Discrete Optimization

Professor Montaz Ali,
School of Computer Science and Applied Mathematics
University of the Witwatersrand, Johannesburg

July 14, 2020

1 Optimization in Network

There are many decision problems in operation research that can be modeled and interpreted using networks or graphs (undirected or directed graphs). These problems are easily illustrated by using a network of arcs (set of edges E for undirected graph, and the set arches A for the directed graph), and nodes (vertices). In most cases, these problems can be formulated mathematically as integer linear programing problems. Hence these are discrete or combinatorial optimization problems.

The network structures of these problems can be exploited in the construction of efficient algorithms for their solution. The undirected graphs deal with may problems involving path in a graph (finding shortest or longest path), involving cycles or tours (Euler tour: Is there a cycle that uses each edge exactly once? Hamilton tour: Is there a cycle that uses each vertex exactly once?), connectivity (Is there a way to connect all of the vertices? maximum spanning tree: What is the best way to connect all of the vertices?), and involving bi-connectivity (Is there a vertex whose removal disconnects the graph?). On the other hand, all network flow problems can be modeled by directed graphs. Some examples of network optimization problem are as follows. In transportation systems goods are transported over transportation networks; fleets of airplanes are scheduled over the air transportation networks. In manufacturing systems goods are scheduled for manufacturing; manufactured items are flown within inventory systems; design and expansion of communication systems network; flow of information across networks. All these problems are better explained through their respective network flow models [10, 26].

The most important application of network flows are shortest paths (also in undirected graphs), maximum flows (what is the largest flow of materials from source to sink that does not violate any capacity constraints?), minimum cost flows, and the assignment problem (as this problem can also be defined using a network). The class of network problems also includes: (1) matchings; (2) minimum spanning trees; (3) multi-commodity flows; (4) the traveling salesman problem; and (5) network design. We will only discuss a few of the network optimization problems in this course.

We begin with some terminologies of networks (which are also illustrated with diagrams). A network or graph consists of points, and lines connecting pairs of points. The points are called nodes or vertices. The lines are called arcs or edges. The arcs may have a direction on them, in which case they are called directed arcs. When an arc (or arcs) has direction we will refer to it as a directed arc. If an arc has no direction, it is often called an edge. However, we will write edges and arcs interchangeably there are no direction associated with them. If all the arcs in a network are directed, the network is a directed network. If all the arcs are undirected, the network is an undirected network. Two nodes may be connected by a series of arcs. A path is a sequence of distinct arcs (no nodes repeated) connecting the nodes. A directed path from node i to node j is a sequence of arcs, each of whose direction (if any) is towards j.

A path that begins and ends at the same node is a cycle (or circuit) and may be either directed or undirected. A network is connected if there exists a (directed or undirected) path between any pair of nodes. A connected network without any cycle is called a tree, mainly because it looks like one.

We will now consider diagrams and examples to illustrate various various concepts in networks. A network is defined as $G = (V, E)$ where V is a set of nodes (vertices) and E is the set of arc (link, edge or branch), directed or undirected. For example, $G = (V, E)$

where $V = \{1, 2, 3, 4, 5, 6, 7\}$ and $E = \{(1, 2), (1, 3), (1, 4), (2, 4), (2, 5)(3, 4), (3, 6), (4, 5), (4, 6), (4, 7)\}$ as shown in Fig. 1. There are directed and undirected graphs or networks. In the undirected the arc $(i, j) = (j, i)$ i.e. there is no distinctions between the arcs (i, j) and (j, i). A undirected complete graph is a graph that contains an edge for every (unordered) pair of nodes. That is, a complete graph has $n(n-1)/2$ edges.

A directed graph $G = (V, E)$ is defined analogously with the only difference that edges are directed. That is, every edge $e = (i, j) \in E$ is associated with an ordered pair $(i, j) \in V \times V$. An example of the directed graph is shown in Fig. 2. Note that, as opposed to the undirected case, arch (i, j) is different from edge/arc (j, i). Hence a directed complete graph has $n(n-1)$ edges. The directed and undirected networks are shown in the figures below.

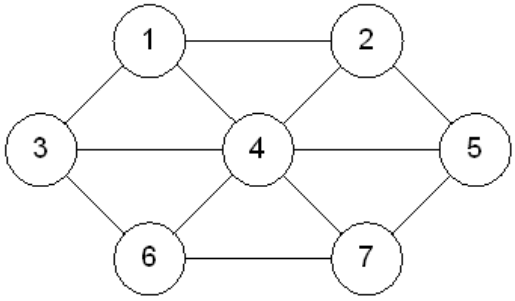

Figure 1: Undirected Graph

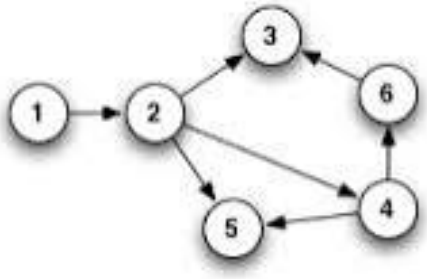

Figure 2: Directed Graph

An undirected *Path* is where no nodes are repeated and directions are ignored (also when no directions are given), e.g. $5, 2, 3, 4$ as shown in the following Fig. 3. A *Directed Path* is where no nodes are repeated and directions are important, e.g. $1, 2, 5, 3, 4$ as shown in the following Fig. 4.

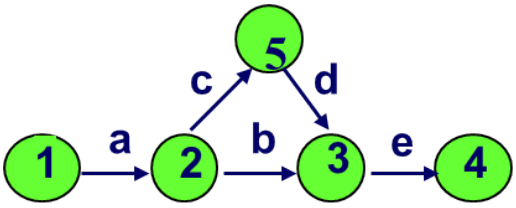

Figure 3: Undirected Path

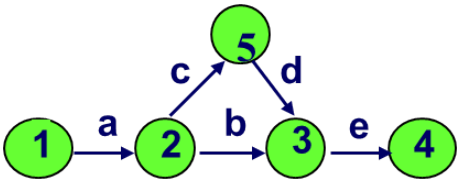

Figure 4: Directed Path

An undirected *Cycle* (or circuit or loop) is a path with 2 or more nodes, except that the first node is the last node, where directions are ignored (also when no directions are given) , e.g. $0, 1, 2, 0$ in Fig. 5. A *directed Cycle* e.g. $1, 2, 3, 4, 1$ in Fig. 6, where no nodes are repeated and directions are important. Euler Circuits and Hamiltonian Circuit are important concepts in networks and introduced next.

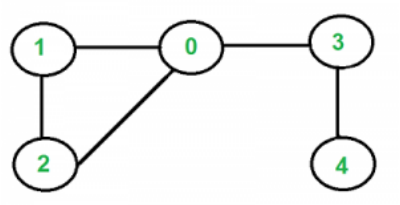

Figure 5: Undirected Cycle

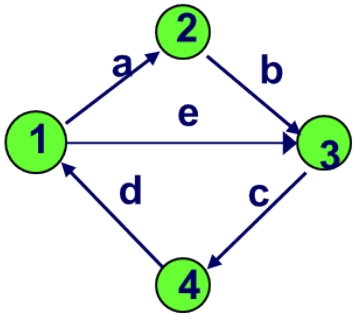

Figure 6: Directed Cycle

Euler Paths and Euler Circuits: An Euler path is a path that uses every edge of a graph exactly once[1]. An Euler circuit is a circuit that uses every edge of a graph exactly once. An Euler path starts and ends at different vertices. An Euler circuit starts and ends at the same vertex. An undirected graph has an Eulerian cycle if and only if (a) every node degree (degree is the number of edges that are adjacent to that vertex) is even and (b) the graph is connected (that is, there is a path from each node to each other node). Any Eulerian cycle visits each node an even number of times. For a directed graph we have:

- A directed graph has an Eulerian circuit if and only if it is connected and each vertex has the same in-degree as out-degree.
- A directed graph has an Eulerian path if and only if it is connected and each vertex except 2 have the same in-degree as out-degree, and one of those 2 vertices has out-degree with one greater than in-degree (this is the start vertex), and the other vertex has in-degree with one greater than out-degree (this is the end vertex).

Hamilton Paths and Circuits: A Hamiltonian path (directed and undirected) passes through each vertex (note not each edge), exactly once[2], if it ends at the initial vertex then it is a Hamiltonian cycle. Unlike, in a Euler path (one might pass through a vertex more than once) in a Hamiltonian path one may not pass through all edges. A Hamilton circuit is a circuit that includes each vertex of the graph once and only once. At the end, of course, the circuit must return to the starting vertex. See the Eulerian and Hamiltonian cycles for undirected networks in the following figures. I will bring these concepts as they become relevant to various optimization problems.

[1] An undirected graph has an Eulerian path if and only if it is connected and all vertices except 2 have even degree. One of those 2 vertices that have an odd degree must be the start vertex, and the other one must be the end vertex.

[2] Hence it is very relevant to the traveling salesman problem which seeks for the minimum Hamiltonian Cycle.

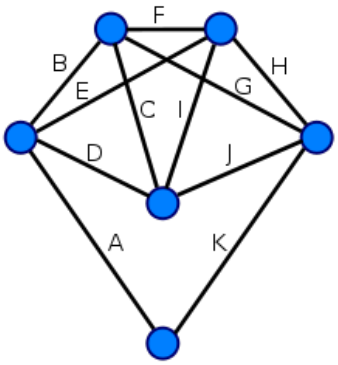

Figure 7: Eulerian Cycle

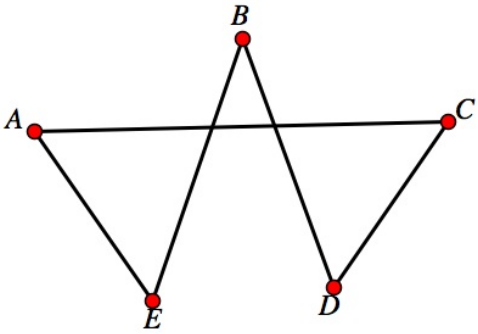

Figure 8: Hamiltonian Cycle

1.1 Shortest Path Problem

This problem can be described by a particular type of network model, called the shortest path problem. In such a problem, we have a network with costs on the edges and two special nodes: a start node (source node s) and a finish node (sink or terminal node t). The goal is to find a path from the start node to the finish node whose total weight is minimized. The shortest path problem can be defined for graphs whether undirected, directed, or mixed. The maximum flow problem is a fundamental problem in combinatorial optimization with many applications in practice, and soon we shall see that the shortest path problem can be modeled as a network flow model.

Consider a phone network. At any given time, a message may take a certain amount of time to traverse each line (due to congestion effects, switching delays, and so on). This time can vary greatly minute by minute and telecommunication companies spend a lot of time and money tracking these delays and communicating these delays throughout the system. Assuming a centralized switcher knows these delays, there remains the problem of routing a call so as to minimize the delays. So, what is the least delay path in a network? How can we find that path quickly?

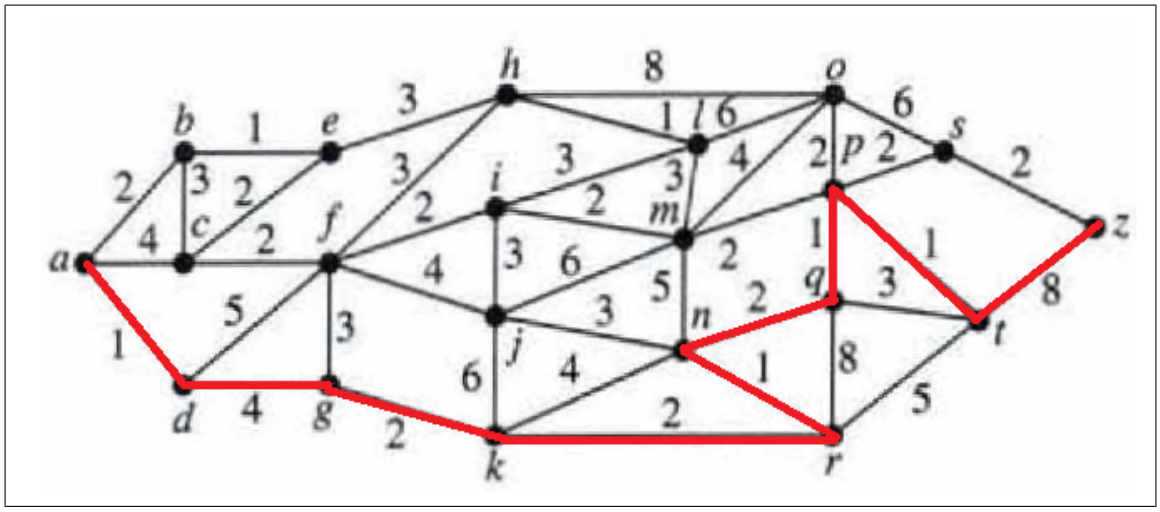

Figure 9: The shortest path between a to z in the weighted graph

The essence of the problem can be stated as follows: given a weighted network (where weight of an edge is travel time, or cost, etc.) with an weight associated with each edge, find a path through the network from a particular origin (source) to a particular destination (sink) that has the shortest total distance. Here I will present the mathematical model for the shortest path problem for a directed graph ((see, Fig 9)), but will discuss the mathematical model for the undirected graph (see, Fig 10) in class.

More formally, let $G = (V, A)$ be the network or graph with $V = \{1, 2, \cdots, n\}$ and edge set E, the nonnegative cost $c : E \rightarrow R^+$ on the edges (i.e c_{ij} is the cost of the arc (i, j)), and a source node $s \in V$ to and a target node $t \in V$. The integer linear programming (ILP) formulation of the shortest-path problem is as follows:

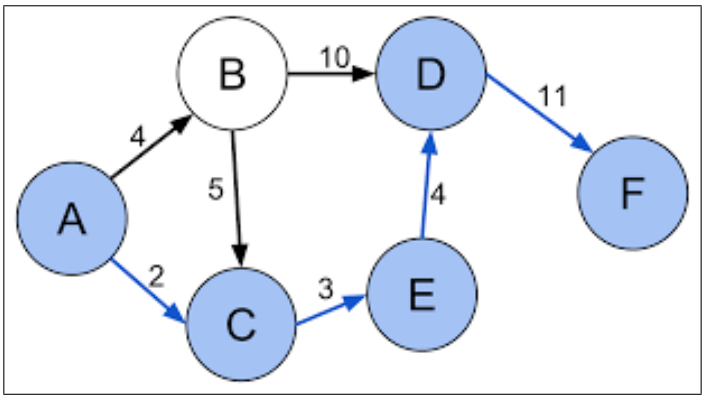

Figure 10: The shortest path between A to F in the weighted graph

$$\text{min} \quad \sum_i \sum_j c_{ij} x_{ij} \tag{1}$$

$$\text{subject to} \quad \sum_j x_{sj} - \sum_k x_{ks} = 1 \qquad s(\text{source node}), \tag{2}$$

$$\sum_j x_{ij} - \sum_k x_{ki} = 0 \qquad i(\text{intermediate node}), \tag{3}$$

$$\sum_j x_{tj} - \sum_k x_{kt} = -1 \qquad t(\text{sink node}), \tag{4}$$

$$x_{ij} \in \{0, 1\} \qquad (i, j) \in E. \tag{5}$$

The 0-1 integer variable x_{ij} is an indicator variable for whether edge (i, j) is part of the shortest path: 1 when it is, and 0 if it is not. Note that the objective or cost function can be written as

$$\sum_{(i,j) \in E} c_{ij} x_{ij}$$

We wish to select the set of edges with minimal weight, subject to the constraint that this set forms a path from s to t. The ILP can be easily relaxed to its corresponding LP (linear programming) model by replacing Eq. (5) with $0 \leq x_{ij} \leq 1$. This LP has the special property that it is integral; more specifically, every basic optimal solution (when one exists) has all variables equal to 0 or 1. This means one can solve LP and obtain the integral solution of ILP.

The question remains as to why the model presented by Eqs. (1)-(5) is a network model? Indeed, we can interpret the shortest-path problem as a network-flow problem very easily. We simply want to send one unit of flow from the source to the sink at minimum cost. At the source, there is a net supply of one unit; at the sink, there is a net demand of one unit; and at all other nodes there is no net inflow or outflow.

1.2 Max-Flow Problem

Another type of model again has a number on each directed arc[3], but now the number corresponds to a capacity. This limits the flow on the arc to be no more than that value. For instance, in a distribution system, the capacity might be the limit on the amount of material (in tons, say) that can go over a particular distribution channel. We would then be concerned with the capacity of the network: how much can be sent from a source node to the destination node?

For the maximal flow problem, we wish to send as much material as possible from s (source) to t (sink). No costs are associated with flow. Let F denote the amount of material sent from node s to t and x_{ij} denote the flow of node i to node j over the arch (i, j). In the LP formulation, the objective is to maximize F. The amount that leaves the origin by various routes. For every intermediate node, what comes in must be equal to what goes out. In some routes the flow can go both ways. The capacity amount that can be sent in a particular direction is also shown on the each route, see Figs. 11 & 12.

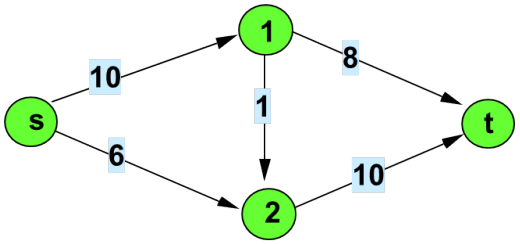

Figure 11: Flow network with capacities u_{ij}

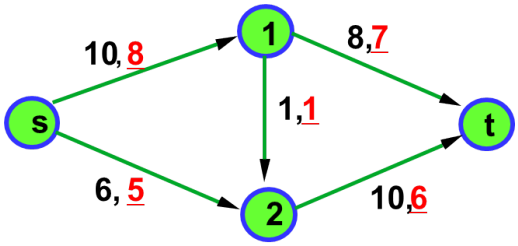

Figure 12: Flow network with capacities u_{ij} and amount of flow x_{ij} (red)

The mathematical formulation of the problem is given by:

$$\text{max} \qquad F \tag{6}$$

$$\text{subject to} \qquad \sum_j x_{sj} - \sum_k x_{ks} = F \qquad s(\text{source node}), \tag{7}$$

$$\sum_j x_{ij} - \sum_k x_{ki} = 0 \qquad i(\text{intermediate node}), \tag{8}$$

$$\sum_j x_{tj} - \sum_k x_{kt} = -F \qquad t(\text{sink node}), \tag{9}$$

$$0 \leq x_{ij} \leq u_{ij} \qquad (i, j) \in E. \tag{10}$$

Notice that the mathematical formulation is an LP and only the integral values are used at each directed arch. This is because if each edge in a flow network has integral capacity, then there exists an integral maximal flow (i.e x_{ij} are integers).

[3]This problem applies to the directed network only.

Remark 1.1. *This problem is very important since there is a polynomial time algorithm (Ford-Fulkerson) for the problem, and an important theorem (max-flow min-cut theorem) associated with it.*

1.3 Minimum Cost Flow Problem

The minimum cost flow problem holds a central position among network optimization models, both because it encompasses such a broad class of applications and because it can be solved extremely efficiently. Like the maximum flow problem, it considers flow through a network with limited arc capacities. Like the shortest-path problem, it considers a cost (or distance) for flow through an arc. Like the transportation problem or assignment problem (see the next problem), it can consider multiple sources (supply nodes) and multiple destinations (demand nodes) for the flow, again with associated costs. Therefore, all of these problems can be seen as special cases of the minimum cost flow problem. The problem is to minimize the total cost subject to availability and demand at some nodes, and upper bound on flow through each arc.

This problem is an example of a transportation problem (or the linear assignment problem). In such a problem, there are a set of nodes called sources, and a set of nodes called destinations. All arcs go from a source to a destination. There is a per unit cost on each arc. Each source has a supply of material, and each destination has a demand. We assume that the total supply equals the total demand (possibly adding a fake source or destination as needed).

Consider a directed and connected network (see Fig. 13) where the n nodes include at least one supply node and at least one demand node. The decision variables are x_{ij} (flow through arc (i, j)), and the given information includes c_{ij} (cost per unit flow through (i, j)), u_{ij} (arc capacity for arc (i, j)), b_i (net flow generated at node i); $b_i > 0$ ($b_i < 0$) means i is a supply node (respectively i is a demand node), $b_i = 0$ (i is a transshipment node). The objective is to minimize the total cost of sending the available supply through the network to satisfy the given demand.

Clearly in this model, we want to talk about multi-source, multi-sink flows than just 'flows from s to t'. We want to impose lower bounds as well as capacities on a given arc. Also, arcs should have costs, see Fig. 13.. Rather than maximize the value (i.e. amount) of the flow through the network we want to minimize the cost of the flow. There is one decision variable x_{ij} for each arch (i, j).

The problem can be mathematically modeled as follows:

$$\text{min} \qquad \sum_i \sum_j c_{ij} x_{ij} \tag{11}$$

$$\text{subject to} \qquad \sum_j x_{ij} - \sum_j x_{ji} = b_i \qquad \text{(for each i)} \tag{12}$$

$$0 \leq x_{ij} \leq u_{ij} \qquad (i, j) \in E. \tag{13}$$

The balanced network can be insured via $\sum_i b_i = 0$. This means The supply/demand constraints must be satisfied exactly.

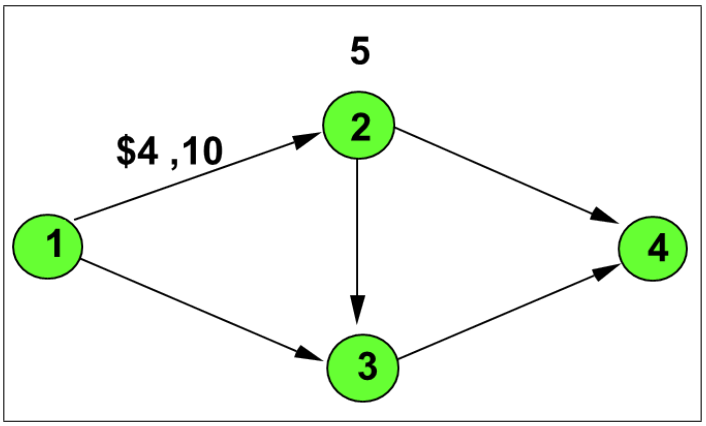

Figure 13: The arc capacity u_{ij} and the cost c_{ij} per unit flow through arc (i, j)

Remark 1.2. *For many applications, b_i and u_{ij} will have integer values, and implementation will require that the flow quantities x_{ij} also be integer. Fortunately, just as for the transportation problem, this outcome is guaranteed without explicitly imposing integer constraints on the variables because of the basic feasible solution (BFS) being integer valued.*

Remark 1.3. *If we constrain the flow value to be 1 (demand at the terminal node t), and all capacities are set to 1, it is pretty clear that the minimum cost flow problem is equivalent to finding the shortest path.*

Remark 1.4. *The transportation problem (or the linear assignment problem) is a network-flow model without intermediate locations, see Fig. 14. To formulate the transportation problem presented as a minimum cost flow problem, 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. All the arcs are directed from a supply node to a demand node, where distributing x_{ij} units from source i to destination j corresponds to a flow of x_{ij} through arc (i, j). The cost c_{ij} per unit distributed becomes the cost c_{ij} per unit of flow. Since the transportation problem does not impose upper bound constraints on individual x_{ij}, Eq. (13) does not apply.*

1.4 Linear Assignment Problem

The linear assignment problem can be described by the way of the following example. Typically, we have a group of n applicants applying for n jobs, and the non-negative cost c_{ij} of assigning the i-th applicant to j-th job is known. The objective is to assign one job to each applicant in such a way as to achieve the minimum possible total cost. The problem is also known as weighted bipartite matching. Define binary variables x_{ij} such that x_{ij}= 1, it indicates that we should assign applicant i to job j. Otherwise (x_{ij} = 0), we should not assign applicant i to job j. The mathematical formulation is as follows:

$$\text{min} \qquad \sum_i \sum_j c_{ij} x_{ij} \tag{14}$$

$$\text{subject to} \qquad \sum_j x_{ij} = 1 \qquad (\text{for each } i) \tag{15}$$

$$\sum_i x_{ij} = 1 \qquad (\text{for each } j) \tag{16}$$

$$x_{ij} \in \{0, 1\} \tag{17}$$

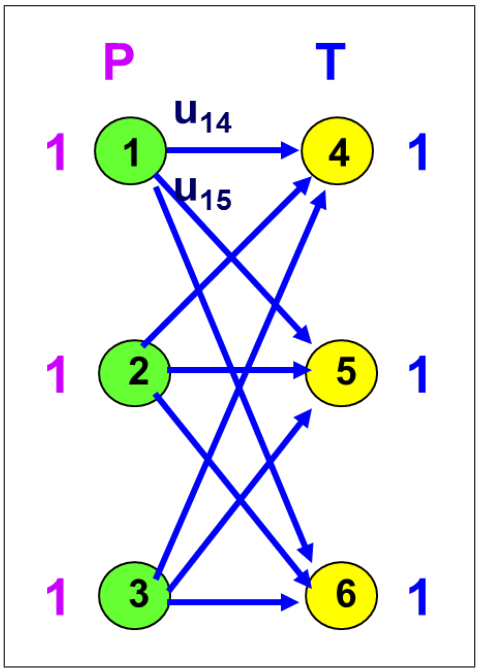

Figure 14: Network for linear assignment problem with P (people) set and T (task) set and the cost c_{ij} (or utilities u_{ij}) for assignment.

Formulate the following linear assignment problem:

The manager of Hotel in Johannesburg have four workers: Agness, Rose, Joyce and Zainab. The manager needs to have one of them clean the bathroom of the hotel, another sweeps the floor of the hotel, the third to be a receptionist and the last person to cook food for the guests and management, but they each demand different pay for their different tasks. The table below represents the costs c_{ij} of the workers i doing the jobs (tasks) j, where ZAR is the South African Rand.

The above problem is the of perfect matching in a bipartite graph since for the bipartite graph $G = (V, E)$ the matching $M \subset E$ is such that every $i \in V$ there is exactly an edge $e \in M$, i.e. every vertex has at exactly one incident edge in M. Hence the problem (14)-(17) can also be written as the maximum weight perfect matching

Table 1: Data showing the cost of workers doing different jobs

	Jobs			
Workers	Cleaning	Sweeping	Receptionist	Cooking
Agness	ZAR9000	ZAR7500	ZAR7500	ZAR800
Rose	ZAR3500	ZAR8500	ZAR5500	ZAR6500
Joyce	ZAR12500	ZAR9500	ZAR9000	ZAR10500
Zainab	ZAR4500	ZAR11000	ZAR9500	ZAR11000

$$\text{max} \quad \sum_{e} c_e x_e \tag{18}$$

$$\text{subject to} \quad \sum_{\text{e incident to } i} x_e = 1 \qquad (\text{for each } i \in V) \tag{19}$$

$$x_e \in \{0, 1\} \tag{20}$$

where c_e is the weight or utility of the edge e. For the case when the matching is not perfect the constraint (19) is replaced with

$$\sum_{\text{e incident to i}} x_e \leq 1, \, \forall i \in V,$$

then this problem is known as bipartite matching problem.

Consider that a matching gives an assignment of people to tasks. You want to get as many tasks done as possible, i.e. you want a maximum matching, one that contains as many edges as possible. Create an instance of bipartite matching. Then create an instance of a network flow where the solution to the network flow problem can easily be used to find the solution to the bipartite matching. The edges used in the maximum network flow will correspond to the largest possible matching!

1.5 Minimum Spanning Tree Problem

A spanning tree is a tree (i.e., a connected acyclic graph) that spans (touches) all the nodes of an undirected network[4]. The cost of a spanning tree is the sum of the costs (or lengths) of its edges. In the minimum spanning tree problem, we wish to identify a spanning tree of minimum cost (or length). A spanning tree of a graph has the following features:

- Every node in the network is connected to every other node by a sequence of arcs from the subnetwork, where the direction of the arcs is ignored.
- The subnetwork contains no loops, where a loop is a sequence of arcs connecting a node to itself, where again the direction of the arcs is ignored.

A subnetwork that satisfies the above two properties is called a spanning tree.

[4]Minimum spanning tree for directed graph can also be constructed but it has limited applications.

The minimum spanning tree (MST) problem is one of the simplest and most fundamental problems in network optimization where given an undirected $G = (V, E)$ and the edge cost $c : E \rightarrow R$ the goal is to find the spanning tree of minimum total cost. If all edges have non-negative costs (i.e $c : E \rightarrow R^+$) , then the MST problem is equivalent to the connected subgraph problem which asks for the computation of a minimum cost subgraph H of G that connects all nodes of G. The MST of the graph in Fig. 15 is shown in bold.

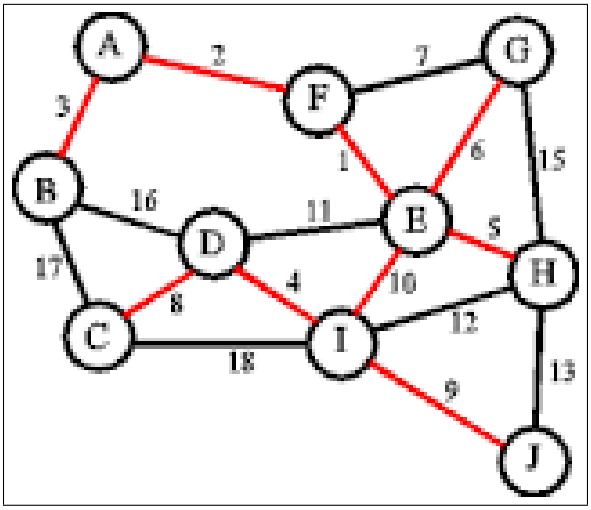

Figure 15: Minimum spanning tree of network in red

As an application, consider the communication company Telkom needs to build a communication network that connects n different users. The cost of making a link joining i and j is c_{ij}. What is the minimum cost of connecting all of the users?

Find the MST of the graph in Fig. 16, where $c_{12} = 1, c_{15} = 1.5, c_{26} = 4, c_{34} = 1.5, c_{36} = 0.5, c_{45} = 1, c_{47} = 2, c_{89} = 2, c_{2,10} = 6, c_{7,10} = 1, c_{67} = 5, c_{68} = 1$.

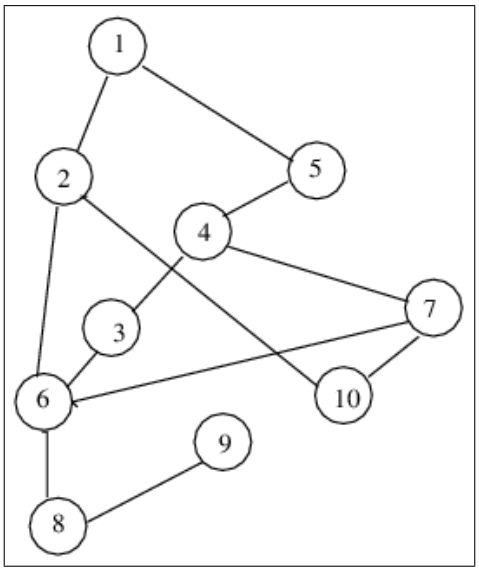

Figure 16: Find the MST of the given network

The mathematical formulation of the MST is based on the fact that the tree must be connected. How can you model this requirement? Let $x_e \in E$ be such that

$$x_e = \begin{cases} 1 & \text{if item } e \text{ included in the spanning tree,} \\ 0 & \text{otherwise.} \end{cases} \tag{21}$$

Let S be any set of vertices. Then S and $V \setminus S$ should be connected. In addition the MST must have $(n-1)$ edges in total. We will now address both of these two requirements in the mathematical formulation. Let

$$\hat{\delta}(S) = \left\{ e = (i,j) \middle| i \in S, j \in V \setminus S \right\}.$$

$$\text{min} \quad \sum_{e \in E} c_e x_e \tag{22}$$

$$\text{subject to} \quad \sum_{e \in \hat{\delta}(S)} x_e \geq 1 \qquad \forall S \subset V, S \neq \emptyset \tag{23}$$

$$\sum_{e \in E} x_e = n - 1 \tag{24}$$

$$x_e \in \{0, 1\}. \tag{25}$$

Clearly, the constraint (23) ensures that the spanning tree is connected, while the number of edges requirement is fulfilled by the constraint (24).

1.6 Solving Max-Flow Problem: The Ford Fulkerson Algorithm

The Ford-Fulkerson algorithm (FFA) [3] is a greedy algorithm that computes the maximum flow in a flow network. The algorithm does not need to maintain the amount of flow on each edge but work with capacity values directly. The algorithm is based on three bright ideas:

- residual flow networks: the graph that shows where extra capacity might be found
- augmenting paths: paths along which extra capacity is possible
- cuts: used to characterize the flow upper bounds in a network

The basic method is iterative, starting from a network with zero flow. Starting with the zero flow, repeatedly augment the flow along any path from s to t in the residual graph, until there is no such path.

The idea behind the algorithm is as follows: as long as there is a path P from the source (start node s) to the sink (end node t), with available capacity on all edges in the path, we send a flow, say x, along the path[5]. We then find another path, add the feasible flow used on each path, and so on. The amount of flow x (from s to t) which satisfies

[5] A path with available capacity is called an augmenting path.

all constraints is called a feasible solution or, a flow. We wish to find the maximum flow from the source node s to the sink node t that satisfies the arc capacities and mass balance constraints at all nodes.

Given a flow x we are able to construct the *residual network* with respect to this flow according to the following intuitive idea. Suppose that an arc $(i, j) \in E$ of the network $G = (V, E)$ carries x_{ij} units of flow. We define the residual capacity of the arc (i, j) as $r_{ij} = u_{ij} - x_{ij}$ (capacity u_{ij}). This means that we can send an additional r_{ij} units of flow from vertex i to vertex j. We can also cancel the existing flow x_{ij} on the arc if we send up x_{ij} units of flow from j to i over the arc (j, i), see Fig. 17.

So, given a feasible flow x (from s to t) we define the residual network with respect to the flow x as follows. A feasible solution x causes a new (residual) network, which we define by $G_x = (V, E_x)$, where E_x is a set of residual edges corresponding to the feasible solution x.

So, what is E_x? We replace each arc (i, j) in E by two arcs $(i, j), (j, i)$: the arc (i, j) has (residual) capacity $r_{ij} = u_{ij} - x_{ij}$, and the arc (j, i) has (residual) capacity $r_{ji} = x_{ij}$. Then we construct the set E_x from the new edges with a positive residual capacity.

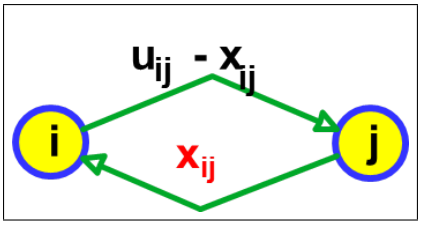

Figure 17: A flow network with residual capacities

Central to the FFA algorithm is the important concept called the *Augmenting Path*. An *Augmenting Path* is a directed path from s to t in the residual network. The residual capacity of an augmenting path is the minimum residual capacity of any arc in the path. Obviously, we can send additional flow from the source to the sink along an *Augmenting Path*.

We will now described the above concepts using flow networks. We begin with a max flow problem with capacities u_{ij} and x_{ij} (in red) are shown in Fig. 18. The corresponding (residual) capacities are shown in the residual network in Fig. 19.

A feasible flow of 7 units can be flown from s to t along the path $s-1-t$. Similarly, 5 units can be flown from s to t along the path $s-2-t$ as shown in Fig. 19. One can find a larger flow from s to t by sending 1 unit of flow along the path $s-2-1-t$ (decreasing flow in $(1, 2)$ is mathematically equivalent to sending flow in $(2, 1)$ with respect to node balance constraints). This is shown in the residual network in Fig. 19 and in the corresponding augmented path P in Fig. 20. The next residual network (corresponding to the path P in in Fig. 20) is shown in Fig. 21.

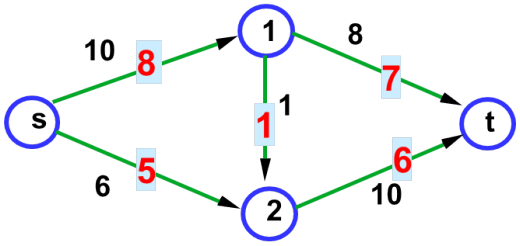

Figure 18: Flow network with feasible flow x_{ij} and capacities u_{ij}

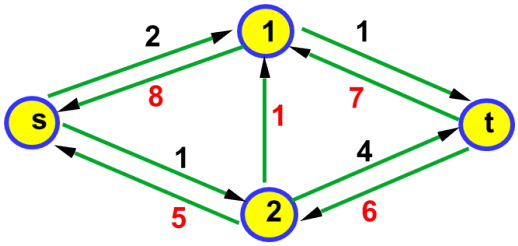

Figure 19: The residual network G_x with residual capacities r_{ij}

The useful idea is the augmenting Paths (an augmenting path is a path from s to t in the residual network G_x). The residual capacity of the augmenting path P is

$$\delta(P) = \min\{r_{ij} \big| (i,j) \in P\}$$

To augment along P is to send $\delta(P)$ units of flow along each arc of the path. We modify x and the residual capacities appropriately

$$r_{ij} = r_{ij} - \delta(P),\ r_{ji} = r_{ji} + \delta(P)$$

These are shown in the following two figures. Can you continue in this way and find the maximum flow?

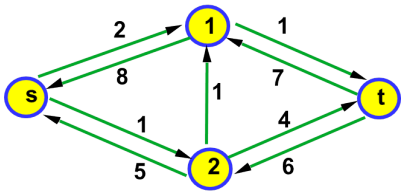

Figure 20: Augmenting path along P

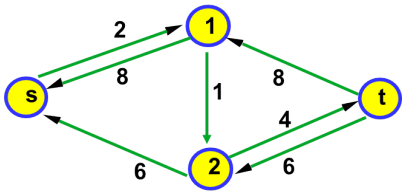

Figure 21: The updated residual capacities in G_x

The Ford Fulkerson Maximum Flow Algorithm

Initial flow x= 0;
Create the residual network G_x
While there is some directed path from s to t in G_x do
Let P be a path from s to t in G_x; G_x is the residual network for flow x
Send $\delta(P)$ units of flow along P;
Update the r_{ij} and return x
End

Definition 1.5. *The capacity$(A; B)$ of an s-t cut $(A; B)$ is the sum of the capacities of edges leaving A.*

Definition 1.6. *An s-t cut $(A; B)$ removes a set of arcs so that A containing s, and B containing t are disjoint.*

Definition 1.7. *An s-t path is a path along which one can push flow from s to t.*

Theorem 1.8. *Let $v(f)$ be the value (it is the amount of material that leaves s, and thus entering t) of an s-t flow and $(A; B)$ is an s-t cut then*

$$v(f) = f^{out}(A) - f^{in}(A)$$

where $f^{out}(A)$ and $f^{in}(A)$ denote the positive flow going out from A (outflow) and coming into A (inflow), respective.

Theorem 1.9. *Let $v(f)$ be the value of an s-t flow and $(A; B)$ is an s-t cut then*

$$v(f) \leq \textit{capacity}(A; B).$$

Remark 1.10. *The above theorem says that any cut is bigger than any flow. Therefore, cuts constrain flows. The minimum capacity cut constrains the maximum flow the most. In fact, the capacity of the minimum cut always equals the maximum flow value.*

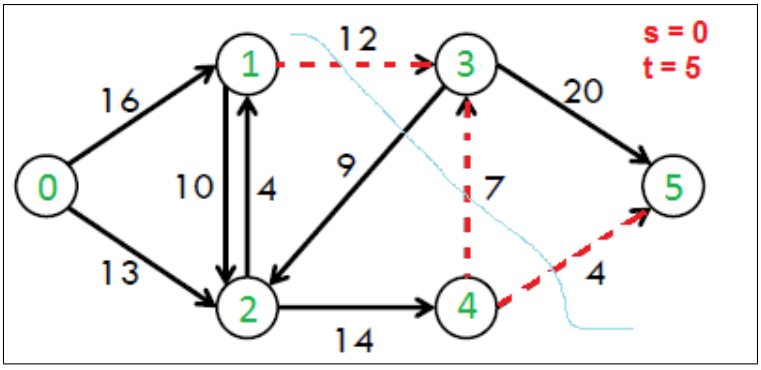

Figure 22: A cut, cut value and the cut sets of a network

Remark 1.11. *Finiteness (assuming capacities are integral and finite): The residual capacities are always integer valued. The residual capacities out of node s decrease by at least one after each update. If there is no augmenting path, then the flow must be maximum.*

Theorem 1.12. *(Max-flow Min-Cut). The maximum flow value is the minimum value of a cut.*

Corollary 1.13. *If the capacities are finite integers, then the Ford-Fulkerson Augmenting Path Algorithm terminates in finite time with a maximum flow from s to t.*

Corollary 1.14. *If the capacities are finite rational numbers, then the Ford-Fulkerson Augmenting Path Algorithm terminates in finite time with a maximum flow from s to t. (why?)*

Remark 1.15. *To understand the celebrated Max-Flow Min-Cut theorem the concept of an s-t cut, the cut capacity value and cut sets are very important. An example of these are given in Fig. 22, where the cut sets are $A = \{0, 1, 2, 4\}$ and $B = \{3, 5\}$; cut capacity value is 23 (added capacities of red arcs) and the cut is the purple line. This is a min cut. Why?*

2 Combinatorial Optimization and Integer Programming Problems

Combinatorics is concerned with the selection, arrangement, sequencing, etc. of a collection of objects. Combinatorial optimization is the study of the best selection, arrangement, sequence, etc., with respect to some appropriately chosen objective function. Therefore, the combinatorial optimization (or discrete optimization) involves problems in which we choose the best solution from a finite (but very large) set of solutions.

These problem can be formulated mathematically by linear integer programming problems or integer quadratic programming problems.

Below we describe a variety of types of combinatorial optimization problems that display varied combinatorial characteristics and have received wide coverage within the literature.

2.1 Knapsack Problem

In a knapsack problem, there are n possible items that can be included in a knapsack, see Fig. 23. Each item i has a weight w_i and a value v_i, where $w_i \geq 0$ and $v_i \geq 0$. There is a limit of W on the total weight of items that can be included in the knapsack. The objective is to select items for the knapsack for which the total value is maximized. The problem can be formulated as a zero-one integer programming problem:

$$\text{max} \qquad \sum_{1=1}^{n} v_i x_i \tag{26}$$

$$\text{subject to} \qquad \sum_{i=1}^{n} w_i x_i \leq W \tag{27}$$

$$x_i \in \{0, 1\} \tag{28}$$

where x_i has the following meaning:

$$x_i = \begin{cases} 1 & \text{if item } i \text{ included in the knapsack,} \\ 0 & \text{otherwise.} \end{cases} \tag{29}$$

It is assumed (for obvious reason) that $\sum_{i=1}^{n} w_i > W$.

The capital budgeting problem have a similar integer programming formulation. A firm has n projects that it would like to undertake but because of budget limitations not all can be selected. In particular, project j is expected to produce a revenue of c_j but requires an investment of a_{ij} in the time period i for $i \in \{1, 2, \cdots, m\}$. The capital available in time period i is b_i. The problem of maximizing revenue subject to the budget constraints can be formulated as follows: let x_j=0 or 1 correspond to not proceeding or respectively

proceeding with project j. The integer programming model of the problem is as follows:

$$\text{max} \qquad \sum_{j=1}^{n} c_j x_j \tag{30}$$

$$\text{subject to} \qquad \sum_{j=1}^{n} a_{ij} x_j \leq b_i \qquad i \in \{1, 2, \cdots, m\} \tag{31}$$

$$x_j \in \{0, 1\}. \tag{32}$$

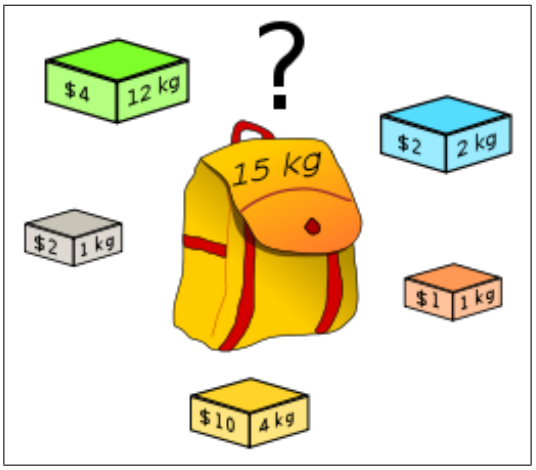

Figure 23: The knapsack problem

Many problems have financial constraints, production capacity constraints, space constraints, etc. Thus, knapsack problems often feature as subproblems of more complex problems.

The quadratic knapsack problem (QKP) is a generalization of the knapsack problem (KP) and it is formulated as:

$$\begin{cases} \max_x & \sum_{j=1}^{n} c_j x_j + \sum_{k=1}^{n-1} \sum_{j=k+1}^{n} d_{kj} x_k x_j, \\ s.t. & \sum_{j=1}^{n} a_{ij} x_j \leq b_i, i = 1, \cdots, m, \\ & x \in \{0, 1\}^n, \end{cases}$$

where the coefficients c_j, d_{ij}, a_{ij}, and $b_i \geq 0$; indices i and j are used to denote i constraint and j-th variable, respectively. Without loss of generality, it is assumed that $\sum_{j=1}^{n} a_{ij} > b_i, i = 1, \cdots, m$. Both KP and QKP are NP-hard in the strong sense [5]. QKP has been intensively studied due to its wide applications.

2.2 Traveling Salesman Problem

The traveling salesman problem (TSP) is one of the most widely studied combinatorial optimization problems [4, 6] because of its wide applicability. Here we describe the problem and present some solution methods. The TSP can be easily stated as follows. A salesman wants to visit n distinct cities and then returns to home city. He wants to

determine the sequence of the travel so that the overall traveling distance (or travel time) is minimized while visiting each city not more than once. Hence, the salesman wants to find a tour of minimum length (or least travel time).

More formally, in the traveling salesman problem, a salesman has to perform a tour which involves visiting each of the cities, say $V = \{1, 2, \cdots, n\}$ exactly once and then returning to the starting point. The cost c_{ij} of traveling directly from city i to city j is given, and it is required to find a tour of minimum total cost.

It can be defined using a complete graph (graph-based version), where the cities are given within the framework of a weighted graph $G = (V, E)$ that dictates the connections between cities so that the problem must be solved with constraints on the order that cities can be visited in. The TSP can therefore be modeled as a graph problem by considering a graph and a distance function $c : A \to R^+$. The usual terminology of the TSPs the vertices are called cities and Hamiltonian circuits are called tours. The problem is then to find a Hamiltonian circuit of minimum total length (or travel time) that contains each vertex of V exactly once. A tour is then a cycle in G which touches every node in V exactly once. The set of these cities V together with their distances $D = (c_{ij})$ are known, where the distance (or travel time) between city i and j is c_{ij}. The graph stated above can be a directed graph or an undirected graph. If it is not directed (since the distance between each pair of cities is independent of the direction in which the sales man travels) then the above TSP is a symmetric TSP [8], since $c_{ij} = c_{ji}$ for all $i, j \in V$. This restriction is not always valid and we then obtain in the case of the directed graph what is known as the asymmetric TSP [7].

Although the TSP is conceptually simple, it is difficult to obtain an optimal solution. In an n-city situation, any permutation of n cities yields a possible solution. As a consequence, $(n-1)!$ possible tours ($n!$ if the home city is any city) must be evaluated in the search space in the asymmetric TSP (and $\frac{1}{2}(n-1)!$ for the symmetric TSP with home city being fixed).

Remark 2.1. *The matrix D is said to satisfy the triangle inequality iff $c_{ij} + c_{jk} \leq c_{ik}$ for all $i, j, k \in V$. This occurs, e.g., in Euclidian TSP where V corresponds to a set of points in R^2 and c_{ij} is the straight-line distance between i and j (e.g. in two-dimensional plane). In this form of the problem, distances between cities are calculated using the Euclidean distance formula.*

2.2.1 Mathematical Model of Asymmetric TSP

The asymmetric TSP can be modeled as a graph problem by considering a complete directed graph $G = (V, E)$ with $E = V \times V$, and assigning a cost c_{ij} to every arc $e = (i, j)$. The problem can now be formulated as a linear integer programming problem where the variables are defined as:

$$x_{ij} = \begin{cases} 1 & \text{if saleman goes from city } i \text{ to city } j, \text{or } e = (i, j) \text{ in the tour} \\ 0 & \text{otherwise.} \end{cases} \tag{33}$$

Hence the mathematical model is the following (linear) binary integer program (BIP):

$$\text{min} \quad \sum_{1=1}^{n} \sum_{j=1}^{n} c_{ij} x_{ij} \tag{34}$$

$$\text{subject to} \quad \sum_{i=1}^{n} x_{ij} = 1 \qquad \forall j \in V, (i \neq j), \tag{35}$$

$$\sum_{j=1}^{n} x_{ij} = 1 \qquad \forall i \in V, (i \neq j), \tag{36}$$

$$\sum_{i \in S} \sum_{j \in S} x_{ij} \leq |S| - 1, \qquad S \subset V, \, 2 \leq |S| \leq n-2, (i \neq j), \tag{37}$$

$$x_{ij} \in \{0, 1\} \qquad (i, j) \in E. \tag{38}$$

Fig. 24 gives the definition of x_{ij} when $e = (i, j)$ is used in a tour (and when not used).

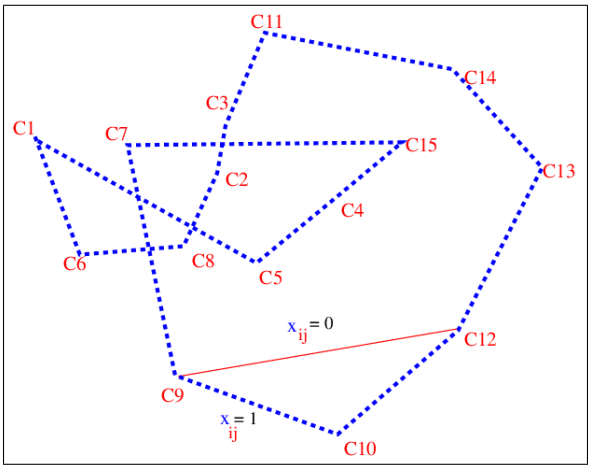

Figure 24: Definition of variable x_{ij}

The constraints specified eqs. (35)-(36) and (38) do not ensure that a binary solution forms indeed a tour (why?). Indeed, the solution without (37) is not a tour but a collection of directed cycles called subtours. Can you create an example?

2.2.2 Mathematical Model of Symmetric TSP

The given complete graph $G = (V, E)$ is undirected, hence it is symmetric - once we obtain a tour, it does not matter which direction we proceed. TSP aims to find the least-cost tour that visits all the vertices. Such a tour (or cycle) visiting all vertices is called a Hamiltonian tour. Defining binary variables x_e for each $e \in E$ and edge costs (or travel times) c_e, symmetric TSP can be formulated as follows:

$$\text{min} \quad \sum_{e \in E} c_e x_e \tag{39}$$

$$\text{subject to} \quad \sum_{e \in \delta(i)} x_e = 2 \qquad \forall i \in V, \tag{40}$$

$$\sum_{e \in E(S)} x_e \leq |S| - 1, \qquad S \subset V,\ 3 \leq |S| \leq n-3, \tag{41}$$

$$x_e \in \{0,1\}, \qquad (i,j) \in E. \tag{42}$$

Here, $\delta(i)$ refers to all edges connected to vertex i, and $E(S)$ denotes all the edges connecting the vertices of S to S. Constraint (40) ensures that each vertex is visited exactly once, and (41) are the subtour elimination constraints. Even though this is a correct formulation, it has the obvious issue that there are exponentially many subtour elimination constraints. In practice, one can attempt to solve the problem without sub-tour elimination constraints. If the solution contains subtours, add constraints eliminating those subtours, and repeat. In each case, if the integrality constraint is relaxed, the problem is a LP. If the solution of the LP is integral and contains no subtour, that solution is optimal.

The symmetric TSP can also be formulated as follows:

$$\text{min} \quad \sum_{i,j:i<j} c_{ij} x_{ij} \tag{43}$$

$$\text{subject to} \quad \sum_{i<k} x_{ik} + \sum_{j>k} x_{kj} = 2 \qquad \forall k \in V, \tag{44}$$

$$\sum_{i \in S} \sum_{j \in S} x_{ij} \leq |S| - 1, \qquad S \subset V,\ 3 \leq |S| \leq n-3, (i<j), \tag{45}$$

$$x_{ij} \in \{0,1\}, \qquad (i,j) \in E. \tag{46}$$

2.3 Heuristics and Approximate Solution Methods for TSP

Like many combinatorial optimization problems TSP is NP-hard and thus there is very little hope that we will be able to develop efficient algorithms for these problems. Nevertheless, many of these problems are fundamental and solving them is of great importance. There are various approaches to cope with these hardness results. Here, we discuss the following two solution approaches:

- Approximation Algorithms: Approximation algorithms are efficient algorithms that compute suboptimal solutions with a provable approximation guarantee. That is, here we insist on polynomial-time computation but relax the condition that the algorithm has to find an optimal solution by requiring that it computes a feasible solution that is close to optimal (see Figs. 25 & 26).

- Heuristics: Any approach that solves the problem without a formal guarantee on the quality of the solution can be considered as a heuristic for the problem. Some heuristics provide very good solutions in practice. An example of such an approach is local search: Start with an arbitrary solution and perform local improvement steps until no further improvement is possible. Moreover, heuristics are often practically appealing because they are simple and thus easy to implement.

We present one local search heuristic (the 2-Opt Heuristic) and one approximate algorithm (The Christofides Algorithm). Other meta-heuristics such as genetic algorithm and simulated annealing will be presented in a later section.

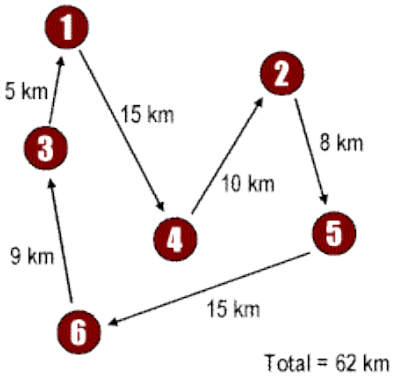

Figure 25: Near optimal solution

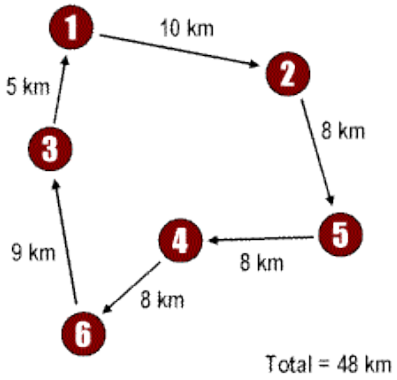

Figure 26: Optimal solution

2.3.1 The Approximation Algorithm

Suppose that P is a minimization problem with set of instances $\mathscr{I}$, and let A be an algorithm that returns for every instance $I \in \mathscr{I}$ a feasible solution $A(I)$ for P. The algorithm A is an α-approximation algorithm with $\alpha \geq 1$ if A finds a feasible solution for all instance I of P in polynomial time such that

$$A(I) \leq \alpha OPT(I), \forall I \in \mathscr{I}, \tag{47}$$

where $OPT(I)$ is the optimal solution of the instance I and $\alpha : \mathscr{I} \rightarrow R^{+}$. For the maximization problem P, the α-approximation can be correspondingly defined with

$$A(I) \geq \frac{1}{\alpha} OPT(I), \forall I \in \mathscr{I}. \tag{48}$$

What would you call the algorithm which find the solution in Fig. 25 in polynomial time? Indeed, the algorithm will be an approximation algorithm and may be called 1.29-approximation algorithm.

2.3.2 The Christofides Algorithm

The Christofides Algorithm is an approximation algorithm for the STSP (symmetric salesman problem). The algorithm guarantees that the distance for the TSP solution found is at maximum $3/2$ times as much as the distance in the optimal solution [26]. Central to

algorithm is fact that every finite undirected graph has an even number of vertices with odd degree (the number of edges touching the vertex). The algorithm is depicted in Figures 27.a - e. and works as follows:

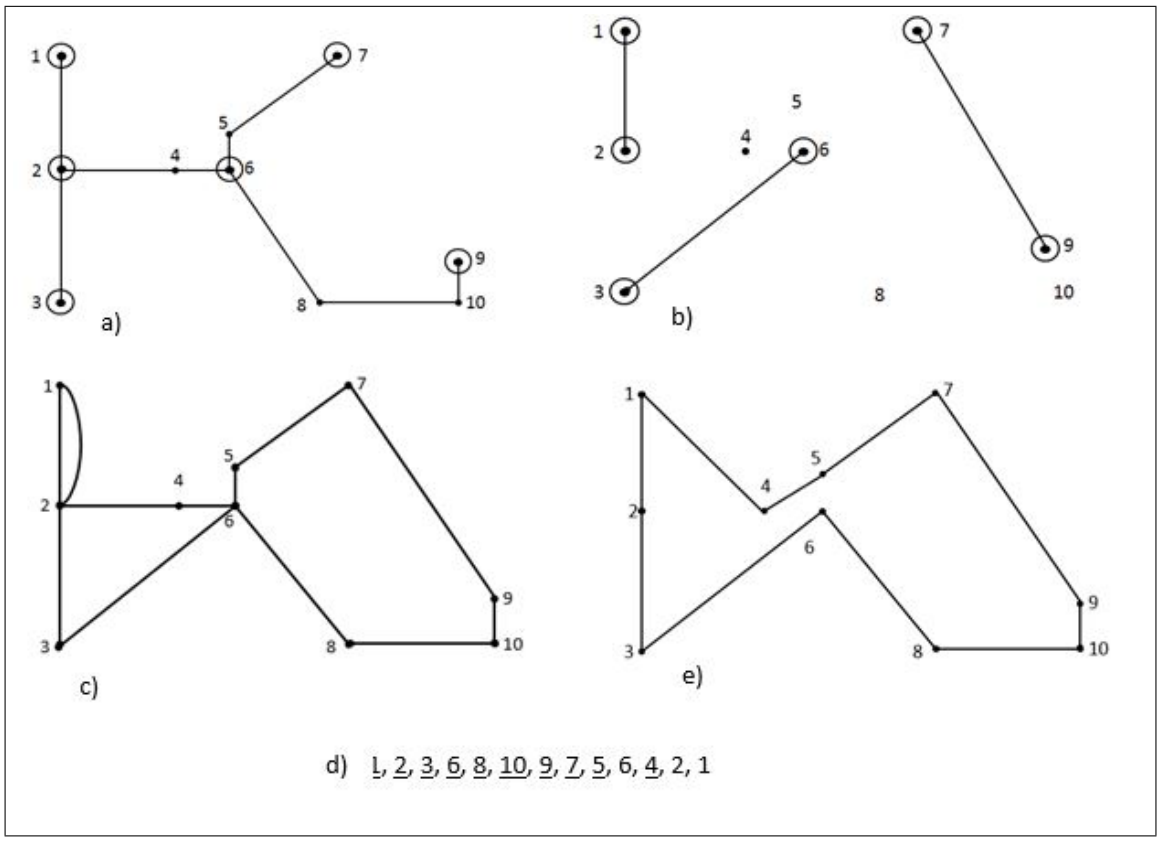

Figure 27: Christofides algorithm [26]

1. Find the minimum spanning tree (MST), the shortest path that will connect all the vertices in the graph G, representing the TSP.

2. Identify all vertices with an uneven number of edges in G, in Figure 27a) the nodes with uneven edges have been circled, these are 1; 2; 3; 6; 7 and 9.

3. Find the shortest set of edges to match all these vertices in pairs. In Figure 27b) edges 1-2; 3-6 and 7-9 were matched.

4. Add these edges to the graph G, where an edge is in both the MST and the matching edges, the edge is doubled. This can be seen in Figure 27c).

5. Construct an Eulerian walk, going through every edge exactly once. Figure 27d) gives the Eulerian walk or cycle for the example.

6. Delete all duplicate vertices in the Eulerian circuit. The remaining sequence of vertices is a Hamiltonian cycle that visits each vertex once and is a solution to the TSP, see Figure 27e).

2.3.3 2-Opt Improvement Heuristic

The 2-Opt Heuristic is probably the most basic local search heuristic for the TSP. 2-Opt starts with an arbitrary initial tour (T) and incrementally improves this tour by making successive improvements that exchange two of the edges in the tour with two other edges. Hence 2-Opt Heuristic generates a random initial tour (city permutation) and iteratively improves it until a local minimum is reached. Given the current tour T (the best tour) the 2-Opt Heuristic searches for a better tour within the neighborhood $N(T)$ of T. Whenever a better tour found within $N(T)$ then current best solution becomes the new best and the process is repeated from the current best T. The current T is considered a local minimum if no neighboring solution is found in $N(T)$ that replaces the current best tour T. $N(T)$ is the set of all two adjacent tours.

What are two adjacent tours? Two TSP tours are called 2-adjacent if one can be obtained from the other by deleting two edges and adding two edges. A TSP tour T is called 2-optimal if there is no 2-adjacent tour to T with lower cost than T.

Basically, the 2-Opt Heuristic looks for a 2-adjacent tour with lower cost than the current tour. If one is found, then it replaces the current tour[6]. This continues until there is a 2-optimal tour. The algorithm terminates in a local optimum in which no further improving step is possible. An example showing how a 2-adjacent tour in $N(T)$ is obtained is shown in Fig. 28. The neighborhood $N(T)$ consists of all such 2-adjacent tours for a given tour T.

In each iteration, the 2-Opt Heuristic applies best possible 2-opt move. This means finding the best pair of edges $(i, i+1)$ and $(j, j+1)$ (not adjacent) such that replacing them with the edges (i, j) and $(i+1, j+1)$ minimizes the tour length[7]. The 2-Opt Heuristic stops when no such tour is found in $N(T)$. Notice that one only needs to check: dist$(i, i+1)$+dist$(j, j+1)$ >dist(i, j)+dist$(i+1, j+1)$ (where dist means distance or edge length i.e. sub tour length), and not the entire tour length in order to determine the neighboring tour is better.

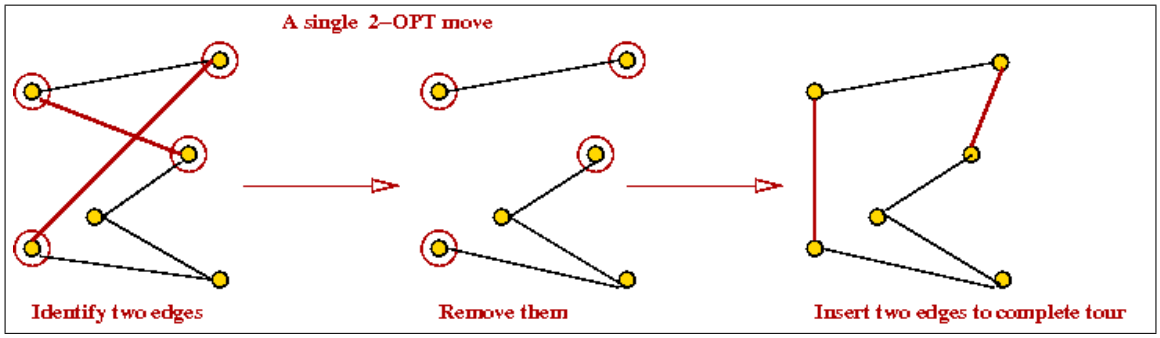

Figure 28: The 2-Opt move

The 2-Opt Heuristic can be written in a general form as follows. Improvement heuristics are based on searching the neighborhood $N(T)$. In this case, the $N(T)$ consists of all tours that can be obtained from T deleting two arcs and inserting two arcs.

Consider the following example of 7 city TSP. Let $T = (c_1, c_2, c_3, c_4, c_5, c_6, c_7, c_1)$ where we assume that the starting city is c_1 which is fixed. For i=1 and j=3,4,5,6 the

[6]This is known as the 2-Opt Move.

[7]This procedure is then iterated until no such pair of edges is found. The resulting tour is called 2-optimal. Note that the global optimum tour is 2-optimal, of course.

2-adjacent neighbors obtained considering all $(i, i+1)$ and $(j, j+1)$ are:

$$\begin{array}{ll} (c_1, c_3, c_2, c_4, c_5, c_6, c_7, c_1), & (c_1, c_4, c_3, c_2, c_5, c_6, c_7, c_1), \\ (c_1, c_5, c_4, c_3, c_2, c_6, c_7, c_1), & (c_1, c_6, c_5, c_4, c_3, c_2, c_7, c_1) \end{array} \tag{49}$$

Similarly for i=2 and j=4,5,6 the 2-adjacent neighbors obtained considering all $(i, i+1)$ and $(j, j+1)$ are:

$$\begin{array}{ll} (c_1, c_2, c_4, c_3, c_5, c_6, c_7, c_1), & (c_1, c_2, c_5, c_4, c_3, c_6, c_7, c_1), \\ (c_1, c_2, c_6, c_5, c_4, c_3, c_7, c_1), & \end{array} \tag{50}$$

For i=3 and j=5,6 the 2-adjacent neighbors obtained considering all $(i, i+1)$ and $(j, j+1)$ are: $(c_1, c_2, c_3, c_5, c_4, c_6, c_7, c_1), (c_1, c_2, c_3, c_6, c_5, c_4, c_7, c_1)$. For i=4 and j=6, the only 2-adjacent neighbor is $(c_1, c_2, c_3, c_4, c_6, c_5, c_7, c_1)$. Hence $N(T)$ (the 2-adjacent neighbors of T) consists of all above 10 tours.

The Basic Steps for the 2-Opt Heuristic

Start with tour T
If there is a tour $\hat{T} \in N(T)$ such that dist($\hat{T}$)<dist(T), then replace T by $\hat{T}$ and repeat
Otherwise, quit with a locally optimal solution.

The Detailed Steps for the 2-Opt Heuristic

Step 1. Create an initial $T = (c_1, c_2, \cdots, c_n, c_1)$
Step 2. Set i=1, D be the length of tour T
Step 3. Set $j = i + 2$;
Step 4. Break th link (c_i, c_{i+1}) and (c_j, c_{j+1}), as shown in Fig. 28, and create new tour $\hat{T}$=$(c_1, c_2, \cdots, c_i, c_j, \cdots, c_{i+1}, c_{j+1}, c_{j+2}.., c_1)$. If cost($\hat{T}$) $< D$ then set $T := \hat{T}$, update D, and go to Step 2. Else go to Step 5.
5. Set $j = j + 1$. If $j < n$ then go to Step 4. In the opposite case increase i by 1 (i.e. $i = i + 1$). If $i < (n - 2)$ then go to Step 3. Otherwise finish

---

The frame-work of the 3-Opt Heuristic can be summarized as follow:

- 3-opt neighborhood: Two TSP tours are called 3-adjacent if one can be obtained from the other by deleting three edges and adding three edges.
- A TSP tour T is called 3-optimal if there is no 3-adjacent tour to T with lower cost than T.
- 3-Opt Heuristic looks for a 3-adjacent tour with lower cost than the current tour. If one is found, then it replaces the current tour. This continues until there is a 3-optimal tour.

Remark 2.2. *In the next chapter, we will present the simulated annealing (SA) algorithm and the genetic algorithm (GA) for solving TSPs. Pseudo-codes of SA and GA will also be provided.*

2.4 Vehicle Routing Problem

Vehicle routing problems generalize the TSP. In the most basic vehicle routing problem, customers at given geographical locations are to be supplied from a depot using identical vehicles each with a capacity of Q. Each customer i has a demand of d_i, and the total demand of customers assigned to the same delivery vehicle cannot exceed Q. The cost the travel between each pair of customers and also between the depot and each customer are given. It is required to assign each customer to a delivery route of one vehicle. There are two objective functions: one is to minimize the number of vehicles used, and the other is to minimize the total cost of travel.

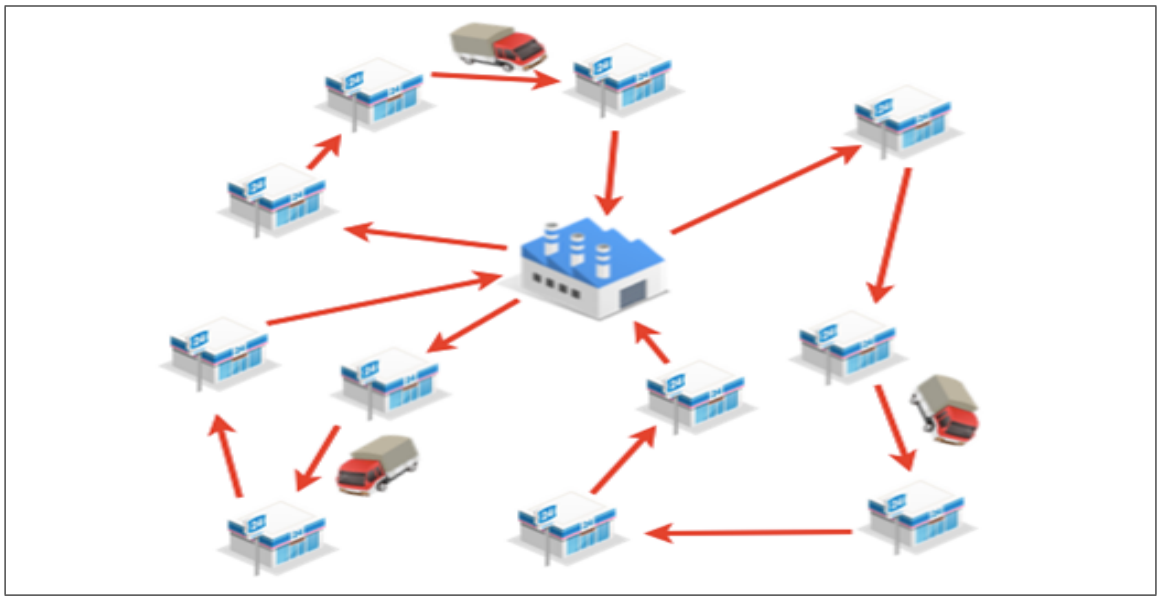

Figure 29: The vehicle routing problem

There are many generalizations of the basic problem. For example

- there is a time window for each delivery during which a vehicle must visit the customer
- there is a non-homogeneous fleet of vehicles with each having its own capacity
- some deliveries involve collection instead of delivery, in which case the vehicle capacity must be satisfied after each collection.

Remark 2.3. *We will not present the mathematical formulation of the VRP as this can be obtained as a generalization of the model for TSP.*

2.5 Facility Location Problem (FLP)

Facility location is defined by Ceselli et al. [20] as a collective term for problems with the goal of finding the best locations to place facilities given a company's distribution network. Facilities are sometimes referred to as plants, depots, warehouses or distribution centres. The aim of FLP is to assign sites (also called customers or demand points) to

the depots, based on a given criteria such as shortest distance, minimizing costs or maximizing coverage. The FLP assumes that all customers are to be serviced separately using single trips between the depots and back. This can be seen in Fig. 30.

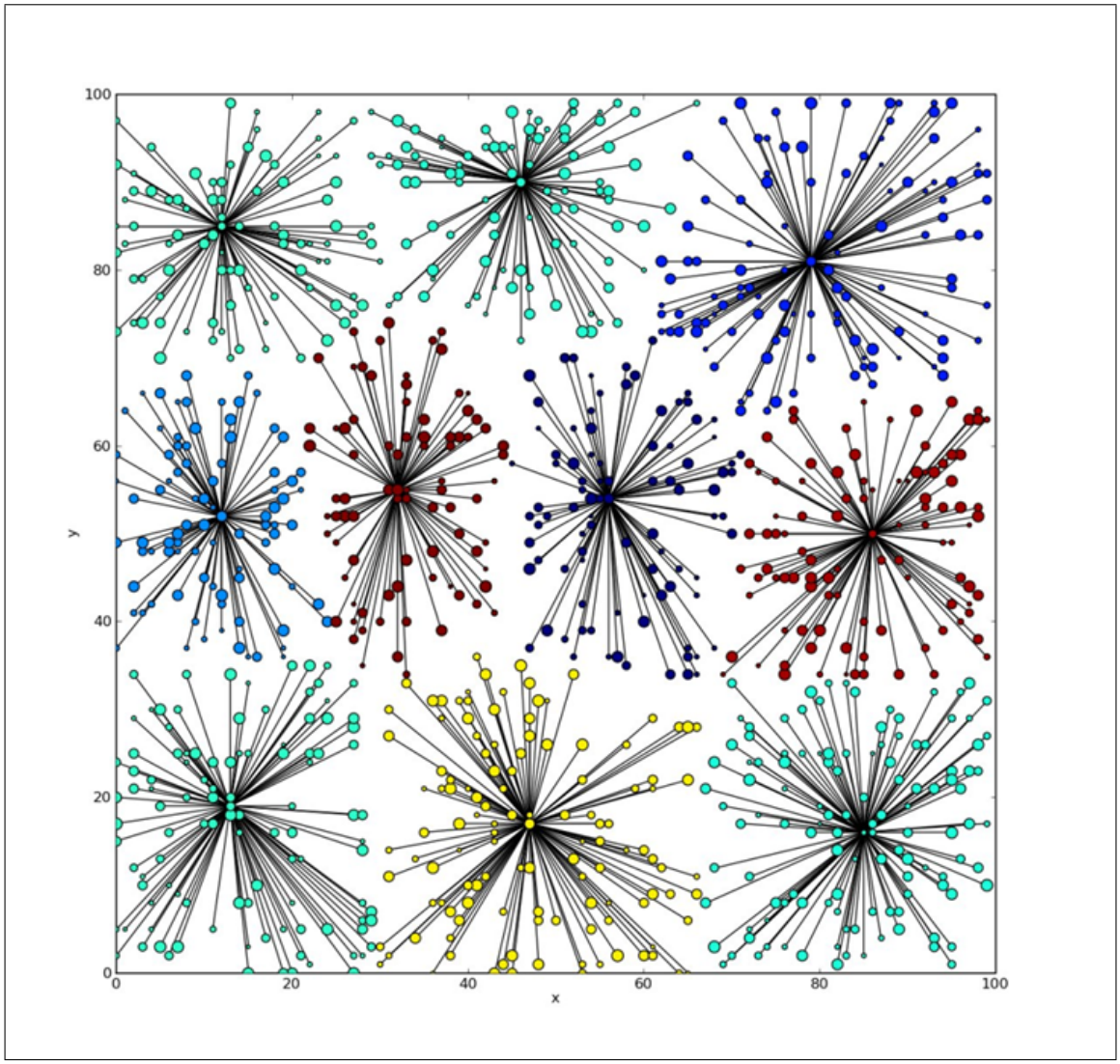

Figure 30: Placement of facilities around customers

A customer's demand can be met by various depots or restricted to a singular depot. The latter is called the single-source FLP. The problem can have no depot capacity supply restrictions (called the uncapacitated facility location problem or UFLP) or have depot capacity constraints, referred to as the capacitated facility location problem (CFLP).

Facility location problems require locations for depots/service centres to be chosen from a set of potential given locations. One of the classical facility location problems is the plant location problem (fixed charged facility location problem). There are m potential facility locations and n customers. There is a fixed cost f_i of establishing a facility at potential location i. Moreover, there is a cost c_{ij} of supplying customer j from a facility established at potential location i. The problem can be formulated as a zero-one programming problem (i.e. BIP) as:

$$\text{min} \quad \sum_{i=1}^{m}\sum_{j=1}^{n} c_{ij}x_{ij} + \sum_{i=1}^{m} f_i y_i \tag{51}$$

$$\text{subject to} \quad \sum_{i=1}^{m} x_{ij} = 1, \qquad \forall j \in \{1, 2, \cdots, n\}, \tag{52}$$

$$x_{ij} \leq y_i, \qquad \forall i \in \{1, 2, \cdots, m\}, \forall j \in \{1, 2, \cdots, n\}, \tag{53}$$

$$\sum_{j=1}^{n} d_j x_{ij} \leq W_i y_i, \qquad \forall i, \tag{54}$$

$$x_{ij}, y_i \in \{0, 1\}, \qquad \forall i, j, \tag{55}$$

where the decision variables are binary integers and given by

$$y_i = \begin{cases} 1 & \text{if a facility is established at the potential location } i \\ 0 & \text{otherwise,} \end{cases} \tag{56}$$

$$x_{ij} = \begin{cases} 1 & \text{if customer } j \text{ is served from a facility at potential location } i \\ 0 & \text{otherwise.} \end{cases} \tag{57}$$

The above mathematical model is for the constrained (see the capacity constraint (54)) single source FLP (see constraint (52)). The capacity can differ per depot. This is referred to as W_i while d_j is the demand of the customer j. The total demand of customers assigned to a depot i should be equal or less than W_i as shown in constraints (54).

In the fixed charge FLP, the goal is to minimize costs that also include fixed depot costs. It can be seen that in the objective function (51) the first term is related to the distribution costs from depots to customers while the second term is the fixed costs due to the establishment of depots. The number of depots becomes a trade-off between the fixed costs and distribution costs which will be obtained by optimization. If there are no capacity constraints, customers should be assigned to the closest open depot in order to minimize distribution costs (if the cost is calculated based on distance).

A related facility location problem is the p-median problem in which it is specified that facilities are to be established at p of the potential locations. There are no fixed costs of establishing facilities. The problem is formulated using the same variables as for the above plant location problem as:

$$\text{min} \quad \sum_{i=1}^{m}\sum_{j=1}^{n} c_{ij}x_{ij} \tag{58}$$

$$\text{subject to} \quad \sum_{i=1}^{m} x_{ij} = 1, \qquad \forall j \in \{1, 2, \cdots, n\}, \tag{59}$$

$$x_{ij} \leq y_i, \qquad \forall i \in \{1, 2, \cdots, m\}, \forall j \in \{1, 2, \cdots, n\}, \tag{60}$$

$$\sum_{j=1}^{n} d_j x_{ij} \leq W_i y_i, \qquad \forall i, \tag{61}$$

$$\sum_{i=1}^{m} y_i = p, \tag{62}$$

$$x_{ij}, y_i \in \{0, 1\}, \qquad \forall i, j. \tag{63}$$

The number of depots to select is restricted to p. In the p-median problem costs c_{ij} can be treated as distances between the depot i and customer j. The objective is to minimize the distance from the chosen p depots to the customers assigned to them. Each customer must be assigned to a depot and can only be assigned to one depot to enforce the single-source constraints.

Remark 2.4. *The facility location problems assign sites (also called customers or demand points) to the depots, based on a given criteria such as shortest distance, minimizing costs or maximizing coverage. Different variants have different objective functions. In the location of emergency facilities, often referred to as a centre problem, the maximum distance to a customer replaces the sum of distances. In a maximum cover problem, a customer within a given distance of a facility is said to be covered, and it is required to maximize the number of customers who are covered.*

2.6 Set Covering Problem

The set covering problem is a combinatorial optimization problem which has many practical applications. The problem can be formulated as integer programming problem mathematically. We begin with describing a cover of a set before presenting the problem formally. Let $Q = \{S_1, S_2, \cdots, S_n\}$ be n subsets of a given set S, $S_j \subseteq S$, $j = 1, 2, \cdots, n$. Let T be an index set such that $T \subseteq \{1, 2, \cdots, n\}$. A cover of S is a subfamily $S_j, j \in T$ such that $\cup_j S_j = S$ (the goal in the set covering problem is to select as few subsets as possible from Q such that their union covers S). Assume that each subset S_j has a cost $c_j > 0$ associated with it. We define the cost of a cover to be the sum of the costs of the subsets included in the cover.

The problem of finding a cover of minimum cost is of particular practical significance, as can be seen in the following description (as well as the hospital example that follows). Let there are some duties $S = \{1, 2, \cdots, m\}$ to be performed (at the least cost) and a

set $\{S_1, S_2, \cdots, S_n\}$ of work patterns is given. Such problem arises in many practical situations. The set covering is useful for air crew scheduling where the duties are flight legs and each work pattern is a feasible collection of flight legs that can be handled by a single crew.

We present the mathematical formulation of the above optimization problem by introducing a matrix $A = (a_{ij})$ from available information (given data) of the of the problem as:

$$a_{ij} = \begin{cases} 1 & \text{if the duty } i \text{ is included in the work pattern } S_j, \\ 0 & \text{otherwise.} \end{cases} \tag{64}$$

We define the integer variables x_j

$$x_j = \begin{cases} 1 & \text{if the work pattern } S_j \text{ is chosen,} \\ 0 & \text{otherwise.} \end{cases} \tag{65}$$

Hence x_j=1 (0) means the set S_j is included (respectively not included) in the cover. The mathematical formulation of the problem is as follows:

$$\text{min} \qquad \sum_{j=1}^{n} c_j x_j \tag{66}$$

$$\text{subject to} \qquad \sum_{j=1}^{n} a_{ij} x_j \geq 1, \qquad \forall i \in \{1, 2, \cdots, m\}, \tag{67}$$

$$x_j \in \{0, 1\}, \qquad \forall j \in \{1, 2, \cdots, n\}. \tag{68}$$

Take the following example and study its mathematical model to clear up understanding of the problem.

A hospital in Johannesburg needs to keep doctors on call, so that a qualified individual is available to perform every medical procedure that might be required (there is an official list of such procedures). For each of several doctors available for on-call duty, the additional salary they need to be paid, and which procedures they can perform, is known. The goal to choose doctors so that each procedure is covered, at a minimum cost. A set covering problem instance can be created using the data given in the following table:

Table 2: Set Covering Problem Instance

Procedure	Doc_1	Doc_2	Doc_3	Doc_4	Doc_5	Doc_6
P_1	✓			✓		
P_2	✓				✓	
P_3		✓	✓			
P_4	✓					✓
P_5		✓	✓			✓
P_6		✓				

In the above problem instance there are m procedures $\{P_1, P_2, \cdots, P_6\}$ and n available doctors $\{Doc_1, Doc_2, \cdots, Doc_6\}$. The ✓ denotes the corresponding doctor can perform the procedure in column 1. The problem data can be represented by the matrix

$A = (a_{ij})$ where $a_{ij} = 1$ if the doctor j can perform the procedure P_i and 0 otherwise. Also, let c_j , $j = 1, 2, \cdots, n$ be the additional salary that will need to be paid to doctor j for on-call duty. Let S_j be the set of procedures that can be performed by Doc_j. Hence the variable x_j=1 if doctor j is on call (i.e. S_j is included in the cover), and 0 otherwise. Clearly the Eqs. (66)-(68) have the following meaning:

- Eq. (66) means the minimum salary needs to be paid.
- Eq. (67) means at least one doctor must perform the procedure P_i (i.e at least one cover must include the Procedure P_i).
- Eq. (68) means an appropriate number of doctors need to be chosen.

2.7 Winner Determination Problem

The winner determination problem (WDP) arises in combinatorial auctions. It is one of the most challenging problems and is known to be NP-hard.

Auctions are an important class of market mechanisms. In combinatorial auctions, bidders are allowed to bid for a combination of items referred to as a bundle of items for a price, and the auctioneer selects a subset of those bids to maximize the revenue of the auctioneer. A bidder of a selected bid is called a 'winner'. The combinatorial auctions have a wide range of practical applications such as supply chain management [21], allocation of airport take-off and landing time slots [14], trading [11], sensor management [12, 16], resource allocation with real-time constraints [13].

The WDP is formally defined as follows: Let $S = \{1, 2, \cdots, m\}$ be a set of m items to be auctioned and $B = \{B_1, B_2, \cdots, B_n\}$ be a set of bids submitted by the bidders. Henceforth, the indices i and j will be used to denote items and bids, respectively. Each bid $B_j \in B$ is a tuple $\prec S_j, p_j \succ$ where S_j is a set of items, and $p_j \geq 0$ is the price of the items of S_j. The WDP is to decide which bids win and which ones lose, so as to maximize the auctioneer's revenue where each item can be allocated to at most one bidder.

Take $A = (a_{ij})$ to be a m rows and n columns 0-1 matrix, where $a_{ij} = 1$ iff the item i belongs to S_j, $a_{ij} = 0$, otherwise. Let $x_j \in \{0, 1\}, j = 1, \cdots, n$, be variables such that $x_j = 1$ if the bid B_j is accepted (a winning bid), and $x_j = 0$ otherwise (a losing bid). The WDP can be formulated as following integer program:

$$\begin{cases} \max_x & \sum_{j=1}^{n} p_j x_j \\ s.t. & \sum_{j=1}^{n} a_{ij} x_j \leq 1, i \in \{1, \cdots, m\}, \\ & x_j \in \{0, 1\}, j \in \{1, \cdots, n\}. \end{cases}$$

2.8 Quadratic Assignment Problem

The QAP is one of the fundamental combinatorial optimization problems. It was introduced in 1957 by Koopmans and Beckmann [17]. The QAP considers the problem of allocating a set of n facilities to a set of n locations, with the cost being a function of the distance and flow between facilities, plus costs associated with a facility being placed at a certain location. The formal definition of the problem is as follows. Let n be the number of facilities and locations, $F = (f_{ij})_{1 \leq i,j \leq n}$, $D = (d_{kl})_{1 \leq k,l \leq n}$ and $B = (b_{ik})_{1 \leq i,k \leq n}$ be three $n \times n$ matrices, where f_{ij} is the flow between the facilities i and j, d_{kl}, the distance

between the locations k and l, and b_{ik} the cost of the facility i being placed at the location k. Let $\mathscr{S}_n$ be the set of all the permutations $\phi : \{1, \ldots, n\} \longrightarrow \{1, \ldots, n\}$. The QAP was originally defined as follows:

$$\min_{\phi \in \mathscr{S}_n} \quad \sum_{i=1}^{n} \sum_{j=1}^{n} f_{ij} d_{\phi(i)\phi(j)} + \sum_{i=1}^{n} b_{i\phi(i)}, \tag{69}$$

The term $b_{i\phi(i)}$ in (69) is the cost associated with placing facility i at location $\phi(i)$. On the other hand, the product $f_{ij}d_{\phi(i)\phi(j)}$ represents the cost of placing facility j at location $\phi(j)$ while facility i is placed at location $\phi(i)$. In the context of facility location the matrices F and D are symmetric with zeros in the diagonal, and all the matrices are nonnegative.

A more general version of the QAP was introduced by Lawler [18]. In this version we are given a four-dimensional array or matrix $C = (c_{ijkl})$ of coefficients instead of the two matrices F and D and the problem can be stated as

$$\min_{\phi \in \mathscr{S}_n} \quad \sum_{i=1}^{n} \sum_{j=1}^{n} c_{ij\phi(i)\phi(j)} + \sum_{i=1}^{n} b_{i\phi(i)}. \tag{70}$$

Clearly, a Koopmans-Beckmann problem QAP in Eq. (69) can be formulated as a Lawler QAP Eq. (70) by setting $c_{ijkl} = f_{ij}d_{kl}$ for $i \neq j$ or $k \neq l$ and $c_{iikk} = f_{ii}d_{kk} + b_{ik}$ otherwise. Although extensive research has been done for more than three decades, the QAP, in contrast with its linear counterpart the linear assignment problem (LAP), remains one of the hardest optimization problems and no exact algorithm can solve problems of size $n > 20$ in reasonable computational time. A lot of interests have been given to the QAP since its introduction in 1957. The QAP has been used as the mathematical model of many real life problems arising in facility location, computer manufacturing, scheduling, building layout design, process communications.

2.8.1 Mathematical Formulations

Since its introduction, the QAP has been formulated in many different ways ranging from the linear form to the SDP form. Here we present some selected mathematical formulations.

Quadratic Integer Programming Formulation:

Considering the fact that for every permutation of $\{1, \ldots, n\}$, there is a corresponding element X in $\mathscr{X}_n$, where $X = (x_{ij})_{1 \leq i,j \leq n}$. The permutation ϕ in the original QAP formulation (69) can therefore be replaced by a permutation matrix $X = (x_{ij})_{1 \leq i,j \leq n}$ where

$$x_{ij} = \begin{cases} 1 & \text{if facility } i \text{ is placed at location } j, \\ 0 & \text{otherwise.} \end{cases} \tag{71}$$

Using this notation, Koopmans and Beckmann [17] gave the following quadratic integer programming formulation:

$$\min \quad \sum_{i=1}^{n} \sum_{j=1}^{n} \sum_{k=1}^{n} \sum_{l=1}^{n} f_{ij} d_{kl} x_{ik} x_{jl} + \sum_{i=1}^{n} \sum_{j=1}^{n} b_{ij} x_{ij}, \qquad s.t. \tag{72}$$

$$\sum_{i=1}^{n} x_{ij} = 1 \qquad \text{for } j = 1, \ldots, n, \tag{73}$$

$$\sum_{j=1}^{n} x_{ij} = 1 \qquad \text{for } i = 1, \ldots, n, \tag{74}$$

$$x_{ij} \in \{0, 1\} \qquad \text{for } i, j = 1, \ldots, n, \tag{75}$$

where f_{ij} is the flow between facility i and facility j, d_{kl} the distance between location k and location l and b_{ij} the cost of placing facility i at location j. This formulation can be found in most of the linearisation approaches for the QAP.

Trace Formulation:

A QAP in Koopmans-Beckmann form can be formulated in a more compact way if we define an inner product between matrices. Let the inner product of any two real $n \times n$ matrices A and B be defined by

$$\langle A, B \rangle = tr(AB^T) = \sum_{i=1}^{n} \sum_{i=1}^{n} a_{ij} b_{ij}. \tag{76}$$

Given the $n \times n$ matrix A, a permutation $\phi \in \mathscr{S}_n$ and the associated permutation matrix $X \in \mathscr{X}$, then AX^T and XA permute the columns and rows of A respectively, according to the permutation ϕ and therefore

$$XAX^T = (a_{\phi(i)\phi(j)}). \tag{77}$$

Considering a QAP instance with flow matrix F, distance matrix D and cost matrix B, we set $\bar{D} = XD^TX^T$, which leads to $\bar{d}_{ji} = d_{\phi(i)\phi(j)}$ for $i, j = 1, \ldots, n$. It then follows that

$$tr(FXD^TX^T) = tr(F\bar{D}) = \sum_{i=1}^{n} \sum_{j=1}^{n} f_{ij} \bar{d}_{ji} = \sum_{i=1}^{n} \sum_{j=1}^{n} f_{ij} d_{\phi(i)\phi(j)},$$

where ϕ is the permutation associated with the permutation matrix X.

Therefore the original formulation of the QAP can equivalently be reformulated in the following form:

$$\min \qquad tr\left[(FXD^T + B)X\right], \qquad s.t. \tag{78}$$

$$X \in \mathscr{X}_n, \tag{79}$$

which is equivalent to

$$\min \qquad tr\left[(FXD^T + B)X\right], \qquad s.t. \tag{80}$$

$$X^T e = e, \tag{81}$$

$$Xe = e, \tag{82}$$

$$x_{ij} \in \{0, 1\} \text{ for all } i, j, \tag{83}$$

where e is the column n–vector of ones. This formulation was introduced by Edward [19].

Remark 2.5. *Given any two real $n \times n$ matrices A and B, recall the well known properties $tr(AB) = tr(BA)$, $(AB)^T = B^T A^T$, $tr(A) = tr(A^T)$. Given $F = F^T$ we can write the quadratic term in Eq. (78) as*

$$tr(FXD^T X^T) = tr(FXDX^T), \tag{84}$$

where D is not necessarily symmetric. Therefore, given a QAP Instance where only one of the matrices is symmetric (say, F), we can transform it into a QAP instance where both matrices are symmetric.

Remark 2.6. *In the next chapter will be dealing with algorithms for combinatorial optimization problems where I will present a genetic algorithm for solving QAP using swap and insertion crossovers operations.*

2.9 Vertex Cover Problem

A vertex cover of an undirected graph $G = (V, E)$ is a subset of its vertices such that for every edge $(i, j) \in E$ of the graph, either i or j is in vertex cover. Although the name is vertex cover (sometimes node cover), the set covers all edges of the given graph. An edge is covered if one of its endpoint is chosen. Therefore a vertex cover of G is a set $S \subseteq V$ such that for each $(i, j) \in E$ either $i \in S$ or $j \in S$ (or the both in S). Given an undirected graph, the vertex cover problem is to find minimum size vertex cover. Any vertex cover problem can be changed into set cover problem (find minimum no of vertices that hit all edges). Hence, vertex cover is a special case of set cover (can you prove it?).

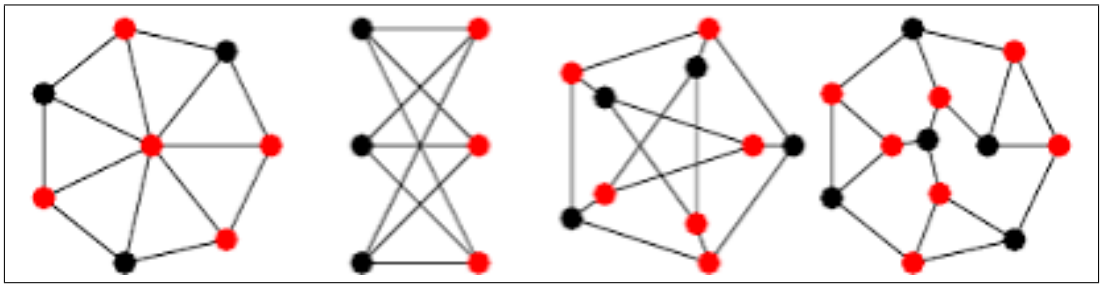

Figure 31: Examples of vertex cover of graph using red nodes

In the weighted version of the problem, a weight function $c : V \to R^+$ is given such that for $i \in V$ there is a weight c_i. The goal is to find a minimum weight vertex cover of G. The unweighted version of the problem is also known as Cardinality Vertex Cover. The mathematical formulation of the unweighted version can be formulated by introducing a x_i for each $i \in V$ where $x_i = 1$ means i in included in the cover (and zero means not included in the cover). Moreover at least one vertex of each arc (i, j) must be included in the vertex cover and thus we have the constraint $x_i + x_j \geq 1$. Hence the integer linear

programming (ILP) formulation is given by:

$$\text{min} \qquad \sum_{j=1}^{n} x_j \tag{85}$$

$$\text{subject to} \qquad x_i + x_j \geq 1, \qquad \forall (i,j) \in E, \tag{86}$$

$$x_j \in \{0,1\}, \qquad \forall j \in V. \tag{87}$$

The above problem can be relaxed to give rise a linear program (LP) by simply replacing Eq. (87) with $x_j \geq 0$. The ILP of the minimum weight vertex cover is given by:

$$\text{min} \qquad \sum_{i=1}^{n} c_i x_i \tag{88}$$

$$\text{subject to} \qquad x_i + x_j \geq 1, \qquad \forall (i,j) \in E, \tag{89}$$

$$x_i \in \{0,1\}, \qquad \forall i \in V. \tag{90}$$

A relax LP version is similarly found by relaxing Eq. (90). Let x be LP solution of problem (88)-(90). Let C is such that $C = \{i \in V : x_i \geq \frac{1}{2}\}$ then C is a cover? Why? We have $x_i + x_j \geq 1$ for any edge $(i,j) \in E$. We know $\max\{x_i, x_j\} \geq \frac{1}{2}$ due to the constraint (89). Clearly rounding of LP solution will put at least one of nodes $\{i,j\}$ in C. Let us define the following solutions. Let x is the solution of the WVC (weighter vertex cover) associated with the set C i.e. the rounding solution of LP. Let also x^* be the optimal solution of WVC. Let $\hat{x}$ be the relaxed solution of LP relaxation. Then we have the following relationships:

$$c^T x = \sum_{\hat{x}_i \geq \frac{1}{2}} c_i \leq \sum_{\hat{x}_i \geq \frac{1}{2}} 2\hat{x}_i c_i \leq 2 \sum_{i=1}^{n} \hat{x}_i c_i = 2c^T \hat{x} \leq 2c^T x^*$$

This shows that the solution in C is 2-approximation solution of the minimum weight vertex cover problem.

There are two versions of the vertex cover problem: the decision and optimization versions. In the optimization version vertex cover is defined as finding the minimum number of vertices that hit all edges of G. In the decision version, the task is to verify for a given graph G whether there exists a vertex cover of a specified size k.

The decision problem is known as the k-vertex cover. Can one find at most k vertices that hit all edges of G. The greedy option will be to consider all subsets of vertices of size k. This will take the running time $\mathcal{O}(n^k)$. This parameterization with k is very useful in designing efficient greedy rules for solving the k-vertex cover problem. The greedy rule is as follows:

(a) find a vertex v of degree at least $k+1$, add v to the cover and delete it from the graph G (with all adjacent edges),

(b) repeatedly apply (a) until no vertices of degree greater than k.

What is left in the remaining graph is known as kernel which can be solved to find the cover of size at most k in running time less than $\mathcal{O}(n^k)$.

The minimum vertex cover (weighted and unweighted versions) problem has many applications. A direct application is placing ATMs in a (big) city. Each additional ATM costs money to install and operate. The bank wants to have an ATM in every street (block, district). Where should ATMs be placed so that the bank needs as little ATMs as possible?

2.10 Graph Coloring Problem

In the graph colouring problem, it is required to assign colours to the vertices/nodes of a graph such that adjacent nodes are assigned different colours as can be seen in Fig. 32. The colouring should minimize the number of colours used.

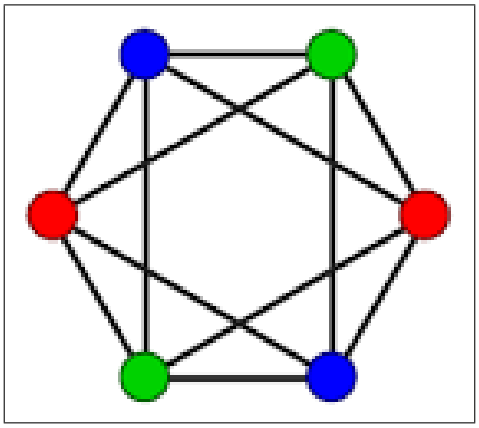

Figure 32: Coloring of graph with minimum no of colors

The graph colouring problem has applications in timetabling. In this application, the vertices/nodes correspond to the classes to be timetabled (colored). Two vertices/nodes that have students or teachers in common are connected by an edge, meaning that they cannot be timetabled to take place at the same time slot (or cannot be colored with the same color), see Fig. 33. The minimum number of colours represents the minimum number of time periods to timetable all classes.

The graph coloring (or the node coloring) problem can be formulated as an ILP. Let $V = \{1, 2, \cdots, n\}$ and that we have m colors at our disposal (in general m colors are at disposal with $m \leq n$). We introduce binary variables y_k, $k = 1, 2, \cdots, m$, to indicate if color k is used ($y_k = 1$) or not ($y_k = 0$). Furthermore we introduce variables x_{ik} to indicate whether node i receives color k ($x_{ik} = 1$, 0 if does not receives color k). The

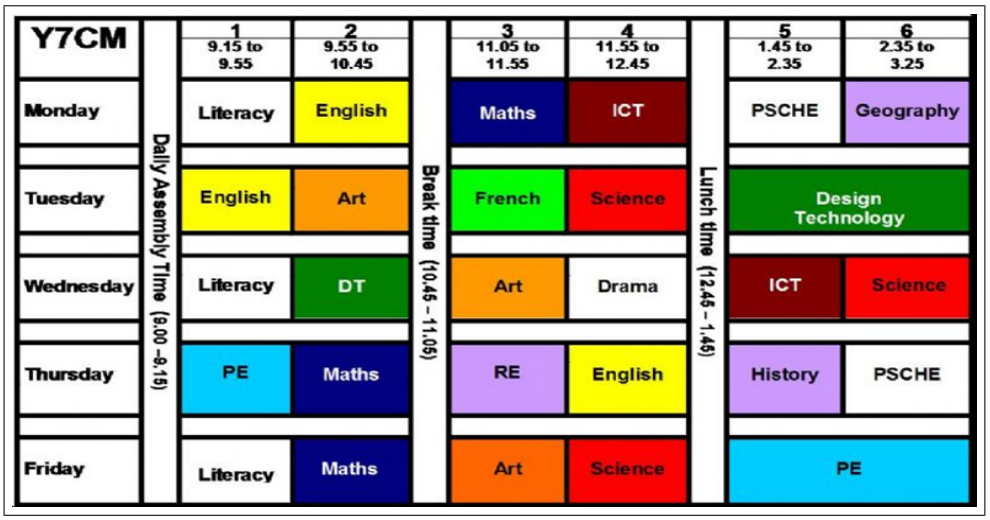

Figure 33: Example of graph coloring problem

formulation is as follows:

$$\text{min} \qquad \sum_{k=1}^{m} y_k, \tag{91}$$

$$\text{subject to} \qquad x_{ik} \leq y_k, \forall i \in V, k = 1, 2, \cdots, m, \tag{92}$$

$$x_{ik} + x_{jk} \leq 1, \quad \forall (i,j) \in E, k = 1, 2, \cdots, m, \tag{93}$$

$$\sum_{k=1}^{m} x_{ik} = 1, \ \forall i \in V, \tag{94}$$

$$y_k, x_{ik} \in \{0, 1\}, \ \forall i \in V, k = 1, 2, \cdots, m. \tag{95}$$

Consider the following problem. You have the map of all the counties in Gauteng. What is the fewest number of colors need to color all of the counties so that no counties with a common border have the same color? Here the common border between two counties i and j means the arc (i, j). Although there are many counties only few colors are needed. Why? In this graph coloring problem $G = (V, E)$ where $V = \{1, 2, \cdots, n\}$ is the set of counties and E is the set of arch, $(i, j) \in E$ if counties i and j are adjacent. Write an integer program whose solution gives the minimum number of colors to color a map of Gauteng. The formulation (91)-(95) can be solved for this problem, where the constraint (92) means if county i is assigned color k, then color k is used. The constraint (93) means if counties i and j share a common boundary, then they are not both assigned color k.

The graph coloring problem is frequently used in time tabling problem such as exam time table scheduling (coloring problem). Consider the following problem. The University of Witwatersrand has to schedule 500 exams in 28 exam periods so that there are no exam conflicts. Here $V = \{1, 2, \cdots, 500\}$ is the set of exams and m=28 colors (or exam periods in this case). E is a set of arch i.e. $(i, j) \in E$ if a student needs to take exam i and exam j. The ILP model for this problem is the model presented in (91)-(95) with the

following definition:

$$x_{ik} = \begin{cases} 1 & \text{if exam } i \text{ is assigned in period } k, \\ 0 & \text{otherwise.} \end{cases} \tag{96}$$

where $k \in \{1, 2, \cdots, 28\}$.

2.11 Graph Partitioning Problem

There are a number of different graph partitioning problems of practical interest (see the max-cut problem in the next sub-section). Here, we will discuss the simplest one namely the maximum bisection problem and present its mathematical formulation.

Let $G = (V, E)$ be an undirected and connected graph, where $V = \{1, \cdots, n\}$, and E is the edge set. If the edge $(i, j) \in E$ connects vertices $i, j \in V$, we associate a weight $w_{ij} \geq 0$ with the edge. Assume that G has an even number of vertices. We denote the number of edges by m. The max-bisection problem is to find a partition of the vertices in two equally sized subsets (called the cut sets) such that the sum of the weights of the edges between vertices in the different subsets is maximized.

The max-bisection problem consists of partitioning V into two disjoint subsets V_1 and V_2 with the same cardinality while maximizing the sum of weights of the edges between V_1 and V_2. Let $x_i \in \{1, -1\}$ for $i = 1, 2, \cdots, n$, be variables with $x_i = 1$ if $i \in V_1$ and $x_i = -1$ if $i \in V_2$.

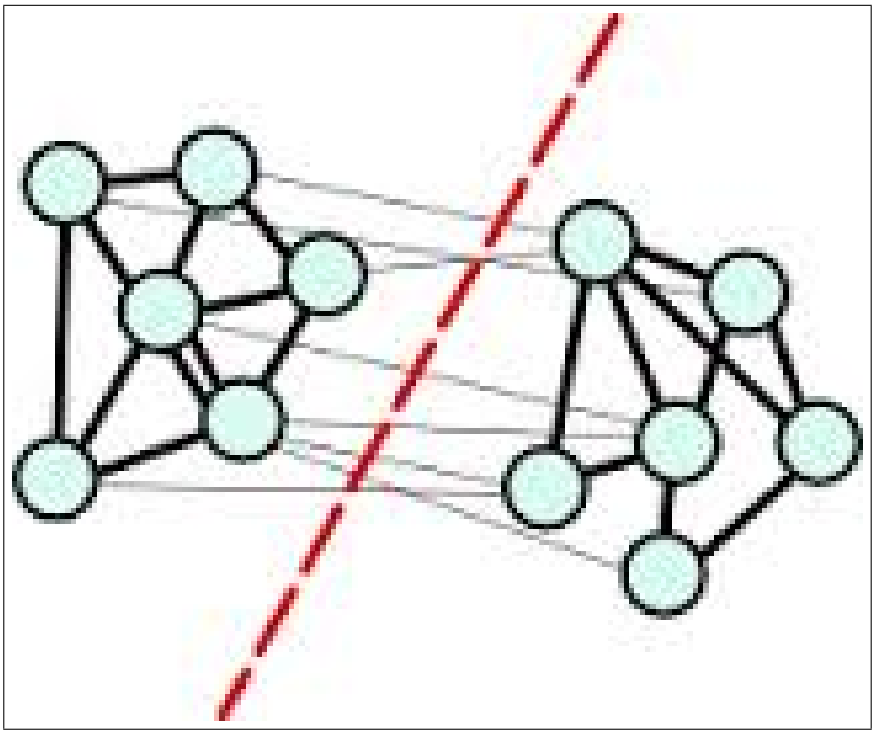

Figure 34: Example of graph bisection

Thus the maximum bisection problem can be formulated as the following integer

quadratic program [23, 24]:

$$\begin{cases} \max_x & \frac{1}{4}\sum_{i=1}^{n}\sum_{j=1}^{n} w_{ij}(1 - x_i x_j) \\ s.t. & \sum_{i=1}^{n} x_i = 0, \\ & x_i \in \{1, -1\}, i \in \{1, \cdots, n\}. \end{cases}$$

Note that x_i takes a value either 1 or -1, so the constraint $\sum_{i=1}^{n} x_i = 0$ ensures that the solution $x = (x_1, x_2, \cdots, x_n)^T$ is a bisection. Let $S = \{1, -1\}^n$ the feasible solution set of the above problem is given by: $\{x \big| x \in S, \sum_{i=1}^{n} x_i = 0\}$. Let $x = (x_1, x_2, \cdots, x_n)^T \in S$ and its symmetric solution $\bar{x} = (-x_1, -x_2, \cdots, -x_n)^T \in S$. It can be seen that if x is feasible so is $\bar{x}$.

2.12 Max-Cut Problem

The maximum cut (Max-Cut) problem is one of the simplest graph partitioning problems to conceptualize, and yet it is one of the most difficult combinatorial optimization problems to solve [25]. The objective of Max-Cut is to partition the set of vertices of a (undirected) graph $G = (V, E)$ into two subsets, such that the sum of the weights (cut value) of the edges having one endpoint in each of the subsets is maximum. This problem is known to be NP-Complete. However, it is interesting to note that the inverse problem, i.e., that of looking for the minimum cut in a graph is solvable in polynomial time using network flow techniques (Ford Fulkerson Algorithm). Max-Cut is an important combinatorial problem and has applications in many fields including VLSI circuit design and statistical physics [22].

We start with the more general version of the Max-Cut problem called the Max-k-Cut problem (see Fig. 35) which reduces to Max-Cut (Max-2-Cut) for k=2. Let the graph G be an undirected and connected graph, where $V = \{1, \cdots, n\}$, and $E \subseteq \{(i, j) \big| 1 \leq i < j \leq n\}$. If the edge $(i, j) \in E$ connects vertices $i, j \in V$, we associate a weight w_{ij} with the edge. The max-k-cut problem is to partition V into subsets $\{V_1, V_2, \cdots, V_k\}$, $k \in [2, n]$, $V_i \neq \emptyset$, $V_i \cap V_j = \emptyset$, $\forall i \neq j$, such that the weight function

$$w(V_1, V_2, \cdots, V_k) = \sum_{1 \leq r < s \leq k} \sum_{i \in V_r, j \in V_s} w_{ij}, \tag{97}$$

is maximized, where the edge weights $w_{ij} = w_{ji}$ are such that $w_{ij} = 0$ for $[i, j] \notin E$. The problem (97) is referred to as unweighted max-k-cut when $w_{ij} \in \{0, 1\}$, and it is referred to as Max-Cut when k=2. The unweighted Max-k-Cut version is equivalent to maximum-k-colorable subgraph, see [21].

We now present the Max-Cut problem as a mathematical optimization problem. In the Max-Cut problem a cut as a partition of V into two disjoint subsets (not necessarily equal) S and $\bar{S}$ (=$V \setminus S$). The weight (cut value) of the cut $(S, \bar{S})$ is given by the function $W : S \times \bar{S} \to R$ and is defined as

$$W(S, \bar{S}) = \sum_{i \in S,\, j \in \bar{S}} w_{ij}.$$

Clearly, a maximum cut is defined by the following optimization problem:

$$\text{Max-Cut} = \max_{\forall S \subseteq V} W(S, V \setminus S).$$

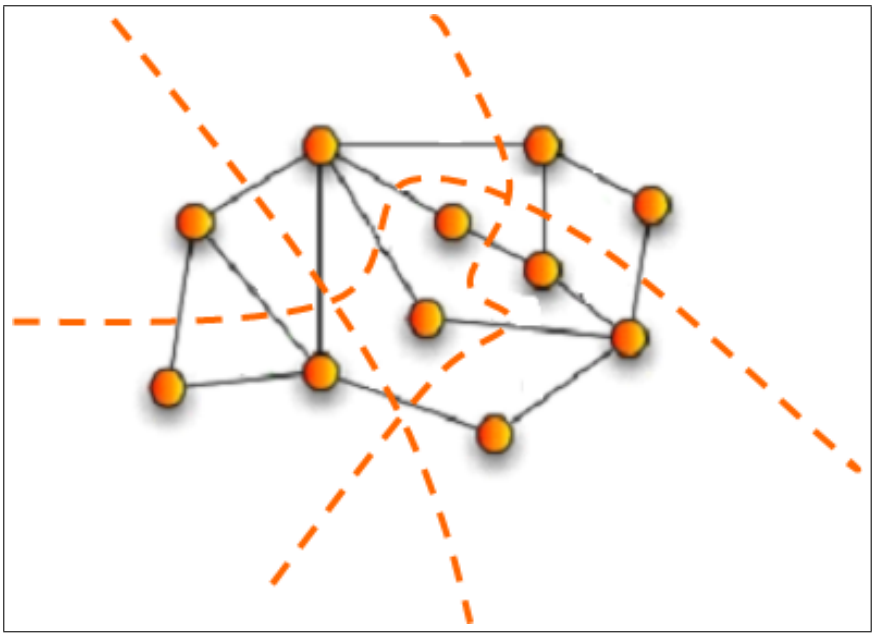

Figure 35: Example of max k-cut

We can formulate Max-Cut as the following integer quadratic programming problem:

$$\begin{cases} \max & f(x) = \frac{1}{2}\sum_{1\le i<j\le n} w_{ij}(1-x_ix_j) \\ s.t. & x_i \in \{1,-1\}, i \in V. \end{cases} \tag{98}$$

To check the above formulation (98) is correct, we define the subset $S \subseteq V$ such that $S = \{i \big| x_i = 1\}$. It can be easily seen that S induces a cut $(S, \bar{S})$ with corresponding weight (cut value) equal to:

$$W(S, V \setminus S) = \frac{1}{2} \sum_{1\le i<j\le n} w_{ij}(1-x_ix_j).$$

3 Computational Integer Programming: Algorithms for Discrete Optimization

Discrete optimization deals mainly with problems where we have to choose an optimal solution from a finite (or sometimes countable) number of possibilities. There are many such practical problems including the ones we have already discussed such as network-flow problems, cover problems (e.g. set covering problem), traveling salesman, facility location, knapsack problems, assignment problems, graph partitioning problems and scheduling. Discrete optimization problems are also referred to as combinatorial optimization problems. Most of these problems are computationally expensive with significant theoretical and practical interest.

This section presents some well known algorithms for solving discrete optimization problems. We will present both deterministic as well as meta-heuristics algorithms. We first present an efficient local search for 0-1 binary integer programming problem. The other deterministic algorithm will be presented is the Branch and Bound (BB) algorithm. This algorithm can handle general integer programming problem including 0-1 binary integer program. These deterministic algorithms require mathematical formulation of the discrete optimization problems, e.g. the integer programming formulation of the problem. We will then discuss three meta-heuristics techniques, namely Genetic Algorithm (GA), Tabu Search (TS) and Simulated Annealing (SA). These meta-heuristics can be directly applied to solve many discrete optimization problems at ease, e.g. they do not require explicit mathematical formulation of the problem.

3.1 An Efficient Local Search Method

In this section, we present a modified local search method for solving binary integer programming problems (BIP). This local search method will be referred to as the 'iterative improvement local search' (IILS). The IILS has its origin in the Fiduccia and Mattheyses (FM) algorithm [32] for hypergraph partitioning. Before presenting IILS we present the FM algorithm in the context of a BIP (such as the quadratic knapsack or the linear knapsack problem[8]).

The FM algorithm performs *passes* repeatedly until a *pass* fails to produce a better solution. Starting at x^0, a *pass* progresses in epochs. At each epoch unlocked components (at the first epoch all components, i.e. variables, are unlocked) of a solution (best solution found in the previous epoch, initially x^0) are flipped[9] and the best solution obtained by flipping is identified. The flipped component of the identified solution is then locked until the end of the current *pass*. This solution is then goes to the next epoch where there is one less component to flip. A *pass* ends when all variables are locked. Let x^k be the best solution at the end of the k-th epoch and $x = \arg\max_k\{L(x^k)\}$. The FM algorithm stops if $L(x) \leq L(x^0)$ (for the minimization problem the condition becomes $L(x) \geq L(x^0)$, where $x = \arg\min_k\{L(x^k)\}$), otherwise the next *pass* starts with $x^0 = x$.

[8] A function from equations (26)-(28) can be constructed as $L(x) = f(x) - R\sum_{i=1}^{m}\phi_i(x)$, where $f(x) = \sum_{j=1}^{n} c_j x_j$, $\phi_i(x) = \max\left(0, \sum_{j=1}^{n} a_{ij}x_j - b_i\right)$, and $R > 0$ is the penalty parameter. When $f(x)$ is a minimization problem write $L(x) = f(x) + R\sum_{i=1}^{m}\phi_i(x)$.

[9] A variable x_j is flipped means it is changed to $1 - x_j$.

It is imperative that we present a *pass* used in FM in a more explicit manner before presenting its modified version IILS. At the beginning of a pass, each variable is unlocked, i.e., variables are free to be flipped. Flipping of the j component of $x^0 = (x_1, x_2, \cdots, x_n)$ at the first epoch at a *pass* is carried out to obtain $(x_1, \cdots, 1-x_j, \cdots, x_n)$, $j = 1, 2, \cdots, n$. Hence, at the end of the first epoch the best solution say $(x_1, \cdots, 1 - x_t, \cdots, x_n)$ out of n solutions is found where the t-th component will remain fixed (locked) until the end of the current *pass*. At the next epoch only $n - 1$ components are flipped and the best out of $n - 1$ solutions is found to be taken to the following epoch where $n - 2$ components remain unlocked. The *pass* ends when all components are locked. When the j-th unlocked variable of x at an epoch is flipped a $gain(j, x)$ of a variable x_j is calculated[10] as:

$$gain(j, x) = L(\hat{x}) - L(x), \text{ for all unlocked } x_j, \tag{99}$$

where $\hat{x} = (x_1, \cdots, x_{j-1}, 1 - x_j, x_{j+1}, \cdots, x_n)$. Clearly, a gain can be position or negative. A positive gain means function value has improved (increased for maximization and decreased for minimization).

3.1.1 Iterative Improvement Local Search (IILS)

There are two weaknesses of the FM algorithm presented above. The first weakness is that the locked variable of the best solution of first epoch remains locked for the next $(n - 1)$ epochs. Similarly, the locked variable at the end of the second epoch will remain locked for the next $(n - 2)$ epochs and so on. This phenomenon is extremely prohibitive when the dimension, n, of the problem is high. Hence, IILS overcomes this by introducing a parameter τ. After a variable is locked at the end of an epoch, it is forbidden to be flipped in the next τ epochs of the pass only, after which this variable will be allowed to be flipped. That is every locked variable is made unlock after τ epochs in IILS. The second weakness is that the FM algorithm does not have a mechanism to check the number of negative gains in (99) produced by the best solution at the end of consecutive epochs. Indeed, in practice, at each *pass* of FM, only a small fraction of the epochs actually leads to increasing the objective function value (decreasing objective function value for minimization problem). In fact the objective function value tends to decrease (tends to increase for minimization) with the increase of the number of epochs. This is a negative phenomenon of FM. A measure therefore is needed to stop the *pass* i.e. not to perform the epoch when such a phenomenon is identified. A parameter called $passbreaker$ is introduced in IILS to identify such a phenomenon.

Specially, we check the consecutive number of negative gains produced by the best solution in epochs using $passbreaker$. Hence, if the maximum gain, using (99), at the end of a epoch is positive then $passbreaker$ is set to zero, as this would mean that the epoch has produced a better solution (this is recorded as the new x^0). Otherwise, consecutive negative gains are added together and stored in $passbreaker$. If the value of the $passbreaker$ at the end of an epoch is less than a preset negative value then the current *pass* is stopped and the next *pass* begins (or the IILS stops). The parameter $presetN$ is used to store the preset negative value. i.e. $presetN < 0$. A pass executes epochs until the sum of the negative gains is less than $presetN$. The steps for IILS for BIP is now presented below.

[10]For a minimization problem write: $gain(j, x) = L(x) - L(\hat{x})$, for all unlocked x_j.

The IILS Algorithm

---

1. REQUIRE: an initial solution $x^0 = (x_1^0, \cdots, x_n^0)$, the value of parameters $\tau > 0$, and $presetN < 0$
2. ENSURE: an improved solution x^{max}
3. $x^{max} = x^0$; $flag = 1$.
4. WHILE $flag = 1$
4.1 Let $flag = 0$, and $passbreaker = 0$, $x^0 = x^{max}$.
4.2 Let $free = \{1, 2, \cdots, n\}$; Set $d_j = 0$, for each $j \in \{1, \cdots, n\}$
4.3 Calculate gains $gain(j, x^0), j \in free$, according to (99).
4.4 WHILE $passbreaker \geq presetN$
4.4.1 Let $gain(t, x^0) = \max\{gain(j, x^0), j \in free\}$
4.4.2 Flip the variable x_t, i.e., let $x^0 = (x_1^0, \cdots, 1 - x_t^0, \cdots, x_n^0)$
4.4.3 Set $d_j = d_j - 1$, for each $j \in \{1, \cdots, n\}$, and $d_j > 0$. Set $d_t = \tau$
4.4.4 Let $free = \{j \in \{1, \cdots, n\} | d_j = 0\}$
4.4.5 $passbreaker = passbreaker + gain(t, x^0)$
4.4.6 IF $passbreaker > 0$
4.4.6.1 $x^{max} = x^0$, $flag = 1, passbreaker = 0$
4.4.7 ENDIF
4.5 ENDWHILE
5. ENDWHILE
6. RETURN x^{max}

Remarks on the IILS algorithm:

1. Let $free$ be the set of unlocked variables. We use $flag$ to record whether a better solution is found. More formally, if $flag = 1$ it means that a better solution is found at the end of an epoch in a *pass*.

2. A pass consists of lines 4∼5 while lines 4.4∼4.5 constitute an epoch in the IILS algorithm. Lines 4.4.6∼4.4.7 checks if the current epoch produces a positive gain i.e. improved function value.

3. Each pass starts from an initial solution x^0. At the first epoch of any pass, all variables are unlocked (line 4.2), and the initial gains are calculated by (99) (line 4.3). Then, the variable with the largest gain, denoted by x_j, is locked (line 4.4.2), and it is forbidden to flipped in the next τ iterations (line 4.4.3). The set $free$ is updated (line 4.4.4).

3.2 Branch and Bound Algorithm

Computationally, the most important aspect of solving integer programs is obtaining good bounds on the value of the optimal solution. This fact will be clear in the Branch and Bound algorithm.

The Branch and Bound (BB) method is referred to as intelligent enumeration [35] and used to solve any mixed integer linear problems and mixed integer quadratic program (which will be convex problem when relaxed). This means that BB can be used to solve integer linear programs and integer quadratic program (convex when relaxed). I would like to introduce BB using a minimization problem i.e. an integer program which has to be minimized (I will make appropriate references to maximization problem whenever needed). The concept of lower bounds and an upper bound (upper bounds and a lower bound for the maximization problem) are very important for BB. An initial best integer feasible solution is used as the initial upper bound (lower bound for maximization) of the problem solved. This feasible solution may be found by solving the problem with a heuristic method [37]. The upper bound (lower bound for maximization) is updated every time a better integer feasible solution is found.

The BB method starts by solving the relaxed version of given (original) problem. If the original problem is a linear programming (LP) problem (like the linear knapsack, TSP, p-median, set covering, graph coloring etc) then the problem will be solved optimally where all the integer constraints are ignored [36]. On the other hand if the problem is quadratic (such as the quadratic knapsack, quadratic assignment, graph partitioning etc), its relaxed version (as if all variables are real) will be solved using an appropriate solver. This relaxed solution will be a lower bound (upper bound for a problem which will be maximized) of the given problem.

Next, the BB method branches on the non-integer values found in the relaxation solution. The idea of branching is to create two subproblems per branching. For the 0-1 ILP (integer linear program) or IQP (integer quadratic program) this means fixing $x_j = 0$ (or $x_{ij} = 0$ say, for p-median problem or quadratic assignment problem) in the first subproblem and $x_j = 1$ in the second subproblem for a non-integer variable x_j (or x_{ij} for some problems). Indeed, for the branching purpose the variable with the most non-integer value is considered (e.g. the j-th variable above). Two subproblems are then solved[11] and branching continues until all variables return integer values (or no more branching possible).

A key part of the branch–and–bound method is to know when a subproblem is not worth investigating. There are two such cases:

(a) A branch cannot better the value of the relaxation solution it originally split from, called the branch's lower bound (upper bound for maximization). When it becomes clear that the lower bound (upper bound for maximization) of a particular branch is higher (lower for maximization) than the upper bound (lower bound for maximization), the branch (the subproblem) will not be investigated further.

(b) When the relaxed solution of any subproblem produced integer feasible solution then the upper bound is updated with this solution, and the subproblem is deleted (will not be investigated further).

[11] This means the original problem is again solved twice optimally with the restrictions (i) $x_j = 0$, and (ii) $x_j = 1$ respectively.

The process in (a) and (b) is known as bounding. There is another reason when a subproblem is deleted i.e. it is not investigated further is when the subproblem is infeasible.

The speed of the BB is therefore highly dependent on finding an effective heuristic that will produce a good initial upper bound (lower bound for max), and a solver that solves the relaxed (sub) problems and produces tighter lower bounds (upper bound for max). The good initial upper bound (lower bound for max) and the tighter lower bounds (upper bound for maximization) for subproblems are important to eliminate as many branches as possible. We will examine branch and bound in more detail. Here I summarize the BB method before presenting the details:

The basic idea behind BB is to use the continuous relaxation of the original problem to obtain valid lower bounds (upper bound for maximization) and explore the space of the integer variables using a tree search. Specifically, this method proceeds by solving the original relaxed problem. If the solution of the relaxed problem is integer feasible for the original problem then the optimal solution has been found and the algorithm is stopped. Otherwise the solution of the continuous relaxation provides a lower bound on the solution of original problem and the algorithm selects and branches (one of the subproblems) on one of the integer variables (see more details later on). One of the new problems is then selected, its continuous relaxation is solved and the branching process is repeated. The algorithm stops once all of the leaf nodes (subproblems) of the search tree have been fathomed and returns the (lowest) upper bound as the solution.

3.2.1 Fundamental Steps of BB

Branch and bound is the most commonly-used algorithm for solving MILPs (mixed integer linear programs) or MIQPs (mixed integer quadratic program). Naturally, then it is the most used algorithm for solving IPs (integer programs) i.e. integer linear and integer quadratic programs. BB is a divide and conquer approach. For the ease of explanation I will use a integer linear program[12] e.g. the set covering problem (66)-(68). Suppose S is the feasible set for an ILP (or IQP which is convex in its relaxed form) and we wish to solve $\min c^T x,\ x \in S$ ($\max c^T x,\ x \in S$ if we consider linear knapsack). In this case the feasible set is given by

$$S = \{(x_1, x_2, \cdots, x_n) \Big| \sum_{j=1}^{n} a_{ij} x_j \leq b_i,\ \forall i,\ x_j \in \{0, 1\}\}$$

Considers the following partitions of S into the subsets $S_1, S_2, \cdots S_k$ (not necessarily disjoint), then its follows that

$$\min_{x \in S} c^T x = \min_{\{1 \leq i \leq k\}} \left\{ \min_{x \in S_i} c^T x \right\}. \tag{100}$$

(For the maximization problem replace the minimization in the above with maximization.) We see in (100) that we can optimize over each subset separately. The basic idea is as follows. If we cannot solve the original problem directly, we might be able to solve the smaller subproblems recursively. Dividing the original problem into subproblems is called branching. Taken to the extreme, this scheme is equivalent to complete enumeration.

[12] We can also consider the maximization problem such as the linear knapsack problem or the quadratic knapsack problem which are maximization problems.

However, we have seen the 'bounding' concept before. Indeed, the bounding enables us to avoid enumerating all possible subsets (i.e to avoid solving all subproblems). Let us now go in little deeper of 'branching' and 'bounding' of BB.

Any (integer) feasible solution to a given (the original) integer programming problem provides an upper bound $u(S)$ (a lower bound $l(S)$ for the maximization problem) on the optimal solution value of the problem. We can use heuristic methods to obtain an (overall) upper bound $u(S)$ (a overall lower bound $l(S)$ for the maximization problem). After branching, try to obtain a lower bound $l(S_i)$ on the optimal solution value for each of the subproblems (an upper bound $u(S_i)$ on the optimal solution value for each of the subproblems, for the maximization problem). If $l(S_i) > u(S)$ then we delete the i-th subproblem altogether from further consideration of branching ($u(S_i) < l(S)$, then we do not need to consider subproblem i, for maximization problem). One easy way to obtain a lower bound is by solving the LP relaxation by dropping the integrability constraints (an upper bound is found by solving the LP relaxation of the maximization problem).

Notice that the integer linear program I have considered is a binary integer program relaxation of which is a linear program (LP). What follows next is the detailed presentation of a linear programming based BB (LP-based BB). Notice also that variables in ILP not necessarily only take binary values; they can assume/take other integer values. Hence, we can assume all variables have finite upper and lower bounds.

3.2.2 LP-based Branch and Bound

- In LP-based branch and bound, we first solve the LP relaxation of the original problem. The result is one of the following:

  1. The LP is infeasible then this implies ILP is infeasible.
  2. We obtain a (integer) feasible solution for the ILP then this means the optimal solution of ILP has been found.
  3. We obtain an optimal solution to the LP that is not (integer) feasible for the ILP then this means we have a lower bound for ILP (upper bound for maximization).

- In the first two cases, we are done.

- In the third case, we must branch and recursively solve the resulting subproblems.

Branching in LP-based Branch and Bound: A systematic method of choosing a branching is called a branching rule. To branch, we identify the relaxed LP solution $\hat{x}$ and look at a $\hat{x}_i$ in Step 3 which are (integer) infeasible, i.e. real values. We then select the i-th variable whose value $\hat{x}_i$ is the most fractional in the LP solution (not close to the integer value). We the create two subproblems such that:

(a) In one subproblem, impose the constraint $x_i \leq \lfloor \hat{x}_i \rfloor$

(b) In the other subproblem, impose the constraint $x_i \geq \lceil \hat{x}_i \rceil$

Remark 3.1. *What does (a) and (b) mean in a 0-1 integer program?*

Remark 3.2. *When one uses the BB algorithm to solve a QIP (quadratic integer program) the steps are very similar to what I have described for the LP-based BB. The only difference is that the LP solver has to be replaced by an appropriate solver for relaxed QIP (the relaxed is a convex problem).*

3.2.3 Steps of the BB Algorithm

We will now described how one implements the BB algorithm i.e. its algorithmic steps. Let $P(S)$ be the given (original) minimization problem with its feasible set S. At each iteration of the BB algorithm either we create two subproblems from a chosen subproblem or delete the chosen subproblem. Hence, at the beginning of each iteration k there will be a number of subproblems from which to chose one. Let $L(k) = \{P(S_1), P(S_2), \cdots, P(S_{p(k)})\}$ be the set of subproblems at the beginning of iteration k. Hence $L(0) = \{P(S)\}$ where $P(S_{p(0)}) = P(S)$ when k=0. The relaxed solution of the j-th subproblem $P(S_j)$ is a lower bound given by $l(S_j)$. At the beginning of of each k-th iteration we have an overall upper bound $u(S)$ corresponding to a feasible integer solution, called the incumbent solution, $\bar{x}$. The BB algorithms stops when $L(k)$ is empty.

There are two more issues remaining: which subproblem from $L(k)$ to consider for branching[13], and which variable will be the branching variable.

Selection of branching nodes: It has been found that the selection method for branching nodes significantly affect the performance of *Branch-and-Bound* as does the selection method for the branching variables. A popular choice is the recency based strategy. In this recency-based branching strategy, whenever a branching is carried out, the nodes corresponding to the new problems are given preference over the rest of the unfathomed nodes. The node that is the newest in the list of unfathomed sub-problems is selected for branching. This strategy is also known as *depth-first strategy*. It has the advantage of saving storage space.

Choice of the branching variable: Let $\hat{x} = (x_1, \ldots, x_n)^T$ be the optimal solution of the continuous relaxation of the sub-problem $P(S_j)$ at the node j. A popular approach is to consider a variable $\hat{x}_i$ that is most fractional in $\hat{x}$. The k-th iteration begins with selecting a sub-problem. Once a sub-problem, say the j-th sub-problem, has been selected, the following steps are carried out:

- Solution $l(S_j)$ of the continuous relaxation of the sub-problem $P(S_j)$ is obtained. Being the relaxed solution $l(S_j)$ is the lower bound of $P(S_j)$. If $l(S_j) > u(S)$ (overall upper bound corresponding to the incumbent solution $\bar{x}$) then $P(S_j)$ is fathomed from $L(k)$ and a new sub-problem is selected.

- If the solution is an integer solution, then the incumbent $\bar{x}$ and $u(S)$ are updated. The process in this case is ended without creating any new sub-problem from the selected sub-problem, i.e. $P(S_j)$ is deleted from $L(k)$. A new sub-problem is selected from L again.

- If the solution of the continuous relaxation of $P(S_j)$ is not integer feasible, then two new sub-problems are created from $P(S_j)$ by searching a variable, say the i-th variable $\hat{x}_i$, to branch upon. The feasibility of the new sub-problems $P(S_j^1)$

[13]Either the chosen subproblem in $L(k)$ will be replaced by two new subproblems or it will be deleted from $L(k)$ altogether.

and $P(S_j^2)$ with the corresponding feasible sets[14] $S_j^1 = S_j \cap \{x_i = 0\}$ and $S_j^2 = S_j \cap \{x_i = 1\}$ are then checked. A sub-problem is fathomed if it is not feasible. This type of fathoming occurs after a certain number of iteration. The unfathomed sub-problem is now added to L.

When BB stops the overall upper bound (which will be lowest upper bound after the BB process) $\bar{x}$ will be taken as the minimizer x^*.

[14]When the variables are not binary but rather can assume general integer values, $S_j^1 = S_j \cap \{x_i \leq \lfloor \hat{x}_i \rfloor\}$ and $S_j^2 = S_j \cap \{x_i \geq \lceil \hat{x}_i \rceil\}$

3.3 The Simulated Annealing Algorithm

Kirkpatrick et al. [30] designed the simulated annealing algorithm for optimization problems by simulating the physical annealing process. The formulation of the optimization algorithm using the above analogy consists of a series of Metropolis chains used at different values of decreasing temperatures.

The general SA consists of two loops. In the inner loop, a number of points in a Markov chain (a Markov chain is a sequence of trial solutions) in the configuration space is produced and some of them are accepted. A trial solution is accepted only if it satisfies the Metropolis criterion. On the other hand, in the outer loop, the temperature is progressively decreased. The whole process depends on the cooling schedule which will be discussed shortly. The Metropolis acceptance rule is given by:

$$A_{xy}(T_t) = \min\{\, 1,\, \exp(-(f(y) - f(x)\,)/T_t\,)\,\}, \tag{101}$$

where T_t is the temperature at the t-th Markov chain (counter of the outer loop). SA has two loops: the outer loop controls the temperature at each t-th Markov chain (MC) and the inner loop generates L_t number of trial points at each MC. We present below the algorithm for SA.

The SA algorithm.

begin
initialize T_t, $t = 0$, and the starting point x
while stop criterion = false do
begin
for i := 1 **to** L_t **do**
begin
generate y from x;
if $f(y) - f(x) \leq 0$ **then** accept;
else if $\exp(-(f(y) - f(x)\,)/T_t\,)$ >random$(0, 1)$ **then** accept;
if accept **then** $x := y$;
end;
set $t := t + 1$ and lower T_t;
end;
end.

Remark 3.3. *In the SA algorithm a function $f(x)$ is minimized, where x is a solution e.g. a route in the TSP and $f(x)$ is distance traversed due to x. It can be seen that SA can be directly applied so long as a trial solution (a new route) y can be generated from a give solution (the route x). Hence no mathematical formulation of the TSP is required in order to solved using SA.*

3.3.1 Cooling Schedule for SA

The choice of a cooling scheduling has an important bearing on the performance of the SA algorithm. We implement the cooling schedule suggested by Dekkers and Aarts [31], except the stopping condition.

The initial temperature, T_0: T_0 is obtained by generating a sample of size m_0, and requiring that the initial acceptance ratio χ_0 is close to 1, where χ_0 is the ratio of the

number of accepted transitions and the number of proposed transitions. The initial T_0 is given by

$$T_0 = \overline{\Delta f^+} \left(\ln \frac{m_2}{m_2 \chi_0 - m_1(1-\chi_0)} \right)^{-1}, \tag{102}$$

where m_1 and m_2 denote the number of trials ($m_1 + m_2 = m_0$) with $\Delta f_{xy} \leq 0$ and $\Delta f_{xy} > 0$, respectively. $\overline{\Delta f^+}$ is the average value of those Δf_{xy}-values, for which $\Delta f_{xy} > 0$ ($f(y) - f(x) = \Delta f_{xy}$).

The length of Markov chain: Dekkers and Aarts [31] suggested an approach which generates a fixed number of points, i.e.,

$$L_t = 10n. \tag{103}$$

The decrement rule for T_t: T_t is decreased at the end of each MC. Dekkers and Aarts [31] suggested the following scheme:

$$T_{t+1} = T_t \left(1 + \frac{T_t \ln(1+\delta)}{3\sigma(T_t)} \right)^{-1}, \tag{104}$$

where $\sigma(T_t)$ denotes the standard deviation of the values of the objective function at the points in the MC at T_t, and δ is called the distance parameter.

Stopping condition: The stopping condition used here is different from [31]. We make the following simplistic choice to terminate the algorithm i.e. we stop SA when

$$T_t \leq \varepsilon, \tag{105}$$

where ε is chosen judiciously.

Remark 3.4. *For the case of TSP, $f(x)$ is the tour length for the tour x. Often educated guess can be used for the initial temperature rather than calculated by the given formula. Similarly, temperature can be decreased using $T_{t+1} = \alpha T_t$ where $\alpha \in (0.9, 0.97)$.*

3.4 Genetic Algorithm

The GA technique was introduced by Holland [33] in 1975 and since then has been applied to a wide range of global optimization problems. Its popularity is due both to its suitability for solving such problems and its ease of implementation. GA methodology is based on analogies to the current theory of biological evolution and hereditary. It mimics natural selection strategies from evolution, embracing the 'survival of the fittest' principle.

The idea behind genetic algorithm (GA) is to do what nature does. Let us take rabbits as an example : at any given time there is a population of rabbits. Some of them are faster and smarter than the other rabbits. These faster and smarter rabbits are less likely to be eaten by foxes, therefore more of them survive to do what rabbits do best : make more rabbits. Of course, some of slower and dumber will survive just because they are lucky. The breeding results in a good mixture of rabbit genetic material : some slow rabbits breed with fast rabbits, some fast with fast, some smart rabbits with dumb rabbits, and so on. The resulting baby rabbits will (on average) be faster and smarter than those in the original population because more faster, smarter parents survived the foxes.

A genetic algorithm follows a step by step procedure that closely matches the story of the rabbits. Genetic algorithm use a vocabulary borrowed from nature genetics. We would talk about individual (genotype) in a population; quite often these individuals are called also strings or chromosomes. Chromosome are made of units–genes (also features or characters)– arranged in linear succession; every gene controls the inheritance of one or several characters. Gene of certain characters are located in certain places of the chromosome, which are called loci (string position). Each genotype (a string or a chromosome) would represent a potential solution to a problem; an evolutionary process run on a population of chromosomes (say, a population consists of N chromosomes) corresponds to a search through a space of potential solutions. Such a search requires balancing two (apparently conflicting) objectives : exploiting the best solutions and exploring the search space.

GA has attracted a lot of interest in recent years. GAs have been quite successfully applied to optimization problems like wire routing, scheduling, adaptive control, game playing, cognitive modeling, transportation problems, traveling salesman problems, optimal control problems, in business, in telecommunication and other problem involving optimization in general [34]. Many variants of GAs have been developed for the purposes of accommodating different applications.

I will present two versions of GA. The first version can handle problems with binary variables. I will use applications of GA on problems with explicit mathematical formulations[15]. The second version of GA is suitable for direct implementation without mathematical sophistication (mathematical formulation of the problem). The standard GA method which constitutes the basis of these variants can be described as follows:

1. An initial population of parental individuals is randomly created. Each individual is represented by a chromosome, a string of characteristic genes.

2. All the individuals are evaluated and ranked with a fitness function appropriate to the problem in hand. The most fit of these passes directly to the following generation; a process referred to as 'elitism'.

[15]There are however many problems where binary values does not come from mathematical formulation but rather from inherent problem structure, where GA can be applied suitably.

3. A breeding population is formed by selecting top-ranking individuals from those that remain. This is the natural selection step.

4. These selected individuals undergo certain transformation *via* genetic operators to reproduce children for the next generation. Operators include recombination by crossover (two chromosomes mate by partly exchanging genes to form two new chromosomes) and mutation (one randomly chosen gene, or more, of a chromosome is altered). Mutation ensures a level of genetic diversity in the population.

The process (from 2 to 4) in repeated until a certain number of generations (iterations) or some termination criterion is reached. As optimization continues, subsequent generations will consist of increasingly fit or 'superior' individuals (chromosomes). Relatively 'strong' individuals survive and reproduce, while relatively 'week' individuals die.

The algorithm starts with a first generation of strings (initial population). The population consists of N different chromosomes generated randomly, with the i-th chromosome say $x^i = (x_1, x_2, \cdots, x_n) = (1, 0, \cdots, 1, 0)$. Consider the $L(x)$ in subsection 3.1. The function can be evaluated for the chromosome x^i. $L(x^i)$ is known as fitness value of x^i. Since there are N different chromosome so there will be N fitness values: some good (with respect to the maximization of $L(x)$), some poor while other average fitnesses. Let

$$G(t) = \{x^{(1,t)}, x^{(2,t)}, \cdots, x^{(N,t)}\}$$

be the set of chromosomes (population) at iteration t. A good string has high fitness (function value). We can think of at the initial iteration of GA, we have N solutions x^i with corresponding function values $L(x^{(i,t)}), i = 1, 2, \cdots N$. The very basic of GA is to transform these N solutions at an iteration t of GA to the next iteration $t + 1$ using some operations such that on average fitnesses of the entire population improves. Hence, as the iteration increases average function values of $L(x^{(i,t)}), i = 1, 2, \cdots N$ improves too.

3.4.1 Elitism

The most intuitive operation of GA is the elitism whereby an small number of superior chromosomes (say 10% best $x^{(i,t)}$ out of the total N, call them good children in $G(t+1)$) are directly taken to $G(t + 1)$. The remaining members of population (90% of the N chromosomes) are first transformed and then taken to $G(t + 1)$. GA operations such as selection, crossover and mutation used for the transformation:

3.4.2 Selection, Crossover and Mutation

We have copied 10% best chromosomes (best solutions) from $G(t)$ to $G(t + 1)$ we have still N solutions with their function values (fitnesses) $L(x^{(i,t)}), i = 1, 2, \cdots N$, in $G(t)$. The remaining 90% children in $G(t + 1)$ are created pairwise. For each pair of children two good parents, say A and B, are selected (by the selection operation of GA). I will now explain how the first two children, say $x^{(1,t+1)}$ and $x^{(2,t+1)}$, are created in $G(t + 1)$. The remaining are created pairwise similarly. There are many ways to select two good parents A and B, but I will use what is known as the *tournament selection*. This is as follows. Select two parents, say $x^{(i,t)}$ and $x^{(j,t)}$, $i \neq j$, randomly from $G(t)$, and then assign A=best $\{x^{(i,t)}, x^{(j,t)}\}$. Similarly, select another two parents, say $x^{(p,t)}$ and $x^{(q,t)}$,

$p \neq q$. Repetitions are allowed here i.e. $x^{(p,t)}$ and/or $x^{(q,t)}$ may have been chosen earlier but we are okay as long as $p \neq q$. As before assign B=best $\{x^{(p,t)}, x^{(q,t)}\}$. This is known as the *tournament selection*. The two children (offsprings), say C and D, are created using either a single point (needing one cross point) crossover or a double point (needing two cross point) crossover, as shown in Figs 36 and 37.

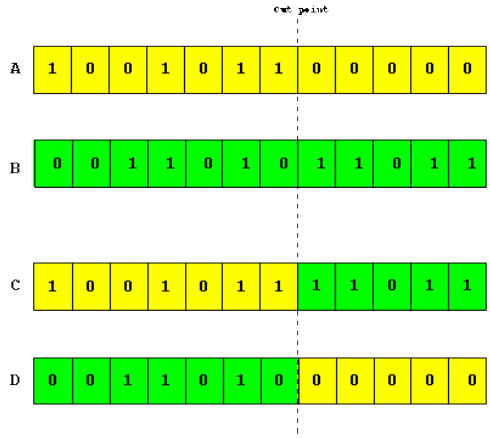

Figure 36: Single point crossover

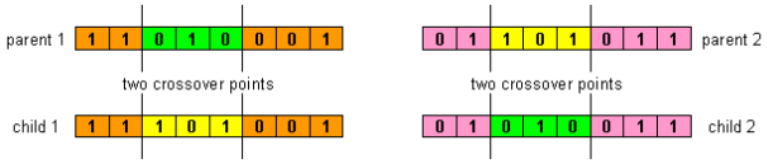

Figure 37: Two point crossover

After the two children are created using one of the crossovers (single or double) the offsprings need to go through what is known as the *mutation* operation. I have shown this in the Fig. 38 using one of the parents created by crossover. Mutation must be applied to both C and D with small probability p_μ. This probability is applied to each (binary) value in the chromosome. The mutation process is shown in Fig. 38.

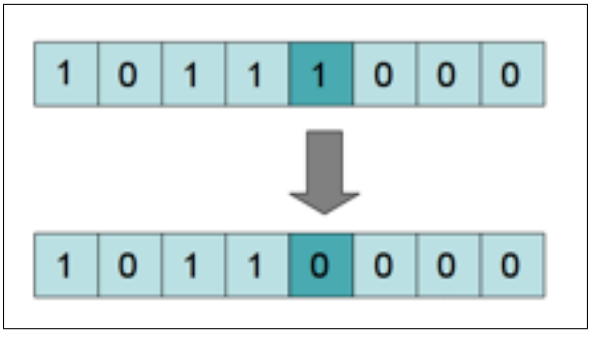

Figure 38: Mutation applied to one of the parent obtained by crossover

After crossover and mutation two offspring, say $x^{(1,t+1)}$ and $x^{(2,t+1)}$ in $G(t+1)$. The

above process is repeated $\frac{m}{2}$ times to create m new solutions ($m = 0.9N$). These $0.9N$ solutions and $0.1N$ solutions used in the elitism constitute N solution in $G(t+1)$.

What I have described can be summarized as follows. A good number of parents from $G(t)$ goes to $G(t+1)$ under elitism. A pair of strings from $G(t)$ is selected for crossover. Those string with higher fitness (lower function values if we minimize; higher function values if we maximize) have a better change of being selected and taking part in the crossover process. Two new offprings are created after crossover and mutation. Crossover followed by mutation. During mutation process strings also have a small probability of being mutated. Mutation arbitrarily alters one or more genes of a selected chromosome, by a random chance with a probability equal to the mutation rate p_μ.

The above process is repeated until m offsprings are generated. The resulting offsprings form part of the new generation, $G(t+1)$, and this process repeats until the deviation within the population is less than a certain tolerance. A typical genetic algorithm is as follows:

Step 1. Generate an initial population $G(0)$ of size N.

Step 2. Generate offspring.

- Selection : selecting $m \leq N$ strings as parents, with probability of each string being selected proportional to its fitness.
- Crossover : the parents are paired and generate m offspring, which replaces the m least fittest strings.
- Mutation : each string has a small probability p_μ of being mutated.

Step 3. Repeat Step 2 until the function values of the points ($L(x^{(i,t)}, i = 1, 2, \cdots, N)$ are closed to each other, that is to say the standard deviation is very small.

Remark 3.5. *There are many variants of the above representation of GA. For example, in the crossover phase of step 2, the m offspring may replace their parents, rather than m least fittest strings.*

3.4.3 The Direct Version of GA

We have seen above how GA can be implemented in 0-1 binary integer problem, where the population consists of a number of solutions (chromosomes of stings of 0s and 1s). Now I present a version of GA that can be applied directly[16] (without any mathematical formulation) on problem such as TSP and QAP. In this version of GA mutation is not applied (or applied with p_μ=0). A crossover, called the swap crossover, is used. The structure of this version is very similar to the previous one. We will use elitism to copy a small fraction of good solutions (say 5%) from t-th iteration to the $(t+1)$-th iteration. The remaining solutions (m) of the $(t+1)$-th iteration are generated using the swap crossover (aka the swap path crossover). To generate an offspring swap crossover generates two parents (say solutions $P1$ and $P2$) at random from $G(t)$. Hence the swap crossover is repeated m times.

[16]SA and the Tabu Search can also be applied directly to solve discrete or combinatorial optimization problems.

In the swap path crossover scheme, we start at some position of the chromosomes, which is determined randomly, and examine the chromosomes corresponding to $A1$ and $A2$ from left to right in a cyclic fashion. If the alleles at the position (or gene) being looked at are the same, we move to the next position; otherwise, we perform a swap of the alleles of two genes in $A1$ or in $A2$, whichever gives the fitter solution, so that the alleles at the position being looked at become alike. We repeat this process until all alleles have been considered. All chromosomes obtained using this process are valid children of $A1$ and $A2$; out of them we let A denote the fittest child. We illustrate the swap path crossover below using the QAP (or you can think of as the TSP):

$$P1 : 5 - 2 - \mathbf{3} - \mathbf{4} - 1 - 7 - 6 \qquad \text{swap in P1:} \qquad 2 - 5 - \mathbf{3} - \mathbf{4} - 1 - 7 - 6$$
$$P2 : 2 - 1 - \mathbf{3} - \mathbf{4} - 6 - 5 - 7 \qquad \text{swap in P2:} \qquad 5 - 1 - \mathbf{3} - \mathbf{4} - 6 - 2 - 7$$

Suppose we start at the first position. In $A1$, facility 5 (city 5 in the case of TSP) is located at the first site and in $A2$ facility 2 (city 2 in the case of TSP) is located at this site. There are two ways in which the two parents can move closer to one-another; by swapping the sites of the facilities 2 and 5 in parent 1 ($A1$) or in parent 2 ($A2$). We then consider the alleles in the two resulting solutions starting at the second position and obtain possibly several children; the best among these is A. In this example, we start at the first position; in general, we start at a position selected randomly and examine subsequent positions from left to right in a wrap-around fashion until all positions have been considered. The steps of the direct version of GA are as follows:

Step 1. Generate an initial population $G(t)$, $t = 0$, of size N.

Step 2. Copy $N - m$ best solutions from $G(t)$ to $G(t + 1)$. Update t=$t + 1$.

Step 3. Generate offspring. For $i = 1, 2, \cdots, m$ do

- Selecting two parents, parent 1 and parent 2, at random (i.e. every parent has equal probability).
- Swap Crossover : Perform swap crossover using the two parents and generate a number of offspring. Take the best amongst the two parents and the offsprings just created as the i-th child (offspring) of $G(t)$.

Step 4. Repeat Steps 2 & 3 until a maximum number of iterations.

3.5 Tabu Search Heuristics

Tabu search [27] is a meta-heuristic optimization algorithm that has been widely used in many practical problems of discrete nature.

We consider the minimization problem such as the TSP or the QAP (quadratic assignment problem). TS can be directly applied to these types of problems (instead of their binary counter-parts). It is an iterative method and at each iteration it performs a sequence of moves that lead from one trial solution to another. The tabu search algorithm starts at some initial solution and then move to a neighboring solution. A neighboring solution is generated by a set of admissible moves (i.e the moves that are allowed). At each iteration TS takes the best solution in the neighborhood of the current solution [29]. Tabu search allows the search to explore solutions that do not decrease the objective function value only in those cases where these solutions are not forbidden. This is usually obtained by keeping track of the last solutions in terms of the action used to transform one solution to the next. When an action is performed it is considered tabu for a number of iterations [29]. Since TS has been designed to solve combinatorial optimization problems, it can be explained better with reference to a problem e.g. the symmetric TSP. We begin with basic structure of TS together with the operations used in it before its full description and the algorithm.

TS may be useful to find a good, or possibly optimal solution of problems which are of the type:

$$\text{minimize } f(x), \text{ such that } x \in X, \tag{106}$$

where $f(x)$ is any function of a discrete variable x ($f(x)$ returns the distance traveled corresponding to the route x), and X is the set of feasible solutions. Hence for the STSP $f(x)$ can be treated as the distance involved in the tour x (e.g. the objective function) and X is the set of all such x that satisfy all the constraints (e.g. a city is not visited more than once; no subtour formation). TS generates a new solution $\bar{x}$ from the existing solution x, within the neighborhood of x, by the operation called a 'move'.

We denote a move at x by $m(x)$ (hereafter m) and the set of moves at x by $M(x)$. Hence, $m(x) = \bar{x}$ and $m \in M(x)$. The neighborhood of x is defined by

$$N(x) = \{\bar{x} \big| \bar{x} = m(x),\, x, \bar{x} \in X; m \in M(x)\} \tag{107}$$

To prevent the process from cycling in a small set of solutions, some attribute ℓ (associated with m and x) of recently visited solutions is stored in a Tabu List, which prevents their occurrence for a limited period. This attribute is stored in a set TB_ℓ called tabu list (a short term memory). For the STSP, the attribute may be a pair of nodes (cities) that have been exchanged (exchanged positions in the sequence in which they were visited) recently. The elements of TB_ℓ define all tabu moves that cannot be applied to the current solution. The number of TB_ℓ is bounded by a parameter s, called taboo list size. If $|TB_\ell| = s$, before adding ℓ to TB_ℓ, one must remove oldest element from TB_ℓ.

By the above flexible memory structures TS narrows down the examination of the neighborhood $N(x)$. Memory structures result in the neighborhood $N(x)$ being modified to $N^*(x)$ where $N^*(x) \subseteq N(x)$, due to tabu list (or forbidden list). Elements of $N(x)$ that are excluded from $N^*(x)$ are classified as tabu, whilst those within $N^*(x)$ make up candidate solutions [27, 28]. At times, due to defined 'aspiration criteria', a solution within the tabu list may be chosen allowing the TS to achieve better results, relying on the

supposition that a bad strategic choice can yield more information than a good random choice. We further elaborate the above concepts below.

Tabu search starts from initial solution and attempts to determine a better solution in manner like greedy method, the basic idea of the method is to explore the search space of all feasible solutions by a sequence of moves. A move is a change which transforms the current solution into one of its neighboring solutions. Associated with each move is a move value, which represents the change in the objective function value as a result of the move. At every iteration of the algorithm an admissible best move is applied to the current solution to obtain a new solution to be used in the next iteration. A move is applied even if it is a non-improving one that it does not lead to a solution better than the current solution. To escape from local optimal and to prevent cycling, a subset of moves is classified as tabu (or forbidden) for a number of iterations the classification depends on the history of the search.

The tabu tenure (the duration for which a move will be kept tabu) is an important feature of tabu search because it determines how restrictive the neighborhood search is (for the TSP this short term structure is described by Fig. 39). The tabu restrictions are not inviolable under all circumstances and a tabu move can overridden under some conditions. A condition that allows such an override is called an aspiration criterion. The aspiration criterion might allow overriding a tabu move if the move leads to a solution, which is the best obtained so far.

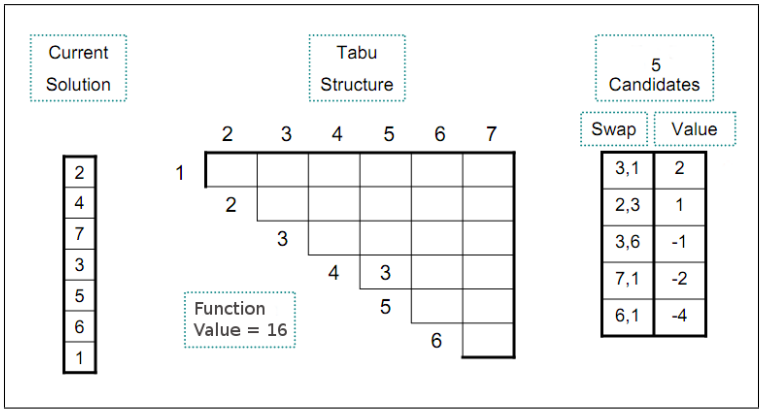

Figure 39: The short term memory structure in TS for TSP

We now summarize *initial starting solution*, *neighborhood structure*, *tabu list*, and *aspiration criterion*. Whenever needed we will use STSP for explanation.

Initial Solution: A good feasible, yet not-optimal, solution to the TSP can be found quickly using a greedy approach. This is as follows. Starting with the first node (city) in the tour, find the nearest node. Each time find the nearest unvisited node from the current node until all the nodes are visited.

Swapping: It is a move mechanism in TSP. The swap move mechanism replaces a selected distance between two cities by another distance. Or the swapping operation neighborhood considers each possible exchange of two (cities) in the problem.

Neighborhood: We will use STSP. A neighborhood to a given solution is defined as any other solution that is obtained by a pair wise exchange of any two nodes in the so-

lution. This always guarantees that any neighborhood to a feasible solution is always a feasible solution (i.e, does not form any sub-tour). If we fix node 1 as the start and the end node, for a problem of n nodes (cities), there are ${}^{n-1}C_2$ such neighborhoods to a given solution. Let us assume $x = (1, 2, 3, 4, 5)$ is the current solution, and so one can construct $N(x)$ which has 6 solutions obtained by pairwise exchange, i.e $N(x) = \{(1, 3, 2, 4, 5), (1, 4, 3, 2, 5), (1, 5, 3, 4, 2), (1, 2, 4, 3, 5), (1, 2, 5, 4, 3), (1, 2, 3, 5, 4)\}$. Hence for every $x \in X$ there is a set of neighbors call the neighborhood $N(x)$ of the solution x. The neighborhood $N(x)$ consists of solutions obtained by making small changes in X. At each iteration the best neighboring solution is chosen as shown in the following figure:

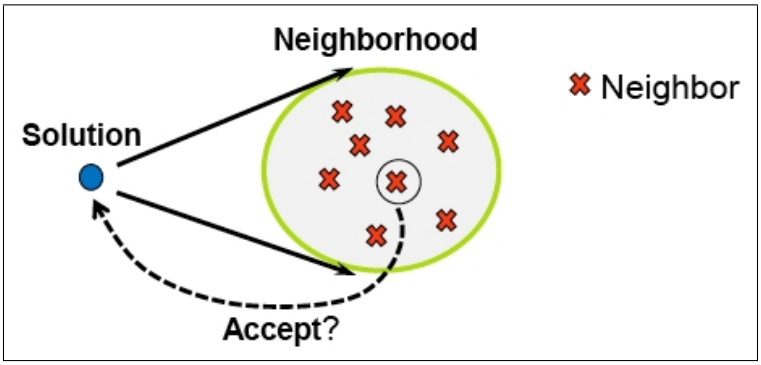

Figure 40: The best solution within a neighborhood

There are many possible ways of selecting the best neighbor, including: first, choosing the first neighboring solution that improves objective function. Second, considering a subset of the neighbors and selecting the best of the set. Third, evaluating the whole set of neighboring solutions and selecting the best in terms of the objective functions. The second method is quite popular. For the case of TSP, the neighborhood with the best objective value (minimum distance) is selected, per iteration.

Tabu List TB_ℓ: To prevent the process from cycling in a small set of solutions, some attribute of recently visited solutions is stored in a Tabu List, which prevents their occurrence for a limited period. For the TSP, the attribute used is a pair of nodes that have been exchanged recently. A Tabu structure stores the number of iterations ($|TB_\ell| = s$) for which a given pair of nodes is prohibited from exchange.

Aspiration criterion: Tabus may sometimes be too powerful: they may prohibit attractive moves, even when there is no danger of cycling, or they may lead to an overall stagnation of the searching process. It may, therefore, become necessary to revoke tabus at times. In order to override the tabu list when there is a good tabu move, aspiration criterion is used, the tabu move is accepted if it produces better solution than the best obtained so far.

Termination criterion The most commonly used stopping criteria in tabu search are: (a) Terminate the search after number of iterations; (b) After some number of iterations without an improvement in the objective function value.

An implementation of TS is characterized by:

- The set $M(x)$ of moves applicable to a feasible solution x (neighborhood).

- The type of the elements of the set TB_ℓ which define the tabu moves (tabu list).
- The size s of the set TB_ℓ (tabu list size).
- A stopping condition.

We now present the flowchart in Fig. 41. The step by step description of TS technique is as follows (where k denote the iteration number):

Step 1 Start with any feasible solution x^k, $k = 0$, an empty tabu list TB_ℓ. Let $x^* = x^k$, $f^* = f(x^k)$. (x^* is the best solution found up to now and f^* the value of the objective function for this solution.)

Step 2 In $M(x^k)$ choose m, a move transforming x^k that minimizes $f(m(x^k))$ and that is not forbidden by the elements of TB_ℓ. The move can be chosen by complete or partial examination of $M(x^k)$. Let $x^{k+1} = m(x^k)$.

Step 3 If $f(x^{k+1}) < f^*$, let $f^* = f(x^{k+1})$ and $x^* = x^{k+1}$.

Step 4 If $|TB_\ell| = s$ remove the oldest element of TB_ℓ; add the element ℓ defined by m and x^{k+1}. Set $k = k + 1$.

Step 5 Go back to Step 2 if the stopping condition (optimum reached, k larger than a fixed number) is not satisfied.

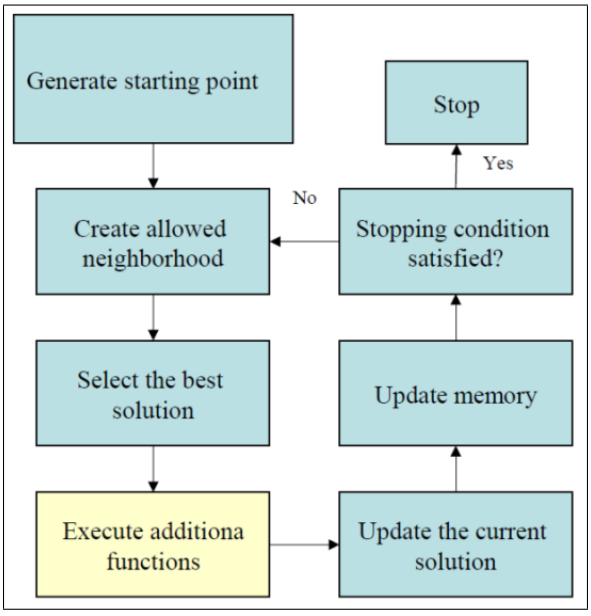

Figure 41: The flowchart of TS

References

[1] C. H. Papadimitriou and K. Steiglitz (1998), Combinatorial Optimization: Algorithms and Complexity. Dover Publications Inc., New York, 2nd edition.

[2] Ravindra K. Ahuja, Thomas L. Magnanti, and James B. Orlin (1993). Network Flows: Theory, Algorithms, y, g , and Applications, Englewood Cliffs, N.J., Prentice-Hall

[3] L. R. Ford and D. R. Fulkerson (1956). Maximal flow through a network, Canadian Journal of Mathematics, Vol. 8, pp.399.

[4] G. L. Nemhauser and L. A. Wolsey. (188), Integer and Combinatorial Optimization. John Wiley and Sons, New York.

[5] D. Pisinger D.(2007), The quadratic knapsack problem–a survey. Discrete Applied Mathematics 155 (5) 623-648.

[6] Shmoys, David B. and Rinnooy Kan, Alexander H. G. and Lenstra, Jan K. and Lawler, Eugene L. (1987), The Traveling Salesman Problem: A Guided Tour of Combinatorial Optimization, 1st Edition, J. Wiley and sons, Chichester, New York, Brisbane.

[7] Zhang Weixiong (1997), A note on the complexity of the Assymetric Traveling Salesman Problem, Operation Research Letters, 20, pp.31-38.

[8] T. S. Arthanari and M. Usha (2000), An alternate formulation of the symmetric travelling salesman problem and its properties, Discrete Applied Mathematics, 98, pp.173–190.

[9] C. H. Papadimitriou and K. Steiglitz (1998), Combinatorial Optimization: Algorithms and Complexity. Dover Publications Inc., New York, 2nd edition.

[10] Ravindra K. Ahuja, Thomas L. Magnanti, and James B. Orlin (1993). Network Flows: Theory, Algorithms, y, g , and Applications, Englewood Cliffs, N.J., Prentice-Hall

[11] J. Abrache, T. G. Crainic and M. Gendreau (2005), Models for bundle trading in financial markets. European Journal of Operational Research 160 (1) 88-105.

[12] V. Avasarala, T. Mullen, D. L. and A. Garga (2005), MASM: a market architecture or sensor management in distributed sensor networks. In: SPIE Defense and Security Symposium, Orlando FL, pp. 5813-5830.

[13] V. Avasarala, H. Pllavarapu, T Mullen (2006), An approximate algorithm for resource allocation using combinatorial auctions. In: international Conference on Intelligent Agent Technology, pp. 571-578.

[14] M. Ball, G. Donohue, and K. Hoffman (2006), Auctions for the safe, efficient and equitable allocation of National Airspace Systems esources. In R. Steinberg, P. Cramton, Y. Shoham (Eds.), Combinatorial auctions. Cambridge: MIT, Chap. 22.

[15] W. E. Walsh, M. Wellman and F. Ygge (2000), Combinatorial auctions for supply chain formation. In: ACM Conf. on Electronic Commerce, pp 260-269.

[16] T. Mullen, V. Avasarala and D. L. Hall (2006), Customer-driven sensor management. IEEE Intelligent Systems 21 (2) 41-49.

[17] T. C. Koopmans and M. J. Beckmann (1957), Assignment problems and the location of economic activities, Electronica 25, 53-76.

[18] E. L. Lawler (1963), The quadratic assignment problem, Management Science 9, no. 4, 586-599.

[19] C. S. Edwards (1880), A branch and bound algorithm for the koopmans beckmann quadratic assignment problem, Mathematical Programming Study 13, 35-52.

[20] Ceselli, A., Liberatore, F., and Righini, G. (2009). A computational evaluation of a general branch-and-price framework for the capacitated network location problems. Annals of Operations Research, 167:209-251.

[21] C. H. Papadimitriou and M. Yannakakis (1991), Optimization, approximation and complexity classes, Journal of Computer and System Sciences 43, 425-440.

[22] C. De Simone, M. Diehl, M. J$\ddot{u}$nger, P. Mutzel, G. Reinelt and G. Rinaldi (1995), Exact ground states of Ising spin glasses: new experimental results with a branch-and-cut algorithm, Journal of Statistical Physics 80, 487-496.

[23] F. M. Xu, C. X. Xu, H. G. Xue (2006), A multiple penalty function method for solving max-bisection problems, Applied Mathematics and Computation, 173 (2): 757-766.

[24] A. F. Ling, C. X. Xu, L. Tang (2008), A modified VNS metaheuristic for maxbisection problems, Journal of Computational and Applied Mathematics, 220 (1-2): 413-421.

[25] W Zhu, G Lin, MM Ali (2013), Max-k-cut by the discrete dynamic convexized method, INFORMS Journal on Computing 25 (1), 27-40.

[26] C. H. Papadimitriou and K. Steiglitz (1998), Combinatorial Optimization: Algorithms and Complexity. Dover Publications Inc., New York, 2nd edition.

[27] Glover F. and Laguna M.,'Tabu Search', Kluwer Academic Publishers, 1997.

[28] Glover F., 'Tabu Search: A Tutorial', Interfaces, 20(4):74-94,1990.

[29] Glover, F., Future Paths for Integer Programming and links to Artificial Intelligence, Computers and Operations Research, 13, pp.533-549, 1986.

[30] Kirkpatrick, S., Gelatt, C.D. and Vecci, M.P. Optimization by simulated annealing. *Science 220*, (1983), 671–680.

[31] Dekkers, A. and Aarts, E. Global optimization and simulated annealing. *Mathematical Programming 50*, (1991), 367–393.

[32] Fiduccia C. M., Mattheyses R. M., 1982. A linear time heuristic for improving network partitions. In Proc. ACM/IEEE DAC, pp. 175-181.

[33] J. H. Holland, *Adaptation in Natural and Artificial Syayems*, University of Michigan Press, Ann Arbor, MI., 1975.

[34] D. E. Goldberg, *Genetic Algorithms : in search, Optimization & Machine Learning*, Addition-Wesley Publishing Company, Inc., 1989.

[35] Christos H. Papadimitriou and Kenneth Steiglitz, *Combinatorial Optimization: Algorithms and Complexity*, Dover Publications Inc., 2nd Edition, New York, 1998.

[36] Wayne L. Winston, *Operations Research: Applications and Algorithms*, Brooks / Cole - Thomson Learning, 2004.

[37] M. J. Brusco and S. Stahl, *Branch–and–Bound Applications in Combinatorial Data Analysis* Springer Science + Business Media, Inc., 1st Edition, 2005.