! = 6$.

[2]A Hamiltonian circuit is a cycle that passes through all the vertices of the graph exactly once.

Tour	Total cost	
$a \to b \to c \to d \to a$	$2 + 8 + 1 + 7 = 18$	
$a \to b \to d \to c \to a$	$2 + 3 + 1 + 5 = 11$	optimal
$a \to c \to b \to d \to a$	$5 + 8 + 3 + 7 = 23$	
$a \to c \to d \to b \to a$	$5 + 1 + 3 + 2 = 11$	optimal
$a \to d \to b \to c \to a$	$7 + 3 + 8 + 5 = 23$	
$a \to d \to c \to b \to a$	$7 + 1 + 8 + 2 = 18$	

We notice that three pairs of tours differ only by the tours direction, we can cut the number of vertex permutations by half. This improvement makes the total number of permutations needed into $(n-1)!/2$. Also notice that the number of permutations increase so rapidly, cutting the number of vertex permutations by half doesn't make a big difference when it comes to computational complexity.

Our brute-force approach used to solve the example above is called *exhaustive search* and is very useful when working with small instances because of its simple non sophisticated implementation. The algorithm generates each and every element of the problems domain, selecting those of them that satisfy the problem's constraints and then finds a desired element that optimizes the objective function.

When dealing with combinatorial objects such as permutations, combinations and subsets of a given set, we often don't have the computational power to use exhaustive search to find the optimal value as the instances grow in size. There are just too many tours to be examined. For our modest problem of 4 cities we get only 6 tours to examine. So the computations can easily be done by hand. A problem of 10 cities require us to examine $9! = 362880$ tours. This can easily be carried out by a computer today with its multi core architecture. What if we had 40 cities to visit? The number of tours becomes gigantic and grows to 10^{45} different permutations. Even if we could examine 10^{15} tours per second, which is really fast for the most powerful supercomputers today, the required time for completing this calculation would be several billion lifetimes of the universe!

Exhaustive search is not the best way to go and is impractical for all but very small instances of the problem. Fortunately there exists much efficient algorithms for solving problems like this.

3.2 Computational Complexity

We can classify all computational problems into two categories: those that can be solved by algorithms and those that cannot. The TSP problem is solvable in principle, but it cannot be solved in any practical sense by computers due to the excessive time requirements on bigger instances. The first concern is whether a given problem can be solved in polynomial time by some algorithm. If an algorithms worst-case time efficiency belongs to $O(p(n))$ where $p(n)$ is a polynomial of the problems input size n, then the algorithm solves the problem in *polynomial time*[3].

Definition 32. Problems that can be solved in polynomial time are called *tractable* or easy problems. Problems that can not be solved in polynomial time are called *intractable* or hard problems.

[3] If $T(n)$ is the time for an algorithm on n inputs, then we write $T(n) = O(p(n))$ to mean that the time is bounded above by the function $p(n)$.

Table 3.1 show that we cannot solve arbitrary instances of intractable problems in a reasonable amount of time unless such instances are very small.

n	$\log_2 n$	n	$n \log_2 n$	n^2	n^3	2^n	$n!$
10^1	3.3	10^1	3.3×10^1	10^2	10^3	1.0×10^3	3.6×10^1
10^2	6.6	10^2	6.6×10^2	10^4	10^6	1.3×10^{30}	9.3×10^{157}
10^3	10.0	10^3	1.0×10^4	10^6	10^9		
10^4	13.0	10^4	1.3×10^5	10^8	10^{12}		
10^5	17.0	10^5	1.7×10^6	10^{10}	10^{15}		
10^6	20.0	10^6	2.0×10^7	10^{12}	10^{18}		

Table 3.1: Values of several functions important for analysis of algorithms. Problems that grow too fast are left empty.

3.2.1 The Class $\mathcal{P}$

Given a particular input, an algorithm that always produces the same output is called a *deterministic algorithm*. This is of course the way most programs are executed on a computer. An algorithm that generate only a zero or one (true or false, or yes or no) as its output is a *decision algorithm* and a *decision problem* is a problem with a yes-or-no answer. No explicit output statements are permitted in a decision algorithm.

Definition 33. Class $\mathcal{P}$ is a class of decision problems that can be solved in polynomial time by a deterministic algorithm.

Examples of problems that belong to class $\mathcal{P}$ are searching, element uniqueness, graph connectivity and graph acyclicity.

3.2.2 The Class $\mathcal{NP}$

In contrast to a deterministic algorithm, a *nondeterministic algorithm* can produce different outputs or states when run repeatedly with the same input. Computation can branch, choosing among different execution paths in a way that does not depend only on the input and current execution state.

Definition 34. A *nondeterministic algorithm*, with an instance I of decision problem as its input, is an abstract two-stage procedure where in:

Stage one (The Nondeterministic Stage) generates an arbitrary string S by guessing. The string S becomes a candidate solution to the instance I of the problem.

Stage two (The Deterministic Stage) or verification stage, verifies whether this solution is correct in polynomial time by taking the instance I and the arbitrary generated string S as its input and outputs yes if S represents a solution to instance I, otherwise the algorithm either returns no or is allowed not to halt at all.

The algorithm can behave in a nondeterministic way, when it operates in a way that is timing sensitive. For example if it has multiple processors writing to the same data at the same time. The precise order in which each processor writes its data will affect

the result. Another cause is if the algorithm uses external state other than the input such as a hardware timer value or a random value determined by a random number generator.

Definition 35. A *nondeterministic polynomial algorithm* is an algorithm where its time efficiency of its *verification stage* is polynomial.

We can now define the class $\mathcal{NP}$.

Definition 36. *Class* $\mathcal{NP}$ is the class of decision problems that can be solved by nondeterministic polynomial algorithms.

Any problems in class $\mathcal{P}$ is always also in $\mathcal{NP}$:

$$\mathcal{P} \subseteq \mathcal{NP} \tag{3.14}$$

The open question that still remains today is whether or not class $\mathcal{P}$ is a proper subset of $\mathcal{NP}$, or if the two classes $\mathcal{P}$ and $\mathcal{NP}$ are actually equivalent. If classes $\mathcal{P}$ and $\mathcal{NP}$ are not the same, then the solution of $\mathcal{NP}$-problems requires in the worst case an exhaustive search.

Nobody has yet been able to prove whether $\mathcal{NP}$-complete problems are solvable in polynomial time, making this one of the great unsolved problems of mathematics. An award of $1 million is offered by the Clay Mathematics Institute in Cambridge, MA to anyone who has a formal proof that class $\mathcal{P} = \mathcal{NP}$ or that class $\mathcal{P} \neq \mathcal{NP}$.

Class $\mathcal{NP}$ contains the Hamiltonian circuit problem, the partition problem, the knapsack problem, graph coloring, and many hundreds of other difficult combinatorial optimization problems. If class $\mathcal{P} = \mathcal{NP}$ then many hundreds of difficult combinatorial decision problems can be solved by a polynomial time algorithm.

3.2.3 The Classes $\mathcal{NP}$-Hard and $\mathcal{NP}$-Complete

In the *proposition calculus* [7] [8], a *formula* is an expression that can be constructed using literals and the operations **and** (denoted $\wedge$) and **or** (denoted $\vee$). A *literal* is either a variable or its negation. A formula is in *conjunctive normal form* (CNF) if it is represented as $\wedge_{i=1}^{k} c_i$ where the c_i are clauses[4] each represented as $\vee l_{ij}$, and where the l_{ij} are literals. It is in *disjunctive normal form* (DNF) if it is represented as $\vee_{i=1}^{k} c_i$ and each clause c_i is represented as $\wedge l_{ij}$.

Example 13. Formula

$$(x_1 \wedge x_2) \vee (x_3 \wedge \overline{x}_4) \tag{3.15}$$

is in DNF while formula

$$(x_3 \vee \overline{x}_4) \wedge (x_1 \vee \overline{x}_2) \tag{3.16}$$

is in CNF.

Satisfiability

The *satisfiability problem* is to determine if a formula F is true for some assignment of truth values to the variables.

In an algorithm, we may use *boolean logic* for expressing compound statements. We use *boolean variables* $x_1, x_2, \ldots, x_i$ and the *negations* $\overline{x}_1, \overline{x}_2, \ldots, \overline{x}_j$ to denote the

[4] A clause is a disjunction of literals.

individual statements. Each statement can be true or false independently of the truth value of the others. We then use *boolean connectivity* to combine boolean variables into a *boolean formula*. For example,

$$F = \overline{x}_3 \vee (x_1 \wedge \overline{x}_2 \wedge x_3) \tag{3.17}$$

is the boolean formula and given a value $t(x)$ for each variable x, we can evaluate the boolean formula just the same way we would do with an algebraic expression. The *truth assignment* $t(x_1) = true$, $t(x_2) = true$, and $t(x_3) = false$ gives the value true to F in Equation 3.17, thus the boolean formula becomes *satisfiable*.

Reducibility

A problem (or a language) L_1 can be *reduced* to another problem L_2 if any instance I of L_1 can be "easily rephrased" as an instance of L_2 with the solution s to which provides a solution to the instance of L_1. For example, the problem of solving linear equations where x reduces to the problem of solving quadratic equations. Given an instance $ax + b = 0$, we transform it to $0x^2 + ax + b = 0$, whose solution provides a solution to $ax + b = 0$. Thus, if a problem L_1 reduces to another problem L_2 then L_1 is, "no harder to solve" than L_2.

Definition 37. Let L_1 and L_2 be decision problems. We say that L_1 *reduces in polynomial time* to L_2, also written $L_1 \propto L_2$, if and only if there is a way to solve L_1 by a deterministic polynomial time algorithm A_1 using a deterministic algorithm A_2 that solves L_2 in polynomial time.

This definition implies that if we have a polynomial time algorithm for L_2 then we can solve L_1 in polynomial time.

Definition 38. A problem or language L is *$\mathcal{NP}$-hard*[5] if and only if it is at least as hard as any problem in $\mathcal{NP}$. A problem or language L is *$\mathcal{NP}$-complete* if and only if L is $\mathcal{NP}$-hard and $L \in \mathcal{NP}$.

Alternative definitions of the $\mathcal{NP}$-hard class do exist based on satisfiability and reducability which is computable by a deterministic Turing machine in polynomial time. The Venn diagram in Figure 3.2 depicts the relationship between the different classes.

So, an *$\mathcal{NP}$-complete problem* is a problem in $\mathcal{NP}$ that is as difficult as any problem in this class. We refer to it as being *$\mathcal{NP}$-complete* if it is in $\mathcal{NP}$ and is as "hard" as any problem in $\mathcal{NP}$. Only a decision problem can be $\mathcal{NP}$-complete but $\mathcal{NP}$-hard problems may be of any type: decision problems, search problems or optimization problems. An example of a $\mathcal{NP}$-hard decision problem that is not $\mathcal{NP}$-complete is the halting problem [9].

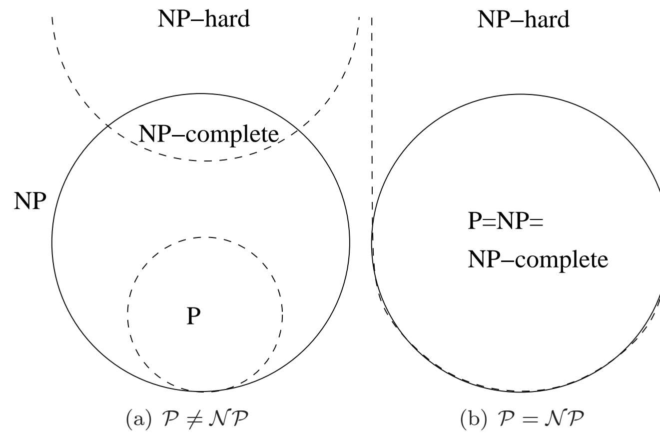
Finally, the following theorem shows that a fast algorithm for the traveling salesman problem is unlikely to exist.

Theorem 3. *The traveling salesman problem is $\mathcal{NP}$-hard.*

Proof. Proof can be found in [10]. □

A list of more than 200 $\mathcal{NP}$-hard optimization problems can be found in [11].

[5] $\mathcal{NP}$-hard = nondeterministic polynomial time hard. A common mistake is to think that $\mathcal{NP}$ in $\mathcal{NP}$-hard stands for non polynomial.



(a) $\mathcal{P} \neq \mathcal{NP}$ (b) $\mathcal{P} = \mathcal{NP}$

Figure 3.2: The Venn diagram for $\mathcal{P}$, $\mathcal{NP}$, $\mathcal{NP}$-complete, and $\mathcal{NP}$-hard set of problems.

3.3 Heuristics

The term *heuristic* comes from the Greek "heurisko", which means "I find". You may recognize it as a form of the same verb as heureka, the word Archimedes once screamed naked on the streets of Syracuse.

Because of the complexity of a combinatorial optimization problem $\mathcal{P}$, it may not always be possible to search the whole search space using conventional algorithms to find an optimal solution. In such situations, it is still important to find a good feasible solution that is at least reasonably close to being globally optimal. *Heuristic methods* are used on $\mathcal{NP}$-hard problems to search for such a solution. A heuristic method is a procedure that is likely to find a very good feasible solution, but not necessarily an optimal solution for the specific instance. A well designed heuristic method can usually provide a solution that is at least nearly optimal or conclude that no such solution exist, but no guarantee can be given about the quality of the solution obtained. It should also be very efficient to deal with larger instances and is often an iterative algorithm, where each iteration involves conducting a search for a new solution that might be better than the best solution found in a previous search. When the algorithm is terminated after a reasonable amount of time or simply when it reaches a number of predefined iterations, the solution it provides is the best one that was found during any iteration.

3.4 Heuristic Methods

3.4.1 Local Search

Local search is a very important *heuristic method* for solving a computationally hard optimization problem $\mathcal{P}$. It is based on perhaps the oldest optimization method, known as *trial and error*. In trial and error, one selects a possible optimal solution and apply it to the problem. If it is not the optimal solution, then one generates or selects another possibility that is subsequently tried. The process ends when a possibility yields the optimal solution.

The is a difference between the trial and error method in local search and the

exhaustive search method applied to the traveling salesman problem in previous section. Exhaustive search, which is based on the primitive *brute-force method* where we generate all solutions and search for the globally optimal one. Local search, which is non-exhaustive in the sense that it doesn't guarantee to find a feasible or optimal solution, searches non-systematically until a specific stop criterion is satisfied. This is actually one of the reasons that makes local search so successful on a variety of difficult combinatorial optimization problems compared to exhaustive search.

The local search algorithm operates in a simple way. Given an instance I of a combinatorial optimization problem $\mathcal{P}$, we associate the search space, $\widehat{\mathcal{L}}_{\mathcal{P}}$ to it. Each element $s \in \widehat{\mathcal{L}}_{\mathcal{P}}$ corresponds to a potential solution of I, and is called a *state* of I. The local search algorithm relies on a function N which assigns to each $s \in \widehat{\mathcal{L}}_{\mathcal{P}}$ its neighborhood $N(s) \subseteq \widehat{\mathcal{L}}_{\mathcal{P}}$. Each state $s' \in N(s)$ is called a neighbor of s. The neighborhood is composed by the states that are obtained by the local changes called *moves*. The local search algorithm starts from an initial state s_0 and enters a loop that navigates the search space, moving from one state s_i to one of its neighbors s_{i+1} in hope of improving (minimizing) a function f. The function f measures the quality of solutions.

A move from a solution to a neighboring solution is defined by the concepts of neighborhood, local optimality and *transition graph*. The move is controlled by a *legality condition* function L and a *selection rule* function S to help local search escape local minima and find a high-quality local optimum.

Definition 39. The *transition graph* $G(\widehat{\mathcal{L}}_{\mathcal{P}}, N)$ associated to a neighborhood $(\widehat{\mathcal{L}}_{\mathcal{P}}, N)$ is the graph whose nodes are solutions in $\widehat{\mathcal{L}}_{\mathcal{P}}$ and where an arc $a \to b$ exists if $b \in N(a)$. The reflexive and transitive closure of $\to$ is denoted by $\to^*$.

Definition 40. A *legality condition* L is a function $(2^{\widehat{\mathcal{L}}_{\mathcal{P}}} \times \widehat{\mathcal{L}}_{\mathcal{P}}) \to 2^{\widehat{\mathcal{L}}_{\mathcal{P}}}$ that filters sets of solutions from the search space. A *selection rule* $S(\mathcal{M}, s)$ is a function S : $(2^{\widehat{\mathcal{L}}_{\mathcal{P}}} \times \widehat{\mathcal{L}}_{\mathcal{P}}) \to 2^{\widehat{\mathcal{L}}_{\mathcal{P}}}$ that picks an element s from $\mathcal{M}$ according to some strategy and decides to accept it or to select the current solution s instead.

Definition 41. A *local search algorithm* for $\mathcal{P}$ is a path

$$s_0 \to s_1 \to \ldots s_k \tag{3.18}$$

in the transition graph $G(\widehat{\mathcal{L}}_{\mathcal{P}}, N)$ for $\mathcal{P}$ such

$$s_{i+1} = S(L(N(s_i), s_i), s_i) \qquad \text{where} \quad 1 \le i \le k. \tag{3.19}$$

A local search produces a final computation state s_k that belongs to a set of locally optimal solutions with respect to N, e.g., $s_k \in \mathcal{L}^{+}_{\mathcal{P}}$.

At a specific iteration, some of the neighbors may be *forbidden*, and therefor may not be selected, or they may be *legal*. Once the legal neighbors are identified by operation L, the local search selects one of them and decides whether to move to this neighbor or to stay at s (operation S). We illustrate these concepts in Figure 3.3.

Algorithm 1 depicts a simple generic local search template parameterized by the objective cost function f, the neighborhood N, as well as the functions L and S specifying legal moves and selecting the next neighbor and for different initial solutions s_0.

The search starts from any initial solution s and stores that as the best solution found so far (line 2) and performs a number of iterations (line 3). Line 4 checks if the

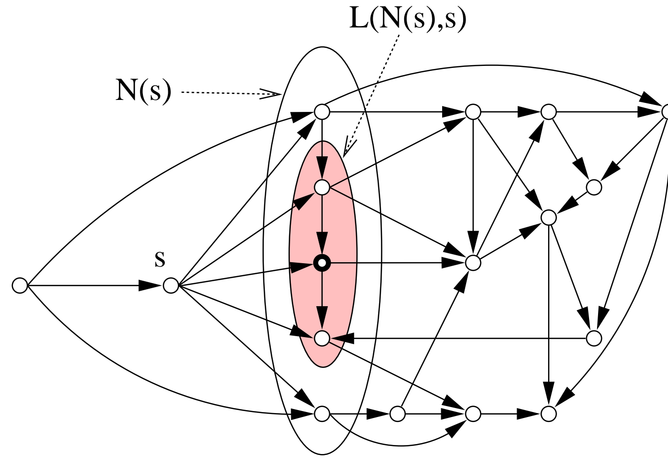


Figure 3.3: The local search move showing the solution s, its neighborhood $N(s)$, its set $L(N(s), s)$ of legal moves and the selected solution in bold thick circle.

Algorithm 1: Generic local search algorithm.

Input: Objective Function f, Neighborhood N, Function L, Function S, Initial Solution s_0

Output: Best Solution s_{best}

```
1 begin function LocalSearch(f, N, L, S, s0)
2 |   s_best := s0;
3 |   for i := 1 to MaxTrials do
4 |   |   if satisfiable(s_i) ∧ f(s_i) < f(s_best) then
5 |   |   |   s_best := s_i;
6 |   |   s_{i+1} := S(L(N(s_i), s_i), s_i);
7 |   end
8 |   return s_best;
9 end
```

solution satisfies the constraints and if there exists a neighbor solution s_i better than s_{best}. If there exists such solution then it stores s_i to the new best solution s_{best} (line 5) to always keep track of the best solution encountered so far. The move is set to action in line 6 and consists of selecting a new solution by composing operations N, L and S.

As you may have noticed, there are different implementations of local search with very simple changes of some of the N, L and S operations. For example, some local search algorithms may set all moves to legal or some to forbidden and in others, these operations may be rather complex and rely on sophisticated data structures and algorithms as well as on randomizations. In some cases the local search algorithm is executed from several different starting points to *diversify* the the search for global optimum. In cases like these we must decide on how many starting points we need to try and how to distribute them in the search space $\widehat{\mathcal{L}_{\mathcal{P}}}$.

3.4.2 Hill Climbing

Hill climbing is a relatively simple optimization technique that belongs to the family of local search. It is actually the basic form of local search. The technique is easy to implement, but the lack of sophistication makes more advanced algorithms a better choice. There are some situations where hill climbing is preferred because of its simplicity and it is widely used in the fields of *artificial intelligence* (AI) [12] [13].

There are various optimization techniques of the hill climbing search algorithm like the *steepest ascend* (also known as *gradient ascend*) and the *steepest descent* (also known as *gradient decent*) that goes back all the way to the famous mathematician Carl Friedrich Gauss. Actually the term “hill climbing” should strictly be applied only to maximization problems. Techniques for minimization problems should be identified as “valley descending”. However a simple reversal of sign converts a minimization problem into a maximization problem. It is customary to use the “hill climbing” term to cover both situations.

The steepest ascent technique normally starts with a random solution in the search space and sequentially moves in the direction of increasing value, that is uphill. At each step, it makes small changes to the current solution at hand by replacing it with the neighbor if the its solution is better, and each time improving the best solution found so far a little bit more. Heuristics can be used for choosing from, among the best solutions when more than one exists. A certain degree of randomness is often found when choosing among the best solutions. The hill climbing algorithm terminates when it reaches a “peak”.

To further understand local search and hill climbing, we need to consider the *search space landscape* as shown in Figure 3.4. For example, we know from [14], that the traveling salesman problems have a search space landscape called a *rugged landscape* that would make hill climbing a very inefficient algorithm to use on such a problem. A landscape have both *location*, defined by a *state*[6] and *elevation*, defined by the value of the heuristic cost function or objective function. If the elevation corresponds to cost, then the goal is to find the lowest valley - a global minimum. If elevation corresponds to an objective function, then the aim is to find the highest peak - a global maximum. Local search and hill climbing algorithms explore this landscape.

Hill climbing is sometimes called *greedy local search* because it grabs a good neighbor solution without thinking ahead about where to go next. But it often makes vary rapid

[6]A state is discrete mathematical object such as a permutation (that maps to a solution).

progress towards a solution, because it is usually quite easy to improve a bad solution in the beginning of a search, assuming we start from a relatively bad one. Unfortunately, hill climbing often finds a local optimum, or gets "stuck" in ridges[7] or plateau[8] as shown in Figure 3.4. In each case, the algorithm reaches a point where no progress is being made.

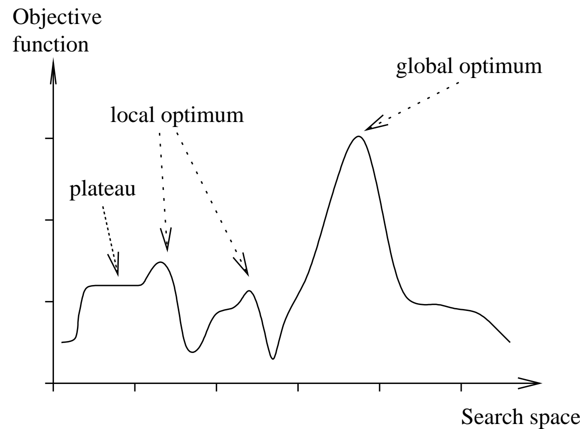


Figure 3.4: The possible landscape of a search space and problems that may occur for the hill climbing algorithm. The algorithm can get stuck in plateaus or local optimum.

3.4.3 Local Improvement Procedures

Heuristics focus on choosing the next neighbor to move to, according to some other heuristics, strategies, functions, procedures or methods. They are called *local improvement procedures* and deals with the problem of converging to a local optimum, or reaching a plateau, or a ridge. Although the procedures are not guaranteed to always succeed, because the optimum it finds depends on where the procedure begins the search. The procedure will find the global optimum only if it happens to begin the search in the neighborhood of this global optimum.

Heuristics can be classified in various ways, depending on the procedures including which changes they allow for the objective value and whether they are systematic or randomized. They are typically specified by instantiations of the selection function S, which can be combined with some restrictions on the neighborhood defined by the function L. For example, some procedures consists on choosing the neighbor with the best evaluation. The heuristic is called the best-neighbor heuristic as shown in Algorithm 2 with a random selected best neighbor to break ties.

Some heuristics require the procedures to consider all neighbors with no restrictions, in which case the L implementation becomes the function in Algorithm 3.

Improvement heuristics like hill climbing, can be formulated in terms of the following implementation of legal moves as shown in Algorithm 4.

[7]A ridge result in a sequence of local maxima that is very difficult for greedy algorithms to navigate.

[8]A plateau is an area of the search space landscape where the cost function or objective function is flat.

Algorithm 2: The best neighbor heuristic function S-Best(N,s).

Input: Neighborhood N, Solution s
Output: Best Neighbor n

1 **begin function** `S-Best(`N,s`)`
2 $\quad N_{best} := \{n \in N \mid f(n) = \min_{s \in N} f(s)\}$;
3 $\quad$ **return** $n \in N$ *with probability* $1/size_of(N)$;
4 **end**

Algorithm 3: The heuristic function L-All(N,s).

Input: Neighborhood N, Solution s
Output: Neighborhood N

1 **begin function** `L-All(`N,s`)`
2 $\quad$ **return** N;
3 **end**

A best improvement local search can then be specified as the following implementation (Algorithm 5) of the generic local search using the functions previously defined.

The *metropolis heuristic* is a random walk local improvement procedure based on randomness, in which some moves that doesn't improve the objective function are allowed. It selects a random neighbor n. If that neighbor improves the current solution, that is if $f(n) \leq f(s)$, it is accepted as the next solution. If it does not improve the objective function, the metropolis heuristic accepts the move with a small probability:

$$exp\left(\frac{-(f(n) - f(s))}{t}\right) \tag{3.20}$$

that depends on the distance between $f(n)$ and $f(s)$ and a parameter t called the temperature. Otherwise it rejects n. The metropolis heuristic is specified by Algorithm 6

Other procedures define different variants of local search, such as *stochastic hill climbing* which chooses at random from among the uphill moves. The probability of selection can vary with the steepness of the uphill move. This usually converges more slowly than steepest ascent, but in some cases it finds better solutions.

Random restart hill climbing conducts a series of hill climbing searches from randomly generated initial solutions, stopping when a good solution is found. The *restart method* can restart the local improvement procedure a number of times from randomly

Algorithm 4: The heuristic function L-Improvement(N, s).

Input: Neighborhood N, Solution s
Output: Neighbor n

1 **begin function** `L-Improvement(`N,s`)`
2 $\quad$ **return** $n \in \{N \mid f(n) < f(s)\}$;
3 **end**

Algorithm 5: The heuristic function L-BestImprovement(s).

```
  Input: Solution s
1 begin function L-BestImprovement(s)
2 |  return LocalSearch(f,N,L-Improvement,s_best)
3 end
```

Algorithm 6: The heuristic function S-Metropolis[t](N, s).

```
  Input: Temperature t, Neighborhood N, Solution s,
  Output: Solution s
1 begin function S-Metropolis(N,s)
2 |  choose n ∈ N with probability 1/size_of(N);
3 |  if f(n) < f(s) then
4 |  |  return n;
5 |  else
6 |  |  with probability exp((-(f(n)-f(s)))/t);
7 |  |  return n;
8 |  end
9 end
```

selected initial trial solutions. This will lead to a new local optimum. If we repeat restarting the search a number of times, we increase the chance that the best of the local optimum obtained will actually be the global optimum. This approach works well on small instances and but is much less successful on larger instances with many variables and a rugged landscape with complicated feasible regions.

The local search ideas need to be carefully tweaked and tailored to fit the problem. Each method is usually designed to fit a specific problem type rather than a variety of applications. This means that we always have to implement it from scratch each time we want to develop a heuristic method to a specific problem. To overcome the drawbacks of local improvement procedures, we need sophisticated strategies and procedures with a more structured approach that uses the information being gathered to guide the search toward the global optimum. This is the role that a *metaheuristic* plays.

3.5 Metaheuristics

The term *metaheuristic* is also written meta-heuristic, where "meta" is the Greek prefix which means "beyond". It was first introduced in the same paper [15] that also introduced the term *tabu search* [16]. A metaheuristic is a general solution heuristic method with a master strategy that guides and modifies other heuristics. The method collects information on the execution stages and aims primarily at escaping local minima and at directing the search towards global optimality to produce feasible solutions beyond those that are normally generated in a search for local optimality. This is fundamentally quite different from a heuristic method which focuses on choosing the next solution from the neighborhood using only local information on the quality of the neighbors. Heuristics typically drive the search toward high quality local minima. As

a consequence, heuristics are often characterized as memoryless, while metaheuristics typically include some form of memory or learning which explains the great diversity and the wealth of results in this field.

3.6 Metaheuristic Methods

A few popular metaheuristic local search algorithms for solving combinatorial optimization $\mathcal{NP}$-hard and problems include *iterated local search* [17] which iterates a specific local search from different starting points in order to sample various regions of the search space and to avoid returning a low quality local minimum. *Evolutionary algorithms* which uses improvement procedures inspired by biological evolution such as reproduction, mutation, recombination, natural selection and survival of the fittest. Evolutionary algorithms maintain a pool of solutions that are combined to produce new solutions fitted for parallel processor usage to diversify the search and escaping local minima. *Tabu search* [16] which uses a type of memory, basically a tabu list of forbidden moves to avoid cycling behavior while searching. Other examples shown below include *greedy algorithms*, the metaheuristic *simulated annealing* which is based on the Metropolis heuristic given previously and a *hybrid evolutionary algorithm* which combines local search and *genetic algorithm* operators.

3.6.1 Greedy Algorithms and Greedy Satisfiability

Greedy Algorithms

Most algorithms for optimization problems go through a sequence of steps, with a set of choices at each step. This behavior applies to *greedy algorithms*. A *greedy algorithm* is any algorithm that follows the problem solving metaheuristic at obtaining a locally optimal solution to a problem by making a sequence of choices at each stage with the hope of leading to a global optimal solution. For each decision point in the algorithm, the choice that looks best at the moment is chosen. The choice made on each step must fulfill the following three properties: *feasible* – it has to satisfy the problems constraints, *locally optimal* – it has to be the best local choice among all feasible choices available on that step and *irrevocable* – once made, it cannot be changed on subsequent steps of the algorithm. Greedy algorithms are very powerful and work quite well for a wide range of problems, but they do not always yield optimal solutions. Examples of greedy algorithms are *Prim's algorithm* and *Kruska's algorithm* [18] for finding the minimum spanning tree [19] [20], *Dijkstra's algorithm* [18] for finding single source shortest paths, and *Chvátal's greedy heuristic* [21] for the set covering problem.

Greedy Satisfiability

Greedy satisfiability (GSAT) [22] is a greedy local search procedure operating on a propositional logic formula in conjunctive normal form (CNF). It is based on the main idea of minimizing the number of unsatisfied clauses. The algorithm was first developed in 1989 by Gu and independently by Bart Selman in 1992, who called it GSAT and showed that it was capable of solving a whole range of very hard problems very quickly.

3.6.2 Simulated Annealing

Simulated annealing is another metaheuristic that enables the search process to escape local optimum. It is based on the Metropolis algorithm with a sequence of decreasing temperatures starting with a relatively large value of t. A large value of t makes the acceptance of a solution relatively large, which enables the search to proceed in almost random directions. Gradually decreasing the value of t as the search continuous decreases the probability of acceptance. Thus the choice of the values of t over time controls the degree of randomness in the process. This random component provides more flexibility for moving toward another part of the feasible region in the hope of finding a global optimum. The template for simulated annealing is shown in Algorithm 7.

Algorithm 7: The metaheuristic function `SimulatedAnnealing`(f,N).

Input: Function f, Neighborhood N
Output: Solution s_{best}

```
1  begin function SimulatedAnnealing(f,N)
2      s_0 := GenerateInitialSolution();
3      t_0 := InitTemperature(S);
4      s_best := s_0 ;
5      for i := 0 to MaxIterations do
6          s_best := LocalSearch(f,N,L-All,S-Metropolis[t_i],s_i);
7          if f(s_i) < f(s_best) then
8              s_i := s_best;
9          end
10         t_{i+1} := UpdateTemperature(s, t_i) ;
11     end
12     return s_best;
13 end
```

The local search algorithm in line 6 is a Metropolis algorithm with temperature t_i. Line 3 specifies the starting temperature and line 10, the cooling schedule which specifies how to decrease the temperature. Both of these two critical decision can be chosen experimentally or adapted to specific instances by performing random walks at different temperatures.

3.6.3 Hybrid Evolutionary Algorithm

In the real world, the process of natural selection controls evolution. *Evolutionary algorithms* or *genetic algorithms* are greatly influenced by the biological *theory of evolution* [23] formulated by Charles Darwin. The ideas taken from the theory of evolution are transfered over to dealing with optimization problems in a natural way. Members of a particular *species* correspond to feasible solutions where each member is measured by the value of the objective function. Rather than processing a single trial solution at a time, as we do with basic forms of simulated annealing (and tabu search), we now work with an entire *population* of trial solutions. For each iteration (*generation*) of the algorithm, the current population consists of the set of trial solutions currently under consideration. They are thought of as the currently living members of the species.

Some of the members are randomly paired and become *parents* who then have *children* (new trial solutions) who share some of the new genes of both parents. As the algorithm proceeds the fittest members of the population generate improving populations of trial solutions. When parents reproduce, there is no guarantee that resulting children will be perfect and occasionally a *mutations* occurs. This happens with a random process and helps the algorithm to explore a new, perhaps better parts of the feasible region.

A Hybrid Algorithm

To further increase the chance to find high quality solutions, we can combine different algorithms into a *hybrid algorithm*. The *hybrid evolutionary algorithm* exploit the strength of both local search and evolutionary search by applying local search to the solutions in the population before combining them. We benefit from the effectiveness of local search in finding high quality solutions, while the evolutionary aspects, provide novel ways of diversifying the search and escaping local minima. The hybrid local search algorithm is depicted in Algorithm 8. It starts by generating an initial population

Algorithm 8: The metaheuristic hybrid evolutionary search algorithm `HybridLocalSearch(`f,N,L,S`)`.

Input: Function f, Neighborhood N, Function L, Function S
Output: Solution s_{best}

```
1  begin function HybridLocalSearch(f,N,L,S)
2      P := InitPopulation(f,N,L,S);
3      s_best := select s ∈ P minimizing f(s);
4      for i := 0 to MaxIterations do
5          select s_1, s_2 ∈ P ;
6          s := Crossover(s_1, s_2);
7          s := LocalSearch(f,N,L,S,s);
8          if f(s) < f(s_best) then
9              s_best := s;
10         end
11         P := UpdatePopulation(s, s_1, s_2, P);
12     end
13     return s_best;
14 end
15 begin function InitPopulation(f,N,L,S)
16     P := {};
17     for j := 0 to MaxPopulation do
18         s := GenerateInitialSolution();
19         s := LocalSearch(f,N,L,S,s);
20         P := P ∪ {s};
21     end
22     return P;
23 end
```

$\mathcal{P}$ of solutions (line 2 or line 16) obtained by randomly generating a solution in which local search is applied (line 17). In line 5, we select two solutions (the parents) from the population, and then cross them (in line 6) to obtain a new solution s (a child).

We then apply local search to find an improved solution (line 7), and update in line 11 the population using s and its parents.

The heuristics and metaheuristics presented so far can all be applied to solve $\mathcal{NP}$-hard and $\mathcal{NP}$-complete combinatorial optimization problems like the traveling salesman problem. Another important example is the fundamental optimization problem called *quadratic assignment problem* which concerns the placement of facilities in order to minimize transportation costs, avoid placing hazardous materials near housing, and saving lives under the ambulance location problem in a big city by minimize the path from an ambulance to a patient in need.

Chapter 4

Network Optimization Problems

Network problems arise in numerous ways. It might be a physical network such as transportation in a road system, an electrical network or a communication network of telephone lines. In fact, network representations are widely used for problems in areas as production, distribution, project planning, and facilities location to name just a few examples. In all cases, network representation provides a very powerful visual aid. Table 4 show some examples of typical networks.

Nodes	Arcs	Flow
Airports	Air lines	Aircraft
Intersections	Roads	Vehicles
Pumping stations	Pipes	Fluids
Switching points	Wires, channels	Messages

Table 4.1: Components of some typical networks in todays modern society.

4.1 Network Flow Problem Terminology

In graph theory, a network flow is a directed graph where the maximum amount of *flow* that can be carried on a directed vertices is referred to as the *edge capacity*. Each edge receives a flow where the maximum amount of flow may not exceed the capacity of the edge. In *operations research*, a directed graph is often called a *network*, the vertices are called *nodes*, the edges are called *arcs* and the edge capacity is called *arc capacity*. For nodes, a distinction is made among those that are net generators of flow, net absorber of flow, or neither. A *supply node* (a.k.a. source node or source) has the property that the flow out of the node exceeds the flow into the node. The reverse case is called a *demand node* (a.k.a. sink node or sink), where the flow into the node exceeds the flow out of the node. A *transshipment node* (a.k.a. intermediate node) satisfies *conservation of flow*, i.e, flow in equals flow out.

4.2 The Minimum Cost Network Flow Problem

The *minimum cost network flow problem* (MCNFP) is to send flow from a set of supply nodes, through the arcs of a network, to a set of demand nodes, at minimum total cost, and without violating the lower and upper bounds on flows through the arcs.

Let $G = (V, E)$ be a directed graph consisting of a finite set of nodes $V = \{1, 2, \ldots, n\}$, and a set of directed arcs, $E = \{1, 2, \ldots, m\}$, linking pairs of nodes in V. We associate with every arc of $(i, j) \in E$, a flow x_{ij}, a cost per unit flow c_{ij}, a lower bound on the flow l_{ij} and a capacity u_{ij}. To each node $i \in V$ we assign an integer number b_i representing the available supply of, or demand for flow at that node. If $b_i > 0$ then node i is a supply node, if $b_i < 0$ then node i is a demand node, and otherwise, if $b_i = 0$, node i is referred to as a transshipment node. The total supply must equal the total demand.

The minimum cost network flow problem $N = (i, j, V, E, b)$ is to determine the flows $x_{ij} \geq 0$ in each arc of the network so that the net flow into each node i is b_i while minimizing the total cost. In mathematical terms, the linear programming formulation of the problem becomes

$$\min \sum_{i=1}^{n} \sum_{j=1}^{n} c_{ij} x_{ij} \tag{4.1}$$

subject to

$$\sum_{j=1}^{n} x_{ij} - \sum_{j=1}^{n} x_{ji} = b_i \qquad \text{for each node } i, \tag{4.2}$$

and

$$0 \leq x_{ij} \leq u_{ij} \qquad \text{for each arch } i \to j. \tag{4.3}$$

The decision variables for the linear programming formulation are

$$x_{ij} \quad = \quad \text{flow through arc } i \to j \tag{4.4}$$

with the given information

$$\begin{aligned} c_{ij} &= \text{cost per unit flow through arc } i \to j, && (4.5) \\ u_{ij} &= \text{arc capacity for arc } i \to j, && (4.6) \\ b_i &= \text{net flow generated at node } i. && (4.7) \end{aligned}$$

The last value b_i has a sign convention that depends on the nature of the node i, where

$$\begin{aligned} b_i &> 0 \qquad \text{if node } i \text{ is a supply node,} && (4.8) \\ b_i &< 0 \qquad \text{if node } i \text{ is a demand node,} && (4.9) \\ b_i &= 0 \qquad \text{if node } i \text{ is a transshipment node.} && (4.10) \end{aligned}$$

4.3 The Transportation Problem

The *transportation problem* is special case of the minimum cost network flow problem. Every node in the network is either a source or a sink, and every arc goes from a source node to a sink node. A supply node is provided for each source, as well as a demand

node for each destination, but no transshipment nodes are included in the network. The conservation constraints have one of the following forms

$$\sum_{j=1}^{n} x_{ij} = b_i \tag{4.11}$$

for a source with $b_i > 0$ or

$$-\sum_{j=1}^{n} x_{ji} = b_i \tag{4.12}$$

for a sink with $b_i < 0$. The transportation problem is used to model the movement of goods from suppliers to costumers with some cost associated to the shipments.

4.4 The Assignment Problem

The assignment problem is a special case of the transportation problem where $b_i = 1$ for a source and $b_i = -1$ for a sink. It is an optimization model for assigning n people or workers to n jobs. If worker i is assigned to a particular job j, there is a benefit of the cost coefficient c_{ij} indicating the value of a person based on their education, experience or skill requirements of the job. Each worker must be assigned to exactly one job, and each job must have one assigned worker.

The general formulation of the assignment problem is to find x_{ij} to

$$\min \sum_{i=1}^{n} \sum_{j=1}^{n} c_{ij} x_{ij}, \tag{4.13}$$

subject to

$$\sum_{j=1}^{n} x_{ij} = 1 \qquad \text{for each worker (node) } i, \tag{4.14}$$

$$\sum_{i=1}^{n} x_{ij} = 1 \qquad \text{for each job } j, \tag{4.15}$$

and

$$x_{ij} \in \{0, 1\} \qquad \text{for each arch } i \to j. \tag{4.16}$$

The variables x_{ij} must take the value 0 or 1, otherwise the solution is not meaningful because it is not possible to make fractional assignments of a person to a job.

4.5 The Quadratic Assignment Problem

The *quadratic assignment problem* (QAP) was introduced by Koopmans and Beckmann [24] back in 1957 to model a plant location problem. Quadratic assignment problems are still considered attractive and important in theory and practice. The number of real life problems that are mathematically modeled by a quadratic assignment problem has been continuously increasing even today. There are a number of well known combinatorial optimizations problems that can be formulated as quadratic assignment problems. Typical examples are the traveling salesman problem and a large number

of optimizations problems in graphs such as the maximum clique problem, the graph partitioning problem and the minimum feedback arc set problem.

From a computational point of view, quadratic assignment problems are very difficult problems. They are actually $\mathcal{NP}$-hard combinatorial optimization problems. It is proven in [25] that QAP is $\mathcal{NP}$-hard and even with todays fast multi-core CPUs, it is still considered hard to solve problems of a modest size as $n = 30$ within reasonable time limits.

The QAP can be described as the problem of assigning a set of facilities to a set of locations with given distances between the locations and given flows between the facilities. The goal then is to place the facilities on locations in such a way that the sum of the product between flows and distances is minimal.

4.5.1 Mathematical Formulation of the QAP

There are several similar formulations of the QAP in the literature. Each formulation pose different characteristics of the problem and lead to different solution approaches.

The Koopmans-Beckman version of the QAP of size n can be formulated as follows. Given a set $\mathcal{N} = \{1, 2, \ldots, n\}$ and three $n \times n$ matrices $A = (a_{ij})_{n\times n}$, $B = (b_{kl})_{n\times n}$, and $C = (c_{ik})_{n\times n}$. Let $\Pi_{\mathcal{N}}$ be the set of all permutations of $\mathcal{N}$. The objective is to find a permutation $p = (p(1), p(2), \ldots, p(n)) \in \Pi_{\mathcal{N}}$ such that

$$\min_{p\in\Pi_{\mathcal{N}}} f(p) = \sum_{i=1}^{n}\sum_{j=1}^{n} a_{ij} b_{p(i)p(j)} + \sum_{i=1}^{n} c_{ip(i)}. \tag{4.17}$$

The QAP can also be formulated as a "0-1" integer optimization problem[1] The term "quadratic" actually comes from the formulation of the problem as an optimization problem with a quadratic objective function. From a mathematical point of view, an assignment is a one-to-one correspondence (i.e. a bijection) of the finite set $\mathcal{N}$ into itself, in the sense that, permutation p assigns some $j = p(i)$ to each $i \in \mathcal{N}$. Every permutation p of the set $\mathcal{N}$ corresponds uniquely to a permutation matrix $X_p = (x_{ij})_{n\times n}$ with $x_{ij} = 1$ for $j = p(i)$ and $x_{ij} = 0$ for $j \neq p(i)$. The entries of such permutation matrix can be defined as a matrix that must satisfy the assignment constraints similar to Equation 4.14, Equation 4.15 and Equation 4.16 in the following way:

$$\sum_{j=1}^{n} x_{ij} = 1, \qquad \text{for all } i = 1, \ldots, n \tag{4.18}$$

$$\sum_{i=1}^{n} x_{ij} = 1, \qquad \text{for all } j = 1, \ldots, n \tag{4.19}$$

$$x_{ij} \in \{0, 1\}, \qquad \text{for all } i, j = 1, \ldots, n \tag{4.20}$$

and the clarification that

$$x_{ij} = \begin{cases} 1 & \text{if facility } i \text{ is assigned to location } j \\ 0 & \text{otherwise.} \end{cases} \tag{4.21}$$

[1] "0-1" optimization problems with integer coefficients are special cases of integer programming, where variables are required to be 0 or 1.

With the above constraints on x, we have our equivalent formulation of the QAP in terms of permutation matrices $QAP(A,B,C)$

$$\min \sum_{i=1}^{n}\sum_{j=1}^{n}\sum_{k=1}^{n}\sum_{l=1}^{n} a_{ij}b_{kl}x_{ik}x_{jl} + \sum_{i=1}^{n}\sum_{j=1}^{n} c_{ij}x_{ij}. \tag{4.22}$$

4.5.2 Location Theory

One of the major applications of the QAP is in *location theory.* Given a set of n locations and n facilities. Each facility is assigned to a location. The size of an instance of the QAP then becomes n with a total amount of $n!$ possible assignments of locations to facilities. Let matrix $F = (f_{ij})$ be the flow matrix, i.e., f_{ij} is the flow of materials from facility i to facility j, and let matrix $D = (d_{kl})$ be the distance matrix, i.e., d_{kl} represents the distance from location k to location l, and let matrix $C = (c_{ik})$ be the cost matrix, where c_{ik} is the cost of placing facility i at location k. The cost of assigning facility i to location k and facility j to location l is $f_{ij}d_{kl}$. The main goal is to find an assignment (i.e., a permutation $p \in \Pi_{\mathcal{N}}$) of all facilities to all locations, such that the total cost of the assignment is minimized

$$\min_{p\in\Pi_{\mathcal{N}}} f(p) = \sum_{i=1}^{n}\sum_{j=1}^{n} f_{ij}d_{p(i)p(j)} + \sum_{i=1}^{n} c_{ip(i)} \tag{4.23}$$

in the same way as in Equation 4.17.

An Example of a Combinatorial Optimization Problem

We are given an instance of size $n = 4$ to a quadratic assignment problem. The total number of permutations or arrangements for assigning facilities to locations are $P(4,4) = 4! = 24$. All the 24 possible permutations are generated with the following assignments.

1. (1,2,3,4)	7. (3,4,2,1)	13. (4,3,2,1)	19. (1,2,4,3)
2. (4,1,2,3)	8. (1,3,4,2)	14. (3,2,1,4)	20. (2,4,3,1)
3. (3,4,1,2)	9. (1,3,2,4)	15. (2,1,4,3)	21. (4,2,3,1)
4. (2,3,4,1)	10. (4,1,3,2)	16. (1,4,3,2)	22. (2,3,1,4)
5. (2,1,3,4)	11. (2,4,1,3)	17. (4,3,1,2)	23. (3,1,4,2)
6. (4,2,1,3)	12. (3,2,4,1)	18. (3,1,2,4)	24. (1,4,2,3)

Permutations 13 to 24 are the mirror version of permutations 1 to 12, meaning that we simply reverse the index of permutations 1 to 12 to generate permutations 13 to 24.

We start by randomly picking one possible solution to the quadratic assignment problem or facility location problem with four facilities. We pick permutation number 15 with the assignment $(2,1,4,3)$. This means that the index of the permutation corresponds to the location while the value in the permutation corresponds to the facility as shown in Figure 4.1 and in the following way.

- To location 1, we assign facility 2.
- To location 2, we assign facility 1.
- To location 3, we assign facility 4.

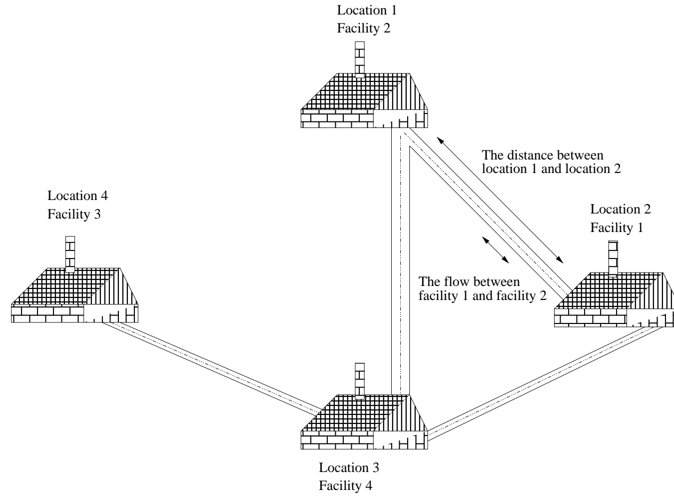


Figure 4.1: One possible solution to QAP with four facilities.

- To location 4, we assign facility 3.

We are also given in advance the flow matrix $F = (f_{ij})_{n\times n}$ between facilities

$$F = (f_{ij})_{4\times 4} = \begin{bmatrix} f_{11} & f_{12} & f_{13} & f_{14} \\ f_{21} & f_{22} & f_{23} & f_{24} \\ f_{31} & f_{32} & f_{33} & f_{34} \\ f_{41} & f_{42} & f_{43} & f_{44} \end{bmatrix} = \begin{bmatrix} 0 & 3 & 0 & 2 \\ 3 & 0 & 0 & 1 \\ 0 & 0 & 0 & 4 \\ 2 & 1 & 4 & 0 \end{bmatrix} \tag{4.24}$$

and the distance matrix $D = (d_{kl})_{n\times n}$ between locations.

$$D = (d_{kl})_{4\times 4} = \begin{bmatrix} d_{11} & d_{12} & d_{13} & d_{14} \\ d_{21} & d_{22} & d_{23} & d_{24} \\ d_{31} & d_{32} & d_{33} & d_{34} \\ d_{41} & d_{42} & d_{43} & d_{44} \end{bmatrix} = \begin{bmatrix} 0 & 22 & 53 & 0 \\ 22 & 0 & 40 & 0 \\ 53 & 40 & 0 & 55 \\ 0 & 0 & 55 & 0 \end{bmatrix} \tag{4.25}$$

The assigned cost f (with cost matrix $C = 0$) of the randomly chosen permutation with

$$p(1) = 2 \tag{4.26}$$
$$p(2) = 1 \tag{4.27}$$
$$p(3) = 4 \tag{4.28}$$
$$p(4) = 3 \tag{4.29}$$
$$\tag{4.30}$$

becomes

$$\sum_{i=1}^{n}\sum_{j=1}^{n} f_{ij} d_{p(i)p(j)} =$$

$$\begin{aligned}
&(f_{11} \times d_{p(1)p(1)} + f_{12} \times d_{p(1)p(2)} + f_{13} \times d_{p(1)p(3)} + f_{14} \times d_{p(1)p(4)} \\
&+f_{21} \times d_{p(2)p(1)} + f_{22} \times d_{p(2)p(2)} + f_{23} \times d_{p(2)p(3)} + f_{24} \times d_{p(2)p(4)} \\
&+f_{31} \times d_{p(3)p(1)} + f_{32} \times d_{p(3)p(2)} + f_{33} \times d_{p(3)p(3)} + f_{34} \times d_{p(3)p(4)} \\
&+f_{41} \times d_{p(4)p(1)} + f_{42} \times d_{p(4)p(2)} + f_{43} \times d_{p(4)p(3)} + f_{44} \times d_{p(4)p(4)})/2
\end{aligned}$$

$$\begin{aligned}
&(f_{11} \times d_{22} + f_{12} \times d_{21} + f_{13} \times d_{24} + f_{14} \times d_{23} \\
&+f_{21} \times d_{12} + f_{22} \times d_{11} + f_{23} \times d_{14} + f_{24} \times d_{13} \\
&+f_{31} \times d_{42} + f_{32} \times d_{41} + f_{33} \times d_{44} + f_{34} \times d_{43} \\
&+f_{41} \times d_{32} + f_{42} \times d_{31} + f_{43} \times d_{34} + f_{44} \times d_{33})/2
\end{aligned}$$

$$\begin{aligned}
&=(0 \times 0 + 3 \times 22 + 0 \times 0 + 2 \times 40 \\
&+3 \times 22 + 0 \times 0 + 0 \times 0 + 1 \times 53 \\
&+0 \times 0 + 0 \times 0 + 0 \times 0 + 4 \times 55 \\
&+2 \times 40 + 1 \times 53 + 4 \times 55 + 0 \times 0)/2 \\
&=\frac{838}{2} \\
&=419
\end{aligned}$$

We divided the right side of the equation above by two because the cost between each location and facility is calculated twice. Looking at Figure 4.1, we can keep track of the direction of the flows, ignore the zero flows, the zero distances and the loops. In this way our calculation becomes a bit more compact thanks to the symmetry in the flow and distance matrices. This actually helps to reduce the time to obtain a single solution.

$$\begin{aligned}
\sum_{i=1}^{n}\sum_{j=1}^{n} f_{ij} d_{p(i)p(j)} &= f_{12} \times d_{21} + f_{14} \times d_{23} + f_{24} \times d_{13} + f_{34} \times_{43} \\
&= 3 \times 22 + 2 \times 40 + 1 \times 53 + 4 \times 55 \\
&= 419.
\end{aligned}$$

This is not the best possible permutation. By trying out the rest of the 23 permutations you might find it though. Note that because of the symmetries or the undirected graph in Figure 4.1, you only need to check 12 permutations if you include the one we just solved. The major concern when dealing with larger instances, is that the number of permutations grow extremely fast in the same way as with the traveling salesman problem. This means that it is very time consuming to generate and calculate every single permutation. This is one of the reasons why different algorithms have been developed for QAPs.

Other examples of some problems expressed in terms of QAP include: *Computer Aided Design (CAD)*, more precisely, the placement of logical modules in a chip such

that the total length of the connections on a board (chip) is minimized. In this case, a_{ij} is the number of connections between electronic module i and module j and b_{kl} is the distance between locations k and l on which modules can be placed. *The assignment of specialized rooms in a building*, where a_{ij} is the flow of people that go from service i to service j and b_{kl} is the time for going from room k to room l. *The assignment of gates to airplanes in an airport*, where a_{ij} is the number of passengers going from airplane i to airplane j and b_{kl} is the walking distance between gates k and l.

The algorithm fitted for solving these types of problems is the one we have developed for the QAP. It is a simple metaheuristic algorithm called *parallel tabu search*. We mainly test it to solve problems taken from the *quadratic assignment problem library* on the Internet.

4.5.3 A Quadratic Assignment Problem Library

The *quadratic assignment problem library* (QAPLIB) [26] was first published in 1991, with the main purpose of providing a unified test bed for QAP to the scientific community. The QAPLIB home page[2] include all the QAP instances generated by several researches with a list of best known feasible solutions, best lower bounds, and most of the current best known permutations.

The problem instances are of size $n > 12$, quadratic with the cost matrix $C = 0$ and in most cases symmetric, all listed in alphabetical order by the names of their authors or contributors. The file format of the problem data with the extension ".dat" is

$$\begin{array}{c} n \\ A \\ B \end{array}$$

where n is the size of the instance, A and B correspond to the flow and distance matrices. The format of the solution file for available permutations corresponding to the feasible solutions have extension ".sln" with the file format

$$\begin{array}{l} n \ sol \\ p \end{array}$$

where n is the size of the instance, sol is the objective function value (or solution) and p is the corresponding permutation, i.e.

$$sol = \sum_{i=1}^{n} \sum_{j=1}^{n} f_{ij} d_{p(i)p(j)}. \tag{4.31}$$

Our work is based on solving the quadratic assignment problem called the "Microarray Placement Problem" taken from the QAPLIB home page. We use our own implementation of a parallel version of tabu search, with some unique local improvement procedures and a communication system based on graphs.

[2]http://www.opt.math.tu-graz.ac.at/qaplib/

Chapter 5

Parallel Tabu Search Algorithm

The word *tabu* (or taboo) means "a prohibition imposed by social custom as a protective measure" - Webster's dictionary. The word comes from a language of Polynesian called Tongan and is used by the natives of Tonga island to indicate sacred things that cannot be touched.

Tabu search was originally proposed by Fred Glover [16] in 1986 to allow local search methods to overcome local optima. It is a widely used metaheuristic to solve combinatorial optimization problems that classical optimization methods have great difficulty solving within practical time limits. The algorithm can be viewed as a deterministic alternative to simulated annealing, in which memory, rather than probability, guides the intelligent search. It uses some common sense ideas like a short term memory to enable the search process to escape from a local optimum. This yields solutions whose quality often significantly surpasses that obtained by other methods, typically within 1% of the best known solution found in the litterature.

5.1 Tabu Search for Combinatorial Optimization Problems

Let $S = \{s_1, s_2, s_3, \dots\}$ be the set of solutions for our QAP (or any combinatorial optimization problem) with the objective function $f : S \to \mathbb{R}^1$. The neighborhood function $N : S \to 2^S$ defines for each $s \in S$ a set $N(s) \subseteq S$, which is a set of neighboring solutions of s. Each solution s' can be reached from s by an operation called *move*.

Tabu search begins as an ordinary local search or neighborhood search and continuous iteratively from one solution to another until a chosen termination criterion is met. The local search procedure operates just like a local improvement procedure by only accepting an improved solution. At each iteration, the algorithm uses the move operator from one solution point s to another solution s' in the neighborhood $N(s)$ of s. The move that improves most the objective function value f is chosen. If there are no improving moves, tabu search chooses one that least degrades the objective function, i.e., a move is performed to the best neighbor $s' \in N(s)$ even if $f(s') > f(s)$.

Just like in the ascent method, each iteration selects the available move that goes furthest up the hill, or a move that drops least down the hill if an upward move is not available in the hill climbing process. If all goes well, the process will follow a pattern like that shown in Figure 3.4. A local optimum is left behind in order to climb to the global optimum. The opposite applies when minimizing the objective function (cost function) with the descent method.

Tabu search can be viewed as the combination of a greedy strategy with a definition of legal moves ensuring that a solution is never visited twice. The generic local search in Algorithm 1 is modified to be consistent with handling a sequence of solutions as shown in Algorithm 9. It consists mainly of maintaining the sequence $\tau = \{s_0, s_1, \ldots, s_k\}$ of

Algorithm 9: Modified local search algorithm for tabu search.

Input: Objective Function f, Neighborhood N, Function L, Function S, Solution s_0
Output: Best Solution s_{best}

```
1  begin function LocalSearch(f, N, L, S, s_0)
2  |  s_best := s_0;
3  |  τ := {s_0};
4  |  for k := 1 to MaxTrials do
5  |  |  if satisfiable(s_k) ∧ f(s_k) < f(s_best) then
6  |  |  |  s_best := s_k;
7  |  |  s_{k+1} := S(L(N(s_k), τ), τ);
8  |  |  τ := τ :: s_{k+1}
9  |  end
10 |  return s_best;
11 end
```

solutions found so far. The sequence τ is declared and initialized with solution s_0 in line 3 and new solutions are sequentially explored and added to τ in line 8. Functions L and S are also modified to handle a sequence of solutions. Function S is similar to the heuristic function in Algorithm 2. The heuristic function L is implemented as *L-NotTabu* in Algorithm 10 and later called from the tabu search implementation in Algorithm 11.

Algorithm 10: The heuristic function L-NotTabu(N,τ).

Input: Neighborhood N, Solutions τ
Output: Neighbor n

```
1  begin function L-NotTabu(N,τ)
2  |  return {n ∈ N | n ∉ τ};
3  end
```

Different heuristic *search strategies* are usually implemented in tabu search to increase the chance to find the global optimum. Next section describes some typical search strategies applied to our implementation of parallel tabu search.

Algorithm 11: Tabu search algorithm.

Input: Objective Function f, Neighborhood N, Solution s_0
Output: Best Solution s_{best}

```
1 begin function TabuSearch(f, N, s0)
2 |  return TabuSearch(f, N, L-NotTabu, S-Best);
3 end
```

5.1.1 Short Term Memory

The danger with the hill climbing method in tabu search is that after moving away from a local optimum, the search process will cycle or return right back to the same local optimum. In order to avoid the cycling back to the same local optimal solution just visited after a non improving move, the reverse move must be temporarily forbidden. This is the reason for the term "tabu" in tabu search. It is impossible to keep track of all visited solutions because the memory requirements quickly become prohibited. Instead, we maintain only a small suffix of the recently visited solutions by storing solutions produced by moves in a short term memory. It is managed like a circular list $\tilde{\tau}$ called a *tabu list*. The tabu list contains n $(= |\tilde{\tau}|)$ recorded moves or solutions that have been visited in the recent past which are now temporarily forbidden. The use of memory to guide the search by implementing tabu lists to record some of the recent history is a distinct feature of tabu search borrowed from the field of artificial intelligence.

The length of the tabu list $\tilde{\tau}$ is called *tabu list length* or *tabu list size*. The list size is considered crucial. Cycling will occur if it is too small, whereas if it is too large, appealing moves will be forbidden. This will reduce the search of "promising regions", yielding low quality solutions and produce a large number of iterations to find the solution desired. The *optimal tabu list size* is highly related to a specific problem instance. Different problem instances require different optimal tabu list sizes. However, by randomly choosing from a set of tabu list sizes, we are able to create a dynamic tabu list length to partially overcome that problem.

5.1.2 Long Term Memory

The tabu search algorithm include only a small suffix of the solution sequence and cannot capture long term information. The consequence of a short term memory is that the search process will sometimes take long walks[1] whose solutions have low quality objective values or spend too much time in the same region, leaving other parts of the search space unexplored. It is sometimes fruitful to implement the tabu search algorithm with a long term memory structures to intensify and diversify the search.

Intensification involves exploring a portion of the feasible region more thoroughly than usual by storing high quality solutions during the search and returning to these solutions periodically. The simplest intensification scheme consists of returning to the best solution found so far while more sophisticated intensification criterion stores multiple solutions and explore all their legal neighbors.

[1] A search process that takes a long time or a vast amount of iterations and does little or no improvement on the objective function.

Diversification involves forcing or directing the search toward or into other unexplored regions of the search space. There are many ways to achieve such goal. One way is to perturb or *restart* the search when it reaches a predefined or random number of iterations. Another way, which we will not implement in our algorithm, include *strategic oscillation*. It consists of changing the objective function to balance the time spent in the feasible and infeasible regions.

5.2 Parallel Tabu Search

Parallel tabu search (PTS) [27], [28], [29] is a *parallel metaheuristic* that enables multiple tabu search algorithms to execute at the same time and on multiple CPUs. By doing this, we satisfy the demand for computational speed and gain a speedup compared to a single execution of tabu search. The parallel tabu search algorithm presented in this thesis was implemented to take advantage of the multi-core architecture and its exciting opportunities for solving combinatorial optimization problems.

Parallel metaheuristics and its applications [30] have a very young history because of the recent development of cheap parallel computers on desktop machines. Today, anyone with a new computer can easily learn and experiment with parallel programming. Researches have successfully solved hard combinatorial problems with the help of parallel programming, where parallel cooperative *solvers*[2] (sometimes referred to workers or agents) collaborate through a *communication strategy* by sending and receiving solutions to each other. This method provide heuristic guidance to speed up the search and increase the chance to find the global optimal value.

In this thesis we propose a simple *communication strategy*. We first describe how to exchange solutions between solvers according to a communication graph, then present two policies of treating solutions obtained from other solvers.

5.2.1 Communication Procedure

A solver runs a tabu search algorithm (could also be any other parallel metaheuristic algorithm) and interacts with other solvers by sending and receiving solutions according to a *communication strategy* based entirely on *communication graphs*.

Each solver holds an *elite set*, where every element in the set is of a high quality solution meaning that the solutions are the best ones found so far by a solver. An elite set enables a solver to keep track of high quality solutions, either found by itself or obtained from other solvers during communication. A solver can only communicate directly with its neighbors and use a communication graph to define a solvers' neighborhood. Each node in the undirected communication graph represents a solver or tabu search thread (or CPU) and each edge corresponds to a two-way traffic of messages.

5.2.2 Communication Graph Topology

The level of communication is related to the graph density ρ. It is defined as the number of edges divided by the total number of edges of the complete graph $G(V, E)$ in the following way.

$$\rho(e, n) = \frac{e}{n(n-1)/2}, \tag{5.1}$$

[2]A solver in our case is simply a tabu search thread.

where e is the number of edges and n is the number of nodes (or CPUs in this case). Note that $n(n-1)/2$ gives the total number of edges as shown in Equation 2.15. Let us examine the four-solver, eight-solver, and twelve-solver interaction with different communication graph structures created when increasing the density by adding more edges.

In the four-solver communication graph, we can choose not to communicate at all by simply starting with a density 0 containing no edges. So no communication takes place and each solver attempts a problem independently. By adding three edges, we get a path connecting all the four nodes with density 0.50 as shown in Figure 5.1(a). Sequentially adding more edges, we form the other graph structures of density 0.67 in Figure 5.1(b), 0.83 in Figure 5.1(c) and 1.00 in Figure 5.1(d).

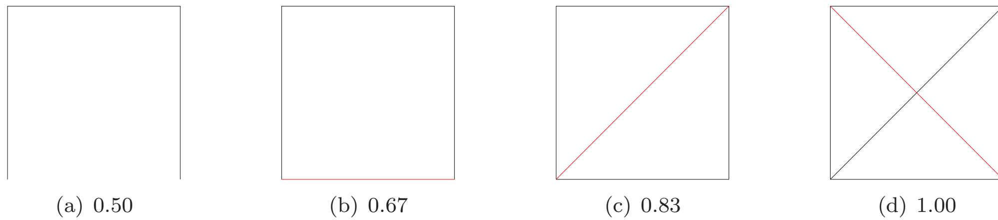


Figure 5.1: Communication graphs for four solvers with different densities.

The no communication graph for the eight-solver consists of eight independent nodes starting with density 0 and no edges. By adding seven edges to a path connecting eight solvers, we get density 0.25 as shown in Figure 5.2(a). Adding one extra edge to connect all the nodes in a circle, we get density 0.29 in Figure 5.2(b). Sequentially adding four more edges in the same similar process as with the four communication graph topology, we get different densities ranging from 0.43 to 1.00 in Figure 5.2(c) to Figure 5.2(g).

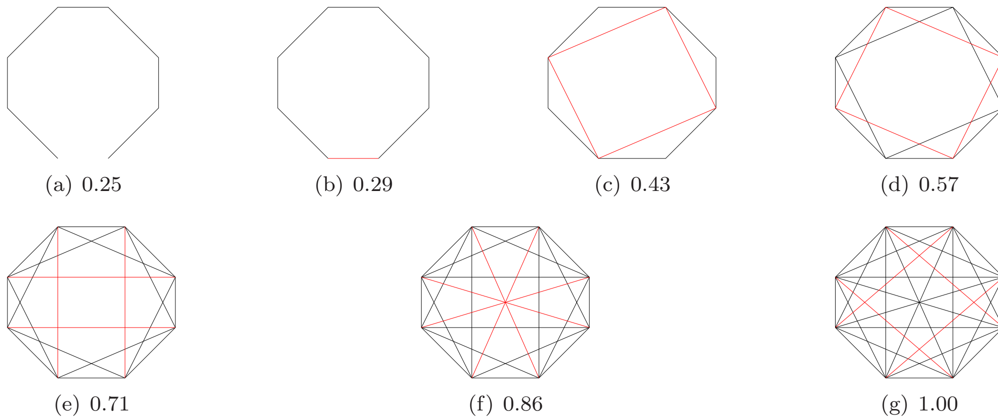


Figure 5.2: Communication graph for eight solvers with different densities.

The no communication graph for the twelve-solver consists of twelve independent nodes starting with density 0 and no edges. Starting the communication graph by

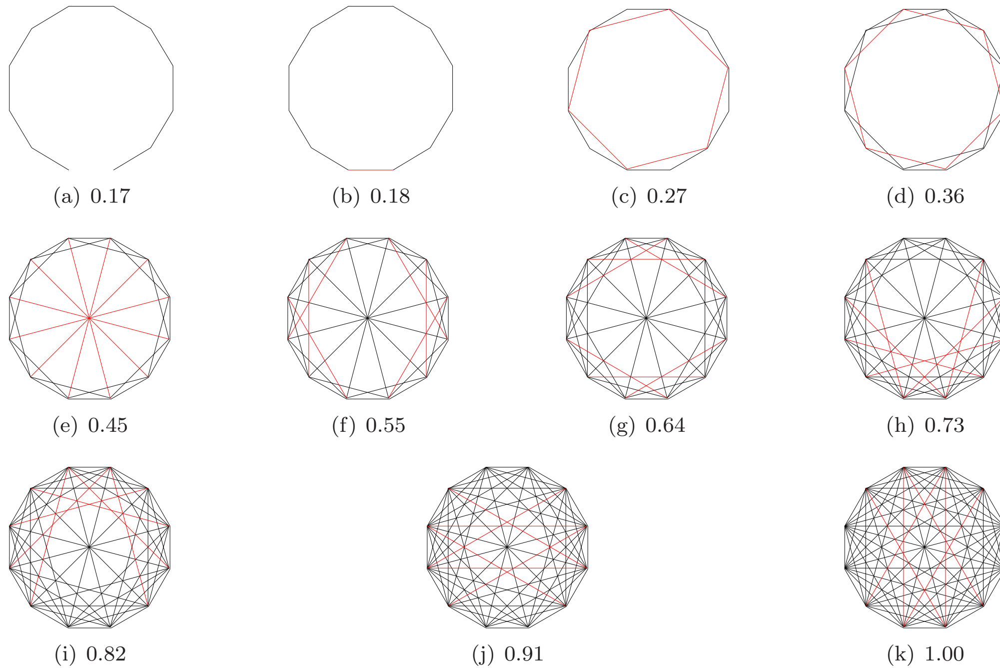


Figure 5.3: Communication graph for twelve solvers with different densities.

adding eleven edges to a path connecting twelve solvers, we begin with density 0.17 in Figure 5.3(a). Adding one extra edge to form a complete circle, we arrive to density 0.18 in Figure 5.3(b). By sequentially adding six more edges to each graph we get different densities ranging from 0.27 to 1.00 in Figure 5.3(c) to Figure 5.3(k). Note that the empty graph for Figure 5.1, Figure 5.2 and Figure 5.3 are not depicted.

5.2.3 Boltzmann Selection Procedure

With the help of communication graphs, a solver can periodically receive foreign solutions from its neighbors. It then decides how to import the foreign solutions into its elite set. An *import-policy* selects one foreign solution at a time to replace the worst elite, if the selected solution is better than the worst elite solution. Otherwise it will be discarded with all the other foreign solutions.

A simple *selection method* is to always choose the best solution. We refer to it as an *import-best policy.* In order to diversify the search with different solutions, we introduce an *import-softmax*[3] policy. The import-softmax policy probabilistically chooses one foreign solution based no the idea of giving every solution a probability of being chosen but favoring solutions of better quality. The procedure for our import-softmax policy is the *Boltzmann selection procedure* [31].

The Boltzmann selection procedure is an implementation of solution selective pressure based on probability. It is modeled after the use of temperature T analogous to the temperature in the metropolis heuristic used in simulated annealing. Mathematically it

[3]In contrast, *import-hardmax* can be thought as always importing the best solution or import-best.

can be described in the following way. Given a set N of solutions $\{s_1, s_2, \ldots, s_N\}$ with their corresponding objective values $\{f_1, f_2, \ldots, f_N\}$ for optimization with the goal of minimization, the probability of choosing a solution s_i is mapped by the Boltzmann selection equation.

$$p(s_i) = \frac{exp(Q_t(s_i)/T)}{\sum_{k=1}^{N} exp(Q_t(s_k)/T)}, \tag{5.2}$$

where $Q_t(s_i) = f_i - \min_{k=1}^{N}\{f_k\}$, and the temperature $T = -(\max\{f_k\} - \min\{f_k\})$. The numerator contains the Boltzmann wighting term and the denominator is a normalization factor. If $T = 0$, then we define $p(s_i) = 1/N$. This mapping assigns a greater probability to a solution with fewer unassigned variables or a better objective value, ensuring that a better solution is more likely to be selected by import-softmax. It is easy to verify that $\sum_{i=1}^{N} p(s_i) = 1$.

An import policy has a local impact at each solver, while a communication graph impacts the global distribution of elite solutions. We call a combination of an import policy and a communication graph a cooperation configuration. How a configuration affects the solver performance will be studied in the experiments.

Chapter 6

Simulation and Results

In this chapter we apply parallel cooperative solvers with tabu search for the quadratic assignment problem. The main objective of the experiments in this thesis is to find out whether adding more solvers helps to find a solution more quickly and how different communication graphs affect the performance.

6.1 Experiment Details

Tabu search tries to escape local optima by forbidding some recent moves by evaluating all the 2-*opt* moves in the current neighborhood. The neighborhood is the 2-opt exchange neighborhood [32] which requires $O(n^2)$ to evaluate, and n is the problem size of the QAP. The 2-opt move consists of selecting two different elements from the original permutation representing a solution and swapping or exchanging their positions, thus yielding a new permutation representing a new solution. If the move is in the tabu list, it is referred as a tabued move. Our tabu search algorithm selects a move in the following order.

1. If the best move in the tabu list leads to a better solution than the current best solution, the search will accept this move.
2. If the best tabu move does not improve the current best solution, then the search will choose the best non-tabu move.
3. If every move in the neighborhood is in the tabu list the search will choose the best tabu move if it improves the present solution or the move that has been kept in the tabu list the longest time. Ties are broken randomly when choosing a move.

Each tabu search thread run by a solver is configured in the following way.

- *Tabu list size:* A solver fixes its tabu list length by randomly choosing it from the set of $\{5, 7, 9, 11, 15, 17\}$.
- *One elite solution:* A solver keeps track of the best solution found so far, either found by itself or imported from its neighboring solvers.
- *Bounding search:* Each individual tabu search is bounded by 500 iterations after no improvement. The maximum number of iterations is 10000.

- *Proportion of searching guided by elite solutions:* After 500 iterations without improvement, a solver restarts either from a randomly generated solution, or from the elite solution. The probability of restarting from the elite solution is set to 0.5.
- *Communication:* A solver sends the best solution to its neighboring solvers every 50 iterations and imports a solution at the end of each individual search. The imported solution will replace the current elite solution, if it is better.

6.2 Computing Environment

The computing environment consists of 31 nodes. Each node has two 2.0 GHz AMD Dual Core processors (i.e., 4 CPUs) with 4GB RAM running Red Hat Enterprise Linux 4. Parallel tabu search is implemented in ANSI C/C++ with the MPICH2 library for communication. The QAP instances are a set of recently released problems based on micro array layout.[1] Each instance was attempted 5 times with the communication graphs combined with the import-softmax policy.

6.3 Experimental Results

The parallel tabu search (cooperative solvers) outperform several best known solutions on the micro array instance set. In Table 6.3 we compare objective values on the microarray-layout border length minimization (bl) and conflict index minimization (ci) QAP instances [33] using 12 cooperative solvers with the best configuration to the best known results. The relative error is calculated as

$$\frac{Obj - Obj_B}{Obj_B} \times 100\%, \tag{6.1}$$

where Obj is the objective value obtained by the solvers and Obj_B is the best known in the litterature [33]. A bold entry indicates a new best solution. The wall time the duration between the start of executing the solvers and the end at which all the solvers terminate.

The 06×06 ci instances show the best relative error of -0.24%. This means that our PTS algorithm managed to find an optimal value of 0.24% better than the value found so far for the same instance. The other interesting instances are 07×07 ci with an improvement of -0.20%, 08×08 bl with an optimized value of -0.13%, 08×08 ci with -0.10% and last the 07×07 bl with the best improvement of -0.09%.

In Figure 6.1, the speedup on the mean concurrent iterations is shown. It is calculated as follows.

1. The best solution ever found is identified and a very close range within the best objective value, e.g. 0.1%, is targeted.
2. For each run, the iteration at which the objective value first reaches the target range is recorded. If a run fails to reach the range, the maximum iteration (10000) is used instead.

[1] Details at http://gi.cebitec.uni-bielefeld.de/comet/chiplayout/qap/index.html

Instance	Best Known	Cooperative Solvers	Relative Error (%)	wall time (min.)
06×06 bl	3, 296	3, 296	0.00	5.0
07×07 bl	4, 564	**4,560**	**−0.09**	15.6
08×08 bl	6, 048	**6,040**	**−0.13**	29.7
09×09 bl	7, 644	7, 648	+0.05	71.8
10×10 bl	9, 432	9, 452	+0.21	128.2
11×11 bl	11, 640	11, 676	+0.31	181.8
12×12 bl	13, 832	13, 848	+0.12	429.1
06×06 ci	169, 016, 907	**168,611,971**	**−0.24**	5.1
07×07 ci	237, 077, 377	**236,613,631**	**−0.20**	17.6
08×08 ci	326, 696, 412	**326,376,790**	**−0.10**	29.8
09×09 ci	428, 682, 120	430, 224, 089	+0.36	53.6
10×10 ci	525, 401, 670	528, 471, 212	+0.58	118.6
11×11 ci	658, 317, 466	662, 898, 977	+0.70	247.6
12×12 ci	803, 379, 686	809, 244, 786	+0.73	411.4

Table 6.1: The best values are shown in bold. These values are even better than those found in the literature.

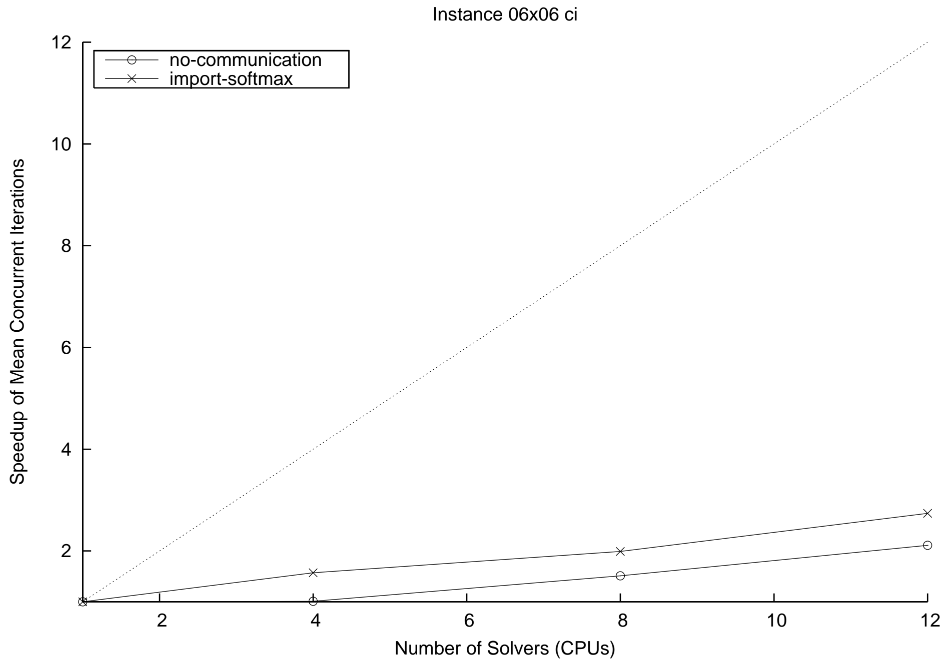


Figure 6.1: Speedup on the mean concurrent iterations of 4, 8, and 12 solvers.

3. The iterations from 2. for a single solver are averaged over all the runs, then divided by the corresponding mean iterations for multiple solvers.

Figure 6.1 shows the speedup (dotted line correspond to the theoretical speedup) for the 0.1% target range within the best overall solution on the chip size 6×6 conflict index instance. Other instances show similar speedup results. Adding more solvers helps to reach a high-quality solution more quickly, and communication with the best configuration outperforms non-communication. However, the speedup develops more slowly: the 12 solvers achieved a mere 2.7 speedup with the best cooperation configuration and with non-communication only a speedup of 2.1 is achieved.

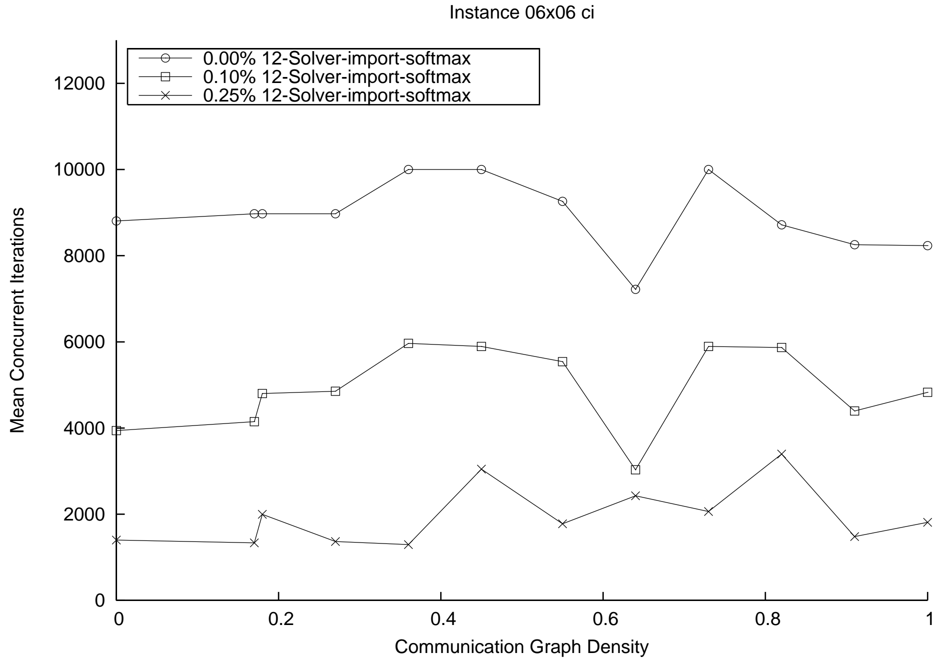


Figure 6.2: Comparing the mean concurrent iterations across different communication graphs for 12 solvers. The best solution found has an objective value of 168611971. The target ranges are 0%, 0.1%, and 0.25% away from this objective value.

Figure 6.2 presents the mean concurrent iterations for 0%, 0.1%, and 0.25% target ranges across different communication graphs for 12 solvers. The solvers are not always finding the optimal solutions, so we decided to target the solutions at .10% and .25% away from the optimal. This way we can plot a that the set of solvers will find at least once. Looking at the 0.00% and 0.10% we notice a large improvement of the mean concurrent number of iterations at 64% communication density. Clearly communication has some positive impact on the twelve solver for 64% communication density because we managed to find the best solution quicker than without communication (0% communication density). But the 0.25% range undeniably show that communication does not help. Only a few topologies help the performance in the PTS case.

Nevertheless, the cooperative solvers outperform several best know solutions on this QAP instance set[2]. We managed to always beat the GRASP-PR algorithm [33] on all

[2]Please visit http://gi.cebitec.uni-bielefeld.de/comet/chiplayout/qap for more details about the

the instances and always with communication for all densities. We also managed to find better solutions than GATS algorithm [33] with PTS for the instances $07x07x$ and $08x08$ with solutions $4,560$ and $6,040$, thus beating it by 0.09% and 0.13% for the border line minimization problem. For the conflict index minimization problem, we managed to get much better values than the GATS algorithm. We beat instance $06x06$ with our solution $(168,611,971)$ by 0.24%, instance $07x07$ $(236,613,631)$ by 0.20% and instance $08x08$ $(326,376,790)$ by 0.10% with the help of our PTS algorithm compared to the GATS algorithm.

6.4 Conclusion and Future Work

The presented parallel interactive solvers owns the entire problem and collaborates by exchanging elite solutions. Experimental results demonstrate that adding more agents helps to improve the performance for optimization problems. We also introduced communication graphs and import policies to change the way that solvers influence guide search. Experimental results showed that solvers benefit slightly on the quadratic assignment problem compared to the quasigroup-with-holes completion problem in [1]. Also, we observed that a communication configuration has a significant impact on search performance. Therefor further investigations are needed to understand how elite solutions disseminate to affect the performance. Also, further investigations are needed to understand how communication affects the performance.

For future work, each specific instance require a fine tuning of PTS. In this way, we should be able to find even better solutions. By enabling a solver to dynamically change its neighbors and to adjust its import policy, we should be able to overcome the problem of having to experiment with a large number of combinations of fine tuning and configuring PTS for each instances by hand. For example, several solvers can start with a fully connected communication graph and end up with an empty one with no communication. Also, solvers can adapt themselves when more solvers join or some leave the search.

---

compared algorithms.

Bibliography

[1] Lei Duan, Samuel Gabrielsson, and J. Christopher Beck. Solving combinatorial problems with parallel cooperative solvers, 2007.

[2] Richard A. Brualdi. *Introductory Combinatorics*, pages 45–53. Prentice-Hall, 1999.

[3] Ralph P. Grimaldi. *Discrete and Cominatorial Mathematics: An Applied Introduction*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1998.

[4] Fred Buckley and Marty Lewinter. *A Friendly Introduction to Graph Theory*, pages 57–58, 145–146. Pearson Education, Inc., Upper Saddle River, New Jersey, USA, 2003.

[5] Joan M. Aldous and Robin J. Wilson. *Graph and Applications An Introductory Approach*, pages 64–65. Springer-Verlag Berlin, 2000.

[6] R. Balakrishnan and K. Ranganathan. *A Textbook of Graph Theory*, pages 102–107. Springer-Verlag New York, Inc., 2000.

[7] Peter B. Andrews. *An Introduction to Mathematical Logic and Type Theory: To Truth Through Proof*, pages 5–67. Kluwer Academic Publisher, 2002.

[8] Nils J. Nilsson. *Artificial Intelligence: A New Synthesis*, pages 215–238. Morgan Kaufmann Publishers, Inc., 1998.

[9] Ellis Horowitz and Sartaj Sahni. *Fundamentals of Computer Algorithms*, pages 501–604. Computer Science Press, Inc., 1978.

[10] Alan Gibbons. *Algorithmic Graph Theory*, page 234. Cambridge University Press, 1985.

[11] G. Ausiello, P. Crescenzi, G. Gambosi, V. Kann, A. Manchetti-Spaccamela, and M. Protasi. *Complexity and Approximation Combinatorial Optimization problems and their Approximability Properties*. Springer, 2003.

[12] Stuart Russel and Peter Norvig. *Artificial Intelligence A modern Approach*. Pearson Education, Inc., 2. edition, 2003.

[13] Patrick Henry Winston. *Artificial Intelligence*. Addison-Wesley, 3. edition, 1992.

[14] Yoshizawa H. and Hashimoto S. Landscape analyses and global search of knapsack problems. In *Systems, Man, and Cybernetics, 2000 IEEE International Conference on*, volume 3, pages 2311–2315, 2000.

[15] Fred Glover. Future paths for integer programming and links to artificial intelligence. *Comput. Operations Research*, 13(5):533–549, 1986.

[16] Fred Glover and Manuel Laguna. *Tabu Search*. Kluwer Academic Publishers, 1997.

[17] Helena R. Lorenco, Oliver Martin, and Thomas Sttzle. Iterated local search.

[18] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*, pages 595–601. MIT Press and MCGraw-Hill, 2 edition, 2001.

[19] Nesetril, Milkova, and Nesetrilova. Otakar boruvka on minimum spanning tree problem: Translation of both the 1926 papers, comments, history. *DMATH: Discrete Mathematics*, 233, 2001.

[20] Seth Pettie and Vijaya Ramachandran. An optimal minimum spanning tree algorithm. *J. ACM*, 49(1):16–34, 2002.

[21] Václav Vasek Chvátal. A greedy heuristic for the set covering problem. *Mathematics of Operations Research*, 4(3):233–235, 1979.

[22] Bart Selman, Hector J. Levesque, and David Mitchell. A new method for solving hard satisfiability problems. In Paul Rosenbloom and Peter Szolovits, editors, *Proceedings of the Tenth National Conference on Artificial Intelligence*, pages 440–446, Menlo Park, California, 1992. AAAI Press.

[23] Charles Darwin. *On the Origin of Species*. London John Murray, 1. edition, 1859.

[24] Tjalling C. Koopmans and Martin J. Beckmann. Assignment problems and the location of economic activities. Cowles Foundation Discussion Papers 4, Cowles Foundation, Yale University, 1957.

[25] Sartaj Sahni and Teofilo Gonzalez. P-complete approximation problems. *J. ACM*, 23(3):555–565, 1976.

[26] Rainer E. Burkard, Stefan E. Karisch, and Franz Rendl. Qaplib-a quadratic assignment problem library. *European Journal of Operational Research*, 55:115–119, 1991.

[27] Teodor Gabriel Crainic and Michel Gendreau. Cooperative parallel tabu search for capacitated network design. *Journal of Heuristics*, 8(6):601–627, 2002.

[28] R. M. Aiex, S. L. Martins, C. C. Ribeiro, and N. D. L. R. Rodriguez. Cooperative multi-thread parallel tabu search with an application to circuit partitioning. *Lecture Notes in Computer Science*, 1457:310, 1998.

[29] S. Porto and C. Ribeiro. Parallel tabu search message-passing synchronous strategies for task scheduling under precedence constraints, 1995.

[30] Theodor Gabriel Cranic and H. Nourredine. *Parallel Meta-Heuristics Applicatoins*. John Wileys and Sons, 2005.

[31] Michael de la Maza and Bruce Tidor. An analysis of selection procedures with particular attention paid to proportional and boltzmann selection. In *Proceedings of the Fifth International Conference on Genetic Algorithms*, pages 124–131, San Fransisco, California, USA, 1993. Morgan Kaufmann Publisher Inc.

[32] Holger H. Hoos and Thomas Stützle. *Stochastic Local Search Foundations and Applications*, chapter 9, pages 372–373. Morgan kaufmann, 1 edition, 2004.

[33] Sérgio A. de Carvalho Jr. and Sven Rahmann. Microarray layout as a quadratic assignment problem. In *Proceedings of the German Conference on Bioinformatics (GCB)*, volume P-83, pages 11–20, 2006.

Index

2-opt, 55
$\mathcal{NP}$-complete, 26, 27
$\mathcal{NP}$-complete problem, 27
$\mathcal{NP}$-hard, 26

adjacent, 10
adjacent list, 16
adjacent matrix, 16
arc, 9, 39
arc capacity, 39
artificial intelligence, 30

Boltzmann selection procedure, 53
boolean connectivity, 26
boolean formula, 26
boolean logic, 26
boolean variables, 26
brute-force, 28

children, 36
Chvátal's greedy heuristic, 35
circuit, 11
class $\mathcal{NP}$, 25
closed, 11
closed walk, 11
combination, 7
combinatorial, 20
combinatorial optimization problem, 20
combinatorics, 5
communication graphs, 50
communication strategy, 50
complement, 14
complete graph, 14
Computer Aided Design (CAD), 45
conjunctive normal form, 26
connected graph, 12
conservation of flow, 39
constraints, 19
continuous problems, 20
convex programming problem, 20
cost function, 19
costs, 16
cycle, 12

decision algorithm, 24
decision problem, 24
degree, 15
demand node, 39
dense graph, 17
deterministic algorithm, 24
digraph, 10
Dijkstra's algorithm, 35
directed graph, 9
disconnected graph, 12
disjunctive normal form, 26
diversification, 50
diversify, 30

edge, 9
edge capacity, 39
edge set, 10
elevation, 31
elite set, 50
Euler circuit, 15
Euler path, 15
Euler trail, 15
evolutionary algorithm, 35
evolutionary algorithms, 34
exhaustive search, 23

feasible, 35
feasible solution, 21
feasible solutions, 19
flow, 39
forbidden, 29
formula, 25
fundamental principles of counting, 5

generation, 36
genetic algorithm, 34, 35
globally optimum, 19
gradient ascend, 30

gradient decent, 30
graph, 8
greedy algorithm, 34
greedy algorithms, 34
greedy local search, 31
greedy satisfiability, 35

Hamiltonian circuit, 22
heuristic, 27
heuristic method, 28
heuristic methods, 27
hill climbing, 30
hybrid algorithm, 36
hybrid evolutionary algorithm, 34, 36

import-best policy, 53
import-hardmax, 53
import-policy, 53
import-softmax, 53
incident, 10
instance, 19
intensification, 49
intractable, 24
irrevocable, 35
isolated vertex, 15
iterated local search, 34

Kruska's algorithm, 35

legal, 29
legality condition, 28
length, 11
linear programming problem, 20
literal, 26
local improvement procedures, 31
local search, 28
local search algorithm, 29
locally optimal, 19, 21, 35
location, 31
location theory, 43
loop, 10

mathematical programming, 19
metaheuristic, 34
metropolis heuristic, 32
minimum cost network flow problem, 40
move, 47
moves, 28
multigraph, 13
multiple edges, 10
mutation, 36

negations, 26
neighborhood, 20, 21
network, 39
node, 39
nodes, 9
nondeterministic algorithm, 24, 25
nondeterministic polynomial algorithm, 25
nonlinear programming problem, 20
null graph, 14

objective function, 19
open walk, 11
operations research, 20, 39
optimal solutions, 21
optimal tabu list size, 49
optimization, 19
optimization problem, 19
Optimization problems, 20
optimum, 19

parallel computing, 3
parallel metaheuristics, 50
parallel tabu search, 46, 50
parents, 36
path, 11
pendant vertex, 15
permutation, 6
polynomial time, 24
population, 35
Prim's algorithm, 35
proposition calculus, 25

quadratic assignment problem, 36, 41
quadratic assignment problem library, 46
quadratic programming, 20

random restart hill climbing, 33
reduced, 26
reduces in polynomial time, 26
regular graph, 15
restart, 50
restart method, 33
rugged landscape, 31

satisfiability problem, 26
satisfiable, 26
search space, 19, 21
search space landscape, 31

search strategies, 48
selection, 7
selection method, 53
selection rule, 28, 29
simplex algorithm, 20
simulated annealing, 34, 35
singleton graph, 14
solution, 21
solver, 50
spanning subgraph, 13
sparse graph, 17
species, 35
state, 28, 31
steepest ascend, 30
steepest descent, 30
stochastic hill climbing, 33
strategic oscillation, 50
subgraph, 13
subgraph induced, 14
subset, 19
supply node, 39

tabu, 47
tabu list, 49
tabu list length, 49
tabu list size, 49
Tabu search, 47
tabu search, 34
The assignment of gates to airplanes in an airport, 46
The assignment of specialized rooms in a building, 46
the assignment problem, 41
the rule of product, 5
The traveling salesman problem, 22
theory of evolution, 35
tour, 22
tractable, 24
trail, 11, 12
transition graph, 28
transportation problem, 40
transshipment node, 39
traveling salesman problem, 19
trial and error, 28
trivial walk, 11
truth assignment, 26

valence, 15
verification stage, 25
vertex set, 10
vertices, 9

walk in a graph, 11
weighted digraph, 16
weighted graph, 16
weights, 16