DISSERTATION

TWO TOPICS IN COMBINATORIAL OPTIMIZATION: THE DOMINO PORTRAIT PROBLEM AND THE MAXIMUM CLIQUE PROBLEM

Submitted by

Bader Alshamary

Department of Mathematics

In partial fulfillment of the requirements

for the degree of Doctor of Philosophy

Colorado State University

Fort Collins, Colorado

Spring 2007

COLORADO STATE UNIVERSITY

March 26, 2007

WE HEREBY RECOMMEND THAT THE DISSERTATION PREPARED UNDER OUR SUPERVISION BY BADER ALSHAMARY ENTITLED "TWO TOPICS IN COMBINATORIAL OPTIMIZATION: THE DOMINO PORTRAIT PROBLEM AND THE MAXIMUM CLIQUE PROBLEM " BE ACCEPTED AS FULFILLING IN PART REQUIREMENTS FOR THE DEGREE OF DOCTOR OF PHILOSOPHY.

Committee on Graduate Work

Dr. Ross Beveridge

Dr. Michael Kirby

Dr. Christopher Peterson

Adviser: Dr. Anton Betten

Department Head: Dr. Simon Tavener

ABSTRACT OF DISSERTATION

TWO TOPICS IN COMBINATORIAL OPTIMIZATION: THE DOMINO PORTRAIT PROBLEM AND THE MAXIMUM CLIQUE PROBLEM

Combinatorial Optimization plays a significant role in applied mathematics, supplying solutions to many scientific problems in a variety of fields, including computer science and computational networks. This dissertation first reviews a number of problems from combinatorial optimization and the algorithms used to solve them.

The author then presents original solutions to the domino portrait problem, which involves arranging complete sets of dominos to resemble photographic portraits when seen from a distance. The first approach makes use of a greedy algorithm. Because the greedy algorithm often encounters blockages, a new technique was developed to avoid these blockages. Next, a local search algorithm was used to solve the problem. In both new solutions, the cost function was modified so that important positions in the portrait such as facial features were emphasized, thus improving the results. A singular value decomposition (SVD) was used to construct a "support matrix" necessary for this new cost function. Algorithms used in computing the SVD include the Householder method and the QR method.

The second problem dealt with is the maximum clique problem and its application of finding ovoids in finite polar spaces. Again, local search

provides an efficient way to search for maximum cliques in graphs and hence for finding ovoids in finite polar spaces.

Bader Alshamary
Department of Mathematics
Colorado State University
Fort Collins, Colorado 80523
Spring 2007

ACKNOWLEDGEMENTS

I would like to express my deepest gratitude to Dr. Anton Betten who agreed to supervise my work and who encouraged and guided me patiently to the finish. I wish to thank every member in my committee, Dr. Ross Beveridge, Dr. Michael Kirby, and Dr. Chris Peterson, for all the patience, advice and care they extended to me, which made this work possible. Finally, I would like to thank every one how provide me help to this work.

TABLE OF CONTENTS

Chapter 1

INTRODUCTION

Combinatorial optimization is an important topic in applied mathematics which plays an important role in many scientific fields, among them computer science and computational networks. In this dissertation, numerous combinatorial optimization problems will be presented, followed by two original applications of combinatorial optimization to two specific problems, the domino portrait problem and the maximum clique problem.

Chapter 2 presents background information about various combinatorial optimization problems and the algorithms used to solve them. Section 2.1 explains the following problems: the traveling salesman problem, the minimum spanning tree, the single source shortest path problem, the all pairs shortest path problem and maximum network flow. Section 2.2 presents a number of greedy algorithms which have been used to solve the problems discussed in Section 2.1. These algorithms include Kruskal's algorithm, used for the traveling salesman problem and the minimal spanning tree and Prim's algorithm, used for the minimal spanning tree. Dijkstra's algorithm is used for solving the single source shortest path problem, and Floyd's Warshall algorithm helps to solve the all-pair shortest path problem. Finally, Ford-Fulkerson's algorithm is presented as a means for solving the maximal network flow problem.

Local search algorithms are another group of algorithms which are significant in solving combinatorial optimization problems. These algorithms are defined in Section 2.3 and the Lin-Kernighan algorithm for a symmetric traveling salesman problem is presented. Important issues related to local search algorithms are discussed as well.

Section 2.4, the last section of Chapter 2, presents linear programming problems including the simplex method.

Chapter 3 presents the combinatorial optimization problem called the domino portrait problem (DPP). A domino portrait consists of complete sets of dominos arranged in a matrix to create an approximate replication of an image when seen from a distance. This problem was first solved by Robert Bosch using the integer linear programming method with the software program CPLEX. In this dissertation, we employ two significant combinatorial optimization algorithms known as greedy algorithms and local search algorithms to solve it. Section 3.2 presents the steps of creating a domino portrait. The domino structure is described in section 3.4, followed by an explanation of the integral linear programming formulation of the problem including the parameters, decision variables, objective function, constraints and the standard form of the linear programming problem.

Section 3.6 discusses the use of the greedy algorithm for solving the DPP. It includes an explanation of the search strategy of the algorithm, and shows how the problem must be remodeled to fit this algorithm. An analysis of why the algorithm is sometimes blocked in solving the DPP is presented along with suggestions as to how to avoid this blockage.

In Section 3.7, the application of the local search algorithm to the DPP is presented, including neighborhoods of the DPP and ways to construct the set of feasible solutions F for the DPP.

Section 3.8 deals with ways to improve the cost function in the DPP and presents a new cost function. Section 3.9 discusses the singular value decomposition method (SVD) used to construct the support matrix necessary for this new cost function. Algorithms used in computing the SVD are given, including the Householder's method and the QR method.

Chapter 4 discusses the maximum clique problem (and hence the independent set problem). Section 4.2 relates the maximum clique problem to the problem of finding ovoids in finite polar spaces, which are maximum cliques in a certain graph associated to the polar space.

briefly describes the applications of the maximum clique problem including an important application called finding ovoids in projective space.

Section 4.3.1 presents a new technique based on the local search algorithm for finding cliques of a given size in a given graph. The neighborhoods of the maximum clique problem are explained and the pseudo code for the algorithm are given.

Section 4.4, the last section of Chapter 4, shows computational results of the algorithm for finding ovoids in small polar spaces.

Finally, in chapter 5, we discuss our conclusion.

Chapter 2

BACKGROUND

Combinatorial optimization plays an important rule in various sciences. Some of these areas are applied mathematics, computer science and computational networks. In addition, it arises in management science, like finance marketing and data base management. Combinatorial optimization is also used in engineering sciences, for example, optimal designs of waterways or bridges and analysis of data networks. Additional applications can be found in Karla Hoffman [26].

Combinatorial Optimization can be considered as operations that search for one or more good solutions for optimization problems. This can be done by studying and analyzing mathematically the problem and composing a set of possible solutions called the *feasible solutions set.* The goal of combinatorial optimization is to find the best (optimal) feasible solution in this set.

In the following sections we present combinatorial optimization problems and their applications. Moreover, we provide important related definitions and theorems.

2.1 Combinatorial Optimization Problems

Combinatorial optimization problems are optimization problems that have discrete variables. That is, the domain of possible solutions that satisfy the constraints of the problem is finite, countable infinite or can be reduced

to a countable infinite set. The following definitions are stated in order to understand combinatorial optimization problems.

Definition 2.1. An **instance** of an optimization problem is a pair (F, c), where F is the domain of feasible points, and c is a real value function $c : F \rightarrow \mathbb{R}$, that sometimes refers to capacity, distance, or cost.

The goal of combinatorial optimization problems is to find an $y \in F$ such that

$$c(y) \leq c(g) \quad \forall \quad g \in F.$$

In this case, y is called an ***optimal solution*** to the given instance.

We need to distinguish between an optimization problem A and an instance x of a problem. An instance x is a special case of an optimization problem A. That is, a problem is a general case. The following is the definition of an optimization problem.

Definition 2.2. *An optimization problem A is a quadruple (I, F_x, c_x, t), where*

- *I is a set of instances of the optimization problem A.*
- *F_x is the set of feasible solutions associated with a given instance $x \in I$.*
- *$c_x(y)$ is the value of the feasible solution $y \in F_x$ for some instance $x \in I$. $c_x(y) \geq 0$ in many examples and is called the cost function.*
- *t is the type of problem. It can be either a maximization problem $(t = max)$ or a minimization problem $(t = min)$.*

Therefore, the goal of the optimization problem is to find the optimal $(min\ or\ max)$ feasible solution y of an instance x of an optimization problem A. That is,

$$\text{to find } y \in F_x \text{ s.t. } \begin{cases} c_x(y) \leq c_x(\acute{y}) \quad \forall\ \acute{y} \in F_x, & \text{if } t = min; \\ c_x(y) \geq c_x(\acute{y}) \quad \forall\ \acute{y} \in F_x, & \text{if } t = max, \end{cases}$$

In the following sections, we will present several important combinatorial optimization problems.

2.1.1 Traveling salesman problem (TSP)

The traveling salesman problem is the following. A sales person wants to visit his clients in n different cities and wants to optimize his travel time by taking into account the various different travel times between the cities. That is, he wants to visit each city exactly once and return to his hometown at the end. He is looking for the fastest route possible. More precisely, we are given n cities $a_1, \ldots, a_n$ and the various costs $c_{i,j}$ of traveling from city a_i to city a_j. The problem is to find the cheapest tour that visits all cities exactly once and returns to the starting point. In general, the assumption $c_{i,j} = c_{j,i}$ is made. That is, the cost of traveling from city i to city j is the same cost as traveling from j to i. The tour is described by a permutation $(i_1, \ldots, i_n)$ of the integers $1, \ldots, n$. That is, we start in city a_{i_1}, then move to city a_{i_2}, then to city a_{i_3} and so on. After visiting city a_{i_n} we return to city a_{i_1}. Therefore, the set of feasible solutions is

$$F_x = Sym_n,$$

where Sym_n denotes the group of permutations of n objects. Moreover, for $y = (i_1, \ldots, i_n) \in F_x = Sym_n$, the cost of traveling tour y is

$$c_x(y) = \sum_{t=1}^{n-1} c_{i_t, i_{t+1}} + c_{i_n, i_1}. \tag{2.1}$$

The problem is a minimization problem in that we want to find the cheapest tour $y \in F_x$. That is, we want to find $y \in F_x$ such that

$$c_x(y) \leq c_x(\acute{y}) \text{ for all } \acute{y} \in F_x$$

The following example presents an instance of the traveling salesman problem.

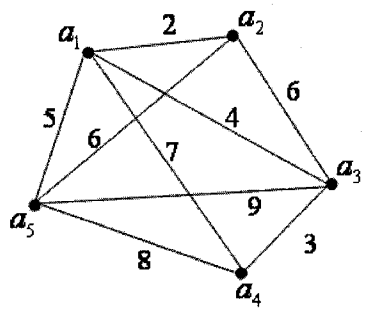


Figure 2.1: A five cities instance of TSP

Example 2.1.

Figure 2.1 shows an instance of TSP of five cities, a_1, a_2, a_3, a_4 and a_5. The cost $c_{i,j}$ of traveling between city a_i and city a_j is indicated along the edge in the graph between a_i and a_j. If a_i and a_j are not connected, the cost is $+\infty$. For example, the cost of traveling between city a_1 and city a_3 is $c_{1,3} = 4$ and the cost of traveling between city a_2 and city a_3 is $c_{2,4} = +\infty$. The following table shows all possible feasible tours of this instance.

Tour	$(i_1 i_2 i_3 i_4 i_5)$	$c_x(y_i)$
y_1	(12345)	24
y_2	(12354)	∞
y_3	(12435)	∞
y_4	(12453)	∞
y_5	(12534)	27
y_6	(12543)	23
y_7	(13245)	∞
y_8	(13254)	31
y_9	(13425)	∞
y_{10}	(13524)	∞
y_{11}	(14235)	∞
y_{12}	(14325)	27

A table of the feasible tours

However, some of these tours are better than the others. For instance, the

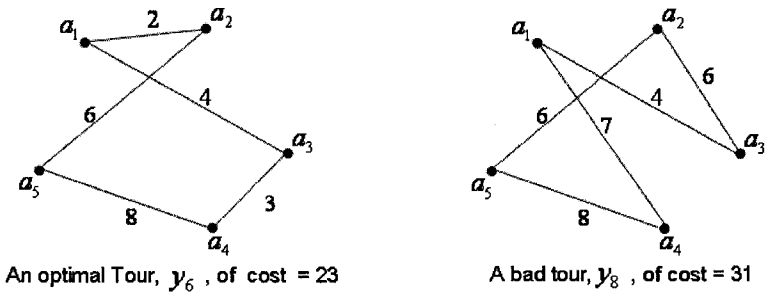


Figure 2.2: Two tours of the instance of the TSP of Example 2.1. Left: An optimal tour, y_6, with cost = 23. Right: A bad tour, y_6, with cost = 31.

tour $y_6 = (12543)$ is an optimal tour whereas tour $y_8 = (13254)$ is a bad tour, (Figure 2.2). This is because the cost of tour y_6 is

$$c_x(y_6) = c_{1,2} + c_{2,5} + c_{5,4} + c_{4,3} + c_{3,1} = 23$$

which is the smallest cost where $c_x(y)$ is defined by equation 2.1. While the cost of the tour y_8 is

$$c_x(y_8) = c_{1,3} + c_{3,2} + c_{2,5} + c_{5,4} + c_{4,1} = 31,$$

which is a large cost. In addition, some of these tours have a cost of $+\infty$ because there is at least two cities in this tour where the cost of traveling between them are $+\infty$. For example, tour $y_2 = (12354)$ has cost $c_x(y_2) = +\infty$, since the cost of traveling between city a_2 and city a_3 is $c_{2,3} = +\infty$. Moreover, tour $y_1 = (12345)$ is a good solution since it has a cost $c_x(y_1) = 24$ which is close to the optimal tour which is y_6, which has a cost $c_x(y_6) = 23$.

In fact, these are not the only feasible tours of this example. This is because the total number of all possible permutation of n objects is equal to $n!$. Therefore, the total number of all possible feasible tours of this example is $5! = 120$. However, we excluded several feasible tours from the total number of possible permutations. A tour is excluded if it satisfies one of the following conditions:

1. The tour has the same route as a tour that already has been taken but starts from a different city. This is because we do not care about which city we start in. For example, the following tours are equal

$$y_1 = (12345), \qquad y_{1,2} = (23451),$$
$$y_{1,3} = (34512), \qquad y_{1,4} = (45123), \quad \textit{and}$$
$$y_{1,5} = (51234).$$

2. The tour has the same route as a tour that already has been taken but has a different direction. This condition is included if the cost of traveling from city a_i to city a_j is the same as the cost of traveling from city a_j to city a_i. That is, if

$$c_{i,j} = c_{j,i}$$

In this case we do not care which direction we go in a given tour. For example, a tour $y_1' = (15432)$ is equal to the tour $y_1 = (12345)$.

From this discussion we conclude that the total number of feasible solutions of an instance of TSP with n cities is given by the following equations

$$|F_x| = \frac{n!}{n} \qquad \textit{if the second condition, } c_{i,j} = c_{j,i}, \textit{ is satisfied.} \tag{2.2}$$

$$|F_x| = \frac{n!}{2.n} \qquad \textit{otherwise.} \tag{2.3}$$

Therefore, the total number of feasible solutions of a five city instance of TSP is

$$|F_x| = \frac{5!}{2.5} = 12.$$

Moreover, if we have a six city instance of TSP, the total number of feasible solutions is

$$|F_x| = \frac{6!}{2.6} = 60.$$

Hence, the total number of feasible solutions increases if the number of cities increases.

One way to represent the TSP and many other combinatorial optimization problems is with graph theory. To illustrate this more clearly, we shall give some basic definitions in graph theory.

Definition 2.3.

1. *A* ***graph*** *is a pair* (V, E) *where* V *is a set whose elements are called vertices(nodes) and* E *is a collection of two subsets of* V *called edges(links).*

2. *An* ***edge(link)*** $e \in E$ *is a link between two vertices (nodes) in* V*.*

3. *A* ***walk*** *from vertex* a *to vertex* b *is a sequence of one or more edges* $e_1, e_2, \ldots, e_k$ *such that* $e_1 = \{a, a_2\}, e_2 = \{a_2, a_3\}, \ldots, e_k = \{a_k, b\}$ *where* $\{a, a_2, \ldots, a_k, b\} \in V$*. This walk is denoted as* $a, a_1, a_2, \ldots, a_k, b$ *or* $e_1, e_2, \ldots, e_k$*.*

4. *A* ***path*** *from vertex* a *to vertex* b *is a walk from vertex* a *to vertex* b *where the vertices are all distinct. The* ***length*** *of a path is the number of edges in this path.*

5. *A* ***weighted graph*** *is a graph* $G = (V, E)$*, together with a function*

$$w : E \to \mathbb{Z}^+.$$

If $e = \{a_i, a_j\}$*, then* $w(e) = w_{ij}$ *is called the* ***weight*** *of the edge* e*. The* $w(e)$ *represents a cost, capacity or distance between vertex* a_i *and vertex* a_j*. Moreover, the* ***weight of a path*** *is the sum of the weights of all edges in this path.*

For example, in the graph in Figure 2.3, the sequence of edges e_1, e_2, e_3, e_4, e_5 and e_6 forms a walk from vertex a to vertex b. However, they do not form a path from vertex a to vertex b since the vertex a_2 is used twice. On the other hand, the sequence of edges e_1 and e_6 form a path from vertex a to vertex b. Moreover, the length of this path is 2 and its weight is 18.

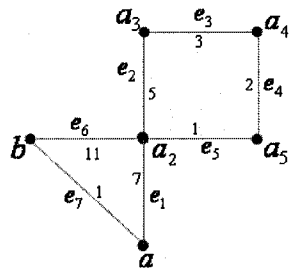


Figure 2.3: A connected weighted graph of six vertices

Now, we can describe the traveling salesman problem by a weighted graph $G = (V, E)$ where the cities are the vertices V and the cost function w_{ij} represents the cost of traveling between cities i and j (here, we assume that the cost of travel does not depend on whether we go from city i to city j or vice-versa). Therefore, the goal is to find a path that has minimum weight and starts and ends at the same vertex. Moreover, each vertex must be visited exactly once.

Usually the global optimal solution of an optimization problem is hard to find. However, sometimes it is possible to find a solution that is close to the optimal one. In the next chapters we present some significant algorithms that are capable of finding the optimal solution or near to the optimal solution to the traveling salesman problem and other combinatorial optimization problems. For example, we will use Kruskal's algorithm [13], which is an example of a greedy algorithm, and a local search algorithm to solve the TSP problem.

2.1.2 Minimal Spanning Tree

Finding the minimal spanning tree is one of the most important combinatorial optimization problems. To illustrate the problem we state the following definitions in graph theory:

Definition 2.4.

1. A graph $G = (V, E)$ is *connected* if there is a path from any vertex $a \in V$ to any vertex $b \in V$.

2. A *circuit* is a path that begins and ends at the same vertex. A circuit is *simple* if it does not contain two similar edges.

3. A *tree* is a connected undirected graph that does not contains any simple circuits. The weight of a tree is the sum of the weight of its edges.

4. Let G be a graph. A tree is called *spanning tree* of G if contains all vertices of G.

Example 2.2. *Graph G in Figure 2.3 is a connected weighted graph since there is a path between any two vertices in G. In addition, the path $e_1, e_2, e_3, e_4, e_5, e_6$ does not form a circuit because it does not start and end with the same vertex. Whereas, the path e_2, e_1, e_7, e_6, e_2 is a circuit, it is not simple since it contains e_2 twice. The edges e_1, e_6 and e_5 form a tree; however, they do not form a spanning tree because they do not contain every vertex of the graph. The weight of this tree is 19.*

Now we can define the minimal spanning tree by the following definition.

5. Let G be a connected weighted graph. A *minimal spanning tree* of the graph G is a spanning tree that has the smallest weight. That is, a minimal spanning tree is a connected weighted subgraph of the graph G with no simple circuits , containing all vertices of the graph G, and having a minimum weight.

Usually there are various spanning trees in a connected weighted graph. The problem in the minimal spanning tree is to determines which of these spanning trees, which contains all vertices of the graph, has minimal weight.

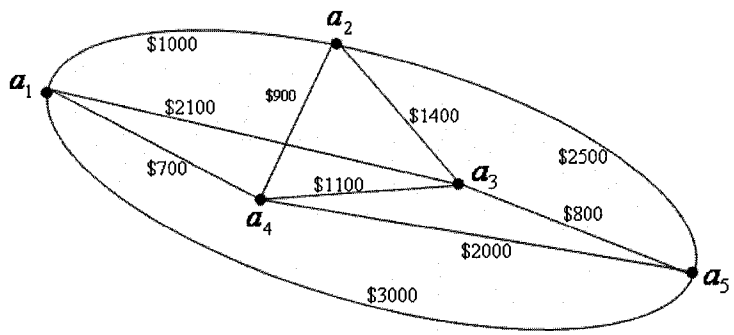


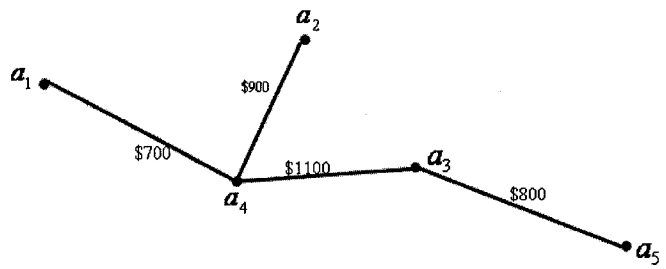
Figure 2.4: A connected weighted graph representation of a phone network design with five cities

Many important problems can be modeled as minimal spanning tree problems. A phone network design is an example of a minimal spanning tree problem that is described in the following example.

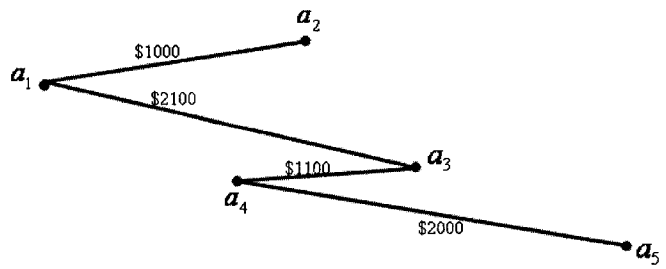
Example 2.3. *(A phone network design)*

Assume that you are the owner of a business company consisting of five offices located in five different cities. You want to build a phone network to connect them with each other. A phone company can link each pair of these offices at various costs. The problem is we want to find which links connect all the offices so that the total cost of these links is minimum.
This problem can be solved by modeling it as a connected weighted graph and finding a spanning tree that has a minimum weight. The connected weighted graph representative of this problem is shown in Figure 2.4. The vertices represent the cities where the offices are located. The edges represent the links of each pair, and the weights of these edges represent the cost of the link represented by the edges. The goal here is to find a spanning tree that has minimum weight. That is, we want to find a tree that contains all vertices , that has no simple circuits, and in which the sum of the weights of its edges has minimum value. This problem is considered as a minimal spanning tree problem since we can cancel some links and reduce the cost.

There are several spanning trees of this problem. However, some of them are better than others. For example, an optimal spanning tree is



A: An optimal spanning tree for the phone network design of cost $3500



B: A bad spanning tree for the phone network design of cost $6200

Figure 2.5: **An Optimal and a bad spanning tree of a phone network design**

shown in Figure 2.5-A, and it has a cost of $3900; whereas, a bad spanning tree is shown in Figure 2.5-B and its cost equals $6200.

Several algorithms can solve this problem, and we will present two of the most important ones. These algorithms are called Prim's algorithm and Kruskal's algorithms [43]. These algorithms are examples of greedy algorithms that we will explain in the next chapter.

2.1.3 Single-Source Shortest Path Problem

One of the major questions associated with a connected weighted graph is finding the path between two vertices that has the smallest weight. This problem is called the *shortest path problem*, and it can be solved by Dijkstra's algorithm. This algorithm is an example of a greedy algorithm and will be discussed in the next chapter.

There are many applications that can be modeled as shortest path problems. For example, airline routing systems, computer networks , and subway systems are all shortest path problems. For instance, an airline routing system can be represented in a connected weighted graph where the vertices represent cities and the edges on the graph represent flights between these cities. The weight assigned each edge represents the distance, fare, or flight times between these cities. Figure 2.6 - A, displays the problem involving the distance between cities; Figure 2.6 - B, displays the problem involving the fare between cities; And Figure 2.6- C, displays the problem involving the flight times between cities.

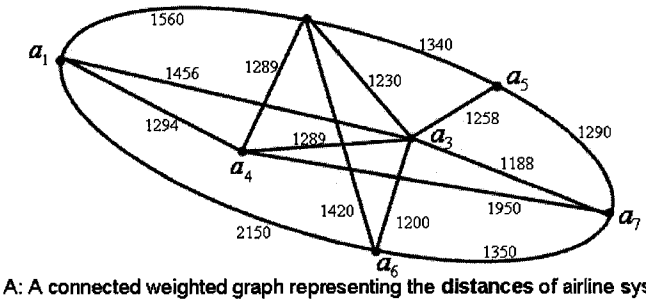
The following questions arise here:

1. What is the shortest distance that can be found to reach one city from another city?
2. What is the lowest fare which can be found to travel between two cities?
3. What is the shortest time needed to travel between two cities?

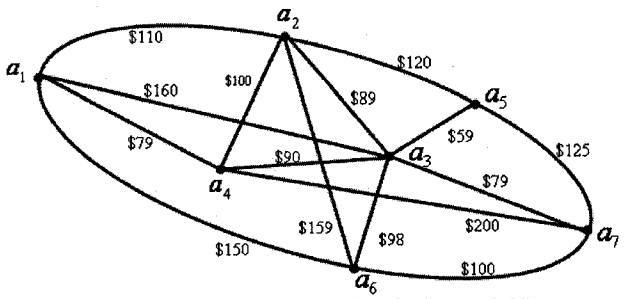
These questions can be solved by finding the shortest path in its corresponding graph. The shortest path means that a path that has minimal weight. The weight of a path can be determined by the sum of the weight assigned to each edge in this path. For example, the question involving the shortest times needed to travel between two cities can be solved by finding the shortest path between these cities.

Several algorithms can be used to solve the shortest path problem. One of these algorithms is called Dijkstra's algorithm, which is one of greedy algorithms that will be discussed in the next chapter.

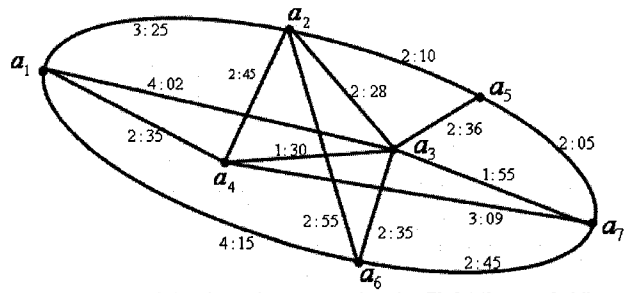
2.1.4 All Pairs Shortest Path (APSP) Problem



A: A connected weighted graph representing the **distances** of airline system



B: A connected weighted graph representing the **fares** of airline system



C: A connected weighted graph representing the **flight times** of airline system

Figure 2.6: **A connected weighted graph representation of distances, flight times, and fares of an airline system**

The all-pairs shortest path problem is a generalization of the single-source shortest path problem where the single-source shortest path problem searches for the lowest weight of a path between two vertices in a weighted directed graph, while the all-pair shortest path problem searches for the shortest paths between all pairs of vertices of the graph. In the other words, given a weighted directed graph $G = (V, E)$, the goal of the APSP is to compute the shortest path from each vertex $a \in V$ to every other vertex $b \in V$. The solution of APSP can be considered as a table or a matrix $D(d_{ij})$ where the entry d_{ij} is the shortest path between vertex i and j for all $i, j \in V$.

The all-pairs shortest path problem has many applications in communication, electronic and transportation problems [16]. For example, in a road atlas, it is important to determine the distance between all pairs of cities [13]. That is, we need to construct a table that shows the distance between all pairs of cities in the road atlas. This can be solved by modeling it in to a weighted directed graph and solving the APSP problem of this graph.

Many algorithms can solve the all-pairs shortest path problem. Floyd's Warshall algorithm, which is an example of a greedy algorithm, can solve the APSP problem. This algorithm can compute the weight of the shortest path between all pairs of vertices in a weighted directed graph [43]. Moreover, the APSP also can be solved using Dijkstra's algorithm by applying this algorithm to all pairs of vertices of a weighted directed graph $G(V, E)$ with non-negative weights, while Floyd's algorithm works for both negative and non-negative weights. These algorithms will be explained in the next chapter.

2.1.5 Maximum Network Flow

A *flow network* is a weighted directed graph that displays a system of rules of moving liquid, electronics, or material from a node s called *source*

where the material is supplied to a demand node t called the *sink*. These materials flow from the source s to the sink t through *intermediate* nodes, which are any node in the graph other than s and t. A material that flows between two nodes is restricted by an amount called capacity. That is, the amount of flow of material between two nodes cannot exceed this capacity. Moreover, for any node on this graph, except the source s and the sink t, the amount of material that flows into a node should be the same amount that leaves from this node. The source s has only outgoing flow and the sink t has incoming flow, on the other hand.

The maximum network flow is a problem that searches for a way of moving a maximum amount of material from a source s to a sink t without breaking the capacity restrictions, and computes the value of this flow.
The definition of a flow network is as follows:

Definition 2.5. A *flow network* is a weighted directed graph $G(V, E)$ where V is a set of nodes containing a source node s, a sink node t and additional intermediate nodes. For any intermediate node, there is a path from the source s to the sink t that passes through this node. Moreover, for any edge $(a_i, a_j) \in E$ is assigned a value called a capacity $c(a_i, a_j) \geq 0$ such that the material passing through this edge cannot exceed its capacity. We give a zero capacity for any edge that is not in E. That is, if an edge $(a_i, a_j) \notin E$, then $c(a_i, a_j) = 0$.

The definition of a flow network is defined as follows:

Definition 2.6. A *flow network* is a weighted directed graph $G(V, E)$ where V is a set of nodes containing a source node s, a sink node t and additional intermediate nodes. For any intermediate node, there is a path from the source s to the sink t that passes through this node. Moreover, for any edge $(a_i, a_j) \in E$ is assigned a value called a capacity $c(a_i, a_j) \geq 0$ such that the material passing through this edge cannot exceed its capacity. We give a zero capacity for any edge that is not in E. That is, if an edge $(a_i, a_j) \notin E$, then $c(a_i, a_j) = 0$.

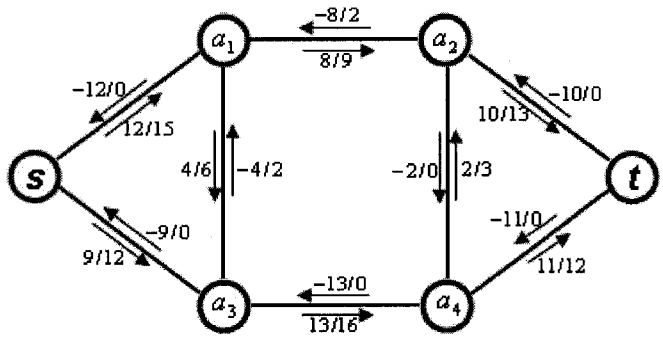


Figure 2.7: **A flow network showing *flow/capacity***

Now we define a flow in a network by the following definition.

Definition 2.7. Given a flow network $G(V, E)$ with capacity $c(a_i, a_j) \geq 0$ for all $a_i, a_j \in V$, let s be the source node and t be the sink node of this network. A *network flow* $f(a_i, a_j)$ from node a_i to node a_j is a real function $f : V \times V \rightarrow \mathbb{R}$ with the following properties for all nodes $a_i, a_j \in V$:

Capacity Constraints: $f(a_i, a_j) \leq c(a_i, a_j)$. The flow along an edge cannot exceed its capacity.

Skew symmetry: $f(a_i, a_j) = -f(a_j, a_i)$. The net flow from a_i to a_j must be the opposite of the net flow from a_j to a_i.

Flow conservation: $\sum_{a_j \in V} f(a_i, a_j) = 0$, $a_i \in V - \{s, t\}$, the net flow to a node is zero, except for the source s, which supplies flow, and the sink t, which demands flow.

The quantity $f(a_i, a_j)$ is called the *net flow* from node a_i to node a_j. The total net flow out of the source s is called the *flow f of a network* and can be computed as:

$$|f| = \sum_{a \in V} f(s, a).$$

Example 2.4.

Consider the weighted directed graph $G(V, E)$ shown in Figure 2.7 that represents a network flow of a source s, sink t, and 4 additional intermediate

nodes a_1, a_2, a_3 and a_4. A weight assigned in an edge is a pair f/c that represents the flow and the capacity. The flow f of this network is

$$|f| = f(s, a_1) + f(s, a_3) = 12 + 9 = 21.$$

There is no edge from node a_2 to node a_4, therefore the capacity $c(a_2, a_4) = 0$. We can observe that this graph satisfies all network flow properties. For example, the amount of flow does not exceed the capacity for any node on this graph. For instance,

$$f(a_2, a_3) = 4 < c(a_2, a_3) = 6.$$

Moreover, the net flow from node a_i to node a_j is the negative of the net flow from node a_j to node a_i for all nodes in G. For example,

$$f(a_2, a_3) = 4 = -f(a_3, a_2) = -4.$$

In addition, for any intermediate node $a_i \in V$ we have $\sum_{a_j \in V} f(a_i, a_j) = 0$. For example,

$$\sum_{a_j \in V} f(a_1, a_j) = f(a_1, s) + f(a_1, a_3) + f(a_1, a_2) = -12 + 4 + 8 = 0.$$

A *maximum network flow* problem in a flow network with source s and sink t is a problem of finding the maximum value of flow from the source s to the sink t, and/or computing the value of this flow. There are several algorithms for solving this problem. In the next chapter we will discuss one of these algorithms called the Ford-Fulkerson algorithm, which can solve a maximum network flow.

2.2 Greedy algorithms for Solving Combinatorial Optimization Problems

There are various significant algorithms that solve combinatorial optimization problems. Examples of these algorithms are the local search algorithm, greedy algorithms, the grasp algorithm and the genetic algorithm [9]. The choice of which of these algorithms to use depends on the kind of problem. In this study we are particularly interested in greedy algorithms and the local search algorithm. In this chapter, we will discuss greedy algorithms and their applications.

2.2.1 Greedy Algorithms

Greedy algorithms are algorithms that search for the best (an optimal) solution to combinatorial optimization problems by building the solution using several iterations. At each iteration, they make the optimal choice. In general, greedy algorithms have the following characteristics:

1. They compile a candidate set whose elements are all candidate parts of the solution from which the final solution is built.

2. They make a selection rule that determines which element we can choose from the candidate set.

3. They obtain a decision rule that decides whether or not the candidate solution will be added to the final solution.

4. They have an objective function that computes the value of the final solution that they built.

5. Finally, they state a solution function that decides if the final solution is a complete solution or not.

The way the greedy algorithms is used sometimes does not end with an optimal solution, or also it might not completely solve the problem. This is because the final solution that the greedy algorithm builds does not completely satisfy all the constraints of the problem. When this happens, it gets blocked and cannot find a solution. On the other hand, some greedy algorithms can solve completely some combinatorial optimization problems and can find a solution. This solution might not be the optimal one, but it will be close to the optimal. For example, Prim's algorithm and Kruskal's algorithm, which are examples of greedy algorithms, can find an optimal solution of the minimal spanning tree [43]. Whereas, Kruskal's algorithm can find a solution that is near to the optimal one of the traveling salesman problem. Another example of a greedy algorithm is called Dijkstra's algorithm [13] which can be used to solve a single-source shortest path problem and all-pair shortest path problem. These algorithms will be discussed in the following sections.

2.2.2 Kruskal's algorithm for Traveling salesman problem

In this section we state un example of a greedy algorithm that can be used to solve the traveling salesman problem (TSP)(see Section 2.1.1). This algorithm, which is called Kruskal's algorithm, was discovered by Joseph Kruskal in 1956 [43].

Usually, Kruskal's algorithm can find a solution to an instance of the traveling salesman problem that is near to the optimal solution. It is especially effective when the graph is complete [7]. In general, the steps of solving the TSP using Kruskal's algorithm are as follows:

Step 1 Compile the candidate set whose elements are the edges that form the parts of the solution.

Step 2 Sort the edges in descending order according to their weights.

Step 3 Choose the first immediate candidate edge from the candidate set.

Step 4 Select or ignore the edge chosen in step 3. That is, we select the edge in step 3 if it satisfies the following two conditions when compared with the all edges that already have been chosen:

1. No vertex has a degree of more then two.
2. These edges do not form a cycle in the result graph, except if the number of the vertices is equal to the number of edges that have already been chosen.

If the chosen edge satisfies these conditions, then it becomes part of the solution. That is, we add it to the final solution and continue the procedure by repeating step 3 and step 4. Otherwise, we ignore the chosen edge and choose another edge by repeating step 3 and step 4. The algorithm is terminated when n edges have been found.

Example 2.5.

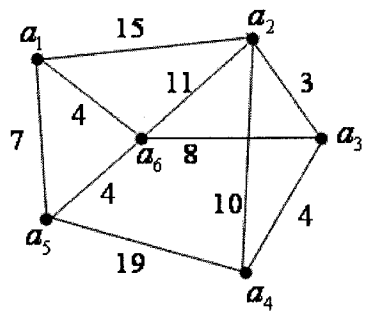


Figure 2.8: A six cities instance of TSP

An instance of TSP of six cities along with edges and their corresponding weights, is shown in Figure 2.8. According to the steps in Kruskal's algorithm, we have the following:

1. The candidate set p is of the form

$$p = \{(e_{12}, 15), (e_{15}, 7), (e_{56}, 4), (e_{23}, 3), (e_{24}, 8),$$
$$(e_{26}, 11), , (e_{16}, 4), (e_{36}, 10), (e_{45}, 19), (e_{34}, 4)\}$$

2. The candidate set p after being sorted in descending order is as follows:

$$p = \{(e_{23}, 3), (e_{56}, 4), (e_{16}, 4), (e_{34}, 4), (e_{15}, 7),$$
$$(e_{24}, 8), (e_{36}, 10), (e_{26}, 11), (e_{12}, 15), (e_{45}, 19)\}$$

3. In step 3 and step 4 we have the following output

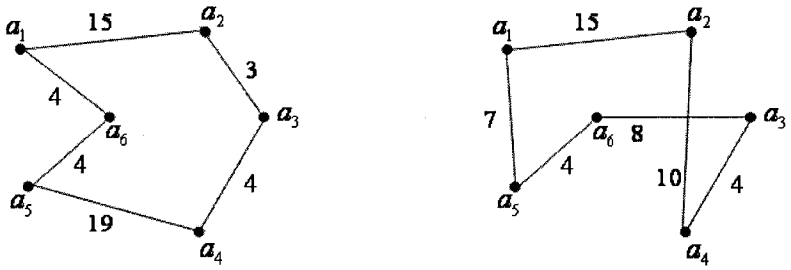


Figure 2.9: A tour of cost = 49, at the left, and an optimal tour of cost = 48 on the right, of the instance of the TSP in Example 2.5.

Step 3 Chosen edge	Step 4: Decision
e_{23}	Select
e_{56}	Select
e_{16}	Select
e_{34}	Select
e_{15}	Ignore: Since it forms a cycle with edges e_{56} and e_{16}
e_{24}	Ignore: Since it forms a cycle with edges e_{23} and e_{34}
e_{36}	Ignore: Since the vertex a_3 will have more than two degrees
e_{26}	Ignore: Since the vertex a_6 will have more than two degrees
e_{12}	Select
e_{45}	Select

When the fifth edge has been reached, we stop this process and conclude that the final tour is reached. We found that the final tour tour is $y = (123456)$ with cost 49 (shown in Figure 2.9). Although this is a better solution, it is not the optimal one. This is because the optimal solution has a cost of 48, (see in Figure 2.9).

Finally, we have reached a tour of cost 49 which is near to the optimal solution that has a cost of 48 (Figure 2.9).

As we see, Kruskal's algorithm has found a solution that is near to the optimal solution of TSP. In addition, this problem can be solved using the local search algorithm, which is one of the interesting algorithms in combinatorial optimization. This will be provided in the next chapter.

Next section will discuss another example of a greedy algorithm. This algorithm is called Prim's algorithm, and it can be used to solve the minimal spanning tree problem.

2.2.3 Prim's Algorithm for Minimal Spanning Tree

Prim's algorithm, an example of a greedy algorithm, can find a spanning tree in a connected weighted graph that has minimal weight. This algorithm was given by Robert Prim in 1957 [43]. Prim's algorithm uses several iterations and through these iterations it builds a tree that contains all vertices of the graph, has no simple circuits, and has minimum weight.

The steps of building a minimal spanning tree in a connected weighted graph G where Prim's algorithm is used are as follows:

Step 1 Select the edge that has the lowest weight. If there is more than one, we choose one of them arbitrarily. From this edge we build a tree T consisting of this edge and the corresponding two vertices.

Step 2 Select the least weighted edge that goes from only the vertices in the tree T, and that does not form a simple circuit in the tree T. Then, add this edge and the new vertex to the tree T.

Step 3 Repeat step 2 until $n-1$ edges have been added to the tree T, assuming that the graph G has n vertices. At that time, we have created a tree that contains all vertices of the graph G with no simple circuits and with minimum weight. That is, the minimum spanning tree of the graph G is reached.

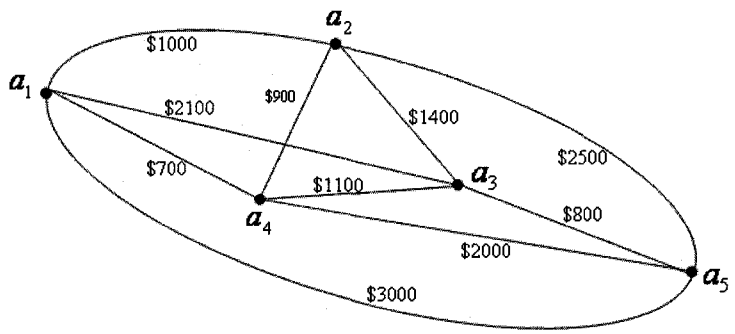


Figure 2.10: A connected weighted graph representation of a phone network design with five cities

The algorithm for finding a minimal spanning tree in a connected weighted graph using Prim's algorithm is as follows [43]:

Algorithm 1: Prim's Algorithm for finding the minimal spanning tree of a connected weighted graph $G = (V, E)$ with n vertices.

Input: The set of edges E of G, and the weights $w(e)$ for every $e \in E$.
Output: A minimal spanning tree T of the graph G

$T := \{\}$, " a tree with no vertices"
for $i := 1$ to $n - 1$
begin $e := e \in E$
  if (e is the least weighted edge that goes from only the vertices in the tree T, and that does not form a simple circuit in the tree T) **do**
    $T := T \cup \{e\}$
  end
end
return T (T is the minimal spanning tree of G)

Example 2.6.
Recall from Chapter 2, the example of the phone network design with five cities, (Example 2.3). The connected weighted graph representation of this

problem is shown in Figure 2.10. The problem is to find links that connect all the offices so that the total cost of these links is minimum. That is, we want to find a spanning tree in this graph that has minimum weight.

The steps that Prim's algorithm uses to solve this problem, by building a spanning tree that has minimal weight, are as follows. First, it starts by choosing an edge that has minimum weight. This edge is the one that connects cities a_1 and a_4 that has a cost of \$700. We call this edge $e_1 = \{a_1, a_4\}$. From this edge and the two vertices, we form a tree denoted by T. The goal is for this tree to be a spanning tree with minimal cost. Second, we select the least weighted edge that goes from only the vertices in the tree T, which are a_1 and a_4, and that does not form a simple circuit in the tree T. We found that, this edge is $e_2 = \{a_4, a_2\}$ with a cost equal to \$900. After that, we add this edge and the new vertex, a_2, to the tree T. Next, we repeat the same procedure to find the remaining edges and form a spanning tree that has minimum weight. Note that, although the edge $\{a_1, a_2\}$(with cost \$1000) has less weight than the edge $\{a_4, a_3\}$(with cost \$1100), we selected the edge $\{a_4, a_3\}$. This is because the edge $\{a_1, a_2\}$ forms a simple circuit in the tree T. The complete steps of Prim's algorithm for solving this problem are given in the following table.

Chosen edge	*Decision*	*Weight*
$\{a_1, a_4\}$	*Select*	*\$700*
$\{a_4, a_2\}$	*Select*	*\$900*
$\{a_1, a_2\}$	*Ignore: Forms a simple circuit with edges* $\{a_1, a_4\}$ *and* $\{a_4, a_2\}$	—
$\{a_4, a_3\}$	*Select*	*\$1100*
$\{a_3, a_5\}$	*Select*	*\$800*
		Total: \$3500

Finally, the minimal spanning tree T in this graph has a weight of cost \$3500 (Figure 2.11).

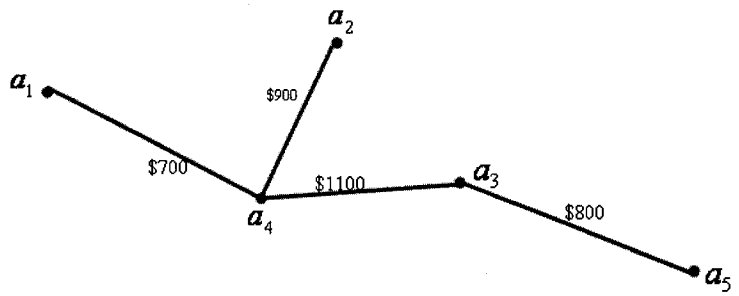


Figure 2.11: An optimal spanning tree for the phone network design of cost $3500

The minimal spanning tree can also be solved by Kruskal's algorithm, which we will provide in the following section.

2.2.4 Kruskal's Algorithm for Minimal Spanning Tree

In Section 2.2.2 of this chapter, we showed that Kruskal's algorithm can solve a traveling salesman problem. In this section we will use this algorithm to solve the minimal spanning tree problem (see Section 2.1.2). A minimal spanning tree problem is a problem that searches for a spanning tree of a weighted connected graph such that this tree has minimal weight. Kruskal's algorithm can be used to find such a tree. It starts by selecting an edge that has minimal weight and consecutively collects edges that have minimal weight and do not form a simple circuit with the edges already chosen. It continues collecting edges until $n - 1$ edges have been collected, assuming that the graph has n vertices.
The steps of finding a minimal spanning tree using Kruskal's algorithm in a connected weighted graph G of n vertices are as follows:

Step 1 Select an edge that has the lowest weight. If there is more than one, we choose one of them arbitrarily.

Step 2 Select the first immediate least weighted edge that does not form a simple circuit with the edges that already have been chosen.

Step 3 Repeat step 2 until $n - 1$ edges have been added. At that time, we have reached a tree that contains all vertices of the graph G with

no simple circuits and has minimum weight. That is, the minimum spanning tree of the graph G is founded.

The following pseudocode is for Kruskal's algorithm for finding a minimal spanning tree for a weighted connected graph $G(V, E)$ of n vertices [43].

Algorithm 2: Kruskal's Algorithm for finding the minimal spanning tree of a connected weighted graph $G = (V, E)$ with n vertices.

Input: The set of edges E of the graph G, and the weights $w(e)$ for every $e \in E$.

Output: A minimal spanning tree T of the graph G

$T := \{\}$, " a tree with no vertices"
for $i := 1$ to $n - 1$
begin
  $e := e \in E$
  if (e is the least weighted edge that does not form a simple circuit with the edges of T) **do**
    $T := T \cup \{e\}$
  end
end
return T, (T is the minimal spanning tree of G)

Example 2.7. *In Example 2.6, we have solved the phone network design problem with five cities, Figure 2.12, using Prim's algorithm. In this example, we want to solve this problem using Kruskal's algorithm. Recall that the goal of this problem is that we have five offices located in five different cities and want to find links that connect all these offices so that the total cost of these links is the minimum. That is, we want to find a spanning tree*

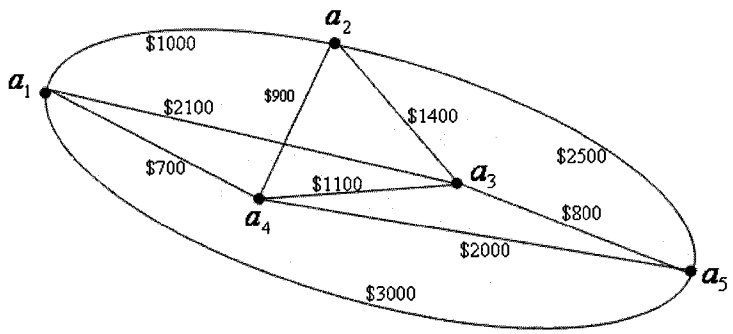


Figure 2.12: A connected weighted graph representation of a phone network design with five cities

which contains all cities and has minimum weight.

Kruskal's algorithm can solve this problem by selecting an edge that has minimum weight and does not form a simple circuit with edges already chosen. Therefore, the first smallest weighted edge, in Figure 2.12, is $e_1 = \{a_1, a_4\}$ *of cost \$700. The second smallest edge is* $e_2 = \{a_3, a_5\}$ *of cost \$800. Clearly these edges do not form a simple circuit with edges already chosen since they are only two edges. Edge* $e_3 = \{a_2, a_4\}$ *is the next smallest weighted edge of cost \$900 that does not form a simple circuit with edges* e_1 *and* e_2 *(see Figure 2.13). The following edge that has the smallest weight is* $e_2 = \{a_1, a_2\}$*, however, this edge forms a simple circuit with edges already chosen, which are edge* e_1 *and edge* e_3*. Therefore, we choose edge* $e_4 = \{a_4, a_3\}$ *of cost \$1100 since it does not form a simple circuit with edges already chosen. Since the total number of selected edges is* $4 = n - 1 = 5 - 1$*, where* n *is the total number of vertices in the graph, we terminate the procedure and the minimal spanning tree is reached. This tree has a total cost equal to \$3500, which is an optimal cost, and composed of edges* $e_1 = \{a_1, a_4\}, e_2 = \{a_3, a_5\}, e_3 = \{a_2, a_4\}$*, and* $e_4 = \{a_4, a_3\}$*. This solution is the same solution that have been found using the Prim's algorithm, (see Figure 2.11).*

In the next section we will state another example of greedy algorithms that can be used to solve single-source shortest path problem.

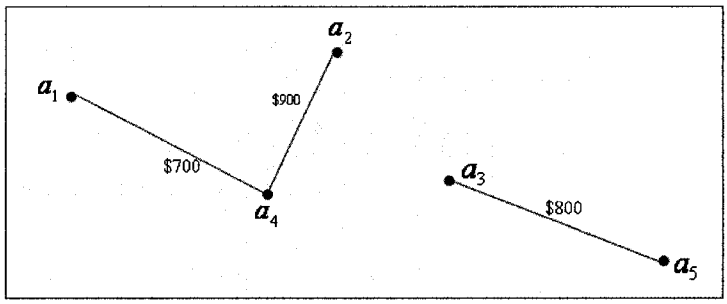


Figure 2.13: A tree (not spanning tree) consists of three edges that were found using the Kruskal's algorithm in Example 2.7

2.2.5 Dijkstra's Algorithm for Single-Source Shortest Path problem

Dijkstra's algorithm is an example of a greedy algorithm that can solve the shortest path problem (see Section 2.1.3). The shortest path problem is a problem that searches for the smallest weighted path between two vertices in a weighted directed (or undirected) graph. Dijkstra's algorithm, which was discovered by Dutch mathematician E. Dijkstra in 1959 [43], can find this path by finding the shortest path from the initial vertex to the following vertices, until the terminal vertex is achieved. That is, it searches for the path that has the smallest weight from the initial vertex to the first vertex, the path that has the smallest weight from the initial vertex to the second vertex, and continues the same process until it arrives at the terminal vertex.

Dijkstra's algorithm searches for the shortest path from vertex a to vertex z in a weighted undirected graph using several iterations. Through these iterations, it composes a set that is a subset of the set V. This set is called the *distinguished* set and is denoted by S_k, where k refers to the iteration k. At each iteration, one new vertex is added to the distinguished set S_k. The new vertex is added such that it has the smallest path that contains only vertices from the previous distinguished set S_{k-1}. That is, a vertex u is added to S_k if there is a path from vertex a to vertex u that contains all vertices in S_{k-1} and has the smallest weight. We let $L_k(u)$ be the weight of the smallest path from vertex a to vertex u containing only

the vertices in the distinguished set S_k.

Now, we state the steps of Dijkstra's algorithm for finding the shortest path between two vertices in a weighted undirected graph. Given a weighted undirected graph $G(V, E)$ and assuming that we want to find a shortest path from vertex a to vertex z, the steps of Dijkstra's algorithm in searching for this path are as follows:

Step 1 It starts by initializing all vertices in the graph by labeling the vertex a with zero and all other vertex by ∞. That is,

$$L_0(a) = 0, L_0(u) = \infty, \forall u \in V - \{a\}.$$

Recall that, $L_k(u)$ is the weight at iteration k of the smallest path from vertex a to vertex u containing only the vertices in the distinguished set S_k. Now, since there is no vertex added to S_0, we let $S_0 = \phi$ at iteration zero.

Step 2 At each iteration k, we add the next smallest labeled vertex to the distinguished set S_k that is noted in the previous distinguished set S_{k-1}. That is, a vertex u is added to S_k if there is a path from vertex a to vertex u that contains all vertices in S_{k-1} and has the smallest weight. In other words, a vertex u is added to S_k if:

$$L_{k-1}(u) < L_{k-1}(t), \forall t \notin S_{k-1}.$$

Step 3 Update the labels of all vertices that are not in S_k, that is, the vertices that are in $S_{k-1} \bigcup \{u\}$, where u is the vertex that is added in step 2.

We update the label at iteration k of a vertex v, $L_k(v)$, that is not in S_k by the following. $L_k(v)$ is the weight of the shortest path from vertex a to vertex v that contains only the vertices from S_{k-1}, that is, $S_k - \{u\}$, or is the weight of the shortest path from the vertex a to vertex u at iteration $k-1$ plus the weight of edge $\{u, v\}$. That is,

$$L_k(v) = min\{L_{k-1}(v), L_{k-1}(u) + \omega(u, v)\}, \tag{2.4}$$

where $\omega(u, v)$ is the weight of edge {u,v}.

Step 4 Repeat step 2 and step 3 until the terminal vertex z is added to the distinguished set. That is, the procedure is terminated when $z \in S_k$.

When the vertex z is added to the distinguished set, the shortest path from vertex a to vertex z is reached and has the weight given by $L_k(z)$, where k is the last iteration where vertex z is added.

The pseudocode of Dijkstra's algorithm in finding the shortest path between two vertices, say from vertex a to vertex z, in a weighted connected graph with n vertices and non-negative weights is as follows [43]:

Algorithm 3: Dijkstra's Algorithm for finding the shortest path between vertex a to vertex z in the non-weighted connected graph $G = (V, E)$.

Input: $V = \{a = a_1, a_2, \ldots, a_n = z\}$, $\omega_{ij} \geq 0$ and $\omega_{ij} = \infty$ if $\{a_i, a_j\} \notin E$
Output: $L(z)$ The shortest path from vertex a to vertex z

1. for $i := 1$ to n
$L(a_i) := \infty$

2. $L(a) := 0$

3. $S := \phi$
(steps 1 to 3 Initialize the labels of every vertex in G and let the distinguish set S be the empty set.

while $z \notin S$

begin

4. $v :=$ where $v \notin S$ and has the smallest label $L(v)$
5. $S := S \cup \{v\}$

6. for all vertex not in S
if $L(v) + \omega(v, b) < L(b)$ **then** $L(b) := L(v) + \omega(v, b)$
(Steps 4 to 6, add a new vertex, v, to the set S and update the label of every vertex $b \notin S$)

end

return $L(z)$

(Where $L(z)$ is the weight of the shortest path from vertex a to vertex z)

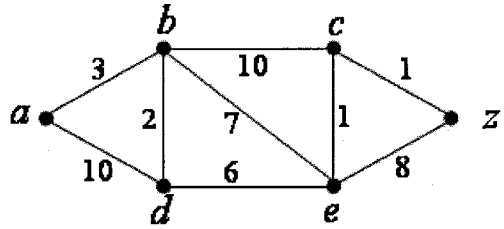


Figure 2.14: A weighted undirected graph with six cities of Example 2.8.

Example 2.8.

Consider a weighted undirected graph $G(V, E)$ with six vertices shown in Figure 2.14. Assume that we want to find the shortest path from vertex a to vertex z. Dijkstra's algorithm can be used to find such a path as follows: First, we begin by labeling vertex a with zero and all other vertices with ∞. That is,

$$L_0(a) = 0, L_0(t) = \infty, \forall t \in V - \{a\},$$

(see Figure 2.15-A). The distinguished set has no elements at this iteration, that is, $S_0 = \{\}$.

Next, since the smallest labeled vertex is a, $L_0(a) = 0$, and all other labeled vertices are ∞, we add vertex a to the set S_1. That is, $S_1 = \{a\}$. Now, S_1 has only vertex a, so the following labels are as follows:

$$L_1(b) = 3, L_1(d) = 6, \ and, \ L_1(v) = \infty \ \ \forall v \in \{c, e, z\}.$$

(see Figure 2.15-B). Now, we repeat the same steps by adding the smallest labeled vertex that is not in $S_1 = \{a\}$. This vertex is b with $L_1(b) = 3$. That is, the distinguished set will be $S_2 = \{a, b\}$. Moreover, we update the labels of all vertices not in S_2 using formula 2.4, at iteration $k = 2$:

$$L_2(v) = min\{L_1(v), L_1(b) + \omega(b, v)\}, \ \forall v \notin S_2 = \{a, b\}.$$

For instance,

$$\begin{aligned} L_2(d) &= min\{L_1(d), L_1(b) + \omega(b,d)\} \\ &= min\{6, 3+2\} \\ &= 5, \end{aligned}$$

(see Figure 2.15-c). We continue this procedure until vertex z is added to the distinguished set. The complete solution of this problem solved by Dijkstra's algorithm is shown in Figure 2.15. In this figure, we circle the elements of the distinguished set at k iteration, S_k and indicate the smallest path between vertex a and each vertex and containing only vertices in S_k. At the final iteration, when vertex z is added to the distinguished set, the shortest path from vertex a and vertex z is reached and has a weight equal to the value of $L_k(z)$, where k is the last iteration. This path is a, b, d, c, z, (see Figure 2.15-F).

In this section, we discussed Dijkstra's algorithm for solving the single-source shortest path problem. In the next section, we will see that this algorithm can be used to solve all-pair shortest path problem with non-negative weights. Moreover, we will discuss another example of a greedy algorithm that can be used to solve the all-pair shortest path problem with negative and non-negative weights. This algorithm is called Floyd's Warshall algorithm.

2.2.6 Floyd's Warshall Algorithm for the All-Pair Shortest Path (APSP) problem

Floyd's Warshall algorithm is an example of a greedy algorithm and can be used to solve the all-pair shortest path problem (Section 2.1.4). This problem can be described as follows. We are given a weighted undirected graph $G(V, E)$ and we want to find the shortest paths between every two vertices in this graph. That is, we need to determine a matrix called the *shortest path matrix*, denoted as $D[d_{i,j}]$, where the entry $d_{i,j}$ is a path from

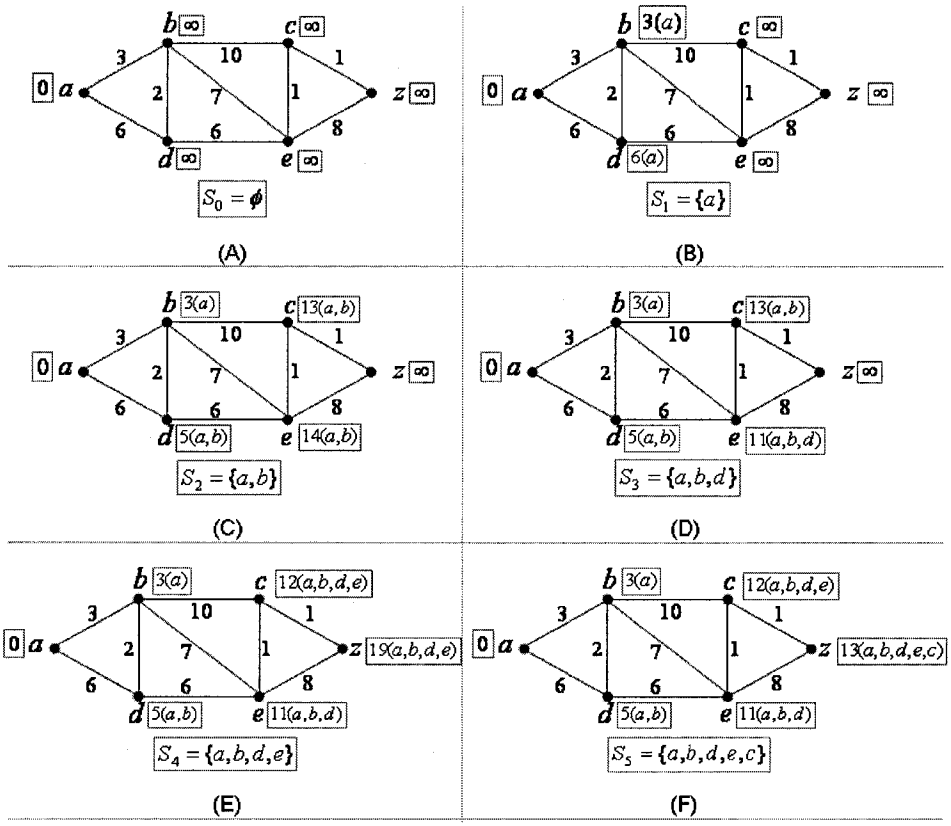


Figure 2.15: Dijkstra's algorithm for finding the shortest path between vertex a and b. The details of this figure are described in Example 2.8.

vertex a_i to vertex a_j, for $a_i, a_j \in V$, and has the smallest weight.

Floyd's Warshall algorithm can solve this problem and find the shortest paths matrix D by the following. Given a weighted directed graph $G(V, E)$ and the set of vertices $V = \{a_1, a_2, \ldots, a_n\}$, we are assuming that the weight of the edge $(a_i, a_j) \in E$ is non-negative for all $a_i, a_j \in V$. That is, the weight $\omega_{i,j}$ for the edge (a_i, a_j) is given by the following equation:

$$\omega_{i,j} = \begin{cases} 0, & \text{if } i = j; \\ \infty, & \text{if i} \neq j \text{ and } i \nsim j \text{ (the edge } (a_i, a_j) \notin E); \\ \omega_{i,j}, & \text{if } i \neq j \text{ and } (a_i, a_j) \in E). \end{cases} \tag{2.5}$$

Let matrix $W[w_{i,j}]$, where the entry $w_{i,j}$ is defined by equation 2.5, be the weight matrix of graph G.

Floyd's Warshall algorithm can find the shortest path matrix $D[d_{i,j}]$ by determining a sequence of matrices $D^{(0)}, D^{(1)}, \ldots, D^{(n)}$. A matrix $D^k[d^k_{i,j}]$ is the shortest path matrix at iteration k, where $0 \leq k \leq n$, and the entry $d^k_{i,j}$ is the weight of the shortest path from vertex a_i to vertex a_j with intermediate vertices only from the set $\{a_1, a_2, \ldots, a_k\}$. That is, matrix $D^{(k)}[d^k_{i,j}]$ considers only those paths where the only intermediate vertices are those from $\{a_1, a_2, \ldots, a_k\}$ and which have the lowest weight. Recall that the definition of the *intermediate vertex* of a path $P =< a_1, a_2, \ldots, a_{k-1}, a_k >$ is any vertex in a path P other than vertex a_1 and vertex a_k, that is, any vertex in the set $\{a_2, \ldots, a_{k-1}\}$. If $k = 0$, there is no intermediate vertex in a path from vertex a_i to vertex a_j. This path consists only of edge (a_i, a_j), and therefore, it has a weight equal to $\omega_{i,j}$. Hence, the shortest path matrix when $k = 0$ is the weighted matrix $W[\omega_{i,j}]$. Thus:

$$D^{(0)}[d^0_{i,j}] = W[\omega_{i,j}], \text{ where } \omega_{i,j} \text{ is defined in equation 2.5.}$$

For $k \geq 1$, a matrix $D^{(k)}[d^k_{i,j}]$ is computed from the matrix $D^{(k-1)}[d^{k-1}_{i,j}]$ as the following:

$$d^k_{i,j} = min\left(d^{k-1}_{i,j}, d^{k-1}_{i,k} + d^{k-1}_{k,j}\right) \tag{2.6}$$

Therefore, the matrix entries are all getting smaller through these iterations. That is, when k is increased, the matrix entries $d_{i,j}^k$ are decreased:

$$d_{i,j}^{k-1} \geq d_{i,j}^k.$$

When $k = n$, the shortest path matrix $D[d_{i,j}]$, which consists of the shortest path between all pairs of vertices in graph G, is reached and equal to $D^{(n)}[d_{i,j}^n]$. In other words,

$$D[d_{i,j}] = D^{(n)}[d_{i,j}^n],$$

and for any vertices $a_i, a_j \in V$, the weight of the shortest path between vertex a_i and vertex a_j is given by the value of the entry $d_{i,j}^n$.

Moreover, the shortest path between each pair of vertices of the graph G can be constructed using the predecessor matrix P. This matrix can be computed using the shortest path matrix $D[d_{i,j}]$. The algorithm constructs a sequence of predecessor matrices $P^{(0)}, P^{(1)}, \ldots, P^{(n)}$ where the entry $p_{i,j}^k$ of the predecessor matrix $P^{(k)}[p_{i,j}^k]$ is the predecessor of vertex j on the shortest path from vertex i to vertex j with intermediate vertices all in the set $\{a_1, a_2, \ldots, a_k\}$. That is, the predecessor matrix $P^{(k)}$ is computed from the matrix $D^{(k)}$ by the following:
At $k = 0$, the shortest path from vertex a_i to vertex a_j has no intermediate vertices. That is, the entry of the predecessor matrix at $k = 0$ has entry $p_{i,j}^0$ defined by:

$$p_{i,j}^0 = \begin{cases} NIL, & \text{if } d_{i,j}^0 = 0 \text{ or } d_{i,j}^0 = \infty \text{ where } d_{i,j}^0 \text{ is the entry of} \\ & \text{the matrix } D^{(o)}; \\ i, & \text{otherwise.} \end{cases}$$

For $k \geq 1$, the entry $p_{i,j}^k$ of the predecessor matrix $P^{(k)}$ is computed as the following:

$$p_{i,j}^k = \begin{cases} p_{i,j}^{k-1}, & \text{if } d_{i,j}^{k-1} \leq d_{i,k}^{k-1} + d_{k,j}^{k-1}; \\ p_{k,j}^{k-1}, & \text{else.} \end{cases}$$

The APSP also can be solved using Dijkstra's algorithm. That is, we apply this algorithm for all pairs of vertices of a connected weighted graph. A

graph G with n vertices has $\binom{n}{2}$ pairs of vertices where

$$\binom{n}{2} = \frac{n!}{2!(n-2)!}.$$

Moreover, for each pair of the vertices of G we should use Dijkstra's algorithm n times. Therefore, to solve the APSP we need to apply Dijkstra's algorithm $n \times \binom{n}{2}$ times. This algorithm can be used only if the weights of a graph are non negative, whereas Floyd's algorithm works even if the weights of a graph are negative.

2.2.7 Ford-Fulkerson's Algorithm for Maximal Network Flow Problem

Ford-Fulkerson's algorithm can solve the maximal network flow problem discussed in Section 2.1.5. Recall that a flow network is a directed graph $G = (V, E)$ where the set of nodes V contains a source node s, a sink node t and extra intermediate nodes (any node other than s and t). A flow network should satisfy the following constructions. Each edge $e \in E$ is assigned a capacity c where the flow along this edge cannot exceed its capacity. The net flow $f(a_i, a_j)$ from node a_i to node a_j must be the opposite of the net flow from a_j to a_i, that is, $f(a_i, a_j) = -f(a_j, a_i)$, for all nodes $a_i, a_j \in V$. Moreover, the flows into a node should equal the flows that leave this node except for the source s, which has only outgoing flow, and the sink t, which has only incoming flow.

The maximum network flow problem is a problem of determining the maximum amount of flow from the source s to the sink t in a flow network without violating the restrictions of the capacity and the flow for each node in the network.

Ford-Fulkerson's algorithm can solve the maximum network flow problem by finding a path from the source s to the sink t with positive capacity along all its edges. This path is called an augmenting path. Along this path, we push flow. We continue finding such paths until no more paths

can be found.

There are three important ideas that the Ford-Fulkerson's algorithm uses for solving such problems. The ideas are the residual network, augmenting paths, and cuts. The max-flow min-cut theorem, which will be described later in this Section 2.1.5, will be used to show that the maximum flow in a flow network is the same as the capacity of some cut in this network. To illustrate the algorithm let us state some important definitions and theorems.

Definition 2.8.
Given a flow network $G(V, E)$ with source s and sink t, let $f(a_i, a_j)$ be a net flow in G from node a_i to node a_j:

1. The *residual capacity* of an edge (a_i, a_j) is the amount of additional flow we can send from node a_i to a_j before exceeding the capacity $c(a_i, a_j)$, denoted as $c_f(a_i, a_j)$ and defined by the following equation:

$$c_f(a_i, a_j) = c(a_i, a_j) - f(a_i, a_j). \tag{2.7}$$

   For example, if $c(a_i, a_j) = 20$ and $f(a_i, a_j) = 15$, then we can send 5 units more through this edge.

2. A *residual network* associated with a flow network $G(V, E)$ with a net flow f is denoted as $G_f(V, E_f)$ where E_f is a set of edges defined as

$$E_f = \{(a_i, a_j) \in V \times V : c_f(a_i, a_j) > 0\}.$$

   and

$$c_f(a_i, a_j) = c(a_i, a_j) - f(a_i, a_j).$$

   The goal of a residual network is to show what amount of capacity is available in each node in the original network. In a residual network, there might exist an edge (a_i, a_j) that does not appear in the original network.

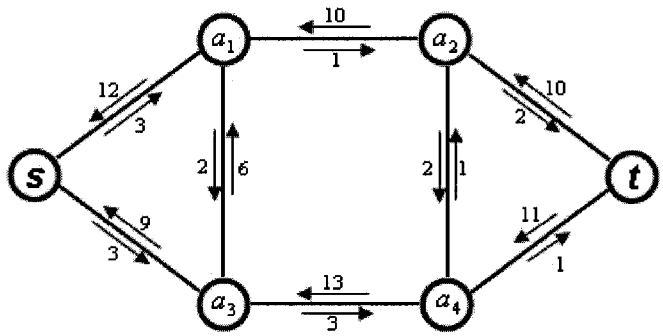


Figure 2.16: **A residual network for Figure 2.7, showing *capacity***

3. A path $(a_1, a_2, \ldots, a_k)$ in a flow network is called *an augmenting path* if it is simple and $a_1 = s, a_k = t$, and $c_f(a_i, a_{i+1}) > 0$ for $i = 1, \ldots, k-1$. That is, along any augmenting path we can send more flow through this path. *The residual capacity of* p is denoted as $c_f(p)$ and given by $c_f(p) = min\{c_f(a_i, a_j) : (a_i, a_j) \in p\}$

4. A *Cut* on the flow network $G(V, E)$ is denoted as (S, T) where S and T are a subset of the set V such that $T = V - S$, $s \in S$, and $t \in T$. We define the *capacity* of the cut (S, T) as $c(S, T)$ and the *net flow* across this cut as $f(S, T)$ where f is a flow.

Example 2.9. The residual network $G_f(V, E_f)$ of a flow network of Figure 2.7 defined in section 1.5 is shown in Figure 2.16. In a path $p = (s, a_1, a_2, t)$ we have the following residual capacities:

$$c_f(s, a_1) = 3, c_f(a_1, a_2) = 1, c_f(a_2, t) = 2,$$

which are greater than 0. That is, this path is an augmenting path with residual capacity :

$$\begin{aligned} c_f(p) &= min\{c_f(s, a_1), c_f(a_1, a_2), c_f(a_2, t)\} \\ &= min\{3, 1, 2\} \\ &= 1 \end{aligned}$$

This means that we can push one unit more from the source s to the sink t along this path.

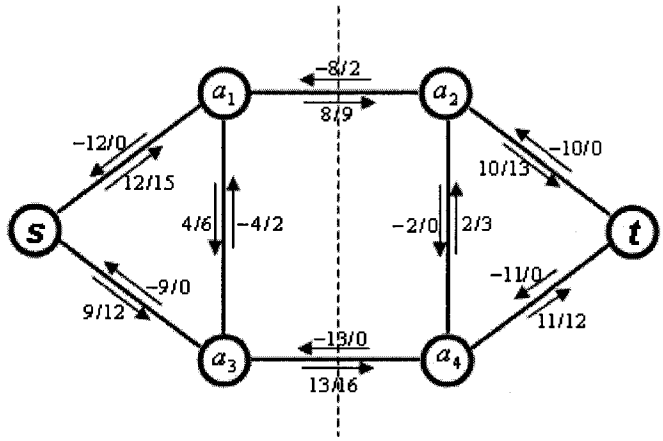


Figure 2.17: **A cut (S,T) of the flow network in Figure 2.7 where $S = \{s, a_1, a_2\}$ and $S = \{a_3, a_4, t\}$. The detail of this figure is shown in Example 2.9**

A cut (S,T) on the flow network of Figure 2.7, is shown in Figure 2.17, where $S = \{s, a_1, a_2\}$ and $S = \{a_3, a_4, t\}$. The net flow of this cut is

$$f(S,T) = f(a_1, a_2) + f(a_3, a_4) = 8 + 13 = 21.$$

The capacity across this cut is

$$c(S,T) = c(a_1, a_2) + c(a_3, a_4) = 9 + 16 = 25.$$

Theorem 2.2.1. *(Max-flow min-cut theorem)*
Let $G = (V, E)$ be a flow network with a source s and a sink t. If f is a flow in G, then the following are equivalent:

1. *f is a maximum flow in G.*
2. *There are no augmenting paths in the residual network G_f.*
3. *There is a cut (S,T) in the flow network G such that $|f| = c(S,T)$.*

Proof. For the proof of this theorem see [13]. □

Now, we can describe the basic steps of the Ford-Fulkerson's algorithm for solving the maximum network flow problem.

Input : A flow network G with capacity c, a source node s, and a sink node t

Output : Flow f such that f is maximal from s to t

Step 1 Initialize flow f:
for each edge $(u, v) \in E$ **do**

- $f(u, v) := 0$
- $f(v, u) := 0$

Step 2 Find an augmenting path p with $c_f(p) > 0$: That is,find any path p in the residual network G_f. If there are no more such paths, return (f) which is the maximum flow of G.

Step 3 Calculate the residual capacity of p:

- $c_f(p) = min\{c_f(u, v) : (u, v) \in p\}$

Step 4 Augment flow f along the path p by the residual capacity $c_f(p)$:
for each edge (u, v) in p **do**

- $f(u, v) := f(u, v) + c_f(p)$
- $f(v, u) := -f(u, v)$

Step 5 Go to step 2.

When this algorithm is terminated, that is, when there are no more augmenting paths in G, the maximum flow of this network is the flow f

As we have seen, greedy algorithms can be used in solving certain combinatorial optimization problems. The next chapter discusses the local search algorithm, which is one of the interesting algorithms in combinatorial optimization.

2.3 The Local Search Algorithms

The local search algorithm is one of the successful algorithms for solving combinatorial optimization problems. It searches through the space of candidate solutions called the *search space*. In the search space it searches for the best solution. It moves from solution to solution until it finds a

solution that can be considered, in some sense, the best solution. This solution sometimes is the optimal solution or near the optimal. To illustrate the local search algorithm, let us start with the following section that states the important definitions and properties of this algorithm.

2.3.1 Definitions and Properties

Definition 2.9. Given an optimization problem with an instance (F, c), a *neighborhood* is a mapping

$$N : F \rightarrow 2^F$$

where 2^F is the *powerset* of F, which is defined as the set of all subsets of F. That is,

$$2^F = \{S : S \subset F\}, \text{ and } |2^F| = 2^{|F|}.$$

Definition 2.10. Given a feasible solution $y \in F$ in a particular problem with instance (F, c), the set

$$N(y) = \{g : g \in F, \text{ g is " close " in some sense to the solution } y\}$$

is called the *neighborhood of y.*

Example 2.10.

The neighborhood of a feasible solution y of an instance of the traveling salesman problem is called k-change and defined as follows:

$N_k(y) = \{g : g \in F$ and g can be obtained from y by removing k edges from y and replacing them with k new edges$\}$.

For example, a feasible solution $y \in F$ of an instance of TSP with 6 cities is shown in Figure 2.18. When $k = 2$, the 2-change neighborhood of y is defined as

$N_2(y) = \{g : g \in F$ and g can be obtained from y by removing 2 edges from y and replacing them with 2 new edges$\}$.

The feasible solution $g \in N_2(y)$ is defined by removing the two edges e_1, e_2 and replacing them with two new edges as shown in Figure 2.18

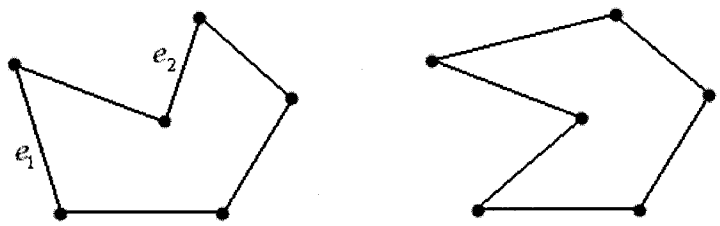


Figure 2.18: A feasible solution y of an instance of TSP and a feasible solution $g \in N_2(y)$

Finding the optimal solution of an optimization problem is usually difficult. On the other hand, we can often find the best solution in the neighborhood of some feasible point $y \in F$. This is called a local optimal solution, which is defined in the following definition.

Definition 2.11. Given an instance (F, c) of an optimization problem and a neighborhood N, a feasible solution $y \in F$ is called *local optimal* with respect to N if

$$c(y) \leq c(g), \qquad \forall \ g \in N(y)$$

assuming that this is a minimizing type problem, i.e., $t = min$.

The local search algorithm searches for the best solution in the neighborhood of a feasible solution of a combinatorial optimization problem, which is called a local optimal solution. That is, it moves from solution to solution in the feasible solution set until it reaches a solution that is no longer improved or some condition is satisfied. The condition is called a stopping rule and sometimes is considered as time bound or iterations bound.

Now, we briefly explain the local search algorithm for a given instance (F, c) of an optimization problem. Let $N(y)$ be the set of neighborhood of $y \in F$. Consider the following subroutine

$$improve(y) = \begin{cases} Any \ \ g & : g \in N(y), \ c(g) < c(y); \\ False & : \text{if no such g exists.} \end{cases}$$

The idea of the local search algorithm is that we start with an arbitrary initial feasible solution $y \in F$. Then we use a loop that searches for a better solution in the neighborhood of y, $N(y)$, using the subroutine $improve(y)$. The loop is stopped when the current solution no longer can be improved, and therefore, this solution is called the local optimal solution. The main body of the general local search algorithm is shown as follows:

Producing a local optima

begin

$y =$ completely random starting feasible solution belongs to F;

While ($improve(y) \neq False$) do

$y = improve(y)$;

return y

end

There are several decisions that we have to decide when we use this algorithm. First, the number of initial feasible solutions that we start with and how we classify them must be determined. Often, a local search algorithm uses several initial feasible solutions to start with and chooses the best result. The next decision that we should make is the neighborhood for the problem and the method for searching for it. This method provides a way of searching for the best solution in the set of the neighborhood. The size of the neighborhood usually determines the quality of the resulting local optimal solution. That is, a neighborhood which has the largest size usually leads to a better solution that is near to the global solution. On the other hand, using a neighborhood that has a large size takes more time to achieve the local optimal.

The local search algorithm can be used to solve important combinatorial optimization problems that arise in various areas. These include, for example, problems from engineering, computer science, operations research, and mathematics.

There are several examples of local search algorithms. GSAT and WALKSAT are examples of local search algorithms that can be used in solving the boolean satisfiability problem [46]. Another example of a local search algorithm is Lin-Kernighan [23] for solving the symmetric traveling salesman problem.

In the next section we will describe briefly the 2-opt (Lin-Kernighan) algorithm for solving the symmetric TSP.

2.3.2 Lin-Kernighan (2-opt) Algorithm for Symmetric Traveling Salesman Problem

In this section we will talk briefly about the Lin-Kernighan (2-opt) algorithm for solving the symmetric traveling salesman problems. For more detail about this method you can refer to [23, 34].

The Lin-Kernighan algorithm is one of the most successful examples of local search algorithms that can find an optimal or near to the optimal solution for the symmetric traveling salesman problems. It has been given the optimal solutions for all solved problem instances that we have been able to obtain, as well as the largest nontrivial problem instance, which is a 7397-city problem that has been solved to optimality today. Moreover, it has improved the best known solution for an 85900-city problem.

Recall that the goal of the traveling salesman problem is to find the cheapest tour that visits all given cities exactly once and then returns to the starting point. In weighted graph representation, we are looking for a path that has the minimum weight, starts and ends at the same vertex and visits each vertex one time. The cost of traveling from city i to city j is denoted as C_{ij}. If $C_{ij} = C_{ji}$, the problem is called a symmetric traveling salesman problem; otherwise it is called asymmetric TSP. Usually, the symmetric TSP is more difficult.

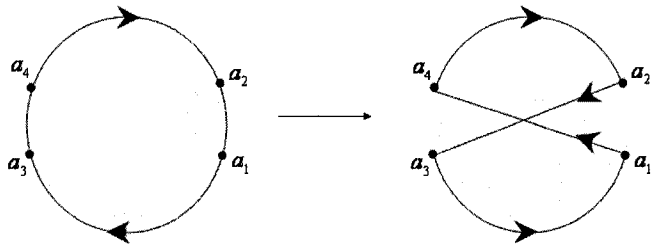


Figure 2.19: A 2-opt move of an instance of a TSP where a_i denotes the city i

The idea of 2-opt algorithm for solving the symmetric TSP is simple. It starts with a completely random tour and tries to find the best tour in its neighborhood using the subroutine improvement(y). It replaces 2 edges with 2 new edges so that the resulting tour is a feasible tour and has less cost. For example, Figure 2.19 shows a 2-opt move of an instance of a TSP where a_i denotes the city i. The algorithm continues improving the current tour until there are no more improvements of the current tour.

The 2-opt (or 2-change) neighborhood of a tour T, $N_2(T)$, of an instant of the TSP with n cities can be described as follows. Assume that the tour $T = \{t_1, \ldots, t_n\}$, where t_i is a city in this tour. A tour $\bar{T} \in N_2(T)$ if a set X of two edges is removed from the tour T and replaced by a set Y of two new edges such that the resulting tour is a feasible tour.

To insure that the resulting tour, $\bar{T}$, is a feasible tour, the edges in the two sets $X = \{x_1, x_2\}$ and $Y = \{y_1, y_2\}$ should be selected such that

$$x_1 = (t_i, t_{i+1}), \qquad y_1 = (t_i, t_j)$$
$$x_2 = (t_j, t_{j+1}), \qquad y_2 = (t_{i+1}, t_{j+1})$$

and

$$1 \leq i < j \leq n, \ j \neq i+1 \ and \ j \neq i-1 \tag{2.8}$$

In other words, a tour $T = (t_1 t_2 \ldots t_i t_j t_{j-1} \ldots t_{i+1} t_{j+1} \ldots t_n)$ is of the form

$$\bar{T} = (t_1 t_2 \ldots t_i t_j t_{j-1} \ldots t_{i+1} t_{j+1} \ldots t_n).$$

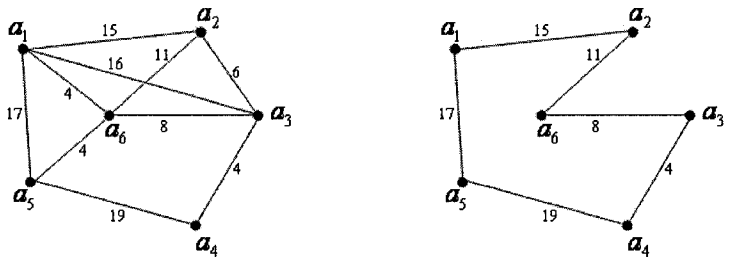


Figure 2.20: An instance of TSP of 5 cities of Example 2.11, on the left, and a feasible tour $T = (a_1a_2a_6a_4a_3a_5)$ on the right.

The following example is used to illustrate the 2-opt neighborhood.

Example 2.11.
Consider an instance of TSP of 6 cities $a_1, \ldots, a_6$ and a feasible tour $T = (a_1a_2a_6a_4a_3a_5)$ which are shown in Figure 2.20. To construct the 2-opt neighborhood of T, $N_2(T)$, first, rename the cities of T so that $T = (t_1t_2t_3t_4t_5t_6)$. That is,

$$t_1 = a_1, t_2 = a_2, t_3 = a_6, t_4 = a_4, t_5 = a_3 \ \ and \ \ t_6 = a_5,$$

(see Figure 2.21-A). Now, a tour $\bar{T} \in N_2(T)$ if $\bar{T}$ is of the form

$$\bar{T} = (t_1t_2 \ldots t_it_jt_{j-1} \ldots t_{i+1}t_{j+1} \ldots t_n)$$

For instance, if $i = 3$, than the edges of X and Y are

$$x_1 = (t_3, t_4), \quad y_1 = (t_3, t_j)$$
$$x_2 = (t_j, t_{j+1}), \quad y_2 = (t_3, t_4)$$

where t_j can be any vertex other than $t_{4=i+1}$ and $t_{2=i-1}$. For example, if $j = 6$, then

$$x_1 = (t_3, t_4), \quad y_1 = (t_3, t_6)$$
$$x_2 = (t_6, t_1), \quad y_2 = (t_4, t_1)$$

and the resulting tour is

$$\bar{T} = (t_1t_2t_3t_6t_5t_4) \in N_2(T),$$

(see Figure 2.21-B).

The cost of a tour $\bar{T} \in N_2(T)$ is denoted by $c(\bar{T})$ and can be computed as follows. If the tour $\bar{T}$ is composed by removing the edges in the sets $X = \{x_1, x_2\}$ from T and replacing them by the edges in the set $Y = \{y_1, y_2\}$, the cost of the tour $\bar{T}$ is given by

$$c(\bar{T}) = c(T) - G, \quad where \quad G = \sum_{s=1}^{2} c(x_s) - \sum_{s=1}^{2} c(y_s)$$

G is the *gain* of exchanging the edges in the set $X = \{x_1, x_2\}$ with the edges in the set $Y = \{y_1, y_2\}$. For example, the tour $\bar{T} \in N_2(T)$ in the previous example was constructed by exchanging the edge $x_1 = (t_3, t_4)$ and $x_2 = (t_6, t_1)$ with the edges $y_1 = (t_3, t_6)$ and $y_2 = (t_4, t_1)$. Since the cost of the tour T is $c(T) = 64$, the cost of the tour $\bar{T}$, $c(\bar{T})$, is equal

$$\begin{aligned} c(\bar{T}) &= c(T) - G \\ &= c(T) - \left[c(x_1) + c(x_2) - c(y_1) - c(y_2)\right] \\ &= 64 - \left[8 + 7 - 4 - 16\right] \\ &= 69 \end{aligned}$$

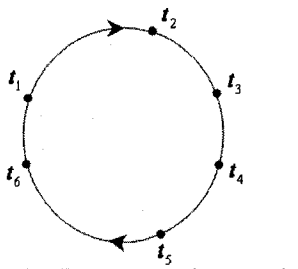
Now, the 2-opt algorithm for solving an instance of TSP is described as follows:

Step 1 Randomly choose an initial feasible tour T

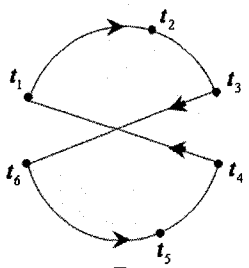
Step 2 At each step, we generate two sets of edges X and Y. Each set contains two edges such that the edges in the set X are removed from the current tour T and replaced with the edges in the set Y so that the resulting tour should have less cost.

Step 3 Stop when the sets X and Y are impossible to generate.

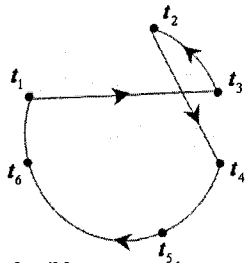
When we reach step 3, the current tour is considered as the local optimal tour. The details of Lin-Kernighan algorithm can be found in [23, 34].



A : A feasibletour $T = (t_1t_2t_3t_4t_5t_6)$



B : A feasibletour $\bar{T} = (t_1t_2t_3t_6t_5t_4) \in N_2(T)$
when $x_1 = (t_3, t_4)$, $y_1 = (t_3, t_6)$
$x_2 = (t_6, t_1)$, $y_2 = (t_4, t_1)$



C : A feasibletour $T_{13} = (t_1t_3t_2t_4t_5t_6) \in N_2(T)$
when $x_1 = (t_1, t_2)$, $y_1 = (t_1, t_3)$
$x_2 = (t_3, t_4)$, $y_2 = (t_2, t_4)$

Figure 2.21: Feasible tours of the instance of TSP of Example 2.11. The tours $\bar{T}$ and T_{13} was constructed from the tour T by exchanging the two edges in the set $X = \{x_1, x_2\}$ with the two edges in the set $Y = \{y_1, y_2\}$.

Example 2.12.

Suppose that a tour $T = (t_1t_2t_3t_4t_5t_6)$ of the instance of six cities of the previous example is selected randomly. Next, we try to find a tour $\bar{T} \in N_2(T)$ such that the cost $c(\bar{T}) < c(T)$. Since the cost of the tour $\bar{T}$ is $c(\bar{T}) = 69$, which is greater than the cost of T, $c(T) = 64$, we ignore this tour and select a different tour. Now, suppose we select a tour T_{13}, which is constructed by exchanging two edges $x_1 = (t_1, t_2)$ and $x_2 = (t_3, t_4)$ with two new edges $y_1 = (t_1, t_3)$ and $y_2 = (t_2, t_4)$, (see Figure 2.21-C). The cost of this tour is $c(T_{13}) = 51$, which is less than the cost of the current tour T. Therefore, we keep this tour and try to improve it until we reach a tour that cannot be improved. This tour is considered the local optimal tour of the problem.

Many surprising results have been found for the TSP using the Lin-kernighan (2-opt) algorithm. The results are different both in the value of the local optimal and in the time consumed for finding that value. This depends on the choice of the integer k in the k-change neighborhood [30]. For example, the 3-change has a better local optimal solution than the 2-change neighborhood. However, the 3-change neighborhood needs more time to find the local optimal solution [30].

2.3.3 Important Issues in Local Search Algorithms

Local search algorithms have some important issues that need to be specified. The first issue is the selection of the neighborhood. Recall that a neighborhood of a feasible solution $y \in F$ is a set of feasible solutions that are close in some sense to y. A feasible solution $g \in F$ can be considered as in the neighborhood of y if it can be easily obtained from y, or if they almost have the same structures.

The second issue is selecting the initial feasible solution t that the algorithm should start with. The resulting local optimal solution corresponding to a neighborhood of some initial feasible solution is, sometimes,

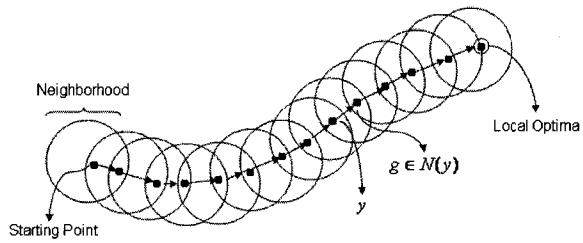


Figure 2.22: The first improvement strategy of local search algorithm where $g \in N(y_i)$, means a feasible solution g belongs to the neighborhood of y_i

near to the global optimal solution, and sometimes not. This usually depends on the selection of the neighborhood and the feasible solution that the algorithm starts with. Therefore, to examine fairly the set of local optimal solutions we need to start with a a completely random initial starting point [39].

The next important issue is specifying the strategy that searches for the neighborhood. There are several search strategies of the local search algorithm. The First-improvement and steepest-decent are the two greatest strategies for this search [6]. The first improvement strategy starts with a random feasible solution as an initial starting point. Then it selects a solution that is better than the current one and considers the new selection as the new starting point. It continues with this procedure until it reaches a solution that can no longer be improved. This solution is considered as the local optimal solution. Figure 2.22 shows a graphical depiction of the first improvement strategy [1]. On the other hand, the strategy of steepest-decent is to examine the entire neighborhood, and the neighborhood that has the greatest improvement becomes the new starting point.

For example, if we use the first improvement strategy in the 2-opt algorithm, then the search will be as follows. It uses a random tour as a starting point. Then it uses its 2-change neighborhood to select any tour in its neighborhood. If the selected tour has less cost, it is considered as a starting point. Otherwise, it selects a different one and repeats the same operation. The

algorithm continues in the same manner until it reaches a solution that can no longer be improved. This solution is called the local optimal solution.

In this chapter we have included a brief description of local search algorithms and their applications for solving important combinatorial optimization problems. In the next chapter we will describe the linear programming (LP) and integer linear programming (IP) problems which play important roles in optimization theory.

2.4 Linear Programming (LP) Problems

In this chapter we will briefly discuss linear programming and integer linear programming problems. Moreover, we will propose the simplex algorithm for solving such problems.

2.4.1 Linear Programming (LP) Problems

Linear programming problems are optimization problems where we maximize or minimize a linear function called *objective function* in n variables called *decision variables* that satisfy some linear conditions called *constraints*. That is, we need to minimize (or maximize) a linear function $f(x_1, \ldots, x_n)$ such that the decision variables $x_1, \ldots, x_n$ satisfy the following linear constraints

$$(A)_{ij} X = a_{i1}x_1 + a_{i2}x_2 + \ldots + a_{in}x_n \geq b_i.$$

The goal of the LP is to determine the values of the decision variables $x_1, \ldots, x_n$ such that the linear function $f(x_1, \ldots, x_n)$ has the largest (or smallest) value, and satisfies the conditions of the linear constraints.

The general form for a linear programming problem is called a standard form LP and is given by the following:

$$\begin{aligned} &Minimize \quad cx = c_1x_1 + \ldots + c_nx_n \ , \quad x \geq 0, \\ &Subject \ \ to \quad Ax = b \qquad (A \text{ is } m \times n \text{ matrix of integers}) \end{aligned}$$

A standard form LP is constructed as follows: minimize a linear function in non-negative n decision variables subject to a system of linear equations with constraints that are satisfied by the decision variables. The constraints model the requirements of the decision variables in the objective function that we need to optimize. For example:

$$\begin{aligned} \textit{Minimize} \quad & f(x_1, \ldots, x_n) = c_1x_1 + c_2x_2 + c_3x_3 \\ \textit{Subject to} \quad & a_{11}x_1 + a_{12}x_2 + a_{13}x_3 = b_1 \\ & a_{21}x_1 + a_{22}x_2 + a_{23}x_3 = b_2 \\ & a_{31}x_1 + a_{32}x_2 + a_{33}x_3 = b_3 \\ & x_1 \geq 0, x_2 \geq 0, x_3 \geq 0 \end{aligned}$$

In other words, the standard form of LP can be modeled in matrix form as follows:

$$\begin{aligned} \text{Minimize} \quad & c^T X = \begin{pmatrix} c_1 & c_2 & \ldots & c_n \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix} \\ \text{Subject to} \quad & Ax = b,\ X \geq 0, \end{aligned}$$

where X is the vector of unknown decision variables to be determined and should be nonnegative [14], A is the $m \times n$ matrix whose entries are the coefficients in the equation constraints, $(A)_{ij} = a_{ij}$, and c and b are vectors of known coefficients.

To understand more clearly the linear programming problems and the standard form for LP, the following example is provided.

Example 2.13. An oil refinery can buy and process two types of crude oil, light crude oil for \$11 per barrel and heavy crude oil for \$8.78 per barrel. The following quantities of gasoline, kerosene and jet fuel are produced per barrel of each type of oil.

	Gasoline	Kerosene	Jet Fuel
Light Crude Oil	.4	.25	.35
Heavy Crude Oil	.32	.4	.28

Assume that the refinery has contracted to deliver 1 million barrels of gasoline, 400,000 barrels of kerosene and 700,000 barrels of jet fuel. Excess quantities of these products can be stored at no cost.
In order to solve a linear programming problem, it is important to understand the problem clearly. There are three important steps for modeling a linear programming problem. These steps are as follows:

Step 1 Specify the decision variables:

In this example, the decision variables are the following:

x_1 = The number of barrels of light crude oil to be processed

x_2 = The number of barrels of heavy crude oil to be processed

Step 2 Specify the constraints:

The constraints in this case will be:

$$0.4x_1 + 0.32x_2 \geq 1000000,$$

since each barrel of light crude oil, x_1, and heavy crude oil, x_2, can produce 40% and 32% of gasoline respectively. Moreover, the refinery is required to deliver 1,000,000 barrels of gasoline. The second constraint is

$$0.25x_1 + 0.4x_2 \geq 400000.$$

This constraint is for the kerosene, of which the refinery should deliver 400,000 barrels. In addition, 25% of one barrel of light crude oil and 40% of one barrel of heavy crude oil must produce kerosene. Finally, the following constraint is for the jet fuel, which is formed using the same ideas as the formulas for the gasoline and the kerosene,

$$0.35x_1 + 0.28x_2 \geq 700000.$$

Therefore, the constraints of this example can be written in a system of linear inequalities as follows:

$$0.4x_1 + 0.32x_2 \geq 1000000$$
$$0.25x_1 + 0.4x_2 \geq 400000$$
$$0.35x_1 + 0.28x_2 \geq 700000$$

Step 3 Specify the objective function. This can be formulated as the cost of buying and processing one barrel of light crude and one barrel of heavy crude oil. This formula is given by

$$11x_1 + 8.78x_2.$$

Hence, the standard form of this LP is as follows:

$$\begin{aligned} \text{minimize} \quad & 11x_1 + 8.78x_2 \\ \text{subject to} \quad & 0.4x_1 + 0.32x_2 = 1000000 \\ & 0.25x_1 + 0.4x_2 = 400000 \\ & 0.35x_1 + 0.28x_2 = 700000, \\ & , x_1 \geq 0,\ x_2 \geq 0 \end{aligned}$$

which can be written in a matrix form

$$\text{minimize} \quad \begin{bmatrix} 11 & 8.78 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}$$

$$\text{subject to} \quad \begin{bmatrix} 0.4 & 0.32 \\ 0.25 & 0.4 \\ 0.35 & 0.28 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 1000000 \\ 400000 \\ 700000 \end{bmatrix}.$$

There are many applications of linear programming problems in various areas. Some examples of these areas are operations research [24, 15], business, economics and engineering problems.
Operations research is a huge field in mathematics which covers various

areas of minimization and optimization [19]. It has significant applications, for example, in data analysis, health care management, emergency and rescue operations, engineering systems design, and financial planning.
In business, examples of its applications include financial portfolios, truck routing, product mix planning, and maximizing profit in a factory that produces various products from the same material. Examples of economics are determination of shadow prices and Leontief's input-output model.

One of the significant methods with which linear programming problems can be solved is called the simplex method, which was created by George Dantzing. This method will be described in the following section.

2.4.2 Simplex Method

The simplex method was developed by George Dantzing in 1947 [25]. It was created to find an optimal solution to a linear programming problem by testing the points in a feasible set that consists of points that satisfy all the constraints for an LP problem. This set is also called the feasible region. The objective function of an LP problem has maximum value at one or more points that lie at the corner of the feasible region. Therefore, the simplex method searches for the optimal solution by moving from corner to corner until there is no more improvement of the objective function.
In this section we briefly discuss the simplex algorithm for solving the LP problem. For more details of this algorithm see [39, 14]. To illustrate the method let us state the following definitions and notations.

Definition 2.12.
Given a standard form of an LP problem

$$\begin{aligned} &min \;\; cx, \quad x \geq 0, \\ &subject\ to \;\; Ax = b \quad (A \text{ is an } n \times n \text{ matrix}) \end{aligned}$$

1. *A solution* $x = (x_1, \ldots, x_n)$ *of* $Ax = b$ *is called a* feasible solution *to the LP if* $x_i \geq 0$ *for all* $i = 1, \ldots, n$.

2. *A solution* $x = (x_1, \ldots, x_n)$ *of* $Ax = b$ *is called a* basic solution *if the columns of the matrix* A *corresponding to the nonzero entries in* x *are linearly independent.*

3. *A* basic feasible solution *(BFS) is a solution that is basic and feasible.*

Example 2.14.
Consider the following linear programming problem

$$\begin{aligned} \textit{min} \quad & x_1 + 2x_2 + 4x_3 \\ \textit{subject to} \quad & x_1 + \frac{2}{3}x_2 + x_3 \leq 1 \\ & \frac{1}{2}x_1 + x_2 + \leq 1, \\ x_1 \geq 0, \; & x_2 \geq 0, \; x_3 \geq 0 \end{aligned}$$

Here we add two slack variables x_4 *and* x_5 *to change the constraints from* " $\leq$ " *to equality* " $=$ ". *The matrix* A *will be*

$$A = \hat{A} = \begin{pmatrix} 1 & \frac{2}{3} & 1 & 1 & 0 \\ 0.5 & 1 & 0 & 0 & 1 \end{pmatrix}$$

A solution $x = (0, 0, 0, 1, 1)$ *is a basic feasible solution to LP since it satisfies the equation* $Ax = b, x \geq 0$ *and the columns of the matrix* A, $a_4 = (1, 0)$ *and* $a_5 = (0, 1)$, *corresponding to the nonzero entries in* x, $x_4 = 1$ *and* $x_5 = 1$, *are linearly independent. Whereas, the* $x = (2, 0, 0, -1, 0)$ *of* $Ax = b$ *is not a feasible solution since* $x_4 = -1 \ngeq 0$

There are two important matrices that can be composed from the constraints matrix A in a standard-from LP. The matrices are called matrix B and matrix V, which are composed as follows. Without loss of generality, suppose that the matrix A has rank m. That is, there are m linearly independent columns of A. If the matrix $A = [a_1, \ldots, a_m, a_{m+1}, \ldots, a_n]$ where $a_1, \ldots, a_m$ are the m independent columns of A, then $B = [a_1, \ldots, a_m]$ and $V = [a_{m+1}, \ldots, a_n]$. The matrix B has rank m and the columns of B form a basis for $\mathbb{R}$.
The matrix B can be composed from the $m \times n$ matrix A of rank m

as follows. Suppose that x is a BFS with $x_i > 0$ for $1 \leq i \leq p$ and $x_j = 0$ for $p < j \leq n$. Let the columns of the matrix A be the set $\{a_1, \ldots, a_p, a_{p+1}, \ldots, a_m, a_{m+1}, \ldots, a_n\}$, then the first p columns of the matrix B are $a_1, \ldots, a_p$. The other $m - p$ columns of B can be chosen from the columns $a_{p+1}, \ldots, a_n$ so that these columns together with the columns $a_1, \ldots, a_p$ form m linear independent columns. The $m-p$ columns can be relabeled as $a_{p+1}, \ldots, a_m$, and the B matrix will be $B = [a_1, \ldots, a_p, a_{p+1}, \ldots, a_m]$.

Example 2.15. *Given a standard form LP*

$$\begin{aligned} &\textit{min} && +2x_3 - x_4 \\ &\textit{subject to} && Ax = \begin{pmatrix} 1 & 0 & 4 & 2 & 2 \\ 0 & 1 & -2 & 3 & 0 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \end{pmatrix} = \begin{pmatrix} 2 \\ 0 \end{pmatrix}, \end{aligned}$$

$$x = (x_1, x_2, x_3, x_4, x_5) \geq 0.$$

A solution $x = (2, 0, 0, 0)$ is a BFS of the LP problem with $p = 1$ and the matrix A has rank 2, $m = 2$, so $m - p = 1$. Therefore, the first column of B is $a_1 = (1, 0)$. Since any of the second, third and fourth columns of A is linearly independent with a_1, we can choose any of them to be the second column of B. That is, B can be any of the following matrices

$$\begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}, \quad \begin{pmatrix} 1 & 4 \\ 0 & -2 \end{pmatrix}, \quad \textit{or} \quad \begin{pmatrix} 1 & 2 \\ 0 & 3 \end{pmatrix}$$

Note that, the fifth column a_5 is not linearly independent with a_1; therefore, it cannot be the second column of matrix B.

There are two important tableaus that the simplex method uses. These tableaus are given by the following definitions.

Definition 2.13. *Given an LP problem in a standard form*

$$\begin{aligned} &\textit{min} \;\; cx = c_1x_1 + \ldots + c_nx_n, \quad x_i \geq 0 \;\; \textit{for} \;\; i = 1, ldots, n, \\ &\textit{subject to} \;\; Ax = b \quad (A \textit{ is an } n \times n \textit{ matrix}) \end{aligned}$$

1. *A* constraint tableau *of the standard form LP is given by*

$$[A|b] = [B|V|b]$$

where B *is the* $m \times m$ *matrix* B

2. *Let a row* $c = [c_1, \ldots, c_m, c_{m+1}, \ldots, c_n]$ *represent the coefficients of the object function* cx. *The simplex tableau of the standard form LP is*

$$\left[\begin{array}{c|c} A & b \\ \hline c & 0 \end{array}\right] = \left[\begin{array}{c|c|c} B & V & b \\ \hline c_B & c_V & 0 \end{array}\right]$$

where $c_B = [c_1, \ldots, c_m]$ *and* $c_v = [c_{m+1}, \ldots, c_n]$

In the following steps we explain briefly the algorithm of the simplex method for finding the optimal solution of a linear programming problem. For more explanation about the algorithm, look at [39, 13, 14]. The method starts with an initial basic feasible solution of the standard-form LP and searches for a better basic feasible solution until the optimal BFS is reached. The steps of the simplex method are as follows:

Step 1 Convert the LP problem into standard-form LP.

Step 2 Determine an initial BFS x_B to start with.

Step 3 Compose the simplex tableau associated to x_B. That is,

$$\left[\begin{array}{c|c} A & b \\ \hline c & 0 \end{array}\right] = \left[\begin{array}{c|c|c} B & V & b \\ \hline c_B & c_V & 0 \end{array}\right]$$

Step 4 Calculate the reduced cost. In this step we apply row reduction to convert the matrix B and the row c_B to the identity I and the zero row, $(0, \ldots, 0)$, respectively. The simple tableau becomes

$$\left[\begin{array}{c|c|c} B & V & b \\ \hline c_B & c_V & 0 \end{array}\right] \rightsquigarrow \left[\begin{array}{c|c|c} I & B^{-1}V & B^{-1}b \\ \hline 0 & c_V - c_B B^{-1} V & -c_B B^{-1} b \end{array}\right] = \left[\begin{array}{c|c|c} I & W & d \\ \hline 0 & \hat{c}_V & -c_B B^{-1} b \end{array}\right]$$

Now, $x_B = B^{-1}b$ is a basic solution to the system $Ax = b$ and is a basic feasible solution if $x_B \geq 0$. The entry of $\hat{c}_V = [\hat{c}_{m+1}, \ldots, \hat{c}_n]$ is called the reduced cost relative to x_B

Step 5 Check for the optimality. The BFS x_B obtained in step 4 is optimal if and only if $c_V - c_B B^{-1} V \geq 0$ [14]. That is, if $\hat{c}_i = c_i - c_B B^{-1} a_i \geq 0$ for all $i = m+1, \ldots, n$, we have reached to the optimal solution and x_B is the optimal BFS to the LP. Otherwise, we continue the procedure. The following example is stated for the illustration.

Example 2.16. *Consider the following standard form of an LP*

$$\begin{array}{lrl} \textit{Min} & 3x_1 + 2x_2 + x_4 + x_5 & \\ \textit{subject to} & x_1 + x_5 & = 2 \\ & x_2 - 2x_3 + x_4 & = 0 \end{array}$$

If we choose the BFS $x_B = (2,0,0,0)$ as an initial BFS to start the simplex algorithm, The simplex tableau is of the form

$$\left[\begin{array}{c|c} A & b \\ \hline c & 0 \end{array}\right] = \left[\begin{array}{c|c|c} B & V & b \\ \hline c_B & c_V & 0 \end{array}\right] = \left[\begin{array}{cc|cc|c} 1 & 0 & 0 & 1 & 2 \\ 0 & 1 & -2 & 3 & 0 \\ \hline 3 & 2 & 1 & 1 & 0 \end{array}\right]$$

In step 3, we apply row reduction to convert the matrix B to the identity matrix and to make the entries of c_B=0. That is,

$$\left[\begin{array}{cc|cc|c} 1 & 0 & 0 & 1 & 2 \\ 0 & 1 & -2 & 3 & 0 \\ \hline 3 & 2 & 1 & 1 & 0 \end{array}\right] \rightsquigarrow \left[\begin{array}{cc|cc|c} 1 & 0 & 0 & 1 & 2 \\ 0 & 1 & -2 & 3 & 0 \\ \hline 0 & 0 & 5 & -8 & -6 \end{array}\right]$$

From step 4, since $\hat{c}_V = [5 \; -8] \not\geq 0, \hat{c}_4 = -8 \not\geq 0$, the solution $x_B = (2,0,0,0)$ is not the optimal BFS. Therefore, we continue the procedure of the algorithm.

Step 6 Choosing a pivot column and pivot point. In step 5, if the BFS x_B is not an optimal, there is one or more nonnegative reduced cost $\hat{c}_j$ in $\hat{c}_V$. Therefore, we improve the BFS x_B by modifying an entry in x_B that corresponds to a point called a pivot point in the matrix tableau defined by the following definition.

Definition 2.14. *In step 5, the simplex tableau is in the following form*

$$\left[\begin{array}{c|c|c} I & B^{-1}V & B^{-1}b \\ \hline 0 & c_V - c_B B^{-1} V & -c_B B^{-1} b \end{array}\right] = \left[\begin{array}{c|c|c} I & W & d \\ \hline 0 & \hat{c}_V & -c_B B^{-1} b \end{array}\right]$$

where $W = [w_j], w_j = [w_{ij}]^T, and d = [d_i]$. *A* pivot column j *in the simplex tableau is a column corresponding to the entry* $\hat{c}_j$ *that has the smallest negative entry in* $\hat{c}_V$. *A* pivot point (i, j) *is a point in the pivot column* j *where* i *corresponds to the smallest positive ratio* $\frac{d_i}{w_{ij}}$ *in* $\frac{d}{w_j}$.

Step 6 Modify the matrix B. In this step we modify the matrix B by replacing a column in B with the pivot column j and repeat the procedure starting from step 4. Note that in step 4 we apply row reduction to make the pivot point (i, j) to 1 and all other entries of the pivot column j to zero. Moreover, in step 5, if the reduced cost $\hat{c}_j < 0$ and $w_j \leq 0$, in this case we say that the objective function is not bounded below and the optimal solution of the LP does not exist.

Example 2.17. *Consider the following standard form of an LP*

$$\begin{array}{lll} Min & +2x_3 - x_4 & \\ subject\ to & x_1 - 4x_3 + x_4 + 3x_5 & = 1 \\ & x_2 + 6x_3 - x_4 & = 2 \end{array}$$

We apply the simplex method to this example starting with the initial BFS $x_B = (1, 2, 0, 0, 0)$. *The simplex tableau of this example is*

$$\left[\begin{array}{c|c} A & b \\ \hline c & 0 \end{array}\right] = \left[\begin{array}{c|c|c} B & V & b \\ \hline c_B & c_V & 0 \end{array}\right] = \left[\begin{array}{cc|ccc|c} 1 & 0 & -4 & 1 & 3 & 1 \\ 0 & 1 & 6 & -1 & 0 & 2 \\ \hline 0 & 0 & 2 & -1 & 0 & 0 \end{array}\right]$$

Since the reduced cost $\hat{c}_V = [2 - 10]$ *relative to the BFS* $x_B = (1, 2, 0, 0, 0)$ *is negative,* $\hat{c}_4 = -1 < 0$, *the BFS* x_B *is not an optimal solution. Therefore, we modify the BFS* x_B *by replacing a column in* B *with the pivot column* j. *The column is the fourth column and the pivot point is* $(1, 4)$. *Applying the simplex method to the simplex tableau we have:*

$$\left[\begin{array}{cc|ccc|c} 1 & 0 & -4 & 1 & 3 & 1 \\ 0 & 1 & 6 & -1 & 0 & 2 \\ \hline 0 & 0 & 2 & -1 & 0 & 0 \end{array}\right] \xrightarrow[(1)\ to\ (3)]{(1)\ to\ (2)} \left[\begin{array}{cc|ccc|c} 1 & 0 & -4 & 1 & 3 & 1 \\ 1 & 1 & 2 & 0 & 3 & 3 \\ \hline 1 & 0 & -2 & 0 & 3 & 1 \end{array}\right]$$

Since the reduced cost $\hat{c}_V < 0$, the BFS $x_{B_2} = (0,3,0,1,0)$ is not an optimal solution. So, we make another pivot step. The next pivot point is $(2,3)$ and the simplex tableau will be as follows:

$$\left[\begin{array}{cc|ccc|c} 1 & 0 & -4 & 1 & 3 & 1 \\ 1 & 1 & 2 & 0 & 3 & 3 \\ \hline 1 & 0 & -2 & 0 & 3 & 1 \end{array}\right] \xrightarrow[\frac{1}{2}(2)]{} \left[\begin{array}{cc|ccc|c} 1 & 0 & -4 & 1 & 3 & 1 \\ \frac{1}{2} & \frac{1}{2} & 1 & 0 & \frac{3}{2} & \frac{3}{2} \\ \hline 1 & 0 & -2 & 0 & 3 & 1 \end{array}\right] \rightarrow$$

$$\xrightarrow[\begin{array}{c}4(2)\ to\ (1)\\ 2(2)\ to\ (3)\end{array}]{} \left[\begin{array}{cc|ccc|c} 3 & 2 & 0 & 1 & 9 & 7 \\ \frac{1}{2} & \frac{1}{2} & 1 & 0 & \frac{3}{2} & \frac{3}{2} \\ \hline 2 & 1 & 0 & 0 & 6 & 4 \end{array}\right]$$

Since all reduced costs are nonnegative, we conclude that $x_{\bar{B}} = (0,0,\frac{3}{2},7,0)$ is the optimal BFS to the LP and the objective function has value -4.

In this chapter we have included a brief description of Linear programming problems and the simplex method that solves these problems. In the next chapter we will provide a new problem that can be considered as one of the combinatorial optimization problems. This problem is called the domino portrait problem, and we will apply some combinatorial optimization techniques to solve this problem. Finally, we will use mathematics properties to improve the solution of this problem.

Chapter 3

DOMINO PORTRAIT PROBLEM (DPP)

In the previous chapters we briefly discussed combinatorial optimization problems and the techniques that solve these problems. In this chapter, we provide a new problem which can be considered as a combinatorial optimization problem. This problem, called the domino portrait problem, was developed by Robert Bosch [7]. We apply some powerful combinatorial optimization algorithms to solve this problem. The algorithms we will use are a greedy algorithm and a local search algorithm. In addition, this chapter introduces a program that can construct instances of domino portraits. Also, singular value decomposition is used to improve the results.

3.1 Introduction

A domino portrait is an image which is constructed from a complete set of dominos, arranged in a matrix to create an approximate image. The artist Ken Knowlton constructed a portrait of columnist Martin Grander, of *Mathematical Scientific American*, from six complete sets of double nine dominoes in 1993. The purpose of this problem is to find a good position of dominoes so that when seen from a distance it looks like the same image. You can observe this when you look at the domino portraits of Marilyn Monroe and John Lennon (Figure 3.1), each of which was constructed from double nine sets of dominoes by Robert Bosch [7].

Robert Bosch used the integer linear programming method to construct domino portraits. He also made a domino portrait of Carl Friedrich Gauss,

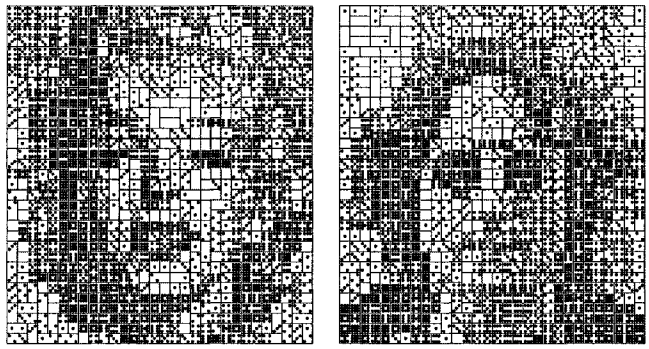


Figure 3.1: A domino portrait of Marilyn Monroe on the left and John Lennon on the right, each of which was constructed from double nine sets of dominoes by Robert Bosch [7]

which was constructed from 48 complete sets of double nine dominoes (Figure 3.2). For more detail about his products see [7].

In the following sections we discuss in detail how to create a domino portrait. This method involves using a two-dimensional wavelet transform, which is a filter in the image process that rescales the target image. In Section 4 we describe the structure of a complete set of dominoes. Section 5 provides the integral linear programming technique that Robert Bosch [7] has used to construct such portraits. In the rest of the sections we develop new techniques to create domino portraits using combinatorial optimization algorithms such as the greedy algorithm and the local search algorithm. Moreover, we use mathematics properties, like singular value decomposition and image processing, to improve our results. Finally we include the conclusions of our results and the C++ code that we use in our work.

3.2 Creating a Domino Portrait

In this section we illustrate the steps of creating a domino portrait. These steps are as follows. First, we start with a regular image. Then, we convert this image into a portable graymap format (PGM)(Figure 3.3-A). Now, the image is divided into pixels which have a grayscale value between

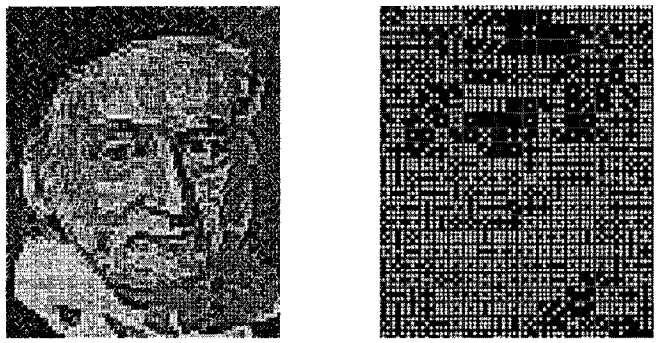


Figure 3.2: The portrait of Carl Friedrich Gauss is on the left and the dominoes portrait is on right, which was constructed from 48 complete sets of double nine dominoes by Robert Bosch.

0 and 255. These values are arranged so that each color has its own number. For example, completely black has 0 value, whereas 255 is for completely white. The values in between correspond to different shades of gray. Next, we to rescale the target image into a matrix (see Figure 3.3-B) using any standard techniques include the Photoshop and Gimp [17]. The matrix has a dimension of $10s \times 11s$ if we are using a complete set of double nine dominoes (Figure 3.6-A) or $7s \times 8s$ if we are using a complete set of double six dominoes (Figure 3.6-B), where s^2 is the number of complete sets of dominoes that we use to create a domino portrait. In addition, the two-dimensional (2D) wavelet transform, which is a filter in the image process, can also be used to rescale the target image. The description of the process using 2D wavelet transform will be explained in the next section.

After that, we compute the mean grayscale value for each square in the results matrix. That is, we convert these squares from the "0 – 255" scale into a domino scale which is 0 (completely white) to 9 (completely black), assuming that we are using white double nine dominoes. If we use black dominoes, we change that conversion. That is, 0 (completely black) to 9 (completely white). To perform this conversion, simply divide the maximum value of the grayscale, which is 255, by the largest numerated number in a domino plus one for the empty square. For example, the largest numerated

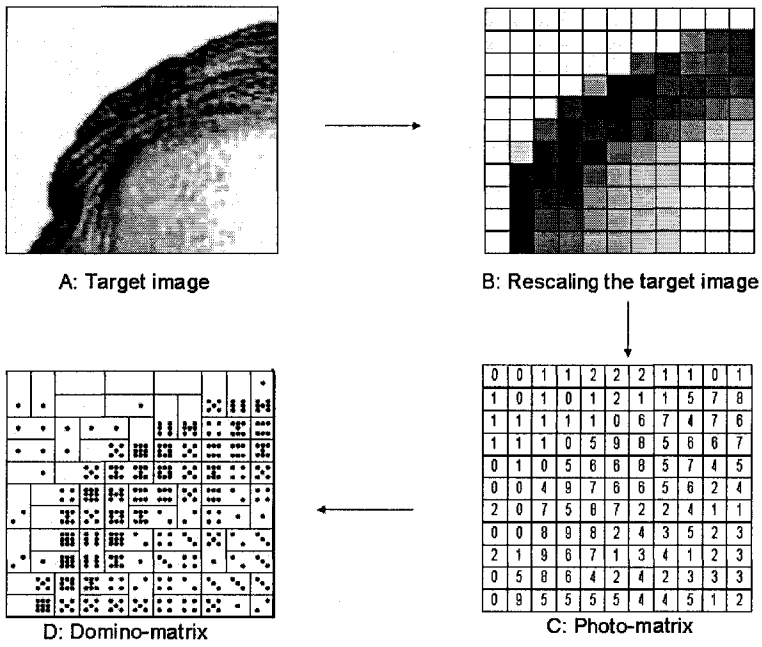


Figure 3.3: The steps of creation domino portrait

number in double nine dominoes is 9. The mean value is $255/10 = 25.5$. That is, for each square that has a grayscale value between 0 and 25.5, we give it a 0. Whereas, if it has a value between 25.5 and 50 we give it a 1 and so on.

Now, we have a matrix of $10s \times 11s$ squares that have values from 0 to 9. This matrix is called a *photo-matrix* (Figure 3.3-C). We denote the square (i, j) in this matrix by $p_{i,j}$. For instance, in Figure 3.3-C, the first square of the first row, which has value 0, is denoted as p_{00}.

Finally, we use the combinatorial optimization algorithms to convert the photo-matrix into a matrix that is composed of dominoes by finding a good place for these dominoes, such that they create an image that resembles the real image when seen from a distance. This matrix is called a *domino matrix.*

In the next section we briefly illustrate the 2D wavelet transform to rescale the target image.

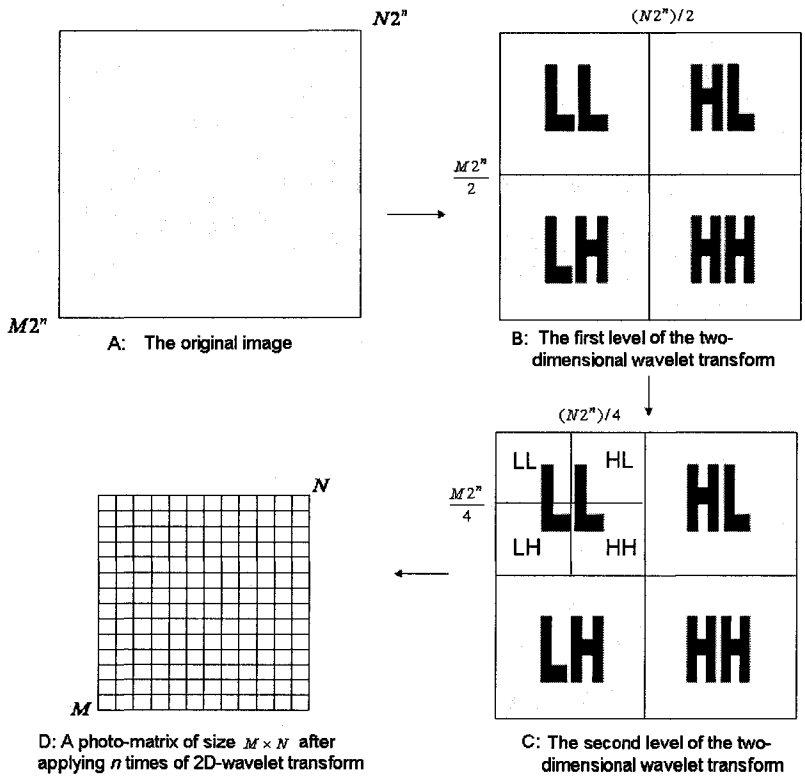


Figure 3.4: The steps of rescaling the target image, figure A, using the two-dimensional wavelet transform and composing the photo-matrix, figure D.

3.3 Two-Dimensional Wavelet Transform

In this section we will use the two-dimensional wavelet transform when we rescale the target image (Figure 3.3-A). The two-dimensional wavelet transform is a filter in the image process that has many applications. For example, it used by the Federal Bureau of Investigation (FBI) in its fingerprint identification system [6].

The two-dimensional wavelet transform rescales the target image by reducing the resolution of its columns and rows using four combinations of low and high pass filters, which are denoted as H and L respectively. First, we start with an $M2^n \times N2^n$ target image (see Figure 3.4-A) where n is an integer number that refers to the number of times the two-dimension wavelet transform is applied so that we end up with an image of size $M \times N$. Second, we divide the original image into four subimages, each of which is of

size $\frac{M2^n}{2} \times \frac{N2^n}{2}$. We apply the low pass filter to the columns and the rows of the top left image. The bottom left image can be obtained by applying the low pass filter to its columns and the high pass filter to its rows. For the top right image, we apply the high pass filter to its columns and the low pass filter to its rows. Finally, the bottom right image is obtained by applying the high pass filter to its columns and rows. These subimages are shown in Figure 3.4-B.
This is one level of the decomposition of the two-dimensional wavelet transform. The second level is the same as the first level; however, we apply it to the subimage that is in the top left where the low pass filter is applied to its columns and rows. That is, we divide this image into four subimages of size $\frac{M2^n}{4} \times \frac{N2^n}{4}$. Then, we apply the same operation to these four subimages (see Figure 3.4-C).

After applying the two-dimensional wavelet transform n times, we will have a matrix of size $M \times N$ (Figure 3.4-D) with real numbers that can be mapped to 0 to $(D-1)$, which is the photo-matrix defined in Figure 3.3-C.

Example 3.1. *Figure 3.5-A, shows an original image of an instance of Marilyn Monroe of size* 264×240, *i.e.,* $33(2^3) \times 30(2^3)$. *Assume that we need to construct a domino portrait of Marilyn Monroe of size* 33×30; *Therefore, we rescale the original image to a size of* 33×30. *That is, we apply the two-dimensional wavelet transform three times to the original image. Figure 3.5-B, shows the first reduced resolution of the original image where we applied the low pass filter to the columns and rows of its top left subimage. The size of this image is* $33(2^2) \times 30(2^2)$. *Applying the two-dimensional wavelet transform a second time to the image in Figure 3.5-B, we obtain a reduced resolution image of size* $33(2) \times 30(2)$ *which is shown in Figure 3.5-C. Finally, Figure 3.5-D, shows a reduced resolution of the image of Marilyn Monroe. The size of this image is* 33×30 *which is a matrix of real values that can be mapped to 0 to 9.*

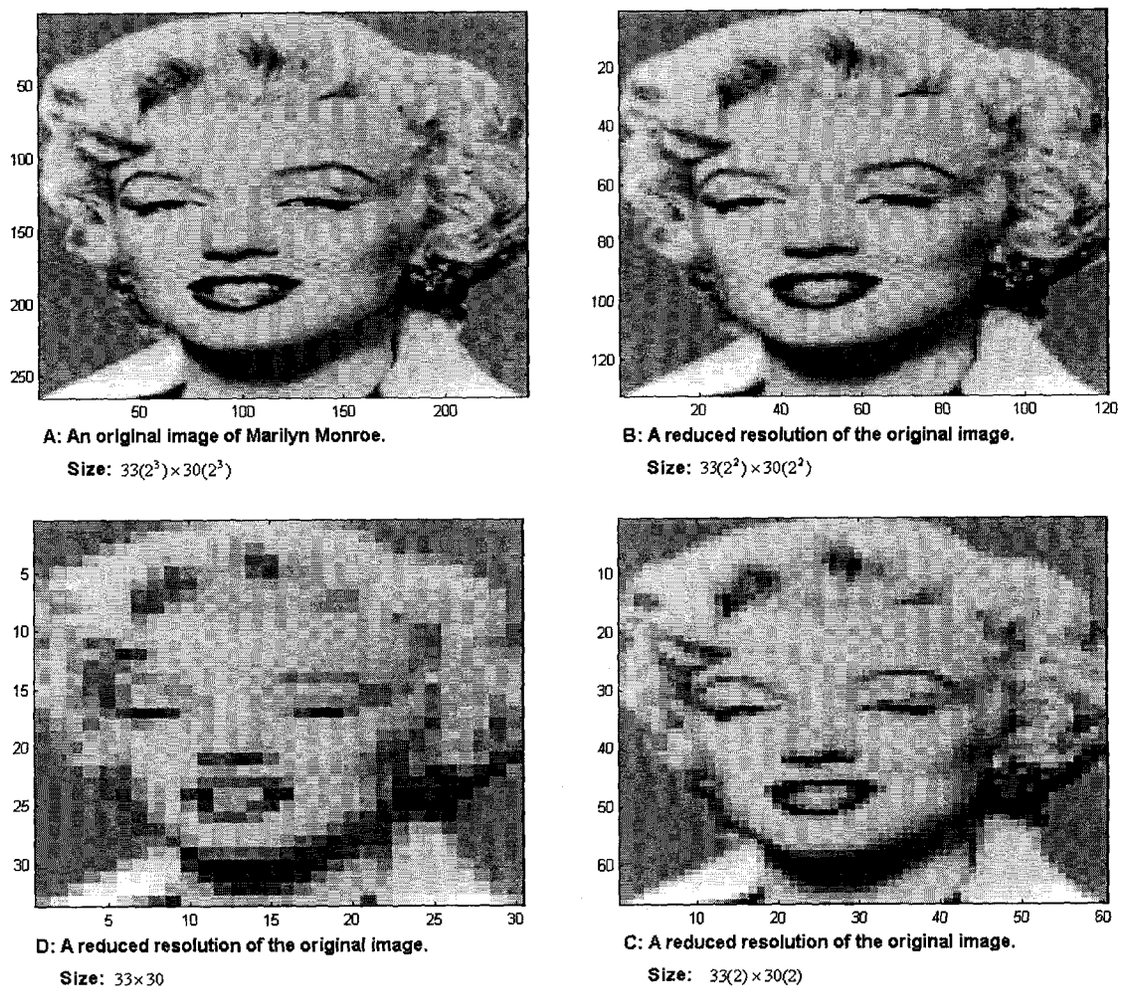


Figure 3.5: Rescaling the image of Marilyn Monroe of size 264 × 240, (A), using the two-dimensional wavelet transform. The reduced resolution of the Marilyn Monroe image of size 33 × 30 is on the (D). The detail of these images is described in Example 3.1

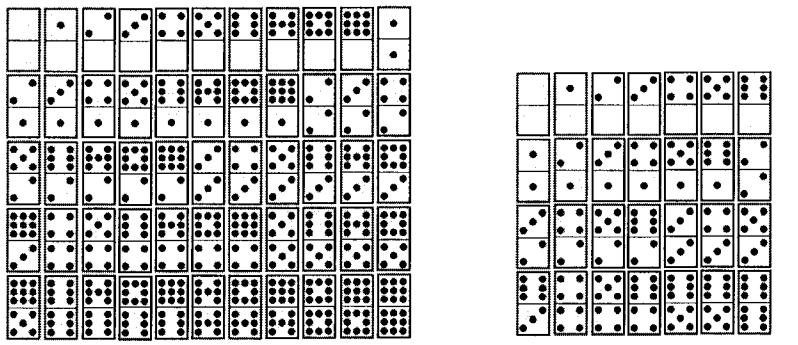


Figure 3.6: Complete sets of double nine and six dominoes

Figure 3.7: A black and white dominoes

By using the two-dimensional wavelet transform, we will get a better resolution that allows us to find a better solution for the domino portrait problem.

3.4 Domino Structure

There are several kinds of complete sets of dominoes. We are interested in two kinds of them, which are a complete set of double nine and double six dominoes (see Figure 3.6) and completely white or completely black (see Figure 3.7). A complete set of double nine dominoes is composed of 55 individual dominoes, with each domino made from two squares. These dominoes are divided into 10 double dominoes and 45 non-double dominoes. Whereas, the complete set of double six dominoes is composed of 28 dominoes which are divided into 7 double dominoes and 21 non-double (see Figure 3.6).

The next important issue in a domino structure is the orientation of each domino in the domino-matrix. That is, we denote each domino by letter

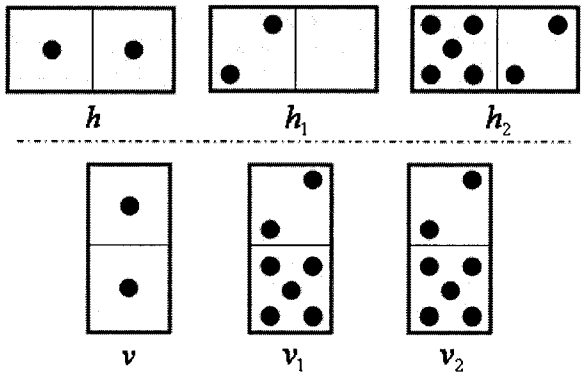


Figure 3.8: The notations of the dominoes according to their orientation in the domino-matrix

according to its orientation in the domino matrix. These orientations (Figure 3.8) are as follows: The non-double dominoes have four orientations. The dominoes that have vertical orientation with the lower number square on top are denoted by v_1. If the lower number square is on the bottom, it is denoted as v_2; If a domino has a horizontal orientation and the lower number square is on left, we denote it by h_1; and h_2 if the square that has the lower number is on the right. In the case of double dominoes, however, there are only two orientations, v for vertical and h for horizontal, as both sides are equal. All these notations are shown in Figure 3.8.

After we describe the structure of a complete set of dominoes and the steps in creating domino portraits, we can formulate the domino portrait problem as a combinatorial optimization problem. That is, we define the objective function and the constraints of the domino portrait problem that was developed by Robert Bosch [7].

3.5 Integral Linear Programming Formulation

In this section, we will convert the domino portrait problem to a combinatorial optimization problem. That is, we will define the objective function and the constraints of the domino portrait problem that were developed by R. Bosch [7].

3.5.1 Parameters

First, we define new parameters of this problem. Let D be the largest number in a complete set of dominoes plus one. For example, $D = 10$ and $D = 7$ in a complete set of double nine and double six dominoes respectively. The total number of dominoes in a complete set of double D dominoes is given by:

$$D + \binom{D}{2} = D + \frac{D(D-1)}{2}$$
$$= \frac{D^2 + D}{2}$$

A complete set of double D dominoes creates $D^2 + D$ squares of a portrait. The goal is to construct a domino portrait of dimensions $sD \times s(D+1)$ using s^2 complete sets of double $(D-1)$ dominoes, $s \geq 1$. The dimensions of a portrait is $M \times N$, where

$$M = s(D+1) \quad and \quad N = sD$$

For example, if $D = 10, s = 3$, we have a domino portrait of dimensions 30×33 , with 9 complete sets of double nine dominoes.

3.5.2 Decision Variables

The decision variables of this problem are determined according to the position and the orientation of each domino in the domino-matrix. Let $x(m, n, o, i, j)$ be the decision of placing the domino (m, n) in the position (i, j), row i and column j, in the domino-matrix with orientation o. The variable $x(m, n, o, i, j)$ takes only two possible numbers, 1 or 0. That is, if

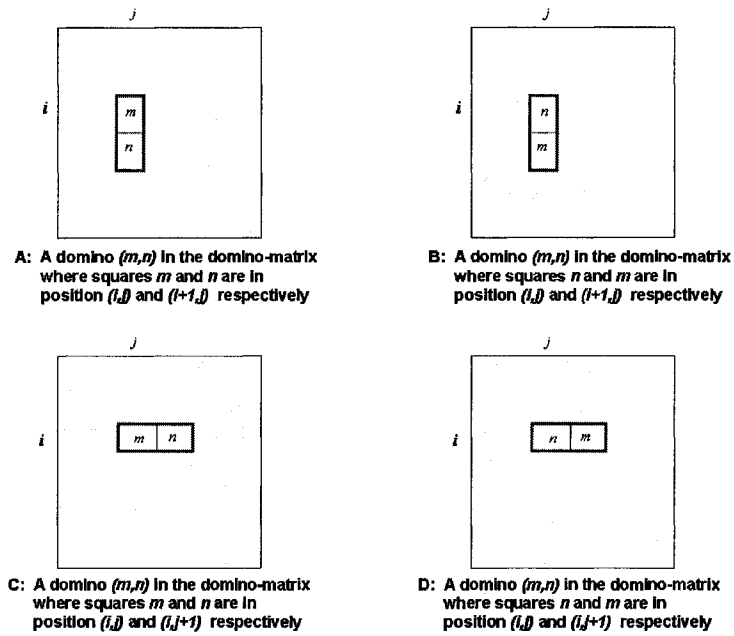


Figure 3.9: The position of the domino (m, n) in the domino-matrix according to the orientation o in the decision variable $x(m, n, o, i, j)$. The detail of this figure is described in Remark 3.5.1

the domino (m, n) is placed in position (i, j) , in the domino-matrix with orientation o , we let $x(m, n, o, i, j) = 1$ or 0 if not. For example, if the domino (1,2) is placed in position (3,4) of the domino-matrix with orientation $o = v_1$, the decision variable $x(1, 2, v_1, 3, 4) = 1$.

The position of each square of the domino (m, n) in the domino-matrix depends on the orientation o in the decision variable $x(m, n, o, i, j)$. To illustrate this, we state the following remarks:

Remark 3.5.1. Let the *square* m be the square that has value m in the domino (m, n) and the *square* n be the square that has value n. Let $m \leq n$ throughout.

1. If the orientation $o = v_1$ in the decision variable $x(m, n, v_1, i, j)$, the domino (m, n) takes the position in the domino-matrix such that the square m is in the position (i, j) and the square n in the position $(i + 1, j)$ of the domino-matrix, (see Figure 3.9-A).

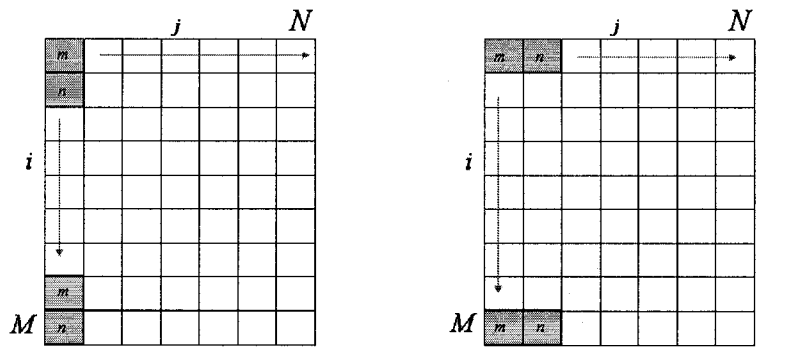


Figure 3.10: All possible places in the domino-matrix that the domino (m, n) can takes with vertical orientations (A), and horizontal orientations (B).

2. If $o = v_2$ in the decision variable $x(m, n, v_2, i, j)$, the square m will be in the position $(i+1, j)$ of the domino-matrix and the square n in the position (i, j), see Figure 3.9-B.

3. If the orientation $o = h_1$ in the decision variable $x(m, n, h_1, i, j)$, the domino (m, n) takes the position in the domino-matrix such that the square m is in the position (i, j) and the square n is in the position $(i, j + 1)$ of the domino-matrix, (see Figure 3.9-C).

4. If $o = h_2$ in the decision variable $x(m, n, h_2, i, j)$, the square m will be in the position $(i, j + 1)$ of the domino-matrix and the square n will be in the position (i, j), (see Figure 3.9-D).

5. If the orientation $o = v$ or $o = h$ $(m = n)$ is the decision variable, the domino (m, n) is placed in the domino-matrix so that the square m is in the positions (i, j) and $(i + 1, j)$ if $o = v$ and in the positions (i, j) and $(i, j + 1)$ if $o = h$.

These remarks are important when we construct the objective function of this problem.

The total number of decision variables can be computed by answering the following question: How many ways can the domino (m, n) with orientation

o take place in the domino-matrix of dimensions $M \times N$?
We classify all dominoes (m, n) into two cases: double dominoes $(m = n)$; and non-double dominoes $(m \neq n)$. Each of them has two subcases according to the orientation o. If the orientation is vertical, $o \in \{v, v_1, v_2\}$, these kinds of dominoes can take $(M-1) \times N$ possible places in the domino-matrix of dimensions $M \times N$, (see Figure 3.11-A). Whereas, the horizontal dominoes, $o \in \{h, h_1, h_2\}$, can take $M \times (N-1)$ possible places in the domino-matrix, (see Figure 3.11-B). The following table describes all possible decision variables $x(m, n, o, i, j)$. We let $m \leq n$ throughout.

$1-$ *case* $m = n, o = v$: $x(m, m, v, i, j)$,	for $m := 0$ *to* $D-1$ for $i := 1$ *to* $M-1$ for $j := 1$ *to* N
There are $D \times (M-1) \times N$ decision variables,	
$2-$ *case* $m = n, o = h$: $x(m, m, h, i, j)$,	for $m := 0$ *to* $D-1$ for $i := 1$ *to* M for $j := 1$ *to* $N-1$
There are $D \times M \times (N-1)$ decision variables,	
$3-$ *case* $m < n, o = v_1, v_2$: $x(m, n, o, i, j)$,	for $m := 0$ *to* $D-2$ for $n := m+1$ *to* $D-1$ for $i := 1$ *to* $M-1$ for $j := 1$ *to* N
There are $2 \times \binom{D}{2} \times (M-1) \times N$ decision variables	
$4-$ *case* $m < n, o = h_1, h_2$: $x(m, n, o, i, j)$,	for $m := 0$ *to* $D-2$ for $n := m+1$ *to* $D-1$ for $i := 1$ *to* M for $j := 1$ *to* $N-1$
There are $2 \times \binom{D}{2} \times M \times (N-1)$ decision variables	

Table 3.5.1: The decision variables of the domino portrait problem

Note that, a complete set of double $(D-1)$ dominoes has $\binom{D}{2}$ non-double dominoes $(m \neq n)$ and D double dominoes $(m = n)$, where D is the largest numerated number in a complete set of dominoes plus one. Case 1 and 2 include double dominoes with vertical orientation $(o = v)$ and horizontal orientation $(o = h)$ respectively. Since a complete set of dominoes

has D double dominoes, the total number of possible places that the dominoes with vertical orientation is equal $D \times (M-1) \times N$ and $D \times M \times (N-1)$ for the dominoes with horizontal orientation.

In Case 3 and 4, we have non-double dominoes with vertical orientation ($o = \{v_1, v_2\}$) and horizontal orientation ($o = \{h_1, h_2\}$) respectively. Moreover, there are $\binom{D}{2}$ non-double dominoes in a complete set of dominoes, so the vertical orientation dominoes have $2 \times \binom{D}{2} \times D \times (M-1) \times N$ possible places in the domino-matrix and the horizontal orientation dominoes have $2 \times \binom{D}{2} \times D \times M \times (N-1)$ possible places.

Consequently, the total number of decision variables is equal to the summation of all these cases. That is, the total number of the decision variables (Tdv) needed to create an $M \times N$ domino portrait using s^2 of complete sets of double $(D-1)$ dominoes is given by the following equation:

$$Tdv = D^2(2M^2 - M - N) \tag{3.1}$$

where $M = s(D+1)$ and $N = sD$. For example, If $D = 10$ and for any $s \geq 1$, we have an $M \times N$ domino-matrix where $M = 10s$ and $N = 11s$. The total number of decision variables for creating a domino portrait using s^2 of complete set of double 9 dominoes using equation 3.1 is:

$$\begin{aligned} Tdv &= (10)^2\big[2(10s)^2 - 10s - 11s\big] \\ &= 100\big[200s^2 - 21s\big] \\ &= 22000s^2 - 2100s \end{aligned}$$

If $s = 3$, the total number of decision variables is equal to 191,700 variables. In the next section we introduce the objective function of the domino portrait problem.

3.5.3 Objective Function

The objective function of the domino portrait problem is the summation of the costs or the penalties of placing each domino (m, n) in the domino-matrix. The cost of placing a domino (m, n) in domino-matrix can

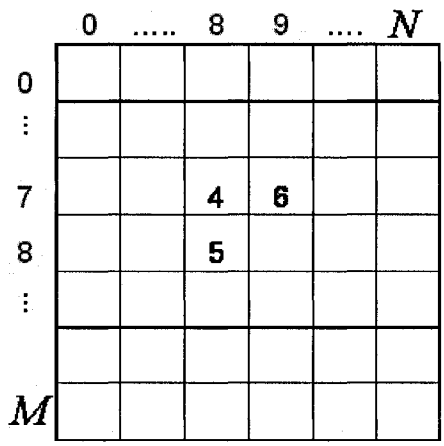


Figure 3.11: The photo-matrix $P[p_{ij}]$ of the Example 3.2.

be determined by computing the L2-norm between the domino (m, n) and each square,$g_{i,j}$, in the photo-matrix (Figure 3.3-C). This cost also depends on the orientation o of the domino (m, n) in the domino-matrix (remarks 3.5.1). Let the variable $c(m, n, o, i, j)$ be the cost of placing domino (m, n) with orientation o in position (i, j) in the domino-matrix. We use L2-norm to compute the cost $c(m, n, o, i, j)$. Using the remarks in 3.5.1, the cost of placing domino (m, n) with orientation o in position (i, j) of the domino-matrix, $c(m, n, o, i, j)$, is given by the equation:

$$
\begin{array}{lllll}
1- & If & o = v & \Rightarrow & c(m,n,v,i,j) = (m - p_{i,j})^2 + (n - p_{i+1,j})^2 \\
2- & If & o = h & \Rightarrow & c(m,n,h,i,j) = (m - p_{i,j})^2 + (n - p_{i,j+1})^2 \\
3- & If & o = v_1 & \Rightarrow & c(m,n,v_1,i,j) = (m - p_{i,j})^2 + (n - p_{i+1,j})^2 \\
4- & If & o = v_2 & \Rightarrow & c(m,n,v_2,i,j) = (n - p_{i,j})^2 + (m - p_{i+1,j})^2 \\
5- & If & o = h_1 & \Rightarrow & c(m,n,h_1,i,j) = (m - p_{i,j})^2 + (n - p_{i,j+1})^2 \\
6- & If & o = v_2 & \Rightarrow & c(m,n,h_2,i,j) = (n - p_{i,j})^2 + (m - p_{i,j+1})^2
\end{array}
\tag{3.2}
$$

where $p_{i,j}$ is the entry of row i and column j of the photo-matrix $P[p_{i,j}]$.

Example 3.2. Assume that a photo-matrix has the following positions $p_{7,8} = 4$, $p_{8,8} = 5$ and $p_{7,9} = 6$, (see Figure 3.2). The costs of placing the following dominoes are as follows:

1. $c(2, 3, v_1, 7, 8) = (2 - 4)^2 + (3 - 5)^2 = 8$
2. $c(3, 6, v_1, 7, 8) = (3 - 4)^2 + (6 - 5)^2 = 2$

3. $c(3,6,h_1,7,8) = (3-4)^2 + (6-6)^2 = 1$

From 1 and 2, the cost of placing domino (3,6) with orientation v_1 in position $(7,8)$ is 2, which is less than the cost of placing domino (2,3) with the same orientation and position. Hence, the domino (3,6) is a candidate to be placed in row 7 and column 8 in the domino-matrix. That is, we let the decision variable $x(3,6,v_1,7,8) = 1$ and $x(2,3,v_1,7,8) = 0$. Furthermore, from equation 3 and 4, we can determine the best direction of domino (3,6) in position $(7,8)$. Since the cost of placing domino (3,6) with $o = h_1$ is 1, which is less than the cost of placing it with orientation v_1, it is a better candidate to be in horizontal orientation $o = h_1$. Consequently, we let the decision variable $x(3,6,h_1,7,8) = 1$ and $x(3,6,v_1,7,8) = 0$.

The objective function of this problem is determined as follows: First for each decision variable $x(m,n,o,i,j)$, we compute its corresponding cost $c(m,n,o,i,j)$. After that, we multiply each decision variable x by its corresponding cost c. Finally, we minimize the summation of these multiplication terms. That is, the objective function of creating a domino portrait of dimensions $M \times N$ using s^2 complete set of double D dominoes is given by the following linear function:

$$Minimize \quad \sum_{x \in X} c(m,n,o,i,j)x(m,n,o,i,j), \tag{3.3}$$

where X is the set of all decision variables $x(m,n,o,i,j)$ defined in table 3.5.1.

3.5.4 Constraints

The constraints of the domino portrait problem can be classified into two types. Type-one (T_1) is for the dominoes to not overlap if they are placed in the domino-matrix. That is, if a domino (m,n) is placed in position (i,j) in the domino matrix, all other dominoes that cause overlap with this domino should be excluded from the solution. For example, if a domino $(2,3)$ is placed in position $(1,2)$ of the domino-matrix with orientation v_2 that is, the value of the decision variable $x(2,3,v_2,1,2) = 1$, all other dominoes that cause overlap with this domino should be eliminated.

That is, if the decision variable $x(2,3,v_2,1,2) = 1$, the value of all other decision variables corresponding to those dominoes that cause overlap with domino $(2,3)$ should equal zero. These variables are of the following form:

- $x(m,n,o,1,2) = 0 \quad : m \neq 2, n \neq 3, o = v_1, v_2, h_1, h_2$
- $x(m,m,o,1,2) = 0 \qquad : \forall\ m \ \ and \ \ o = v, h$
- $x(m,n,o,2,2) = 0 \quad : \forall\ m,n \ \ and \ \ o = v_1, v_2, h1, h_2$
- $x(m,m,o,2,2) = 0 \qquad : \forall\ m \ \ and \ \ o = v, h$

That is, the summation of the decision variable $x(2,3,v_2,1,2)$ and all decision variables that satisfy the above form should equal 1.

Therefore, the type-one constraints of creating a domino portrait of dimensions $M \times N$ from s^2 complete set of double $(D-1)$ dominoes is given by the following system of linear equations:

$$\sum_m x(m,m,v,i,j) + \sum_m x(m,m,v,i+1,j) + \sum_m x(m,m,h,i,j)$$
$$+ \sum_m x(m,m,v,i,j+1) + \sum_{m<n} x(m,n,v_1,i,j) + \sum_{m<n} x(m,n,v_2,i,j) +$$
$$+ \sum_{m<n} x(m,n,v_1,i+1,j) + \sum_{m<n} x(m,n,v_2,i+1,j) + \sum_{m<n} x(m,n,h_1,i,j) +$$
$$\sum_{m<n} x(m,n,h_2,i,j) + \sum_{m<n} x(m,n,h_1,i,j+1) + \sum_{m<n} x(m,n,h_2,i,j+1) = 1 \tag{3.4}$$

where

$$0 \leq m \leq D-1, \quad 1 \leq n \leq D-1,$$
$$1 \leq i \leq M, \quad i \neq M-1 \ \ if \ \ o \in \{v, v_1, v_2\}$$
$$1 \leq j \leq N, \quad j \neq N-1 \ \ if \ \ o \in \{h, h_1, h_2\}$$

and $M = s(D+1), \quad N = s(N+1)$. Since the domino-matrix is of dimensions $M \times N$, we have $M \times N$ squares to be filled with exactly one domino. That is, each square of the domino-matrix has one constraint. Therefore, type-one constraint has $M \times N$ constraints.

The type-two constraint of the domino portrait is as follows. Because the problem requires to use s^2 complete sets of domino, each domino (m,n)

must be used exactly s^2 times. That is, the type-two constraint is the following system of linear equations

$$\sum_{o,i,j} x(m,n,o,i,j) = s^2, \tag{3.5}$$

which is one constraint for each domino (m,n) in a complete set of double $(D-1)$ dominoes. For instance, the domino (2,3), case $s = 3$, since we are required to use $s^2 = 3^2$ complete sets of dominoes, the constraint is as follows:

$$\sum_{i,j} x(2,3,v_1,i,j) + \sum_{i,j} x(2,3,v_2,i,j) + \sum_{i,j} x(2,3,h_1,i,j) +$$

$$\sum_{i,j} x(2,3,h_2,i,j) = 9$$

where the indices i and j run as follows:

for $i := 1$ *to* $M-1$

for $j := 1$ *to* $N-1$

$M = s(D+1)$ and $N = sD$

Since a complete set of dominoes consists of $D + \binom{D}{2}$, dominoes, we have $\left(D + \binom{D}{2}\right)s^2$ type-two constraints. Consequently, the total number of constraints (Tc) is the summation of type-one and type-two constraints, which is defined by the following equation:

$$Tc = M \times N + \left[D + \binom{D}{2}\right]s^2 \tag{3.6}$$

For example, in the case where $D = 10$ and $s = 3$, the total number of constraints, both type-one and type-two, is 1045.

The standard form of the domino portrait problem, which was developed by Robert Bosch [7], will be introduced in the next section.

3.5.5 Standard Form LP for the Domino Portrait Problem

We conclude from the previous section that the domino portrait problem maybe cast as a linear programming problem. This is because the objective function is a linear function of the decision variables, $x(m,n,o,i,j)$

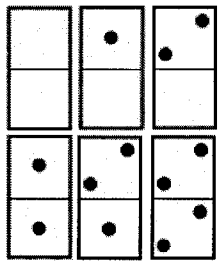


Figure 3.12: A complete set of double two dominoes

, subject to a system of linear equations being satisfied by these decision variables. Hence, this problem is a linear programming problem and the standard form of this problem is as follows:

Standard Form LP

$$\textit{Minimize} \quad \sum_{x \in X} c(m,n,o,i,j)x(m,n,o,i,j)$$

subject to

$$A_1\bar{X} = 1$$

$$A_2\bar{X} = s^2,$$

where X is the set of all decision variables defined in table 3.5.1 and $\bar{X}$ is the column vector whose entries are all elements in X. The matrices A_1 and A_2 are the coefficient matrices of equation 3.4 and 3.5 respectively. Moreover, since the decision variables take only the values 0 or 1, that is, integer values, this problem becomes an integer linear programming problem.

This problem is not simple because the decision variable and the constraints are huge. To see that, let us consider the following example where $D = 3$ and $s = 1$, which is a simple example compared with a problem where $D = 10$ and $s = 3$:

Example 3.3. Case $D = 3$, Double two dominoes, Figure 3.12, and $s = 1$, we have a photo-matrix $P[p_{ij}]$ and a domino-matrix $D[d_{i,j}]$ of dimensions $M \times N$ where $M = s(D+1) = 4$ and $N = sD = 3$, that is, these matrices are of dimensions 4×3 which are given by the following:

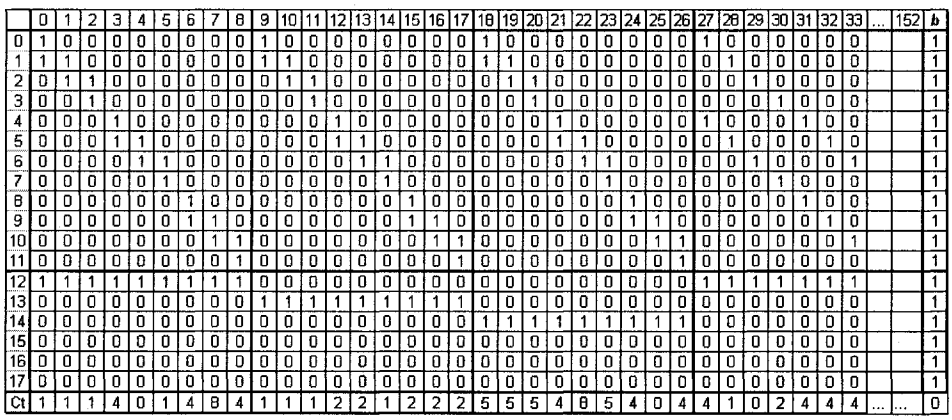


Figure 3.13: The simplex tableau of the domino portrait problem with $D = 3$ and $s = 1$. The first row represents the decision variables after they are relabeled. The last row represents the cost function. The detail of this tableau will be provided later.

0	2	0
1	0	2
0	0	2
1	1	0

Photo-Matrix

Domino-Matrix

The domino portrait of dimensions 4×3 should be created from one complete set of double two dominoes. There are 153 decision variables and 18 constraints in this problem. The first 33 columns of the simplex tableau of this problem is shown in Figure 3.13.

Example 3.4.

If $D = 10$ and $s = 9$, we have a domino portrait of dimensions 33×30. That is, the domino-matrix should be created from 9 complete sets of double nine dominoes. There are 191700 decision variables and 1045 constraints in this case. The simplex tableau is given by Figure 3.14.

As we have seen from the above tableaus, these kinds of problems are quite difficult to solve.

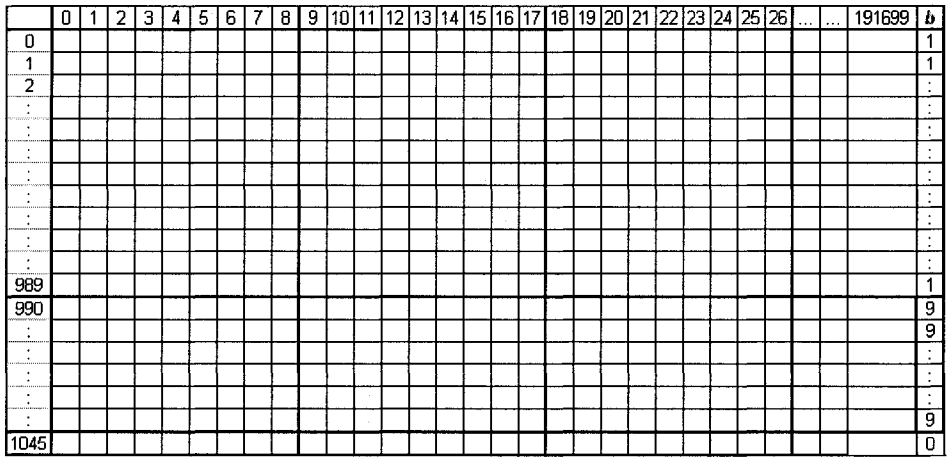


Figure 3.14: The simplex tableau of the domino portrait problem with $D = 10$ and $s = 3$.

There are some methods that might solve such kind of problem, for example, *Branch-And-Bound* method [36] or *Branch-and-cut* method [4]. These kinds of algorithms need subroutines to work with such as the simplex method or interior point algorithms which are based on many steps that are appropriate to use to solve small problems, however are inadequate for long problems similar to the current problem. Robert Bosch [7] has used software, called CPLEX (version 6.6), in his computation. In the following sections we will formulate the domino portrait problem as a combinatorial optimization problem and solve it using significant combinatorial optimization algorithms. In the next section, we use a greedy algorithm to solve the domino portrait problem.

3.6 Greedy Algorithm For Solving The Domino Portrait Problem

The domino portrait problem is a problem of finding a good rectangular arrangement of dominoes such that it forms a resemblance to an ordinary portrait when seen from a distance. An instance of the domino portrait is that we are given two integers s and D and an $M \times N$ photo-matrix (Figure 3.3-C) where $M = s(D+1)$ and $N = sD$. The problem is to create a domino portrait of dimensions $M \times N$ from s^2 complete sets of double $(D-1)$ dominoes. That is, each domino (m, n) should be placed in the domino-matrix (Figure 3.3-D) with orientation o. A feasible solution of this problem can be considered as a feasible domino arrangement. The solution space, from which a feasible solution is created, is the set of all decision variables $x(m, n, o, i, j)$ defined in tableau 3.5.1. Since these variables and therefore the solution space are finite, this problem can be formulated as a combinatorial optimization problem.

A greedy algorithm can be used to solve an instance of the domino portrait problem. A greedy algorithm (Section 2.2) is an algorithm that searches for a best (optimal) solution of a problem using several stages. At each stage, it makes its choice optimal. Some times, because of the strategy that greedy algorithms use, they reach a solution that is not feasible. This is because the constraints are not exhaustively satisfied.

In the next section we illustrate the search strategy that is used in a greedy algorithm to solve the domino portrait problem.

3.6.1 Search Strategy

The greedy strategy that is used for solving an instance of the domino portrait problem depends on several stages. Through these stages, it builds a best solution consisting of decision variables $x(m, n, o, i, j)$ which are elements of the solution space X where

$$X = \{x : x \text{ is a decision variable defined in tableau 3.5.1,}\}$$

and $x = x(m, n, o, i, j)$ means the domino (m, n) has position row i and column j in the domino-matrix with orientation o.

The search strategy of the greedy algorithm for solving this problem can be described as follows. First, we start by sorting all decision variables $x(m, n, o, i, j) \in X$ in increasing order according to their corresponding cost $c(m, n, o, i, j)$ of the objective function (section 5.5.3). Second, we choose a decision variable $x(m, n, o, i, j)$ that has the smallest corresponding cost $c(m, n, o, i, j)$ to be a part of the solution that we are trying to build. That is, the domino (m, n) is placed in position (i, j) of the domino-matrix with orientation o. Consequently, in the next stage, we ignore all decision variables that conflict with the one already chosen. After ignoring these variables, we create a new set that consists of all the remaining decision variables. This set is called the *active set*, which is the set of decision variables which do not conflict with the choice already made. After that, we choose the next decision variable from the active set that has the smallest corresponding cost and update the active set by deleting from the active set all decision variables that conflict with the chosen decision variable. We continue the same procedure until the algorithm is terminated. The algorithm is terminated if one of the following two condition is satisfied:

1. If the number of chosen decision variables reaches the total required dominoes (Trd) which is given by the following equation

$$Trd = s^2\left[D^2 + \frac{D(D-1)}{2}\right]. \tag{3.7}$$

   In this case, the algorithm has successfully found a feasible solution that satisfies the all constraints of the problem. This solution may or may not the best feasible solution.

2. If the active set has no remaining elements. In this case, the algorithm has reached a solution that is not a feasible solution. This is because constraints have not been completely satisfied.

Usually, as we mention above, the solution that is built by the greedy algorithm does not satisfy all the constraints (section 5.5.4) of this problem.

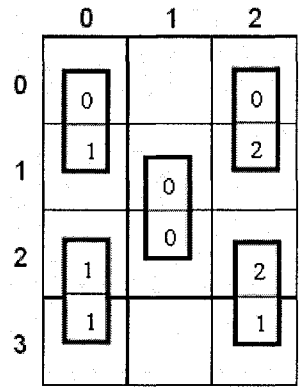


Figure 3.15: A solution, not feasible, of the instance of the DPP in Example 3.5 found using the greedy algorithm

Example 3.5.

Figure 3.15 shows a solution y of the problem in example 3.3 which is an instance of domino portrait problem with $D = 3$ and $s = 1$. The solution y which is given by

$$y = \{x_1(0,1,v_1,1,1), x_2(0,2,v_1,1,3), x_3(0,0,v,2,2),$$

$$x_4(1,1,v_1,3,1), x_5(1,2,v_2,3,3)\},$$

is found using the greedy algorithm. The algorithm built the solution y by choosing these decision variables x_1, x_2, x_3, x_4 and x_5 according to their corresponding costs 0,0,0,1 and 1 respectively, which are the smallest costs of the objective function. Since the solution y does not contain all the required dominoes, which are 6 using equation 3.7, this solution is not a feasible solution. Later, we will see how can we force the greedy algorithm to build a feasible solution that satisfies all the constraints of the problem. This can be done by analyzing mathematically the problem and finding a new strategy that can build a feasible solution to the domino portrait problem.

In the following section we will remodel the domino portrait problem so that we can use the computer language to apply the greedy algorithm to the domino portrait problem.

3.6.2 Remodeling the Domino Portrait Problem

In this section we remodel the domino portrait problem so that we can apply the greedy algorithm to solve it. This will be shown in the following steps.

1. **Labeling the Decision Variables:**
   In this step we re-label the decision variable, $x(m,n,o,i,j)$, with a new variable X_t where t is arranged from 0 to $Tdv-1$, where Tdv is the total number of decision variables defined by equation 3.1. The new decision variables are of the form:

   $$X_0, X_1, \ldots, X_{Tdv-1}$$

   For example, in Example 3.5, we label $x(0,1,v_1,1,1)$ by X_4 and $x(1,2,v_2,1,0)$ by X_{79}, where we denoted the orientations v, h, v_1, v_2, h_1 and h_2 by 0, 1, 2, 3, 4, and 5 respectively. Moreover, since the decision variables $x(m,n,o,i,j)$ take values 0 or 1, the variables X_t also take values of 0 or 1, that is, $X_t \in \{0,1\}$

2. **Constructing the simplex tableau:**
   We began by constructing a tableau, called the *simplex tableau* which is the tableau that is defined in the simplex method (section 4.2). The number of rows in the new matrix is the total number of constraints Tc defined in equation 3.6, plus one row for the objective function. Whereas, the number of columns is the total number of decision variables, Tdv defined in equation 3.7, plus one column for the matrix b, which is defined below. Therefore, the dimensions of the simplex tableau is $(Tc+1) \times (Tdv+1)$, and it is of the form:

   $$\text{simplex tableau} = \begin{pmatrix} A & b \\ d & 0 \end{pmatrix}, \qquad \text{b} = \begin{pmatrix} 1 \\ \vdots \\ 1 \\ s^2 \\ \vdots \\ s^2 \end{pmatrix}$$

where A is $Tc \times Tdv$ matrix, representing the constraints of the domino portrait problem. The row d represents the coefficients of the objective function. The entries 1's and s^2's in column d are the right hand side of the type-one and type-two constraints respectively and defined in equation 3.4 and 3.5 respectively.

3. **Creating the Boundary Variables**
   In this step we create the boundary variables, $X\_\min[t]$ and $X\_\max[t]$, for each variable X_t. These boundaries take values of 0 or 1 and bound the variable X_t. That is,

$$X\_\min[t] \leq X_t \leq X\_max[t]$$

   These boundaries determine whether or not X_t is a part of the solution of the problem. Therefore, if the value of $X\_\min[t]$ and $X\_\max[t]$ are both equal to one, then $X_t = 1$, so X_t is a solution and hence we place the corresponding domino in the domino-matrix (Figure 3.3-D). However, if $X\_\min[t]$ and $X\_\max[t]$ are both equal to zero, then $X_t = 0$ and we exclude the corresponding domino. At the beginning, we require that

$$X\_\min[t] = 0 \text{ and } X\_\max[t] = 1, \text{ for all } t \in \{0, 1, \ldots, Tdv - 1\}$$

   that is,

$$0 = X\_min[t] \leq X_t \leq X\_max[t] = 1, \quad \forall \ t \in \{0, 1, \ldots, Tdv - 1\}$$

3.6.3 The Greedy algorithm applied to the New Model of DPP

After creating the simplex tableau and the boundary variables, we apply the greedy algorithm steps, defined in Section 3.6.1, for the new model of the domino portrait problem (DPP). We create a recursive function called *choose_function*. The function builds a solution for the DPP using the steps of the greedy algorithm. The choose_function stops when one of the conditions defined in Section 3.6.1 is satisfied.
Following is the procedure the greedy algorithm uses for solving the new model of the domino portrait problem.

Step 1. Ordering the Columns and Composing the Active Set Matrix

Before we start with the procedure of choose_function, we sort the columns, X_t , in increasing order according to the smallest cost $c(m, n, o, i, j)$ of the objective function defined in equation 3.2. That is, the column i_1 is before i_2, if the corresponding value in the last row of the simplex tableau of i_1 is less than the corresponding value of i_2 . In other words, the first column we start with must have the least value in the corresponding last row in the simplex tableau. In the case where the values of the costs are equal, we sort them according to the least index. For example, in Example 3.3, case $D = 3$ and $s = 1$, the first row of the simplex tableau that is defined below shows the index t, which represents the decision variable X_t. The last row represents the cost for the corresponding column t.

column	0	1	2	3	4	5	6	7	8	$\cdots$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
Last row	1	1	1	4	0	1	2	5	8	$\cdots$

(The first and the last rows of the simplex tableau for the instance of the DPP with $D = 3$ and $s = 1$.)

That is, the last row in the simplex tableau is the corresponding cost for each column. As we see from the above tableau, column 4 has a cost of 0, which is the smallest cost, so the first column in that list is 4. Moreover, columns 0, 1, 2, and 5 have the same costs value, so we sort them as 0, 1, 2, 5. Hence, we sort all these columns as follows:

$$4, \ldots, 0, 1, 2, 5, \ldots, 6, \ldots, 3, \ldots, 7, \ldots, 8, \ldots, \text{ and so on.}$$

After we sort these columns, we save them in the first row of a new matrix, called the *active set* matrix (A-S), which is a matrix of order $Trd \times Tdv$, where Trd is the total required dominoes and Tdv is the total decision variables defined in equation 3.7 and 3.1 respectively. The active set, A-S, is the set of columns which do not conflict with the choice already made. The entries of the first row of this

matrix represent the decision variables $x(m, n, o, i, j)$ in descending order according to the smallest cost value $c(m, n, o, i, j)$. The active set $(A - S)$ matrix is of the following form:

depth/dv	0	1	...	...	Tdv-1
0					
1					
:					
Trd					

(Active set (A-S) matrix)

The columns in the A-S matrix are ordered with respect to the objective function.

Step 2. Choose_function [depth]

In this step, the choose_function starts by making a loop of several steps, which are as follows:

Step 2.1 Select Column

After sorting those columns, we select the first column in the A-S matrix, say t_c. This column is a candidate column to be part of the solution that we want to build. Therefore, we let the corresponding boundary variable, $X\_min[t_c]$, equal to one. That is,

$$1 = X\_min[t_c] \leq X_{t_c} \leq X\_max[t_c] = 1 \Rightarrow X_{t_c} = 1$$

Now, the variable t_c represents some decision variable $x(m_{t_c}, n_{t_c}, o_{t_c}, i_{t_c}, j_{t_c})$. This means, the domino (m_{t_c}, n_{t_c}) is placed in position row i_{t_c} and column j_{t_c} with orientation o_{t_c} in the domino matrix.

Step 2.2 Ignoring the conflicting columns

In the previous step we selected the candidate solution, X_t that customizes a position in the domino matrix for the corresponding domino. Therefore, we have to ignore all the columns that conflict with X_t.

Step 2.3. Saving the remaining active columns

After selecting the candidate column, t_c, and ignoring the conflicting columns, we save the remaining active columns in the next row of the A-S matrix. To illustrate this, we consider the following example:

Example 3.6.

Suppose that row i of the A-S matrix before the selecting and ignoring steps, steps 2.1 and 2.2 respectively, is

column	0	1	2	3	4	5	6	7	8	$\cdots$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
i	15	26	3	72	80	85	90	93	95	$\cdots$

Also, suppose that the columns 15, 72, 85, and 93 have the same orientation. Then, in step 2.1 we select column 15 and ignore the conflicted columns in step 2.2, which are 72, 85, and 93. Moreover, in step 2.3, we save the remaining columns which are 26, 3, 80, 90, 95,..., in row $i+1$, of the A-S matrix. Therefore, the row $i+1$ of A-S matrix will be as follows:

column	0	1	2	3	4	5	6	$\cdots$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
i	15	26	3	72	80	85	90	$\cdots$
i+1	26	3	80	90	95	85	$\cdots$	$\cdots$

Step 2.4 Calling the Choose_function $[depth + 1]$

After finding the solution X_{t_c} and saving the remaining columns of the A-S matrix, we are prepared to select the next solution. Therefore, we call the choose_function, with entry equal $depth+1$, (choose_function $[depth + 1\ ]$) where the *depth* here determines the number of solutions that we have found so far. In this case, we repeat the same procedure starting from step 2.1.

The stopping rules of the choose_function is when the one of the following conditions is satisfied:

1. If the *depth* reaches the total required dominoes Trd which is defined in equation 3.7. That is, if $depth = Trc$.

2. If the last row of the A-S matrix has no data. This means, there are no remaining active columns in the last row of the A-S matrix.

Condition one of the stopping rules means the choose_function has reached a feasible solution that satisfies all the constraints of the DPP. This solution may be the optimal or close to the optimal.
Condition two of the stopping rules means the choose_function has reached a solution that is not a feasible solution. This is, because this solution did not satisfy all the constraints of the problem.

In the next section, we will analyze the reasons why the reasons of why the greedy algorithm failed to build a feasible solution of the DPP. Moreover, we will develop a new constraint of the DPP that forces the greedy algorithm to build a feasible solution that satisfies the all constraints of the DPP.

3.6.4 Analyzing The Reasons Of getting Blockage

In this section we analyze the reasons why the greedy algorithm failed to find a feasible solution to an instance of the DPP. When we apply the greedy algorithm to DPP we found that it reaches a solution that is not a feasible solution. This is because this solution did not satisfy some constraints of the problem and it was blocked as usually happens in greedy algorithms.
Therefore, to use the greedy algorithm we have to avoid being blocked. This can be done by answering the following two questions:

Q1: Why did we get blocked?

Q2: How do we avoid being blocked?

The answer of these questions is as follows:
For the first question, we study the domino portrait graphically and use the graph theory to analyze the problem of why we get blocked. We discovered that there were three reasons for being stuck. To understand these reasons lets first state the following definitions of the graph theory.

Definition 3.1. *Two columns,* $X_{t_1} = (m_1, n_1, o_1, i_1, j_1)$ *and* $X_{t_2} = (m_2, n_2, o_2, i_2, j_2)$ *have the* ***same position*** *if the following conditions are satisfied:*

1. $i_1 = i_2$
2. $j_1 = j_2$
3. $o_1 \cup o_2 \subseteq \{0, 2, 3\}$ *or* $o_1 \cup o_2 \subseteq \{1, 4, 5\}$, *i.e.,* o_1 *and* o_2 *are both vertical or horizontal orientation.*

Example 3.7. *suppose that* $X_{t_1} = (2, 3, 2, 1, 3), X_{t_2} = (4, 5, 3, 1, 3), X_{t_3} = (0, 1, 4, 1, 3)$, *and* $X_{t_4} = (0, 0, 0, 1, 3)$, *then* X_{t_1}, X_{t_2} *and* X_{t_4} *have the same position. Whereas* X_{t_1} *and* X_{t_3} *have different positions.*

Definition 3.2.

1. *A* ***graph*** *is a pair* (V, E) *where* V *is a set whose elements are called points and* E *is a collection of two subsets of* V, *called edges.*
2. *A point* $x \in V$ *is* ***incident*** *with an edge* $e \in V$ *if* x *is in* E. *We say that a vertex* $x \in V$ *has* ***degree*** d *if* x *is incident with exactly* d *edges in* E. *A vertex of degree zero is call* ***isolated***.

Definition 3.3. *The domino matrix can be considered as a* ***graph board*** $G = (V, E)$ *where the vertices set,* V, *are the squares* (i, j) *of the domino-matrix and the edges set,* E, *are the neighbors of these squares. That is,*

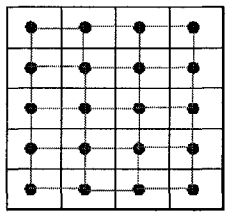


Figure 3.16: The graph board of the domino portrait

the graph board is a graph $G = (V, E)$ *where*

$$V = \{(i,j) : 1 \leq i \leq m \ \text{ and } \ 1 \leq j \leq n\}$$
$$E = \big\{(i,j) - (i,j+1) : i = 1,\ldots,m \ \text{ and } \ j = 1,\ldots,n-1,$$
$$(i,j) - (i+1,j) : i = 1,\ldots,m-1 \ \text{ and } \ j = 1,\ldots,n\big\}$$

The vertices in the center have degree 2, in the sides have degree 3 and in the middle have degree 4. This can be shown in Figure 3.16.

We recall some definitions from Chapter 2.

Definition 3.4. *A* ***walk*** *from vertex* a *to vertex* b *is a sequence of edges* $e_1, e_2, \ldots, e_k$ *such that* $e_1 = \{a, a_2\}, e_2 = \{a, a_2\}, \ldots, e_k = \{a_k, b\}$.

Definition 3.5. *A* ***path*** *from vertex* a *to vertex* b *is a walk from vertex* a *to vertex* b *whose vertices are all distinct. The length of the path is the number of edges.*

For example, in the graph in Figure 3.17, the sequence of edges e_1, e_2, e_3, e_4, e_5 and e_6 form a walk from vertex a to vertex b. However, they do not form a path from vertex a to vertex b since the vertex a_2 is used twice. On the other hand, the sequence of edges e_1 and e_6 form a path from vertex a to vertex b. Moreover, the length of this path is 2.

Definition 3.6. *A graph* $G = (V, E)$ *is* ***connected*** *if there is a path from any vertex* $a \in V$ *to any vertex* $b \in V$.

Definition 3.7. *A* ***subgraph*** *of a graph* $G = (V, E)$ *is a graph* $H = (V_1, E_1)$ *such that* $V_1 \subseteq V$ *and* $E_1 \subseteq E$.

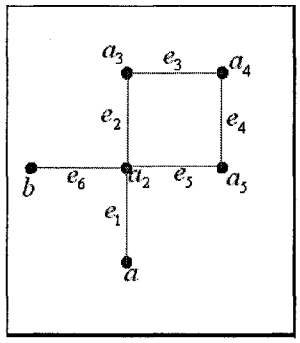


Figure 3.17: A connected weighted graph of six vertices.

Figure 3.18: The subgraph on the right is an induced subgraph, while the subgraph on the left is not.

Definition 3.8. *An **induced subgraph** of a graph $G = (V, E)$ is a graph $H = (V_1, E_1)$ such that $V_1 \subseteq V$ and $\forall x, y \in V_1 : \{x, y\} \in E_1 \iff \{x, y\} \in E$*

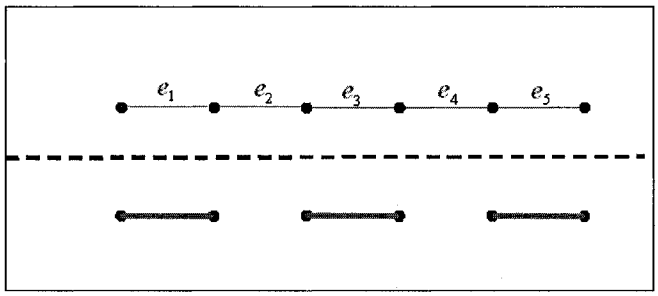
For example, Figure 3.18 shows an induced subgraph on the right and a not induced subgraph on the left.

Definition 3.9. *A **connected component** of a graph G is a subgraph of G which is connected.*

Definition 3.10. *Let $G = (V, E)$ be a graph where V is the vertex set of **even size** and E is the edge set. Assume $|V|$ is even. A **one factor** or a **matching** is a subset E_1 of the edge set E such that for any $x \in V$ there is exactly one edge $\{x, y\} \in E_1$, $y \in V$. That is, if $\{x, y\} \in E_1$, there is no $z \neq y \in V$ such that $\{x, z\} \in E_1$.*

Lemma 3.6.1. *A path has 1-factor if and only if its length is odd*

This can be shown in the following graph.



(This path has length 5 and has 1-factor)

Since the length of this path is 5, which is an odd length, this path has one factor.

In the following graphics, the gray part stands for a subgraph $D = (V_D, E_D)$ of G which has a one factor. It corresponds to the dominos which have been placed already. The white part is the subgraph $H = (V_H, E_H)$ of G, which has not yet been tiled by dominos. The vertices of the subgraphs D and H satisfy the following:

1. $V_H = V \backslash V_D$.
2. H is an induced subgraph of G
3. The size of V_D always gets bigger and V_H gets smaller.

As an example, consider the graph G_1 shown in Figure 3.19. The gray area indicates a partial domino tiling. The dominoes are indicated by edges. The white part has not yet been tiled, and has all possible edges shown. The corresponding subgraphs D_1 and H_1 of the graph G_1 are shown in figures 3.20 and 3.21 respectively. The graph G_1 can be completed by dominos since its corresponding subgraph H_1 has a one factor. Note that the bold lines in these figures refer to the edges that form one factor.

Whereas, graph G_2 in Figure 3.22 cannot be completed by dominos since its corresponding subgraph H_2, Figure 3.23 has no one factor.

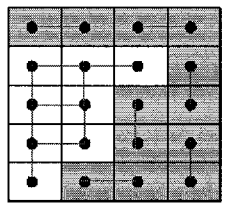


Figure 3.19: The graph G_1 that can be completed by dominoes

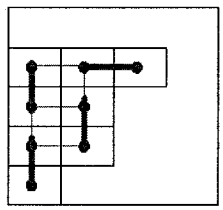


Figure 3.20: The corresponding subgraph H_1 of the graph G_1 and has one factor

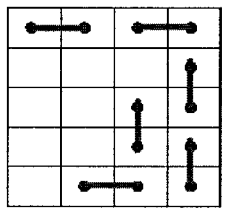


Figure 3.21: The corresponding subgraph D_1 of the graph G_1

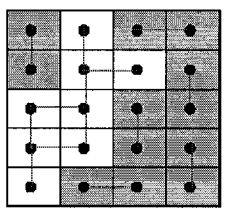


Figure 3.22: The graph G_2 cannot be completed by dominoes

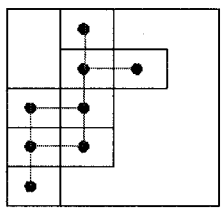


Figure 3.23: The corresponding subgraph H_1 of the graph G_2 has no one factor

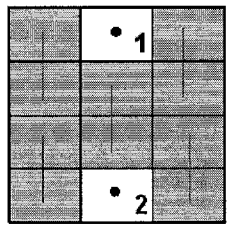


Figure 3.24: isolated square

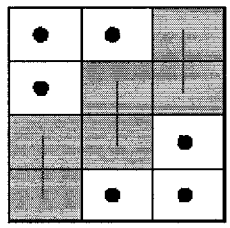


Figure 3.25: Connected components of odd size

Now we go back to those reasons of getting blocked which are the following:

1. An isolated square: an ***isolated square*** is a square in the domino-matrix that has no empty neighbor. For example, in Figure 3.24, squares 1 and 2 are isolated squares where the gray area indicates a partial domino tiling and the dominoes are indicated by edges. The white part indicates to the squares that have not yet been tiled.

2. Connected components of an odd size. (See Definition 7). For example, Figure 3.25 has two connected components of size 3, i.e., of odd size.
   Note: Case 1 is a special cases of case 2.

3. There is at least one subgraph $H \subseteq G$ that has no one factor. Note: Case 1 and 2 are special case of case 3.

From these cases we observe the following: *For the domino tiling to be complete, it is necessary and sufficient that the induced subgraph of the empty squares has 1-factor.*
That is, the greedy algorithm has reached a solution that is not a feasible

solution. This is because at the end of this solution, the A-S matrix will be out of data with no remaining active columns. Therefore, to solve this problem we need to check first, after we choose a column in step 2.1, whether that column causes the one-factor property or not and then decide either to choose or ignore it. Furthermore, we not only ignore that column but also ignore all other columns that have the same position, Definition 3.1, as that column. Otherwise, they will cause the same problem which is the one-factor property problem. That is, we need to find a quick search algorithm for one-factor property in a given graph, which will be provided in the following section.

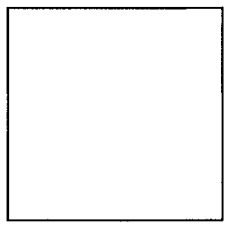
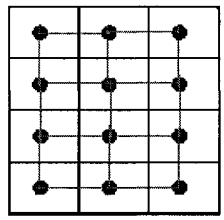
3.6.5 Greedy algorithm that avoids blockage

After we study these reasons, we found a technique to avoid being blocked. This technique is called a one-factor/matching technique, where we start with an initial one factor and then we modify it.
Before we start with the idea of this technique, lets consider the sequence of vertex complementary graphs $(D^{(i)}, H^{(i)}), i = 0, 1, \ldots, (Trd - 1)$, where Trd is defined in equation 3.7. These graphs satisfy the following:

- $D^{(0)} = (\emptyset, \emptyset)$ the empty graph and $H^{(0)} = (V_H, E_H) = (V, E) = G$ the whole graph board (see Fig 3.16).
- $D^{(i+1)}$ results from $D^{(i)}$ by addition of 2 vertices connected by one edge. $H^{(i+1)}$ results from $H^{(i)}$ by removing those two vertices and all incident edges.
- $V_H^{(i)} = V \backslash V_D^{(i)}$, ($V_H^{(i)}$ and $V_D^{(i)}$ are vertex complementary).

The idea is to only consider such domino tiles for which the new empty places subgraph $H^{(i+1)}$ have a matching (1-factor).

To illustrate this method in detail we recall Example 3.3, which is an instance of the DPP with $D = 3$ and $s = 1$. The subgraphs $D^{(0)}$ and $H^{(0)}$ are in the following figures

$D^{(0)} =$  , $H^{(0)} =$ 

One-factor search method starts with labeling the squares of the domino-matrix from 0 to $(M \times N - 1)$. The new matrix is called *matching tableau.* In this case the matching tableau will be of the form

/0	/1	/2
/3	/4	/5
/6	/7	/8
/9	/10	/11

(Matching tableau)

Before we start, let us describe the following notation. In matching tableau we give the square (i, j) a sign to tell us the location of its neighbor in the one-factor. This sign is the letter S, N, E or W according to the location of its neighbor in one factor, where S, N, E and W stand for the South, North, East and West respectively. For example, if we have chosen the square $(i, j+1)$ to be the neighbor, in one factor, of the square (i, j), then we give the square (i, j) the letter E, since its neighbor's location is on the right. Also,we give the square $(i, j+1)$ letter W, since its neighbor's location is on the left (see Figure 3.26-A). Whereas, if we have chosen the square $(i+1, j)$ to be the neighbor of the square (i, j), we assign the squares $(i+1, j)$ and (i, j) by S and N respectively (see Figure 3.26-B). We use the same manner for the other squares.

Now, let us start by choosing the following initial one factor:

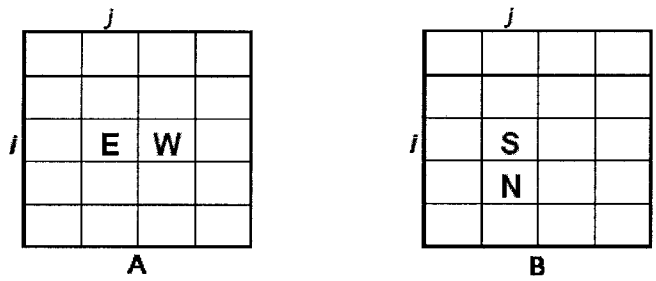
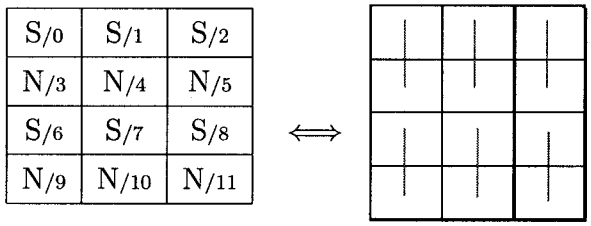
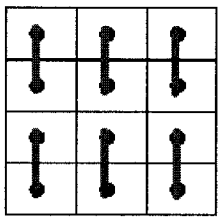


Figure 3.26: The matching tableaus where their squares are assigned by the letters S,N,E, or W according to the location of their neighbors



(Matching tableau)

We let $M^{(i)}$ to be a graph that refers to the one factor in the subgraph $H^{(i)}$. In this example, the one factor in $H^{(0)}$ is given by the following:

$M^{(0)} =$ 

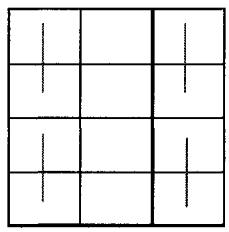
We want to be able to modify the one factor so that we can put the dominos anywhere we wish in the domino matrix. Assume that in step 3 we have chosen a column, X_{t_1}, in which we must place in position squares 4 and 7. Now, we are in the process of producing the $H^{(1)}$ and $D^{(1)}$. So, we need to change the neighbor, in the 1-factor, of the square 4 to be the square 7 and

to know whether this column will give us a one-factor or not. To see that, we have to modify the neighbor of all squares other then the squares 4, 7 and the non empty squares. The matching tableau starts with the following form:

S/0	?/1	S/2
N/3	!/4	N/5
S/6	!/7	S/8
N/9	?/10	N/11

(Matching tableau)

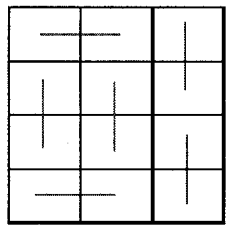
$\Longleftrightarrow$



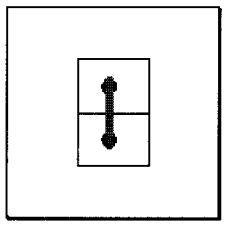
The idea here is that we find a path in $H^{(i+1)}$ connecting square 10 with square 1. Such a path necessarily has odd length and hence, by lemma 3.6.1, it can be tiled by dominoes. That is, we start with square 10 and end with square 1. If we could not reach square 1, in this case we conclude that there is no one factor. Now, square 10 can choose square 11 or 9. We choose first square 9 to be its neighbor. If we could not reach square 1, we choose square 11 instead of square 9 and continue the procedure. Consequently, we modify the sign of square 10 to be W and square 9 to be E. Square 6 can choose only square 3 to be its neighbor. That is, we give squares 6 and 3 the letters N and S respectively. Finally, square 0 has only one choice, which is square 1 and we give square 0 letter E and 1 letter W. Since we have reached square 1 and found a neighbor of each square in the matching tableau, we conclude that this column, X_{t_1}, gives a one-factor. Hence, this column is good to be part of the solution of our main problem. Therefore, we give squares 4 and 7 the letters S and N respectively. After this step, the matching tableau and figure will be as follows:

E/0	W/1	S/2
S/3	S/4	N/5
N/6	N/7	S/8
E/9	W/10	N/11

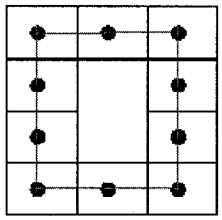
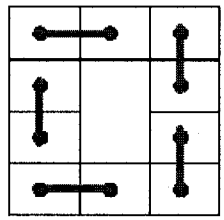
(Matching tableau)



the subgraph $D^{(1)}$ is

$D^{(1)} =$ 

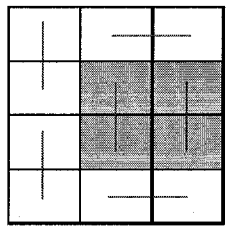
and the subgraph $H^{(1)}$ and the corresponding $M^{(1)}$ are given by the following

$H^{(1)} =$  , $M^{(1)} =$ 

Therefore, we continue the procedure of choose_function with $depth = i + 1$ to search for the next solution. That is, the greedy algorithm continues searching for the solution without being blocked.

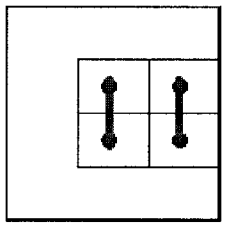
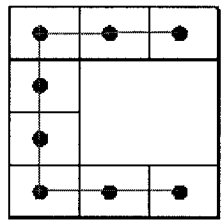
Now we want to see the other case. That is, when we could not find a one-factor in the subgraph $H^{(i+1)}$ for some i. Consider the following case of the matching tableau

S/0	E/1	W/2
N/3	T/4	T/5
S/6	T/7	T/8
N/9	E/10	W/11

$\Longleftrightarrow$ 

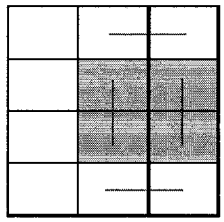
(Matching tableau)

where T means a taken (non empty) square. Assume, for some i, we want to choose a column, X_{t_i}, that has to be placed in the positions of squares 3 and 6. In this case the subgraph $D^{(i)}$ and $H^{(i)}$ are given by the following figures

$D^{(i)} =$  , $H^{(i)} =$ 

We apply the same procedure and start with the matching tableau:

?/0	E/1	W/2
!/3	T/4	T/5
!/6	T/7	T/8
?/9	E/10	W/11

$\Longleftrightarrow$ 

(Matching tableau)

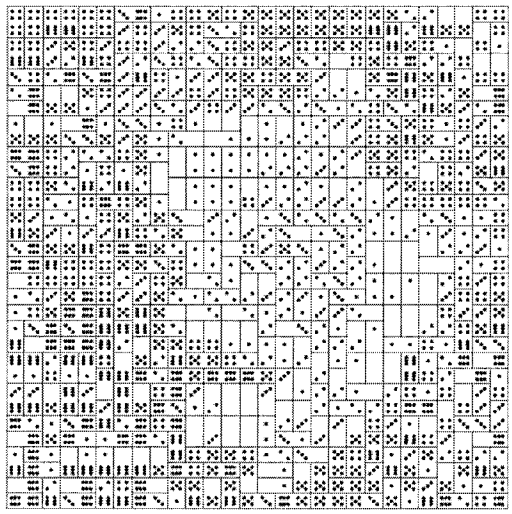


Figure 3.27: A domino portrait of Einstein which was constructed from 16 complete sets of double six dominoes using the greedy algorithm

We need to know whether this column returns a one-factor or not. We are in the process of composing the subgraph $H^{(i+1)}$. One-factor matching starts with square 9 and ends with square 0. Square 9 has only one choice, which is square 10, so we give square 9 letter E and 10 letter W. At this time, square 11 has only one choice, which is square 8; however, it is a non empty square, that is, we cannot choose it. As a result, square 11 has no neighbor and we cannot reach square 0, which is the end point. Consequently, there is no one-factor in this case and we need to ignore this column and all other columns that have the same position. That is, the choose_function continues to search for the next candidate column.

By this technique, we have our greedy algorithm which proceeds without being blocked and we find a feasible solution that satisfies all constraints of the domino portrait problem. For example, we found a feasible solution for an instance of the domino portrait of Einstein which was constructed from 16 complete sets of double six dominoes, that is, $D = 3$ and $s = 4$ (see Figure 3.27).

In this section we used the greedy algorithm strategy to solve the domino portrait problem. Moreover, we developed a one-factor technique to apply the greedy algorithm without being blocked and found a feasible solution to an instance of the domino portrait problem.
In the next section we will use the local search algorithm (chapter 3) to solve the domino portrait problem.

3.7 A Local Search Algorithm for the Domino Portrait Problem

In this section we will use local search algorithm to find the best solution of an instance of the domino portrait problem. The local search algorithm, which already was described in Section 2.3, is one of the powerful algorithms that can solve difficult combinatorial optimization problems. For example, it can be applied to solve the traveling salesman problem (Section 2.3.2). It searches for the best solution in the neighborhood of feasible solutions of an instance of an optimization problem. An instance of an optimization problem is the pair (F, c), where F is the set of feasible solutions and c is the cost function over the solutions. Solving the optimization problem means finding an $f \in F$ such that

$$c(f) \leq c(g) \quad \forall \quad g \in F$$

f is called a ***globally optimal*** solution to the given instance.
An instance of the domino portrait problem is that we are given an $sD \times s(D+1)$ photo-matrix (Figure 3.3-C) that represents a target image (Figure 3.3-B) where s and D are integers. The problem is we want to create a domino portrait consisting of s^2 complete sets of double $(D-1)$ dominoes. That is, we need to compose a domino-matrix (Figure 3.3-D) of dimensions $sD \times s(D+1)$. Each feasible domino arrangement is a feasible solution. This means, we can take the set of feasible solutions F as

$$F = \{\text{all feasible domino arrangements in the domino-matrix}\}.$$

That is,

$F = \{y :$ y is a feasible solution that satisfies the constraits of the domino

portrait defined in equations 3.4 and 3.5$\}$.

A feasible solution $y \in F$ consists of the decision variables $x(m, n, o, i, j)$. The decision variable $x(m, n, o, i, j)$ is the domino (m, n) has position row i and column j in the domino-matrix with orientation o. The cost function c

is considered as the penalty of placing these dominoes in the domino-matrix. That is, if a feasible solution $y \in F$ such that

$$y = \{x_i : \ x_i \text{ is the decision variable } x_i(m_i, n_i, o_i, i_i, j_i)\},$$

the cost of the solution y is

$$c(y) = \sum_{x_i \in y} c(x_i) \tag{3.8}$$

where $c(x_i) = c_i(m_i, n_i, o_i, i_i, j_i)$, defined in equation 3.2, is the cost of placing the domino (m_i, n_i) in the row i and column j of the domino-matrix with orientation o_i. Later in this section we will describe in detail the set of feasible solutions F of an instance of the domino portrait problem.

Recall from Section 2.3 that the neighborhood of a feasible solution $y \in F$ of an instance (F, n) is denoted as $N(y)$ and defined by

$$N(y) = \{g : g \in F, \text{ g is " close " in some sense to the point } f\}.$$

In the domino portrait problem, we can define four neighborhoods of a given feasible point $f \in F$ which are called the rotation, shift, flip and swap neighborhoods. These neighborhoods will be defined in the next section.

Usually the global optimal solution of an optimization problem is hard to find. However, sometimes we can find the best solution in the neighborhood of some feasible point $f \in F$. This solution is called a local optimal solution. The goal of the domino portrait problem is that we are given an instance (F, c) of the DPP and we want to find a feasible solution $y \in F$ such that y is the optimal or close to the optimal solution. We will use the local search algorithm to find the local optimal solution $y \in F$ such that

$$c(y) \leq c(g) \ \ \forall g \in N(y)$$

Given an instance (F, c) of the domino portrait problem and the neighborhood N, the idea of the local search algorithm for finding a local optimal solution in the neighborhood of $y \in F$ is as follows. First, we start with an arbitrary initial solution $y \in F$. We search for a better solution $g \in N(y)$

such that $c(g) \leq c(y)$. We let the solution g be the starting point and search for a better solution in the neighborhood of g. That is, we search for a solution $\bar{g} \in N(g)$ such that $c(\bar{g}) \leq c(g)$. We keep the solution $\bar{g}$ as a starting point and repeat the same procedure until we reach a solution that cannot be improved. This solution is called the locally optimal solution. The following pseudo code provides a template for local search:
Let the subroutine

$$improve(y) = \begin{cases} g & : \text{for any } g \in N(y),\ c(g) < c(y); \\ False & : \text{if no such g exists.} \end{cases}$$

begin

$y =$ an arbitrary initial solution in F;

While ($improve(y) \neq False$) do
$y = improve(y)$;

return y

end

If the local optimal is reached, the while loop will be terminated.

Before we start with the local search algorithm, we need to specify some important issues. The first issue is the selection of the neighborhood. In the domino portrait problem, we use rotating, shifting, flipping, and swapping as neighborhoods. Second, what is the feasible solution y that the algorithm starts with? Here we should start with a completely random initial starting point. The details of choosing the starting point randomly and the neighborhoods of the DPP will be in the next sections. Finally, we should specify the search strategy for the local optima. First-improvement and steepest-descent are the two greatest strategies for this search [6]. In the first-improvement strategy (see Figure 2.22), the first better solution is found and considered as the new starting point. On the other hand, the strategy of steepest-descent is to examine the entire neighborhood and the neighbor that has greatest improvement becomes the new starting point.

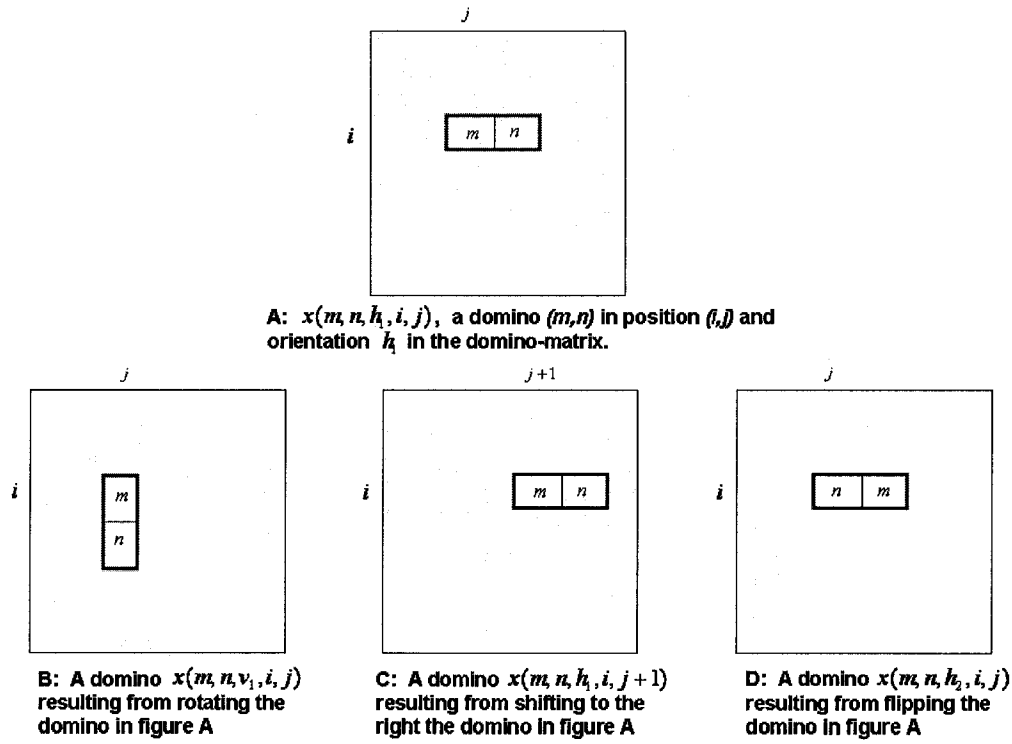


Figure 3.28: A domino (m, n) in position row i and column j of the domino-matrix with orientation $o = h_1$ shown in figure A. The rotation, shift, and flip neighborhood of the domino in figure A, $x(m, n, h_1, i, j)$, are shown in figure B,C, and D respectively.

Before applying the local search algorithm to the domino portrait problem, lets describe the neighborhoods and the set of feasible solution F of the DPP. This will be in the following section.

3.7.1 The Neighborhoods of the DPP

In the domino portrait problem, we can define four neighborhoods of a given feasible point $f \in F$, which are called the rotation, shift, flip, and swap neighborhoods. These neighborhoods are defined in the following definition:

Definition 3.11. *Assuming that we have a domino-matrix of dimensions $M \times N$ and we let the $h_0 = h$ and $v_0 = v$ overall,*

1. **Rotation neighborhood:** *Which is defined as*

   $N_r(f) = \{g : g \in F$ *and* g *can be obtained from* f *by rotating a*

domino in f 90° counterclockwise, if it is in vertical orientation, and clockwise if it is in horizontal orientation.}

The rotation depends on the orientation o. That is, the rotation of a domino represented by $x(m,n,v_t,i,j)$ is $x(m,n,h_t,i,j)$ for $t \in \{0,1,2\}$, (see Figure 3.29-B); and vice versa, the rotation of the domino $x(m,n,h_t,i,j)$ is $x(m,n,v_t,i,j)$, (see Figure 3.28-B.)

2. **Shift neighborhood:** *This neighborhood is defined as*

   $N_t(f) = \{g : g \in F$ and g can be obtained from f by shifting a domino in f one step down if it is in vertical orientation, and one step to the right if it is in horizontal orientation.$\}$

   That is, the vertical shift of the domino $x(m,n,o,i,j)$ is $x(m,n,o,i+1,j)$ for $1 \leq i \leq M-2$ or $x(m,n,o,i-1,j)$ for $i = M-1$ for all j and $o \in \{v_0,v_1,v_2\}$ (see Figure 3.29-C.) The horizontal shift of a domino $x(m,n,o,i,j)$ is $x(m,n,o,i,j+1)$ for $1 \leq j \leq N-2$ or $x(m,n,o,i,j-1)$ for $j = N-1$ for all i and $o \in \{h_0,h_1,h_2\}$ (see Figure 3.28-C.)

3. **Flip neighborhood:**

   $N_f(f) = \{g : g \in F$ and g can be obtained from f by flipping a domino in f.$\}$

   This means, the flip of the domino $x(m,n,v_1,i,j)$ is $x(m,n,v_2,i,j)$ (see Figure 3.29-D) whereas, the flip of the domino $x(m,n,h_1,i,j)$ is $x(m,n,h_2,i,j)$, (see Figure 3.28-D.) Note that, in vertical or horizontal flip we change the orientation o from v_2 to v_1 and h_2 to h_1 respectively. Moreover, if the orientation $o = v_0$ or h_0, that is , $m = n$, we do not apply the flip neighborhood, otherwise, it would be the same domino.

4. **Swap neighborhood:** *Which is defined as,*

   $N_s(f) = \{g : g \in F$ and g can be obtained from f by swapping two dominoes in f.$\}$

   The swap of the two dominoes $x_1(m_1,n_1,o_{1_{t_1}},i_1,j_1)$ and $x_2(m_2,n_2,o_{2_{t_2}},i_2,j_2)$

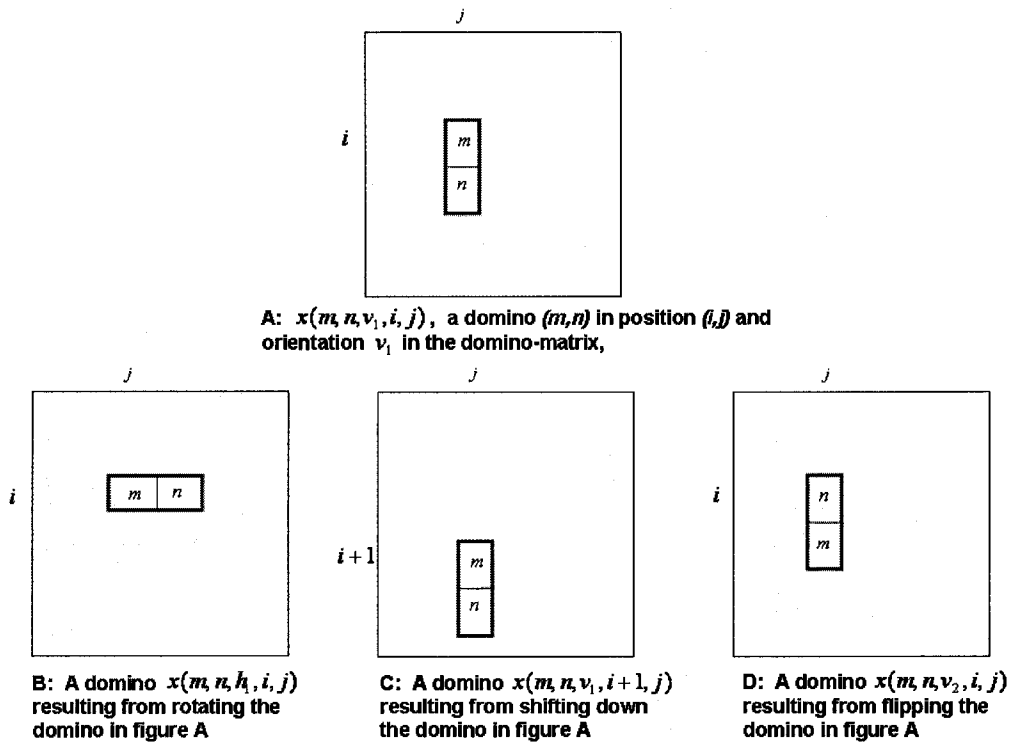


Figure 3.29: A domino (m, n) in position row i and column j of the domino-matrix with orientation $o = v_1$ shown in figure A. The rotation, shift, and flip neighborhood of the domino in figure A, $x(m, n, v_1, i, j)$, are shown in figure B,C, and D respectively.

is as follows:

$$x_1(m_1, n_1, o_{1_{t_1}}, i_1, j_1) \rightsquigarrow x_1(m_2, n_2, o_{1_{t_2}}, i_1, j_1),$$

and

$$x_2(m_2, n_2, o_{2_{t_2}}, i_2, j_2) \rightsquigarrow x_2(m_1, n_1, o_{2_{t_1}}, i_2, j_2),$$

where the o_1 and o_2 take the vertical or horizontal orientation. For example, if we swap the domino $x_1(m_1, n_1, h_1, i_1, j_1)$ with the domino $x_2(m_2, n_2, v_2, i_2, j_2)$, these dominoes will be $x_1(m_2, n_2, h_2, i_1, j_1)$ and $x_2(m_1, n_1, v_1, i_2, j_2)$ (see Figure 3.30.)

The rotation and shift neighborhoods cause changes to the positions of two or more dominoes. This is because the new domino in the instance g is making an overlapped problem. In other words, two dominoes are overlapping.

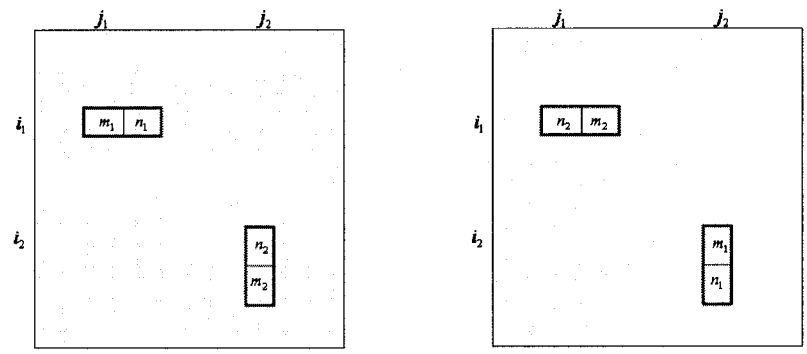


Figure 3.30: The two dominoes $x(m_1, n_1, h_1, i_1, j_1)$ and $x(m_2, n_2, v_2, i_2, j_2)$ are on the left and the swapping of these dominoes is on the right.

Therefore, we need to repair this problem by using the one-factor/matching algorithm which was described in Section 3.6.5. This algorithm will solve the problems caused by these neighborhoods.

3.7.2 Constructing the Set of Feasible Solutions F of the DPP

According to the constraints of the domino portrait problem, we construct the set of feasible solutions F. The constructing strategy of this set depends on the one-factor property (see Definition 3.10) in the domino matrix and all possible flipping and swapping of the dominoes in this matrix. Recall that the domino matrix can be represented as a graph board (see Figure 3.16. The graph board has several variations of the one-factor property. Each one-factor can be considered as a feasible solution. Moreover, each possible flip and swap of the dominoes in this one-factor can also be a feasible solution.

Let $F = \{f_0, f_1, ...\}$ be the set of the feasible solutions of an instance of the DPP. The construction of this set is described in the following steps:

Step 1 We start by letting f_0 be a feasible solution so that each domino in the domino matrix has a vertical orientation. For example, in Example 3.3, which is an instance of the DPP when $D = 3$ and $s = 1$, the feasible solution f_0 is shown in Figure 3.31.

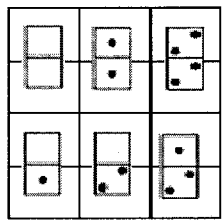


Figure 3.31: f_0 case $D = 3, s = 1$

Step 2 A set of feasible solutions $F_{i_1} \subseteq F$ is obtained from f_0 by flipping every non-double domino. That is, for each single flip of a domino in f_0 creates an element $f_i \in F_{i_1}$.

Step 3 From each element in F_{i_1}, we can obtain a set $F_{i_2} \subseteq F$ by swapping two non-similar dominoes.

At this time we have considered all possible solutions derived from the one-factor in the feasible solution f_0. Next, we obtain a new one-factor in the domino matrix by using the rotation neighborhood or shift neighborhood.

Step 4 In this step we rotate a domino in the feasible solution f_0. This rotation obtains a new one-factor different from the one in f_0. The new one-factor can be considered as a feasible solution.

Step 5 The new subsets can be obtained from the feasible solution constructed in step 4 by repeating steps 2 and 3.

Therefore, every time we rotate or shift a domino in the feasible solution obtained from step 1, a new one factor is generated. Hence, new subsets of F follow by repeating steps 2 and 3 to the generated one-factor.

Consequently, the set of feasible solutions F is the union of all subsets obtained from the previous steps.

From the way that we construct the set of feasible solutions F, we conclude that the probability of choosing any feasible solution in F is equally

likely. For example, starting with the feasible solution f_0, we can choose a random feasible solution $f \in F$ by choosing randomly the numbers k_1, k_2, k_3 and k_4. These values refer to the number of times that we do a rotation, shift, flip and swap neighborhood respectively. Moreover, each time we do the rotation, shift, flip, and swap neighborhood, we also choose randomly which domino this neighborhood applies to.

Now, we describe the steps of the local search algorithm for solving the domino portrait problem. Given an instance (F, c) of the domino portrait problem

Step 1 Select the starting point $y \in F$: We start with a completely random feasible solution $y \in F$. As mentioned above, starting with the solution f_0 defined in Figure 3.31, the solution y can be chosen randomly by selecting randomly the number of times that we do the rotation, shift, flip, and swap neighborhoods, defined in Definition 3.11, to the feasible solution f_0.

Step 2 Compute the cost of the solution y. Assume that the solution $y = \{x_1, \ldots, x_{Trd}\}$ where x_i is a decision variable $x_i(m, n, o, i, j)$ and Trd is the total required dominoes defined in 3.7. The cost of the solution y, $c(y)$, is given by the equation 3.8, which is

$$c(y) = \sum_{x_i \in y} c(x_i)$$

where $c(x_i) = c_i(m_i, n_i, o_i, i_i, j_i)$, defined in equation 3.2.

Step 3 Search for a feasible solution $g \in F$ such that it is in the neighborhood of y, that is, $g \in N(y)$. The solution g is determined using the rotation, shift, flip or swap neighborhood defined in Definition 3.11.

Step 4 Compute the cost of the solution g, $c(g)$, found in step 3. To compute the cost of the solution g, we will use the following: Assume that the solution g is obtained from solution y by replacing the decision

variables $x_1, \ldots, x_r$ with $\tilde{x_1}, \ldots, \tilde{x_r}$. The cost of g can be computed easily without using the formula in equation 3.8, by using the equation

$$c(g) = c(y) + \sum_{\tau=1}^{r} \left[c(\tilde{x}_\tau) - c(x_\tau)\right]$$

Step 5 Decide whether we choose or ignore the solution g. The solution $g \in N(y)$ must be chosen so that the it is better than the solution y. That is, we choose the solution g if

$$c(g) < c(c)$$

Otherwise, we ignore it. In the case of choosing the solution g, the algorithm considers the solution g as the starting point and continues searching for a better solution by repeating the same procedure starting from step 3. If the solution g is not better than the solution y, the algorithm searches for a different solution $\tilde{g} \in N(y)$ such that $\tilde{g} < y$.

Step 6 Stop until there is no more improvement of the current solution. The algorithm is terminated when all possible neighbors of the current solution is examen and no more improvement is found. In this case, the current solution is called the locally optimal solution.

Applying the local search algorithm to the domino portrait problem, we can construct a domino portrait of dimensions $sD \times s(D+1)$ from s^2 complete sets of double $D-1$ dominoes. For instance, we construct two domino portraits, Marilyn Monroe and George W Bosh, the president of the USA, each of which were constructed from 3^2 complete sets of double nine dominoes (Figure 3.32).

For example, when we apply the local search algorithm for the instance of Marilyn Monroe, we start with a completely random initial feasible solution y_0 as a starting point, (see Figure 3.33-A). Figure 3.33-B shows the first of 500 improvement solutions. Since this solution has less cost, we keep it

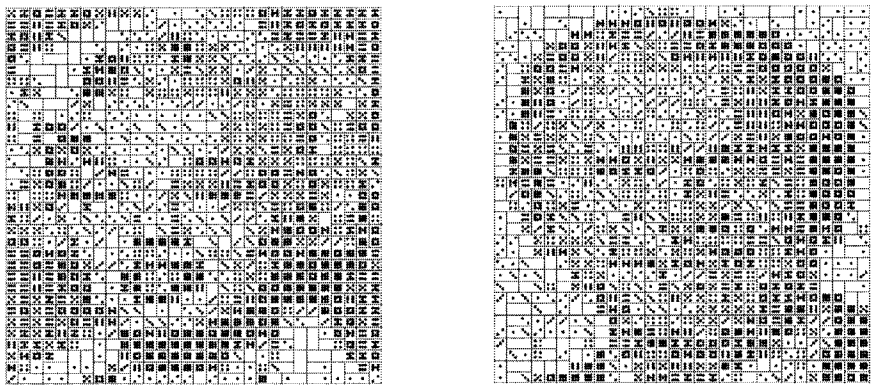
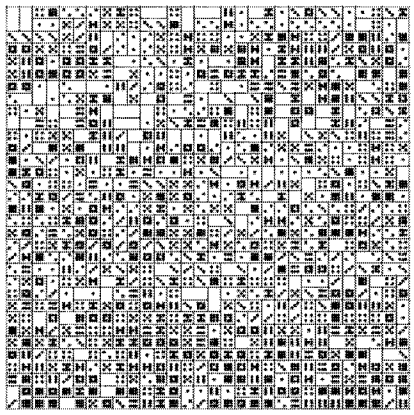


Figure 3.32: A domino portrait of Marilyn Monroe on the left and George W. Bush on the right, each of which were constructed from 3^2 complete sets of double nine dominoes using the local search algorithm

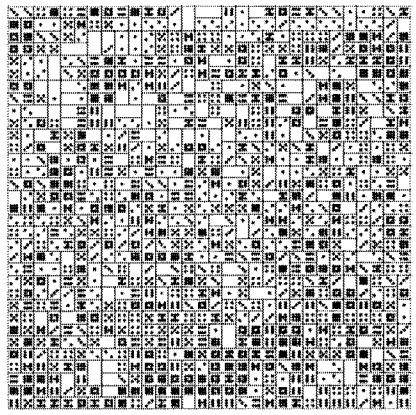
and search for a better solution that is in the neighborhood of the previous solution. Some of these improvements that we have found are shown in Figures 3.33-C-E. Finally, we have reached a solution that cannot be improved, which is shown in Figure 3.33-F.

The local optimal solution can be reached when we first search for all possible improvements of the rotation-neighborhood and then all possible improvements of the shift-neighborhood. This result is obtained when we run the local search algorithm 300 times of the instance of Marilyn Monroe with different random starting points. The table in Figure 3.34 shows the different outcomes of the cost function (the second column) and the number of times that these outcomes are repeated (the third column).

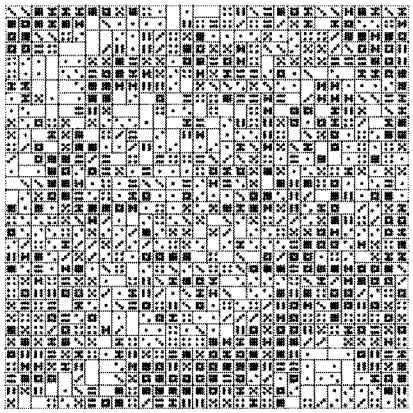
This table shows 30 different outcomes. Comparing between these outcomes we found that they are approximately close to each others. The maximum different between two outcomes is equal 68. The largest cost function is equal 4865 which appears one time; whereas, the smallest (the best) cost function is equal 4797 and appears one time. The most repeated outcome has value 4825 which appears 28 times.



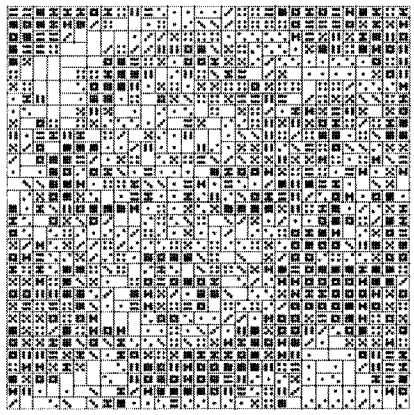
A: An initial random starting point y_0



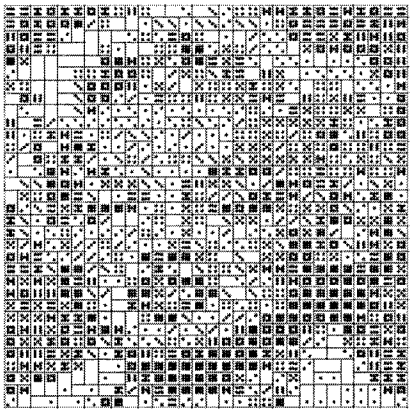
B: After 500 improvements



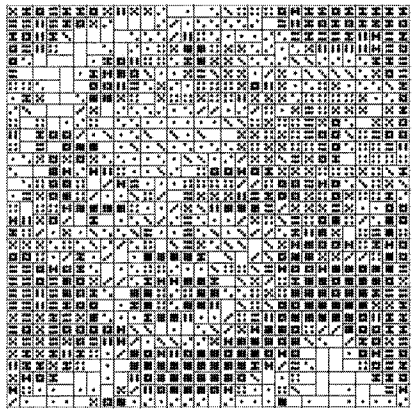
C: After 1000 improvements



D: After 1500 improvements



E: After 2000 improvements



F: The local optimal solution

Figure 3.33: A sequence of feasible solutions of the instance of Marilyn Monroe that was found using the local search algorithm. Figure A shows the starting point y_0. Figures B through E show the first 500 to 2000 improvement solutions. Figure F shows the local optimal solution.

	Cost Function	Repetitions
1	4865	1
2	4857	2
3	4853	2
4	4849	3
5	4847	2
6	4845	1
7	4843	2
8	4841	2
9	4839	7
10	4837	2
11	4835	14
12	4833	11
13	4831	17
14	4829	17
15	4827	16

	Cost Function	Repetitions
16	4825	28
17	4823	21
18	4821	20
19	4819	25
20	4817	21
21	4815	14
22	4813	20
23	4811	16
24	4809	10
25	4807	8
26	4805	6
27	4803	5
28	4801	3
29	4799	2
30	4797	1

Figure 3.34: 30 different outcomes of applying the local search algorithm 300 times to the instance of Marilyn Monroe with different random starting points.

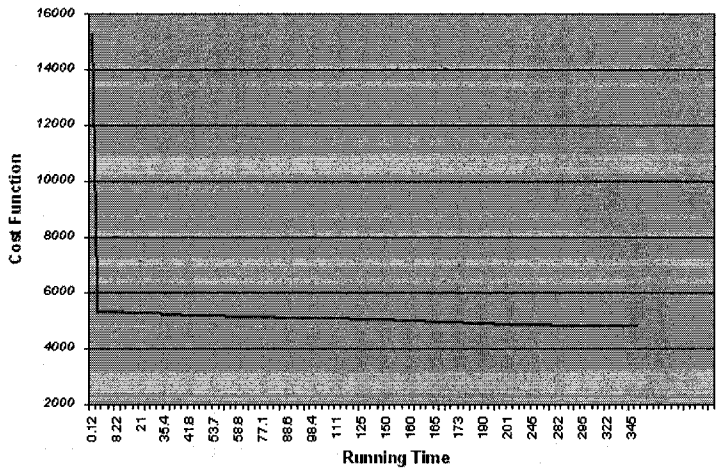


Figure 3.35: A plot of the decreasing cost function of the instance of Marilyn Monroe when the local search algorithm is applied.

The average outcomes is equal 4822. Since all outcomes are close to this average, we conclude that these outcomes are approximately equal and the maximum error is equal

$$\begin{aligned}\text{the maximum error} &= \text{the largest outcome} - \text{the average outcome}\\ &= 4865 - 4822\\ &= 43.\end{aligned}$$

Therefore, to search for the local optimal solution of an instance of the DPP using the local search algorithm we first search for all possible improvements of the rotation-neighborhood followed by all possible improvements of the flipping and swapping neighborhoods; and then search for all possible improvements of the shift-neighborhood followed by all possible improvements of the flipping and swapping neighborhoods.

For example, applying the local search algorithm for the instance of Marilyn Monroe (see Figure 3.33) with random starting point, we have the following. The cost function of the starting point is equal 15,285. This solution is improved step by step using first the all possible shift-neighborhood and then all possible rotation-neighborhood. This improvement of this solution can be reached faster in the beginning of the search and then it becomes more difficult. For instance, the value of the cost function decreases to 5433 in the first second and then it decrease slowly until it reaches the local optimal solution. The local optimal solution of this instance is equal 4809 with running time equal 411 seconds on an 800 M_Z Pentium 3 PC (see Figure 3.35).

Figure 3.36 shows the decreasing cost function, with running times, for the various runs of the instance of Marilyn Monroe.

Although, some of the local optimal solutions of the DPP are repeated more than once, the domino constructions of the portraits corresponding to these solutions are different. That is, two local optimal solutions having the same value of the cost function need not have same domino constructions in their portraits. For example, Figure 3.37- A and B, shows two local

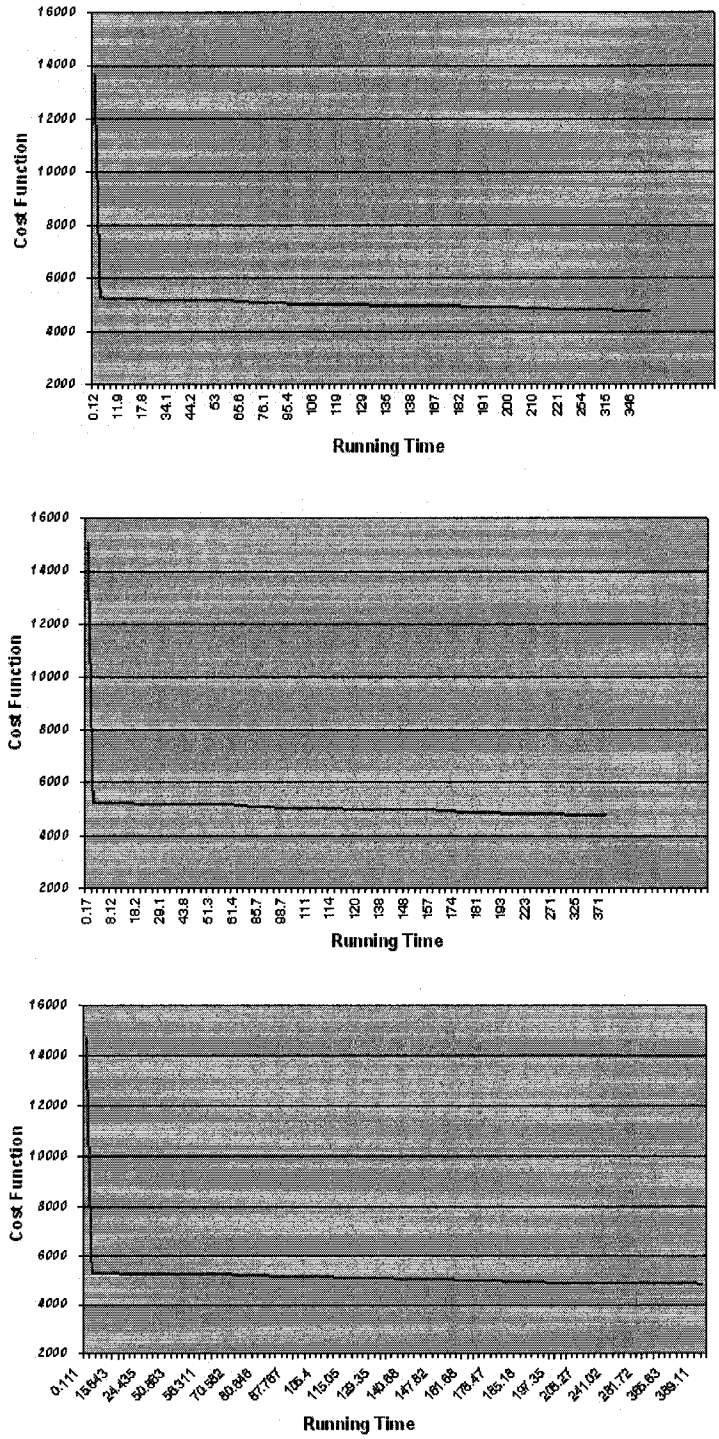


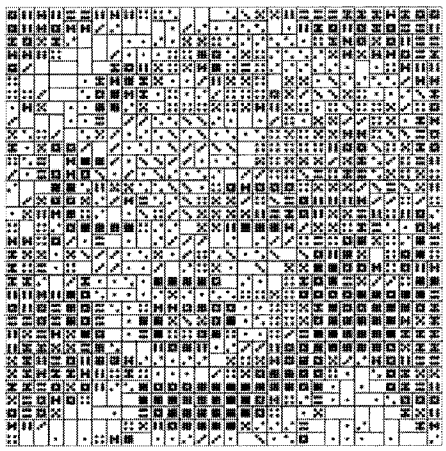
Figure 3.36: Plots of the decreasing cost function for the various runs of the instance of Marilyn Monroe using the local search algorithm.

optimal solutions of the instance of Marilyn Monroe, each of which were constructed from nine complete sets of double nine dominoes using the local search algorithm. These solutions have the same value of the cost function which is equal 4833. These images have two different domino constructions. This can be shown by determining the intersection of all dominoes in both images. That is, we consider only the dominoes that have the same position in both images. These dominoes are shown in Figure 3.37-C. Therefore, the domino portrait problem has several local optimal solutions.

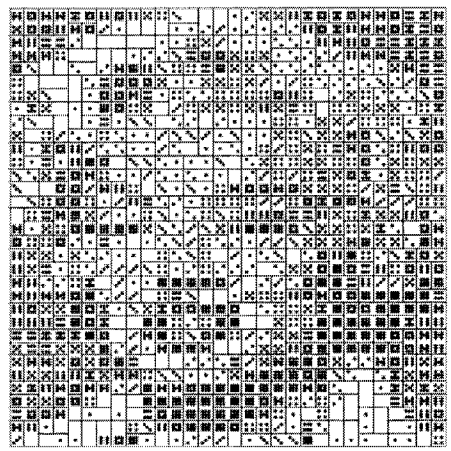
As we have seen the local search algorithm can find a local optimal solution of an instance of the domino portrait problem. This solution can be improved by modifying the cost function. The current cost function does not care about the important data in the photo-matrix (see Figure 3.3-C) for example, eyes, mouth and nose. In the next section we will modify the cost function so that the algorithm starts filling the dominoes in the important positions. This can be done by constructing a matrix called the support matrix that tells us the important data in the photo-matrix. The support matrix is constructed using a significant method called the singular value decomposition.

3.8 Improving The Cost Function

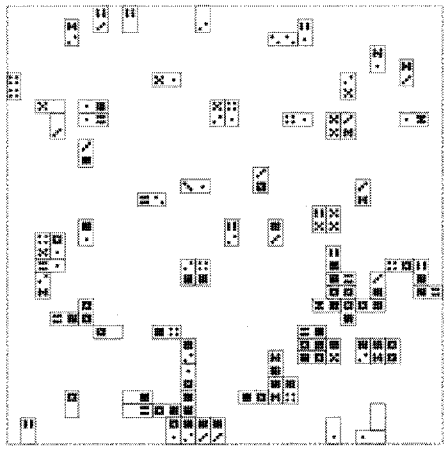
In this section we modify the cost function so that we can get a better solution for the domino portrait. The current cost function does not care about the important data in the photo-matrix (see Figure 3.3-C) like eyes, nose, and mouth, and insignificant data like background. The function is improved using a matrix called the *support matrix*. The purpose of the support matrix is to help in *determining* facial features and to *weigh* the entries of the photo matrix. That is, the entries in the support matrix that have higher numbers are referred to the position of important data in the photo matrix. The support matrix can be constructed using the significant method called the singular value decomposition. In the following, we show how can we modify the current function.



A: A local optimal solution of cost function equal 4833.



B: A second local optimal solution of cost function equal 4833.



C: The intersections of the dominoes in Figure A and B.

Figure 3.37: Two local optimal solutions of the instance of Marilyn Monroe (A and B) and the intersection between these solutions (in C).

3.8.1 New Cost Function

According to the information that we will get from the support matrix, we modify the old cost formula of the objective function and construct a new one. This formula can tell us which important data we have to start with. This can be done by increasing or decreasing the cost of the decision variables $x(m,n,o,i,j)$ according to whether this variable is important or not. This means, the penalty of placing a domino in an important position is more than the penalty of placing it in an insignificant position like the background.

The steps of developing the new cost function are as follows: Let $A(a_{ij})$ be the $M \times N$ photo-matrix (Figure 3.3-C) and $B(b_{ij})$ be the $M \times N$ support matrix where the entry b_{ij} is the measure of the entry a_{ij} in the photo-matrix A. That is, if $b_{i_1,j_1} \geq b_{i_2,j_2}$, the entry a_{i_1,j_1} is more important than the entry a_{i_2,j_2}.

Now, we modify the cost $c(m,n,o,i,j)$ of placing the domino (m,n) with orientation o in the position (i,j) of domino-matrix (Figure 3.3-D). Without loss of generality, we let $o = v_1$. For the other orientations v, h, v_2, h_1 and h_2 the procedure is the same.

Step 1: We start with the old cost $c(m,n,v_1,i,j) = c_1(m,n,v_1,i,j)$ which is defined as

$$c_1(m,n,v_1,i,j) = (m - a_{i,j})^2 + (n - a_{i+1,j})^2,$$

The cost c_1 take the values between 0 and $2(D-1)^2$, where $(D-1)$ is the kind of the complete set that we are using. For example if we are using a complete set of double 9 dominoes, then $D = 10$. That is,

$$0 \leq c_1 \leq 2(D-1)^2.$$

The best value of the cost c_1 is 0 and the worst (bad) value is $2(D-1)^2$. The real line presentation of the cost c_1 is as follows

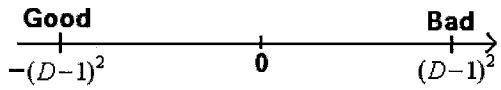
Step 2 : We shift the cost c_1 to the left with the amount of $D_1 = \lfloor \frac{(D-1)^2}{2} \rfloor$, where $\lfloor \cdot \rfloor$ is the greatest integer function. The cost c_1 becomes c_2 which is defined as

$$c_2(m, n, v_1, i, j) = \left[(m - a_{i,j})^2 - D_1\right] + \left[(n - a_{i+1,j})^2 - D_1\right]$$

that is,

$$-(D-1)^2 \leq c_2 \leq (D-1)^2$$

The real line presentation of the cost c_2 is

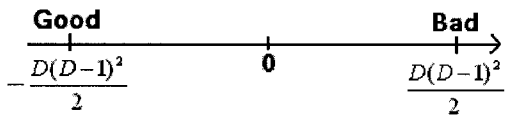


Step 3 : Now, we use the support matrix $B(b_{ij})$ to weigh the entries of the photo matrix A. That is, we multiply each term of the cost c_2 by $(1 + b_{i,j})$ and rename it as c_3. The cost c_3 is given by

$$c_3(m, n, v_1, i, j) = (1+b_{i,j})\left[(m-a_{i,j})^2-D_1\right]+(1+b_{i+1,j})\left[(n-a_{i+1,j})^2-D_1\right]$$

and bounded by

$$-\frac{D(D-1)^2}{2} \leq c_3 \leq \frac{D(D-1)^2}{2},$$



Step 4 :

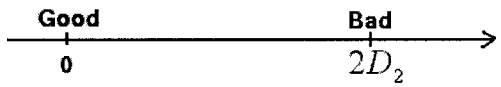
Finally, we shift c_3 back by $D_2 = D(D-1)^2$ so that $c_3 \geq 0$. The cost c_3 becomes c_4 which is defined as

$$c_4(m, n, v_1, i, j) = (1+b_{i,j})\left[(m-a_{i,j})^2-D_1\right]+(1+b_{i+1,j})\left[(n-a_{i+1,j})^2-D_1\right]+D_2$$

and so, it is bounded by 0 and $2D_2$, that is,

$$0 \leq c_4 \leq 2D_2.$$

The corresponding real line representation of the cost c_4 is given by



Hence, the new cost of placing the domino (m, n) with orientation v_1 in position row i and column j in the domino-matrix is denoted as $\bar{c}(m, n, v_1, i, j)$ and defined by the following equation

$$\bar{c}(m,n,v_1,i,j) = (1+b_{i,j})\big[(m-a_{i,j})^2 - D_1\big] + (1+b_{i+1,j})\big[(n-a_{i+1,j})^2 - D_1\big] + D_2 \tag{3.9}$$

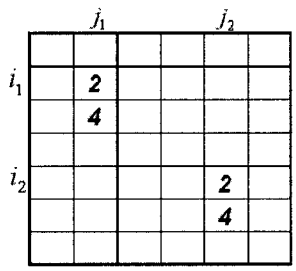
where

$$D_1 = \lfloor \frac{(D-1)^2}{2} \rfloor \quad and \quad D_2 = D(D-1)^2$$

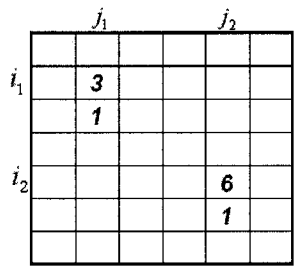
Example 3.8. *Assume that the following data are taken from a photo matrix A, and the corresponding support matrix B, case $D = 7$ and $s = 1$:*

$a_{i_1,j_1} = a_{i_2,j_2} = 2, a_{i_1+1,j_1} = a_{i_2+1,j_2} = 4, b_{i_1,j_1} = 3, b_{i_2,j_2} = 6$ *and* $b_{i_1+1,j_1} = a_{i_2+1,j_2} = 1$.

The photo-matrix and the support matrix are as follow

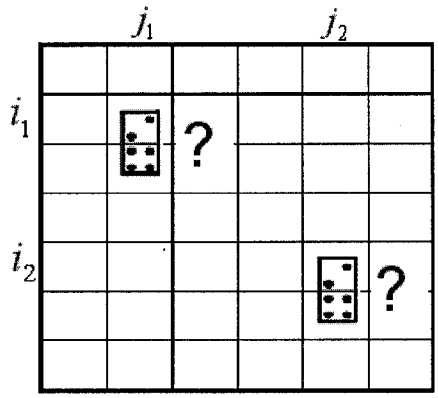


(photo matrix A)



(Support matrix B)

Since, $s = 1$, we have only one complete set of dominos. Namely, each domino (m, n) must be used exactly one time. Therefore, we need to use them in places that have important data. Here we use the support matrix B to determine that. Assume that we need to use the domino $(2, 4)$ in the domino matrix and want to know which place is important so that this domino is placed in



(The domino-matrix)

Now, by formula (3), The cost of placing domino $(2,4)$ *with orientation* v_1 *in position* (i_1, j_1) *of the domino-matrix can be computed as follows:*

Note:

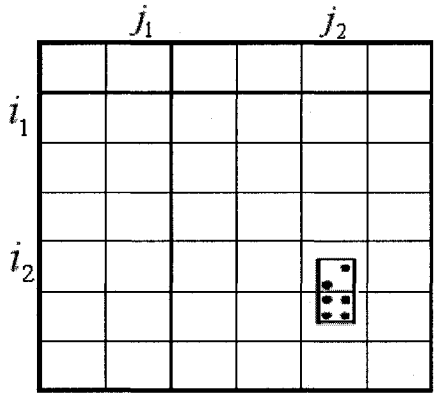
$D_1 = \lfloor \frac{(D-1)^2}{2} \rfloor = 18$ *and* $D_2 = D(D-1)^2 = 252$, *so*

$$\begin{aligned} c(2,4,v_1,i_1,j_1) &= (1+b_{i_1,j_1})\left[(2-a_{i_1,j_1})^2 - D_1\right] + \\ &\quad (1+b_{i_1+1,j_1})\left[(4-a_{i_1+1,j_1})^2 - D_1\right] + D_2 \\ &= (1+3)[(2-2)^2 - 18] + (1+1)[(4-4)^2 - 18] + 252 \\ &= 144 \end{aligned}$$

whereas, the cost if it placed in position (i_2, j_2) *with the same orientation is equal to*

$$\begin{aligned} c(2,4,v_1,i_2,j_2) &= (1+b_{i_2,j_2})\left[(2-a_{i_2,j_2})^2 - D_1\right] + \\ &\quad (1+b_{i_2+1,j_2})\left[(4-a_{i_2+1,j_2})^2 - D_1\right] + D_2 \\ &= (1+6)[(2-2)^2 - 18] + (1+1)[(4-4)^2 - 18] + 252 \\ &= 90 \end{aligned}$$

Hence, we place the domino, $(2,4)$ *in position* (i_2, j_2) *of the domino matrix (see Figure). That is, we let the corresponding decision variables* $x(2,4,v_1,i_2,j_2) = 1$ *and* $x(2,4,v_1,i_1,j_1) = 0$. *The domino matrix will be of the form*



(Domino matrix)

Therefore, once the support matrix determines the important data, like eyes, nose and mouth in the photo-matrix we use the new cost function to tell us which important position we should start with. By this modification we will find a better solution of the domino portrait problem when we apply the greedy algorithm and the local search algorithm. For example, Figure 3.38 shows the domino portrait of Marilyn Monroe which was constructed from 3^2 complete set of double nine dominoes. The portrait is constructed using the local search algorithm with the new cost function defined in equation 3.9 and the support matrix. The matrix is determined using the singular value decomposition.

In the next section, we will use the singular value decomposition to construct the support matrix so that we can use the new cost function defined above.

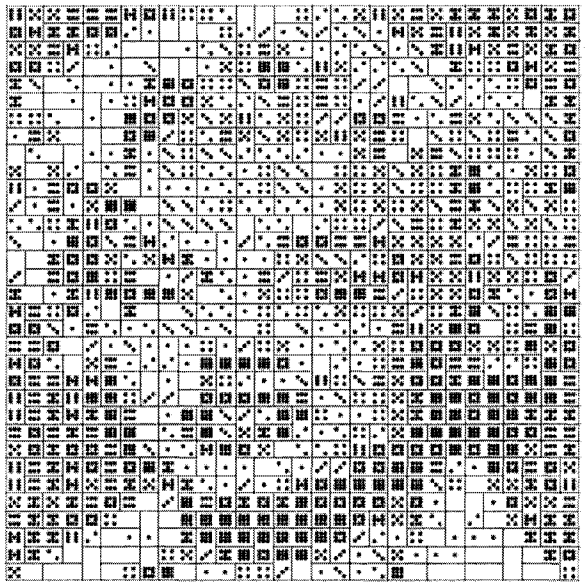


Figure 3.38: A domino portrait of Marilyn Monroe that was constructed using the local search algorithm with the new cost function and the support matrix which was determined using the SVD.

3.9 Singular Value Decomposition (SVD)

A singular value decomposition is a factorization of a given $m \times n$ matrix. It has a lot of important applications in many areas. Specifically in our area it can help us to determine the important data in the photo matrix. In the following sections we will state the algorithm of the singular value decomposition and related theorems and definitions. Moreover, we will show how it determines the important data in the photo matrix.

3.9.1 Definitions and Theorems

Definition 3.12.
An $n \times n$ matrix Q is said to be an orthogonal matrix if the column vectors of Q form an orthogonal set in $\mathbb{R}^n$.

Definition 3.13.
Let A be an $m \times n$ matrix with $(m \geq n)$. The factorization $U\Sigma V^T$ is called ***a singular value decomposition*** *of A where: U is an $m \times m$ orthogonal*

matrix, V is an $n\times n$ orthogonal matrix and Σ is an $m\times n$ matrix of the form

$$\Sigma = \begin{pmatrix} \sigma_1 & & \boldsymbol{0} \\ & \ddots & \\ \boldsymbol{0} & & \sigma_n \\ & \boldsymbol{0} & \end{pmatrix}, \quad \sigma_1 \geq \sigma_2 \geq \cdots \geq \sigma_n \geq 0$$

and σ_i's are called the ***singular values*** *of A*

Remark 3.9.1.
The rank of A = the number of non zero singular values.

Definition 3.14.
An $n\times n$ matrix A is said to be diagonalizable if there exists a nonsingular matrix X and a diagonal matrix D such that $X^{-1}AX = D$. We say that X diagonalizes A.

Theorem 3.9.1. *(Singular Value Decomposition)*
If A is a real $m\times n$ matrix, then A has a singular value decomposition. That is, $A = U\Sigma V^T$ where U is an $m\times m$ orthogonal matrix, V is an $n\times n$ orthogonal matrix and Σ is an $m\times n$ matrix of the form

$$\Sigma = \begin{pmatrix} \sigma_1 & & \boldsymbol{0} \\ & \ddots & \\ \boldsymbol{0} & & \sigma_n \\ & \boldsymbol{0} & \end{pmatrix}, \quad \sigma_1 \geq \sigma_2 \geq \cdots \geq \sigma_n \geq 0$$

Proof.
Since A^TA is an $n\times n$ symmetric matrix, its eigenvalues are real and it has an orthogonalizing matrix V. Moreover, the eigenvalues are non-negative. To see that let λ be an eigenvalue of A^TA and X be the eigenvector corre-

sponding to λ. We have the following:

$$\begin{aligned}(A^TA)X &= \lambda X \\ X^TA^TAX &= \lambda X^TX \\ (AX)^T(AX) &= \lambda X^TX \\ \| AX \|^2 &= \lambda \| X \|^2 \\ \Rightarrow \quad \lambda &= \frac{\| AX \|^2}{\| X \|^2} \geq 0\end{aligned}$$

Hence, $\lambda \geq 0$.

We can order the columns of Matrix V so that the corresponding eigenvalues of A^TA satisfy

$$\lambda_1 \geq \lambda_2 \geq \cdots \geq \lambda_n \geq 0$$

The singular values, σ_i , of A are given by

$$\sigma_i = \sqrt{\lambda_i}, \quad i = 1, 2, \cdots, n$$

Remark 3.9.2.

The rank of A equals the rank of A^TA, say r, which is equal to the number of non-zero eigenvalues λ_j, i.e., the rank of $A = r$.

Now let

$$\lambda_1 \geq \lambda_2 \geq \cdots \geq \lambda_r > 0 \ \ and \ \ \lambda_{r+1} = \lambda_{r+2} = \cdots = \lambda_n = 0,$$

also

$$\sigma_1 \geq \sigma_2 \geq \cdots \geq \sigma_r > 0 \ \ and \ \ \sigma_{r+1} = \sigma_{r+2} = \cdots = \sigma_n = 0$$

Let

$$\Sigma_1 = \begin{pmatrix} \sigma_1 & & \mathbf{0} \\ & \ddots & \\ \mathbf{0} & & \sigma_1 \end{pmatrix}$$

so,

$$\Sigma = \begin{pmatrix} \Sigma_1 & \mathbf{0} \\ \mathbf{0} & \mathbf{0} \end{pmatrix}$$

is the $m \times n$ diagonal matrix. Let $V_1 = (v_1, \cdots, v_r)$ and $V_2 = (v_{r+1}, \cdots, v_n)$ where v_i is the eigenvector of $A^T A$ corresponding to the eigenvalue λ_i, $i = 1, \cdots, n$. We have $V = V_1 + V_2$. Since $v_j = 0, j = r+1, \cdots, n$, then

$$A^T A v_j = \lambda v_j = \mathbf{0} v_j = \mathbf{0}$$

Hence the column vectors of V_2 form an orthogonal basis for
$N(A^T A) = N(A)$
That is,

$$AV_2 = \mathbf{0}$$

Now, since V is Orthogonal matrix, $I = VV^T$ and hence

$$I = VV^T = V_1 V_1^T + V_2 V_2^T$$

also,

$$A = AI = A(V_1 V_1^T + V_2 V_2^T) = AV_1 V_1^T + AV_2 V_2^T = AV_1 V_1^T,$$

the matrix A can be written as

$$A = AV_1 V_1^T$$

We can Construct the $m \times m$ matrix U of the singular value decomposition $U\Sigma V^T$ by the following:
since U is orthogonal matrix we have

$$U^T U = I,$$

so

$$A = U\Sigma V^T \iff AV = U\Sigma.$$

By comparing the first r columns of each side of equation 1, we get

$$Av_j = \sigma_j u_j, \quad j = 1, \cdots, r$$

which is equivalent to

$$u_j = \frac{1}{\sigma_j} Av_j, \quad j = 1, \cdots, r. \tag{3.10}$$

If we let

$$U_1 = (u_1, \cdots, u_r),$$

then

$$AV_1 = U_1\Sigma_1.$$

Moreover, the column vectors of U_1 form an orthogonal set. This is because

$$\begin{aligned} u_i^T u_j &= (\frac{1}{\sigma_i} v_i^T A^T)(\frac{1}{\sigma_j} A v_j) \quad 1 \le i \le r, \ \ 1 \le j \le r \qquad \text{(by equation 3.10)} \\ &= \frac{1}{\sigma i \sigma j} v_i^T (A^T A v_j) \\ &= \frac{1}{\sigma i \sigma j} v_i^T (\lambda_j v_j) \quad \text{(since } v_j \text{ is an eigenvector of } A^T A \text{ corresponding to } \lambda_j) \\ &= \frac{\sqrt{\lambda_j}}{\sqrt{\lambda_j}} \frac{\lambda_j}{\sigma_i \sqrt{\lambda_j}} = \frac{\sigma_j}{\sigma_i} v_i^T v_j \\ \Longrightarrow \quad u_i^T u_j &= \frac{\sigma_j}{\sigma_i} v_i^T v_j \end{aligned}$$

Because V is an orthogonal matrix and its column vectors form an orthogonal set, i.e.,

$$v_i^T v_j = \begin{cases} 1, & \text{for } i = j; \\ 0, & i \neq j \end{cases}$$

therefore

$$\begin{aligned} u_i^T u_j = \frac{\sigma_j}{\sigma_i} v_i^T v_j &= \begin{cases} 1, & i = j; \\ 0, & i \neq j. \end{cases} \quad = \delta_{ij} \\ &= \delta_{ij}. \end{aligned}$$

Furthermore, from equation 1, we have u_j, $1 \le j \le r$, in the column space of A and the dimension of the column space is r, so $u_1, \ldots, u_r$ form an orthogonal basis for $R(A)$.

Definition 3.15.

Let Y be a subspace of $\mathbb{R}^n$. The set of all vectors in $\mathbb{R}^n$ that are orthogonal to every vector in Y is denoted by $Y^\perp$, i.e.,

$$Y^\perp = \{X \in \mathbb{R}^n \ : \ X^T y = 0 \ \ \forall \ y \in Y\}$$

The vector space $R(A)^\perp = N(A^T)$ has dimension $m - r$.

Now since $u_1, u_2, \ldots, u_r$ form an orthogonal basis for $R(A)$ and $u_{r+1}, \ldots, u_m$ form an orthogonal basis for $R(A)$, then $u_1, \ldots, u_m$ form an orthogonal basis for $\mathbb{R}^m$.

Hence U is an orthogonal matrix.

Because

$$u_i^T u_j = \delta_{ij} \quad 1 \leq i \leq r, \quad 1 \leq j \leq r,$$

and

$$A = AV_1V_1^T, \quad \Sigma = \begin{pmatrix} \Sigma_1 & \mathbf{0} \\ \mathbf{0} & \mathbf{0} \end{pmatrix}$$

$$\begin{aligned} \Longrightarrow \quad U\Sigma V^T &= \begin{pmatrix} U_1 & U_2 \end{pmatrix} \begin{pmatrix} \Sigma_1 & \mathbf{0} \\ \mathbf{0} & \mathbf{0} \end{pmatrix} \begin{pmatrix} V_1^T \\ V_2^T \end{pmatrix} \\ &= U_1\Sigma_1 V_1^T \\ &= AV_1V_1^T \\ &= A \end{aligned}$$

Hence

$$A = U\Sigma V^T.$$

□

In the following we will explain how the singular value decomposition can be used to construct the support matrix that is described in Section 3.8. Let the $M \times N$ matrix A be the matrix of the target image, after we rescale it. Theorem 3.9.1, guarantees that the singular value decomposition is defined for matrix A. Now, let $r > 0$ be the rank of matrix A, the singular value decomposing can be used to determine an $M \times N$ matrix $\bar{A}$ of rank k and $0 < k < r$ that is close to A with respect to Frobenius norm $\| \cdot \|_F$ [33], where

$$\| A \|_F = \left(< A, A > \right)^{\frac{1}{2}} = \left(\sum_{i=1}^{m} \sum_{j=1}^{n} a_{ij}^2 \right)^{\frac{1}{2}}.$$

That is, if

$$A = U\Sigma V^T = \sigma_1 u_1 v_1^T + \sigma_2 u_2 v_2^T + \ldots + \sigma_r u_r v_r^T$$

then

$$\bar{A} = U\Sigma V^T = \sigma_1 u_1 v_1^T + \sigma_2 u_2 v_2^T + \ldots + +\sigma_k u_k v_k^T$$

where $0 < k < r$. After that, we find the $M \times N$ *different matrix* (ΔA) which is defined by

$$\Delta A = |A - \bar{A}|$$

and rescale it from 0 to $D - 1$. Finally, the entries of the different matrix, ΔA, that have higher scale are reflected to the positions of the important data in the photo matrix, which is the support matrix. That is, the different matrix, ΔA, is the support matrix described in Section 3.8.

Example 3.9. *Figure 3.39 shows an original image of Marilyn Monroe of size* 195×250. *Applying the singular value decomposition to this image we have the following low rank images* $A_1, \ldots, A_5, A_{10}$ *and* A_{50}. *These images getting better when the rank is is increasing. Moreover, Figure 3.40 show the different between the original image and every images with low rank, i.e.,* $|A - A_1|, \ldots, |A - A_5|, |A - A_{10}|$ *and* $|A - A_{50}|$.

Consequently, we compose the support matrix so that we can use the new cost function, defined in equation 3.9, that forces the algorithms, the greedy and local search algorithms, to start with the important positions in the image.

3.9.2 Algorithm for Computing the SVD

As we have seen, the singular value decomposition can be used in our main problem to determine the important data in the photo matrix. Because we are using C++ language, we need, as well, to use it for the singular value decomposition. There are some programs, for example, CLAPACK ,

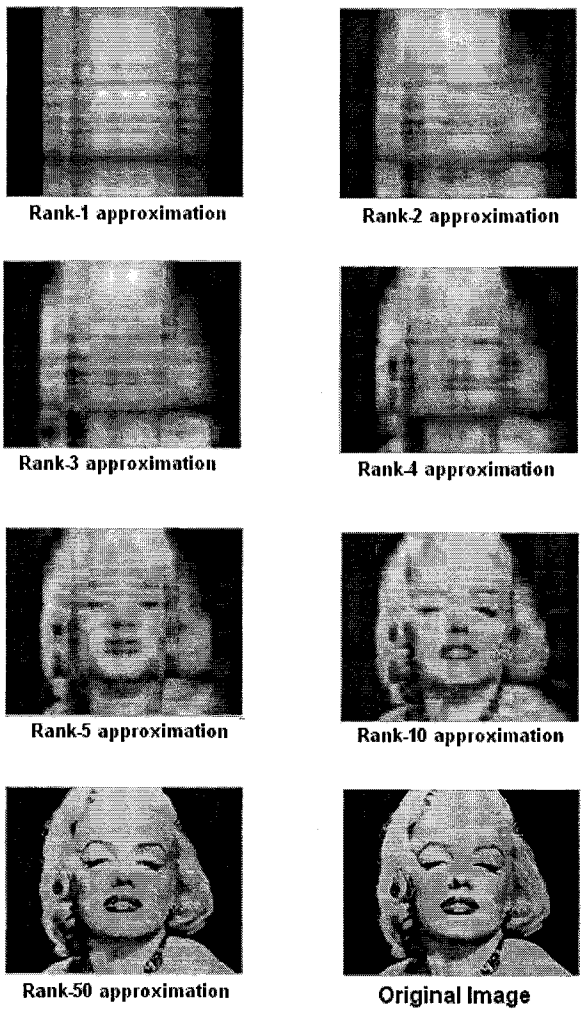


Figure 3.39: An original image of Marilyn Monroe and the low rank approximations when the SVD is applied. The details of this image is described in Example 3.9

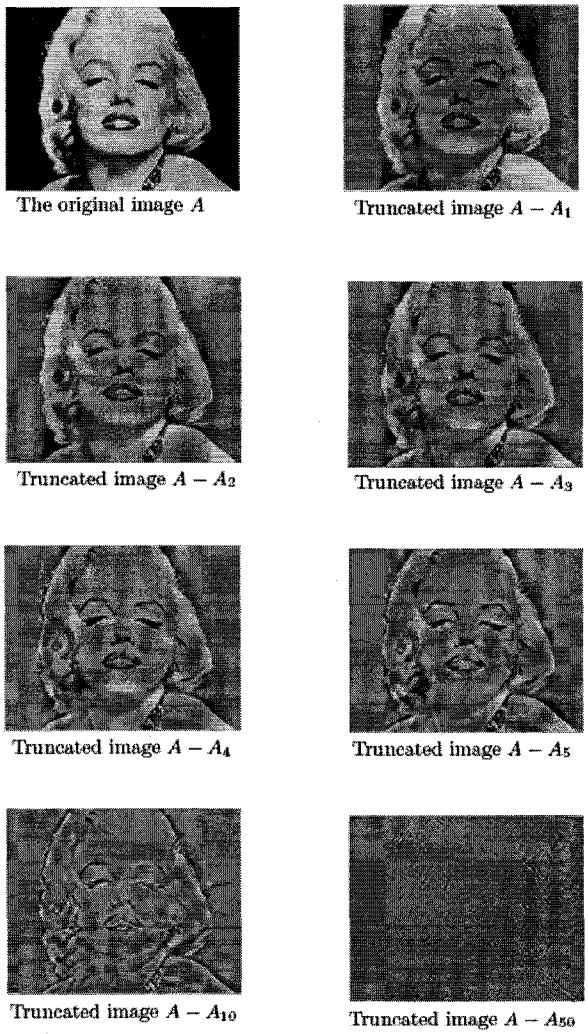


Figure 3.40: The differences between the original image of Marilyn Monroe (top left) and the low rank approximations images defined in Figure 3.39

that compute the SVD. In this thesis, we developed a C++ program that computes the SVD. Determining the approximate eigenvalues of A^TA is a complicated step in computing the singular value decomposition. Therefore, we need an efficient method to do that. Such methods are called Householder's and the QR methods [9].

Given $m \times n$ matrix A, $(m \geq n)$, The singular value decomposition, $U\Sigma V^T$ of the matrix A, can be computed by the following steps:

Step 1 Compute the A^TA.

Step 2 Evaluate the eigenvalues λ_i and the eigenvectors $\bar{v}_i$ of A^TA.

Step 3 Let $V_1 = (v_1, \ldots, v_r)$ and $V_1 = (v_{r+1}, \ldots, v_n)$ such that, $v_i = \frac{1}{\|\bar{v}_i\|}\bar{v}_i$, where $\bar{v}_i$ are the eigenvectors of A^TA corresponding to the eigenvalues λ_i,

$$\lambda_1 \geq \lambda_2 \geq \ldots \lambda_r > 0, \quad and \quad \lambda_{r+1} = \cdots = \lambda_n = 0.$$

Step 4 Let $U = \begin{pmatrix} U_1 & U_2 \end{pmatrix}$.

$$U_1 = (u_1, u_2, \ldots, u_r); \quad u_i = \frac{1}{\sigma_i} A v_i, \quad i = 1, \ldots, r$$
$$U_2 = (u_{r+1}, u_{r+2}, \ldots, u_m); \quad A^T u_j = 0, \quad j = r+1, \ldots, m.$$

i.e., $u_{r+1}, u_{r+2}, \ldots, u_m$ must form an orthogonal basis for $N(A^T)$.

Householder's Method

Householder's method [9] has a large application in mathematical areas. One of the famous areas is called an eigenvalue approximation. We will use this method to help us approximate the eigenvalues of A^TA in order to determine the singular values of a given matrix A. In this section we discuss Householder's method briefly. The idea of Householder's method is to determine a symmetric tridiagonal matrix that is similar to a given symmetric matrix. The new similar matrix has the same eigenvalues that the original matrix does. To approximate the eigenvalues of the similar

matrix, we use one of the proficient methods called the QR method which we will talk about in the following section.

Definition 3.16. *Let $w \in \mathbb{R}^n$ with $w^\top w = 1$. The $n \times n$ matrix*

$$P = I - 2ww^\top$$

is called a Householder's transformation.

There are some properties of the householder's transformation. Two important properties are the symmetry and the orthogonality which are stated in the following theorem:

Theorem 3.9.2. *A Householder's transformation, $P = I - 2ww^\top$, is symmetric and orthogonal, i.e., $P^{-1} = P$.*

Proof.
For the symmetry we need to show that $P^\top = P$. we have

$$P^\top = (I - 2ww^\top)^\top = I^\top - 2(ww^\top)^\top = I - 2w^{\top\top}w^\top = I - 2ww^\top = P.$$

And for the orthogonality we need to proof that $PP^\top = I$, which can be proved as follow: since P is symmetric we have $P^\top = P$, i.e.,

$$(I - 2ww^\top)^\top = (I - 2ww^\top)$$

moreover, by the definition of the Householder's transformation, we have

$$w^\top w = 1$$

and hence

$$\begin{aligned} PP^\top &= (I - 2ww^\top)(I - 2ww^\top)^\top = (I - 2ww^\top)(I - 2ww^\top) \\ &= I - 2ww^\top - 2ww^\top + 4w(w^\top w)w^\top = I - 4ww^\top + 4ww^\top \\ &= I, \end{aligned}$$

consequently,

$$P^\top = P^{-1} = P.$$

□

The tridiagonal symmetric matrix A^{n-1} that we want to be similar to the symmetric $n \times n$ matrix A is defined by the following:

$$A^{(n-1)} = P^{(n-2)}P^{(n-3)} \dots P^{(1)}AP^{(1)} \dots P^{(n-3)}P^{(n-2)}.$$

where $P^{(i)}$ are the Householder's transformation matrices. The Householder's method starts by finding the first transformation $P^{(1)}$ such that the entries of $A^{(2)} = P^{(1)}AP^{(1)}$ satisfy the following properties:

$$a_{j1}^{(2)} = a_{1j}^{(2)} = 0, \quad for \quad each \quad j = 3, 4, \dots, n \tag{3.11}$$

Now our aim is to determine $P^{(1)}$ which is equal to $P^{(1)} = I - 2ww^\top$. That is to determine the vector $w = (w_1, w_2, \dots, w_n)^\top \in \mathbb{R}^n$. We chose w_i according to two conditions given by the following:

1. $w^\top w = 1$

2. The entries in the matrix

$$A^{(2)} = P^{(1)}AP^{(1)} = (I - 2ww^\top)A(I - 2ww^\top)$$

have the property that

$$a_{11}^{(2)} = a_{11} \quad and \quad a_{j1}^{(2)} = a_{1j}^{(2)} = 0, \quad for \quad each \quad j = 3, 4, \dots, n$$

by this choice we have n conditions on the n unknown w_i, $i = 0, 1, \dots, n$. To satisfy $a_{11}^{(11)} = a_{11}$ we let $w_1 = 0$ and hence the transformation $P^{(1)}$ becomes

$$P^{(1)} = \begin{pmatrix} 1 & 0 & \cdots & 0 \\ 0 & & & \\ \vdots & & \hat{P} & \\ 0 & & & \end{pmatrix},$$

where $\hat{P} = I - 2\hat{w}\hat{w}^\top$ is $(n-1) \times (n-1)$ Householder's transformation and $\hat{w} = (w_2, \dots, w_n)^\top \in \mathbb{R}^{(n-1)}$.
To determine the remaining w_i we multiply $P^{(1)}$ by the first column of A and equating the result by the following

$$P^{(1)}(a_{11},\ldots,a_{n1})^\top = (a_{11},\alpha,0,\ldots,0)^\top \tag{3.12}$$

where α will be determined later.
now if we let $\hat{\mathbf{y}} = (a_{21},\ldots,a_{n1})^\top \in \mathbb{R}^{(n-1)}$, then equation 3.12 becomes

$$P^{(1)}\begin{pmatrix} a_{11} \\ a_{21} \\ \vdots \\ a_{n1} \end{pmatrix} = P^{(1)} = \begin{pmatrix} 1 & 0 & \cdots & 0 \\ 0 & & & \\ \vdots & & \hat{P} & \\ 0 & & & \end{pmatrix} \cdot \begin{pmatrix} a_{11} \\ \\ \hat{\mathbf{y}} \\ \\ \end{pmatrix} = \begin{pmatrix} a_{11} \\ \alpha \\ 0 \\ \vdots \\ 0 \end{pmatrix}$$

Now

$$\begin{aligned} \hat{P}\hat{\mathbf{y}} &= (I_{(n-1)} - 2\hat{w}\hat{w}^\top)\hat{\mathbf{y}} = \hat{\mathbf{y}} - 2\hat{w}\hat{w}^\top)\hat{\mathbf{y}} \\ &= \hat{\mathbf{y}} - 2\hat{w}(\hat{w}^\top\hat{\mathbf{y}}), \quad (\hat{w}^\top\hat{\mathbf{y}} = r \ \ scaler) \\ &= \hat{\mathbf{y}} - 2r\hat{w} = (\alpha,0,\ldots,0)^\top \end{aligned} \tag{3.13}$$

That is

$$(a_{21} - 2rw_2, a_{31} - 2rw_3, \ldots, a_{n1} - 2rw_n)^\top = (\alpha,0,\ldots,o)^\top$$

and by equating the component of each side we get

$$\begin{aligned} \alpha &= a_{21} - 2rw_2 \\ 0 &= a_{31} - 2rw_3 \\ &\vdots \\ 0 &= a_{n1} - 2rw_n \end{aligned}$$

that is,

$$w_2 = \frac{a_{21} - \alpha}{2r} \tag{3.14}$$

$$w_j = \frac{a_{j1}}{2r}, \quad j = 3,4,\ldots,n. \tag{3.15}$$

consequently, once we determine α and r we know $w_j, j = 2,3,\ldots,0$. We observe from equations 3.14 and 3.15 that w_j is defined for $r \neq 0$. That

means we need to choose α so that $r \neq 0$.

Now to find α we rewrite equations 3.14 and 3.15 as follows

$$2rw_2 = a_{21} - \alpha \tag{3.16}$$

$$2rw_j = a_{i1}, \quad j = 3, 4, \ldots, n. \tag{3.17}$$

and then we square both sides of the equations and add them togethers

$$4r^2(w_2^2 + w_3^2 + \ldots + w_n^2) = (a_{21} - \alpha)^2 + \sum_{j=3}^{n} a_{j1}^2 \tag{3.18}$$

Now since

$$w_1 = 0 \quad and \quad 1 = w^\top w = w_2^2 + w_3^2 + \ldots + w_n^2$$

$$\Longrightarrow w_2^2 + \ldots + w_n^2 = 1$$

and hence, equation 3.18 becomes as follows

$$4r^2 = \sum_{j=2}^{n} a_{j1}^2 - 2\alpha a_{21} + \alpha^2 \tag{3.19}$$

From equation 3.13 we have

$$\hat{P}\hat{\mathbf{y}} = (\alpha, 0, \ldots, 0)^\top$$

so

$$\begin{aligned}\alpha^2 &= (\alpha, 0, \ldots, 0)(\alpha, 0, \ldots, 0)^\top = (\hat{P}\hat{\mathbf{y}})^\top(\hat{P}\hat{\mathbf{y}}) = \\ &= \hat{\mathbf{y}}^\top(\hat{P}^\top\hat{P})\hat{\mathbf{y}} = \hat{\mathbf{y}}^\top\hat{\mathbf{y}} \\ &= a_{21}^2 + a_{31}^2 + \ldots + a_{n1}^2 = \sum_{j=2}^{n} a_{j1}^2.\end{aligned}$$

Note: $\hat{P}^\top\hat{P} = 1$, by the orthogonality of P.

thus

$$\alpha^2 = \sum_{j=2}^{n} a_{j1}^2 \Longrightarrow \alpha = \Big(\sum_{j=2}^{n} a_{j1}^2\Big)^{\frac{1}{2}} \tag{3.20}$$

Hence equation 3.19 will be

$$4r^2 = \sum_{j=2}^{n} a_{j1}^2 - 2\alpha a_{21} + \sum_{j=2}^{n} a_{j1}^2 = 2\sum_{j=2}^{n} a_{j1}^2 - 2\alpha a_{21}$$

By dividing both sides by 2, we get

$$2r^2 = \sum_{j=2}^{n} a_{j1}^2 - \alpha a_{21}$$

As we mentioned earlier, w_j is defined as whenever $r \neq 0$.
Now, $r = 0$ when

$$\sum_{j=2}^{n} a_{j1}^2 - \alpha a_{21} = 0$$

from equation 3.20 we have

$$\sum_{j=2}^{n} a_{j1}^2 - \Big(\sum_{j=2}^{n} a_{j1}^2\Big)^{\frac{1}{2}} a_{21} = 0 \tag{3.21}$$

Let $x = \sum_{j=2}^{n} a_{j1}^2$, the solution of equation 3.21 will be as follows

$$x - x^{1/2} a_{21} = 0 \quad \Longrightarrow \quad x^{1/2}(x^{1/2} - a_{21}) = 0$$

by solving this equation we get

$$x^{1/2} = 0 \quad \Longrightarrow \quad \sum_{j=2}^{n} a_{j1}^2 = 0 \quad \Longleftrightarrow \quad a_{21} = a_{31} = \ldots = a_{n1} = 0$$

or

$$x^{1/2} - a_{21} = 0 \quad \Longrightarrow \quad a_{21}^2 = \sum_{j=2}^{n} a_{j1}^2 = a_{21}^2 + a_{31}^2 + \cdots + a_{n1}^2$$

$$\Longrightarrow \quad a_{31} = \ldots = a_{n1} = 0$$

So, for any number of a_{21} and $a_{31} = \ldots = a_{n1} = 0$, we have $r = 0$. That is, for $j = 2, 3, \ldots, n$, we have w_j, is defined whenever $a_{j1} \neq 0$.
To define w_j more efficiently, we add the following condition $a_{21} = 0$ to the above condition, i.e. $a_{21} = a_{31} = \ldots = a_{n1} = 0$, and change the α to be equals

$$\alpha = -sgn(a_{21})\Big(\sum_{j=2}^{n} a_{j1}^2\Big)^{1/2}.$$

This can be shown in claim 3.9.1. Before we state the claim, let us put forth the following definition,

Definition 3.17. *(Signum of x, sgn(x))*

$$sgn(x) = \begin{cases} 1 & if\ x > 0 \\ 0 & if\ x = 0 \\ -1 & if\ x < 0 \end{cases}$$

Claim 3.9.1.

Let

$$2r^2 = \sum_{j=2}^{n} a_{j1}^2 - \alpha a_{21} \tag{3.22}$$

and $\alpha = -sgn(a_{21})\big(\sum_{j=2}^{n} a_{j1}^2\big)^{1/2}$, *then*

$$r = 0 \ \ \textit{if and only if} \ \ a_{21} = a_{31} = \ldots = a_{n1} = 0.$$

Proof.

($\Leftarrow$) Let $a_{21} = a_{31} = \ldots = a_{n1} = 0$, so we have

$$2r^2 = \sum_{j=2}^{n} a_{j1}^2 - \alpha a_{21} = 0 - \alpha(0) = 0.$$

hence, $r = 0$

($\Rightarrow$) Let $r = 0$ and $\alpha = -sgn(a_{21})(\sum_{j=2}^{n} a_{j1}^2)^{1/2}$, we need to show that $a_{21} = a_{31} = \ldots = a_{n1} = 0$.

We have

$$\begin{aligned} 0 = 2r^2 &= \sum_{j=2}^{n} a_{j1}^2 - a_{21}\big[-sgn(a_{21})\big(\sum_{j=2}^{n} a_{j1}^2\big)^{1/2}\big] \\ &= \sum_{j=2}^{n} a_{j1}^2 + a_{21}sgn(a_{21})\big(\sum_{j=2}^{n} a_{j1}^2\big)^{1/2} \end{aligned}$$

now, applying Definition 3.17 to a_{21} we get the following three cases.

Case 1, when $a_{21} > 0$,

we have

$$sgn(a_{21}) = 1, \ \ \sum_{j=2}^{n} a_{j1}^2 > 0 \ \ and \ \ \big(\sum_{j=2}^{n} a_{j1}^2\big)^{1/2} > 0$$

which implies that

$$2r^2 = \sum_{j=2}^{n} a_{j1}^2 + a_{21}\Big(\sum_{j=2}^{n} a_{j1}^2\Big)^{1/2} > 0$$
$$\Longrightarrow \quad 2r^2 > 0 \quad \Longrightarrow r = 0$$
$$\Longrightarrow \text{ contradiction since } r = 0 \text{ and so } a_{21} \not< 0$$

Case 2, when $a_{21} < 0$, in this case

$$sgn(a_{21}) = -1, \ and \ a_{31} = \ldots = a_{n1} = 0$$

so

$$2r^2 = a_{21}^2 - a_{21}(a_{21}^2)^{1/2} = a_{21}^2 - a_{21}(-a_{21}) = 2a_{21}^2 \neq 0$$
$$\Longrightarrow r = 0$$

which is also a contradiction.
Hence, a_{21} should be zero, moreover,

$$a_{21} = a_{31} = \ldots = a_{n1} = 0$$

□

From claim 3.9.1 and choice of α and r we have w_j, $j = 1, 2, 3, \ldots, n$ is obtained and the construct of $P^{(2)}$ is as follows:
let

$$\alpha = -sgn(a_{21})\Big(\sum_{j=2}^{n} a_{j1}^2\Big)^{1/2},$$
$$r = \Big(\frac{\alpha^2 - a_{21}\alpha}{2}\Big)^{1/2},$$
$$w_1 = 0,$$
$$w_2 = \frac{a_{21} - \alpha}{2r},$$

and

$$w_j = \frac{a_{j1}}{2r}, \text{ for each } j = 3, \ldots, n.$$

Thus

$$A^{(2)} = P^{(1)}AP^{(1)} = \begin{pmatrix} a_{11}^{(2)} & a_{12}^{(2)} & 0 & \cdots & 0 \\ a_{21}^{(2)} & a_{22}^{(2)} & a_{23}^{(2)} & \cdots & a_{2n}^{(2)} \\ 0 & a_{32}^{(2)} & a_{33}^{(2)} & \cdots & a_{3n}^{(2)} \\ \vdots & \vdots & \vdots & & \vdots \\ 0 & a_{n2}^{(2)} & a_{n3}^{(2)} & \cdots & a_{nn}^{(2)} \end{pmatrix}$$

After computing $P^{(1)}$ and $A^{(2)}$, we do the same process for $k = 2, 3, \ldots, n-1$ which is as follows:

$$\begin{aligned} \alpha &= -sgn(a_{k+1,k}^{(k)})\Big(\sum_{j=k+1}^{n}(a_{jk}^{(k)})^2\Big)^{1/2}, \\ r &= \Big(\frac{\alpha^2 - \alpha a_{k+1,k}^{(k)}}{2}\Big)^{1/2}, \\ w_1^{(k)} &= w_2^{(k)} = \ldots = w_k^{(k)} = 0, \\ w_{k+1}^{(k)} &= \frac{a_{k+1,k}^{(k)} - \alpha}{2r}, \\ w_j^{(k)} &= \frac{a_{jk}^{(k)}}{2r}, \text{ for each } j = k+2, k+3, \ldots, n, \\ P^{(k)} &= I - 2w^{(k)}.(w^{(k)})^{\top}, \end{aligned}$$

and

$$A^{(k+1)} = P^{(k)}A^{(k)}P^{(k)}$$

The tridiagonal and symmetric matrix $A^{(n-1)}$ is computed when k reaches $n-2$, i.e.,

$$A^{(n-1)} = P^{(n-2)}P^{(n-3)}\ldots P^{(1)}AP^{(1)}\ldots P^{(n-3)}P^{(n-2)}$$

Remark 3.9.3.
If $a_{k,k+1}^{(k)} = a_{k,k+2}^{(k)} = \cdots = a_{k,n}^{(k)} = 0$ where $k = 1, 2, \ldots, n-2$, in this case we let $A^{(k+1)} = A^{(k)}$ and continue the process for the rest.

Householder's algorithm in C++

In this program we obtain a symmetric tridiagonal $n \times n$ matrix $A^{(n-1)}$ similar to a given $n \times n$ matrix $A = A^{(1)}$, where $A^{(k)} = (a_{ij}^{(k)})$ for $k =$

$1, 2, \ldots, n-1$ [33]

INPUT Dimension n; matrix A.
OUTPUT $A^{(n-1)}$

Step 1 For $k = 1, 2, \ldots, n-2$ do steps 2 - 14.

Step 2 Recall: " Remark 3.9.3 ":
If $a_{k,k+1}^{(k)} = a_{k,k+2}^{(k)} = \cdots = a_{k,n}^{(k)} = 0$ For $k = 1, 2, \ldots, n-2$,
Set
$A^{(k+1)} = A^{(k)}$ and continue (i.e., start from step 1 with $k = k+1$).

Step 3 Set

$$q = \sum_{j=k+1}^{n} \left(a_{jk}^{(k)}\right)^2.$$

Step 4

$$If \quad a_{k+1,k}^{(k)} = 0 \quad then \quad set \quad \alpha = -q^{\frac{1}{2}}$$

$$else \quad set \quad \alpha = -\frac{q^{\frac{1}{2}} a_{k+1,k}^{(k)}}{|a_{k+1,k}^{(k)}|}$$

Step 5 Set $RSQ = \alpha^2 - \alpha a_{k+1,k}^{(k)}$ (here $RSQ = 2r^2$)

Step 6
For $i = 1, \ldots, k$ set $v_i = 0$
$v_{k+1} = a_{k+1,k}^{(k)} - \alpha$
For $j = k+2, \ldots, n$ set $v_j = a_{j,k}^{(k)}$
(Note:$\mathbf{w} = \frac{1}{2r}\mathbf{v}$)

Step 7 For $j = k, \ldots, n$ set $u_j = (RSQ) \sum_{j=k+1}^{n} a_{ji}^{(k)} v_j$
(Note: $\mathbf{u} = \frac{1}{r} A^{(k)} \mathbf{w}$)

Step 8 Set $PROD = \sum_{i=k+1}^{n} v_i u_i$ (Note: $PROD = \mathbf{v}^\top \mathbf{u}$)

Step 9 For $j = k, \ldots, n$ set $z_j = u_j - \left(\frac{PROD}{2RSQ}\right) v_j$
(Note: $\mathbf{z} = \mathbf{u} - \mathbf{w}\mathbf{w}^\top \mathbf{u}$)

Step 10 For $l = k+1, \ldots, n-1$ do step 11 and 12
(Note: computed $A^{(k+1)} = (I - 2\mathbf{w}\mathbf{w}^\top) A^{(k)} (I - 2\mathbf{w}\mathbf{w}^\top)$

Step 11 For $j = l+1, \ldots, n$ set

$$a_{jl}^{(k+1)} = a_{jl}^{(k)} - v_l z_j - v_j z_l$$
$$a_{lj}^{(k+1)} = a_{jl}^{(k+1)}$$

Step 12 Set $a_{ll}^{(k+1)} = a_{ll}^{(k)} - 2 v_l z_l$.

Step 13 Set $a_{nn}^{(k+1)} = a_{nn}^{(k)} - 2 v_n z_n$

Step 14 For $j = k+2, \ldots, n$ set $a_{kj}^{(k+1)} = a_{jk}^{(k+1)} = 0$

Step 15 set

$$a_{k+1k}^{(k+1)} = a_{k+1,k}^{(k)} - v_{k+1} z_k$$
$$a_{k,k+1}^{(k+1)} = a_{k+1,k}^{(k+1)}.$$

(Note: The other elements of $A^{(k+1)}$ are the same as $A^{(k)}$)

Step 16 Print $A^{(n-1)}$ which is symmetric, tridiagonal and similar to A.

After applying the Householder's Method for $A^T A$, we have produced the

matrix $A^{(n-1)}$. This matrix is symmetric, tridiagonal and similar to A^TA. In the next step we will use the QR method to determine the eigenvalues of $A^{(n-1)}$ which are approximately the same as those of the matrix A^TA. This will be in the following section.

QR Method

As we have seen, the Householder's method was used first to determine the symmetric tridiagonal $A^{(n-1)}$ matrix. In this section we continue the procedure and use the QR method [33] to convert the matrix $A^{(n-1)}$ to a diagonal matrix that has approximately the same eigenvalues as the original matrix A^TA.

The idea of the QR method is to obtain a sequence of symmetric tridiagonal matrices

$$A^{(n-1)} = A^{(1)}, A^{(2)}, A^{(3)}, \ldots,$$

such that the matrix $A^{(i+1)}$ has the same eigenvalues as $A^{(i)}$. Moreover, it tends to a diagonal matrix and its diagonal entries are approximately the eigenvalues of $A^{(n-1)}$ as well as A^TA.
QR starts with the matrix

$$A^{(n-1)} = A = \begin{pmatrix} a_1 & b_2 & 0 & \cdots & 0 \\ b_2 & a_2 & \ddots & \ddots & \vdots \\ 0 & \ddots & \ddots & \ddots & 0 \\ \vdots & \ddots & \ddots & \ddots & b_n \\ 0 & \cdots & 0 & b_n & a_n \end{pmatrix}$$

If $b_2 = 0$ or $b_n = 0$, a_2 or a_n is the eigenvalue of the matrix A. If $b_j = 0$ for some $2 < j < n$, we split the matrix A into two small matrices and repeat the procedure for both of them individually. The small matrices are of the

following form:

$$\begin{pmatrix} a_1 & b_2 & 0 & \cdots & 0 \\ b_2 & a_2 & \ddots & \ddots & \vdots \\ 0 & \ddots & \ddots & \ddots & 0 \\ \vdots & \ddots & \ddots & \ddots & b_{j-1} \\ 0 & \cdots & 0 & b_{j-1} & a_{j-1} \end{pmatrix} \quad \textit{and} \quad \begin{pmatrix} a_j & b_{j+1} & 0 & \cdots & 0 \\ b_{j+1} & a_{j+1} & \ddots & \ddots & \vdots \\ 0 & \ddots & \ddots & \ddots & 0 \\ \vdots & \ddots & \ddots & \ddots & b_n \\ 0 & \cdots & 0 & b_n & a_n \end{pmatrix}$$

If there is no $b_j = 0$, we continue the procedure by producing the matrices $A^{(1)}, A^{(2)}, A^{(3)}, \ldots$ as follows:

1. The matrix $A^{(n-1)} = A^{(1)}$ is factored as a product $A^{(1)} = Q^{(1)}R^{(1)}$, where $Q^{(1)}$ is an orthogonal matrix and $R^{(1)}$ is an upper triangular matrix.

2. The matrix $A^{(2)}$ is defined as

$$A^{(2)} = R^{(1)}Q^{(1)}.$$

3. The rest of the matrices are defined as follows: $A^{(i)}$ is factored as $A^{(i)} = Q^{(i)}R^{(i)}$ and $A^{(i+1)} = R^{(i)}Q^{(i)}$, where $Q^{(i)}$ is an orthogonal matrix and $R^{(i)}$ is an upper triangular matrix.

The construction of the factors $Q^{(i)}$ and $R^{(i)}$ use the *rotation matrix* which is defined by this definition:

Definition 3.18.
A rotation matrix P differs from the identity in at most four elements. These four elements are of the form

$$p_{(ii)} = p_{(jj)} = \cos\theta \quad \textit{and} \quad p_{(ij)} = -p_{(ji)} = \sin\theta$$

for some θ and $i \neq j$

The angle θ can be chosen so that the product PA has zero entries at $(PA)_{ij}$. The factorization $R^{(1)}$ ia a product of $n-1$ rotation matrices $P_2, P_3, \ldots, P_n$ and $A^{(1)}$, i.e.,

$$R^{(1)} = P_n, P_{n-1}, \ldots, P_2 A^{(1)}.$$

QR starts by choosing the rotation matrix P_2 such that

$$p_{(11)} = p_{(22)} = \cos\theta_2 \quad and \quad p_{(12)} = -p_{(21)} = \sin\theta_2.$$

Where

$$\sin\theta_2 = \frac{b_2}{\sqrt{b_2^2 + a_1^2}} \quad and \quad \cos\theta_2 = \frac{a_1}{\sqrt{b_2^2 + a_1^2}}$$

The entries in the positions $(2,1)$ and $(1,4),\ldots,(1,n)$ in the matrix $A_2^{(2)} = P_2 A^{(1)}$ are zeros since

$$-a_1\sin\theta_2 + b_2\cos\theta_2 = \frac{-b_2 a_1}{\sqrt{b_2^2 + a_1^2}} + \frac{a_1 b_2}{\sqrt{b_2^2 + a_1^2}} = 0.$$

and the only entry in position $(1,3)$ may not equal to zero.
In general

$$A_k^{(1)} = P_k A^{(k-1)}$$

where the matrix P_k is chosen so that the entry in position $(k, k-1)$ are zero, and therefore, the $(k-1, k+1)$ entry becomes non zero. The construction of the matrices $A_k^{(1)}$ and P_{k+1} are of the following form

$$A_k^{(1)} = \begin{pmatrix} z_1 & q_1 & r_1 & 0 & \cdots & \cdots & \cdots & \cdots & 0 \\ 0 & \ddots & \ddots & \ddots & \ddots & & & & \vdots \\ 0 & \ddots & \ddots & \ddots & \ddots & \ddots & & & \vdots \\ \vdots & \ddots & 0 & z_{k-1} & q_{k-1} & r_{k-1} & \ddots & & \vdots \\ \vdots & & \ddots & 0 & x_k & y_k & 0 & \ddots & \vdots \\ \vdots & & & \ddots & b_{k+1} & a_{k+1} & b_{k+2} & \ddots & 0 \\ \vdots & & & & \ddots & \ddots & \ddots & \ddots & 0 \\ \vdots & & & & & \ddots & \ddots & \ddots & b_n \\ 0 & \cdots & \cdots & \cdots & \cdots & \cdots & 0 & b_n & a_n \end{pmatrix}$$

and

$$P_{k+1} = \begin{array}{|c|cc|c|} \hline \mathbf{I}_{k-1} & \multicolumn{2}{c|}{\mathbf{0}} & \mathbf{0} \\ \hline & c_{k+1} & s_{k+1} & \\ \mathbf{0} & & & \mathbf{0} \\ & -s_{k+1} & c_{k+1} & \\ \hline \mathbf{0} & \multicolumn{2}{c|}{} & \mathbf{I}_{n-k-1} \\ \hline \end{array}$$

where

$$s_{k+1} = \frac{b_{k+1}}{\sqrt{b_{k+1}^2 + x_k^2}} \quad and \quad c_{k+1} = \frac{x_k}{\sqrt{b_{k+1}^2 + x_k^2}}$$

and

$$A_{k+1}^{(1)} = \begin{pmatrix} z_1 & q_1 & r_1 & 0 & \cdots & \cdots & \cdots & \cdots & 0 \\ 0 & \ddots & \ddots & \ddots & \ddots & & & & \vdots \\ 0 & \ddots & \ddots & \ddots & \ddots & \ddots & & & \vdots \\ \vdots & \ddots & 0 & z_k & q_k & r_k & \ddots & & \vdots \\ \vdots & & \ddots & 0 & x_{k+1} & y_{k+1} & 0 & \ddots & \vdots \\ \vdots & & & \ddots & b_{k+2} & a_{k+2} & b_{k+3} & \ddots & 0 \\ \vdots & & & & \ddots & \ddots & \ddots & \ddots & 0 \\ \vdots & & & & & \ddots & \ddots & \ddots & b_n \\ 0 & \cdots & \cdots & \cdots & \cdots & \cdots & 0 & b_n & a_n \end{pmatrix}$$

By continuing this construction for the rotation matrices $P_1, \ldots, P_n$ we get the matrix

$$R^{(1)} \equiv A_n^{(1)} = \begin{pmatrix} z_1 & q_1 & r_1 & 0 & \cdots & 0 \\ 0 & \ddots & \ddots & \ddots & & \vdots \\ \vdots & \ddots & z & \ddots & \ddots & 0 \\ \vdots & & \ddots & \ddots & \ddots & r_{n-2} \\ \vdots & & & \ddots & z_{n-1} & q_{n-1} \\ 0 & \cdots & \cdots & \cdots & 0 & x_n \end{pmatrix}$$

which is an upper triangular matrix.

The construction of the other factor, which is $Q^{(1)}$, is defined by the following:

$$Q^{(1)} = P_2^\top P_3^\top \ldots P_n^\top.$$

where the matrix P is the rotation matrix that is defined in the factor $R^{(1)}$. Since the rotation matrices are orthogonal, then

$$Q^{(1)}R^{(1)} = (P_2^\top P_3^\top \ldots P_n^\top).(P_2 P_3 \ldots P_n)A^{(1)} = A^{(1)}.$$

and so, the matrix $A^{(1)}$ is equal $A^{(1)} = R^{(1)}Q^{(1)}$ which is closer to being a diagonal matrix than is $A^{(1)}$. This because the magnitude of the entries off diagonal of the matrix $A^{(2)}$ are, in general, smaller than the corresponding entries of the matrix $A^{(1)}$. We do the same procedure for other matrices $A^{(3)}, A^{(4)}, \ldots$.

Now, let $\lambda_i, i = 1, 2, \ldots, n$ be the eigenvalues of the matrix $A^{(n-1)}$. If $|\lambda_1| > |\lambda_2| > \ldots > |\lambda_n|$, the rate of convergence of $\frac{|\lambda_{j+1}|}{|\lambda_j|}$ determines the rate of convergence of the entry b_{j+1}^{i+1} to 0 in the matrix $A^{(i+1)}$. In addition, the rate of convergence of the entry $a_j^{(i+1)}$ to the eigenvalue λ_i is determined by the rate of convergence of b_{j+1}^{i+1} to 0. That is, if

$$b_{j+1}^{i+1} \to 0, \quad then \quad a_j^{(i+1)} \to \lambda_i$$

Usually this convergence is slow. We use a shift technique to make this convergence faster. A shift technique uses a constant s, which is called a shift constant s. This constant, s_i is determined at each step, say step i, by the eigenvalue of the matrix

$$E^{(i)} = \begin{pmatrix} a_{n-1}^{(i)} & b_n^{(i)} \\ b_n^{(i)} & a_n^{(i)} \end{pmatrix}$$

that is close to the entry $a_n^{(i)}$. After that, we use this constat, s_i to modify the factorization matrices $Q^{(i)}$ and $R^{(i)}$ so that

$$A^{(i)} - s_i I = Q^{(i)}R^{(i)} \quad and \quad A^{(i+1)} = R^{(i)}Q^{(i)} + s_i I.$$

QR algorithm in C++

In this program we obtain the eigenvalues of a symmetric, tridiagonal $n \times n$ A using the recursive function, where

$$A \equiv A_1 = \begin{pmatrix} a_1^{(1)} & b_2^{(1)} & 0 & \cdots & 0 \\ b_2^{(1)} & a_2^{(1)} & \ddots & \ddots & \vdots \\ 0 & \ddots & \ddots & \ddots & 0 \\ \vdots & \ddots & \ddots & \ddots & b_n^{(1)} \\ 0 & \cdots & 0 & b_n^{(1)} & a_n^{(1)} \end{pmatrix}$$

INPUT : Dimension n; $a_1^{(1)}, \ldots, a_n^{(1)}, b_1^{(1)}, \ldots, b_n^{(1)}$;
tolerance TOL; maximum number of iterations M

OUTPUT : Eigenvalues of A.

Step 1 Set $k = 1$; $SHIFT = 0$. (Accumulated shift)

Step 2 While $k \leq M$ do steps 3 - 19

Step 3 if $(n == 1)$

- $\lambda = a_1^{(k)} + SHIFT$
- Print out (λ)
- return (return from where you came)

Step 4 if $(|b_n^{(k)}| < TOL)$

- $\lambda = a_n^{(k)} + SHIFT$
- Print out (λ)
- $n = n - 1$
- *continue* (repeat steps starting from step 2)

Step 5 if $(|b_1^{(k)}| < TOL)$

- $\lambda = a_1^{(k)} + SHIFT$
- Print out (λ)
- $n = n - 1$
- $a_1^{(k)} = a_2^{(k)}$
- for $j = 2, ldots, n$
  set
  - $a_j^{(k)} = a_{j+1}^{(k)}$
  - $b_j^{(k)} = b_{j+1}^{(k)}$
- *continue* (repeat steps starting from step 2)

Step 6 for $j = 2, \ldots, n-1$

if $(|b_j^{(k)}| < TOL)$ (*Splitting case*) in this case we split the matrix A into two small matrices $\{a_1^{(k)}, \ldots, a_{j-1}^{(k)}, b_2^{(k)}, \ldots, b_{j-1}^{(k)}\}$ and $\{a_j^{(k)}, \ldots, a_n^{(k)}, b_{j+1}^{(k)}, \ldots, b_n^{(k)}\ SHIFT\}$ and we apply QR method for the first matrix and then for the second one.

Step 7 (*Computing the Shift*)

set

$$b = -(a_{n-1}^{(k)} + a_n^{(k)})$$
$$c = a_n^{(k)} a_{n-1}^{(k)} - [b_n^{(k)}]^2$$
$$d = (b^2 - 4c)^{1/2}$$

Step 8 If $(b > 0)$ then set

$$\mu_1 = \frac{-2c}{b+d}$$
$$\mu_2 = \frac{-(b+d)}{2}$$

else set

$$\mu_1 = \frac{d-b}{2}$$
$$\mu_2 = \frac{2c}{(d-b)}$$

Step 9 If $(n == 2)$ then

- set

$$\lambda_1 = \mu_1 + SHIFT$$
$$\lambda_2 = \mu_2 + SHIFT$$

- Print out (λ_1, λ_2)
- return (return from where you came)

Step 10 Choose s so that $s = \min\{|\mu_1 - a_n^{(k)}|, |\mu_2 - a_n^{(k)}|$

Step 11 Set $SHIFT = SHIFT + s$

Step 12 *(Perform shift)*

For $j = 1, \ldots, n$ set $d_j = a_j^{(k)} - s$

Step 13 (Steps 14 and 15 compute $R^{(k)}$)

$$x_1 = d_1$$
$$y_1 = b_2^{(k)}$$

Step 14 For $j = 2, \ldots, n$

- set

$$z_{j-1} = \left[x_{j-1}^2 + (b_j^{(k)})^2\right]^{1/2}$$
$$c_j = \frac{x_{j-1}}{z_{j-1}}$$
$$s_j = \frac{b_j^{()k}}{z_{j-1}}$$
$$q_{j-1} = c_j y_{j-1} + s_j d_j$$
$$x_j = -s_j y_{j-1} + c_j d_j$$

- If $j \neq n$ then set

$$r_{j-1} = s_j b_{j+1)}^{(k)}$$
$$y_j = c_j b_{j+1)}^{(k)}$$

($A_{j)}^{(k)} = P_j A_{j-1)}^{(k)}$ has just been computed and $R^{(k)} = A_n^{(k)}$)

Step 15 *(Steps 16 - 18 compute $A^{(k+1)} = R^{(k)}Q^{(k)}$)* set

$$z_n = x_n$$
$$a_1^{(k+1)} = s_2 q_1 + c_2 z_1$$
$$b_1^{(k+1)} = s_2 z_2$$

Step 16 For $j = 2, \ldots, n-1$ set

$$a_j^{(k+1)} = s_{j+1} 2 q_j + c_j c_{j+1} z_1$$
$$b_{j+1}^{(k+1)} = s_{j+1} z_{j+1}$$

Step 17 Set $a_n^{(k+1)} = c_n z_n$

Step 18 Set $k = k + 1$

Chapter 4

MAXIMUM CLIQUE PROBLEM (MCP)

The maximum clique problem is one of the most important problems in combinatorial optimization. In this chapter we briefly discuss the maximum clique problem including its applications and the algorithms that can solve the problem. Moreover, we will present an important application of the (MCP) called finding ovoids in finite polar spaces. In addition, we develop a new technique based on the local search algorithm to find cliques of a given size in agiven graph. To understand clearly the problem let us start with the following definitions.

4.1 Definitions

In this section we provide the definitions that help us understand the maximum clique and independent set problems and how they are related. Also, we recall some definitions from previous chapters.

Definition 4.1. Let $G = (V, E)$ be an undirected graph where V is the set of vertices and $E \subseteq V \times V$ is the set of edges. The graph G is *complete* if for every two distinct vertices a_i, a_j there is an edge $(a_i, a_j) \in E$. A graph $\bar{G} = (\bar{V}, \bar{E})$ is called a *subgraph* of the graph G if $\bar{V} \subseteq V$ and $\bar{E} \subseteq E$ such that if $(a_i, a_j) \in \bar{E}$, then $a_i, a_j \in \bar{V}$.

Definition 4.2. A subgraph $C = (V_c, E_c)$ of a graph $G = (V, E)$ is called a *clique* if it is complete. That is, for every two vertices $a, b \in V_c$, there is an edge $(a, b) \in E_c$. The *size* of a clique C is denoted as $\omega(c)$ or $|C|$ and defined by the number of vertices contained in V_c.

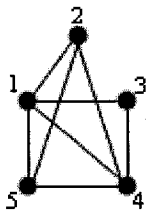


Figure 4.1: Graph $G_1 = (V, E)$, $V = \{1, \ldots, 5\}$, of example 4.1

The maximum clique problem involves finding a clique C in a given graph such that it has the maximum size. In other words, given a graph G, we want to find a clique C such that $|C| \geq |\bar{C}|$ for any clique $\bar{C}$ in G.

Example 4.1. In figure 4.1, we have a graph $G_1 = (V, E)$ where $V = \{1, \ldots, 5\}$. Graph G_1 is not complete since there are at least two vertices not connected with an edge; for instance, vertex 2 and 3, are not adjacent. In the following subgraphs

1. $C_1 = (V_{c_1}, E_{c_1})$, where $V_{c_1} = \{1, 3, 5\}$ and $E_{c_1} = \{(1, 3), (1, 5)\}$.
2. $C_2 = (V_{c_2}, E_{c_2})$, where $V_{c_2} = \{1, 3, 4\}$ and $E_{c_2} = \{(1, 3), (1, 4), (3, 4)\}$.
3. $C_3 = (V_{c_3}, E_{c_3})$, where $V_{c_3} = \{1, 2, 4, 5\}$ and
   $E_{c_3} = \{(1, 2), (1, 4), (1, 5), (2, 4), (2, 5), (4, 5)\}$.

we have the following: the subgraph C_1 is not a clique since not all its vertices are pairwise adjacent, i.e., the edge $(1, 5) \notin E_{c_1}$. The subgraphs C_2 and C_3 are cliques of size 3 and 4 respectively. The maximum clique of this graph is C_3 since it has the maximum size.

Definition 4.3. A set S in a graph $G = (V, E)$ is called an *independent set*, if it is a subset of V such that every two vertices in S are nonadjacent. In other words, if $S \subseteq V$ such that $\forall \ \ a, b \in S$, the edge $e = (a, b) \notin E$. The size of an independent set S is the number of vertices contained in S and is denoted as $\omega(S)$ or $|S|$.
Finding the maximum independent set in a given graph is called the maximum independent set problem. That is, we need to find an independent set $S \subseteq V$ in a graph $G = (V, E)$ such that $|\bar{S}| \leq |S|$ for all independent sets $\bar{S}$ in G.

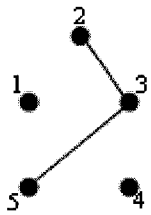


Figure 4.2: Graph $G_2 = (V, E)$, $V = \{1, \ldots, 5\}$, of example 4.2

Example 4.2. In figure 4.2, we have a graph $G_2 = (V, E)$ where $V = \{1, \ldots, 5\}$. The set $S_1 = \{1, 3, 5\}$ is not an independent set since there are two vertices $3, 5 \in S_1$ which are connected in G_2, i.e., the edge $e = (3, 5) \in \bar{E}$. The sets $S_2 = \{1, 3, 4\}$ and $S_3 = \{1, 2, 4, 5\}$ are independent sets and have size 3 and 4 respectively. The set S_3 is the maximum independent set of this example since it has the largest size.

Definition 4.4. Given a graph $G = (V, E)$ where V is the set of vertices and E is the set of edges, the *complement* of a graph G is a graph $\bar{G} = (\bar{V}, \bar{E})$ such that the two following conditions are satisfied:

1. The set of vertices $\bar{V}$ of $\bar{G}$ is the same as the set of vertices V of graph G, i.e., $\bar{V} = V$

2. Two vertices in $\bar{G}$ are connected if and only if they are not connected in G.

For example, graph G_2 in figure 4.2 is the complement of graph G in figure 4.1.

It is not hard to see that the set C is an independent set of a graph G if and only if C is a clique of its complement [40].

Therefore, finding the maximum independent set in a graph G corresponds to finding the maximum clique in its complement. For example, the maximum independent set S_3 of graph G_2 in example 4.2, is the maximum clique of the graph G_1 in example 4.1, which is the complement of the graph G_2.

4.2 Applications

The maximum clique problem arises in various areas including computer vision, cluster analysis, and coding theory [27, 22, 12].
For example, in coding theory, one question which arises is to compute $A(n,d)$ which is the maximum number of binary vectors of size n with Hamming distance d. The Hamming distance between two codewords $u(u_1,\ldots,u_n)$ and $v(v_1,\ldots,v_n)$ is the number of coordinates where they differ, and is denoted by $d(u,v)$. The idea is to determine the graph $G=(V,E)$ where the set of vertices V has size 2^n and corresponds to all possible codewords, while E is all possible edges such that two vertices are connected with an edge $e \in E$ if their Hamming distance is at least d. The value of $A(n,d)$ can be computed by determining the maximum clique of the graph G.

Another problem in coding theory involves finding a weighted binary code $A(n,w,d)$. This value is determined by finding the maximum clique corresponding to the graph $G=(V,E)$ where the size of the set V is $\binom{n}{w}$. For more details regarding these applications see [22, 2].

Yet another interesting application of the maximum clique problem is finding ovoids in finite polar spaces. This will be described in the following section.

4.2.1 Finding ovoids in finite polar space

One important application for the maximum clique problem is searching for ovoids in finite polar spaces, which is equivalent to finding an independent set with a specific size in a specific graph. To understand this application, the following definitions are necessary.

Definition 4.5. The *n-dimensional projective space* over a finite field $GF(q)$, denoted $PG(n,q)$ is defined by means of an $(n+1)-$dimensional vector space $V(n+1,GF(q))$. The 1-dimensional subspaces of V are the points, the 2-dimensional subspaces are the lines and the 3-dimensional subspaces of V are the planes [11].

There are three important properties of projective space, which are given as follows:

1. Two points lie on at most one line.

2. Two intersecting lines lie in a unique plane.

3. Two lines in the same plane must intersect.

Definition 4.6. A *quadratic form* in indeterminates $x_0, \ldots, x_n$ over a finite field $\mathbb{F}$ is a homogeneous polynomial of degree two in those indeterminates,.i.e.,

$$Q(X) = Q(x_0, \ldots, x_n) = \sum_{i,j=0}^{n} (c_{i,j} x_i x_j)$$

Where the coefficients c_{ij} are in $\mathbb{F}$ [48].

For $x \in GF(q)$, we write $P(x)$ to denote the projective point which is the one-dimensional subspace spanned by x. From now on, q will always be a prime power (i.e., $q \in \{2, 3, 2^2, 5, 7, \ldots\}$).

Definition 4.7. Let Q be a quadratic form in $(n+1)$ variables. A *quadric* $\hat{Q}$ in $PG(n, q)$ is the zero set of Q. That is,

$$\hat{Q} = \{P(\mathbf{x}) : Q(\mathbf{x}) = 0\}$$

Let f and g be two quadratic forms. We say that f and g are *projectively equivalent* if g can be obtained from f using an invertible linear substitution of the variables. That is, in matrix form, f and g are projectively equivalent if there is a nonsingular $n \times n$ matrix A such that

$$g(X) = f(XA).$$

Given a quadratic form Q in n variables $x_0, \ldots, x_n$ over a field $\mathbb{F}$, then Q has *rank* r if r is the least number of variables that occur in any projective equivalent quadratic form [48].

Definition 4.8. Let Q be a quadratic form in r variables. A quadric $\hat{Q}$ is call *nondegenerate* if Q has degree r. That is, if Q cannot be transformed into a homogeneous polynomial with fewer degrees.

The nondegenerate quadrics in $PG(n,q)$ can be put into three types up to projective equivalence. These types are as follows (we let n be the dimension of the projective space):

1. Parabolic quadrics: The formula of this type is given by

$$Q(n,q) = x_0x_1 + x_2x_3 + \ldots + x_{2m-2}x_{2m-1} + ax_{2m}^2.$$

where $n = 2m$ is even and a is a non-zero scalar.

2. Hyperbolic quadrics: This is presented by

$$Q^+(n,q) = x_0x_1 + x_2x_3 + \ldots + x_{2m-2}x_{2m-1}.$$

where $n = 2m - 1$ is odd.

3. Elliptic quadrics: The formula of this type is given by

$$Q^-(n,q) = x_0x_1+x_2x_3+\ldots+x_{2m-2}x_{2m-1}+ax_{2m}^2+bx_{2m}x_{2m+1}+cx_{2m+1}^2.$$

with $ax^2 + bx + c$ irreducible over the field $\mathbb{F}_q$. The number n in this type is odd and equals $2m + 1$, i.e., $n = 2m + 1$.

where m is an integer called the *witt*-index, which is the largest dimension of a vector subspace in which the quadratic form vanishes completely.

For example, Figure 4.3 shows the hyperbolic quadric $\hat{Q}^+(3,q)$ with equation $x_0x_1 + x_2x_3 = 0$, Cameron [10].

A line of a quadric $\hat{Q}$ is a projective line (i.e., a 2-dimensional subspace) which is completely contained in the quadric. Let $\hat{Q}_l$ denote the lines of $\hat{Q}$.

There are formulas of the number of projective points and the number of projective lines for each type of nondegenerate quadric $\hat{Q}$ in $PG(n,q)$. These formulas are as follows:

1. The parabolic quadric $\hat{Q}(n,q)$, where $n = 2m$ and is even:

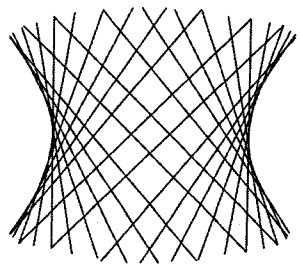


Figure 4.3: The hyperbolic quadric $\hat{Q}^+(3,q)$ with equation $x_0x_1 + x_2x_3 = 0$, which is called "ruled quadric", Cameron [10].

- The number of projective points:

$$|\hat{Q}| = \frac{q^n - 1}{q - 1}. \tag{4.1}$$

- The number of projective lines:

$$|\hat{Q}_l| = \frac{(q^m - 1)(q^{m-1} - 1)}{(q-1)(q^2-1)}.(q^m + 1)(q^{m-1} + 1). \tag{4.2}$$

2. The hyperbolic quadric $\hat{Q}^+(n,q)$, where $n = 2m - 1$ and is odd:

- The number of projective points:

$$|\hat{Q}^+| = \frac{(q^{(n+1)/2} - 1)(q^{(n-1)/2} + 1)}{q - 1}. \tag{4.3}$$

- The number of projective lines:

$$|\hat{Q}^+_l| = \frac{(q^m - 1)(q^{m-1} - 1)}{(q-1)(q^2-1)}.(q^{m-1} + 1)(q^{m-2} + 1). \tag{4.4}$$

3. The elliptic quadric $\hat{Q}^-(n,q)$, where $n = 2m + 1$ and is odd:

- The number of projective points:

$$|\hat{Q}^-| = \frac{(q^{(n+1)/2} + 1)(q^{(n-1)/2} - 1)}{q - 1} \tag{4.5}$$

Definition 4.9. An *ovoid* **O** of a quadric $\hat{Q}(n,q)$ in $PG(n,q)$ is a set of s points such that there are no two collinear in a line of the quadric, where

$$s = \begin{cases} q^m + 1, & \text{if } \hat{Q} \text{ is parabolic, i.e., } \hat{Q}(n,q), \text{ and } n = 2m; \\ q^{m-1} + 1, & \text{if } \hat{Q} \text{ is hyperbolic, } \hat{Q}^+(n,q), \text{ and } n = 2m-1; \\ q^{m+1} + 1, & \text{if } \hat{Q} \text{ is elliptic, } \hat{Q}^-(n,q), \text{ and } n = 2m+1; \end{cases} \tag{4.6}$$

For example, the size of an ovoid of a hyperbolic quadric $\hat{Q}^+(3,q)$ is $q+1$ (since in this case $3 = n = 2m-1$, we have $m = 2$ and $s = q^{(2-1)}+1$).

In order to describe the search for ovoids as an instance of the maximum clique problem, we define the collinearity graph of a polar space.

Definition 4.10. Let $Q = \hat{Q}(n,q)$ be a nondegenerate quadric in $PG(n,q)$. Let Q_l be the set of lines of Q. The *collinearity graph* of Q has as vertices the points of Q. Two vertices are adjacent if the corresponding points are collinear in a line of Q_l.

Theorem 4.2.1. *The following are equivalent:*

1. **O** *is an ovoid in* $Q(n,q)$.
2. **O** *is an independent set of size* s *in the collinearity graph of* Q.
3. **O** *is a clique of size* s *in the complement of the collinearity graph of* Q.

where s *is as in formula 4.6.*

Therefore, the search for ovoids in polar spaces is an instance of the maximum clique problem.

Example 4.3. Figure 4.3 shows the hyperbolic quadric $\hat{Q}^+(3,q)$, Cameron [10]. An ovoid of this quadric has $q+1$ points. For instance, if $q = 2$, i.e., $\hat{Q}^+(3,2)$, then we have an ovoid of size 3. The number of points and lines of the quadric $\hat{Q}^+(3,2)$ are 9 and 6 respectively (see Figure 4.4-A). Therefore, the corresponding collinearity graph has 9 vertices and is shown in Figure 4.4-B. In this graph, two vertices are adjacent if they are connected by a line in $\hat{Q}^+(3,2)$. For instance, since vertex 1 is connected with vertex 2 and vertex 3 by a line in $\hat{Q}^+(3,2)$, these points are connected by edges in the collinearity graph. The complement of the collinearity graph is shown in Figure 4.4-C. All possible cliques of size 3 of this graph are $C_1\{1,5,9\}, C_2\{1,6,8\}, C_3\{2,4,9\}, C_4\{2,7,6\}, C_5\{3,4,8\}$ and $C_6\{3,5,7\}$. By theorem 4.2.1, these cliques are independent sets in the collinearity graph and are ovoids in the quadric $\hat{Q}^+(3,2)$.

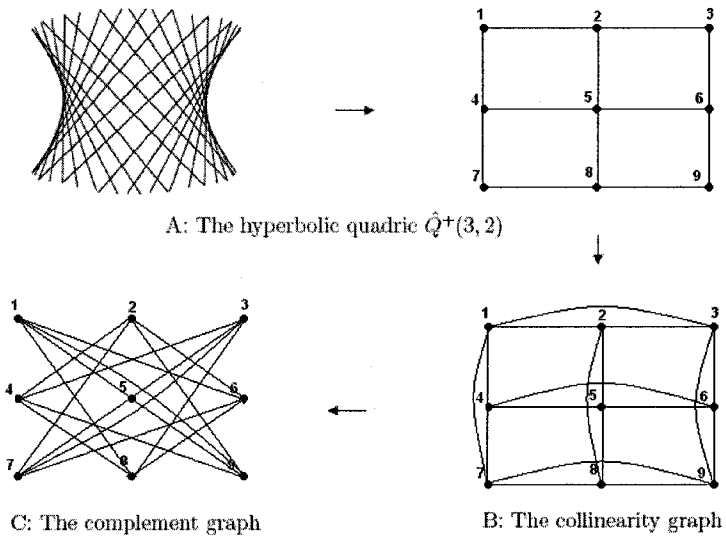


Figure 4.4: The hyperbolic quadric $\hat{Q}^+(3,2)$ and its collinearity graph and the complement of the collinearity. The details of this figure is described in example 4.3.

In the following section, we will present algorithms which are used in combinatorial optimization to solve this problem.

4.3 Algorithms

Much research has been done on the maximum clique problem and various algorithms have been developed to solve it. For example,(Johnson, D. S. [29, 22] used the greedy algorithm to solve the maximum clique problem. This algorithm is based on several iterations. It starts by choosing a vertex that has the maximum degree and adds at each iteration a new vertex to the current clique so that the result is also a clique. This algorithm continues adding vertices until no more vertices can be added.
Other examples of algorithms used for solving MCP are the enumerative algorithms including Harary and Ross [21], Maghout [35], and Paull and Unger [41]. The details regarding the enumerative algorithm can be found in [2].
One of the more interesting algorithms used for MCP is the one based on the local search algorithm e.g. the reactive local search for the maximum clique

problem by Roberto Battiti and Marco Protasi [3]. This algorithm solves the maximum clique problem based on the local search. Another example of an algorithm that is based on the local search algorithm is the k-opt local search for solving MCP by Kengo, Kihiro, and Hiroyuki [30]. This algorithm searches for a k-opt neighborhood using several 1-opt moves. The 1-opt move is either an add or a drop move. The add move is when a possible vertex is added to the current clique, whereas the drop move is when one vertex is dropped from the current clique. The details of this algorithm can be found in [30].

In the next section we develop a new technique for solving the maximum clique problem. This technique is based on the local search algorithm.

4.3.1 New Technique for Solving MCP Based on Local Search Algorithm

In this section we develop a new technique for solving the maximum clique problem. The technique is based on the local search algorithm (see Section 2.3). Before discussing our technique, let us state the following. Given a graph $G = (V, E), V = \{1, \ldots, n\}$ is the set of vertices and E is the set of edges represented by a $\{0,1\}$ $n \times n$ matrix $A[e_{ij}]$, called the adjacency matrix. That is, if $e_{ij} \in E$, then we let the entry $e_{ij} = 1$ and 0 otherwise. An instance of the maximum clique problem is a pair (F, c), where F is the set of feasible solutions. This set considers all subgraphs which are cliques in the graph G. That is,

$$\begin{aligned} F &= \{\text{the set of all clique in the graph } G\} \\ &= \{y : \text{y is a clique in } G\}. \end{aligned}$$

Whereas, c represents the cost function, which is defined by the size of the clique $y \in F$. That is, cost of a feasible solution $y \in F$ given by

$$c(y) = |y| = \{\text{the number of vertices contained in the clique } y\}.$$

The problem is to find a clique $y \in F$ such that this clique has the maximum (or close to the maximum) size. In other words, we are looking for a

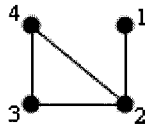


Figure 4.5: Graph $G = (V, E)$, where $V = \{1, \ldots, 4\}$, of example 4.4.

feasible solution $y \in F$ such that $c(y) \geq c(g)$ for all cliques $g \in F$.

Example 4.4. In the graph in Figure 4.5, the set of feasible solutions F is as follows:

$$F = \{\phi, \{1\}, \{2\}, \{3\}, \{4\}, \{1,2\}, \{2,3\}, \{2,4\}, \{3,4\}, \{2,3,4\}\}$$

The cost of the clique $y_1 = \{3\}$ and $y_2 = \{2,4\}$ is $c(y_1) = 1$ and $c(y_2) = 2$ respectively, while the cost of $y_3 = \{2,3,4\}$ is $c(y_3) = 3$. Clearly, the maximum clique of this graph is given by the feasible solution y_3 since it has the maximum cost.

In the following section we introduce a definition of the neighborhood of the maximum clique problem.

4.3.2 The Neighborhoods of MCP

In the maximum clique problem we may develop three neighborhoods of a given feasible clique $y \in F$. These neighborhood are called add, exchange, and remove neighborhoods and are given by the following definition.

Definition 4.11. Given a graph $G = (V, E)$ with n vertices where V is the set of vertices and E is the set of edges, the neighborhoods of a feasible solution $y \in F$ are

1. *Add*-neighborhood: This neighborhood is defined as
   $N_a(y) = \{g : g \in F$ and g can be obtained from y by adding one vertex to $y.\}$
   This neighborhood can be determined by the set of all vertices in $V \backslash y$ that connect to all vertices of y. That is, $g \in N_a(y)$ if there is a

vertex $u \in V\backslash y$ such that $u \sim b$, $\forall\ b \in y$ where $u \sim b$ means vertex u is adjacent to vertex b. Therefore, $g = y \cup \{u\}$. For example, in the graph in Figure 4.5, the clique $g = \{1, 3, 4\}$ belongs to the add-neighborhood of y, $N_a(y)$, where $y = \{1, 3\}$. This is because vertex 4 connects to vertex 1 and 3.

2. *Exchange*-neighborhood: This neighborhood is defined as
   $N_e(y) = \{g : g \in F$ and g can be obtained from y by exchanging one vertex in y with one vertex in $V\backslash y.\}$
   This neighborhood can be determined by the set of all vertices in $V\backslash y$ that are adjacent to all vertices in $y\backslash\{u\}$ for some $u \in y$. In other words, $g \in N_e(y)$ if $g = y\backslash\{u\} \cup \{a\}$ where $a \in V\backslash y$ and $a \sim b$, $\forall\ b \in y\backslash\{u\}$. For instance, in the graph in Figure 4.5, let $y \in F$ be a clique $y = \{1, 3, 4\}$. Since vertex 2 is adjacent to all vertices except 3, we can exchange vertex 3 with 2 to obtain the clique $g = \{1, 2, 4\}$ which belongs to the exchange-neighborhood of y, $N_e(y)$.

3. *Remove*-neighborhood: This neighborhood is given by
   $N_r(y) = \{g : g \in F$ and g can be obtained from y by removing one vertex from $y.\}$

Now we discuss our technique for solving the maximum clique problem.

Given a graph $G = (V, E)$ of size n, assume that we need to find all possible cliques of size s. The steps for finding these cliques are as follows:

Step 1: Select a starting point $y \in F$. We select a completely random clique y as the starting point. The starting point can be chosen by randomly selecting any vertex $v \in V$. This is because any vertex in V is a clique of size 1.

Step 2: Add a new vertex, if possible, to the current clique y. In this step, we search for any feasible solution $g \in N_a(y)$. That is, we search randomly for a vertex $v \in V\backslash y$ such that $v \sim b$, $\forall b \in y$.

Step 3: Repeat step 2 until no more vertices can be added to the current clique. In this step, we continue adding a new vertex to the current clique y until one of the following conditions is satisfied:

1. The size of the current clique reaches the value s, which is the size of the maximum clique.
2. No more vertices can be added to the current clique and the size of the current clique is less then s. That is, the set of the add-neighborhood of y, $N_a(y)$, is empty.

Step 4-i: If condition 1 is satisfied, remove one vertex from the current clique and go to step 2. This step follows step 3 if condition 1 ($|y| = s$) holds. This means that the maximum clique has been found and therefore we search for another clique that has the maximum size s. In this case, we remove one vertex from the current clique y and go to step 2 to search for another clique.

Step 4-ii: If condition 2 is satisfied, exchange one vertex, if possible, from the current clique with a new vertex in the set $V\backslash y$ so that the result is a clique, and go to step 2. This step follows step 3 if it is impossible to add a vertex in the current clique. In this case, we randomly choose a vertex $v \in V\backslash y$ such that $v \sim b$ for all $b \in y\backslash\{a\}$ for some $a \in y$. Therefore, we exchange vertex v with vertex a. The result must be a clique and belong to the exchange-neighborhood of y, $N_e(y)$. After that, we go to step 2 to add a new vertex to the resulting clique.

Step 5: Remove one vertex from the current clique and go to step 2. If it is impossible to add or exchange a vertex to the current clique, we remove one vertex from the current clique and go to step 2 to add vertices to the resulting clique.

Step 6: Stop when all possible maximum cliques are found. The stopping rule of the algorithm is when all possible maximum cliques are found or the current clique cannot be improved.

The pseudo code of the algorithm for finding the cliques of a size in a given graph is as follows:

Algorithm: Local search algorithm for finding all possible cliques of a given size in a given graph.

Input:

1. The set of vertices $v = \{1, \ldots, n\}$
2. The $n \times n$ adjacency matrix $Adj[e_{ij}]$
3. The size, s, of the cliques that we are looking for

Output: All possible cliques $y_1, y_2, \ldots, y_t$ of size s.

Definition of the following sets:

- $y :=$ the current clique.
- $A := \{a : a \in V \backslash y \ \ and \ \ a \sim b, \ \ \forall\, b \in y\}$.
- $E := \{e : e \in V \backslash y \ \ and \ \ e \sim b, \ \ \forall\, b \in y, \text{ for some } v \in y\}$.
- $R := \{r : r \in y \quad \text{and } y \text{ is the current clique}\}$.

begin

1. $y := \{v\}$ where $v \in V$ is chosen randomly as a starting point;
2. $t := 0$; (The number of cliques of size s that have been found so far.)
3. **repeat**
4. **if** $(A \neq \emptyset)$ **do**
5. $a := a \in A$ and a is chosen randomly;
6. $y := y \cup \{a\}$;
7. **if** $(|y| = s)$ **do**
8. **if** $(y \ is \ new \ )$ **do**
9. $y_t = y$;

10. $t = t + 1$;
11. **endif**
12. $r := r \in R$ and r is chosen randomly;
13. $y := y \backslash \{r\}$;
14. **goto** line 4.
15. **else** " (if $|y| \neq s$) " **goto** line 4
16. **else** " (if $A = \emptyset$) "
17. **if** $(E \neq \emptyset)$ **do**
18. $e := e \in E$ and e is chosen randomly;
19. $v := v \in y$ and $e \sim b$, $\forall\, b \in y \backslash \{v\}$;
20. $y := y \cup \{a\}$;
21. **goto** line 4.
22. **else** " (if $E = \emptyset$)
23. $r := r \in R$ and r is chosen randomly;
24. $y := y \backslash \{r\}$;
25. **goto** line 4.
26. **until** No more cliques of size s can be found;
27. **end**

4.4 Results

Since we are interested in finding ovoids in quadrics $\hat{Q}$ in $PG(n, q)$ with prime power q, we use these quadrics as instances to evaluate the performance of our new algorithm for finding cliques of a given size in a given graph. These include graphs which are the complement of the collinearity graphs corresponding to both the hyperbolic quadrics $\hat{Q}^+(n, q)$ with $n = 2m$ and parabolic quadrics $\hat{Q}(n, q)$ with $n = 2m-1$, where $m \geq 2$.

The graphs are constructed using a program called ovoid.cpp which was developed by Dr. Anton Betten. This program takes parameters ϵ, n, q as the input where the $\epsilon = 0, 1$ or -1 represents the parabolic quadrics $\hat{Q}(n, q)$, hyperbolic quadrics $\hat{Q}^+(n, q)$, and elliptic quadrics $\hat{Q}^-(n, q)$ respec-

tively, where the parameters n and q are described as above. The output of this program has a lot of information related to the ovoid, the quadric $\hat{Q}$, and the complement of its corresponding collinearity graph including the number of vertices and the adjacency matrix of this graph. Moreover, this program provides the size of cliques, which is the size of the ovoids of the quadric..

Table [4.1], shows comparative results of our algorithm applied to various graphs which are the complement of collinearity graphs which correspond to quadrics. The algorithm is applied to the instances of the hyperbolic quadrics $\hat{Q}^+(n,q)$ and the parabolic quadrics $\hat{Q}(n,q)$. For each instance we indicate the name of the quadric $\hat{Q}$ (the second column), the number of vertices $|\hat{Q}|$ of the complement of the collinearity graph corresponding to the quadric $\hat{Q}$ (the third column), the size of the clique s that we need to find (the fourth column), and the running time (Time), in seconds, till the first clique is found (the last column).

	Instance	$\lvert\hat{Q}\rvert$	s	Time
1	$\hat{Q}(4,7)$	400	50	0
2	$\hat{Q}(4,8)$	585	65	1
3	$\hat{Q}(4,9)$	820	82	15
4	$\hat{Q}^+(5,4)$	357	17	0
5	$\hat{Q}^+(5,5)$	806	26	0
6	$\hat{Q}^+(5,7)$	2850	50	20
7	$\hat{Q}^+(7,2)$	139	9	0.00
8	$\hat{Q}^+(7,3)$	1120	28	5
9	$\hat{Q}^+(7,4)$	5525	65	138

Table 4.1: Computational results of the new technique, which is based on the local search algorithm, for finding cliques of a size s, where s is as in formula 4.6, in the collinearity graph corresponding to quadrics $\hat{Q}$. (Times in seconds).

From Table [4.1], we can see the algorithm found ovoids of small instances of quadrics $\hat{Q}$ in $PG(n,q)$. For example, the algorithm found ovoids of the

parabolic quadrics $\hat{Q}(4,q)$ for $2 \leq q \leq 9$ (rows 1-3). The largest number of q of this quadric that the algorithm solved is 9, i.e., $\hat{Q}(4,9)$ with a running time of 15 seconds (row 3). The number of points of this quadric is 820 and the size of the ovoid is 82. The algorithm also found ovoids for the hyperbolic quadrics $\hat{Q}^+(5,q)$ for $2 \leq q \leq 7$ (rows 4-6). The running time needed for the algorithm to solve the instance of this quadric where $q = 7$, i.e. $\hat{Q}^+(5,7)$ (row 6), which is the largest number of q in this quadric, is 20 seconds. This instance has has 2850 points and the size of ovoid is 50. In addition, the algorithm found ovoids for the hyperbolic quadrics $\hat{Q}^+(7,q)$ for $2 \leq q \leq 4$ (rows 7-4). The largest number of q of this quadric that the algorithm solved is 4, i.e., $\hat{Q}^+(7,4)$ with a running time of 138 seconds (row 6). The number of points of this quadric is 5525 and the size of the ovoid is 65.

From these computational results, we conclude that our algorithm, which is based on the local search algorithm can find ovoids in small polar spaces.

Chapter 5

CONCLUSIONS

Combinatorial optimization consists of operations that search for one or more solutions to an optimization problem. We have discussed briefly a number of combinatorial optimization problems and the algorithms for solving these problems. Examples of these problems are the traveling salesman problem, minimal spanning tree, shortest path problem and maximum network flow. In addition, we provided discussion about some significant combinatorial optimization algorithms including greedy algorithms and local search algorithms. The properties and the characters of each algorithm are also provided. Example of the greedy algorithms are Kruskal's, Prim's, Dijkstra's, Floyd's, and Ford-Fulkerson's.

We have introduced the domino portrait problem and converted it to a combinatorial optimization problem. A domino portrait is an image which is constructed from complete sets of dominoes. These dominoes are arranged in a matrix, creating an approximation image when seen from a distance. The problem is to create a domino portrait of dimensions $M \times N$ from s^2 complete sets of double $(D-1)$ dominoes where $M = s(D+1)$ and $N = sD$. In this, we have used one of the powerful methods of image processing called two-dimensional wavelet transform.

In addition, we have solved the domino portrait problem using the greedy algorithm and the local search algorithm. Because of the search strategy that the greedy algorithms use, they usually usually get blocked and cannot find complete solutions to problems. To avoid such blocks in

the DPP, we have developed a technique called the one-factor search technique. This technique allows us to apply the greedy algorithm without its being blocked and to solve the domino portrait problem. Moreover, we have used the local search algorithm to solve one instance of the domino portrait problem. Using these algorithms, we can create an $M \times N$ domino portrait constructed from s^2 complete sets of double $(D-1)$ dominies. For example, we created domino portraits of Marilyn Monroe and George W. Bush, each of which was constructed from 9 complete sets of double nine dominoes (see Figure 3.32).

Since we are restricted to using a specific number of dominoes, we need to place them in important positions in the image, such as eyes, nose, and mouth. In this, we have developed a new cost function by modifying the old cost. This function forces our methods, the greedy algorithm and local search algorithm, to start filling the dominoes in the important positions in the image first. The new cost function is obtained by developing a matrix called the support matrix. This matrix is determined using a method called the singular value decomposition. Finally, we have discussed briefly about the Householder's and QR methods and used them to compute the singular value decomposition.

The C++ language is used in all our computations. We developed a C++ program that solved the domino portrait problem using the greedy algorithm and the local search algorithm.

By successfully applying the greedy algorithm and the local search algorithm to solve the DPP we have illustrated the usefulness of these algorithms in new arenas. In particular, by solving the problem of blockage faced by the greedy algorithm, we have proved that this algorithm has good potential. It is possible that further research will yield more uses for these algorithms.

The second combinatorial optimization problem addressed in this paper is the Maximum Clique Problem. The local search algorithm was used to find cliques of a given size in a graph. We used the relation between the

maximum clique problem and the independent set problem to find ovoids of quadrics in $PG(n, q)$ with prime power q.

The successful solution of several instances of this problem by the local search algorithm has shown the versatility of this tool. Although we were able to find ovoids in small polar spaces, perhaps the lessons learned in this research will enable other researchers to improve the results in graphs of larger size.

Bibliography

[1] Jose Luis Ambite; Craig A. Knoblock, *Planning by Rewriting*, Journal of Artificial Intelligence Research 15 (2001) 207-261.

[2] E. Balas; V. Chvfital; J. Ne etril, *On the Maximum Weight Clique Problem*, Math. of Operations Research 12 (3), 522-535, (1987).

[3] R. Battiti; M. Protasi, *Reactive Local Search for the Maximum Clique Problem*, Printed in: Algorithmica, (2001), vol. 29, no. 4, 610-637.

[4] John E. Beasley, *Branch and cut algorithms*, The Management School, Imperial College, London SW7 2AZ, England.

[5] Anton Betten *Twisted Tensor Product Codes and a Review of BLT Sets*, Combinatorics Seminar, Colorado State University. 27 Oct. 2006.

[6] J. Ross Beveridge, *Local Search Algorithm for Geometric Object Recognition: Optimal Correspondence and Pose*, Ph.D., University of Massachusetts (1993).

[7] Robert Bosch, *Constructing Domino Portrait*, Department of Mathematics, Oberlin, OH USA, 44074.

[8] Coen Bron; Joep Kerbosch, *Algorithm 457 Finding All Cliques of an undirected Graph*, Communications of the ACM. Sept. 1973: 16-9.

[9] Richard L. Burden; Douglas J. Faires, *Numerical Analysis*, Brooks/Cole, 511 Forest Lodge Road, Pacific Grove, CA 93950 USA.

[10] Peter J. Cameron, *Projective and Polar Spaces*, The School of Mathematical Sciences, Queen Mary and Westfield College (University of London), Mile End Road, London E1 4NS,(1991) U.K.

[11] Peter J. Cameron, *Combinatorics: Topics, Techniques and Algorithms*, 1st ed. New York, NY: Cambridge University Press, 2001.

[12] Lus Caviquea; Csar Regob; Isabel Themidoc, *A Scatter Search Algorithm for the Maximum Clique Problem*, School of Business Administration, University of Mississippi, University, MS 38677, USA, 2001.

[13] Thomas H. Cormen; Charles E. Leiserson; Ronald L. Rivest; Clifford Stein, *Introduction to Algorithms*, Second Edition. MIT Press and McGraw-Hill, 2001.

[14] Richard B. Darst, *Introduction To Linear Programming*, Marcel Dekker, INC, 270 Madison Avenue, New York, New York 10016.

[15] Joseph G. Ecker; Michael Kupferschmid, *Introduction to Operations Research*, Krieger Pub Co ISBN 0-89464-576-5

[16] Ian Foster, *Designing and Building Parallel Programs*, An Online Publishing Project of Addison-Wesley Inc., Argonne National Laboratory, and the NSF Center for Research on Parallel Computation, http://www-unix.mcs.anl.gov/dbpp/.

[17] GIMP Version 2.2, 03 Jun. 2007, TGNU Image Manipulation Program. 10 Feb. 2007, http://www.gimp.org.

[18] Javier Garcia; Zeev Zalevsky; David Mendlovic, *Two-Dimensional Wavelet Transform by Wavelength Multiplexing*, APPLIED OPTICS, Vol. 35, No. 35, 10 December 1996.

[19] R. E. Gomory, *Mathematical programming*, Amer. Math. Monthly 72 1965 no. 2, part II 99–110. MR30#4595

[20] Jonathan L. Gross; Jay Yellen, *Handbook of Graph Theory*, 1st ed. New York, NY: CRC Press, 2004.

[21] F. Harary; I. C. Ross, *A Procedure for Clique Detection Using the Group Matrix*, Sociometry 20, 205-215, (1957).

[22] J. Hasselberg; M. Panos; G. Vairaktarakis, *Test Case Generators and Computational Results for the Maximum Clique Problem* , Department of Industrial and Systems Engineering, University of Florida, Gainesville, FL 32611, USA, 1992.

[23] Keld Helsgaun,"*An Effective Implementation of the Lin-Kernighan Traveling Salesman Heuristic*," Department of Computer Science, Roskilde University, DK-4000 Roskilde, Denmark.

[24] Frederick S. Hillier; Gerald J. Lieberman *Introduction to Operations Research*, McGraw-Hill: Boston MA. Eight edition. International edition. (2005) ISBN 0-07-321114-1

[25] Frederick S. Hillier; Gerald J. Lieberman: *Introduction to Operations Research*, 8th edition. McGraw-Hill. ISBN 0-07-123828.

[26] Karla Hoffman, *Combinatorial Optimization and Integer Programming*, http://iris.gmu.edu/ khoffman/papers/newcomb1.html.

[27] M. Immanuel; B. Marco; M. Panos; P. Marcello, *The Maximum Clique Problem*, ISE Department University of Florida, Gainesville, FL 32611 USA, 1999.

[28] Arnold T. Insel; Stephen H. Friedberg; Lawrence E. Spence, *Linear Algebra*, Pearson Education, Inc. Upper Saddle River, New Jersey 07458.

[29] D.S. Johnson, *Approximation algorithms for combinatorial problem*, J. Comput. Syst. Sci., 9, 256-278, (1974).

[30] Kengo Katayama, Akihiro Hamamoto, Hiroyuki Narihisa, *Solving the Maximum Clique Problem by k-opt Local Search*, Proceedings of the 2004 ACM symposium on Applied computing, March 14-17, 2004, Nicosia, Cyprus.

[31] C.T. Kelley, *Iterative Methods for Linear and Nonlinear Equations*, Society for Industrial and Applied Mathematics, 3600 University City Science Center, Philadelphia, PA 19104-2688.

[32] Michael Kirby, *Geometric Data Analysis*,John Wiley and Sons, Inc. 605 Third Avenue, New York, NY 10158-0012.

[33] Steven J. Leon, *Linear Algebra With Applications*, Pearson Education, Inc. Upper Saddle River, NJ 07458.

[34] S. Lin; B. W. Kernighan, *An Effective Heuristic Algorithm for the Traveling Salesman*, Bell Telephone Laboratories, Incorporated Murray Hill, N.J. (1971).

[35] K. Maghout, *Sur la Determination des Nombres de Stabilit6 et du Nombre Chromatiqued'un Graphe*, C.R. Acad. Sci., Paris 248, 2522-2523, (1959).

[36] John E. Mitchell, *Branch-And-Bound Methods for integer programming*, Department of Mathematics Sciences, Rensselaer Polytechnic Institute, Troy, NY 12180-3590,(1997) USA.

[37] John E. Mitchell, *Interior point algorithm for integer programming*, Department of Mathematics Sciences, Rensselaer Polytechnic Institute, Troy, NY 12180-3590, USA.

[38] Y. Narahari, *Data Structures and Algorithms*, Web-Enabled Lecture Notes, http://lcm.csa.iisc.ernet.in/hari/content_10pages.html.

[39] Christos H. Papadimitriou; Kenneth Steiglitz *Combinatorial Optimization: Algorithms and Complexity*, Corrected reprint of the 1982 original. Dover Publications, Inc., Mineola, NY, 1998.

[40] PANOS M. PARDALOS, *The Maximum Clique Problem*, Department of Industrial and Systems Engineering, 303 Weil Hall, University of Florida, Gainesville, FL 32611, (1993) USA.

[41] M. C. Paull; S. H. Unger, *Minimizing the Number of States in Incompletely Specified Sequential Switching Functions*, 1RE TRANS. Electronic Computers, EC-8: 356-367, (1959).

[42] Alex Pentland; Baback Moghaddam; Thad Starner, *View-Based and Modular Eigenspaces for Face Recognition* ,Computer Vision and Pattern Recognition, 201 Broadway Cambridge MA 02139 USA.

[43] Kenneth H. Rosen, *Discrete Mathematics and Its Applications*, Second Edition, McGraw-Hill, INC Mineola, NY 11501.

[44] Yousef Saad, *Iterative Methods For Sparse Linear Systems*, PWS Publishing Co., 20 Park Plaza Boston, MA 02116.

[45] Kristian Sandberg, *The Daubechies Wavelet Transform*, http://amath.colorado.edu/courses/5720/2000Spr/Labs/DB/db.html.

[46] Bart Selman; Henry A. Kautz; Bram Cohen, *Noise Strategies for Improving Local Search*, Proceedings of AAAI94, Seattle, WA, July 1994.

[47] Steven S. Skiena, *The Algorithm Design Manual*, Springer-Verlag, New York, 1997.

[48] J.H. Van Lint; R.M. Wilson, *Course in Combinatorics*, Cambridge, UK: Cambridge University Press, 1992.