THEORETICAL AND COMPUTATIONAL ADVANCES IN FINITE-SIZE FACILITY PLACEMENT AND ASSIGNMENT PROBLEMS

BY

KETAN HEMANT DATE

DISSERTATION

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Industrial Engineering
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2016

Urbana, Illinois

Abstract

The goal of this research is to develop fundamental theory and exact solution methods for the optimal placement of multiple, finite-size, rectangular facilities in presence of existing rectangular facilities, in a plane. Applications of this research can be found in facility layout (re)design in manufacturing, distribution systems, services (retail outlets, hospital floors, etc.), and printed circuit board design; where designing an efficient layout can save millions of dollars in operational costs. Main difficulty of this optimization problem lies in its continuous non-convex/non-concave feasible space, which makes it tough to escape local optimality. Through this research, novel approaches will be proposed which can be used to distill this continuous space into a finite set of candidate solutions, making it amenable to search for the global optimum. The first two parts of this research deal with establishing a unified theory for the finite-size facility placement problem and establishing the theory of dominance for pruning the sub-optimal solutions. Traditionally, the facility location/layout problems are modeled as the Quadratic Assignment Problem (QAP), which is strongly NP-hard. Also, for getting strong lower bounds in the dominance procedure, we may need to solve an instance of the NP-hard Quadratic Semi-Assignment Problem (QSAP). To this end, the third part of this research deals with investigating parallel and High Performance Computing (HPC) methods for solving the Linear Assignment Problem (LAP), which is an important sub-problem of the QAP. The final part of this research deals with investigating parallel and HPC methods for obtaining strong lower bounds and possibly solving large QAPs. Since QAP is known to be a computationally intensive problem, it should be noted that large in this context means problem instances with up to 30 facilities and locations.

This work is dedicated to my wife and my parents for their never-ending love and support.

Acknowledgments

I would like to express sincere gratitude towards my adviser, Professor Rakesh Nagi, for his excellent guidance and insightful critique during the course of this research. I am grateful for his constant support and encouragement through the ups and downs of this arduous task.

I would like to thank my Doctoral committee members: Professors Xin Chen, Sheldon Jacobson, and Ramavarapu Sreenivas for their support and guidance, which significantly improved this dissertation.

I would also like to thank our Assistant Director of Graduate Studies, Ms. Holly Kizer, for helping me and other students throughout the course of our study.

Last but not least, I would like to thank my friends and colleagues Hossein, Alok, Sushant, and Geoff, from both Urbana-Champaign and Buffalo, who made the journey of graduate school enjoyable.

Contents

List of Figures

List of Tables

Chapter 1

Prologue

As mentioned in Heragu (2008), established manufacturing companies need to redesign their layouts every 2 to 3 years on an average. This redesign may include addition/removal of departments and machines in the current facility, or moving into a new facility altogether. In addition, as much as 30–75% of a product's cost can be associated with the material handling, and these expenses account for 20–50% of the total operational budget of a manufacturing company. Therefore, designing an efficient layout that minimizes the material handling overhead can potentially save millions of dollars for the company.

As is the case with many design problems, the facilities design problems require both mathematical analysis and human intuition. The mathematical aspect of facilities design is addressed by facility-location and facility-layout theories, both of which enjoy a rich body of literature (see Section 2.2). The location problems typically deal with finding optimal location of n infinitesimal (or point) facilities on a network or on a plane; while the layout problems typically deal with finding the optimal and non-overlapping arrangement of n rectangular departments within a rectangular facility. Although there exist sophisticated models and algorithms for location and layout problems in the literature, there are some real applications for which these models prove to be inadequate. To be specific, there is a class of problems that inherit the characteristics of both the location and layout problems, and there is lack of unifying/underlying theory and analytical framework for these intermediary problems. As motivating examples, let us consider the following cases from two different sectors of industry.[1]

Manufacturing Layout Redesign. First, consider the case of a manufacturing company which produces shock absorbers and dampers.[2] This company had experienced significant business growth, which placed additional strain on their operations. To accommodate the increased demand, they had constructed a much larger facility, which would be able to house their existing machines plus some new high capacity machines that they were planning to purchase. A total

[1]Facility layout image courtesy: `https://www.pinterest.com/matteoamela/bram_manufacturing-facility/` PCB image courtesy: `http://www.nexfy.co.in/`

[2]This was an actual project and the first author was a part of the project team. We have anonymized the details for privacy purposes.

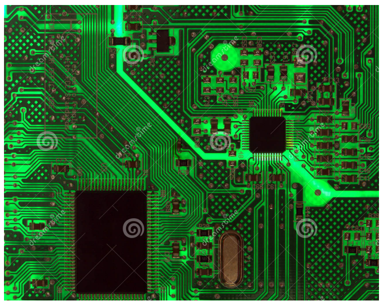


Figure 1.1: Motivating applications: Facility layout design and PCB design

of 23 machining centers (and departments) were identified and they had some type of interaction with each other as dictated by the product route sheets. Some of the machines and departments had fixed locations due to physical constraints, e.g., the location of loading/unloading dock was fixed; the furnace had to be situated next to a vent near the outer perimeter; one of the heavier computerized numerical control (CNC) lathes had to be situated in a pre-constructed trench; etc. Each of the machines were required to have a small designated area for storing work-in-process (WIP) inventory and each department had at least one egress point. The objective of this project was to design an efficient shop-floor layout for the new facility, which would minimize the overall material handling cost as a function of product volumes and travel distances.[3]

VLSI Chip Design. Now, consider the process of very-large-scale-integration (VLSI) design, as described in Sarrafzadeh et al. (2003) and Baker (2011). In VLSI, thousands or more *metal-oxide-semiconductor field-effect transistors* (MOSFETs) are integrated on a single silicon wafer. Usually, the MOSFETs are grouped into rectangular *cells* as per functionality, i.e., each cell represents a single functional component of the chip (for example a *latch* or *flip-flop* represents a cell of a *memory* module). Each cell is rectangular in shape and has at least four connections (power, ground, input, and output). One of the objectives in VLSI chip design is to find an optimal arrangement of the cells on the silicon wafer which minimizes the lengths of interconnecting wires between the cells, which may potentially reduce the inductive/capacitive coupling and crosstalk. The capacitive crosstalk is one of the reasons of heat loss which is estimated to increase the power consumption of computers by 22% annually (DeMone, 2004). As a result, more than 10% of the average budget of IT companies is spent on cooling solutions (King, 2007), which can be mitigated by designing efficient layouts for the PCBs and VLSI circuits. There are other objectives in VLSI chip design such as maximizing the utilization of the wafer area, minimizing the congestion (for better routability), minimizing the circuit delay, etc., which may be considered separately or within the wirelength optimization model itself. Many facility-layout models and algorithms are used for cell placement and floorplanning in VLSI chip design. For more details, the readers are directed to Sherwani (1999) and Sarrafzadeh et al. (2003).

[3]At the time, this problem was modeled as a facility layout problem and the new layout was designed using the systematic layout planning (SLP) approach with the help of activity relationship charts and diagrams (REL).

Both the applications discussed above exhibit the following traits.

1. The main facility has $M+N$ sub-facilities with finite area and rectangular shapes, and $N \geq 0$ of those facilities have fixed locations due to physical constraints.

2. The sub-facilities have a non-negative interaction with each other.

3. Each sub-facility has one or more input/output (I/O) points, through which it can interact with the other facilities.

4. The objective is to find the arrangement of the M sub-facilities which does not overlap with any other facility and minimizes the total weighted distance between all the interacting I/O points.

For these types of problems, existing location models cannot be used because the area occupied by the sub-facilities is sufficiently large as compared to the total area of the main facility. Also, existing layout models cannot be used because in most of these models, the travel distances are measured between the centroids of the rectangular sub-facilities, which is unrealistic in the motivating cases discussed. Therefore, there is a need to formulate and analyze this new class of problems. In this work we have developed a framework which unifies and advances the theories of facility-location and facility-layout so that they can be effectively used to address the above type of problems. To this end, the goal of this research is to develop fundamental theory and exact solution methods for the optimal placement of multiple, finite-size, rectangular facilities in presence of existing rectangular facilities, in a plane. This dissertation is divided into four chapters dealing with four interconnected problems. A brief description of each chapter is presented below.

In Chapter 2, a theoretical framework is developed for the problem of placing M finite-size rectangular facilities in a planar layout. It is proved that the feasible candidate solutions for this problem are finite in number, and an explicit enumeration procedure is proposed which is exponentially bounded in the number of new facilities. This research unifies the theories of facility location and facility layout analyses, through advancement of the location theoretic methods to the finite-size facility placement in a layout.

Due to its exponential complexity, the explicit enumeration procedure may become intractable for a large number of new and existing facilities. To address this issue, a procedure is needed which can efficiently prune sub-optimal solutions and reduce the number of candidate solutions to be evaluated. In Chapter 3, dominance results are established for placement of finite-size facilities in a planar layout and their effectiveness is shown empirically for the placement of 1 and 2 new facilities in an existing layout. These results are further generalized to the placement of M new facilities, by modeling the problem as a Quadratic Semi-assignment Problem (QSAP). The work regarding the dominance theory for a single new facility has been published in *Computers & Operations Research.*

For obtaining the lower bounds in the M facility dominance procedure, the QSAP has to be solved efficiently, which is a specialization of the QAP. The Linear Assignment Problem (LAP) is an important sub-problem of the QAP, and in a typical lower bounding technique for the QAP, large number of LAPs need to be solved in an efficient manner. To this end, Chapter 4 of this dissertation deals with developing parallel versions of two variants of the Hungarian algorithm, specifically designed for Compute Unified Device Architecture (CUDA) enabled NVIDIA Graphics Processing Units (GPU). The main contribution of this work is an efficient parallelization of the augmenting path search phase of the Hungarian algorithm, which is the most time consuming phase of the algorithm. Our algorithm finds more than one vertex-disjoint augmenting paths per iteration as opposed to a single augmenting path per iteration of the sequential algorithm. All these augmenting paths can be used to increase the cardinality of the matching. Therefore, the accelerated algorithm converges to the optimal solution in fewer number of iterations. In addition, we tested a scalable multi-GPU version of the algorithm, which provides a good alternative for solving larger problems which cannot be solved on a single GPU due to memory limitations. This work has been published in *Parallel Computing*.

Chapter 5 of this dissertation deals with developing a novel GPU-based parallelization of the Lagrangian Dual Ascent procedure, for obtaining strong lower bounds on the RLT2 linearization for the QAP. In this scheme, a large number of LAPs need to be solved and a large number of dual multipliers need to be updated during each iteration. Both these steps can be parallelized on a bank of GPUs, i.e., each GPU can solve a subset of LAPs using our GPU-accelerated Hungarian algorithm; while each dual multiplier can be adjusted by a single GPU thread. This GPU-based parallel/distributed algorithm shows excellent scaling as compared to the sequential algorithm, and obtains competitive lower bounds. Further, this lower bounding scheme is coupled with a parallel branch-and-bound algorithm which is used for solving the QAPs from the literature, on the Blue Waters supercomputing facility. This study shows that our scheme can be effective in solving large QAPs, subject to the availability of required number of GPUs. Since QAP is known to be a computationally intensive problem, it should be noted that large in this context means problem instances with up to 30 facilities and locations.

Chapter 2

Theoretical Advances in Finite-size Facility Placement Problem

In this chapter we investigate a new problem of optimal placement of M finite-size rectangular facilities with known dimensions in presence of existing rectangular facilities. Applications of this problem can be found in facility layout (re)design in manufacturing, distribution systems, services (retail outlets, hospital floors, etc.), and electronic circuit design. We consider three types of facility interactions: interaction between the new facilities and existing facilities; interaction between pairs of existing facilities; and interaction between pairs of new facilities. All interactions are serviced through a finite number of input/output points located strictly on the boundary of each facility. We assume that all travel occurs according to the rectilinear (or Manhattan) metric and travel through the facilities is not permitted. The objective is to find the simultaneous and non-overlapping placement of the new facilities such that the sum of weighted distances between all the interacting facilities is minimized. The simultaneous placement of new facilities introduces new challenges because the placement of any new facility could affect the distances between the various pairs of new and/or existing facilities. To arrive at a solution, we divide the feasible region into sub-regions and we prove that the candidates for the optimal placement of the new facilities can be drawn from the corners of these sub-regions. The solution complexity of this procedure is exponential in the number of new facilities and the numerical results corroborate this analysis. Heuristic procedures also perform well in practice with a maximum optimality gap of 0.94%. Main contribution of this work is an analytical framework that unifies and generalizes the facility location/layout problems for minisum objective and rectilinear distance metric.

2.1 Introduction

2.1.1 Problem Statement

The layout under consideration is a rectangular, closed region with finite area. There are N *existing facilities* (EFs), with rectangular shapes and edges parallel to the travel axes. M *new* *facilities* (NFs) having rectangular shapes and known dimensions are to be placed in the layout in presence of the EFs with their edges parallel to the travel axes. Each EF has one or multiple I/O point(s) while each NF has a single I/O point. The I/O points are strictly located on the boundary of each facility and flow between the facilities is serviced through them. We assume that the travel occurs according to the *Rectilinear* metric and the travel through a facility is not permitted (i.e., NFs and EFs act as barriers to travel). Three types of interactions are considered, whose values are assumed to be known:

- Pairwise interactions between the I/O points of existing facilities.
- Pairwise interactions between the I/O points of new and existing facilities.
- Pairwise interactions between the I/O points of new facilities.

The objective is to determine the optimal placement of the NFs (designated by the location of their top left corners) such that there is no overlap between the NFs and the EFs, and the total cost of travel (calculated as the weighted sum of rectilinear distances between the interacting facilities) is minimized.

2.1.2 Difficulties and Solution Approach

Let us consider the layout shown in Fig. 2.1. This layout has $N = 3$ EFs and we need to place $M = 3$ NFs in presence of these EFs. Initially let us relax the size restrictions on the NFs by assuming that they are infinitesimal. A variation of this problem was first studied by Larson and Sadiq (1983), in which the NF–NF interactions were absent. According to their solution procedure initially, a grid is constructed by passing horizontal and vertical lines through the vertices of each EF and its I/O point(s). Since travel through the EFs is prohibited, these gridlines will terminate at an EF boundary or at the layout boundary, whichever is encountered first. The authors showed that the $O(N^2)$ gridline intersection points are the candidates for optimal location of the NFs. Now, for the example problem, let us assume that EF_1 has high interaction with NF_1, EF_2 has high interaction with NF_2, and EF_3 has high interaction with NF_3. Let us also assume that the interactions between the other pairs of EFs and NFs are negligible. Then, Fig. 2.1 (a) shows the optimal solution for the infinitesimal NFs, while Fig. 2.1 (b) shows the optimal solution for the finite-size NFs with I/O points at the top left corner.

When the NFs have finite size, they act as barriers to travel, and could destroy the existing gridlines. In addition, the NFs are not allowed to overlap with each other or with any of the EFs. As a result, many of the feasible candidate points (including the infinitesimal optimum) may become infeasible. For a particular placement, the NFs introduce new gridlines and elongate the existing shortest feasible rectilinear paths. As a result, the shortest distances between the pairs of EF I/O points also become functions of the NF placements. This causes the objective function to become non-convex and non-concave over the $\Re^2$ space and use of gradient methods may result in a local minimum. Therefore, we need a better procedure for finding the optimal placement

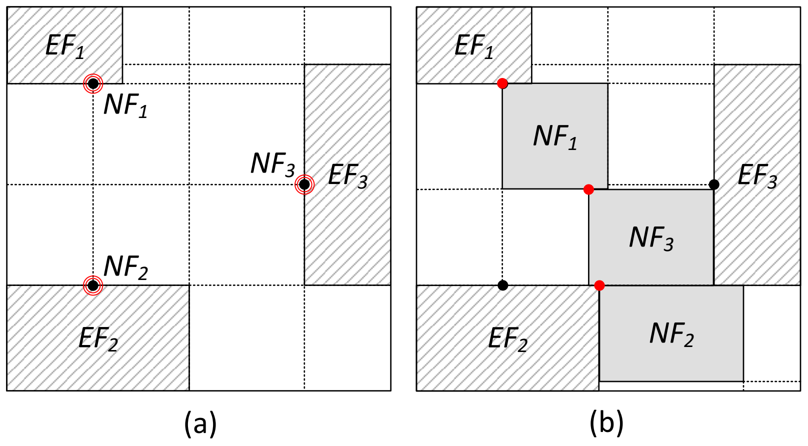


Figure 2.1: Example layout

candidates for the NFs. Our solution approach is to first divide the feasible region into sub-regions and then obtain the candidate points for the placement of the NFs by systematically studying their interactions over these sub-regions. This method ensures a global optimal solution. Since, we have to evaluate every feasible candidate point for the NFs, the time complexity of this procedure is quite high. We will formally analyze the solution complexity in Section 2.5 and show that it is, in fact, exponential in the number of NFs. So, for a large number of NFs, the procedure might become intractable. This is not due to any limitations of the proposed algorithm, but merely due to the difficulty of the problem itself. Even if relax the constraints on the NF sizes, our problem reduces to an instance of the Quadratic Semi-Assignment Problem (QSAP) (Burkard, 2002, Pitsoulis, 2009), which is NP-hard (see Section 2.5). Therefore, it is quite unlikely that one can come up with a polynomial algorithm which can solve this problem to optimality. For a large number of NFs, we might have to rely on some dominance rules or heuristic procedures, which can reduce the number of candidate points to be evaluated.

2.1.3 Chapter Organization

The overall chapter is organized as follows. Section 2.2 contains a brief literature review on the location, layout, and the new class of single facility "placement" problems. Section 2.3 contains some preliminary results, which will be helpful in the analysis. In Section 2.4, we establish our results and develop a procedure for obtaining the candidate points for the placement of the new facilities. In Section 2.5, we evaluate the solution complexity of our procedure and discuss some special cases which might be easier to solve. Section 2.6 discusses the specifics of implementing the procedure on a computer. Section 2.7 contains numerical results of our algorithm and its comparison with some heuristic approaches. Finally, in Section 2.8, the chapter is concluded with a summary and future research directions.

2.2 Literature Review

2.2.1 Facility Location Problem

In the most basic location problems, there are two main assumptions: (1) The new and existing facilities are *infinitesimal* in size, and (2) The new facilities do not interact with each other. Hakimi (1964) proved the *vertex-optimality* property for the problem of locating p new facilities on a network layout. Larson and Sadiq (1983) studied the problem of locating p new facilities on a planar layout consisting of arbitrarily shaped *barriers*[1] and developed a discrete search algorithm for the same. Batta et al. (1989) extended the results from Larson and Sadiq (1983), to include convex forbidden regions, through which only travel is permitted but not the facility location. Later, Hamacher and Schöbel (1997) developed a polynomial procedure for locating p new facilities in presence of existing infinitesimal facilities and forbidden polyhedra, with the center objective and Euclidean distance metric. Wang et al. (2002) contributed to this body of research by developing polynomial-time algorithms for locating input/output (I/O) points on the existing rectangular facilities. The problem of locating multiple new *interacting* facilities has also received substantial attention in the literature. One variation of this problem was studied by Wesolowsky and Love (1971), in which the authors developed a gradient based algorithm for locating p new interacting facilities in presence of m point and n area destinations. Several other variations of this problem have been studied by Love and Kraemer (1973), Love and Morris (1975), Love and Yerex (1976), Juel and Love (1976), Hamacher and Nickel (1995) (for restricted location problems).

2.2.2 Facility Layout Problem

In the most basic layout problems, there are two main assumptions: (1) All the facilities/departments have the same dimensions, and (2) All the locations are known *a priori*. With these assumptions, the layout problem can be modeled as a Quadratic Assignment Problem (QAP) (Koopmans and Beckmann, 1957), which is known to be NP-hard (Sahni and Gonzalez, 1976). Many formulations and algorithms have been developed over the years for solving the QAP optimally or sub-optimally. For a list of references on QAP, we direct the readers to the survey papers by Burkard (2002) and Loiola et al. (2007). For *space-filling* layout problems with unequal departmental areas and aspect ratios, several mixed integer linear programming (MILP) formulations were introduced by Montreuil (1991), which are typically much more difficult to solve than the standard QAP. Several improvements to Montreuil's basic formulation were proposed by Meller et al. (1998), Sherali et al. (2003), and Castillo et al. (2005) which provide tighter lower bounds when used in a branch-and-bound scheme and also produce layouts with better quality. The survey papers by Kusiak and Heragu (1987) and later by Meller and Gau (1996) provide an excellent overview of the various formulations and algorithms, that have been used to solve the layout problems optimally or sub-optimally. We also direct the readers to the book by Heragu (2008), for additional references.

[1]Barriers are objects in the layout through which travel is not permitted and the location of new facilities is forbidden.

2.2.3 Facility Placement Problem

The term *facility placement* was coined by Savaş et al. (2002)[2] for the problem of "locating" one finite-size facility in an existing layout. This problem can be considered as an intersection between location and layout problems. Since the new facilities are to be located on a plane, it certainly falls under location problems. Also, due to the finite size of the new facilities, non-overlap restrictions must be obeyed, similar to the layout problems. Drezner (1986) studied one variation of finite-size facility location problem, with median objective, and Euclidean and squared-Euclidean distance metrics. The facilities were assumed to have circular area and the service was assumed to be uniformly distributed over the area. The author found an analytical solution to calculate the *effective distance* (d_e) between the various interacting facilities. The effective distance can be multiplied by the respective weights and then added together to obtain the objective function to be minimized.

In the work by Savaş et al. (2002), the authors developed new procedures for placement of one arbitrarily shaped facility with a fixed I/O point (or *server*) location, in presence of barriers to travel, with median objective and rectilinear distance metric. Sarkar et al. (2005, 2007) extended this work to *generalized congested regions* (GCR), through which travel is permitted for a penalty. The authors considered rectangular GCRs in their work and established procedures for finding the optimal location of the GCR, its exact dimensions, and the optimal location of the I/O point on that GCR. Kelachankuttu et al. (2007) extended the contour line construction procedure to single facility placement problem. Zhang et al. (2009) developed polynomial-time algorithms for single facility placement problem in presence of existing facilities and aisle structure.

From the literature referenced above, it is clear that the field of planar facility placement is relatively new and unexplored. Moreover, the problem of placing *multiple* new finite-size facilities with non-negative interactions in the presence of existing finite-size facilities, is a novel one. To the best of our knowledge, this problem has not received any attention in the literature, due to its complexity. In this chapter, we intend to study this important problem rigorously and develop results for the optimal placement of M new facilities. Our main contribution is an analytical framework that generalizes the facility location/facility placement problem, with minisum objective and rectilinear distance metric. This analysis can be applied seamlessly to many of the single and multi-facility location problems studied in the literature, by relaxing the constraints on the facilities.

2.3 Preliminaries

2.3.1 Notation and Problem Definition

The layout under consideration is a rectangular, closed region in $\Re^2$ space, with finite area. Each EF in the layout is assumed to be a rectangular region in $\Re^2$ space, with closed boundary and finite area. Let H_a (an open set) be the set of points $(x, y) \in \Re^2$ contained strictly within EF$_a$.

[2]This term was already being used in VLSI chip design literature but not in facility-layout literature.

Let $\bar{H}_a = H_a \cup \{\text{boundary of EF}_a\}$, which is a closed set. We let $\mathbf{H} = \bigcup H_a$ and $\bar{\mathbf{H}} = \bigcup \bar{H}_a$. Let B_i denote the set of points contained strictly within NF$_i$ and $\bar{B}_i = B_i \cup \{\text{boundary of NF}_i\}$. Let $\mathbf{B} = \bigcup B_i$ and $\bar{\mathbf{B}} = \bigcup \bar{B}_i$. It is important to distinguish between the inside of a facility and its boundary, because the travel is permitted on the boundary of a facility but not on the inside. Let $E_q(\bar{H}_a)$, $q \in \{1,2,3,4\}$ denote the corners of EF$_a$, starting from the bottom-left corner and labeling them in the counter-clockwise direction. Let $E_r(\bar{B}_i)$, $r \in \{1,2,3,4\}$ denote the corners of NF$_i$, labeled in the similar fashion. Let X_i denote the location of the I/O point on the boundary of NF$_i$, with respect to its top left corner $E_4(\bar{B}_i)$. Let $D_i = \{E_1(\bar{B}_i), E_2(\bar{B}_i), E_3(\bar{B}_i), E_4(\bar{B}_i), X_i\}$ denote the set of all vertices of NF$_i$ and let $\tilde{n}_i \in D_i$ denote a generic element from this set. Let region $\mathbf{G} = \mathbf{H} \cup \mathbf{B}$ represent the set through which travel is not permitted. Let $\mathbf{Z}$ be the rectangular region representing the total layout area.

We can define the placement of the NFs in $\Re^2$ space using the coordinates of their top left corners. Let us define the *placement vector* for the NFs as $\mathbf{p} = \langle E_4(\bar{B}_1), E_4(\bar{B}_2), \cdots, E_4(\bar{B}_M) \rangle$. Note that the coordinates of all the NFs need to be considered in the placement vector to emphasize their disjointedness. Let $B_i(\mathbf{p})$ (an open set) denote the set of points contained within NF$_i$, when its placement is $\mathbf{p}$. We also define $\bar{B}_i(\mathbf{p}) = B_i(\mathbf{p}) \cup \{\text{boundary of NF}_i\}$, which is a closed set. The feasible region for the placement of M NFs can be defined as follows:

$$F_{(M)} = \left\{ \mathbf{p} \mid \bar{B}_i(\mathbf{p}) \subset \mathbf{Z};\ \bar{B}_i(\mathbf{p}) \cap \mathbf{H} = \emptyset;\ \bar{B}_i(\mathbf{p}) \cap B_j(\mathbf{p}) = \emptyset;\ \forall i,j \in \{1, \cdots, M\};\ i \neq j \right\}. \quad (2.1)$$

We consider three types of interactions in our problem. Firstly, there is an interaction between an EF I/O point a and NF I/O point X_i, denoted by $u_{ai} \geq 0$. Secondly, there is an interaction between two EF I/O points, a and b, denoted by $v_{ab} \geq 0$. And thirdly, there is an interaction between two NF I/O points X_i and X_j, denoted by $w_{ij} \geq 0$. We assume that the flows are undirected, i.e., $u_{ai} = u_{ia}$, $v_{ab} = v_{ba}$, and $w_{ij} = w_{ji}$. We also assume that $v_{aa} = w_{ii} = 0$. The interactions u_{ai}, v_{ab}, and w_{ij} basically represent the cost of material handling per unit distance. The interaction between any two facilities will take place through a shortest feasible rectilinear path, which does not penetrate any new or existing facility. Let $d_{\mathbf{p}}(a, X_i)$ denote the length of a shortest feasible path between EF I/O point a and NF I/O point X_i; let $d_{\mathbf{p}}(a,b)$ denote the length of a shortest feasible path between two EF I/O points, a and b; and let $d_{\mathbf{p}}(X_i, X_j)$ denote the length of a shortest feasible path between the NF I/O points of X_i and X_j. All these distances are dependent on the placement vector, hence the subscript. For a particular placement $\mathbf{p}$ of the NFs, let us denote the total weighted travel distance between EFs and NFs as $J(\mathbf{p})$, between EFs as $K(\mathbf{p})$, and between NFs as $L(\mathbf{p})$. Let A denote the set of all EF I/O points. The objective function for the placement problem is given by:

$$J(\mathbf{p}) + K(\mathbf{p}) + L(\mathbf{p}) = \sum_{a \in A} \sum_{i=1}^{M} u_{ai} d_{\mathbf{p}}(a, X_i) + \sum_{a \in A} \sum_{b \in A} v_{ab} d_{\mathbf{p}}(a,b) + \sum_{i=1}^{M} \sum_{j=1}^{M} w_{ij} d_{\mathbf{p}}(X_i, X_j). \quad (2.2)$$

The facility placement problem is to determine the optimal placement $\mathbf{p}^*$ of the NFs such that

$J(\mathbf{p}^*) + K(\mathbf{p}^*) + L(\mathbf{p}^*) \leq J(\mathbf{p}) + K(\mathbf{p}) + L(\mathbf{p}), \forall \mathbf{p} \in F_{(M)}$.

2.3.2 Nodal Paths

We now introduce an important concept of *nodal paths* from Larson and Li (1981). A *node* is defined as any point $(x, y) \in \Re^2$, which is not present in the interior of any barrier (EF or NF). Now, we state the following important results from Larson and Li (1981):

Definition 1. *A* staircase *path between two points* (x_a, y_a) *and* (x_b, y_b) *is a rectilinear path whose length is* $|x_a - x_b| + |y_a - y_b|$.

Definition 2. *Two points are said to* communicate *if there exists a feasible staircase path between them, i.e., the path is not made longer by EFs or NFs.*

Result 1. *Shortest path between any two points* (x_a, y_a) *and* (x_b, y_b) *in* $\Re^2$ *plane can be found by restricting the travel to the nodal paths, i.e., a path containing a sequence of nodes* $a - n_1 - n_2 - \ldots - n_k - b$*; where* (a, n_1), (n_1, n_2), $\ldots$, (n_k, b) *are the pairs of communicating nodes.*

So, a rectilinear path between any two nodes can be generated by tracing staircase paths between the intermediate, communicating nodes. The distance between the origin and the destination node will be the sum of the lengths of the intermediate paths. If n_0 and n_{k+1} denote the origin and destination nodes on a nodal path $P(n_0, n_{k+1})$ consisting of k number of internal nodes, then the expression for the length such a path can be written as:

$$\ell(P(n_0, n_{k+1})) = \sum_{r=0}^{k} |x_{n_r} - x_{n_{r+1}}| + \sum_{r=0}^{k} |y_{n_r} - y_{n_{r+1}}| = \ell(P_x(n_0, n_{k+1})) + \ell(P_y(n_0, n_{k+1})). \quad (2.3)$$

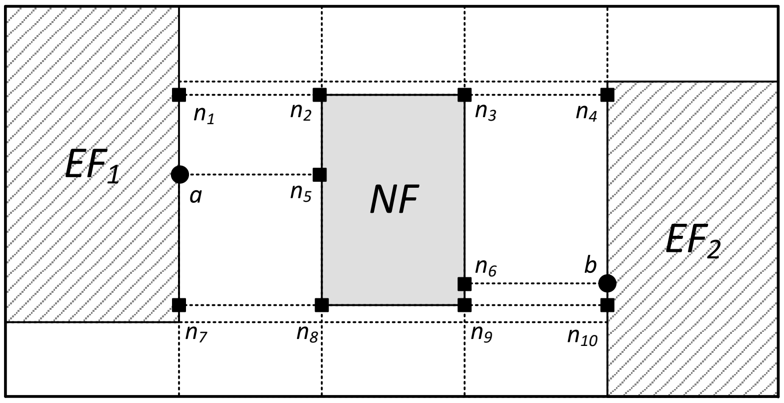


Figure 2.2: Example of nodal paths

The consequence of the above result is that, we can define a nodal path between a pair of I/O points, by tracing a sequence of various EF corners, NF corners, I/O points, and gridline intersection points that are present on the layout. In other words, we can identify a nodal path between any two I/O points by restricting the travel to the new and existing gridlines. Multiple such nodal paths might exist between a pair of I/O points, depending upon the sequence of the nodes traversed. For example, in Fig. 2.2, we can draw a path $a - n_1 - n_2 - n_3 - n_4 - b$ and another path $a - n_5 - n_2 - n_6 - b$, both of which have the same length. The travel will always take place along the shortest of the nodal paths, to minimize the overall objective function. Therefore, the central theme of our solution procedure would be to analyze the behavior of the nodal paths between the various I/O points and obtain the candidates for the placement of the NFs, which minimize the lengths of those paths.

2.3.3 $\mathcal{Q}$ Sets

When we draw the gridlines for a given layout, they divide the feasible region F into a number of *cells*. Each cell boundary is entirely composed of gridline segments. The cells are rectangular in shape, as a result of the rectangular shape of the EFs and the procedure followed for the construction of the gridlines. The cell corners are nothing but gridline intersection points, which can be treated as *nodes* in the various nodal paths. The importance of dividing the feasible region into rectangular cells is elaborated by the following results:

Result 2. *A shortest feasible rectilinear path from a point located outside a cell to an infinitesimal facility inside the cell can be assumed to pass through the cell corner (Larson and Sadiq, 1983).*

Result 3. *For an infinitesimal point X, $J(X)$ is concave within any cell (Batta et al., 1989).*

When NFs are fully contained within a cell, the EF–EF interaction remains unaffected, however if dimensions of any of the NFs are greater than the dimensions of a cell, then existing gridlines are cut off and the shortest rectilinear paths between the NFs and EFs are potentially destroyed. Additionally, the NFs will create some new gridlines, rerouting the flow along them, and possibly increasing the travel distances. To characterize the behavior of the distance function due to placement of the NFs, we will use the concept of $\mathcal{Q}$ sets introduced by Savaş et al. (2002).

Consider an initial feasible placement $E_4(\bar{B})_{ini}$ of a single NF such that:

- The NF intersects a set of gridlines, $L_g = \{l_1, ..., l_t\} \subset L$;
- The boundary of the NF does not coincide with any of the existing gridlines; and
- The I/O point X of the NF does not coincide with any of the existing gridlines.

Let $\mathcal{Q}$ denote the set of all placements such that when $E_4(\bar{B}) \in \mathcal{Q}$, the NF will always intersect the same set of gridlines L_g and the I/O point X will remain in the same cell C; i.e., $\mathcal{Q} = \{(x_{E_4(\bar{B})}, y_{E_4(\bar{B})}) \mid E_4(\bar{B}) \in \mathcal{Q}; X \in C\}$. Hence set $\mathcal{Q}$ represents the feasible NF placement area and it can be constructed by moving the NF in all directions from the initial location $E_4(\bar{B})_{ini}$. Since the movement of the NF in a $\mathcal{Q}$ set always intersects the same set of gridlines L_g, such a $\mathcal{Q}$

set is bounded by the locations of $E_4(\bar{B})$ such that one of the edges of the NF or the I/O point X, coincides with a gridline. Any further movement of the NF in the same direction will result in addition or deletion of that gridline from the set L_g or will cause X to cross over into a different cell and hence the NF will enter into the domain of a different $\mathcal{Q}$ set. Each cell can have multiple $\mathcal{Q}$ sets depending on the gridlines intersected. If an NF can be completely contained within a cell C, then it does not cut off any existing gridlines. In this case $L_g = \emptyset$, and the EF–EF flow is unaffected. We can still construct a $\mathcal{Q}$ set within the cell C, using the procedure mentioned above. The objective function remains concave over this $\mathcal{Q}$ set, as per Result 3. Let $\boldsymbol{\mathcal{Q}} = \{\mathcal{Q}_1, \mathcal{Q}_2, \cdots, \mathcal{Q}_n\}$ be the collection of all $\mathcal{Q}$ sets for a single NF. Then the feasible region for the placement of that NF can be expressed as:

$$F_{(1)} = \bigcup_{i=1}^{n} \mathcal{Q}_i. \tag{2.4}$$

Figure 2.3 shows the construction of a typical $\mathcal{Q}$ set. The gridlines $h_0, ..., h_4$ and $v_0, ..., v_4$ are formed by the EFs and their I/O points. Consider the initial feasible placement $E_4(\bar{B})$, such that the NF always intersects horizontal gridlines h_1, h_2, h_3 and vertical gridlines v_1, v_2, v_3. Note that these gridlines should terminate at the NF boundary since the flow through the NF is prohibited, however, we shall retain them to help us in our analysis. The NF can be moved in $-x$ direction until its right edge coincides with gridline v_3 or in $+x$ direction until its right edge coincides with gridline v_4. The NF can be moved in $-y$ or $+y$ direction in the similar fashion; forming the said $\mathcal{Q}$ set. Note that $X \in C, \forall E_4(\bar{B}) \in \mathcal{Q}$.

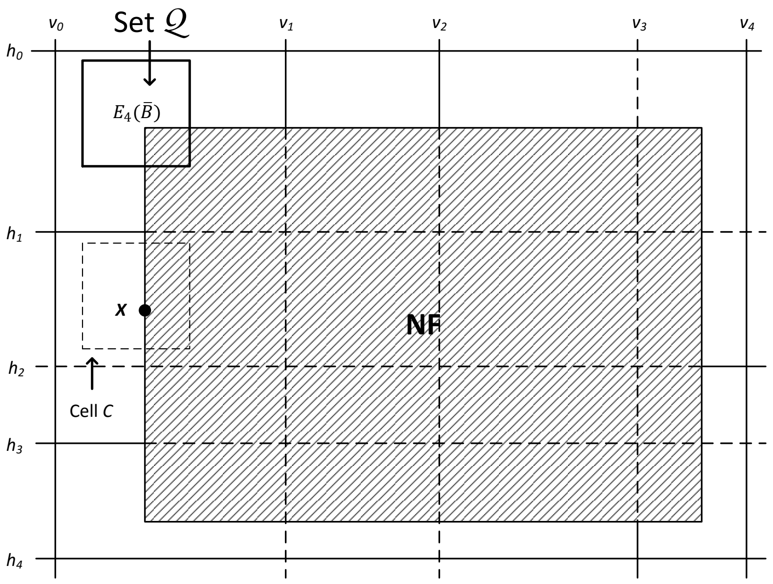


Figure 2.3: Construction of $\mathcal{Q}$ sets

The importance of incorporating the I/O point X in the definition of the $\mathcal{Q}$ set can be explained as follows. If the I/O point X intersects a gridline due to the movement of the NF, then it might start communicating with some of the previously non-communicating EF I/O points. In that case, the EF–NF interaction $J(\mathbf{p})$ might become non-concave over the $\mathcal{Q}$ set. However, if the $\mathcal{Q}$ set is defined by taking into account the I/O point X, then as soon as the I/O point crosses a gridline, the NF will enter into a different $\mathcal{Q}$ set and thus the linearity and concavity of the objective function is retained over individual $\mathcal{Q}$ sets. The $\mathcal{Q}$ sets possess the following important properties.

Result 4. *The $\mathcal{Q}$ sets are rectangular in shape with their edges parallel to the travel axes, because of the rectangular shape of the EFs and the NFs (Sarkar et al., 2005).*

Result 5. *The length of a nodal path between a cell corner and a vertex of an NF is a piecewise linear and concave function over a $\mathcal{Q}$ set (Savaş et al., 2002).*

Result 6. *The objective function $J(\mathbf{p}) + K(\mathbf{p})$ is concave over a $\mathcal{Q}$ set (Savaş et al., 2002).*

2.4 Solution Procedure

With the knowledge of $\mathcal{Q}$ sets and nodal paths, we can develop a procedure to find the candidates for the optimal placement of the NFs. We start by constructing $\mathcal{Q}$ sets, independently for each of the NFs. Let $\boldsymbol{\mathcal{Q}}^i = \left\{\mathcal{Q}_1^i, \mathcal{Q}_2^i, \cdots, \mathcal{Q}_{n_i}^i\right\}$ denote the collection of all $\mathcal{Q}$ sets, constructed for the placement of NF$_i$. Let us define a Cartesian product set $\boldsymbol{\mathcal{T}} = \prod_{i=1}^{M} \boldsymbol{\mathcal{Q}}^i$. For an M-dimensional tuple $\bar{\mathbf{t}} \in \boldsymbol{\mathcal{T}}$, let us define a set $\mathcal{T}_{\bar{\mathbf{t}}}$, which contains all the $\mathcal{Q}$ sets from the tuple $\bar{\mathbf{t}}$. There exist a finite number of these $\mathcal{T}_{\bar{\mathbf{t}}}$ sets, and in Section 2.5, we will provide an upper bound on their number. Let us also define a set $\mathcal{P}_{\bar{\mathbf{t}}} = \prod_{i=1}^{M} \left(\mathcal{Q}^i \in \mathcal{T}_{\bar{\mathbf{t}}}\right)$. This $\mathcal{P}$ set is the analog of a $\mathcal{Q}$ set in M-dimensional space, and any placement vector $\mathbf{p} \in \mathcal{P}$. Then we can write the following relation for the feasible region for the placement of M NFs:

$$F_{(M)} \subseteq \bigcup_{\bar{\mathbf{t}}} \mathcal{P}_{\bar{\mathbf{t}}}. \tag{2.5}$$

Note that the above two sets are not equal, as opposed to those in Equation (2.4), because some of the NF placements in various $\mathcal{P}$ sets might become infeasible due to their overlap with each other. To find the optimal placement of the NFs, we need to analyze the behavior of the objective function over the various $\mathcal{P}$ sets, and identify the placements for which the overall objective function is minimized. Note that from here onward, we will omit some of the subscripts for convenience.

Consider a set $\mathcal{T}$, the corresponding set $\mathcal{P}$, and assume that the NFs are placed at a particular placement vector $\mathbf{p} \in \mathcal{P}$. Consider a nodal path $P(a, b)$ in the resulting layout, where a and b are any two I/O points. This path can be assumed to traverse through a sequence of cell corners, EF corners, EF I/O points, NF corners, and NF I/O points. There may exist a finite number of such nodal paths between the points a and b. Therefore, the expression for the shortest distance between the points a and b, over the set $\mathcal{P}$, can be written as:

$$d_{\mathbf{p}}(a, b) = \min_{P(a,b)} \left\{ \min_{\mathbf{p} \in \mathcal{P}} \left\{ \ell(P(a, b)) \right\} \right\}. \tag{2.6}$$

Remark 1. *According to Equation (2.3), the length of any nodal path $\ell(P(a,b))$ is the sum of two expressions $\ell(P_x(a,b))$ and $\ell(P_y(a,b))$, which are coordinate-wise independent. Consequently, $\ell(P(a,b))$ can be minimized, by separately minimizing the expressions $\ell(P_x(a,b))$ and $\ell(P_y(a,b))$.*

To arrive at a solution, we need to identify the regions within the various $\mathcal{Q}$ sets, such that the lengths of all the nodal paths remain concave. This will guarantee that the shortest distance between any two I/O points, and hence the overall objective function remains concave, which can be minimized at the corners of these regions. In the following sections, we will precisely explain how to find these regions, which retain the concavity of the lengths of the nodal paths.

Let us assume that a nodal path $P(a,b)$ traverses through t number of nodes, out of which exactly $m \leq t$ nodes are NF vertices. Let us consider the following cases:

1. If $m = 0$, then the function $\ell(P(a,b))$ remains constant and can be excluded from the analysis.
2. If $m = 1$, i.e., if $P(a,b)$ traverses through the vertex $\tilde{n}_i$ of NF$_i$, then we can apply Result 5 and conclude that the function $\ell(P(a,b))$ remains piecewise linear and concave over the set $\mathcal{Q}^i$.
3. If $m \geq 2$, then we need to analyze the function $\ell(P(a,b))$ in a systematic manner. In the following sections, we will analyze and develop results for the cases where $m = 2$ and $m = 3$, and then extend those results to the general case.

2.4.1 Analysis for the Case of Two NFs

We will first focus our attention on the problem of optimal placement of two NFs. Let us consider a nodal path $P(a,b)$, which traverses through a vertex $\tilde{n}_i$ of NF$_i$, placed within the set $\mathcal{Q}^i$, and the vertex $\tilde{n}_j$ of NF$_j$, placed within the set $\mathcal{Q}^j$. Without the loss of generality, let us assume that a and b are existing cell corners. If $\tilde{n}_i$ and $\tilde{n}_j$ are not communicating, we can split $P(a,b)$ into two subpaths, such that the length of each subpath is affected by exactly one of the NFs. Then we can apply Result 5 to each of the subpaths to conclude that their lengths remain piecewise linear and concave over the sets $\mathcal{Q}^i$ and $\mathcal{Q}^j$. Consequently, the function $\ell(P(a,b))$ also remains piecewise linear and concave over the sets $\mathcal{Q}^i$ and $\mathcal{Q}^j$. If $\tilde{n}_i$ and $\tilde{n}_j$ are communicating, the expression for the length of the path $P(a,b)$ can be written as:

$$\ell\left(P\left(a,b\right)\right) = \ell(P'(a,\tilde{n}_i)) + \ell(P'(\tilde{n}_i,\tilde{n}_j)) + \ell(P'(\tilde{n}_j,b)). \tag{2.7}$$

Remark 2. *Since the nodes a and b are assumed to be existing cell corners, and there are no other NF vertices on the subpaths $P'(a,\tilde{n}_i)$ and $P'(\tilde{n}_j,b)$, the function $\ell(P'(a,\tilde{n}_i)) + \ell(P'(\tilde{n}_j,b))$ is a piecewise linear and concave function over the sets $\mathcal{Q}^i$ and $\mathcal{Q}^j$.*

Therefore, we only need to analyze the behavior of the function $\ell(P'(\tilde{n}_i,\tilde{n}_j))$, in detail. Since $\tilde{n}_i$ and $\tilde{n}_j$ are assumed to be communicating, the expression for the length of the subpath $P'\left(\tilde{n}_i,\tilde{n}_j\right)$, can be written as:

$$\ell\left(P'\left(\tilde{n}_i,\tilde{n}_j\right)\right) = \left|x_{\tilde{n}_i} - x_{\tilde{n}_j}\right| + \left|y_{\tilde{n}_i} - y_{\tilde{n}_j}\right|. \tag{2.8}$$

Definition 3. *A pair of NF vertices $\tilde{n}_i$ and $\tilde{n}_j$ are called:*

1. Non-interfering vertices*, if their coordinates are ordered, i.e., $x_{\tilde{n}_i} \leq x_{\tilde{n}_j}$ or $x_{\tilde{n}_i} \geq x_{\tilde{n}_j}$, $\forall E_4(\bar{B}_i) \in \mathcal{Q}^i$; and $y_{\tilde{n}_i} \leq y_{\tilde{n}_j}$ or $y_{\tilde{n}_i} \geq y_{\tilde{n}_j}$, $\forall E_4(\bar{B}_j) \in \mathcal{Q}^j$.*

2. Interfering vertices along x-axis*, if their x-coordinates are not ordered, i.e., $\exists E_4(\bar{B}_i) \in int(\mathcal{Q}^i)$ and $\exists E_4(\bar{B}_j) \in int(\mathcal{Q}^j)$ such that: (1) $x_{\tilde{n}_i} = x_{\tilde{n}_j}$, (2) $\exists \epsilon > 0$ such that $(x_{E_4(\bar{B}_i)} \pm \epsilon, y_{E_4(\bar{B}_i)}) \in$ int($\mathcal{Q}^i$) and feasible, and (3) $\exists \epsilon > 0$ such that $(x_{E_4(\bar{B}_j)} \pm \epsilon, y_{E_4(\bar{B}_j)}) \in$ int($\mathcal{Q}^j$) and feasible.*

3. Interfering vertices along y-axis*, if their y-coordinates are not ordered, i.e., $\exists E_4(\bar{B}_i) \in int(\mathcal{Q}^i)$ and $\exists E_4(\bar{B}_j) \in int(\mathcal{Q}^j)$ such that: (1) $y_{\tilde{n}_i} = y_{\tilde{n}_j}$, (2) $\exists \epsilon > 0$ such that $(x_{E_4(\bar{B}_i)}, y_{E_4(\bar{B}_i)} \pm \epsilon) \in$ int($\mathcal{Q}^i$) and feasible, and (3) $\exists \epsilon > 0$ such that $(x_{E_4(\bar{B}_j)}, y_{E_4(\bar{B}_j)} \pm \epsilon) \in$ int($\mathcal{Q}^j$) and feasible.*

Remark 3. *It is possible that $\exists E_4(\bar{B}_i) \in int(\mathcal{Q}^i)$ and $\exists E_4(\bar{B}_j) \in int(\mathcal{Q}^j)$ such that: $x_{\tilde{n}_i} = x_{\tilde{n}_j}$ and $y_{\tilde{n}_i} = y_{\tilde{n}_j}$. However, the other conditions make sure that the vertices cannot be interfering along both axes, without causing infeasibility. As a result, Equation (2.8) may become non-concave only along one of the axes (as seen in Lemma 2).*

Lemma 1. *For a pair of non-interfering NF vertices, the function $\ell(P(a,b))$ given by Equation (2.7) is a piecewise linear and concave function over the sets $\mathcal{Q}^i$ and $\mathcal{Q}^j$.*

Proof. Since the NF vertices $\tilde{n}_i$ and $\tilde{n}_j$ are assumed to be non-interfering, we have $x_{\tilde{n}_i} \leq x_{\tilde{n}_j}$ or $x_{\tilde{n}_i} \geq x_{\tilde{n}_j}$ but not both, $\forall E_4(\bar{B}_i) \in \mathcal{Q}^i$ and $\forall E_4(\bar{B}_j) \in \mathcal{Q}^j$. Therefore, the term $\left|x_{\tilde{n}_i} - x_{\tilde{n}_j}\right|$ in Equation (2.8) is a linear function, which increases or decreases monotonically as the NFs are moved within the sets $\mathcal{Q}^i$ and $\mathcal{Q}^j$. By the similar argument, the term $\left|y_{\tilde{n}_i} - y_{\tilde{n}_j}\right|$ in Equation (2.8) is also a linear and monotone function over the sets $\mathcal{Q}^i$ and $\mathcal{Q}^j$. Therefore, the function $\ell(P'(\tilde{n}_i, \tilde{n}_j))$ is a piecewise linear and concave function. Combining this with Remark 2, we get the desired result. □

If the NF vertices interfere with each other, then this result might not hold, as demonstrated by the following lemma.

Lemma 2. *The function $\ell(P'(\tilde{n}_i, \tilde{n}_j))$ given by Equation (2.8) becomes non-concave over the sets $\mathcal{Q}^i$ and $\mathcal{Q}^j$, if and only if the vertices $\tilde{n}_i$ and $\tilde{n}_j$ interfere with each other.*

Proof. In Lemma 1, we have already shown that the function $\ell(P'(\tilde{n}_i, \tilde{n}_j))$ is piecewise linear and concave, if the vertices $\tilde{n}_i$ and $\tilde{n}_j$ do not interfere with each other. Therefore, it suffices to show that the function $\ell(P'(\tilde{n}_i, \tilde{n}_j))$ becomes non-concave if the vertices $\tilde{n}_i$ and $\tilde{n}_j$ interfere with each other. Let us assume that the NFs are placed at points $E_4(\bar{B}_i) \in \text{int}(\mathcal{Q}^i)$ and $E_4(\bar{B}_j) \in \text{int}(\mathcal{Q}^j)$, and suppose for this particular placement, $y_{\tilde{n}_i} = y_{\tilde{n}_j}$. Therefore, according to Definition 3, the vertices $\tilde{n}_i$ and $\tilde{n}_j$ interfere with each other along the y-axis. The x-coordinates of the NF vertices must be ordered and therefore, the function $|x_{\tilde{n}_i} - x_{\tilde{n}_j}|$ remains linear and concave over the sets $\mathcal{Q}^i$ and $\mathcal{Q}^j$. Since the NFs are placed in the interior of the respective $\mathcal{Q}$ sets, $\exists \epsilon > 0$ such that $(x_{E_4(\bar{B}_i)}, y_{E_4(\bar{B}_i)} \pm \epsilon) \in \text{int}(\mathcal{Q}^i)$ and $(x_{E_4(\bar{B}_j)}, y_{E_4(\bar{B}_j)} \pm \epsilon) \in \text{int}(\mathcal{Q}^j)$. Now if we keep the NF$_i$

placement fixed, and move NF$_j$ in $+y$ direction by ϵ, then the function $|y_{\tilde{n}_i} - y_{\tilde{n}_j}|$, and consequently $\ell\left(P'\left(\tilde{n}_i, \tilde{n}_j\right)\right)$ will increase by ϵ. Similarly, if we move NF$_j$ in $-y$ direction by ϵ, then the function $\ell\left(P'\left(\tilde{n}_i, \tilde{n}_j\right)\right)$ will again increase by ϵ, thus making it piecewise linear and convex. Similar argument holds if the NF vertices interfere along the x-axis. The lemma follows.

□

As a consequence of Lemma 2, we cannot claim the concavity of the function $\ell\left(P\left(a, b\right)\right)$, over the sets $\mathcal{Q}^i$ and $\mathcal{Q}^j$, if the NF vertices interfere with each other. As a result, it becomes essential to identify the regions within the sets $\mathcal{Q}^i$ and $\mathcal{Q}^j$, in which the function $\ell\left(P'\left(\tilde{n}_i, \tilde{n}_j\right)\right)$ will remain piecewise linear and concave. To this end, we introduce a new concept called *non-interference regions.*

Definition 4. *For any arbitrary placement of the NFs: $E_4(\bar{B}_i) \in int\left(\mathcal{Q}^i\right)$ and $E_4(\bar{B}_j) \in int\left(\mathcal{Q}^j\right)$, we define the following sets:*

1. *$\Delta^i(\mathcal{Q}^i) \subseteq \mathcal{Q}^i$, by keeping NF$_j$ fixed and moving NF$_i$ in all directions, until some vertex $\tilde{n}_i \in D_i$ gets aligned with some vertex $\tilde{n}_j \in D_j$, or $E_4(\bar{B}_i)$ reaches the boundary of the set $\mathcal{Q}^i$.*

2. *$\Delta^j(\mathcal{Q}^j) \subseteq \mathcal{Q}^j$, by keeping NF$_i$ fixed and moving NF$_j$ in all directions, until some vertex $\tilde{n}_i \in D_i$ gets aligned with some vertex $\tilde{n}_j \in D_j$, or $E_4(\bar{B}_j)$ reaches the boundary of the set $\mathcal{Q}^j$.*

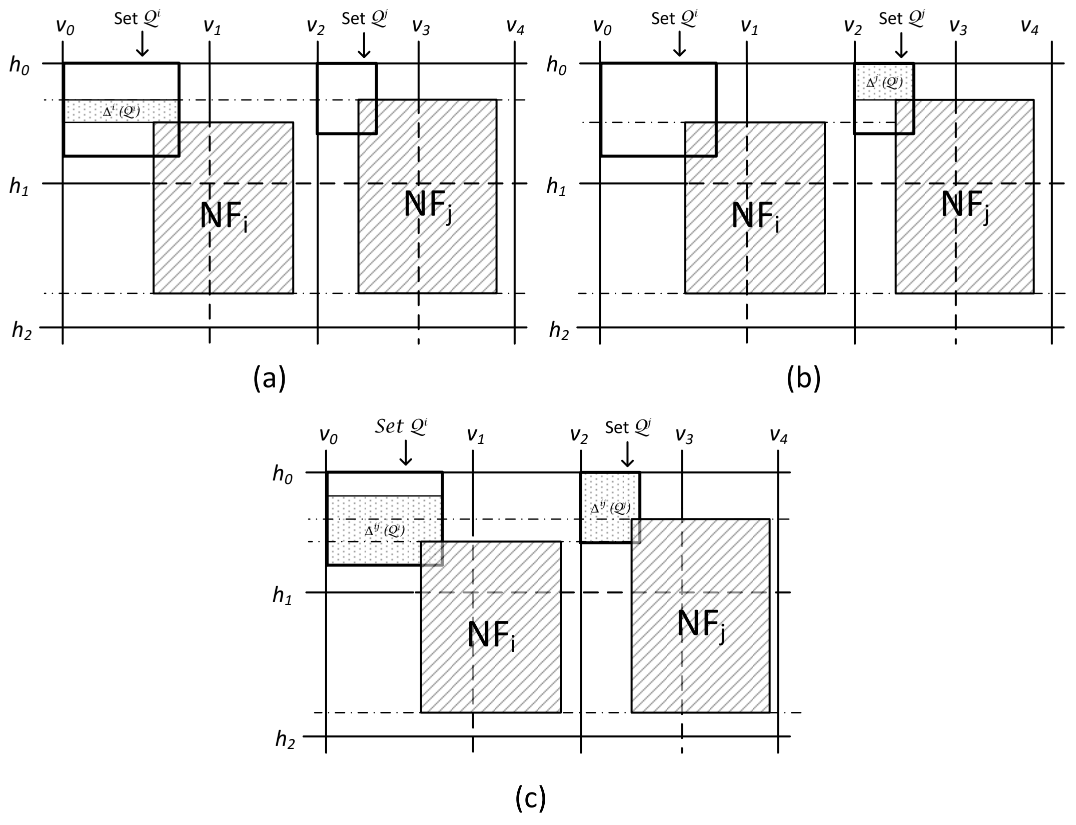
3. *$\Delta^{ij}(\mathcal{Q}^i) \subseteq \mathcal{Q}^i$ and $\Delta^{ij}(\mathcal{Q}^j) \subseteq \mathcal{Q}^j$, which are defined by jointly moving both the NFs in all directions until $E_4(\bar{B}_i)$ reaches the boundary of the set $\mathcal{Q}^i$ or $E_4(\bar{B}_j)$ reaches the boundary of the set $\mathcal{Q}^j$.*

It is easy to see that these Δ sets are rectangular in shape, due to their construction procedure and the rectangular shapes of the NFs/$\mathcal{Q}$ sets. We will call these Δ sets as *non-interference regions* or NIRs of the $\mathcal{Q}$ sets. Figure 2.4 illustrates the construction of the NIRs for one such placement of the NFs.

Lemma 3. *For any path $P(a, b)$ traversing through the vertices of the two NFs, the function $\ell\left(P\left(a, b\right)\right)$ given by Equation (2.7), is a piecewise linear and concave function over the Δ sets constructed as per Definition 4.*

Proof. Consider some placement of the NFs: $E_4(\bar{B}_i) \in \text{int}\left(\mathcal{Q}^i\right)$ and $E_4(\bar{B}_j) \in \text{int}\left(\mathcal{Q}^j\right)$, for which we construct the Δ sets. From Remark 2, we can conclude that the function $\ell(P'(a, \tilde{n}_i)) + \ell(P'(\tilde{n}_j, b))$ is piecewise linear and concave over any of the Δ sets.

1. If we keep NF$_j$ placement fixed, then from the definition of the set $\Delta^i(\mathcal{Q}^i)$, the coordinates of the vertices $\tilde{n}_i$ and $\tilde{n}_j$ remain ordered. Therefore, we have $x_{\tilde{n}_i} \leq x_{\tilde{n}_j}$ or $x_{\tilde{n}_i} \geq x_{\tilde{n}_j}$ but not both, $\forall E_4(\bar{B}_i) \in \Delta^i(\mathcal{Q}^i)$. Similarly, we have $y_{\tilde{n}_i} \leq y_{\tilde{n}_j}$ or $y_{\tilde{n}_i} \geq y_{\tilde{n}_j}$, but not both, $\forall E_4(\bar{B}_i) \in \Delta^i(\mathcal{Q}^i)$. Therefore, the terms $\left|x_{\tilde{n}_i} - x_{\tilde{n}_j}\right|$ and $\left|y_{\tilde{n}_i} - y_{\tilde{n}_j}\right|$ from Equation (2.8), are linear functions which increase or decrease monotonically as NF$_i$ is moved within the set $\Delta^i(\mathcal{Q}^i)$, while keeping NF$_j$ fixed. Consequently, the function $\ell\left(P\left(a, b\right)\right)$ is also a piecewise linear and concave function over $\Delta^i(\mathcal{Q}^i)$, for a fixed placement of NF$_j$.



(a)
(b)
(c)
v_0
v_1
v_2
v_3
v_4
h_0
h_1
h_2
Set $\mathcal{Q}^i$
Set $\mathcal{Q}^j$
$\Delta^i(\mathcal{Q}^i)$
$\Delta^j(\mathcal{Q}^j)$
$\Delta^{ij}(\mathcal{Q}^i)$
$\Delta^{ij}(\mathcal{Q}^j)$
NF$_i$
NF$_j$

Figure 2.4: Construction of non-interference regions: (a) Set $\Delta^i(\mathcal{Q}^i)$; (b) Set $\Delta^j(\mathcal{Q}^j)$; (c) Sets $\Delta^{ij}(\mathcal{Q}^i)$ and $\Delta^{ij}(\mathcal{Q}^j)$

2. Using the similar arguments, we can show that the function $\ell\left(P\left(a,b\right)\right)$ is a piecewise linear and concave function over $\Delta^j(\mathcal{Q}^j)$, for a fixed placement of NF$_i$.

3. Recall that for constructing the sets $\Delta^{ij}(\mathcal{Q}^i)$ and $\Delta^{ij}(\mathcal{Q}^j)$, both NFs are moved simultaneously as a single entity. Therefore, the function $\ell(P'(\tilde{n}_i,\tilde{n}_j))$ remains constant. Hence, the function $\ell\left(P\left(a,b\right)\right)$ is the sum of a linear function and a constant, which makes it piecewise linear and concave over the sets $\Delta^{ij}(\mathcal{Q}^i)$ and $\Delta^{ij}(\mathcal{Q}^j)$. The lemma follows.

□

Remark 4. *In any given layout, there exist multiple nodal paths traversing through different combinations of NF vertices. The way the NIRs are defined, it is guaranteed that the coordinates of all the interfering NF vertices remain ordered. This, in turn, guarantees that the lengths of all the nodal paths, that make use of some subset of NF vertices, will remain piecewise linear and concave, over any Δ set.*

Lemma 4. *For the two facility placement problem (i.e., for $M = 2$), the objective function $J(\mathbf{p})+$*

$K(\mathbf{p}) + L(\mathbf{p})$ given by Equation (2.2), is a concave function over the Δ sets constructed as per Definition 4.

Proof. For the sets $\mathcal{Q}^i$ and $\mathcal{Q}^j$, if none of the NF vertices are interfering, then for any nodal path $P(a,b)$, the function $\ell(P(a,b))$ either remains constant, or it is piecewise linear and concave over the sets $\mathcal{Q}^i$ and $\mathcal{Q}^j$ (Result 5 and/or Lemma 1). In either case, the function $d_{\mathbf{p}}(a,b)$ given by Equation (2.6), is the minimum of a finite number of concave functions, and therefore, it is concave over the sets $\mathcal{Q}^i$ and $\mathcal{Q}^j$. Consequently, the function $J(\mathbf{p})+K(\mathbf{p})+L(\mathbf{p})$, which is the non-negative weighted sum of concave functions, is also a concave function over the entire sets $\mathcal{Q}^i$ and $\mathcal{Q}^j$, and consequently it is concave over any Δ set.

Now let us consider the case where some of the NF vertices are interfering. Then for some feasible placement of the NFs: $E_4(\bar{B}_i) \in \text{int}\left(\mathcal{Q}^i\right)$ and $E_4(\bar{B}_j) \in \text{int}\left(\mathcal{Q}^j\right)$, we construct the Δ sets as per Definition 4. For any nodal path $P(a,b)$, the function $\ell(P(a,b))$ either remains constant, or from Lemma 3, it is linear and concave over the Δ sets. Therefore, the function $d_{\mathbf{p}}(a,b)$ and the objective function $J(\mathbf{p}) + K(\mathbf{p}) + L(\mathbf{p})$ are concave functions over the Δ sets. The lemma follows. $\square$

Using the above results, we can design Algorithm 1, which will enable us to obtain non-dominated solutions to the finite-size facility placement problem.

Lemma 5. *Algorithm 1 converges to a local minimum, after a finite number of iterations.*

Proof. Indeed, after moving one or both NFs to the corners of the appropriately constructed Δ sets, the objective function value decreases or remains the same. Therefore, Procedure 1 produces a non-increasing sequence of Φ values. This sequence is bounded from below by the optimal objective value $\Phi^* \geq 0$. Therefore, we cannot decrease the objective function value indefinitely, and the procedure either converges to a local minimum or cycles without improving the objective function value. In case the procedure cycles, it goes from one solution to the other. Because of the way the Δ sets are constructed, it can be deduced that any such solution has the following characteristics: (1) both NFs are placed at some corner of their respective $\mathcal{Q}$ sets; or (2) one NF is placed at the corner of its $\mathcal{Q}$ set, and the other NF is placed at a point where some vertex $\tilde{n}_i \in D_i$ is aligned with some vertex $\tilde{n}_j \in D_j$. Indeed, there are a finite number of $\mathcal{Q}$ set corners and any NF has a finite number of vertices (at most 5). Therefore, when one NF (say NF$_i$) is placed at a corner of the set $\mathcal{Q}^i$, there is only a finite number of points within the set $\mathcal{Q}^j$, at which some vertex of NF$_i$ can be aligned with some vertex of NF$_j$. Therefore, even if the algorithm cycles, it will go through a finite number of these solutions, before it can be terminated. The lemma follows. $\square$

The above lemma provides us with means to construct all the non-dominated solutions, without having to resort to Algorithm 1. For this purpose we introduce a new concept called *interference lines* or ILs. Consider some initial feasible placement of the NFs: $E_4(\bar{B}_i)_{ini} \in \text{int}(\mathcal{Q}^i)$ and $E_4(\bar{B}_j)_{ini} \in \text{int}(\mathcal{Q}^j)$, such that the vertices $\tilde{n}_i$ and $\tilde{n}_j$ are aligned, i.e., $x_{\tilde{n}_i} = x_{\tilde{n}_j}$ or $y_{\tilde{n}_i} = y_{\tilde{n}_j}$.

Algorithm 1: Procedure for generating non-dominated solutions.

Consider some initial feasible placement of the NFs: $E_4(\bar{B}_i)_{ini} \in \text{int}(\mathcal{Q}^i)$ and $E_4(\bar{B}_j)_{ini} \in \text{int}(\mathcal{Q}^j)$, which has an objective function value of Φ_1. Let us assume that the vertices of the two NFs interfere along the y-axis.

1. For this particular placement, we construct the sets $\Delta^{ij}(\mathcal{Q}^i)$ and $\Delta^{ij}(\mathcal{Q}^j)$. From Lemma 4, we know that the objective function is concave over these sets, for any joint movement of the two NFs. Therefore, we can move the NFs jointly to some corner of the Δ^{ij} sets, if it has the same or better objective function value $\Phi_2 \leq \Phi_1$. From the construction of the Δ^{ij} sets, we know that at least one of the NFs is placed at top or bottom boundary of its $\mathcal{Q}$ set.

2. Suppose that NF$_j$ is placed at the boundary of the set $\mathcal{Q}^j$ and NF$_i$ is still in the interior of the set $\mathcal{Q}^i$. Now we construct the set $\Delta^i(\mathcal{Q}^i)$, by fixing the placement of NF$_j$ and only moving NF$_i$. Again, from Lemma 4, we know that the objective function is concave over this set, for any movement of NF$_i$. Therefore, we can move NF$_i$ to some corner of the set $\Delta^i(\mathcal{Q}^i)$, if it has the same or better objective function value $\Phi_3 \leq \Phi_2$. From the construction of the set $\Delta^i(\mathcal{Q}^i)$, we know that NF$_i$ is placed at the boundary of the set $\mathcal{Q}^i$ or it is placed at a point where a vertex $\tilde{n}_i \in D_i$ gets aligned with a vertex $\tilde{n}_j \in D_j$.

3. At this point, we are either at a local minimum, or we can keep repeating this procedure until we get to a local minimum. If we start from a different initial placement of the NFs or perform a different set of moves than the ones mentioned above, then we may converge to a different local minimum.

Finite convergence of this procedure is proved in Lemma 5.

Since the NFs are placed in the interior of the $\mathcal{Q}$ sets, the boundary of any of the NFs does not coincide with the existing gridlines. Let us construct the sets $\Delta^{ij}(\mathcal{Q}^i)$ and $\Delta^{ij}(\mathcal{Q}^j)$ by moving the NFs simultaneously in all directions, from the initial locations $E_4(\bar{B}_i)_{ini}$ and $E_4(\bar{B}_j)_{ini}$. During the construction of these Δ sets, when the movement of one of the NFs is blocked by the boundary of its $\mathcal{Q}$ set, the movement of other NF is also blocked, and we can define a line $\mathcal{L}^{\tilde{n}_i\tilde{n}_j}(\mathcal{Q}^i) \subset \mathcal{Q}^i$ or $\mathcal{L}^{\tilde{n}_i\tilde{n}_j}(\mathcal{Q}^j) \subset \mathcal{Q}^j$. We shall refer to these lines as *interference lines* or ILs of the sets $\mathcal{Q}^i$ and $\mathcal{Q}^j$, corresponding to the vertices $\tilde{n}_i$ and $\tilde{n}_j$. This procedure needs to be repeated for every pair of interfering vertices of the two NFs. As one might have guessed, these ILs are precisely the non-dominated solutions in the interior of the $\mathcal{Q}$ sets.

An alternate method for constructing the ILs is to fix the placement of one of the NFs at the corners of its $\mathcal{Q}$ set (one at a time); construct new gridlines for each such placement; and then identify the partitions in the $\mathcal{Q}$ set of the other NF that get created due to those new gridlines. These partitions represent the ILs for the second NF. The procedure needs to be repeated by changing the sequence of NF placement, to obtain the ILs for the first NF. At the end, we obtain all the required non-dominated solutions for the $\mathcal{Q}$ set pair. Figure 2.5 illustrates the construction of ILs, for the possible alignment of the top left corners $E_4(\bar{B}_i)$ and $E_4(\bar{B}_j)$, and also the possible

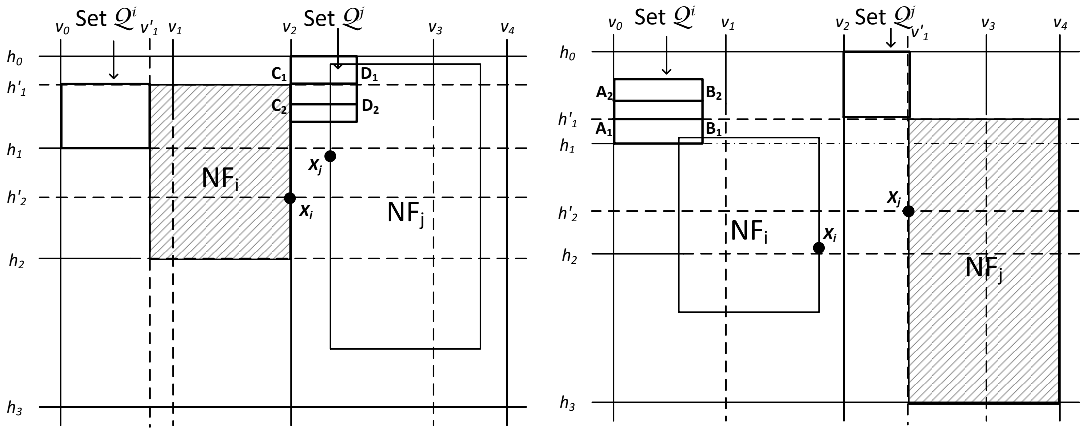


Figure 2.5: Construction of interference lines

alignment of the I/O points X_i and X_j. $\overline{A_1B_1} = \mathcal{L}^{E_4(\bar{B}_i)E_4(\bar{B}_j)}(\mathcal{Q}^i)$, $\overline{C_1D_1} = \mathcal{L}^{E_4(\bar{B}_i)E_4(\bar{B}_j)}(\mathcal{Q}^j)$, $\overline{A_2B_2} = \mathcal{L}^{X_iX_j}(\mathcal{Q}^i)$, and $\overline{C_2D_2} = \mathcal{L}^{X_iX_j}(\mathcal{Q}^j)$ represent the ILs of the sets $\mathcal{Q}^i$ and $\mathcal{Q}^j$. If the interference had existed along the x-axis, then we would have obtained vertical ILs, instead of horizontal ones.

Theorem 1. *For the two facility placement problem (i.e., for $M = 2$), the objective function $J(\mathbf{p}) + K(\mathbf{p}) + L(\mathbf{p})$ given by Equation (2.2) is minimized by placing the NFs at one of the corners of the sets $\mathcal{Q}^i$, $\mathcal{Q}^j$ or the vertices of the ILs $\mathcal{L}(\mathcal{Q}^i)$, $\mathcal{L}(\mathcal{Q}^j)$.*

Proof. For the sets $\mathcal{Q}^i$ and $\mathcal{Q}^j$, if none of the NF vertices are interfering, then from Lemma 4, the objective function $J(\mathbf{p}) + K(\mathbf{p}) + L(\mathbf{p})$ is concave over the sets $\mathcal{Q}^i$ and $\mathcal{Q}^j$. Therefore, it is minimized when the NFs are placed at some corners of those sets.

For the sets $\mathcal{Q}^i$ and $\mathcal{Q}^j$, if some of the NF vertices interfere with each other, then we construct the ILs: $\mathcal{L}(\mathcal{Q}^i)$ and $\mathcal{L}(\mathcal{Q}^j)$ for all the possible alignments of the interfering NF vertices. In Procedure 1 and Lemma 5, we have shown that starting from any solution in the interior of the sets $\mathcal{Q}^i$ and $\mathcal{Q}^j$, we can construct another solution at the corners of the sets $\mathcal{Q}^i$ and $\mathcal{Q}^j$ or at the vertices of the ILs $\mathcal{L}(\mathcal{Q}^i)$ and $\mathcal{L}(\mathcal{Q}^j)$. This new solution has the same or better objective value than that of the interior point solution, and therefore, it is one of the non-dominated solutions. The proof is complete.

$\square$

To summarize, the interior points of the sets $\mathcal{Q}^i$ and $\mathcal{Q}^j$ are dominated by their corner points, and by the vertices of the ILs $\mathcal{L}(\mathcal{Q}^i)$ and $\mathcal{L}(\mathcal{Q}^j)$, and they represent valid candidates for the optimal placement of the two NFs. To find the optimal solution, we need to place the NFs at all possible pairs of these candidate points, evaluate the objective function value for each placement, and select the candidate pair with the minimum value. Thus, the two facility placement problem can be reduced to a discrete search problem for a pair of candidate points.

2.4.2 Analysis for the Case of Three NFs

Now let us analyze the problem of optimal placement of three NFs. Consider the placement of three NFs: NF$_i$, NF$_j$, and NF$_k$, within the sets $\mathcal{Q}^i$, $\mathcal{Q}^j$, and $\mathcal{Q}^k$, respectively. For this particular placement of the NFs, there exist multiple nodal paths that may traverse through some subset of the NF vertices, some of which might encounter non-concavity due to interference of the associated vertices. Therefore, we need to consider the following cases, based on the interference of NF vertices.

- If none of the NF pairs have interfering vertices, then the lengths of all the paths remain linear and concave over the sets $\mathcal{Q}^i$, $\mathcal{Q}^j$, and $\mathcal{Q}^k$; and the corners of these sets represent the candidates for the optimal placement of the NFs.

- If exactly one NF pair has interfering vertices (say NF$_i$ and NF$_j$), then the only paths/subpaths that may encounter non-concavity are the ones which traverse through the vertices of these two NFs. This case is similar to the one discussed in the previous section. We construct the ILs $\mathcal{L}(\mathcal{Q}^i)$ and $\mathcal{L}(\mathcal{Q}^j)$ for the alignment of different vertices of NF$_i$ and NF$_j$. Then the corners of the sets $\mathcal{Q}^i$, $\mathcal{Q}^j$, $\mathcal{Q}^k$, and the vertices of the ILs $\mathcal{L}(\mathcal{Q}^i)$ and $\mathcal{L}(\mathcal{Q}^j)$ represent the candidates for the optimal placement of the NFs.

- If two or more NF pairs have interfering vertices, then there exist some paths that traverse through the vertices of all three NFs, which may encounter non-concavity. To deal with the non-concavity of these nodal paths, we will extend the concepts of NIRs and ILs discussed in the previous section.

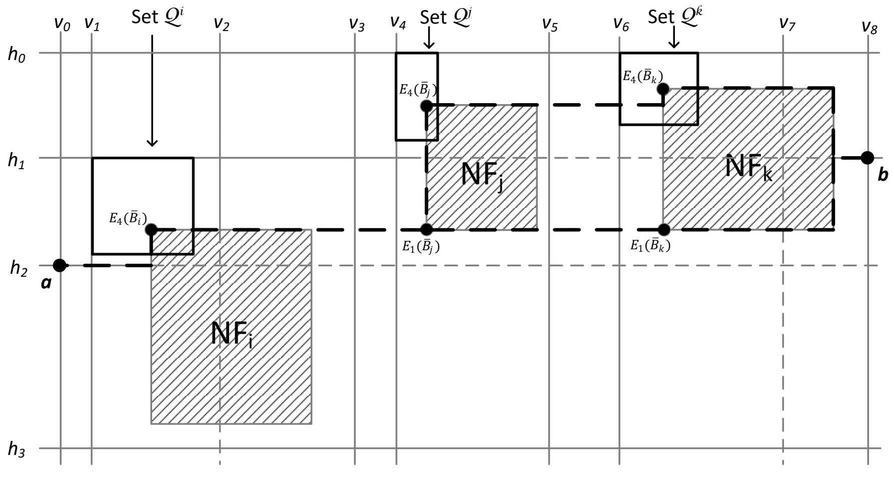


Figure 2.6: Layout with three NFs

To this end, let us consider the following two generic nodal paths, which represent all the nodal paths that traverse through the vertices of the three NFs.

1. A path $P(a,b)$, which traverses through the vertices $\tilde{n}_i \in D_i$, $\tilde{n}_j \in D_j$, and $\tilde{n}_k \in D_k$, in the specified order. The expression for the length of this path can be written as:

$$\ell\left(P\left(a,b\right)\right) = \ell(P'(a,\tilde{n}_i)) + \ell(P'(\tilde{n}_i,\tilde{n}_j)) + \ell(P'(\tilde{n}_j,\tilde{n}_k)) + \ell(P'(\tilde{n}_k,b)). \tag{2.9}$$

   The function $\ell(P(a,b))$ might become non-concave if the vertex $\tilde{n}_i$ interferes with the vertex $\tilde{n}_j$ and/or the vertex $\tilde{n}_j$ interferes with the vertex $\tilde{n}_k$. The path a–$E_4(\bar{B}_i)$–$E_1(\bar{B}_j)$–$E_1(\bar{B}_k)$–b in Fig. 2.6 is an example of this type of nodal path.

2. A path $P(a',b')$ which traverses through the vertices $\tilde{n}'_i \in D_i$, $\tilde{n}'_j, \tilde{n}''_j \in D_j$, and $\tilde{n}'_k \in D_k$, in the specified order. The expression for the length of this path can be written as:

$$\ell\left(P\left(a',b'\right)\right) = \ell(P'(a',\tilde{n}'_i)) + \ell(P'(\tilde{n}'_i,\tilde{n}'_j)) + \ell(P'(\tilde{n}'_j,\tilde{n}''_j)) + \ell(P'(\tilde{n}''_j,\tilde{n}'_k)) + \ell(P'(\tilde{n}'_k,b')). \tag{2.10}$$

   The function $\ell(P(a',b'))$ might become non-concave if the vertex $\tilde{n}'_i$ interferes with the vertex $\tilde{n}'_j$ and/or the vertex $\tilde{n}''_j$ interferes with the vertex $\tilde{n}'_k$. The path a–$E_4(\bar{B}_i)$–$E_1(\bar{B}_j)$–$E_4(\bar{B}_j)$–$E_4(\bar{B}_k)$–b in Fig. 2.6 is an example of this type of nodal path.

Remark 5. *Without the loss of generality, the nodes a, b, a', and b' can be assumed to be existing cell corners. Therefore, the functions $\ell(P'(a,\tilde{n}_i)) + \ell(P'(\tilde{n}_k,b))$ and $\ell(P'(a',\tilde{n}'_i)) + \ell(P'(\tilde{n}'_k,b'))$ are linear and concave functions over the sets $\mathcal{Q}^i$ and $\mathcal{Q}^k$.*

Definition 5. *For any arbitrary placement of the NFs: $E_4(\bar{B}_i) \in int\left(\mathcal{Q}^i\right)$, $E_4(\bar{B}_j) \in int\left(\mathcal{Q}^j\right)$, and $E_4(\bar{B}_k) \in int\left(\mathcal{Q}^k\right)$, we define the following sets:*

1. *$\Delta^i(\mathcal{Q}^i)$, is defined by keeping NF_j and NF_k fixed and moving NF_i in all directions, until (1) $E_4(\bar{B}_i)$ reaches the boundary of the set $\mathcal{Q}^i$; or (2) some vertex $\tilde{n}_i \in D_i$ gets aligned with some vertex $\tilde{n}_j \in D_j$ or $\tilde{n}_k \in D_k$. Sets $\Delta^j(\mathcal{Q}^j)$ and $\Delta^k(\mathcal{Q}^k)$ are defined in the similar fashion.*

2. *$\Delta^{ij}(\mathcal{Q}^i)$ and $\Delta^{ij}(\mathcal{Q}^j)$, which are defined by keeping NF_k fixed, and jointly moving NF_i and NF_j in all directions, until: (1) $E_4(\bar{B}_i)$ reaches the boundary of the set $\mathcal{Q}^i$ or $E_4(\bar{B}_j)$ reaches the boundary of the set $\mathcal{Q}^j$; or (2) some vertex $\tilde{n}_i \in D_i$ or $\tilde{n}_j \in D_j$ gets aligned with some vertex $\tilde{n}_k \in D_k$. Sets $(\Delta^{jk}(\mathcal{Q}^j), \Delta^{jk}(\mathcal{Q}^k))$ and $(\Delta^{ik}(\mathcal{Q}^i), \Delta^{ik}(\mathcal{Q}^k))$ are defined in the similar fashion.*

3. *$\Delta^{ijk}(\mathcal{Q}^i)$, $\Delta^{ijk}(\mathcal{Q}^j)$, and $\Delta^{ijk}(\mathcal{Q}^k)$ which are defined by jointly moving all the three NFs, in all directions until $E_4(\bar{B}_i)$ reaches the boundary of the set $\mathcal{Q}^i$ or $E_4(\bar{B}_j)$ reaches the boundary of the set $\mathcal{Q}^j$ or $E_4(\bar{B}_k)$ reaches the boundary of the set $\mathcal{Q}^k$.*

Now we state the following results without any proof. These results are the extensions of Lemmas 3 and 4 and the main argument in their proof is the fact that the coordinates of any pair of NF vertices remain ordered within any of the Δ sets.

Corollary 1. *For any path $P(a,b)$ traversing through the vertices of any two NFs, the function $\ell\left(P\left(a,b\right)\right)$ given by Equation (2.7) is a linear and concave function over the Δ sets constructed as per Definition 5.*

Lemma 6. *For any path $P(a,b)$ and $P(a',b')$ traversing through the vertices of all the three NFs, the functions $\ell\left(P\left(a,b\right)\right)$ given by Equation (2.9) and $\ell\left(P\left(a',b'\right)\right)$ given by Equation (2.10), are linear and concave functions over the Δ sets constructed as per Definition 5.*

Lemma 7. *For the three facility placement problem (i.e., for $M = 3$), the objective function $J(\mathbf{p}) + K(\mathbf{p}) + L(\mathbf{p})$ given by Equation (2.2), is a concave function over the Δ sets constructed as per Definition 5.*

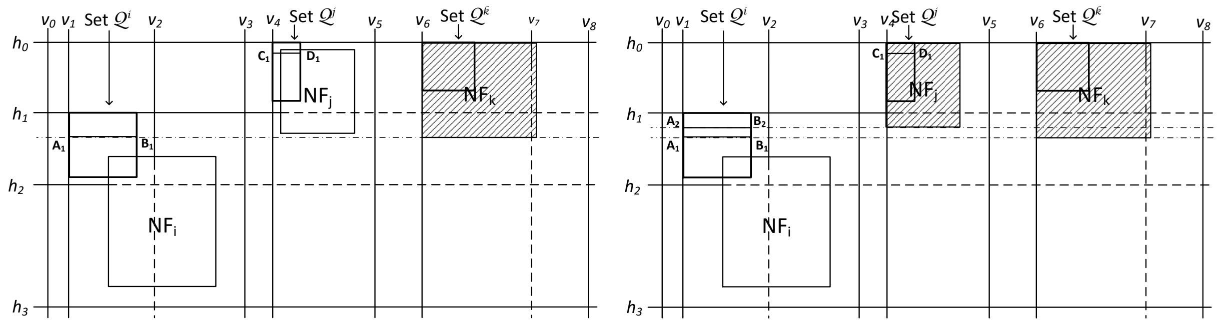
Now let us try and determine the non-dominated solutions for the placement of three NFs, following a similar procedure as Algorithm 1. We start with an arbitrary placement of the NFs within the interior of the respective $\mathcal{Q}$ sets. Then we can move one, two, or all three NFs within the appropriately constructed Δ sets, to obtain a sequence of non-increasing objective function values, until no further improvement is possible. Once again, we can claim that this procedure terminates in a finite number of iterations and the solution converges to a local minimum. The way the Δ sets are constructed, it can be deduced that any such local minimum has the following characteristics: (1) All three NFs are placed at some corners of their respective $\mathcal{Q}$ sets; or (2) Two NFs (say NF$_i$ and NF$_j$) are placed at some corner of their respective $\mathcal{Q}$ sets, and the third NF (NF$_k$) is placed at a point where some vertex $\tilde{n}_k \in D_k$ is aligned with some vertex $\tilde{n}_i \in D_i$ or with some vertex $\tilde{n}_j \in D_j$; or (3) One NF (say NF$_i$) is placed at some corner of its $\mathcal{Q}$ set and the remaining two NFs (NF$_j$ and NF$_k$) are placed at points such that some vertex $\tilde{n}_j \in D_j$ and/or $\tilde{n}_k \in D_k$ is aligned with some vertex $\tilde{n}_i \in D_i$ and some vertex $\tilde{n}'_j \in D_j$ may be aligned with some vertex $\tilde{n}'_k \in D_k$. All the solutions that satisfy the above properties represent non-dominated solutions for the placement of the two NFs. Therefore, we can modify the IL construction procedure to construct these solutions, as follows. We consider a particular placement permutation of the NFs (from the set of 6 possible permutations), and for that permutation:

1. We place the first NF in the sequence at the corners of its $\mathcal{Q}$ set (one at a time) and construct the new gridlines for each such placement.

2. Then we obtain the partitions in the $\mathcal{Q}$ set of the second NF, that are created by the new gridlines. These partitions represent the ILs for the second NF. We place the second NF at all the corners of its $\mathcal{Q}$ set and ILs (one at a time), and again construct the new gridlines for each such placement.

3. Finally, we obtain the partitions in the $\mathcal{Q}$ set of the third NF, that are created by the new gridlines. These partitions represent the ILs for the third and final NF.

This procedure needs to be repeated for all six placement permutations. At the end, we obtain all the required non-dominated solutions for the $\mathcal{Q}$ set triplet.



v_0 v_1 Set Q^i v_2 v_3 v_4 Set Q^j v_5 v_6 Set Q^k v_7 v_8
h_0
h_1
h_2
h_3
C_1
D_1
NF_j
NF_k
A_2
B_2
A_1
B_1
NF_i

Figure 2.7: ILs for three NFs

As an example consider the layout shown in Fig. 2.7, which are constructed for the sequence NF$_k$ – NF$_j$ – NF$_i$. The ILs $\overline{A_1B_1}$, $\overline{A_2B_2}$, and $\overline{C_1D_1}$ represent the ILs for the sets $\mathcal{Q}^i$ and $\mathcal{Q}^j$, when NF$_k$ is placed at the top left corner of the set $\mathcal{Q}^k$ and NF$_j$ is placed at top left corner of the set $\mathcal{Q}^j$. Similar ILs need to be constructed for the placement of the NFs at other corners of the respective $\mathcal{Q}$ sets, and also for all the remaining placement permutations. All these ILs represent the non-dominated solutions for the simultaneous alignment of the interfering vertices of all three NFs.

Theorem 2. *For the three facility placement problem (i.e., for $M = 3$), the objective function $J(\mathbf{p}) + K(\mathbf{p}) + L(\mathbf{p})$ given by Equation (2.2), is minimized by placing the NFs at one of the corners of the sets $\mathcal{Q}^i$, $\mathcal{Q}^j$, and $\mathcal{Q}^k$ or the vertices of the ILs $\mathcal{L}(\mathcal{Q}^i)$, $\mathcal{L}(\mathcal{Q}^j)$, and $\mathcal{L}(\mathcal{Q}^k)$.*

Proof. We will prove this theorem by induction. For the cases where none of the NF vertices are interfering or vertices of only two NFs are interfering, the theorem holds.

For the case where the vertices of two or more pairs of NFs interfere with each other, we construct the ILs. Now, suppose that in absence of the third NF (say NF$_k$), the remaining two NFs are placed at some local minimum, such that NF$_i$ is at the corner of the set $\mathcal{Q}^i$ and NF$_j$ is at the vertex of an IL, such that $\tilde{n}_i$ and $\tilde{n}_j$ are aligned. Assuming that the vertices of all the three NFs interfere along the y-axis, we now place NF$_k$ at some interior point within the set $\mathcal{Q}^k$. For this particular placement of the NFs, we construct the set $\Delta^k(\mathcal{Q}^k)$. From Lemma 7, we can get the same or better objective function value by moving NF$_k$ to one of the corners of the set $\Delta^k(\mathcal{Q}^k)$, and therefore this solution represents one of the non-dominated solutions. From the construction, we know that $E_4(\bar{B}_k)$ is either at the corner of the set $\mathcal{Q}^k$, in which case the proof is complete, or it is placed at a point where a vertex $\tilde{n}_k \in D_k$ is aligned with a vertex $\tilde{n}'_i \in D_i$ or $\tilde{n}'_j \in D_j$. But this point is nothing but the vertex of an IL, which would have been constructed due to the new gridline created either by $\tilde{n}'_i$ or by $\tilde{n}'_j$.

□

Thus, the corners of the sets $\mathcal{Q}^i$, $\mathcal{Q}^j$, and $\mathcal{Q}^k$, and the vertices of the ILs $\mathcal{L}(\mathcal{Q}^i)$ and $\mathcal{L}(\mathcal{Q}^j)$, and $\mathcal{L}(\mathcal{Q}^k)$ represent valid candidates for the optimal placement of the three NFs. To find the optimal solution, we need to place the NFs at all possible triplets of these candidate points, evaluate the objective function value for each placement, and select the candidate triplet with the minimum value.

2.4.3 Generalization to the Case of M NFs

Finally, in this section, we will generalize the results from the previous two sections, to the placement of M finite-size facilities. As we have seen before, the interference of the NF vertices plays an important role in determining the non-dominated solutions. Therefore, we can use the vertex interference to simplify the analysis. To this end, let us consider a set $\mathcal{T}$, and for this set, let us define a partition set $\boldsymbol{\mathcal{U}}$. An element $\mathcal{U}_p \in \boldsymbol{\mathcal{U}}$ is a collection of $\mathcal{Q}$ sets such that:

1. The vertices of any NF$_i$ placed in the set $\mathcal{Q}^i \in \mathcal{U}_p$ interfere with the vertices of at least one other NF$_j$ placed in the set $\mathcal{Q}^j \in \mathcal{U}_p$.

2. The vertices of any NF$_i$ placed in the set $\mathcal{Q}^i \in \mathcal{U}_p$ do not interfere with the vertices of any other NF$_k$ placed in the set $\mathcal{Q}^k \in \mathcal{U}_q$.

Therefore we have $\mathcal{U}_p \cap \mathcal{U}_q = \emptyset, \forall \mathcal{U}_p, \mathcal{U}_q \in \boldsymbol{\mathcal{U}}$ and $\bigcup \mathcal{U}_p = \mathcal{T}$. The motivation behind partitioning the $\mathcal{T}$ set is that the optimal placement candidates can be found independently for each partition. This observation follows from the fact that the length of any nodal path traversing between any two partitions remains linear and concave (Lemma 1).

Now, let us consider a partition $\mathcal{U} \in \boldsymbol{\mathcal{U}}$, which contains $m \leq M$ number of $\mathcal{Q}$ sets. For convenience, let us renumber these $\mathcal{Q}$ sets as $\mathcal{Q}^1, \cdots, \mathcal{Q}^m$, and the corresponding NFs as NF$_1, \cdots,$ NF$_m$. Let $[r]$ denote the set of indices of some $r \leq m$ number of $\mathcal{Q}$ sets from $\mathcal{U}$, and let $[m-r]$ denote the set of indices of the remaining $m-r$ number of $\mathcal{Q}$ sets. Then, we can write the following definition for the NIRs.

Definition 6. *For any arbitrary placement of the NFs, we define the sets $\Delta^{[r]}(\mathcal{Q}^i)$, $\forall 2 \leq r \leq m$, $\forall [r] \in \binom{m}{r}$, and $\forall i \in [r]$; by fixing the placements of $m-r$ NFs and jointly moving the selected r NFs in all directions until for some $j \in [r]$ and $k \in [m-r]$, $E_4(\bar{B}_j)$ reaches the boundary of the set $\mathcal{Q}^j$, or a vertex $\tilde{n}_j \in D_j$ gets aligned with a vertex $\tilde{n}_k \in D_k$.*

The following results hold true for the NIRs, due to the fact that the coordinates of the NF vertices remain ordered within any of the Δ sets defined above.

Lemma 8. *For any path $P(a,b)$ traversing through the vertices of 1 to m NFs, the function $\ell\left(P\left(a,b\right)\right)$ given by Equation (2.3) is a linear and concave function over the Δ sets constructed as per Definition 6.*

Lemma 9. *For the m facility placement problem, the objective function $J(\mathbf{p}) + K(\mathbf{p}) + L(\mathbf{p})$ given by Equation (2.2), is a concave function over the Δ sets constructed as per Definition 6.*

To find the non-dominated solutions, we can adapt Algorithm 1 for the placement of m NFs. This procedure will converge to a local minimum in finite number of iterations, in which $0 \leq r \leq m-1$ NFs are placed at some corner of their respective $\mathcal{Q}$ sets, and the remaining $m-r$ NFs are placed at a point where for some $j \in [r]$ and $k \in [m-r]$, a vertex $\tilde{n}_j \in D_j$ gets aligned with a vertex $\tilde{n}_k \in D_k$. All the solutions that satisfy the above properties represent non-dominated solutions for the placement of the m NFs. Thus, the local minima consist of the $\mathcal{Q}$ set corners and/or the vertices of the ILs. These ILs can be constructed using the following procedure. Let $\mathcal{S}_m$ denote the set of all permutations $\sigma : m \to m$ of the NFs. Let $\sigma(i)$ represent the index of the NF that is present in the i^{th} location. Then for a given permutation, we place the first NF in the sequence (i.e., $\text{NF}_{\sigma(1)}$), at all the corners of the set $\mathcal{Q}^{\sigma(1)}$ (one at a time), and construct the new gridlines for each such placement. Then we find the partitions in the set $\mathcal{Q}^{\sigma(2)}$ for the second NF in the sequence (i.e., $\text{NF}_{\sigma(2)}$) that are created by the new gridlines; and so on and so forth. This procedure is repeated for all the permutations $\sigma \in \mathcal{S}_m$ to obtain the required non-dominated solutions for the tuple of m number of $\mathcal{Q}$ sets. Finally we state the following theorem for finding the optimal placement M finite-size facilities. This theorem can be proved using induction, similar to Theorem 2.

Theorem 3. *For the M facility placement problem, the objective function $J(\mathbf{p}) + K(\mathbf{p}) + L(\mathbf{p})$ given by Equation (2.2) is minimized by placing the NFs at one of the corners of the sets $\mathcal{Q}^i \in \mathcal{T}$ or the vertices of the ILs $\mathcal{L}(\mathcal{Q}^i) \subset \mathcal{Q}^i$.*

To find the optimal solution, we need to place the NFs at all possible candidate point tuples of size M, evaluate the objective function value for each placement, and select the candidate tuple with the minimum value. Thus, the M facility placement problem can be reduced to a discrete search problem for a tuple of candidate points.

2.5 Overall Solution Procedure and Complexity Analysis

The explicit enumeration procedure for finding the optimal placement of M facilities can be written as a recursive algorithm. The pseudocode for this algorithm is presented in Algorithms 2 and 3.

```
Algorithm 2: facility_placement
  Data: Problem layout, Placement permutation set $\mathcal{S}_M$
  Result: Optimal solution $\mathbf{p}^*$, Optimal objective value $\Phi^*$
  $\mathbf{p} \leftarrow \emptyset$; $\mathbf{p}^* \leftarrow \emptyset$; $\Phi^* \leftarrow \infty$ ;          /* initialization */
  foreach $\sigma \in \mathcal{S}_M$ do
      place_and_evaluate(1, $\sigma(1)$, $\mathbf{p}$, $\mathbf{p}^*$, $\Phi^*$);
  end
  print $\mathbf{p}^*$, $\Phi^*$;
```

The algorithm selects an NF placement permutation from the set of permutations. Then it iteratively constructs the $\mathcal{Q}$ sets for each NF in the sequence; places the NF at one of the feasible

Algorithm 3: place_and_evaluate($i, \sigma(i), \mathbf{p}, \mathbf{p}', \Phi'$)

```
if $i = M + 1$ then                                                     /* end of recursion */
    calculate: $\Phi(\mathbf{p}) = J(\mathbf{p}) + K(\mathbf{p}) + L(\mathbf{p})$;
    if $\Phi(\mathbf{p}) < \Phi'$ then $\Phi' \leftarrow \Phi(\mathbf{p})$; $\mathbf{p}' \leftarrow \mathbf{p}$;         /* update incumbent solution */
else
    construct $\boldsymbol{\mathcal{Q}}^{\sigma(i)} = \left\{\mathcal{Q}_1^{\sigma(i)}, \mathcal{Q}_2^{\sigma(i)}, \cdots\right\}$ ;                     /* see Section 2.6.4 */
    foreach $\mathcal{Q}^{\sigma(i)} \in \boldsymbol{\mathcal{Q}}^{\sigma(i)}$ do
        foreach 'corner' of $\mathcal{Q}^{\sigma(i)}$ do
            place NF$^{\sigma(i)}$ at 'corner';
            $\mathbf{p}(\sigma(i)) \leftarrow$ 'corner' ;                         /* update placement vector */
            modify gridlines;
            call: place_and_evaluate($i+1$, $\sigma(i+1)$, $\mathbf{p}$, $\mathbf{p}'$, $\Phi'$) ;  /* repeat for next NF */
        end
    end
end
print $\mathbf{p}^*$, $\Phi^*$;
```

candidate points; and constructs the new gridlines. Once it finds a feasible tuple for all M NFs, it evaluates the objective function value and stores the feasible solution if its objective function value is less than that of the incumbent solution. At termination, the algorithm outputs the optimal solution and the corresponding objective function value.

The solution complexity of this explicit enumeration procedure can be analyzed as follows. Since there are N existing facilities in the layout, the upper bound on the number of horizontal and vertical gridlines is $O(N)$ respectively, and the upper bound on the number of cell corners is $O(N^2)$. Let β denote the maximum number of horizontal or vertical gridlines intersected by an NF. Because of the successive gridline intersection, the maximum number of $\mathcal{Q}$ sets generated will be $O(\beta N)$ for horizontal gridlines and $O(\beta N)$ for vertical gridlines. Therefore, for a permutation $\sigma \in \mathcal{S}_M$ and for a facility NF$_{\sigma(i)}$, the upper bound on the number of $\mathcal{Q}^{\sigma(i)}$ sets is $O(\beta^2 N^2)$. Hence, the upper bound on the number of candidate tuples is $O\left(\beta^2 N^2\right) \times O\left(\beta^2 (N+1)^2\right) \times \cdots \times O\left(\beta^2 (N+M-1)^2\right) = O\left(\left(\beta^2 (N+M)^2\right)^M\right)$. Since there are $M!$ permutations, in the worst case, we need to evaluate $O\left(M! \left(\beta^2 (N+M)^2\right)^M\right)$ candidate tuples to get the optimal solution.

Let us consider some special cases of this problem, which might be considerably easy to solve.

Case 1. If all the NFs are identical (same dimensions and same I/O point location), then any permutation of the NFs will yield the same set of feasible placement candidates. In this case, we can get rid of the $M!$ multiplier and the solution complexity reduces to $O\left(\left(\beta^2 (N+M)^2\right)^M\right)$.

Case 2. First, let us consider the case where the NFs are infinitesimal in size. Then, we can treat these NFs as rectangular facilities with arbitrarily small dimensions ($\epsilon > 0$), and we can invoke our explicit enumeration procedure to find the global optimal solution. In fact, we can show that as $\epsilon \to 0$, the NFs can be placed arbitrarily close to each other, and therefore, the gridline intersection points themselves become the candidates for the optimal placement of the NFs. Moreover, any one of the $M!$ permutations will give us the optimal solution, and the solution complexity reduces to $O(N^{2M})$. Alternatively, this problem can be modeled and solved as an instance of the Quadratic Semi-Assignment Problem (QSAP) as follows. Since the NFs are infinitesimal in size, they do not affect the rectilinear paths between the EF I/O points, and therefore, the EF–EF interaction remains constant. Additionally, the distances between the gridline intersection points can be calculated in advance. Therefore, this problem is equivalent to assigning M NFs to the $O(N^2)$ locations, and its objective function has a constant term (EF–EF interaction), a linear term (EF–NF interaction), and a quadratic term (NF–NF interaction), making it an instance of QSAP. The main advantage of solving this problem as QSAP is that we can use some implicit enumeration procedures like the *branch-and-bound*, which might require less computational effort if coupled with a strong lower bound. The formulation for this M-NF QSAP can be written as follows:

$$M\text{-NF QSAP: } \min \sum_{i=1}^{M} \sum_{p=1}^{O(N^2)} b_{ip} x_{ip} + \sum_{i=1}^{M} \sum_{j=1}^{M} \sum_{p=1}^{O(N^2)} \sum_{q=1}^{O(N^2)} w_{ij} d_{pq} x_{ip} x_{jq}; \tag{2.11}$$

$$\text{s.t.} \sum_{p=1}^{O(N^2)} x_{ip} = 1 \quad \forall i = 1, \ldots, M; \tag{2.12}$$

$$x_{ip} \in \{0,1\} \quad \forall i = 1, \ldots, M;\ \forall p = 1, \ldots, O(N^2). \tag{2.13}$$

The details of this formulation are presented in Sections 3.5 and 5.6. It is important to note that a lower bound on this M-NF QSAP is also a valid lower bound on any feasible placement of the M finite-size facilities in the layout.

Case 3. In Case 2, if the NF–NF interactions are absent, then the problem becomes much easier to solve. To find the optimal solution, we only need to find the optimal 1-median location of all M NFs, one at a time. Readers should note that this is precisely the problem studied by Larson and Sadiq (1983). This problem can be solved in polynomial time, and its solution complexity is $O(M \cdot N^2)$.

Case 4. Finally, we can treat the NFs as *forbidden regions* (as opposed to barriers), so that travel is permitted inside the NFs, but the overlap restrictions must be obeyed. This case may have applications in designing multi-layered circuit boards (where the conductors can be drawn underneath the components), and also in designing facilities with overhead material handling systems. For this case, we first enumerate all the candidates for the infinitesimal I/O points as discussed in Case 2, and sort them in ascending order of their objective function values. Then, we check the feasibility of the finite-sized NFs at those solutions. If a particular placement is feasible (none of the facilities overlap with each other), then we stop this optimal solution to the M facility placement problem. Otherwise we consider the next best solution and so on. In this way, we only need to evaluate $O(N^{2M})$ candidates before we find the optimal solution. It is interesting to note that the feasibility part of this problem is equivalent to a variation of the rectangle packing problem (Huang and Korf, 2013), which is NP-Complete. In other words, the problem that we are studying is a generalization of the rectangle packing/containment problems, which also validates its difficulty.

2.6 Specifics of Implementation

So far we have discussed the theoretical results and established the procedure for finding the optimal placement of finite-size NFs in a layout. However, to be able to implement the procedure in any computer language, we need the help of some algorithms. In this section we will explain these algorithms and some other practical aspects of implementing the procedure on a computer. Note that the algorithms explained in this section use the simplest and most logical data structures. We acknowledge that the complexity of some of the algorithms can be reduced with the use of more complex data structures. The input data consists of the coordinates of the top left corners of the EFs and NFs; the dimensions of the EFs and NFs; the coordinates of the EF I/O points; and the EF–NF, EF–EF, and NF–NF interaction values for the layout. The data is read into the computer memory and a problem is constructed for further processing. This section is reprinted from Date et al. (2014), with permission from Elsevier.

2.6.1 Gridline Construction

The first step is to construct the horizontal and vertical gridlines passing through the vertices of the EFs and the I/O points. The algorithm can be explained as follows:

- First, we construct a set of points, which contains all the EF vertices and the I/O points. These are the fore-bearers of the gridlines.

- From each point in the set, we create four probes in the horizontal and vertical directions. For each probe, we identify the intercepting EFs by comparing their coordinates.

- For a horizontal probe in $-x$ direction, the maximum x-coordinate of the first intercepting EF is used as the leftmost limiting x-coordinate, while for the horizontal probe in $+x$ direction, minimum x-coordinate of the first intercepting EF is used as the rightmost limiting x-coordinate. The limiting y-coordinates for the vertical probes are calculated in the similar fashion. If a probe is not intercepted by any EF, then its limiting coordinates are the coordinates of the layout boundary.

- A horizontal gridline is constructed using the y-coordinate of the point and the two limiting x-coordinates of the horizontal probes created for that point. Similarly, a vertical gridline is constructed using the x-coordinate of the point and the two limiting y-coordinates of the vertical probes created for that point. These new gridlines are added to the respective sets. The process is repeated for all the points in the set.

The complexity of this algorithm is $O(N)$, where N is the number of EFs in the layout.

2.6.2 Network Formation

After the grid is constructed, we transform the layout into a network $\mathcal{G} = (\mathcal{V}, \mathcal{E})$. In this network, $\mathcal{V}$ is the set of *nodes*, which are the EF vertices, EF I/O points or the gridline intersection points; and $\mathcal{E}$ is the set of *edges*, which are segments of either the horizontal or the vertical gridlines. The complexity of the network formation algorithm is $O(N^2)$. The importance of converting the layout to a network can be explained using the following lemma.

Lemma 10. *For the network $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, any two nodes connected by an edge are simply communicating.*

Proof. We know that the nodes in the layout network are the intersection points of the horizontal and vertical gridlines and each edge, connecting a pair of adjacent nodes, is a segment of either a horizontal or a vertical gridline. The lemma follows from the above fact and Definition 2.

□

From Lemma 10 and Result 1, we can conclude that the travel between any two nodes can be restricted to a sequence of vertical and horizontal edges from the edge set $\mathcal{E}$ and we can use a shortest path algorithm such as Dijkstra's Algorithm (Dijkstra, 1959) to find the length of the shortest rectilinear path between those two nodes. Thus, converting the layout to a network proves to be an important step towards calculating the shortest distances between the different I/O points and evaluating the objective function values for the various NF placements.

2.6.3 Cell Formation

Once the layout is converted to a network, we need to construct the set of cells $\mathcal{C}$. Each cell $C \in \mathcal{C}$ is an object bounded by four edges and its vertices are the corresponding network nodes. The algorithm cycles through all the nodes in $\mathcal{V}$ and identifies the four bounding edges using the adjacency information for each node. Once all the four edges are identified, a cell object is created and added to the set $\mathcal{C}$, if it does not overlap with any of the EFs. The complexity of this algorithm is $O(N^2)$.

2.6.4 Identification of $\mathcal{Q}$ sets/ Feasible Placement Candidates

Once we construct the network $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ and identify the set of cells $\mathcal{C}$ we can obtain the feasible placement candidates for the NF. The basic idea behind this algorithm can be explained as follows:

- From the concept of $\mathcal{Q}$ sets explained in Section 2.3.3, it is evident that when an NF is placed at the corner of a $\mathcal{Q}$ set, one of its corners coincides with a cell corner. Conversely, if we place

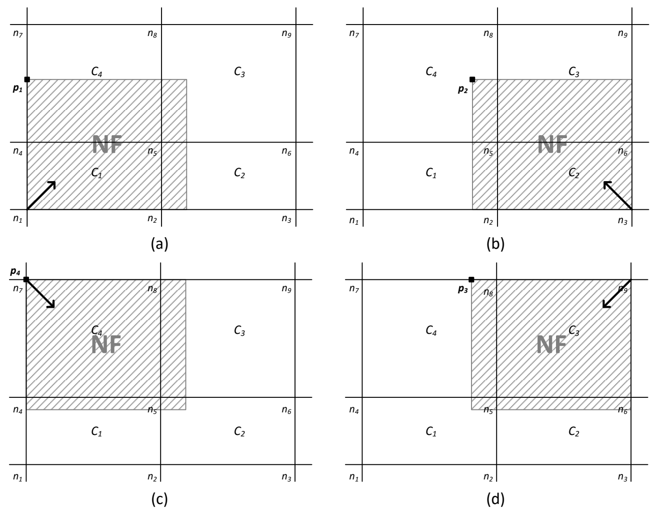


Figure 2.8: Fill directions: (a) NE; (b) NW; (c) SE; (d) SW

the NF such that one of its corners coincides with a cell corner (i.e., network node), we can find the corresponding $\mathcal{Q}$ set corner.

- Consider the layout shown in Fig. 2.8 (a), in which an NF needs to be placed. The layout is comprised of four cells. Let $l(C_i)$ and $w(C_i)$ denote the length and width of a cell C_i. Similarly, let $l(\text{NF})$ and $w(\text{NF})$ denote the length and width of the NF.

- We begin from node n_1 and start filling the NF in the *north-east* direction. We have to check if there is enough space available to place the NF entirely. The first cell in the NE direction of n_1 is C_1, which is added to a *queue*. It is evident that: $w(C_1) < w(\text{NF})$, which means that the NF cannot be completely contained within the cell C_1 and we need additional area to place the NF.

- In the next step, we check if any cells exist to the north of the cell C_1, so that the width of the NF could be extended beyond the cell C_1. We can see that the cell C_4 is present to the north of the cell C_1, which is also added to the *queue*. Now, the total width: $w(C_1)+w(C_4) > w(\text{NF})$, and hence, the width criterion for the NF feasibility is satisfied. If the width of the NF is still greater than the total width, we will have to successively add the adjacent cells in the vertical direction until the criterion is satisfied. If we do not manage to find such cells, then no feasible candidate points are generated for the node n_1. At the end of this step, we will have vertically adjacent cells in the *queue*.

- Next, we check if any cells exist to the east of all the cells in the *queue*, so that the length of the NF could be extended beyond the cell C_1. We can see that the cell C_2 is present to the east of the cell C_1 and the cell C_3 is present to the east of the cell C_4. Now, the total length: $l(C_1) + l(C_2) > l(\text{NF})$ and $l(C_4) + l(C_3) > l(\text{NF})$. Hence, the length criterion for the NF feasibility is satisfied. For any cell in the *queue*, if the length of the NF is still greater than the total length, we will have to successively add the adjacent cells in the horizontal direction until the criterion is satisfied. If we do not manage to find such cells, then no feasible candidate points are generated for the node n_1. At the end of this step, we will have a rectangular region made of various cells, and it can completely contain the NF. The convex hull formed by the four feasible NF placement candidates within this rectangular region represents a $\mathcal{Q}$ set.

- Since we started to fill the NF from the node n_1, in the north-east direction, we obtain the feasible placement candidate point $\mathbf{p_1} \in C_4$, whose coordinates can be calculated using the coordinates of the node n_1 and the dimensions of the NF. We can use the above procedure for all the network nodes in each of the four fill directions (SW, SE, NE, NW) and we can systematically obtain the feasible placement candidates for the NF.

The complexity of this algorithm is $O(hvN^2)$, where h and v are the maximum number of horizontal and vertical gridlines intersected by the NF at a time.

.

2.7 Numerical Results

We conducted extensive computational tests on our procedure, coded in C++. The program was executed on Intel® Core™ i7, 3.50GHz, quad-core processor with 8GB memory. The input problems are divided into 9 categories based on the number of EFs and NFs, both of which are chosen from $\{2, 3, 4\}$. Each facility has a fixed area of 10000 sq. units. The aspect ratio of each facility is generated randomly from $\mathrm{U}[0.5, 2]$ distribution. The main layout in each problem instance has a fixed congestion factor of 0.5, which is the ratio of the area occupied by the facilities (EFs and NFs) to the total area of the layout. The main layout has square shape and its dimensions are calculated based on the congestion factor and total number of EFs and NFs. The EFs are placed at random, non-overlapping locations in the layout. Each EF has a single I/O point, randomly located on its boundary. Each NF also has a single I/O point, but it is assumed to be located at its top left corner. The EF–EF, EF–NF, and NF–NF interaction matrices are randomly generated from the distribution of $\mathrm{U}[0, 1]$.

As a starting point, we considered the 4EF–3NF problem category, and we generated 10 random layouts according to the scheme discussed above. On these 10 layouts, we executed our optimal procedure (which considers all placement permutations) and compared it with a heuristic procedure which considers only a single placement permutation. The placement permutation for this heuristic was generated using a simple scheme, in which the first facility to be placed is the one which has the highest total interaction with all the EFs; the second facility to be placed is the one which has the highest total interaction with the EFs and the first NF; and so on. For each of the 10 layouts, we noted the number of candidates evaluated, the objective function value, the optimality gap, and the execution time. These computational results are shown in Table 2.1. We can see that the optimal procedure evaluates significantly more candidates than the heuristic, and it has correspondingly higher execution time. The upper bound provided by the heuristic is satisfactory in all the problem instances, as compared to the optimal objective function value (largest optimality gap is 0.94%, and 0% gap for 7 problem instances), which suggests that this procedure can be used as a valid alternative to the optimal procedure. The layouts representing optimal and heuristic solutions for all the 10 problems are presented in Fig. 2.9. These layouts are rendered using a viewer developed by us in Java programming language.

Table 2.1: Results for 4EF–3NF problem

Pr. No.	Optimal Procedure				Heuristic Procedure				
	Opt. Perm.	# of Cnd.	Obj. Val.	Time (s)	Opt. Perm.	# of Cnd.	Obj. Val.	% Gap	Time (s)
1	[2, 1, 3]	78662	1951.05	12.19	[1, 2, 3]	11820	1969.45	0.94	1.80
2	[1, 3, 2]	146057	2319.16	22.11	[3, 1, 2]	25904	2319.16	0.00	3.97
3	[2, 1, 3]	54399	2698.32	8.31	[2, 1, 3]	8415	2698.32	0.00	1.23
4	[1, 2, 3]	48528	2195.76	7.65	[1, 3, 2]	9504	2198.19	0.11	1.54
5	[1, 3, 2]	22464	1905.64	3.78	[3, 1, 2]	3519	1905.64	0.00	0.59
6	[1, 2, 3]	3449755	2210.47	527.75	[1, 2, 3]	591667	2210.47	0.00	91.03
7	[1, 2, 3]	164637	1914.41	26.48	[2, 1, 3]	26604	1914.41	0.00	4.27
8	[3, 2, 1]	269327	2162.97	41.70	[3, 1, 2]	54818	2165.13	0.10	8.62
9	[1, 2, 3]	17600	2449.02	2.67	[3, 1, 2]	2912	2449.02	0.00	0.46
10	[2, 1, 3]	7264	2284.71	1.59	[3, 2, 1]	1136	2284.71	0.00	0.25

Finally, we tested the heuristic on all the remaining problem sets and recorded the number of candidates evaluated and the objective function values. The results are shown in Table 2.2. We can see that the number of candidates evaluated (and consequently the execution time) is a function of both the number of EFs and the number of NFs in the layout. The average number of candidates increases sharply with increasing number of NFs, but it does not increase that sharply with the number of EFs. We also see large variability in the number of candidates evaluated, across different instances of the same category. This can be seen in the large difference between the mean and the median values. However, the objective function values are distributed quite evenly. Since we have implemented a heuristic procedure, we need to have a lower bound to provide a guarantee on the feasible solution. For this purpose we can devise a lower bounding technique, in which we remove the size restrictions on the NFs and solve the resulting QSAP. We believe that this technique will provide strong lower bounds on the objective function values. Implementation of this lower bounding procedure is beyond the scope of this work.

2.8 Conclusion

To summarize, we examined the problem of placing M finite-size rectangular facilities with known dimensions, in presence of existing rectangular facilities. There are a large number of applications in manufacturing/warehouse facility (re)design and electronic component placement in VLSI, where

Table 2.2: Results for random layouts

Problemset	# of EFs	# of NFs	# of Infeasible Problems	# of Cnd.				Obj. Val.			
				Avg	Min	Med	Max	Avg	Min	Med	Max
1	2	2	2	263.3	52.0	292.0	426.0	429.60	192.13	481.80	594.56
2	2	3	0	7895.9	253.0	6284.5	26286.0	834.50	427.68	777.37	1274.40
3	2	4	1	491125.4	30521.0	265514.0	1782052.0	1254.47	809.94	1340.54	1624.68
4	3	2	3	742.9	44.0	524.0	2307.0	1049.56	685.41	1044.35	1603.70
5	3	3	0	19620.5	3050.0	16943.0	33433.0	1409.63	1071.60	1376.63	2066.85
6	3	4	0	1911252.0	43734.0	993045.5	5920310.0	2025.66	1502.22	1981.09	2577.02
7	4	2	2	872.1	204.0	744.0	2148.0	1579.55	1097.06	1434.81	2400.22
8	4	3	0	73629.9	1136.0	10662.0	591667.0	2211.45	1905.64	2204.33	2698.32
9	4	4	0	4390462.8	25912.0	4442422.5	13852926.0	3135.94	2281.82	3233.27	3857.63

this problem is highly relevant. We considered three types of interactions in our problem: (1) the interaction between the EFs and the NFs; (2) the interaction between the pairs of EFs and (3) the interaction between the pairs of NFs. The facilities interact with each other through I/O points located on their boundary. The travel occurs according to rectilinear metric and all the facilities act as barriers to travel. The objective is to minimize the sum of weighted rectilinear distances between the interacting facilities.

We proposed a solution procedure, in which we first divide the feasible region into sub-regions and then analyze the behavior of the objective function over these sub-regions, based on the interference of NF vertices. We proved that the candidates for the optimal placement of the NFs are the corner points of these sub-regions. We evaluated the solution complexity of our procedure and showed that it is exponential in the number of NFs. This fact is corroborated by the computational experiments, and we see that the optimal procedure does become a bottleneck for large number of NFs. The computational experiments also show that a heuristic procedure performs quite well, and it may help reduce the execution time. However, the heuristic needs to be coupled with a strong lower bounding approach so as to provide a satisfactory performance guarantee.

It is important to note that the solution complexity of the problem remains exponential, even if we relax the constraints on the NF sizes, which validates the difficulty of the problem that we are solving. In fact, this relaxed problem is an instance of the Quadratic Semi-Assignment Problem, which is known to be NP-hard. Nevertheless, we believe that our work provides a systematic and elegant way of analyzing and solving this problem. Some of the future directions of research include simultaneous placement and I/O point location of the NFs, determining the optimal form factor and orientation of the NFs, etc., which are all extremely relevant to the advancement of the location/layout theory.

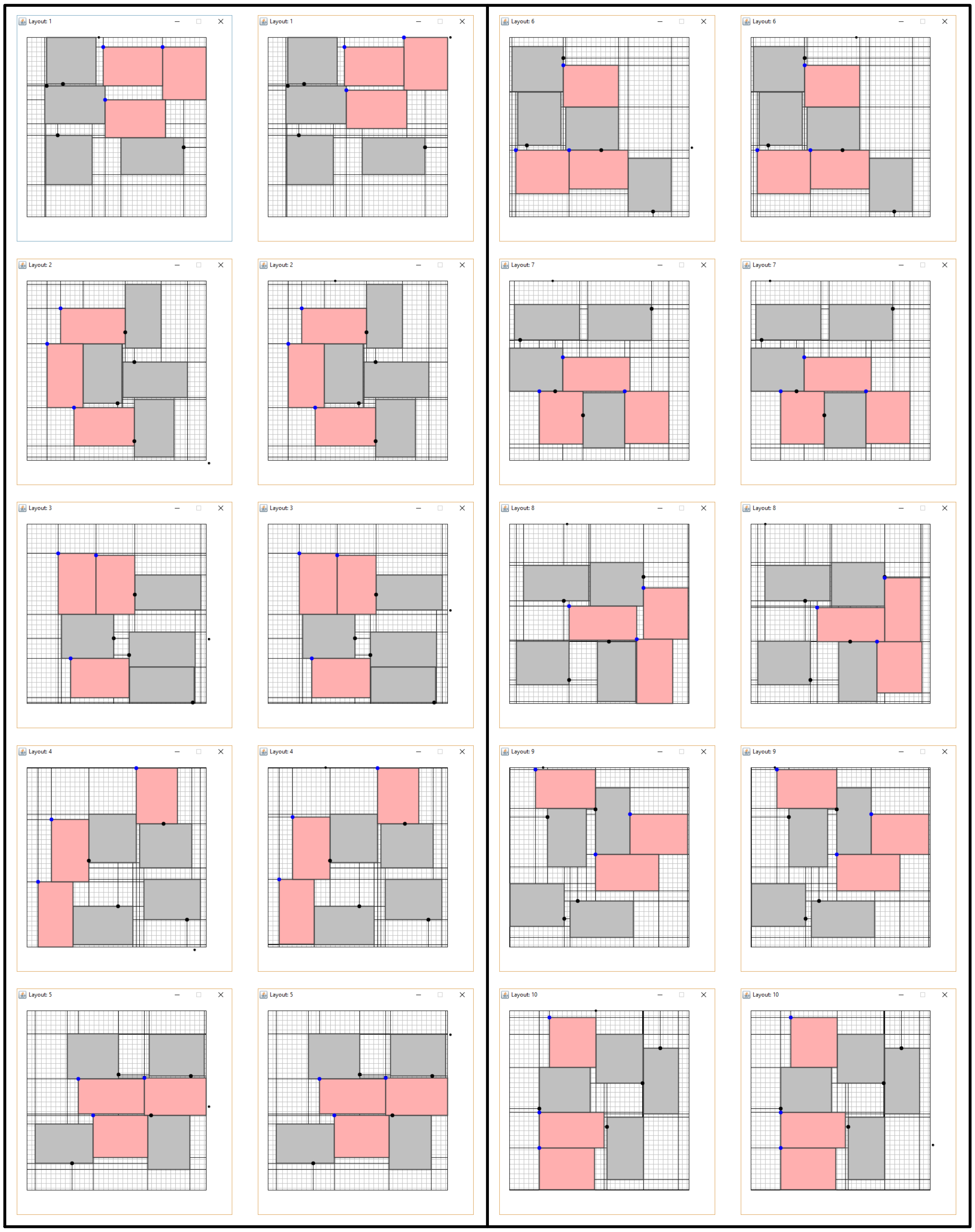


Figure 2.9: Optimal solution (left column) vs. heuristic solution (right column) for 4EF–3NF prolem

Chapter 3

Theory of Dominance for Finite-size Facility Placement Problem

In this chapter, we consider the problem of optimal placement of finite-size, rectangular facilities in presence of other rectangular facilities. It has been established in the previous work that the optimal placement of the new facilities belongs to a finite set of candidate points, and it can be found by evaluating the objective function value at each and every candidate point. This explicit enumeration guarantees the optimal solution, however it might become time consuming for a large number of new and existing facilities. We propose a new procedure based on the lower bounding technique, which can effectively cut down the number of candidate points that need to be evaluated, resulting in significant reduction in the computing time. The procedure was tested on a large number of randomly generated layouts with varying congestion factors (ratio of area occupied by the existing facilities to the total layout area). These extensive numerical tests reveal that, for a moderately congested layout, there is more than 70% reduction in both the number of evaluated candidates and the computing time, for 1 and 2 new facilities. The work regarding 1 facility dominance is reprinted from Date et al. (2014), with permission from Elsevier.

3.1 Introduction

Although this chapter can be considered as a continuation of Chapter 2, we will reintroduce the readers to the finite-size facility placement problem, just to be self-contained. The layout under consideration is a rectangular, closed region with finite area. There are N *existing facilities* (EFs), with rectangular shapes and edges parallel to the travel axes. M *new facilities* (NFs) having rectangular shapes and known dimensions are to be placed in the layout in presence of the EFs with their edges parallel to the travel axes. Each EF has one or multiple I/O point(s) while each NF has a single I/O point. The I/O points are strictly located on the boundary of each facility and flow between the facilities is serviced through them. We assume that the travel occurs according to the *Rectilinear* metric and the travel through a facility is not permitted (i.e., NFs and EFs act as barriers to travel). Three types of interactions are considered, whose values are assumed to be known:

- Pairwise interactions between the I/O points of existing facilities.
- Pairwise interactions between the I/O points of new and existing facilities.
- Pairwise interactions between the I/O points of new facilities.

The objective is to determine the optimal placement of the NFs (designated by the location of their top left corners) such that there is no overlap between the NFs and the EFs, and the total cost of travel (calculated as the weighted sum of rectilinear distances between the interacting facilities) is minimized. The feasible region for the placement of M finite-size NFs is given by Equation (2.1) and the corresponding objective function is given by Equation (2.2).

In the previous chapter, we developed a generalized theory and an explicit enumeration procedure for optimal placement of M finite-size facilities in presence of existing finite-size facilities. In order to find the optimal solution, the objective function needs to be evaluated for all the feasible candidate points. Since the number of feasible candidate points are exponential in the number of new facilities, this explicit enumeration might require considerable amount of computing effort. For this reason, a better solution procedure needs to be developed which can potentially eliminate some of the non-optimal solutions, before evaluating the objective function value. In this work we propose the theory of dominance for pruning the set of feasible candidate locations, resulting in faster convergence to the optimal solution. This procedure can be directly applied to such problems as finding the best location for a new machine in a manufacturing plant, finding the best location for a new building in a campus, etc.

The rest of the chapter is organized as follows. Since this is a continuation of the previous chapter, we will be reusing most of the notation established in Chapter 2. In Sections 3.2 and 3.3, we establish the dominance results and develop the procedure for finding the optimal placement of 1 and 2 NFs (i.e., for $M = 1$ and $M = 2$). In Section 3.4, we empirically validate the effectiveness of our procedure for a large set of randomly generated layouts. In Section 3.5, we discuss a generalization of the dominance procedure for the placement of M finite-size NFs. Finally, in Section 3.6, the chapter is concluded with a summary and future research directions.

3.2 Dominance Results for the Case of a Single NF

Let us first consider the problem of placing a single finite-size rectangular facility in presence of N finite-size rectangular facilities (i.e., $M = 1$). Since there is only a single NF in the layout, $L(\mathbf{p}) = 0$. The feasible region for this problem is given by Equation (2.4). Let X denote the location of the I/O point on the boundary of the NF. Then the objective function for this problem can be written as:

$$\Phi(\mathbf{p}) = J(\mathbf{p}) + K(\mathbf{p}) = \sum_{a \in A} u_a d_{\mathbf{p}}(a, X) + \sum_{a \in A} \sum_{b \in A} v_{ab} d_{\mathbf{p}}(a, b). \tag{3.1}$$

The facility placement problem is to determine the optimal placement $\mathbf{p}^*$ of the NF such that $\Phi(\mathbf{p}^*) \leq \Phi(\mathbf{p}), \forall \mathbf{p} \in F_{(1)}$. To find the optimal solution, we need to first construct all the $\mathcal{Q}$ sets for the NF. Then, we need to evaluate the objective function value at the corners of each $\mathcal{Q}$ set and find the one that yields the minimum value. The solution complexity of this explicit enumeration procedure is $O(\beta^2 N^2)$, where β is the maximum number of gridlines intersected by the NF in any direction.

To find and eliminate the dominated candidate points, we will utilize the technique of lower bounding. The main idea behind this, is to find a valid relaxation to the original problem and use its value as a *lower bound* on the original problem. As the name suggests, lower bound represents the best value that the original problem can possibly achieve, for any solution within a particular solution space. After establishing a lower bound on that solution space, we can solve the original problem to find an *incumbent solution*. If the value of the incumbent solution is same or better than the lower bound then we can eliminate or *fathom* all the solutions in that space because none of them will ever have a value better than the incumbent. This way we can eliminate a number of non-optimal solutions without affecting the global optimal solution which is present in a different solution space. This approach is central to our solution strategy. We will elaborate the above steps with the help of the following theorems.

Theorem 4. *An infinitesimal facility located at a feasible candidate point serves as a valid relaxation for a finite-size facility placed at that point.*

Proof. Consider a finite-size NF placed at any feasible candidate point. The overall objective function value $\Phi(\mathbf{p})$ for this particular placement is given by Equation (3.1). Since all the interactions are non-negative we know that: $J(\mathbf{p}) \geq 0$ and $K(\mathbf{p}) \geq 0$.

Let us relax the constraint on the size of the NF, such that it will coincide with its I/O point X. Now, X can be treated as an infinitesimal facility located at the feasible candidate point. Let $\Phi(\bar{\mathbf{p}})$ denote the overall objective function value for the relaxed problem. From Equation (3.1), we can write:

$$\Phi(\bar{\mathbf{p}}) = J(\bar{\mathbf{p}}) + K(\bar{\mathbf{p}}). \tag{3.2}$$

Since the NF is infinitesimal in size, it does not cut off any of the existing gridlines; and hence, $K(\bar{\mathbf{p}})$ is a constant. It is obvious that: $J(\bar{\mathbf{p}}) \leq J(\mathbf{p})$ and $K(\bar{\mathbf{p}}) \leq K(\mathbf{p})$. Therefore, we can write:

$$\Phi(\bar{\mathbf{p}}) \leq \Phi(\mathbf{p}). \tag{3.3}$$

Also, if the placement of a finite-size NF is feasible at a particular point, then the placement of an infinitesimal NF is also feasible at that point; i.e., feasibility of the constrained problem implies feasibility of the relaxed problem.

Thus, an infinitesimal NF satisfies all the necessary conditions of a valid relaxation. The proof is complete.

□

Theorem 5. *The minimum of the objective function values for an infinitesimal facility located at the corners of a cell provides a valid lower bound on the objective function value for a finite-size facility placed at any feasible candidate point inside the cell.*

Proof. Let $\Phi(\bar{\mathbf{p}})$ and $\Phi(\mathbf{p})$ respectively denote the objective function values for an infinitesimal NF and a finite-size NF placed at a feasible point inside a cell C. From Theorem 4, we know that $\bar{\mathbf{p}}$ is a valid relaxation to $\mathbf{p}$ and $\Phi(\bar{\mathbf{p}}) \leq \Phi(\mathbf{p})$.

Let $\Phi(\bar{\mathbf{p}}_n), n \in \{1,2,3,4\}$, denote the objective function value for an infinitesimal NF located at the respective corners of the cell C; and let $\Phi(\hat{\mathbf{p}}) = \min\{\Phi(\bar{\mathbf{p}}_n)\}$. From Result 3, for an infinitesimal NF, the EF–NF interaction $J(\bar{\mathbf{p}})$ is concave over the cell C, while the EF–EF interaction $K(\bar{\mathbf{p}})$ remains constant. Therefore, it is obvious that:

$$\Phi(\hat{\mathbf{p}}) \leq \Phi(\bar{\mathbf{p}}). \tag{3.4}$$

Combining Equations (3.3) and (3.4), we get:

$$\Phi(\hat{\mathbf{p}}) \leq \Phi(\bar{\mathbf{p}}) \leq \Phi(\mathbf{p}). \tag{3.5}$$

Thus $\Phi(\hat{\mathbf{p}})$ represents the lowest value that the objective function can ever have over the cell C. The proof is complete.

□

Let $\xi(C)$ denote the lower bound for a cell C. Then from Equation (3.5), the expression for the lower bound can be written as:

$$\xi(C) = \Phi(\hat{\mathbf{p}}) = \min\{\Phi(\bar{\mathbf{p}}_n)\}; \tag{3.6}$$

where, $\mathbf{p}_n = [X, E_4(\bar{B})] : E_4(\bar{B}) = E_n(C), n \in \{1,2,3,4\}$.

The importance of Theorem 5 can be explained as follows. We can find the objective function values for an infinitesimal facility at the corners of a cell and use their minimum as a lower bound on the entire cell. Since any $\mathcal{Q}$ set belongs to a particular cell, we can guarantee that the objective function value for a finite-size facility placed at a corner of such a $\mathcal{Q}$ set will always be worse than or equal to the lower bound on that cell. Therefore, if we obtain an incumbent solution whose objective function value is better than the lower bound on a particular cell, then we can fathom all the $\mathcal{Q}$ sets contained within that cell and thus reduce the number of feasible candidate points that need to be evaluated.

Based on the above results, we can now write Algorithm 4 for finding the optimal placement of a finite-size facility using the dominance rules. We know that the feasible candidate points are finite in number. Therefore, the procedure is guaranteed to converge to the optimal solution (or prove infeasibility) in a finite number of steps. Empirical results show that the dominance procedure allows quicker convergence but its worst case complexity remains the same as that of explicit enumeration. It means that if the lower bounds on the cells are very similar, the dominance procedure might have to evaluate all the feasible candidate points before getting the optimal solution.

Algorithm 4: Dominance procedure for $M = 1$.

```
1. For each cell $C_i$, place an infinitesimal NF at each corner $E_n(C_i), n \in \{1,2,3,4\}$ and
   calculate $\Phi(\bar{\mathbf{p}}_n) = J(\bar{\mathbf{p}}_n) + K(\bar{\mathbf{p}}_n)$. Calculate lower bound $\xi(C_i) = \min\{\Phi(\bar{\mathbf{p}}_n)\}$.
2. Identify all the feasible placement candidates $\mathbf{p}$ for the NF. Add them to a candidate
   list $\Omega$ (see Section 2.6.4). If $\Omega = \emptyset$, stop. The problem is infeasible.
3. For all $\mathbf{p} \in C_i$, add tuples $\langle \mathbf{p}, \xi(C_i) \rangle$ to a list $\Theta$ and sort in ascending order of $\xi(C_i)$.
4. Initialize incumbent solution $\mathbf{p}' \leftarrow \emptyset$ and $\Phi(\mathbf{p}') \leftarrow \infty$. Initialize iteration count $k \leftarrow 0$.
5. For each $\langle \mathbf{p}^k, \xi^k \rangle \in \Theta$, repeat:
   (a) Place finite-size NF at $\mathbf{p}^k$ and evaluate $\Phi(\mathbf{p}^k) = J(\mathbf{p}^k) + K(\mathbf{p}^k)$.
   (b) If $\Phi(\mathbf{p}^k) \le \Phi(\mathbf{p}')$, update incumbent solution $\mathbf{p}' \leftarrow \mathbf{p}^k$ and $\Phi(\mathbf{p}') \leftarrow \Phi(\mathbf{p}^k)$.
   (c) If $\Phi(\mathbf{p}') \le \xi^{k+1}$, terminate with the global optimal solution $\mathbf{p}^* = \mathbf{p}'$ and optimal
       objective function value $\Phi(\mathbf{p}^*) = \Phi(\mathbf{p}')$. Else, update $k \leftarrow k + 1$ and continue.
```

3.3 Dominance Results for the Case of Two NFs

Now, let us consider the problem of placing two finite-size rectangular facilities (i.e., $M = 2$) in presence of N finite-size rectangular facilities. The feasible region for this problem is given by Equation (2.1). Let X_1 and X_2 denote the location of the I/O points on the boundary of the NFs. Then the objective function for this problem can be written as:

$$\Phi(\mathbf{p}) = J(\mathbf{p}) + K(\mathbf{p}) + L(\mathbf{p}) = \sum_{a\in A}\sum_{i=1}^{2} u_{ai} d_{\mathbf{p}}(a, X_i) + \sum_{a\in A}\sum_{b\in A} v_{ab} d_{\mathbf{p}}(a, b) + w_{12} d_{\mathbf{p}}(X_1, X_2). \quad (3.7)$$

The facility placement problem is to determine the optimal placement $\mathbf{p}^*$ of the NFs such that $\Phi(\mathbf{p}^*) \le \Phi(\mathbf{p}), \forall \mathbf{p} \in F_{(2)}$. To find the optimal solution, we can follow the steps in Algorithm 2, in which we need to identify pairs of feasible candidate points and evaluate the objective function value at all these pairs. The solution complexity of this explicit enumeration procedure is $O(\beta^4 N^4)$, and therefore, the explicit enumeration procedure may become time consuming for large N. Therefore, finding and eliminating dominated candidate points is much more desirable. To find and eliminate the dominated candidate points, we will utilize the similar concepts as in Section 3.2. We will elaborate the theory with the help of the following theorems.

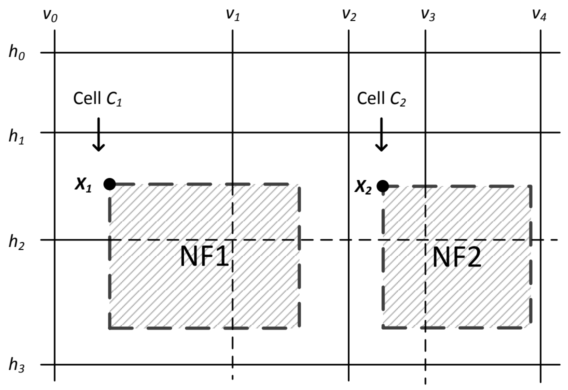


Figure 3.1: Pair of NFs placed within cells

Let us consider the situation shown in Fig. 3.1 in which NF$_1$ and NF$_2$ are placed within cells C_1 and C_2 respectively. Then we can write:

Theorem 6. *A pair of infinitesimal facilities located at a pair of feasible candidate points serves as a valid relaxation for the pair of finite-size facilities placed at those points.*

Proof. Let us relax the constraints on the size of the NFs, such that they will coincide with their respective I/O points X_1 and X_2. Now, X_1 and X_2 can be treated as infinitesimal facilities located at the feasible candidate points. Let $\Phi(\bar{\mathbf{p}})$ denote the overall objective function value for the relaxed problem. From Equation (3.7), we can write:

$$\Phi(\bar{\mathbf{p}}) = J(\bar{\mathbf{p}}) + K(\bar{\mathbf{p}}) + L(\bar{\mathbf{p}}). \tag{3.8}$$

Since the NFs are infinitesimal in size, they do not cut off any of the existing gridlines; and hence, $K(\bar{\mathbf{p}})$ is a constant. It is obvious that: $J(\bar{\mathbf{p}}) \leq J(\mathbf{p})$ and $L(\bar{\mathbf{p}}) \leq L(\mathbf{p})$. Therefore, we can write:

$$\Phi(\bar{\mathbf{p}}) \leq \Phi(\mathbf{p}). \tag{3.9}$$

Also, if the placement of a finite-size NF is feasible at a particular point, then the placement of an infinitesimal NF is also feasible at that point; i.e., feasibility of the constrained problem implies feasibility of the relaxed problem. Thus, infinitesimal NFs satisfy all the necessary conditions of a valid relaxation. The proof is complete.

□

Now let us consider the situation in Fig. 3.2, in which the constraints on the NF size have been relaxed. Let $\Phi(\bar{\mathbf{p}}^{mn}), \forall (m,n) \in \{1,2,3,4\} \times \{1,2,3,4\}$, denote the objective function value for

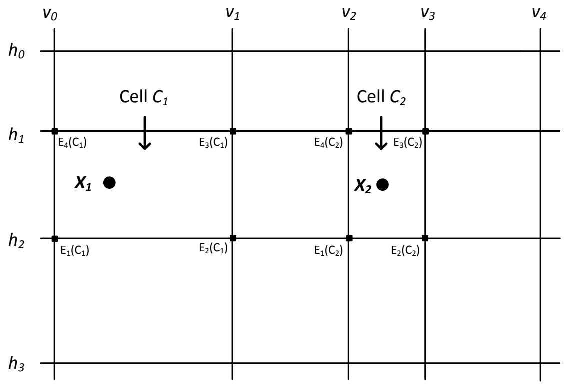


Figure 3.2: Pair of NFs with infinitesimal size

two infinitesimal NFs located at the various combinations of respective cell corners. Let $\Phi(\hat{\mathbf{p}}) = \min\{\Phi(\bar{\mathbf{p}}^{mn})\}$.

Theorem 7. *The minimum of the objective function values for a pair of infinitesimal facilities located at the corners of a pair of cells provides a valid lower bound on the objective function value for a pair of finite-size facilities placed at any feasible candidate point pair inside the respective cells.*

Proof. Let $\Phi(\bar{\mathbf{p}})$ and $\Phi(\mathbf{p})$ respectively denote the objective function values for a pair of infinitesimal NFs and a pair of finite-size NFs placed at a pair of feasible candidate points inside cells C_1 and C_2, respectively. From Theorem 6, we know that $\bar{\mathbf{p}}$ is a valid relaxation to $\mathbf{p}$ and $\Phi(\bar{\mathbf{p}}) \leq \Phi(\mathbf{p})$.

Theorem 1 states that, we can find same or better objective function value by moving the NFs to the corner points of cells C_1 and C_2 respectively. Therefore, we can write:

$$\Phi(\hat{\mathbf{p}}) \leq \Phi(\bar{\mathbf{p}}). \tag{3.10}$$

Combining Equations (3.9) and (3.10), we get:

$$\Phi(\hat{\mathbf{p}}) \leq \Phi(\bar{\mathbf{p}}) \leq \Phi(\mathbf{p}). \tag{3.11}$$

Thus $\Phi(\hat{\mathbf{p}})$ represents the lowest value that the objective function can ever have over the pair of cell (C_1, C_2). The proof is complete.

□

Let $\xi(C_1, C_2)$ denote the lower bound for a pair of cells (C_1, C_2). Then the expression for this lower bound can be written as:

$$\xi(C_1, C_2) = \Phi(\hat{\mathbf{p}}) = \min\{\Phi(\bar{\mathbf{p}}^{mn})\}. \tag{3.12}$$

Based on the above results, we can now write Algorithm 5 for finding the optimal placement of two finite-size facilities using the dominance rules. We know that the feasible candidate points are finite in number. Therefore, the procedure is guaranteed to converge to the optimal solution (or prove infeasibility) in a finite number of steps.

Algorithm 5: Dominance procedure for $M = 2$.

```
1. For each cell pair $(C_i, C_j)$, place infinitesimal NF$_1$ at each corner of $C_i$ and infinitesimal
   NF$_2$ at each corner of $C_j$.
   Calculate $\Phi(\bar{\mathbf{p}}^{mn}) = J(\bar{\mathbf{p}}^{mn}) + K(\bar{\mathbf{p}}^{mn}) + L(\bar{\mathbf{p}}^{mn}), \forall m, n \in \{1, 2, 3, 4\}$.
   Calculate lower bound $\xi(C_i, C_j) = \min\{\Phi(\bar{\mathbf{p}}^{mn})\}$.
2. Identify all the feasible placement candidates $\mathbf{p}$ for the two finite-size NFs. Add them to
   candidate list $\Omega$. If $\Omega = \emptyset$, stop. The problem is infeasible.
3. For all $\mathbf{p} \in (C_i, C_j)$ add tuples $\langle \mathbf{p}, \xi(C_i, C_j) \rangle$ to a list $\Theta$ and sort in ascending order of $\xi$.
4. Initialize incumbent solution $\mathbf{p}' \leftarrow \emptyset$ and $\Phi(\mathbf{p}') \leftarrow \infty$. Initialize iteration count $k \leftarrow 0$.
5. For each $\langle \mathbf{p}^k, \xi^k \rangle \in \Theta$, repeat:
   (a) Place two finite-size NFs at $\mathbf{p}^k$ and evaluate $\Phi(\mathbf{p}^k) = J(\mathbf{p}^k) + K(\mathbf{p}^k) + L(\mathbf{p}^k)$.
   (b) If $\Phi(\mathbf{p}^k) \leq \Phi(\mathbf{p}')$, update incumbent solution $\mathbf{p}' \leftarrow \mathbf{p}^k$ and $\Phi(\mathbf{p}') \leftarrow \Phi(\mathbf{p}^k)$.
   (c) If $\Phi(\mathbf{p}') \leq \xi^{k+1}$, terminate with the global optimal solution $\mathbf{p}^* = \mathbf{p}'$ and objective
       function value $\Phi(\mathbf{p}^*) = \Phi(\mathbf{p}')$. Else, update $k \leftarrow k + 1$ and continue.
```

3.4 Computational Results

We conducted extensive computational comparison of the dominance procedure (DR) with the explicit enumeration procedure (EE). Both the procedures were coded in Java. The single facility dominance procedure was executed on Intel® Core™ i7, 2.20GHz, quad-core processor; while two facility dominance procedure was executed on Intel® Core™ i3, 2.30GHz, dual-core processor.

3.4.1 Computational Results for Single NF Dominance Procedure

The area of the layout in all the problems is 400×400 sq. units. Each layout has four randomly placed EFs, having the same area but different dimensions. Each EF has a single I/O point, randomly located on its boundary. The aspect ratio of each EF is selected using the function $2^{\mathrm{U}[-1,1]}$, to get aspect ratios within the range of $[0.5, 2]$. The size of the NF is 100×100 sq. units, with the I/O point X located at its top left corner $E_4(\bar{B})$. The EF–EF and EF–NF interactions are randomly generated from the Uniform distribution of $[0, 1]$. We divided the problems into five categories based on the congestion factor of the layout, which is selected from a range of 0.1 to

0.5, with the increments of 0.1. For each congestion factor, we tested both the procedures on 100 randomly generated layouts; and for each problem, we noted the number of candidate points evaluated and the computing time.

Table 3.1: Computational results for single facility dominance procedure

Congestion Factor	No. of problems	Infeasible problems	Avg. # of candidates	Avg. # of evaluated candidates			Avg. computing time (ms)			
				EE	DR	% improvement	EE	DR	% improvement	Avg. $\Phi(\mathbf{p}^*)$
0.1	100	0	85.87	85.87	14.71	82.87	1497.35	235.78	84.25	1012.17
0.2	100	0	67.59	67.59	16.60	75.44	1058.69	250.01	76.38	1169.41
0.3	100	4	47.98	47.98	14.82	69.11	690.85	207.20	70.01	1207.03
0.4	100	9	28.79	28.79	11.77	59.12	372.77	148.69	60.11	1345.32
0.5	100	31	21.96	21.96	10.87	50.50	248.14	123.43	50.26	1375.42

The average computational results for each of the five categories are presented in Table 3.1. The percentage improvement in the number of evaluated candidates is calculated using the formula: $\frac{(\#\text{ EE candidates})-(\#\text{ DR candidates})}{(\#\text{ EE candidates})} \times 100$. A similar formula is applied for calculating the percentage improvement in the computing time. We also plotted the number of evaluated candidates and the computation times in milliseconds against the layout congestion factors, as shown in Fig. 3.3.

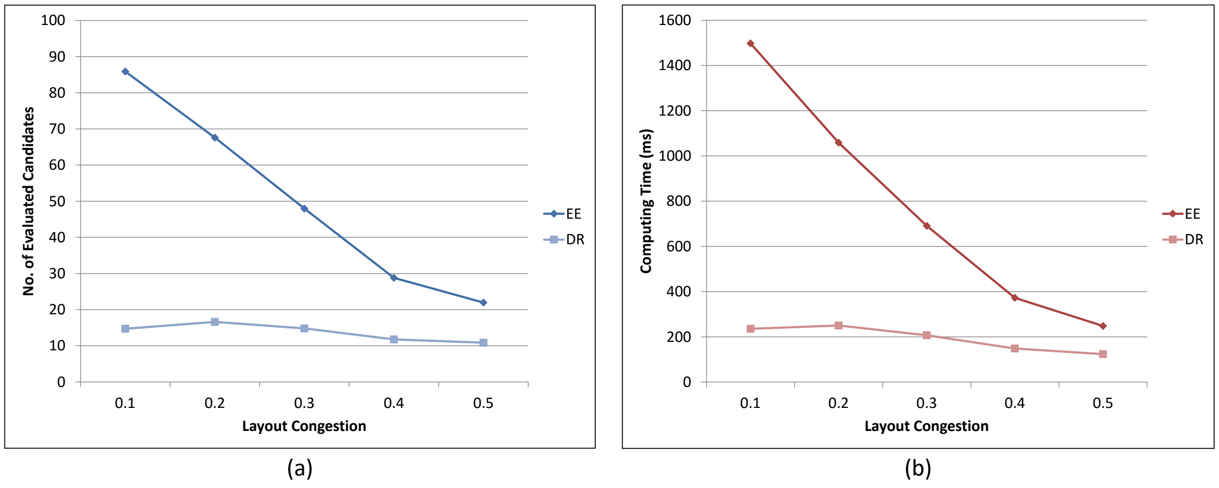


Figure 3.3: Computational results for $M = 1$: (a) Number of evaluated candidates; (b) Computation time (ms)

From Table 3.1 and Fig. 3.3, it is evident that the dominance procedure gives superior results as compared to the explicit enumeration. For an average congestion factor of 0.3, there is approximately 70% reduction in the number of candidates evaluated as well as the computing time. In the worst case, our algorithm will perform as good as the explicit enumeration, i.e., it will evaluate all the feasible candidate points. It is worthwhile to note that, as the layout becomes more and more congested, the percentage improvement goes on decreasing. The reason behind this phenomenon is that, in a congested layout there are fewer number of cells available for the NF placement, which may have very similar lower bounds, and because of this, not many candidate points can be fathomed. The increased congestion in the layout also results in an increased number of infeasible problems and an increased objective function value.

3.4.2 Computational Results for Two NF Dominance Procedure

For the computational experiments for the two NF dominance procedure, the congestion factor is kept constant at 30% and the number of EFs in the layout are increased from 2 to 8, in increments of 2 (total four problem categories). The area occupied by each EF is fixed to 10,000 sq. units and its aspect ratio is generated using the function $2^{\mathrm{U}[-1,1]}$. The I/O point of each EF is located randomly on its boundary. The dimensions of the two NFs are fixed to 70×70 sq. units and their I/O points are located at the top left corners. The EF–EF and EF–NF interactions are randomly generated from the Uniform distribution of $[0, 1]$. For each problem category, the procedure was tested on 50 randomly generated layouts; and for each problem, the number of candidate points evaluated and the computing time, were noted.

The average computational results for each of the five categories are presented in Table 3.2 and Fig. 3.4. From these results, it is evident that the performance of the dominance procedure is superior as compared to the explicit enumeration. For an average congestion factor of 0.3 and for more than 4 EFs in the layout, there is over 90% reduction in the number of candidates evaluated as well as the computing time.

Table 3.2: Computational results for two facility dominance procedure

	# of	Avg. # of evaluated candidates			Avg. computing time (s)		
# of EFs	problems	EE	DR	% improvement	EE	DR	% improvement
2	50	672.56	357.94	46.78	1.11	0.61	45.07
4	50	5730.24	636.70	88.89	101.27	10.34	89.79
6	50	18911.12	859.52	95.45	1269.20	53.20	95.81
8	50	57135.76	1096.86	98.08	82557.12[†]	1651.14	98.00[†]

[†]estimated

Alternate lower bound. We tested an alternate lower bounding procedure in which the distances between the I/O points are calculated as direct rectilinear distances, instead of the ones obtained as the sum of distances between a sequence of communicating nodes. To be more specific, instead of calculating $d_{\mathbf{p}}^{\text{exact}}(a,b)$ (using Equation (2.3)), in a shortest path algorithm, we calculate $d_{\mathbf{p}}^{\text{approx}}(a,b) = |x_a - x_b| + |y_a - y_b|$. Clearly, $d_{\mathbf{p}}^{\text{approx}}(a,b) \leq d_{\mathbf{p}}^{\text{exact}}(a,b)$, and the corresponding objective function provides a valid lower bound. However, calculating $d_{\mathbf{p}}^{\text{approx}}(a,b)$ is extremely quick, as opposed to $O(N^2)$ complexity of Dijkstra's shortest path algorithm. The comparison of lower bounds and execution times is presented in Table 3.3 and Fig. 3.5, which shows that the approximate distance function is extremely efficient and can be used to obtain fairly strong lower bounds.

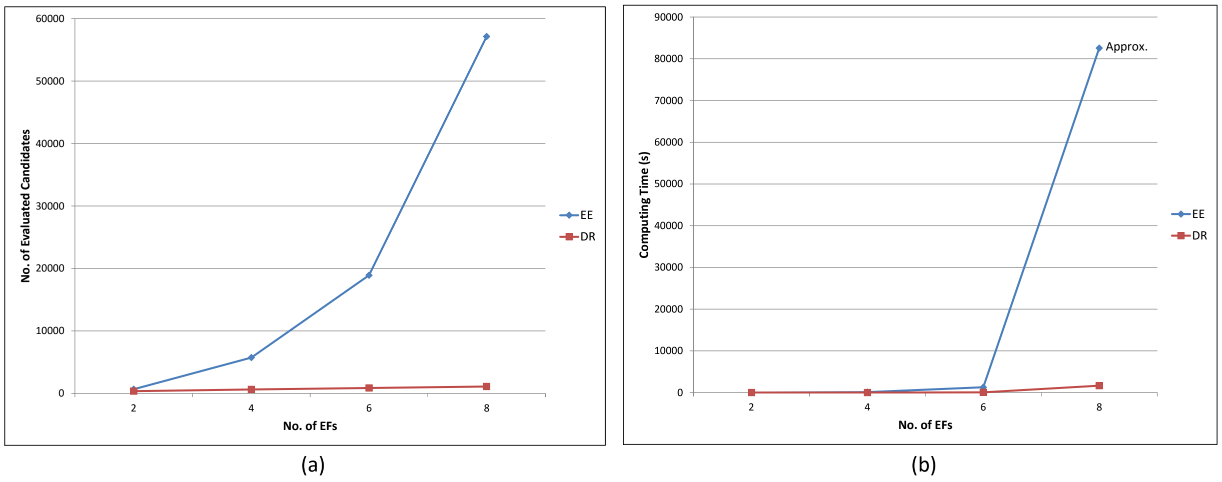


Figure 3.4: Computational results for $M = 2$: (a) Number of evaluated candidates; (b) Computation time (s)

Table 3.3: Comparison of Exact vs. Approximate distance

# of EFs	# of problems	Avg. # of evaluated candidates		Avg. bounding time (ms)	
		Exact	Approx.	Exact	Approx.
2	50	357.94	361.36	5.28	0.02
4	50	636.70	803.12	78.80	0.26
6	50	859.52	1040.26	321.60	0.32
8	50	1096.86	1581.04	4586.96	2.62

3.5 Generalization to M NFs

The results presented above prove the efficacy of the dominance results in pruning the solution space. The results can be generalized for the problem of placing M finite-size facilities, which will undeniably improve the performance of the explicit enumeration. However, the first and foremost challenge is the combinatorial nature of the lower bound calculation step. From Theorems 5 and 7, it can be speculated that to obtain the strongest lower bound, we may need to consider M-sized subsets of cells and their corners. This problem is similar to Case 2 discussed in Section 2.5 and it can be modeled and solved as an instance of Quadratic Semi-Assignment Problem (QSAP).

To be specific, we have M infinitesimal NFs and M cells, with $O(M)$ cell corners which serve as locations. Since the NFs are infinitesimal in size, we can place them arbitrarily close to each other; which means that at the limiting distance, the NFs can be assumed to overlap. In this setting the pairwise distances between all the $O(M)$ locations can be calculated *a priori*. Due to the infinitesimal size, the EF–EF interaction remains constant. For the EF–NF interaction, the cost of assigning NF$_i$ to a cell corner p, can be calculated as: $b_{ip} = \sum_{a \in A} u_{ai} d(a,p)$. Finally, the NF–NF interaction depends upon both the flows between the NFs and the distances between their

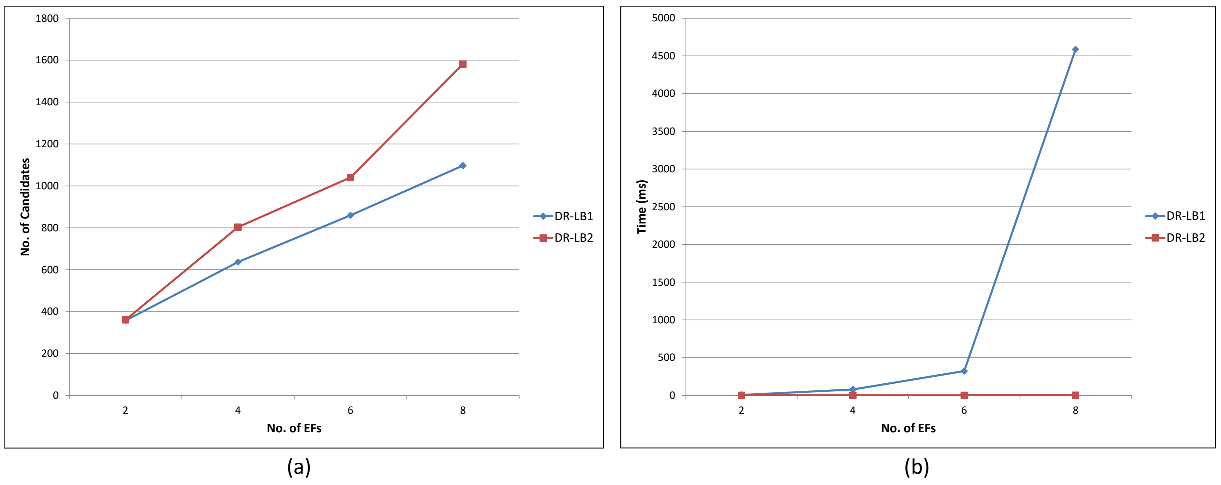


Figure 3.5: Comparison of Exact vs. Approx. distance: (a) # of evaluated candidates; (b) Computation time (ms)

locations. Specifically, the cost of assigning NF_i to a cell corner p and NF_j to a cell corner q is calculated as: $C_{ijpq} = w_{ij}d(p,q)$. Let the variable $x_{ip} = 1$, if NF_i is located at cell corner p, and 0 otherwise. Then, the problem of placing M infinitesimal facilities can be written as an instance of QSAP.

$$M\text{-NF QSAP: } \min \quad \sum_{i=1}^{M}\sum_{p=1}^{O(M)} b_{ip}x_{ip} + \sum_{i=1}^{M}\sum_{j=1}^{M}\sum_{p=1}^{O(M)}\sum_{q=1}^{O(M)} w_{ij}d_{pq}x_{ip}x_{jq}; \tag{3.13}$$

$$\text{s.t.} \sum_{p=1}^{O(M)} x_{ip} = 1 \quad \forall i = 1,\ldots,M; \tag{3.14}$$

$$x_{ip} \in \{0,1\} \quad \forall i = 1,\ldots,M;\ \forall p = 1,\ldots,O(M). \tag{3.15}$$

The optimal objective value of the above QSAP will provide a lower bound on all the feasible placements of the finite-size NFs within the subset of M cells. If this lower bound is greater than the current incumbent solution, then all the feasible placements in those M cells can be fathomed, potentially improving the execution time of the enumeration procedure.

The main advantage of solving this problem as QSAP is that we can use some implicit enumeration procedures like the *branch-and-bound*, which might require less computational effort if coupled with a strong lower bounding technique. In Section 5.7, we discuss an RLT2 linearization for this QSAP, through which we can leverage upon our accelerated dual ascent procedure for obtaining strong lower bounds for the placement of finite-sized NFs.

3.6 Conclusion

To summarize, we examined the problem of placing M finite-size, rectangular facilities with known dimensions, in presence of other finite-size, rectangular facilities. All the facilities interact with each other through the I/O points and the travel between them occurs according to the rectilinear metric. The objective is to find the optimal placement of the new facilities, with the help of some dominance rules, which will reduce the number of feasible placement candidates that need to be evaluated.

To develop the solution procedure, we first proved that an infinitesimal facility serves as a valid relaxation to the finite-size facility and it can be used to establish a lower bound on all the feasible placement candidates inside a particular cell. We showed that the non-optimal solutions can be fathomed by comparing the value of the incumbent solution with the lower bound on each cell. We implemented the procedure in Java and we conducted an extensive numerical analysis on randomly generated problems. We compared the average computing time and the average number of candidate points evaluated in our procedure with those from the explicit enumeration procedure and found that, depending on the problem type, there is over 70% improvement for $M = 1$ and over 90% improvement for $M = 2$.

Finally, we presented a method to generalize the theory of dominance for the placement of M facilities. This procedure would require us to compute lower bounds on all the subsets of 1 to M cells, which can be done by formulating the problem as a Quadratic Semi-Assignment Problem. Although it is an NP-hard problem, we can employ some implicit enumeration procedures to obtain the necessary lower bounds, which can potentially be used to fathom a large number of sub-optimal placement candidates. The future work includes testing this lower bounding procedure for more than 2 facilities, to validate its efficacy. Since the lower bounding procedure for M facilities is NP-hard, future research is aimed at proposing a new method that can provide strong theoretical lower bounds without sacrificing the polynomial-time complexity.

Chapter 4

GPU-accelerated Hungarian Algorithms for the Linear Assignment Problem

In this chapter, we describe parallel versions of two different variants (classical and alternating tree) of the Hungarian algorithm for solving the Linear Assignment Problem (LAP). We have chosen Compute Unified Device Architecture (CUDA) enabled NVIDIA Graphics Processing Units (GPU) as the parallel programming architecture because of its ability to perform intense computations on arrays and matrices. The main contribution of this work is an efficient parallelization of the augmenting path search phase of the Hungarian algorithm. Computational experiments on problems with up to 25 million variables reveal that the GPU-accelerated versions are extremely efficient in solving large problems, as compared to their CPU counterparts. Tremendous parallel speedups are achieved for problems with up to 400 million variables, which are solved within 13 seconds on average. We also tested multi-GPU versions of the two variants on up to 16 GPUs, which show decent scaling behavior for problems with up to 1.6 billion variables and dense cost matrix structure. This work is reprinted from Date and Nagi (2016), with permission from Elsevier.

4.1 Introduction

The objective of the linear assignment problem (LAP) is to assign n resources to n tasks such that the total cost of the assignment is minimized. The mathematical formulation for the LAP can be written as follows:

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

$$\text{s.t.} \sum_{j=1}^{n} x_{ij} = 1 \qquad \forall i = 1,\ldots,n; \tag{4.2}$$

$$\sum_{i=1}^{n} x_{ij} = 1 \qquad \forall j = 1, \ldots, n; \tag{4.3}$$

$$x_{ij} \in \{0, 1\} \qquad \forall i, j = 1, \ldots, n. \tag{4.4}$$

The decision variable $x_{ij} = 1$, if resource i is assigned to task j and 0 otherwise. Constraints (4.2) and (4.3) enforce that each resource should be assigned to exactly one task and each task should be assigned to exactly one resource. c_{ij} is the cost of assigning resource i to task j, and $\mathbf{C}_{n\times n} = [c_{ij}]$ is the cost matrix of the LAP.

LAP is one of the most well-studied optimization problems that can be solved in polynomial time. Until now, many efficient sequential algorithms have been proposed in the literature. These algorithms can be classified into three main classes (Jonker and Volgenant, 1987, Burkard and Çela, 1999): (1) *Linear programming based algorithms*, which involve variants of the primal and dual simplex algorithms; (2) *Primal-dual algorithms* such as the famous Hungarian algorithm (Kuhn, 1955) and the Auction algorithm (Bertsekas, 1990); and (3) *Dual algorithms* such as the successive shortest path algorithm (Jonker and Volgenant, 1987). Due to their polynomial worst-case complexity, the primal-dual and shortest path algorithms generally outperform the simplex-based algorithms. Several variations of the Hungarian and the shortest path algorithms have been proposed in the literature, for improving their execution time (Jonker and Volgenant, 1986, Volgenant, 1996, Jonker and Volgenant, 1999). The theoretical complexity of the most efficient implementation of the primal-dual or shortest path algorithms is $O(n^3)$, where n is the number of people or jobs.

Owing to their cubic worst-case complexity, sequential algorithms can prove to be a significant bottleneck for large instances of the LAP. This calls for the development of a parallel algorithm, which can take advantage of a specific architecture and divide the work among multiple processors, to alleviate the computational burden. Until now many parallel versions of the aforementioned sequential algorithms have been proposed which include parallel asynchronous version of the Hungarian algorithm (Bertsekas and Castañon, 1993); parallel version of the shortest path algorithm (Balas et al., 1991, Storøy and Sørevik, 1997); and parallel synchronous and asynchronous versions of the Auction algorithm (Wein and Zenios, 1990, Bertsekas and Castañon, 1991, Buš and Tvrdík, 2009, Naiem et al., 2010, Sathe et al., 2012). An empirical analysis of the sequential and parallel versions of the Auction and shortest path algorithms was performed by Kennington and Wang (1991). All the above parallel algorithms were designed for prevalent parallel computing architectures and they were shown to achieve significant speedups.

In recent years, there have been significant advancements in the graphics processing hardware. Since graphics processing tasks generally require high data parallelism, the GPUs are built as compute-intensive, massively parallel machines, which provide a cost-effective solution for high performance computing applications. Vasconcelos and Rosenhahn (2009) developed a parallel version of the synchronous Auction algorithm for a single GPU. The authors tested the algorithm on problem instances with up to 16 million variables, which gets automatic scalability through CUDA with increasing number of GPU cores. Roverso et al. (2010) developed a GPU implementation of the *deep greedy switching* (DGS) heuristic of Naiem and El-Beltagy (2009), for solving the LAP under real-time constraints. It was shown that the heuristic sacrifices optimality in favor of significant speedup, on problem instances with up to 100 million variables.

In this work, we are proposing parallel versions of two variants of the Hungarian algorithm, specifically designed for the CUDA enabled NVIDIA GPUs. We have chosen to parallelize the Hungarian algorithm, mainly because it operates on the cost matrix of the LAP and the GPUs are well suited for performing intense computations on arrays and matrices. Our main contribution is an efficient algorithm for the augmenting path search phase, which happens to be the most time intensive phase in the Hungarian algorithm. The prominent feature of our algorithm is that it takes advantage of the race condition to generate multiple vertex-disjoint augmenting paths, which can be used simultaneously to improve the current solution. We show that this parallelization leads to a dramatic reduction in the execution time, for both small and large sized problem instances. LAPs serve as sub-problems to many NP-hard optimization problems such as the Traveling Salesman Problem (TSP), the Quadratic Assignment Problem (QAP), and the Generalized Assignment Problem (GAP). Finding good solutions to these problems generally requires solving multiple LAPs in an iterative fashion. Therefore, having a fast, scalable, and cost effective LAP solver is extremely important. We believe that our GPU-accelerated algorithms stand true on all the three requirements.

The rest of the chapter is organized as follows. In Section 4.2, we briefly describe the two variants of the sequential Hungarian algorithm. Section 4.3 contains some preliminaries, which will be useful in the development of the parallel algorithms. In Sections 4.4 and 4.5, we describe the various stages of our parallel algorithms, and their implementation on single and multi-GPU architectures. Section 4.6 contains the experimental results for randomly generated problem instances. Finally, the chapter is concluded in Section 4.7 with a summary.

4.2 Sequential Hungarian Algorithm

The Hungarian method developed by Kuhn (1955) was the first systematic approach for finding the optimal solution to an LAP. Although, the algorithm is primarily based upon the works of Hungarian mathematicians König and Egerváry, the main idea behind the algorithm can be better explained with the help of linear programming duality (Nering and Tucker, 1993, Bazaraa et al., 2011). The dual of the assignment problem (4.1)–(4.4) can be written as follows:

$$\max \quad \sum_{i=1}^{n} u_i + \sum_{j=1}^{n} v_j; \tag{4.5}$$

$$\text{s.t.} \quad u_i + v_j \leq c_{ij} \qquad \forall i, j = 1, \ldots, n; \tag{4.6}$$

$$u_i, v_j \sim \text{unrestricted} \qquad \forall i, j = 1, \ldots, n; \tag{4.7}$$

where, u_i and v_j are the dual variables corresponding to each constraint of the primal problem.

Using the Karush-Kuhn-Tucker complementary slackness condition for the optimal solution, we can write:

$$(c_{ij} - u_i^* - v_j^*)x_{ij}^* = 0. \tag{4.8}$$

Thus, if we find values for the dual variables u_i and v_j such that slack variables $c_{ij} - u_i - v_j = 0$, then the corresponding x_{ij} can be set to 1 (i.e., resource i can be assigned to task j), as long as they are present in independent rows and columns (necessary condition for primal feasibility). If the zero-valued slack variables are not independent, then we need to update the corresponding dual variables and find a new solution, which satisfies this condition. Thus, we start from dual feasibility and iteratively achieve primal feasibility.

Based on the above result (and the theorems by König and Egerváry), the Hungarian algorithm operates in two stages. In the first stage ("augmenting path search"), the algorithm finds the maximum matching corresponding to the edges with $c_{ij} - u_i - v_j = 0$, by building a directed tree rooted at an unassigned row, potentially ending at an unassigned column, and alternating between assigned and unassigned edges. If the alternating tree manages to terminate at an unassigned column, then it can potentially be used to increase the total number of assignments by one. If the maximum matching found at the end of this stage equals the total number of rows (or columns), the algorithm stops with the optimal assignment. Otherwise the second stage ("dual update") is executed, in which the dual variables are modified to introduce at least one new edge with zero slack. The algorithm continues to iterate between these two stages until an optimal solution is found.

We will now describe the two variants of the Hungarian algorithm: (1) The "classical" Kuhn-Munkres variant developed by Munkres (1957); and (2) The "alternating tree" variant developed by Lawler (1976).

4.2.1 Data Structures

The following data structures are used in this implementation.

1. *Cost matrix* ($\mathbf{C}$): This matrix is stored as an array of n^2 integers (or doubles), in row-major order.

2. *Row/column assignment arrays* (A_r/A_c): Each of these arrays is stored as an array of n integers. They are used for recording the row and column assignments, with -1 as the sentinel value. $A_r[i] = j$ indicates that row i is assigned to column j; and $A_c[j] = i$ indicates that column j is assigned to row i.

3. *Row/column cover arrays* (V_r/V_c): Each of these arrays is stored as an array of n booleans. They are used for recording the row and column covers. $V_r[i] = 1$ indicates that row i is covered, and 0 indicates otherwise. Column cover array V_c follows a similar convention.

4. *Row/column dual variable arrays* (D_r/D_c): Each of these arrays is stored as an array of n doubles. They are used for recording the dual variable values corresponding to the rows and

columns.

5. *Column slack variable array* (*slack*): This array is stored as an array of n doubles. It is used to store the minimum slack for each column, and it is only used in the alternating tree variant.

6. *Row/column predecessor arrays* (P_r/P_c): Each of these arrays is stored as an array of n integers. They are used for recording the predecessor indices of the rows and columns, with -1 as the sentinel value. They are primarily used during the augmenting path search phase of the algorithm (see Section 4.4.3).

7. *Row/column successor arrays* (S_r/S_c): Each of these arrays is stored in the device memory as an array of n integers. They are used for recording the successor indices of the rows and columns, with -1 as the sentinel value. They are also used during the augmenting path search phase of the algorithm.

4.2.2 Classical Hungarian Algorithm

The classical variant of the Hungarian algorithm was proposed by Munkres (1957) which systematizes the Hungarian method of Kuhn (1955). The pseudocode for this variant is presented below. In each iteration, the algorithm either increases the number of assignments by one or introduces new edges with slack $c_{ij} - u_i - v_j = 0$ (each of these steps has complexity of $O(n^2)$). An adjacency list is maintained during each iteration to store the edges with zero slack, which is modified/recreated after the dual update step. Since there are n^2 elements in the cost matrix, the "search" and "update" steps could be executed at most n^2 times, and therefore the classical variant has complexity of $O(n^4)$.

```
algorithm classical_hungarian
input: Matrix $\mathbf{C}$
output: Optimal assignments $A_r$ and $A_c$
begin
    /* Initial reduction */
    foreach $i \in \{1, \cdots, n\}$ do $D_r[i] \leftarrow \min_j\{\mathbf{C}[i,j]\}$;                /* row reduction */
    foreach $j \in \{1, \cdots, n\}$ do $D_c[j] \leftarrow \min_i\{\mathbf{C}[i,j] - D_r[i]\}$;     /* column reduction */
    repeat
        /* Optimality check */
        match_count ← 0; [*** check ***]
        reset $V_r, V_c, P_r, P_c$ to "sentinels";
        foreach $i \in \{1, \cdots, n\}$ do
            if $A_r[i] \neq -1$ then
                $V_r[i] \leftarrow 1$;
                match_count ← match_count + 1;
            end
        end
        if match_count $= n$ then go to exit;
        /* Augmenting path search */
        $ST \leftarrow \emptyset$;                                                          /* stack */
        foreach $i \in \{1, \cdots, n\}$ do [*** search ***]
            if $V_r[i] = 0$ then $ST.push(i)$;
            $Z[i] \leftarrow \emptyset$;                               /* initialize adjacency list for row $i$ */
            foreach $j \in \{1, \cdots, n\}$ do
                if $\mathbf{C}[i, j] - D_r[i] - D_c[j] = 0$ then $Z[i].push(j)$;
            end
        end
        while $ST \neq \emptyset$ do
            $i \leftarrow ST.top()$;
            $ST.pop()$;
            while $Z[i] \neq \emptyset$ do
                $j \leftarrow Z[i].front()$;
                $Z[i].pop()$;
                $i_{new} \leftarrow A_c[j]$;
                if $i_{new} = i$ then continue;                          /* continue on to next $j$ */
                if $V_c[j] = 0$ then                                     /* if column is uncovered */
                    $P_c[j] \leftarrow i$;                                /* update predecessor index */
                    if $i_{new} = -1$ then                                     /* unassigned column */
                        augment($j$);
                        go to check;
                    else
                        $ST.push(i_{new})$;
                        $P_r[i_{new}] \leftarrow j$;                      /* update predecessor index */
                        $V_r[i_{new}] \leftarrow 0$;                                  /* uncover the row */
                        $V_c[j] \leftarrow 1$;                                       /* cover the column */
                    end
                end
            end
        end
        update();
        go to search;
    end [*** exit ***]
end
/* Procedure for augmenting the current assignments by 1 */
procedure augment
input: Unassigned column $j$, Row predecessors $P_r$, Column predecessors $P_c$
output: Updated assignment arrays $A_r$ and $A_c$
begin
    $c_{cur} \leftarrow j$;
    $r_{cur} \leftarrow -1$;
    while $c_{cur} \neq -1$                    /* repeat until current row has no predecessor */
        $r_{cur} \leftarrow P_c[c_{cur}]$;
        $A_r[r_{cur}] \leftarrow c_{cur}$;
        $A_c[c_{cur}] \leftarrow r_{cur}$;
        $c_{cur} \leftarrow P_r[r_{cur}]$;                           /* update current column index */
    end
```

```
end
/* Procedure for updating the dual variables */
procedure update
input: Cover arrays $V_r$ and $V_c$
output: Updated dual variable arrays $D_r$ and $D_c$
begin
    $\theta \leftarrow \infty$;
    foreach $i \in \{1, \cdots, n\}$ do
        if $V_r[i] = 0$ then $\theta \leftarrow \min\{\theta, \min_{j|V_c[j]=0}\{\mathbf{C}[i,j] - D_r[i] - D_c[j]\}\}$;
    end
    foreach $k \in \{1, \cdots, n\}$ do
        if $V_r[k] = 0$ then $D_r[k] \leftarrow D_r[k] + \frac{\theta}{2}$; else $D_r[k] \leftarrow D_r[k] - \frac{\theta}{2}$;
        if $V_c[k] = 0$ then $D_c[k] \leftarrow D_c[k] + \frac{\theta}{2}$; else $D_c[k] \leftarrow D_c[k] - \frac{\theta}{2}$;
    end
end
```

4.2.3 Alternating Tree Hungarian Algorithm

This alternating tree variant of the Hungarian algorithm was proposed by Lawler (1976) which improves the performance of the classical variant with a smarter choice of data structures. The pseudocode for this variant is presented below. During the execution, the predecessor information of the columns is updated dynamically, and therefore, it is not required to construct the adjacency list at the beginning of the "search" step. Additionally the algorithm maintains the minimum "slack" ($c_{ij} - u_i - v_j$) for each column, which reduces the complexity of the "dual update" step from $O(n^2)$ to $O(n)$. In this variant, the "search" step is executed exactly n times before an optimal solution is found, and therefore, the complexity of this variant is $O(n^3)$.

```
algorithm alternating_tree_hungarian
input: Matrix $\mathbf{C}$
output: Optimal assignments $A_r$ and $A_c$
begin
    execute initial_reduction;
    repeat
        execute optimality_check; [*** check ***]
        if match_count $= n$ then go to exit;
        foreach $j \in 1, \cdots, n$ do $slack[j] \leftarrow \infty$;
        /* Augmenting path search */
        $ST \leftarrow \emptyset$;                                        /* stack */
        foreach $i \in \{1, \cdots, n\}$ do
            if $V_r[i] = 0$ then $ST.push(i)$;
        end
        while $ST \neq \emptyset$ do [*** search ***]
            $i \leftarrow ST.top()$;
            $ST.pop()$;
            foreach $j \in \{1, \cdots, n\}$ do
                if $slack[j] > \mathbf{C}[i,j] - D_r[i] - D_c[j]$ then
```

```
                    $slack[j] \leftarrow \mathbf{C}[i,j] - D_r[i] - D_c[j]$;
                    $P_c[j] \leftarrow i$;
                end
                if $\mathbf{C}[i,j] - D_r[i] - D_c[j] = 0$ then
                    $i_{new} \leftarrow A_c[j]$;
                    if $V_c[j] = 0$ then                              /* if column is uncovered */
                        if $i_{new} = -1$ then                            /* unassigned column */
                            augment($j$);
                            go to check;
                        else
                            $ST.push(i_{new})$;
                            $P_r[i_{new}] \leftarrow j$;                  /* update predecessor index */
                            $V_r[i_{new}] \leftarrow 0$;                          /* uncover the row */
                            $V_c[j] \leftarrow 1$;                               /* cover the column */
                        end
                    end
                end
            end
        end
        update_2();
        go to search;
    end [*** exit ***]
end
/* Procedure for updating the dual solution */
procedure update_2
input: Cover arrays $V_r$ and $V_c$
output: Updated dual solution arrays $D_r$ and $D_c$
begin
    $\theta \leftarrow \min_j\{slack[j] > 0\}$;
    foreach $k \in \{1, \cdots, n\}$ do
        if $V_r[k] = 0$ then $D_r[k] \leftarrow D_r[k] + \frac{\theta}{2}$; else $D_r[k] \leftarrow D_r[k] - \frac{\theta}{2}$;
        if $V_c[k] = 0$ then $D_c[k] \leftarrow D_c[k] + \frac{\theta}{2}$; else $D_c[k] \leftarrow D_c[k] - \frac{\theta}{2}$;
    end
    foreach $j \in \{1, \cdots, n\}$ do
        if $slack[j] > 0$ then
            $slack[j] \leftarrow slack[j] - \theta$;
            if $slack[j] = 0$ then $ST.push(P_c[j])$;
        end
    end
end
```

4.3 Preliminaries

In this section, we will first introduce the readers to GPU and CUDA architecture (as described in NVIDIA (2012)), and then explain some concepts which will be helpful in devising the parallel algorithm.

4.3.1 Introduction to GPU and CUDA

GPUs are predominantly used for processing and rendering high quality graphics on a computer display. A GPU is built around an array of multi-threaded *streaming multiprocessors* (SMs), each of which contains an array of processor cores. Each processor core is equipped with data processing transistors and on-chip shared memory, which has very low latency. The GPU itself has a global memory, which can be accessed by all SMs but it is slightly slower than the former one. Since a GPU has more number of transistors devoted for data processing than a CPU, it is suitable for parallel computations with high arithmetic intensity.

CUDA is a general purpose parallel programming platform developed by NVIDIA to take advantage of the compute engine in their GPUs. A CUDA program is divided into two parts: (1) *host code* which is executed on the CPU; and (2) *kernels*, which are executed on the GPU. Kernels are blocks of instruction which are executed by a number of threads in parallel. The threads are logically arranged into *blocks* and the blocks are logically arranged into a *grid*. Each block is randomly scheduled on any available multiprocessor. When the multiprocessor finishes processing that block, next block gets assigned to it, and thus the application gets automatic scalability with increasing number of processor cores.

4.3.2 Parallelization Strategy

In the parallel algorithm(s) that we have implemented, each step of the sequential algorithm(s), described in Section 4.2, is executed on the GPU by one or more CUDA kernels. After the execution of each kernel, the control is given to the CPU, for coordinating the program flow. This also provides natural synchronization points in the parallel algorithm. We make the following observations in the sequential algorithm(s), which will provide insights into the parallelization strategy for each step.

1. The initial reduction, optimality check, and dual update steps can be easily parallelized and they possess a higher degree of granularity. It means that we can easily define one thread for each element of the cost matrix (or at least one thread per row/column), all of which can be processed simultaneously. Therefore these steps will benefit the most from parallelization on GPU.

2. During each iteration of the augmenting path search, we are interested in only a small fraction of elements. For example, the augmenting path search step in the classical variant operates only on $m \ll n^2$ zero-slack edges. Therefore, we need to create an array of these relevant elements so that each element can be processed by a single thread and proper utilization of the threads can be achieved.

3. Finally, the augmenting path search step itself is difficult to parallelize since we cannot avoid its iterative nature. However, it is an application of the parallel breadth-first-search algorithm,

which can be implemented efficiently on a GPU (see Section 4.3.4).

To this end, we will describe the concepts of stream compaction and parallel breadth-first search algorithm, in the next two sections.

4.3.3 Sparse Matrix Representation

To construct an array of relevant elements in CUDA, we have used the concept of stream compaction as described by Harris et al. (2007). The main idea behind this operation can be described as follows. Consider an input array A of size n, from which only $m < n$ elements are relevant. To compress these elements, we first define a "predicate" array R of size $n + 1$. In this array we record "1" corresponding to the relevant elements and "0" corresponding to irrelevant elements. Then we perform a prefix-sum operation on this predicate array, which generates the scatter addresses of the relevant elements in the new array. The entry $R[n]$ represents the size m of the new array. Finally, we create an output array Z of size m and scatter the elements to the respective locations as indicated in the predicate array. Figure 4.1(a) shows an example of array compression with 3 relevant elements.

In the classical variant, we also need to store the adjacency list of the edges with zero slack. For this purpose, we have used the *compressed sparse row* (CSR) storage format for matrix compression. Matrix compression can be achieved using the same operations mentioned above, with the exception that we store the column indices of the relevant elements, rather than elements themselves. For this purpose, we need two arrays: adjacency list $\mathbf{Z}$ and row pointer array P. Array $\mathbf{Z}$ is of size m, equal to the number of relevant elements, and it is used to store their column indices, traversed in row-major order. Array P is of size $n + 1$, and the element $P[i]$ points in the array $\mathbf{Z}$, the first relevant element of row i. The sub-array $\mathbf{Z}[P[i]]$ represents the adjacency list of row i, containing all the relevant elements from that row. Its size can be obtained by simply evaluating the expression $P[i+1]-P[i]$. The element $P[n]$ indicates the size of the array $\mathbf{Z}$. Figure 4.1(b) shows an example of the CSR arrays for a matrix $\mathbf{M}$, containing six relevant elements. The adjacency list of row 2 begins from index 1 in $\mathbf{Z}$, and it contains: $3 - 1 = 2$ elements, in columns 0 and 2 respectively.

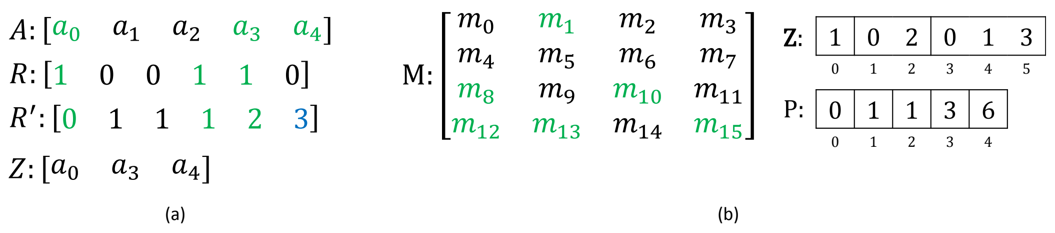


Figure 4.1: (a) Array compression; (b) CSR matrix compression

Parallel prefix-sum is an important operation in array and matrix compression. Given an input array I, the prefix-sum operation produces an output array O in which each element is the sum of all the previous elements of the input array, i.e., $O[i] = \sum_{j=0}^{i-1} I[j]$. Blelloch (1990) first developed an efficient parallel algorithm for the prefix-sum on vector processors, which has a work complexity of $O(n)$ and a step complexity of $O(\log n)$. Sengupta et al. (2006) implemented this work-efficient algorithm on the NVIDIA GPU, which was shown to be significantly faster. Prefix-sum has important applications in sorting, stream compaction, lexical analysis, etc. In our implementation, we have used the prefix-sum function from the Thrust library for CUDA, developed by Hoberock and Bell (2010). The operation of compressing the zero-slack edges has a work complexity of $O(n^2)$ and a step complexity of $O(\log n)$.

4.3.4 Parallel Breadth-first Search Algorithm

Breadth-first search (BFS) is a fundamental algorithm in graph traversal, for finding all vertices satisfying a particular property (Ahuja et al., 1993). The BFS algorithm traverses the graph from a *source* vertex, by successively marking the vertices along the outgoing edges and expanding the frontier. At the termination of this algorithm, we get a *tree* graph, rooted at the source vertex, with the property that a path between the source vertex and any other vertex in the tree, is a shortest path. The complexity of the sequential BFS algorithm is $O(n+m)$, where n is the number of vertices and m is the number of edges in the graph.

Parallelizing the BFS algorithm on a GPU is a non-trivial task. In the simplest implementation of the parallel BFS algorithm, the graph is represented as an adjacency list with n^2 elements. During the execution, the threads scan every edge or at least every vertex and expand the frontier by one hop during each iteration. Since there could be n iterations in the worst case, this parallelization has a quadratic complexity of $O(n^2)$. In most graphs, the number of edges is much smaller than n^2, due to which these quadratic parallelization strategies can prove to be extremely inefficient. For more details on the quadratic parallelization strategies, we direct the readers to Luo et al. (2010).

Recently, Merrill et al. (2012) has proposed a work-efficient parallel algorithm in which each vertex and each edge is scanned exactly once, and hence it has a linear complexity of $O(n + m)$. This parallel algorithm is probably the most efficient implementation of the BFS on a GPU. In this implementation, the graph is stored as a compressed adjacency list $\mathbf{Z}$, using CSR format. During each iteration, the algorithm maintains two *frontier* arrays F_{in} and F_{out}. The array F_{in} contains the vertices which are currently "active," and it is initialized using the *source* vertex(s). The remaining vertices in $\mathbf{Z}$ are marked as "inactive." Each BFS iteration is carried out in the following two phases which are repeated until all the vertices are visited:

1. *Expansion*: In this phase, F_{out} is initialized with a size equal to the total number of neighbors of all the vertices in F_{in}. This operation is also known as *allocation*, which is another application of the parallel prefix-sum. The kernel is executed by defining one thread for each vertex in F_{in}. Each thread traverses its corresponding adjacency list in $\mathbf{Z}$ and *gathers* its neighbors into F_{out}, which serves as a staging ground for the new frontier.

2. *Contraction*: In this phase, F_{out} is compressed by removing the "visited" vertices. After

compression, the array F_{out} represents the new frontier, which is 1-hop distance away from the vertices in F_{in}. Finally, all the vertices in F_{in} are marked as "visited" and they are removed from the array. The vertices from F_{out} are copied into F_{in}, their labels are changed to "active," and the algorithm returns to the expansion phase.

The motivation behind introducing the parallel BFS algorithm in this section is that the augmenting path search of the Hungarian algorithm is similar to constructing multiple trees rooted at some unassigned rows. For this step to have linear time complexity, we need to make sure that each vertex and each edge is scanned at most once. In the sequential algorithm(s), this is achieved with the help of queues and stacks, which are not easy to construct in CUDA. However, using the concept of stream compaction we can construct arrays that mimic the above data structures, for relatively lower computational cost. Thus, to efficiently parallelize the augmenting path search step (both in classical as well as alternating tree variant), we have used the concepts from the parallel BFS algorithm mentioned above. To the best of our knowledge, our algorithm is the first known application of a GPU-based parallel BFS in an LAP solver.

4.4 Accelerating the Hungarian Algorithm

In this section, we will describe the specifics of parallelization for each step of the Hungarian algorithm. All the data structures mentioned in Section 4.2.1 remain the same and they are initialized in the device memory instead of the host memory, so as to minimize host-device memory transactions.

4.4.1 Initial Reduction

In this step, an initial dual feasible solution is found by executing row and column reduction kernel (depicted in Algorithm 6) on the GPU. This kernel is executed with n threads, each corresponding to one row (or column) of the matrix $\mathbf{C}$. At the end of this step, we obtain a dual feasible solution corresponding to the arrays D_r and D_c. These kernels have a work complexity of $O(n^2)$. There is no transfer of data between host and device before and after the execution of this kernel.

4.4.2 Optimality Check

This step is executed using the kernel shown in Algorithm 7, and it serves as an optimality check for the current assignment solution. Initially, the elements from the cover arrays V_r and V_c are reset to 0. Then the kernel is executed with n threads, each corresponding to one element of the assignment array A_r. Each thread checks if the corresponding row is assigned to a column, and if so, it covers that row in the row cover array V_r, and increments an integer variable **match_count**. The work complexity of this kernel is $O(n)$. After the execution of this kernel, we need to transfer the **match_count** (a single integer) from the device to host.

Algorithm 6: Initial reduction kernel

```
Data: Matrix $\mathbf{C}$
Result: Arrays $D_r$ and $D_c$
parallel foreach $i \in \{1, \ldots, n\}$ do
    $D_r[i] \leftarrow \min_j \{\mathbf{C}[i, j]\}$ ;                /* row reduction kernel */
end
synchronization ;
parallel foreach $i \in \{1, \ldots, n\}$ do
    $D_c[j] \leftarrow \min_i \{\mathbf{C}[i, j] - D_r[i]\}$ ;        /* column reduction kernel */
end
```

If all the rows are covered at the termination of this kernel (i.e., **match_count** $= n$), the algorithm is terminated with the optimal assignment corresponding to the arrays A_r and A_c. Otherwise, all the values in the predecessor/successor arrays P_r, P_c, S_r, and S_c are reset to -1, and we go to the augmenting path search step described in the next section. In the alternating tree variant, the *slack* array is reset to ∞.

Algorithm 7: Optimality check kernel

```
Data: Row assignment array $A_r$
Result: Row cover array $V_r$, match_count
parallel for $i \in \{1, \ldots, n\}$ do
    if $A_r[i] \neq -1$ then                                  /* row is assigned */
        $V_r[i] \leftarrow 1$ ;                                /* update row cover */
        match_count $\leftarrow$ match_count $+1$ ;                  /* atomic add */
    end
end
```

4.4.3 Augmenting Path Search

This is the most important step of the Hungarian algorithm, in which an alternating tree is built starting from an unassigned row vertex and potentially ending at an unassigned column vertex, and alternating between assigned and unassigned edges. According to Balas et al. (1991), the augmenting path search can be parallelized in two ways: (1) each processor independently searches for an augmenting path from different unassigned vertices; and (2) several processors jointly attempt to find an augmenting path from the same unassigned vertex. The method that we are proposing can be considered as a hybrid approach, in which multiple CUDA threads jointly search for augmenting paths from all the unassigned rows, and they identify multiple vertex disjoint paths, taking advantage of the "race" condition, all of which can be used to augment the current solution. Although our method is specifically designed for the GPUs, it can be readily extended to multi-core CPUs using OpenMP directives, which adds another facet to our contribution.

The augmenting path search is executed in three phases: forward pass, reverse pass, and augmentation pass, which are described below.

4.4.3.1 Forward Pass

The forward pass is a parallel, iterative BFS, rooted at all unassigned rows containing at least one zero-element. The forward pass algorithms for the classical and alternating tree variants are described below.

Forward Pass in the Classical Variant.

1. Initially, the column indices of zero-slack edges are compressed into the adjacency list **Z**, using the CSR format. Since the matrix compression is an expensive operation, it is performed only if "dual update" was executed in the previous iteration and new zero-slack edges were introduced. Otherwise, the adjacency list from the previous iteration can be reused. This small modification leads to significant improvement in the execution time. After constructing the adjacency list, the indices of the unassigned rows having at least one neighbor column are marked as "active" and they are added to the frontier array F_{in}. The indices of rows with no neighboring columns are marked as "visited." All the remaining row indices are marked as "inactive." All the column indices are also marked as "inactive."

2. Next, the *expansion* phase of the BFS is executed with one thread for each element in F_{in}. Main steps of this phase are outlined in Algorithm 8. During the execution, each thread traverses the "inactive" column indices, from its adjacency list in **Z**; looks up the subsequent row indices from the assignment array A_c; and *gathers* these row indices into F_{out}. During its traversal, the thread updates the predecessor arrays (P_r, P_c), and the cover arrays (V_r, V_c); and marks the column indices as "visited," to prevent cycling. Unassigned column indices are marked as "reverse," which are possible candidates for the reverse pass. All the row indices in F_{in} are marked as "visited" and they are removed from the array.

3. Next, the *contraction* phase of the BFS is executed, in which F_{out} is compressed by removing any "visited" row indices. The remaining row indices from F_{out} are marked as "active;" they are copied into F_{in}; and the algorithm returns to the *expansion* phase with this new frontier. The two phases are repeated until no more "active" row indices can be found.

4. If there exists at least one column marked as "reverse," then the current solution can be improved by executing the reverse and augmentation passes, as explained in Sections 4.4.3.2 and 4.4.3.3. Otherwise, the dual solution needs to be updated to introduce new zero-slack edges, as explained in Section 4.4.4.

Algorithm 8: Forward pass expansion kernel in classical variant

```
Data: Frontier array $F_{in}$, Adjacency list $\mathbf{Z}$, Assignment array $A_c$, Cover arrays $V_r$ and $V_c$
Result: Frontier array $F_{out}$, Modified predecessor arrays $P_r$ and $P_c$, Modified cover arrays
        $V_r$ and $V_c$
parallel foreach $i \in F_{in}$ do
    foreach $j \in \boldsymbol{Z}_i$ do
        if $V_c[j] = 0$ then                              /* column $j$ is uncovered */
            $P_c[j] \leftarrow i$ ;                       /* update predecessor of column $j$ */
            $i_{new} \leftarrow A_c[j]$ ;                 /* lookup assignment of column $j$ */
            if $i_{new} = i$ then continue ;              /* continue on to next $j$ */
            if $i_{new} \neq -1$ then                     /* column $j$ is assigned */
                $P_r[i_{new}] \leftarrow j$ ;             /* update predecessor of row $i_{new}$ */
                $V_r[i_{new}] \leftarrow 0$ ;             /* uncover row $i_{new}$ */
                $V_c[j] \leftarrow 1$ ;                   /* cover column $j$ */
                if $i_{new}$ not "visited" then
                    Mark $i_{new}$ as "active" ;
                end
                Gather $i_{new}$ into $F_{out}$ ;
            else                                          /* column $j$ is unassigned */
                Mark $j$ as "reverse" ;                   /* reverse pass candidate */
            end
        end
    end
    Mark $i$ as "visited" ;
end
```

Forward Pass in the Alternating Tree Variant.

1. Initially, the unassigned row indices are marked as "active" and added to the frontier array F_{in}. All the remaining row indices are marked as "inactive."

2. Next, the *expansion* phase of the BFS is executed with one thread per column vertex (main difference between this variant and the classical one). Main steps of this phase are outlined in Algorithm 9. During the execution, each thread traverses the current frontier and updates the minimum "slack" value and corresponding predecessor row index for the column vertex. Then, the same thread looks up the subsequent row index from the assignment array A_c and marks it as "active" for the next frontier (if it is "inactive" in the current iteration). During this traversal, the thread updates the predecessor arrays (P_r, P_c), and the cover arrays (V_r, V_c); and marks the column indices as "visited," to prevent cycling. Unassigned column indices with zero slack are marked as "reverse," which are possible candidates for the reverse pass. All the row indices in F_{in} are marked as "visited" and they are removed from the array.

3. Next, the *contraction* phase of the BFS is executed, in which the "active" row indices are

compressed into F_{in}; and the algorithm returns to the *expansion* phase with this new frontier. The two phases are repeated until no more "active" row indices can be found.

4. Once again, if there exists at least one column marked as "reverse," then the current solution can be improved by executing the reverse and augmentation passes, as explained in Sections 4.4.3.2 and 4.4.3.3. Otherwise, the dual solution needs to be updated to introduce new zero-slack edges, as explained in Section 4.4.4.

Algorithm 9: Forward pass expansion kernel in alternating tree variant

```
Data: Frontier array $F_{in}$, Matrix $\mathbf{C}$, Dual arrays $D_r$ and $D_c$, Assignment array $A_c$, Cover
      arrays $V_r$ and $V_c$, slack array
Result: Modified predecessor arrays $P_r$ and $P_c$, Modified cover arrays $V_r$ and $V_c$
parallel foreach $j \in \{1, \cdots, n\}$ do
    if $V_c[j] = 0$ then                                          /* column $j$ is uncovered */
        foreach $i \in F_{in}$ do
            if $slack[j] > \mathbf{C}[i,j] - D_r[i] - D_c[j]$ then
                $slack[j] = \mathbf{C}[i,j] - D_r[i] - D_c[j]$ ;          /* update slack of column $j$ */
                $P_c[j] \leftarrow i$ ;                               /* update predecessor of column $j$ */
            end
            $i_{new} \leftarrow A_c[j]$ ;                              /* lookup assignment of column $j$ */
            if $slack[j] = 0$ then
                if $i_{new} \neq -1$ then                                 /* column $j$ is assigned */
                    $P_r[i_{new}] \leftarrow j$ ;                      /* update predecessor of row $i_{new}$ */
                    $V_r[i_{new}] \leftarrow 0$ ;                                /* uncover row $i_{new}$ */
                    $V_c[j] \leftarrow 1$ ;                                        /* cover column $j$ */
                    Mark $i_{new}$ as "active" ;
                else                                                      /* column $j$ is unassigned */
                    Mark $j$ as "reverse" ;                               /* reverse pass candidate */
                end
            end
        end
    end
    Mark $i$ as "visited" ;
end
```

Correctness of Forward Pass. At the termination of of the forward pass, we obtain one or more directed out trees, represented by the predecessor arrays P_r and P_c, each of which is: (a) rooted at unassigned rows, (b) ending at either assigned rows or unassigned columns, and (c) alternating between assigned and unassigned edges. These alternating trees exhibit a very important property, as proved by the following proposition.

Proposition 1. *The alternating trees produced by the forward pass algorithm are vertex-disjoint.*

Proof. This proposition can be proved using the structure of the graph containing assigned and unassigned zero-slack edges. We make the following important observations: (1) each column vertex with an incoming unassigned edge can have at most one outgoing assigned edge; (2) each row vertex with an incoming assigned edge can have multiple outgoing unassigned edges; (3) during the forward pass, only one of the predecessor indices of a column vertex will survive, due to the race condition. Therefore, the structure of any tree obtained during forward pass is such that each row can have at most one column as its predecessor and multiple columns as successors; while each column can have at most one row as its predecessor and one row as its successor.

Now, let us assume that two alternating trees T_1 and T_2 rooted at rows R_{i_1} and R_{i_2} ($R_{i_1} \neq R_{i_2}$) are not vertex-disjoint. It means that the two trees either merge at some common row vertex or column vertex. Let us also assume that the two trees merge at a common row vertex R_{i_k}, such that $R_{i_k} \in T_1$ and $R_{i_k} \in T_2$. It means that the row R_{i_k} must have two predecessor columns $C_{j_p} \in T_1$ and $C_{j_q} \in T_2$. However, the row R_{i_k} can have at most one predecessor. Therefore, either $R_{i_k} \in T_1$, or $R_{i_k} \in T_2$, and not both, which is a contradiction. If we assume that the two trees merge at a common column vertex, we arrive at a similar contradiction. Therefore, the trees T_1 and T_2 must be vertex-disjoint.

□

The importance of having multiple vertex-disjoint trees can be explained as follows. If more than one of those trees contain at least one unassigned column as a leaf vertex, then we can get more than one alternating paths. All these paths can be used to increase the current number of assignments, as opposed to only one potential assignment per iteration in the sequential algorithm. Therefore, the parallel algorithm can converge to the optimal solution in fewer number of iterations, thereby reducing the overall execution time. After the execution of this kernel, we need to transfer a boolean flag from the device to host which indicates whether reverse pass or dual update should be executed next.

4.4.3.2 Reverse Pass

The alternating trees obtained during the forward pass are vertex-disjoint, however, each tree can potentially have multiple unassigned columns as leaf vertices. Therefore, to identify alternating, vertex-disjoint paths, we execute the reverse pass algorithm. To improve the thread utilization, we create a compressed array F_{rev} containing only those column indices which are labeled as "reverse" during forward pass. Then we execute the kernel by defining one thread per element of F_{rev}. The steps involved in the reverse pass are outlined in Algorithm 10. The work complexity of the reverse pass algorithm is $O(n)$ per thread. There is no transfer of data between host and device before and after the execution of this kernel.

During the execution, each thread traverses the tree by looking up the predecessor indices of the rows and columns from the arrays P_r and P_c, and records the successor indices in the arrays S_r and S_c. At the termination, we obtain one or more directed paths, represented by the successor arrays

Algorithm 10: Reverse pass kernel

```
Data: Array $F_{rev}$, Predecessor arrays $P_r$ and $P_c$
Result: Modified successor arrays $S_r$ and $S_c$
parallel foreach $j \in F_{rev}$ do
    $r_{cur} \leftarrow -1$ ;
    $c_{cur} \leftarrow j$ ;
    while $c_{cur} \neq -1$ do          /* repeat until current row has no predecessor */
        $S_c[c_{cur}] \leftarrow r_{cur}$ ;          /* update successor of current column index */
        $r_{cur} \leftarrow P_c[c_{cur}]$ ;                    /* update current row index */
        $S_r[r_{cur}] \leftarrow c_{cur}$ ;             /* update successor of current row index */
        $c_{cur} \leftarrow P_r[r_{cur}]$ ;                 /* update current column index */
    end
    Mark $r_{cur}$ as "augment" ;                 /* augmentation pass candidate */
end
```

S_r and S_c, each of which is: (a) rooted at an unassigned row, (b) ending at an unassigned column, and (c) alternating between assigned and unassigned edges. These paths are also vertex-disjoint, as proved by the following proposition.

Proposition 2. *The alternating paths produced by the reverse pass algorithm are vertex-disjoint.*

Proof. Consider two different trees T_1 and T_2 obtained during forward pass. From Proposition 1, we know that T_1 and T_2 are vertex-disjoint. Additionally, if each of those trees has only one leaf vertex, then the two paths $P_1 = T_1$ and $P_2 = T_2$ must be vertex-disjoint.

Now, consider a single tree with multiple leaf vertices, each of which is assigned to one thread. Each thread traverses the tree and marks the successor indices of the rows and columns. Due to the structure of the tree, any two paths can potentially converge at a row and the thread responsible for each path will try to update the successor index of that row. However, due to the race condition, only one thread will succeed, and therefore only one of the alternating paths will survive. From Proposition 1, this path must be vertex-disjoint from any paths arising from other alternating trees. □

4.4.3.3 Augmentation Pass

In this step, the alternating paths obtained during the reverse pass are used to augment the current assignment solution. Again, we create a compressed array F_{aug} containing only those row indices which are labeled as "augment" during reverse pass. Then, we execute the kernel by defining one thread per element of F_{aug}. The steps involved in the augmentation pass are outlined in Algorithm 11. The work complexity of the augmentation pass algorithm is $O(n)$ per thread. There is no transfer of data between host and device before and after the execution of this kernel.

During the execution, each thread traverses a single path by looking up the successor indices from arrays S_r and S_c; and interchanges the assigned and unassigned edges along that path. At

Algorithm 11: Augmentation pass kernel

```
Data: Array $F_{aug}$, Successor arrays $S_r$ and $S_c$
Result: Modified assignment arrays $A_r$ and $A_c$
parallel foreach $i \in F_{aug}$ do
    $r_{cur} \leftarrow i$ ;
    $c_{cur} \leftarrow -1$ ;
    while $r_{cur} \neq -1$ do        /* repeat until current column has no successor */
        $c_{cur} \leftarrow S_r[r_{cur}]$ ;              /* update current column index */
        $A_r[r_{cur}] \leftarrow c_{cur}$ ;                /* update row assignment */
        $A_c[c_{cur}] \leftarrow r_{cur}$ ;              /* update column assignment */
        $r_{cur} \leftarrow S_c[c_{cur}]$ ;              /* update current row index */
    end
end
```

the termination, the number of assignments in the current solution will be increased by the total number of traversed paths, and we return to the optimality check in Section 4.4.2.

4.4.4 Dual Solution Update

In this step the dual solution is updated and new edges with zero slack are introduced. The algorithms for the two variants are described below.

Dual Update for the Classical Variant. The dual update for classical variant is executed in two stages. Initially, Stage 1 kernel (depicted in Algorithm 12) is executed with n threads, each corresponding to one row of the matrix $\mathbf{C}$. This kernel finds the minimum uncovered slack θ. This kernel has work complexity of $O(n^2)$. There is no transfer of data between host and device before and after the execution of this kernel.

Next, Stage 2 kernel (depicted in Algorithm 13) is executed with n threads, each corresponding to one row or column. The kernel updates the dual variables, which creates at least one new zero-slack edge. This kernel has a work complexity of $O(n)$. After termination of this kernel, the algorithm returns to the augmenting path search step in Section 4.4.3. There is no transfer of data between host and device before and after the execution of this kernel.

Dual Update for the Alternating Tree Variant. The dual update for the alternating tree variant is similar to that of the classical variant, with a few exceptions. Initially, the minimum non-zero slack θ can be found using a simple $O(n)$ reduction operation of the *slack* array, i.e., by computing $\theta = \min_j\{slack[j] > 0\}$. Once we have the θ value, then we can execute the kernel depicted in Algorithm 14 with n threads, which updates the dual variables and the column slacks. If some column has zero slack, then its predecessor row is marked as "active," and the algorithm returns to the augmenting path search step in Section 4.4.3. There is no transfer of data between

Algorithm 12: Dual update kernel in classical variant: Stage 1

```
Data: Cost matrix $\mathbf{C}$, Dual arrays $D_r$ and $D_c$, Cover arrays $V_r$ and $V_c$, Temp array $U$
Result: Minimum uncovered slack $\theta$
parallel foreach $i \in \{1, \dots, n\}$ do
    $U[i] \leftarrow \infty$ ;
    if $V_r[i] = 0$ then                                   /* row $i$ is uncovered */
        foreach $j \in \{1, \dots, n\}$ do
            if $V_c[j] = 0$ then                           /* column $j$ is uncovered */
                $U[i] \leftarrow \min\{\mathbf{C}[i,j] - D_r[i] - D_c[j], U[i]\}$
            end
        end
    end
end
if $tid = 0$ then
    $\theta \leftarrow \infty$ ;                 /* executed by a single thread with id $= 0$ */
    foreach $i \in \{1, \dots, n\}$ do
        $\theta \leftarrow \min\{\theta, U[i]\}$ ;                  /* update the minimum */
    end
end
```

host and device before and after the execution of this kernel.

4.4.5 Overall Algorithm Complexity

The overall complexity of the accelerated algorithms remain the same as their sequential counterparts, however, the work is split between multiple threads, which translates into significant parallel speedup. For non-pathological cases, the accelerated algorithms potentially find more than one augmenting paths per iteration, and therefore they rapidly converge to the optimal solution. In Section 4.6, we will empirically demonstrate the benefits of our accelerated algorithms over the sequential algorithm.

4.5 Multi-GPU Implementation

We implemented both the classical and alternating tree variants of the Hungarian algorithm in a multi-GPU setting. In the single GPU implementation, it may become challenging to store the cost matrix in the GPU memory, for larger problems. One of the alternatives to overcome this problem is to split the cost matrix across multiple GPUs. Consequently, some of the steps of the algorithm can be performed in parallel on multiple GPUs and additional parallel speedup can be achieved. We have used grid architecture with multiple compute nodes, each containing one CPU-GPU pair. Communication between the different CPUs is accomplished using *message passing interface* (MPI). The overall algorithmic scheme for this multi-GPU implementation is described below:

Algorithm 13: Dual update kernel in classical variant: Stage 2

```
Data: Minimum uncovered slack $\theta$, Cover arrays $V_r$ and $V_c$
Result: Modified dual arrays $D_r$ and $D_c$
parallel foreach $k \in \{1, \dots, n\}$ do
    if $V_r[k] = 0$ then
        $D_r[k] \leftarrow D_r[k] + \frac{\theta}{2}$;
    else
        $D_r[k] \leftarrow D_r[k] - \frac{\theta}{2}$;
    end
    if $V_c[k] = 0$ then
        $D_c[k] \leftarrow D_c[k] + \frac{\theta}{2}$
    else
        $D_c[k] \leftarrow D_c[k] - \frac{\theta}{2}$
    end
end
```

1. **Initialization:** The program is initialized with p MPI processes, equal to the number of nodes in the grid. It is assumed that one MPI process gets allocated to exactly one node. The node with rank 0 is chosen as the root. The rows of the cost matrix $\mathbf{C}$ are split evenly between all the devices in the grid, i.e., each device owns a sub-matrix $\mathbf{C}^i$ of size $\left\lceil \frac{n}{p} \right\rceil \times n$. For the alternating tree variant, we divide the columns of the cost matrix among the devices, instead of the rows.

2. **Initial reduction:** The row reduction step from Algorithm 6 can be trivially parallelized, and it is simultaneously executed by all the devices on their individual sub-matrices to obtain partial row dual arrays. These partial arrays are transferred to the host ($O(n/p)$ memory transfer) and gathered at the root to construct an array of global row duals. During the column reduction phase, each device independently finds the local column duals from the sub-matrix $\mathbf{C}^i$ and stores them in an array. These arrays are first transferred from device to host ($O(n)$ memory transfer); then they are gathered at the root using "MPI_Gather" operation; and finally global column duals are identified on the root by finding the minimum for each column. The arrays of global row and column duals are broadcast to all the nodes, which are then transferred to the corresponding devices ($O(n)$ memory transfer).

3. **Optimality check:** In this step, each device independently executes Algorithm 7 on the rows within its scope and counts the number of assigned rows. The total count is calculated at the root using "MPI_Reduce" operation, and it is broadcast to all the nodes. If all the rows are assigned, then the program is terminated, otherwise augmenting path search is executed.

4. **Augmenting path search:**

   (a) In the **classical variant**, the matrix compression operation is the most expensive step

Algorithm 14: Dual update kernel in alternating tree variant

```
Data: Minimum uncovered slack $\theta$, Cover arrays $V_r$ and $V_c$, Predecessor array $P_c$
Result: Modified dual arrays $D_r$ and $D_c$, Modified $slack$ array
parallel foreach $k \in \{1, \ldots, n\}$ do
    if $V_r[k] = 0$ then
        $D_r[k] \leftarrow D_r[k] + \frac{\theta}{2}$;
    else
        $D_r[k] \leftarrow D_r[k] - \frac{\theta}{2}$;
    end
    if $V_c[k] = 0$ then
        $D_c[k] \leftarrow D_c[k] + \frac{\theta}{2}$
    else
        $D_c[k] \leftarrow D_c[k] - \frac{\theta}{2}$
    end
    if $slack[j] > 0$ then
        $slack[j] \leftarrow slack[j] - \theta$;
        if $slack[j] = 0$ then Mark $P_c[j]$ as "active";
    end
end
```

of the algorithm (as seen in Section 4.6), but it can be trivially parallelized. Each device independently compresses the zero-slack edges from each row, into a partial adjacency list $\mathbf{Z}^i$. These partial adjacency lists are first transferred to the host ($O(m/p)$ memory transfer) and then they are gathered at the root node to construct a complete adjacency list $\mathbf{Z}$. After matrix compression, the forward pass (including Algorithm 8) is executed only on the root node, as described in Section 4.4.3.

(b) In the **alternating tree variant**, the matrix compression operation does not exist and the forward pass is the main bottleneck. Recall, that forward pass is an iterative BFS, in which the vertex frontier is expanded during each iteration. To parallelize this step, we divide the column vertices equally among the devices. The initial frontier consists of unassigned row vertices. Each device executes the *expansion* phase of the parallel BFS (Algorithm 9), producing a local frontier of row vertices. In the *contraction* phase, these local frontier lists are gathered at the root node using "MPI_Gather" operation and a global frontier is created. This frontier is broadcast to all the nodes for the next BFS iteration. Since this step requires $O(n)$ memory transfer between host and device plus $O(n)$ MPI communication during each iteration, the benefit of parallelization is outweighed by the communication and this particular implementation has poor scalability (as seen in Section 4.6).

In both variants, if an augmenting path is found, then the current solution is improved by executing the reverse and augmentation passes (Algorithms 10 and 11) on the root node. For the alternating tree variant, the partial column predecessor arrays are gathered at the root node before executing the reverse pass. After the augmentation pass is executed, modified assignment arrays are broadcast to all the nodes and transferred to the corresponding devices ($O(n)$ memory transfer) and the algorithm returns to the optimality check. Otherwise, modified cover arrays are broadcast to all the nodes and transferred to the corresponding devices ($O(n)$ memory transfer), and then the dual update step is executed.

5. **Dual update:** The dual update step can also be trivially parallelized. Initially, each device independently identifies the minimum non-zero slack (from the sub-matrix $\mathbf{C}^i$ for the classical variant using Algorithm 12 or from the column slack array for the alternating tree variant using the reduction operation). The root node reduces these local minima and identifies the global minimum slack θ. This value is broadcast to all the nodes, and it is used to update the dual variables on all devices (using Algorithm 13 or Algorithm 14). The algorithm then returns to the augmenting path search step. The memory transfer between host and device is $O(1)$.

We would like to point out that in the classical variant, the augmenting path search is performed on the root node, for which the adjacency list needs to be stored in the memory of the root GPU. Therefore, this approach is not completely immune to the memory restrictions for substantially large problems. One way to tackle this issue is to store the adjacency list in the CPU memory and copy it in the GPU memory as required. This approach, however, may add significant communication overhead, and alternative options need to be explored. The alternating tree variant does not have this memory restriction, which means that we can potentially solve problems of any size, provided we have sufficient number of GPUs. However, in this approach the forward pass requires synchronization after every BFS iteration, which introduces significant MPI communication overhead and limits the scalability of the algorithm. In a different architecture containing multiple GPUs on a single host, it might be possible to achieve good parallel scalability, with the help of *unified virtual addressing* (UVA) and *peer-to-peer* (P2P) memory access.

4.6 Computational Experiments

We implemented both variants in CUDA C programming language, and the tests were performed on the computational resources from the Blue Waters Supercomputing Facility at the University of Illinois at Urbana-Champaign. The host code was executed on AMD Interlagos model 6276 processor, with 8 cores, 2.3GHz clock speed, and 32GB memory. The device code was executed on NVIDIA GK110 "Kepler" K20X GPU, with 2688 processor cores, and 6GB memory. The total execution times reported in the tables contain the time required for initialization of arrays on the host and device, the execution time of the algorithm, and the time required for cleanup. We also developed an OpenMP version of our parallel algorithm, in which the kernel-analogues were implemented on the host using `# pragma omp parallel for` directives.

4.6.1 Single GPU Experiments: Small Scale

For these tests, the problem instances were generated with the number of vertices ranging from $n = 500$ to $n = 5000$, with increments of 500. The cost matrices were fully dense and the elements were randomly drawn from a uniform distribution of integers in the range $[0, n]$. For each problem size, we generated 3 instances and performed 5 repetitions on each instance (total 15 runs). For each run, we recorded the total execution time (in seconds). We compared OpenMP/CPU implementation of the alternating tree variant (1 thread and 8 thread versions) with the CUDA/GPU versions of classical and alternating tree variants. The average computational results for the two algorithms are shown in Table 4.1 and Fig. 4.2. From the results, we can see that the sequential (single thread OpenMP) implementation is much more efficient for problems with $n \leq 1000$, due to the absence of data transfer and thread invocation overheads involved in the multi-threaded OpenMP and CUDA versions. However, the multi-threaded OpenMP and CUDA versions outperform the sequential version for problems with $n \geq 1500$. Moreover, the performance of our CUDA/GPU algorithms is superior to the multi-threaded OpenMP version due to availability of a lot more threads and processor cores on the GPU, as compared to the CPU. We can also see that the classical variant performs better than the alternating tree variant on these small problems, owing to its higher granularity of parallelization. Table 4.2 shows the number of augmenting paths found during various steps in the GPU accelerated alternating tree variant. We can see that the algorithm finds large number of augmenting paths during initial stages, and the number decreases with the increasing iteration count.

Table 4.1: Test results for small instances

n	Obj Val	Time (s)			
		OMP-1	OMP-8	CU-CLASS	CU-TREE
500	568.0	0.07	0.13	0.51	0.45
1000	1161.0	0.31	0.34	0.60	0.50
1500	1730.3	0.79	0.66	0.68	0.70
2000	2352.3	1.55	1.03	0.84	0.82
2500	2861.0	2.35	1.36	0.86	0.87
3000	3467.3	3.92	1.93	0.98	1.32
3500	4066.0	5.41	2.50	1.11	1.48
4000	4715.7	7.48	3.21	1.19	1.56
4500	5291.7	9.27	3.79	1.24	1.59
5000	5812.0	11.64	4.35	1.39	1.70

4.6.2 Single GPU Experiments: Large Scale

For these tests, we followed a slightly modified experimental setup of Balas et al. (1991). The problems in this set were generated with the number of vertices ranging from $n = 5000$ to $n = 20,000$, with increments of 5000. The cost matrices were fully dense and the elements were randomly

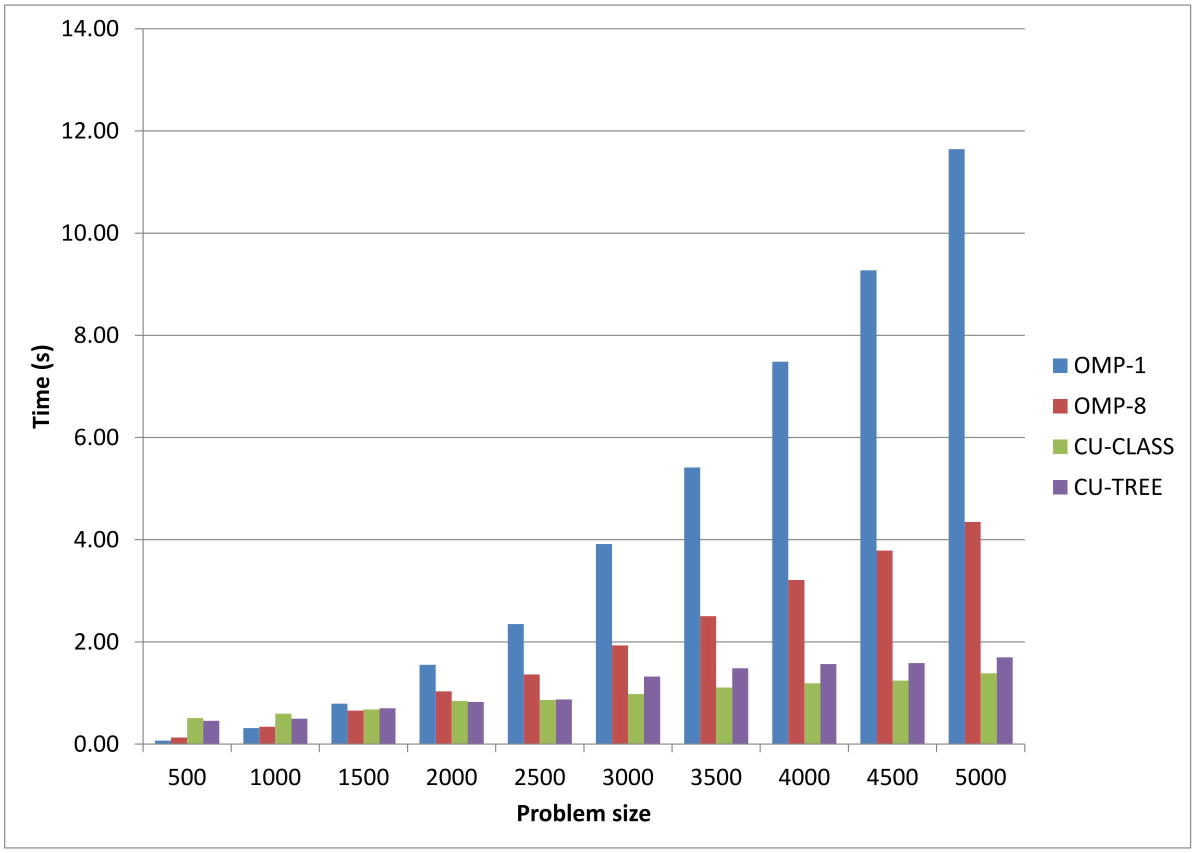


Figure 4.2: Comparison chart for execution time (s)

drawn from a uniform distribution of integers in the ranges $[0, \frac{n}{10}]$, $[0, n]$, and $[0, 10n]$. For each problem size and cost range, we generated 3 instances, and performed 5 repetitions on each instance (total 15 runs). For each run, we recorded the execution times for various steps (in seconds). In Table 4.3, we have presented the average computational results for the OpenMP/CPU version of the alternating tree variant (with 8 threads) and the CUDA/GPU versions of the two variants. In Fig. 4.3 we have shown the contribution of individual operations towards the overall execution time, for $n = 20,000$. Again, we can see that the GPU-accelerated versions are much more efficient than the multi-core CPU version. We can also see that, as the zero-elements in the cost matrix become sparser, the number of initial assignments decreases. Additionally, the relative contribution of the matrix compression (in classical variant), forward pass, and dual update steps increases. This is due to the fact that, in a cost matrix with more dissimilar values, fewer augmenting paths are found during each iteration, and the algorithm spends most of the time finding the maximum matching and updating the dual variables. We can also see that in the classical variant, matrix compression step becomes the primary bottleneck for larger problems due to its $O(n^2)$ complexity, and in those cases, the alternating tree variant starts to dominate. For the classical variant, memory is another

Table 4.2: Number of assignments found during different stages for CU-TREE

n	Initial Assignments	Assignments in Iterations						
		[1-5]	[6-10]	[11-15]	[16-20]	[21-25]	[26-30]	[31-35]
1000	866	96	26	6	6			
2000	1745	177	47	18	10	3		
3000	2608	203	143	24	10	6	6	
4000	3473	254	184	52	21	11	4	1
5000	4316	354	231	61	10	22	4	2

limiting factor for instances with size $n \geq 24,000$ because we need to store both the cost matrix and the adjacency list. Therefore, we have to resort to the multi-GPU implementation, which can address both these bottlenecks to some extent. On the contrary, the alternating tree variant can handle larger problems with $n \leq 35,000$ without having to go to the GPU cluster.

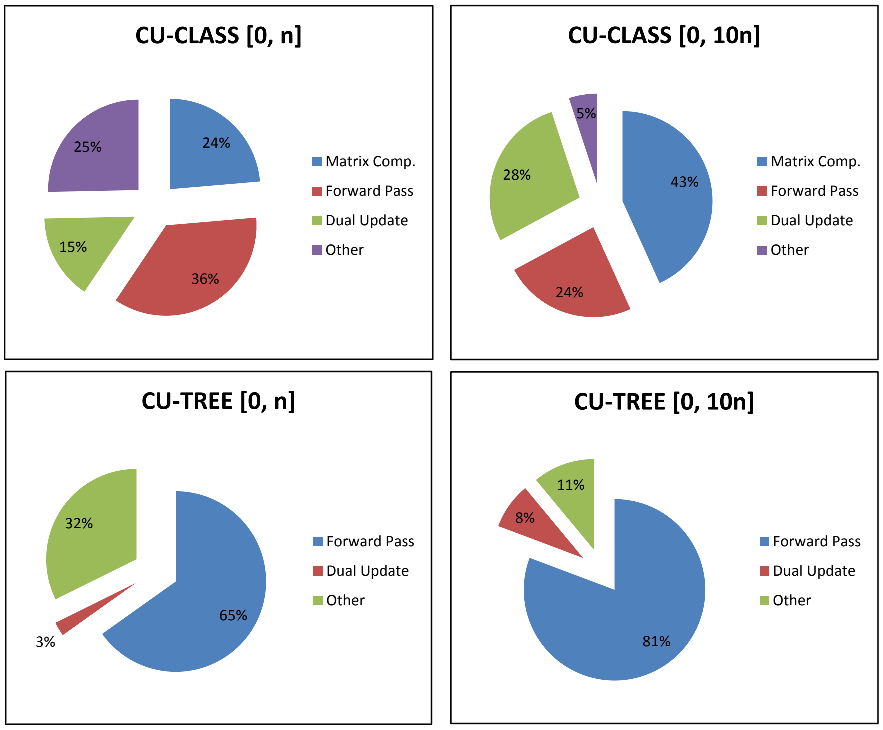


Figure 4.3: Contribution of various steps in execution time for $n = 20,000$

Table 4.3: Test results for large instances

Cost Range $[0, \frac{n}{10}]$[†]

n	Obj Val	Initial Assignment Count	Time (s)									
			OMP-8			CU-CLASS				CU-TREE		
			Forward Pass	Dual Update	Total	Matrix Comp.	Forward Pass	Dual Update	Total	Forward Pass	Dual Update	Total
5000	1.0	5000.0	–	–	1.87	–	–	–	0.44	–	–	0.51
10000	0.7	10000.0	–	–	6.18	–	–	–	0.63	–	–	1.15
15000	0.7	15000.0	–	–	13.99	–	–	–	0.97	–	–	1.55
20000	0.3	20000.0	–	–	26.42	–	–	–	1.41	–	–	2.05

Cost Range $[0, n]$

n	Obj Val	Initial Assignment Count	Time (s)									
			OMP-8			CU-CLASS				CU-TREE		
			Forward Pass	Dual Update	Total	Matrix Comp.	Forward Pass	Dual Update	Total	Forward Pass	Dual Update	Total
5000	5795.3	4326.7	3.91	0.00	4.92	0.08	0.81	0.08	1.43	0.90	0.03	1.39
10000	11748.3	8653.3	12.35	0.00	15.83	0.32	1.25	0.22	2.44	1.48	0.05	2.17
15000	17439.0	13032.7	27.98	0.00	36.61	0.69	1.65	0.51	3.85	2.18	0.07	3.21
20000	23393.3	17350.0	49.03	0.00	65.77	1.34	2.03	0.87	5.69	2.75	0.11	4.22

Cost Range $[0, 10n]$

n	Obj Val	Initial Assignment Count	Time (s)									
			OMP-8			CU-CLASS				CU-TREE		
			Forward Pass	Dual Update	Total	Matrix Comp.	Forward Pass	Dual Update	Total	Forward Pass	Dual Update	Total
5000	5795.3	4049.3	8.11	0.01	9.00	0.72	2.52	0.66	4.40	3.14	0.22	3.86
10000	11748.3	8140.3	30.59	0.02	33.90	2.81	4.28	1.95	9.74	5.52	0.46	6.67
15000	17439.0	12193.7	68.80	0.03	77.11	8.01	5.79	5.92	20.78	7.96	0.83	9.81
20000	23393.3	16268.7	121.88	0.04	137.98	12.99	7.18	8.36	30.05	10.34	1.06	12.81

[†]Augmenting path search and dual update steps are not required to be executed for these problem instances.

4.6.3 Multi-GPU Experiments

For the multi-GPU implementation, we conducted weak and strong scalability studies. In weak scalability tests, the problem size is increased in proportion to the number of processing elements. These results are used to demonstrate the scaling behavior of algorithms which are primarily bounded by the memory. In strong scalability tests, the problem size is kept constant and the number of processing elements are increased. These results are used to demonstrate the scaling behavior of algorithms which are primarily bounded by the CPU.

For weak scalability tests, initial problem is generated with $n = 10,000$ vertices, for a single GPU. The number of GPUs is doubled up to 16, and for each increment, the problem size is multiplied by $\sqrt{2}$, ensuring that each GPU contains about 100 million cost elements. The cost elements are randomly drawn from a uniform distribution of integers between $[0, 10n]$. The weak scaling efficiency of the algorithm is calculated as: $E_{\text{weak}} = \frac{t_1}{t_p}$, where t_p represents the execution time observed for p number of GPUs. The results for weak scalability tests are shown in Table 4.4, and Fig. 4.4. Ideally, the weak scaling efficiency should remain constant (equal to 1). We can see that the overall efficiency of our parallel algorithm(s) deteriorates with increasing number of GPUs (and problem size). The matrix compression step exhibits very good scaling behavior, which is the primary bottleneck in the classical variant. The multi-GPU version of the alternating tree variant shows significant increase in the execution time (even for a single CPU-GPU pair), due to the MPI directives and communication overhead in the parallel BFS.

Table 4.4: Weak scalability results

n	GPUs	CU-CLASS						CU-TREE			
		Time (s)			Scaling Efficiency			Time (s)		Scaling Efficiency	
		Matrix Comp.	AP Search	Overall	Matrix Comp.	AP Search	Overall	AP Search	Overall	AP Search	Overall
10000	1	3.66	4.97	10.42	1.00	1.00	1.00	15.26	16.28	1.00	1.00
14142	2	3.80	6.70	12.48	0.96	0.74	0.83	18.70	20.26	0.82	0.80
20000	4	4.09	8.42	15.03	0.90	0.59	0.69	24.50	26.51	0.62	0.61
28284	8	4.36	11.43	18.58	0.84	0.43	0.56	34.00	37.03	0.45	0.44
40000	16	4.64	14.91	23.59	0.79	0.33	0.44	52.17	58.32	0.29	0.28

For strong scalability tests, problems are generated with $n = 10,000$, $n = 14,142$, and $n = 20,000$ vertices, with cost elements between $[0, 10n]$. For each problem, the number of GPUs is increased from 1 to 16, in powers of 2. The strong scaling efficiency for p number of GPUs is calculated as: $E_{\text{strong}} = \frac{t_1}{p \times t_p}$. The results for strong scalability tests are shown in Table 4.5, and Fig. 4.5. We can see that for all problem sizes, the overall efficiency of our parallel algorithm(s) deteriorates sharply with increasing number of GPUs. Once again, the matrix compression step shows excellent scaling behavior, and the scaling efficiency improves with the increasing problem size.

From the weak and strong scalability results, we can conclude that the multi-GPU implementation of the classical variant is a viable alternative for solving large problems with dense cost structures (e.g., $[0, 10n]$). This is because the matrix compression step is the dominant contributor in the execution time and it is embarrassingly parallelizable. However, for smaller problems with

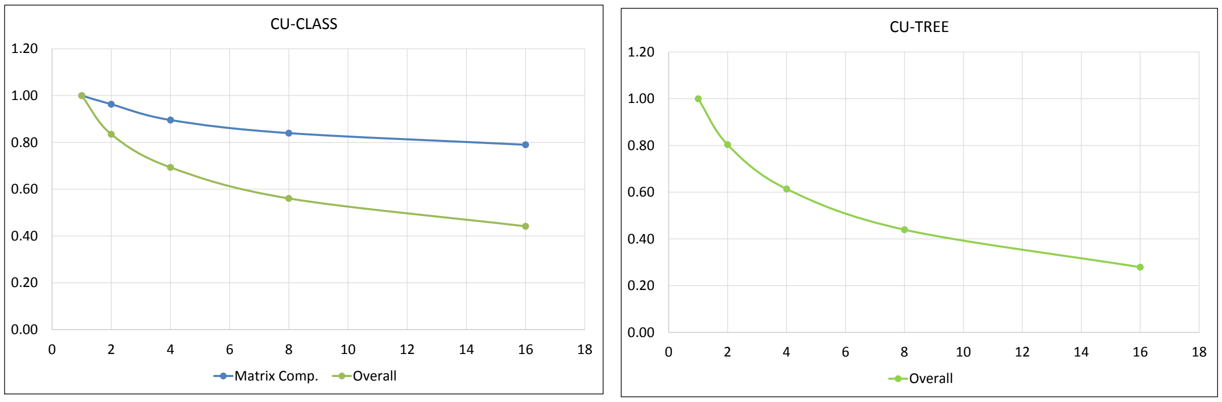


Figure 4.4: Weak scaling efficiency

Table 4.5: Strong scalability results

n	GPUs	CU-CLASS						CU-TREE			
		Time (s)			Scaling Efficiency			Time (s)		Scaling Efficiency	
		Matrix Comp.	AP Search	Overall	Matrix Comp.	AP Search	Overall	AP Search	Overall	AP Search	Overall
10000	1	3.66	4.93	11.07	1.00	1.00	1.00	15.20	16.52	1.00	1.00
10000	2	1.89	4.97	8.15	0.97	0.50	0.68	14.14	15.12	0.54	0.55
10000	4	1.01	5.01	7.84	0.91	0.25	0.35	14.35	15.78	0.26	0.26
10000	8	0.58	5.01	7.39	0.78	0.12	0.19	14.93	16.61	0.13	0.12
10000	16	0.38	4.97	7.17	0.60	0.06	0.10	16.15	18.30	0.06	0.06
14142	1	8.24	6.57	18.25	1.00	1.00	1.00	22.44	24.19	1.00	1.00
14142	2	4.21	6.68	12.94	0.98	0.49	0.71	19.10	20.65	0.59	0.59
14142	4	2.18	6.74	11.03	0.94	0.24	0.41	18.91	20.55	0.30	0.29
14142	8	1.18	6.75	9.98	0.87	0.12	0.23	19.60	21.59	0.14	0.14
14142	16	0.69	6.76	9.50	0.75	0.06	0.12	20.88	23.50	0.07	0.06
20000	1	15.49	8.26	28.50	1.00	1.00	1.00	35.73	38.30	1.00	1.00
20000	2	7.88	8.38	19.47	0.98	0.49	0.73	27.14	29.21	0.66	0.66
20000	4	4.04	8.49	15.07	0.96	0.24	0.47	24.18	26.22	0.37	0.37
20000	8	2.13	8.50	13.00	0.91	0.12	0.27	24.96	27.47	0.18	0.17
20000	16	1.20	8.48	12.03	0.80	0.06	0.15	26.89	30.21	0.08	0.08

sparse cost structures (e.g., $\left[0, \frac{n}{10}\right]$ and $[0, n]$), scaling efficiency would be poor, due to the fact that the forward and reverse pass steps are the main contributors in the execution time, which are limited to a single GPU. The alternating tree variant is not scalable in a multi-GPU setting, however, its primary advantage is that we can solve truly large sized problems, provided we have sufficient number of GPUs.

4.7 Conclusion

To summarize, we developed parallel versions of the classical and the alternating tree variants of the Hungarian algorithm, for solving the linear assignment problem on a single GPU, as well as multiple GPUs in a grid setting. Although, the sequential algorithm does not readily lend itself to massive parallelism like the Auction algorithm, each step of the algorithm can be parallelized on the GPU. Our main contribution is an efficient GPU-based parallel algorithm for the augmenting

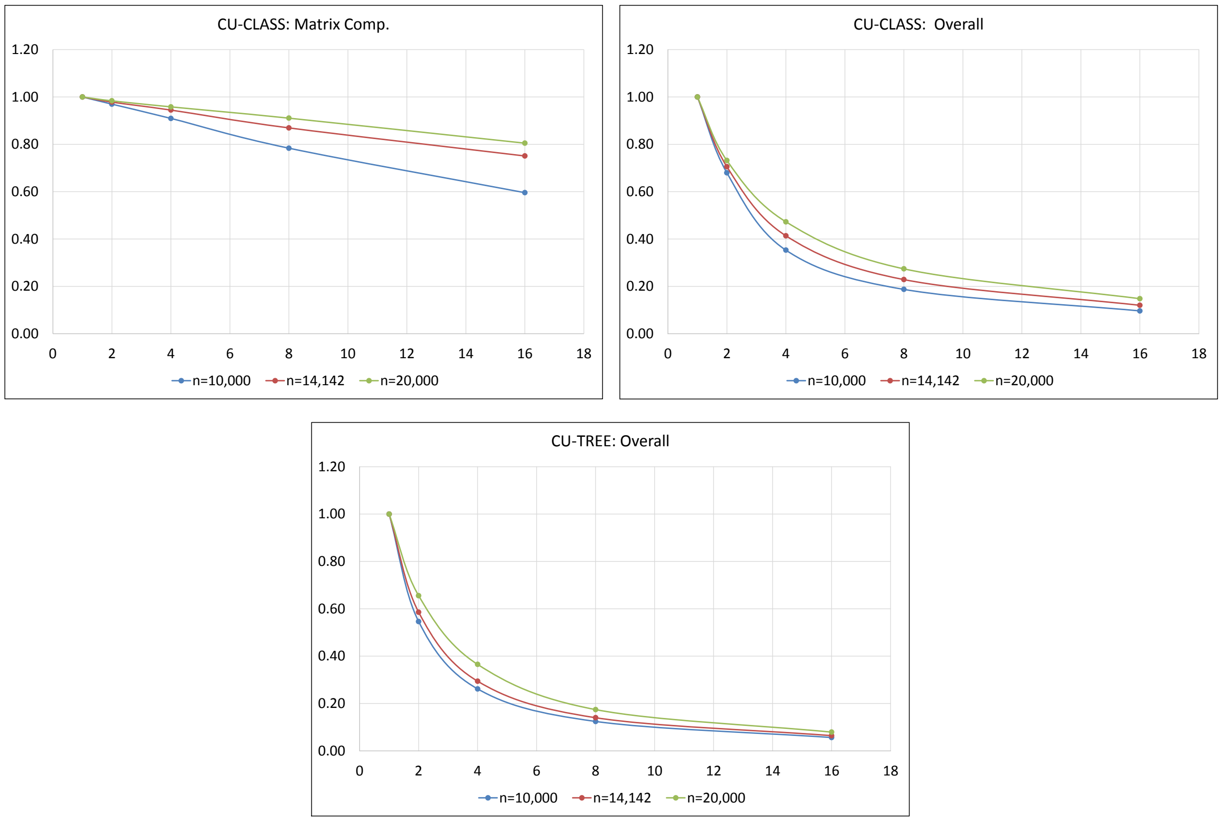


Figure 4.5: Strong scaling efficiency

path search, which happens to be the most time intensive step of the Hungarian algorithm. We showed that our algorithm(s) can find multiple vertex-disjoint paths that can be used to augment the solution during each iteration, which drastically reduces the execution time. We conducted extensive numerical tests on the algorithm(s), on small and large scale problems, which reveal that our algorithm(s) are significantly faster than the sequential and OpenMP implementations solved on a multi-core CPU. Although the OpenMP version implemented on a faster CPU with greater number of cores can potentially outperform the GPU version, such CPUs are extremely costly, making them out of reach for an average researcher. On the contrary, a simple gaming graphics card (NVIDIA GeForce GTX 970) contains 1664 CUDA cores and it can be bought for only about \$400, which makes our implementation all the more attractive. Out of the two GPU-accelerated algorithms, the alternating tree variant has one order of magnitude smaller asymptotic complexity, therefore, it is bound to outperform the classical variant at some point. Additionally, it is best suited for denser cost matrices and therefore it is an excellent choice for LAPs with non-integral costs. The multi-GPU version of the classical variant can be used to efficiently solve larger problems with dense cost matrix structure. For small problems, however, the multi-GPU version does not provide adequate scaling efficiency, because of the limitation imposed by the augmenting path search step. The alternating tree variant shows poor scaling on multi-GPU setting owing to the fact that there is significant MPI communication during the BFS iterations, however, it can be used to solve problems that are extremely large, if we have the required number of GPUs.

Our algorithm(s) can be implemented in various solution methodologies for important NP-hard problems, such as data association and graph matching, which can benefit from its large parallel speedup. We believe that our parallelization strategy can provide an elegant and cost effective way of solving these problems on a desktop computer, simply equipped with a graphics card, without having to rely on the large computational grids. The multi-GPU versions provide good alternative for solving larger problems which cannot be solved on a single GPU due to memory limitations, subject to the availability of the necessary hardware.

Chapter 5

Exact Algorithms for Large Quadratic Assignment Problems on Graphics Processing Unit Clusters

This chapter discusses efficient parallel algorithms for obtaining strong lower bounds on large instances of the Quadratic Assignment Problem (QAP). Since QAP is known to be a computationally intensive problem, it should be noted that large in this context means problem instances with up to 40 facilities and locations, which still remain unsolved to provable optimality. Our parallel architecture is comprised of both multi-core processors and Compute Unified Device Architecture (CUDA) enabled NVIDIA Graphics Processing Units (GPUs) on a computational cluster. We propose novel parallelization of the Lagrangian Dual Ascent algorithm on the GPUs, which is used for solving a QAP formulation based on Level-2 Refactorization Linearization Technique (RLT2). The Linear Assignment sub-problems (LAPs) in this procedure are solved using our parallel Hungarian algorithm. This GPU-accelerated approach can be used to obtain tight lower bounds on the QAP, which are extremely valuable in a branch-and-bound scheme for obtaining provably optimal solutions.

5.1 Introduction

The Quadratic Assignment Problem (QAP) is one of the oldest mathematical problems in the literature and it has received substantial attention from the researchers around the world. QAP was originally introduced by Koopmans and Beckmann (1957) as a mathematical model to locate indivisible economical activities (such as facilities) on a set of locations and the cost of the assignment is a function of both distance and flow. The objective is to assign each facility to a location so as to minimize a quadratic cost function. The generalized mathematical formulation for the QAP, given by Lawler (1963), can be written as follows:

$$\text{QAP: min} \quad \sum_{i=1}^{n}\sum_{p=1}^{n} b_{ip}x_{ip} + \sum_{i=1}^{n}\sum_{j=1}^{n}\sum_{p=1}^{n}\sum_{q=1}^{n} f_{ij}d_{pq}x_{ip}x_{jq}; \tag{5.1}$$

$$\text{s.t.} \sum_{p=1}^{n} x_{ip} = 1 \qquad \forall i = 1,\ldots,n; \tag{5.2}$$

$$\sum_{i=1}^{n} x_{ip} = 1 \qquad \forall p = 1,\ldots,n; \tag{5.3}$$

$$x_{ip} \in \{0,1\} \qquad \forall i,p = 1,\ldots,n. \tag{5.4}$$

The decision variable $x_{ip} = 1$, if facility i is assigned to location p and 0 otherwise. Constraints (5.2) and (5.3) enforce that each facility should be assigned to exactly one location and each location should be assigned to exactly one facility. b_{ip} is the fixed cost of assigning facility i to location p; f_{ij} is the flow between the pair of facilities i and j; and d_{pq} is the distance between the pair of locations p and q.

Despite having the same constraint set as the Linear Assignment Problem (LAP), the QAP is a strongly NP-hard problem (Sahni and Gonzalez, 1976), i.e., it cannot be solved efficiently within a guaranteed time limit. Additionally, it is difficult to find a provable ϵ-optimal solution to QAP. The quadratic nature of the objective function also adds to the solution complexity. One of the ways of solving the QAP is to convert it into a Mixed Integer Linear Program (MILP) by introducing additional variables and constraints. Different linearizations were proposed by Lawler (1963), Kaufman and Broeckx (1978), Frieze and Yadegar (1983) and Adams and Johnson (1994). Table 5.1 presents a comparison of these various linearizations in terms of number of variables and constraints. Many formulations and algorithms have been developed over the years for solving the QAP optimally or sub-optimally. For a list of references on the QAP, readers are directed to the survey papers by Burkard (2002) and Loiola et al. (2007).

Table 5.1: Linearlization models for QAP

Linearization Model	**Binary Variables**	**Continuous Variables**	**Constraints**
Lawler (1963)	$O(n^4)$	–	$O(n^4)$
Kaufman and Broeckx (1978)	$O(n^2)$	$O(n^2)$	$O(n^2)$
Frieze and Yadegar (1983)	$O(n^2)$	$O(n^4)$	$O(n^4)$
Adams and Johnson (1994) RLT1	$O(n^2)$	$O(n^4)$	$O(n^4)$
Adams et al. (2007) RLT2	$O(n^2)$	$O(n^6)$	$O(n^6)$
Hahn et al. (2012) RLT3	$O(n^2)$	$O(n^8)$	$O(n^8)$

The main advantage of formulating the QAP as an MILP is that we can relax the integrality restrictions on the variables and solve the resulting linear program. The objective function value obtained from this LP solution can be used as a lower bound in the exact solution methods such as the branch-and-bound. The most promising formulation was obtained by Adams and Johnson

(1994) by applying level-1 refactorization and linearization technique (RLT1) to the QAP. This was considered to be one of the best linearizations at the time, because it yielded strong LP relaxation bound. Adams and Johnson (1994) developed an iterative algorithm based on the Lagrangian dual ascent to obtain a lower bound for the QAP. Later Hahn and Grant (1998) developed an augmented dual ascent scheme (with simulated annealing), which yielded a lower bound which was close to the LP relaxation bound. This linearization technique was extended to RLT2 by Adams et al. (2007), which contains $O(n^6)$ variables and constraints; and RLT3 by Hahn et al. (2012), which contains $O(n^8)$ variables and constraints. These two formulations provide much stronger lower bounds as compared to RLT1, and for many problem instances they are able to match the optimal objective value of the QAP. However, it is extremely difficult to solve these linearization models using primal methods, because of the curse of dimensionality. Ramakrishnan et al. (2002) used Approximate Dual Projective (ADP) method to solve the LP relaxation of the RLT2 formulation of Ramachandran and Pekny (1996), which was limited to the problems with size $n = 12$. Adams et al. (2007) and Hahn et al. (2012) developed a dual ascent based algorithms to find strong lower bounds on RLT2 and RLT3 respectively, and used them to solve QAPs with $n \leq 30$. As observed by Hahn et al. (2012), LP relaxations of RLT2 and RLT3 provide strong lower bounds on the QAP, with RLT3 being the strongest. However, due to the large number of variables and constraints in RLT3, tremendous amount of memory is required to handle even the small/medium sized QAP instances. In comparison, the RLT2 formulation has modest memory requirements and it provides sufficiently strong lower bounds.

For obtaining a lower bound on the QAP using RLT2 dual ascent, we need to solve $O(n^4)$ LAPs and update $O(n^6)$ Lagrangian multipliers during each iteration, which can become computationally time intensive. However, as described in Section 5.2, this algorithm can benefit from parallization on an appropriate parallel computing architecture. In recent years, there have been significant advancements in the graphics processing hardware. Since graphics processing tasks generally require high data parallelism, the Graphics Processing Units (GPUs) are built as compute-intensive, massively parallel machines, which provide a cost-effective solution for high performance computing applications. Recently Gonçalves et al. (2015) developed a GPU-based dual ascent algorithm for RLT2, which shows significant parallel speed up as compared to the sequential algorithm. Although it is very promising, there algorithm is limited to a single GPU and not scalable for large problems. In this work, we are proposing a distributed version of the RLT2 dual ascent algorithm and a parallel branch-and bound algorithm, specifically designed for the CUDA enabled NVIDIA GPUs, for solving large instances of the QAP to optimality. These algorithms make use of the hybrid MPI+CUDA architecture, on the GPU cluster offered by the Blue Waters Supercomputing facility at the University of Illinois at Urbana-Champaign. This research is radical because, to the best of our knowledge, this is the first scalable GPU-based algorithm that can be used for solving large QAPs in a grid setting.

The rest of the chapter is organized as follows. Section 5.2 describes the RLT2 formulation and the concepts of the sequential dual ascent algorithm. Sections 5.3 and 5.4 describe the various stages of our parallel algorithm, and an implementation on the multi-GPU architecture. Section 5.5 contains the experimental results on the instances from the QAPLIB. Section 5.6 contains an application of our parallel algorithm to the facility placement problem. Finally, the chapter is concluded in Section 5.7 with a summary and some directions for future research.

5.2 RLT2 Linearization and Dual Ascent

5.2.1 RLT2 Linearization for the QAP

As explained by Adams et al. (2007), the refactorization-linearization technique can be applied to the QAP formulation (5.1)-(5.4), to obtain an instance of MILP. Henceforth, it is assumed that the indices i, j, p, q, etc., go from 1 to n unless otherwise stated. Initially, in the "refactorization" step, the constraints (5.2) and (5.3) are multiplied by a variable $x_{ip}, \forall i, p$. After removing the invalid variables of the form $x_{ip} \cdot x_{iq}$ and omitting the trivial constraints $x_{ip} \cdot x_{ip} = x_{ip}$ we obtain $2n^2(n-1)$ new constraints of the form $\sum_{j \neq i} x_{jq} \cdot x_{ip} = x_{ip},\ \forall i, p, q$; and $\sum_{q \neq p} x_{jq} \cdot x_{ip} = x_{ip},\ \forall i, j, p$. Then, in the "linearization" step, the product $x_{ip} \cdot x_{jq}$ is replaced by a new variable y_{ijpq} with cost coefficient $C_{ijpq} = f_{ij} \cdot d_{pq}$; and a set of $\frac{n^2(n-1)^2}{2}$ constraints of the form $y_{ijpq} = y_{jiqp}$ are introduced to signify the symmetry of multiplication. The resulting formulation is called RLT1 by Adams et al. (2007), which is depicted below:

$$\text{RLT1: } \min \sum_{i} \sum_{p} b_{ip} x_{ip} + \sum_{i} \sum_{j \neq i} \sum_{p} \sum_{q \neq p} C_{ijpq} y_{ijpq}; \tag{5.5}$$

$$\text{s.t. } (5.2) - (5.4)$$

$$\sum_{q \neq p} y_{ijpq} = x_{ip}, \qquad \forall (i \neq j, p); \tag{5.6}$$

$$\sum_{j \neq i} y_{ijpq} = x_{ip}, \qquad \forall (i, p \neq q); \tag{5.7}$$

$$y_{ijpq} = y_{jiqp}, \qquad \forall (i < j, p \neq q); \tag{5.8}$$

$$y_{ijpq} \geq 0, \qquad \forall (i \neq j, p \neq q). \tag{5.9}$$

Result 7. *The RLT1 formulation is equivalent to the QAP, i.e., a feasible solution to RLT1 is also feasible to the QAP with the same objective function value (Adams and Johnson, 1994).*

Now the refactorization-linearization technique is applied on the RLT1 formulation. During the refactorization step, the constraints (5.6)-(5.9) are multiplied by variables $x_{ip}, \forall i, p$. The product $y_{jkqr} \cdot x_{ip}$ is replaced by a new variable z_{ijkpqr}, with a cost coefficient of D_{ijkpqr}. The resulting RLT2 formulation is depicted below:

$$\text{RLT2: } \min \sum_{i} \sum_{p} b_{ip} x_{ip} + \sum_{i} \sum_{j \neq i} \sum_{p} \sum_{q \neq p} C_{ijpq} y_{ijpq} + \sum_{i} \sum_{j \neq i} \sum_{k \neq i,j} \sum_{p} \sum_{q \neq p} \sum_{r \neq p,q} D_{ijkpqr} z_{ijkpqr}; \tag{5.10}$$

$$\begin{aligned}
\text{s.t. } & (5.2) - (5.4); && \\
& (5.6) - (5.9); && \\
& \sum_{r \neq p,q} z_{ijkpqr} = y_{ijpq}, && \forall (i \neq j \neq k, p \neq q); \quad (5.11) \\
& \sum_{k \neq i,j} z_{ijkpqr} = y_{ijpq}, && \forall (i \neq j, p \neq q \neq r); \quad (5.12) \\
& z_{ijkpqr} = z_{ikjprq} = z_{jikqpr} = z_{jkiqrp} = z_{kijrpq} = z_{kjirqp}, && \forall (i < j < k, p \neq q \neq r); \quad (5.13) \\
& z_{ijkpqr} \geq 0, && \forall (i \neq j \neq k, p \neq q \neq r). \quad (5.14)
\end{aligned}$$

Result 8. *The RLT2 formulation is equivalent to the QAP, i.e., a feasible solution to RLT2 is also feasible to the QAP with the same objective function value (Adams et al., 2007).*

Lemma 11. *In the RLT2 formulation, constraint set (5.8) is redundant and it is subsumed by the constraint set (5.13).*

Proof. From the constraint set (5.13), for some $(i, j, p, q, r) : (i < j) \wedge (p \neq q \neq r)$, we can write:

$$\begin{aligned}
& z_{ijkpqr} = z_{jikqpr}, \forall k : (k \neq i) \wedge (k \neq j); \\
\Longrightarrow & \sum_{k \neq i,j} z_{ijkpqr} = \sum_{k \neq i,j} z_{jikqpr}; \\
\Longrightarrow & \; y_{ijpq} = y_{jiqp};
\end{aligned}$$

which is nothing but one of the constraints from (5.8). The lemma follows.

□

The main advantage of using RLT2 formulation is that its LP relaxation (LPRLT2) obtained by relaxing the binary restrictions on x_{ip} yields much stronger lower bounds than any other linearization from the literature. However, since this formulation has a large number of variables and constraints, primal methods are likely to fail for large QAPs (as observed by Ramakrishnan et al. (2002)). Adams and Johnson (1994) and Adams et al. (2007) addressed this problem by developing a solution procedure based on Lagrangian dual ascent. In the next sections we will briefly discuss some concepts about Lagrangian duality and then explain the Lagrangian dual ascent algorithm for RLT2.

5.2.2 Lagrangian Duality

Duality is an important concept in the theory of optimization. The Primal problem (P) and its Dual (D) share a very special relationship, known as the "weak duality." If $\nu(\cdot)$ represents the objective function of a problem, then the weak duality states that $\nu(D) \leq \nu(P)$ for minimization problem. Many algorithms make use of this relationship, in cases where one of these problems is easier to solve than its counterpart. The basis of these constructive dual techniques is the Lagrangian relaxation.

Let us consider the following simple optimization problem:

$$\text{P: } \min \quad \mathbf{cx}; \quad \text{s.t.} \quad \mathbf{Ax} \geq \mathbf{b}; \quad \mathbf{x} \in X. \tag{5.15}$$

Here, $\mathbf{Ax} \geq \mathbf{b}$ represent the complicating constraints and $\mathbf{x} \in X$ represent simple constraints. Then we can relax the complicating constraints and add them to the objective function using non-negative Lagrange multipliers $\mathbf{u}$, which gives rise to the following Lagrangian relaxation:

$$\text{LRP}(\mathbf{u})\text{: } \min \quad \mathbf{cx} + \mathbf{u}(\mathbf{b} - \mathbf{Ax}); \quad \text{s.t.} \quad \mathbf{x} \in X. \tag{5.16}$$

For any $\mathbf{u} \geq 0$, $\nu(\text{LRP}(\mathbf{u}))$ provides a lower bound on $\nu(P)$, i.e., $\nu(\text{LRP}(\mathbf{u})) \leq \nu(P)$. To find the best possible lower bound, we solve the Lagrangian dual problem LD($\mathbf{u}$): $\max_{\mathbf{u} \geq 0} \nu(\text{LRP}(\mathbf{u}))$. Hence, the primary goal in these solution procedures is to systematically search for the Lagrange multipliers which maximize the objective function value of the Lagrangian dual problem. The following two solution procedures are most commonly used for obtaining these dual multipliers.

Subgradient Lagrangian Search. The subgradient search method operates on two important observations: (1) $\nu(\text{LRP}(\mathbf{u}))$ is a piecewise-linear concave function of $\mathbf{u}$; and (2) At some point $\hat{\mathbf{u}} \geq 0$, if $\hat{\mathbf{x}} \in X$ is a solution to LRP($\hat{\mathbf{u}}$), then $(\mathbf{b} - \mathbf{A}\hat{\mathbf{x}})$ represents a valid subgradient of the function $\nu(\text{LRP}(\hat{\mathbf{u}}))$. The subgradient search procedure is very similar to the standard gradient ascent procedure, where we advance along the (sub)gradients of the objective function until we reach some solution that is no longer improving. At that point, we calculate the new (sub)gradients and continue. The only disadvantage of using subgradients instead of the gradient is that it is difficult to characterize an accurate step-size which is valid for all the active subgradients. Therefore, taking an arbitrary step along the subgradients might worsen the objective function from time to time. However, for specific step-size rules, it is proved that the procedure converges to the optimal solution asymptotically.

Lagrangian Dual Ascent. Instead of using the subgradients in a naive fashion, they can be used to precisely figure out both the ascent direction and the step-size that gives us the "best" possible improvement in the objective function $\nu(\text{LRP}(\mathbf{u}))$. This is the crux of the dual ascent procedure. During each iteration of the dual ascent procedure, an optimization problem is solved to find a direction $\mathbf{d}$ for some dual solution $\hat{\mathbf{u}}$, which creates a positive inner product with every subgradient of $\nu(\text{LRP}(\hat{\mathbf{u}}))$, i.e., $\mathbf{d}(\mathbf{b} - \mathbf{Ax}) > 0, \forall \mathbf{x} \in X(\hat{\mathbf{u}})$. If no such direction is found, then the solution $\hat{\mathbf{u}}$ and corresponding $\hat{\mathbf{x}}$ is an optimal solution. Otherwise, the "best" step-size is established which gives the maximum improvement in the objective function along $\mathbf{d}$ to find a new dual solution. The most difficult part of the dual ascent algorithm is to find the step-size λ that will provide a guaranteed ascent, while maintaining the feasibility of all the previous primal solutions, and more often than not, finding the optimal step-size is an NP-hard problem. However, the salient feature of the Lagrangian dual of RLT2 linearization is that the improving direction and step-size can be found without having to solve any optimization problem. This can be achieved by doing simple sensitivity analysis and maintaining the complementary slackness for the nonbasic $\mathbf{x}$, $\mathbf{y}$ and $\mathbf{z}$ variables in the corresponding LAPs. In the next section, we will discuss the features of the RLT2 linearization and its Lagrangian dual.

5.2.3 Sequential RLT2-DA Algorithm

Let us consider the LP relaxation of the RLT2 formulation. Initially, the constraints (5.13) are relaxed and added to the objective function using the Lagrange multipliers $\mathbf{v} = \langle v_{ijkpqr} \rangle$, to obtain the Lagrangian relaxation LRLT2. Let $\alpha, \beta, \gamma, \delta, \xi, \psi$ represent the dual variables corresponding to the constraints $(5.2), (5.3), (5.6), (5.7), (5.11), (5.12)$ respectively. Then for some fixed $\hat{\mathbf{v}}$, the Lagrangian relaxation LRLT2($\hat{\mathbf{v}}$) and its dual DLRLT2($\hat{\mathbf{v}}$) can be written as follows.

$$\begin{aligned}
\text{LRLT2}(\hat{\mathbf{v}})\text{:}\ \min & \sum_i \sum_p b_{ip} x_{ip} + \sum_i \sum_{j \neq i} \sum_p \sum_{q \neq p} C_{ijpq} y_{ijpq} \\
& + \sum_i \sum_{j \neq i} \sum_{k \neq i,j} \sum_p \sum_{q \neq p} \sum_{r \neq p,q} (D_{ijkpqr} - \hat{v}_{ijkpqr}) z_{ijkpqr}; && (5.17) \\
\text{s.t. } & (5.2) - (5.3); \\
& (5.6) - (5.7); \\
& (5.11) - (5.12); \\
& x_{ip} \geq 0; \quad y_{ijpq} \geq 0; \quad z_{ijkpqr} \geq 0. && (5.18)
\end{aligned}$$

$$\begin{aligned}
\text{DLRLT2}(\hat{\mathbf{v}})\text{:}\ \max & \sum_i \alpha_i + \sum_p \beta_p; && && (5.19) \\
\text{s.t. } & \alpha_i + \beta_p - \sum_{j \neq i} \gamma_{ijp} - \sum_{q \neq p} \delta_{ipq} \leq b_{ip}, && \forall i, p; && (5.20) \\
& \gamma_{ijp} + \delta_{ipq} - \sum_{k \neq i,j} \xi_{ijkpq} - \sum_{r \neq p,q} \psi_{ijpqr} \leq C_{ijpq}, && \forall (i \neq j, p \neq q); && (5.21) \\
& \xi_{ijkpq} + \psi_{ijpqr} \leq D_{ijkpqr} - \hat{v}_{ijkpqr}, && \forall (i \neq j \neq k, p \neq q \neq r); && (5.22) \\
& \alpha_i, \beta_p, \gamma_{ijp}, \delta_{ipq}, \xi_{ijkpq}, \psi_{ijpqr} \sim \text{unrestricted}, && \forall (i \neq j \neq k, p \neq q \neq r). && (5.23)
\end{aligned}$$

LAP Solution. The problem DLRLT2($\hat{\mathbf{v}}$) can be solved using the decomposition principle explained by Adams et al. (2007). To maximize the dual objective function (5.19) with respect to constraints (5.20), we need large values of α and β, for which the term $\sum_{j \neq i} \gamma_{ijp} + \sum_{q \neq p} \delta_{ipq}$ needs to be maximized subject to constraints (5.21). This requires large values of γ and δ, for which, the term $\sum_{k \neq i,j} \xi_{ijkpq} + \sum_{r \neq p,q} \psi_{ijpqr}$ needs to be maximized with respect to constraints (5.22). Thus we have a three stage problem, as seen in Fig. 5.1.

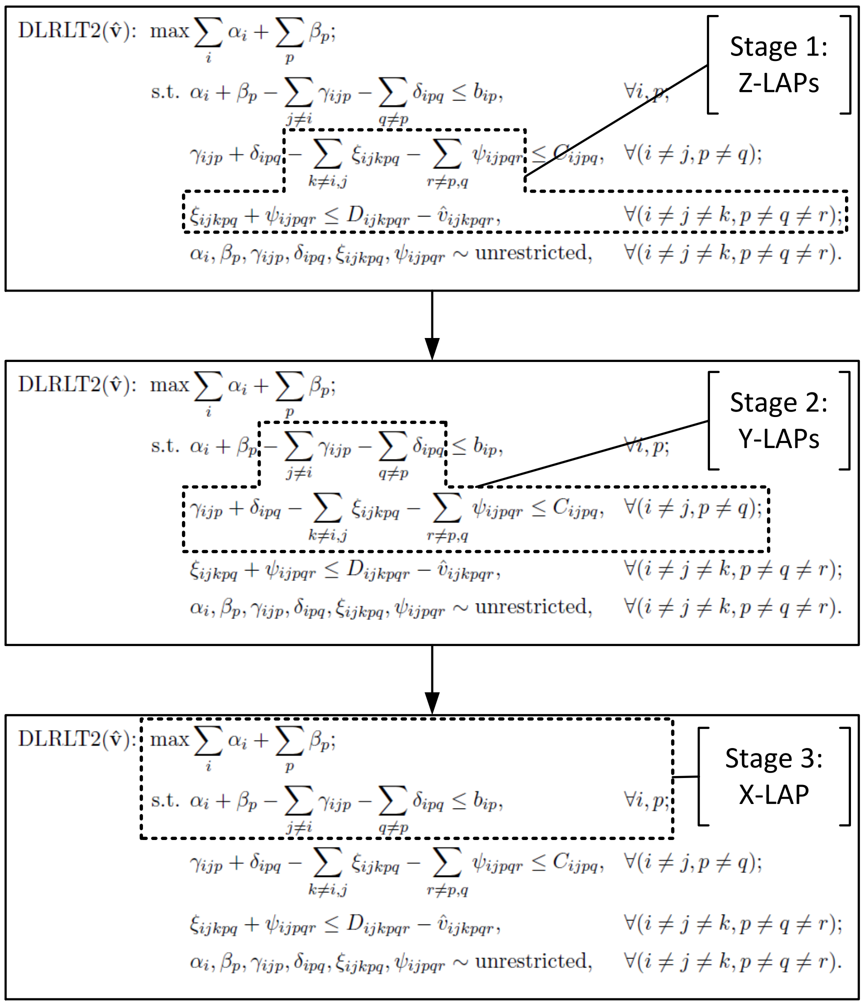

DLRLT2($\hat{\mathbf{v}}$): $\max \sum_i \alpha_i + \sum_p \beta_p$;
s.t. $\alpha_i + \beta_p - \sum_{j\neq i} \gamma_{ijp} - \sum_{q\neq p} \delta_{ipq} \leq b_{ip}$, $\forall i, p$;
$\gamma_{ijp} + \delta_{ipq} - \sum_{k\neq i,j} \xi_{ijkpq} - \sum_{r\neq p,q} \psi_{ijpqr} \leq C_{ijpq}$, $\forall (i\neq j, p\neq q)$;
$\xi_{ijkpq} + \psi_{ijpqr} \leq D_{ijkpqr} - \hat{v}_{ijkpqr}$, $\forall (i\neq j\neq k, p\neq q\neq r)$;
$\alpha_i, \beta_p, \gamma_{ijp}, \delta_{ipq}, \xi_{ijkpq}, \psi_{ijpqr} \sim$ unrestricted, $\forall (i\neq j\neq k, p\neq q\neq r)$.
Stage 1: Z-LAPs
Stage 2: Y-LAPs
Stage 3: X-LAP

Figure 5.1: Three stage solution of LRLT2($\hat{\mathbf{v}}$)

In the first stage, for each (i, j, p, q), with $i \neq j$ and $p \neq q$, we need to solve the problems:

$$\Theta_{ijpq}(\hat{\mathbf{v}}) = \max \left\{ \sum_{k\neq i,j} \xi_{ijkpq} + \sum_{r\neq p,q} \psi_{ijpqr} \,\middle|\, \xi_{ijkpq} + \psi_{ijpqr} \leq \hat{D}_{ijkpqr}, \forall (k \neq i, j; r \neq p, q) \right\}, \tag{5.24}$$

which are nothing but $n^2(n-1)^2$ Z-LAPs in their dual form (with modified cost coefficients). In the second stage, for each (i, p), we need to solve the problems:

$$\Delta_{ip}(\hat{\mathbf{v}}) = \max \left\{ \sum_{j\neq i} \gamma_{ijp} + \sum_{q\neq p} \delta_{ipq} \,\middle|\, \gamma_{ijp} + \delta_{ipq} \leq C_{ijpq} + \Theta_{ijpq}(\hat{\mathbf{v}}), \forall (j \neq i; q \neq p) \right\}, \tag{5.25}$$

which are nothing but n^2 Y-LAPs in their dual form (with modified cost coefficients). In the final stage, we need to solve a single X-LAP (with modified cost coefficients):

$$\nu(\text{LRLT2}(\hat{\mathbf{v}})) = \nu(\text{DLRLT2}(\hat{\mathbf{v}})) = \max\left\{\sum_i \alpha_i + \sum_p \beta_p \middle| \alpha_i + \beta_p \le b_{ip} + \Delta_{ip}(\hat{\mathbf{v}}), \forall (i,p)\right\}, \tag{5.26}$$

which gives us the required lower bound on RLT2. We can see that there are $O(n^4)$ LAPs and the number of cost coefficients in each LAP is $O(n^2)$. The worst case complexity of any primal-dual LAP algorithm for an input matrix with n^2 cost coefficients, is $O(n^3)$. Therefore, the overall solution complexity for solving DLRLT2($\hat{\mathbf{v}}$) is $O(n^7)$.

Dual Ascent. The Lagrangian dual problem for LRLT2 is to find the best set of multipliers $\mathbf{v}^*$, so as to maximize the objective function value ν(LRLT2), i.e.,

$$\text{LDRLT2: } \max_{\mathbf{v}} \{\nu(\text{LRLT2}(\mathbf{v}))\}. \tag{5.27}$$

Since LRLT2($\mathbf{v}$) exhibits integrality property, due to the theorem by Geoffrion (1974), the objective function value ν(LDRLT2) cannot exceed the linear programming relaxation bound ν(LPRLT2), obtained by relaxing the binary restrictions (5.4). Therefore, we can assert the following inequality:

$$\nu(\text{LDRLT2}) \le \nu(\text{LPRLT2}) \le \nu(\text{RLT2}) = \nu(\text{QAP}). \tag{5.28}$$

To solve LDRLT2, one could employ a standard dual ascent algorithm. However, for LDRLT2, finding an ascent direction and a step-size can be done relatively easily, without having to solve any optimization problem. To this end, we will now describe the principle behind the Lagrangian dual ascent for LDRLT2.

1. Let $\pi(\cdot)$ denote the reduced cost (or dual slack) of an LAP variable. Then, for some variable z_{ijkpqr} in an optimal LAP solution,

$$z_{ijkpqr} = 1 \implies \pi(z_{ijkpqr}) = 0; \text{ and } z_{ijkpqr} = 0 \implies \pi(z_{ijkpqr}) \ge 0. \tag{5.29}$$

2. From Equation (5.13), we know that for any $(i,j,k,p,q,r) : i < j < k, p \ne q \ne r$, the variable z_{ijkpqr} is one of the six "complementary" variables appearing in that particular constraint, and in an optimal QAP solution, the values of all the six variables should be the same.

3. If some $z_{ijkpqr} = 0$ and one of its complementary variables $z_{jikqpr} = 1$, then for the constraint $z_{ijkpqr} = z_{jikqpr}$, the direction $(1, -1)$ provides a natural direction of ascent for $(\hat{v}_{ijkpqr}, \hat{v}_{jikqpr})$, because it is a valid subgradient of LRLT2. To obtain a new dual solution, a step may be taken along this direction, i.e., $\hat{v}_{ijkpqr}$ may be increased (i.e., D_{ijkpqr} may be decreased) and $\hat{v}_{jikqpr}$ may be decreased (i.e., D_{ijkpqr} may be increased), using a valid step-size.

4. While determining the step-size, the most important criterion is that the feasibility of the current dual variables $\alpha, \beta, \gamma, \delta, \xi, \psi$ must be maintained. According the constraint (5.22), infeasibility is incurred in the dual space if $\pi(z_{ijkpqr}) = D_{ijkpqr} - \hat{v}_{ijkpqr} - \xi_{ijkpq} - \psi_{ijpqr} < 0$. This means that D_{ijkpqr} is allowed to decrease by at most $\pi(z_{ijkpqr})$, and consequently, the complementary cost coefficient D_{jikqpr} can be increased by the same amount. Since z_{jikqpr} is basic, there is a good chance that this adjustment will increase ν(LDRLT2) by some non-negative value, and therefore, this is a "strong" direction of ascent.

5. For some other pair of variables, if $z_{ijkpqr} = z_{ikjprq} = 0$, then the direction $(1, -1)$ is also a valid direction, i.e., the cost coefficient D_{ijkpqr} can be decreased by at most $\pi(z_{ijkpqr})$ and D_{ikjprq} can be increased by the same amount. Since both variables are non-basic, there will be no change in ν(LDRLT2). This direction is a "weak" direction of ascent.

6. In an "optimal" dual ascent scheme, we would need to find ascent directions which will be "strong" for every pairwise constraint from Equation (5.13), and finding such direction would require significant computational effort. However, we can easily find a direction that is "strong" only for a subset of pairwise constraints, which may provide a non-negative increase in ν(LDRLT2). In other words, we can select a non-basic variable z_{ijkpqr}, decrease its cost coefficient by some amount $0 < \epsilon \leq \pi(z_{ijkpqr})$ and increase the cost coefficients of the five complementary variables by some fraction of ϵ. If some of the directions happen to be "strong," then the objective function ν(LDRLT2) will experience non-negative increase, otherwise it will stay the same. This is the crux of the dual ascent procedure. Mathematically, we adjust the dual multipliers using the rule:

$$\begin{aligned}
v_{ijkpqr} &\leftarrow v_{ijkpqr} + \kappa^z \pi(z_{ijkpqr}); \\
v_{ikjprq} &\leftarrow v_{ikjprq} - \phi_1^z \kappa^z \pi(z_{ijkpqr}); \\
v_{jikqpr} &\leftarrow v_{jikqpr} - \phi_2^z \kappa^z \pi(z_{ijkpqr}); \\
v_{jkiqrp} &\leftarrow v_{jkiqrp} - \phi_3^z \kappa^z \pi(z_{ijkpqr}); \\
v_{kijrpq} &\leftarrow v_{kijrpq} - \phi_4^z \kappa^z \pi(z_{ijkpqr}); \\
v_{kjirqp} &\leftarrow v_{kjirqp} - \phi_5^z \kappa^z \pi(z_{ijkpqr}).
\end{aligned} \tag{5.30}$$

Here, $0 \leq \kappa^z \leq 1$, $0 \leq \phi_\cdot^z \leq 1$, and $\phi_1^z + \phi_2^z + \phi_3^z + \phi_4^z + \phi_5^z = 1$. We will refer to this as "Type 1 ascent rule."

7. Now, let us consider the constraint (5.21). After applying Type 1 rule and solving the corresponding Z-LAP(s); for some (i, j, p, q), it is possible that $\gamma_{ijp} + \delta_{ipq} - \Theta_{ijpq} < C_{ijpq}$, i.e., $\pi(y_{ijpq}) > 0$. In this case, Θ_{ijpq} can be decreased by $\pi(y_{ijpq})$, by decreasing the cost coefficients $D_{ijkpqr}, \forall k, r$ by an amount $\frac{\pi(y_{ijpq})}{(n-2)}$. This allows us to increase the cost coefficients of the complementary variables, providing the objective functions of the corresponding LAPs

a chance to grow. Mathematically, we adjust the dual multipliers using the rule:

$$
\begin{aligned}
v_{ijkpqr} &\leftarrow v_{ijkpqr} + \kappa^{y}\frac{\pi(y_{ijpq})}{(n-2)};\\
v_{ikjprq} &\leftarrow v_{ikjprq} - \phi^{y}_{1}\kappa^{y}\frac{\pi(y_{ijpq})}{(n-2)};\\
v_{jikqpr} &\leftarrow v_{jikqpr} - \phi^{y}_{2}\kappa^{y}\frac{\pi(y_{ijpq})}{(n-2)};\\
v_{jkiqrp} &\leftarrow v_{jkiqrp} - \phi^{y}_{3}\kappa^{y}\frac{\pi(y_{ijpq})}{(n-2)};\\
v_{kijrpq} &\leftarrow v_{kijrpq} - \phi^{y}_{4}\kappa^{y}\frac{\pi(y_{ijpq})}{(n-2)};\\
v_{kjirqp} &\leftarrow v_{kjirqp} - \phi^{y}_{5}\kappa^{y}\frac{\pi(y_{ijpq})}{(n-2)}.
\end{aligned}
\tag{5.31}
$$

Here, $0 \leq \kappa^{y} \leq 1$, $0 \leq \phi^{y}_{\cdot} \leq 1$, and $\phi^{y}_{1} + \phi^{y}_{2} + \phi^{y}_{3} + \phi^{y}_{4} + \phi^{y}_{5} = 1$. We will refer to this as "Type 2 ascent rule."

8. Finally, we can use a similar rule for constraint (5.20), and for some (i, p), if $\pi(x_{ip}) > 0$, we can decrease the cost coefficients $C_{ijpq}, \forall j, q$, by an amount of $\frac{\pi(x_{ip})}{(n-1)}$. This is equivalent to decreasing the cost coefficients $D_{ijkpqr}, \forall j, k, q, r$ by an amount $\frac{\pi(x_{ip})}{(n-1)(n-2)}$. Consequently, we can increase the cost coefficients of the complementary variables, potentially improving the objective function value of the corresponding LAPs. Mathematically, we adjust the dual multipliers using the rule:

$$
\begin{aligned}
v_{ijkpqr} &\leftarrow v_{ijkpqr} + \kappa^{x}\frac{\pi(x_{ip})}{(n-1)(n-2)};\\
v_{ikjprq} &\leftarrow v_{ikjprq} - \phi^{x}_{1}\kappa^{x}\frac{\pi(x_{ip})}{(n-1)(n-2)};\\
v_{jikqpr} &\leftarrow v_{jikqpr} - \phi^{x}_{2}\kappa^{x}\frac{\pi(x_{ip})}{(n-1)(n-2)};\\
v_{jkiqrp} &\leftarrow v_{jkiqrp} - \phi^{x}_{3}\kappa^{x}\frac{\pi(x_{ip})}{(n-1)(n-2)};\\
v_{kijrpq} &\leftarrow v_{kijrpq} - \phi^{x}_{4}\kappa^{x}\frac{\pi(x_{ip})}{(n-1)(n-2)};\\
v_{kjirqp} &\leftarrow v_{kjirqp} - \phi^{x}_{5}\kappa^{x}\frac{\pi(x_{ip})}{(n-1)(n-2)}.
\end{aligned}
\tag{5.32}
$$

Here, $0 \leq \kappa^{x} \leq 1$, $0 \leq \phi^{x}_{\cdot} \leq 1$, and $\phi^{x}_{1} + \phi^{x}_{2} + \phi^{x}_{3} + \phi^{x}_{4} + \phi^{x}_{5} = 1$. We will refer to this as "Type 3 ascent rule."

9. We can also implement a "Type 4 ascent rule," in which we can generate two fractions $0 \leq \kappa^{lb}_{i'} \leq 1$ and $0 \leq \kappa^{lb}_{p'} \leq 1$ such that $(\kappa^{lb}_{i'} + \kappa^{lb}_{p'}) \leq 1$. Then we decrease the current lower bound ν(LDRLT2) by the fraction $(\kappa^{lb}_{i'} + \kappa^{lb}_{p'})$, which is equivalent to decreasing the cost coeffi-

cients $b_{i'p}, \forall p$ by $\frac{\kappa^{lb}_{i'}\nu(\text{LDRLT2})}{n}$ and cost coefficients $b_{ip'}, \forall i$ by $\frac{\kappa^{lb}_{p'}\nu(\text{LDRLT2})}{n}$. This is equivalent to decreasing the corresponding cost coefficients $D_{i'jkpqr}, \forall j, k, p, q, r$ by an amount $\frac{\kappa^{lb}_{i'}\nu(\text{LDRLT2})}{n(n-1)(n-2)}$; and $D_{ijkp'qr}, \forall i, j, k, q, r$ by an amount $\frac{\kappa^{lb}_{p'}\nu(\text{LDRLT2})}{n(n-1)(n-2)}$. Consequently, we can increase the cost coefficients of the complementary variables, potentially improving the objective function values of the corresponding LAPs. This step deteriorates the current lower bound, however, the resulting redistribution provides a much greater increase in ν(LDRLT2). This step can be implemented in the same spirit as the Simulated Annealing (SA) approach with a specific temperature schedule. Hahn and Grant (1998) reported stronger lower bounds for SA based dual ascent for RLT1, as compared to those of the naive dual ascent of Adams and Johnson (1994). Although it was not mentioned explicitly, we suspect that this approach was also used in dual ascent for RLT2 by Adams et al. (2007). In Section 5.5, we compare the lower bounds for various problems, with and without SA.

10. The overall step-size rule for Lagrangian dual ascent is a combination of the four rules discussed above. The solution complexity of the dual ascent phase is $O(n^6)$, which is same as the upper bound on the number of cost coefficients.

Procedure RLT2-DA. Once the dual multipliers are updated, the LAPs need to be re-solved to obtain an improved ν(LDRLT2), which is also a lower bound on the QAP. Thus the RLT2-DA procedure iterates between the LAP solution phase and the dual ascent phase, until a specified optimality gap has been achieved; a specified iteration limit has been reached; or a feasible solution to the QAP has been found. The steps of RLT2-DA procedure are depicted in Algorithm 15.

Feasibility Check. To check whether the primal-dual feasibility has been achieved or not, the complementary slackness principle can be used. After solving the X-LAP and obtaining a primal solution $\mathbf{x}$, we construct feasible $\mathbf{y}$ and $\mathbf{z}$ vectors; and check whether the dual slack values $\boldsymbol{\pi}(\mathbf{y})$ and $\boldsymbol{\pi}(\mathbf{z})$ corresponding to this primal solution are compliant with Equation (5.29). If this is true, then a feasible solution has been found, which also happens to be optimal to the QAP. Otherwise we continue to update the dual multipliers and re-solve LRLT2.

Algorithm Correctness. We will now prove that the RLT2-DA provides us with a sequence of non-decreasing lower bounds on the QAP. This result has been adapted from the result by Adams and Johnson (1994).

Theorem 8. *Given the input parameters $0 \le \kappa \le 1$, $0 \le \phi \le 1$, and $\sum \phi = 1$, the RLT2-DA provides a non-decreasing sequence of lower bounds.*

Proof. Let us consider LRLT2 at some iterations m and $m+1$, with the corresponding dual multipliers $\mathbf{v}^m$ and $\mathbf{v}^{m+1}$. To prove the theorem we need to show that $\nu(\text{LRLT2}(\mathbf{v}^{m+1})) \ge \nu(\text{LRLT2}(\mathbf{v}^m))$. Consider the following dual of LRLT2($\mathbf{v}^{m+1}$). Note that we have not shown the conditions on the

Algorithm 15: RLT2-DA.

```
1. Initialization:
    (a) Initialize $m \leftarrow 0$, $\mathbf{v}^m \leftarrow \mathbf{0}$, $\bar{\nu}(\text{LDRLT2}) \leftarrow -\infty$, and GAP $\leftarrow \infty$.
    (b) Initialize $\mathbf{b}'$, $\mathbf{C}'$ and $\mathbf{D}'$.
2. Termination: Stop if $m >$ ITN_LIM or GAP $<$ MIN_GAP or Feasibility check $= \textit{true}$.
3. Z-LAP solve:
    (a) Update $D'_{ijkpqr} \leftarrow D'_{ijkpqr} - v^m_{ijkpqr}, \forall (i \neq j \neq k, p \neq q \neq r)$
    (b) Solve $n^2(n-2)^2$ Z-LAPs of size $(n-2) \times (n-2)$ and cost coefficients $\mathbf{D}'$.
    (c) Let $\Theta_{ijpq}(\mathbf{v}^m) \leftarrow \nu(\text{Z-LAP}(i,j,p,q)),\ \forall (i \neq j, p \neq q)$.
4. Y-LAP solve:
    (a) Update $C'_{ijpq} \leftarrow C_{ijpq} + \Theta_{ijpq}(\mathbf{v}^m), \forall (i \neq j, p \neq q)$.
    (b) Solve $n^2$ Y-LAPs of size $(n-1) \times (n-1)$ and cost coefficients $\mathbf{C}'$.
    (c) Let $\Delta_{ip}(\mathbf{v}^m) \leftarrow \nu(\text{Y-LAP}(i,p)),\ \forall (i,p)$.
5. X-LAP solve:
    (a) Update $b'_{ip} \leftarrow b_{ip} + \Delta_{ip}(\mathbf{v}^m),\ \forall (i,p)$.
    (b) Solve a single X-LAP of size $n \times n$ and cost coefficients $\mathbf{b}'$.
    (c) Update $\nu(\text{LRLT2}(\mathbf{v}^m)) \leftarrow \nu(\text{X-LAP})$.
    (d) If $\bar{\nu}(\text{LDRLT2}) < \nu(\text{LRLT2}(\mathbf{v}^m))$, update $\bar{\nu}(\text{LDRLT2}) \leftarrow \nu(\text{LRLT2}(\mathbf{v}^m))$ and GAP.
6. Update the dual multipliers $v^{m+1}_{ijkpqr} \leftarrow v^m_{ijkpqr} + \lambda_{ijkpqr}$, according to some combination of
   rules from Equations (5.30)-(5.32) (and SA if applicable).
7. Update $m \leftarrow m + 1$. Return to Step 2.
```

indices $i \neq j \neq k, p \neq q \neq r$ for the sake of brevity.

$$\text{DLRLT2}(\mathbf{v}^{m+1}) = \max \sum_i \alpha_i + \sum_p \beta_p; \tag{5.33}$$

$$\text{s.t. } \alpha_i + \beta_p - \sum_{j \neq i} \gamma_{ijp} - \sum_{q \neq p} \delta_{ipq} \leq b_{ip}; \tag{5.34}$$

$$\gamma_{ijp} + \delta_{ipq} - \sum_{k \neq i,j} \xi_{ijkpq} - \sum_{r \neq p,q} \psi_{ijpqr} \leq C_{ijpq}; \tag{5.35}$$

$$\xi_{ijkpq} + \psi_{ijpqr} \leq D_{ijkpqr} - v^{m+1}_{ijkpqr}; \tag{5.36}$$

$$\alpha_i, \beta_p, \gamma_{ijp}, \delta_{ipq}, \xi_{ijkpq}, \psi_{ijpqr} \sim \text{unrestricted}. \tag{5.37}$$

We can substitute v^{m+1} in Equation (5.36) with the following expression which arises from the three dual ascent rules (5.30), (5.31), and (5.32).

$$v^{m+1}_{ijkpqr} = v^{m}_{ijkpqr} + \kappa^z \pi(z_{ijkpqr}) + \frac{\kappa^y \pi(y_{ijpq})}{(n-2)} + \frac{\kappa^x \pi(x_{ip})}{(n-1)(n-2)} - \Omega_{ijkpqr}, \tag{5.38}$$

where, $\Omega_{ijkpqr} \geq 0$ represents the sum of fractional slacks $\pi(x)$, $\pi(y)$, and $\pi(z)$, of the five complementary variables of z_{ijkpqr}, as given by the rules (5.30)–(5.32). After substituting Equation (5.38) in Equation (5.36) and rearranging the terms, we obtain the following constraint:

$$\begin{aligned}\xi_{ijkpq} + \psi_{ijpqr} \leq & D_{ijkpqr} - v^{m}_{ijkpqr} - \pi(z_{ijkpqr}) - \frac{\pi(y_{ijpq})}{(n-2)} - \frac{\pi(x_{ip})}{(n-1)(n-2)} \\ & + (1-\kappa^z)\pi(z_{ijkpqr}) + \frac{(1-\kappa^y)\pi(y_{ijpq})}{(n-2)} + \frac{(1-\kappa^x)\pi(x_{ip})}{(n-1)(n-2)} + \Omega_{ijkpqr}. \end{aligned} \tag{5.39}$$

After replacing Equation (5.36) with Equation (5.39), and aggregating the $\frac{\pi(y_{ijpq})}{(n-2)}$ and $\frac{\pi(x_{ip})}{(n-1)(n-2)}$ terms, we can write the following expression:

$$\nu(\text{LRLT2}(\mathbf{v}^{m+1})) = \nu(\text{DLRLT2}(\mathbf{v}^{m+1})) = \max_{\mathbf{\Psi}} \left\{ \sum_i \alpha_i + \sum_p \beta_p \right\}, \tag{5.40}$$

where, $\mathbf{\Psi}$ represents the constraint set:

$$\alpha_i + \beta_p - \sum_{j \neq i} \gamma_{ijp} - \sum_{q \neq p} \delta_{ipq} \leq [b_{ip} - \pi(x_{ip})] + [(1-\kappa^x)\pi(x_{ip})]; \tag{5.41}$$

$$\gamma_{ijp} + \delta_{ipq} - \sum_{k \neq i,j} \xi_{ijkpq} - \sum_{r \neq p,q} \psi_{ijpqr} \leq [C_{ijpq} - \pi(y_{ijpq})] + [(1-\kappa^y)\pi(y_{ijpq})]; \tag{5.42}$$

$$\xi_{ijkpq} + \psi_{ijpqr} \leq \left[D_{ijkpqr} - \pi(z_{ijkpqr}) - v^{m}_{ijkpqr}\right] + [(1-\kappa^z)\pi(z_{ijkpqr}) + \Omega_{ijkpqr}]; \tag{5.43}$$

$$\alpha_i, \beta_p, \gamma_{ijp}, \delta_{ipq}, \xi_{ijkpq}, \psi_{ijpqr} \sim \text{unrestricted}. \tag{5.44}$$

If we split the constraint set $\mathbf{\Psi}$ into two constraint sets $\mathbf{\Psi_1}$ and $\mathbf{\Psi_2}$, such that,

$$\mathbf{\Psi_1} \quad : \quad \alpha_i + \beta_p - \sum_{j \neq i} \gamma_{ijp} - \sum_{q \neq p} \delta_{ipq} \leq b_{ip} - \pi(x_{ip}); \tag{5.45}$$

$$\gamma_{ijp} + \delta_{ipq} - \sum_{k \neq i,j} \xi_{ijkpq} - \sum_{r \neq p,q} \psi_{ijpqr} \leq C_{ijpq} - \pi(y_{ijpq}); \tag{5.46}$$

$$\xi_{ijkpq} + \psi_{ijpqr} \leq D_{ijkpqr} - v^{m}_{ijkpqr} - \pi(z_{ijkpqr}); \tag{5.47}$$

$$\alpha_i, \beta_p, \gamma_{ijp}, \delta_{ipq}, \xi_{ijkpq}, \psi_{ijpqr} \sim \text{unrestricted}; \tag{5.48}$$

$$\mathbf{\Psi_2} \quad : \quad \alpha_i + \beta_p - \sum_{j \neq i} \gamma_{ijp} - \sum_{q \neq p} \delta_{ipq} \leq (1-\kappa^x)\pi(x_{ip}); \tag{5.49}$$

$$\gamma_{ijp} + \delta_{ipq} - \sum_{k \neq i,j} \xi_{ijkpq} - \sum_{r \neq p,q} \psi_{ijpqr} \leq (1 - \kappa^y)\pi(y_{ijpq}); \tag{5.50}$$

$$\xi_{ijkpq} + \psi_{ijpqr} \leq (1 - \kappa^z)\pi(z_{ijkpqr}) + \Omega_{ijkpqr}; \tag{5.51}$$

$$\alpha_i, \beta_p, \gamma_{ijp}, \delta_{ipq}, \xi_{ijkpq}, \psi_{ijpqr} \sim \text{unrestricted}; \tag{5.52}$$

then, from the theory of linear programming, we can show that:

$$\max_{\boldsymbol{\Psi}} \left\{ \sum_i \alpha_i + \sum_p \beta_p \right\} \geq \max_{\boldsymbol{\Psi_1}} \left\{ \sum_i \alpha_i + \sum_p \beta_p \right\} + \max_{\boldsymbol{\Psi_2}} \left\{ \sum_i \alpha_i + \sum_p \beta_p \right\}. \tag{5.53}$$

Finally, we can assert that: $\nu(\text{LRLT2}(\mathbf{v}^m)) = \nu(\text{DLRLT2}(\mathbf{v}^m)) = \max_{\Psi_1} \left\{ \sum_i \alpha_i + \sum_p \beta_p \right\}$, and due to the non-negativity of $\pi(x), \pi(y), \pi(z)$, we have: $\max_{\Psi_2} \left\{ \sum_i \alpha_i + \sum_p \beta_p \right\} \geq 0$. Therefore,

$$\nu(\text{LRLT2}(\mathbf{v}^{m+1})) \geq \nu(\text{LRLT2}(\mathbf{v}^m)). \tag{5.54}$$

□

5.3 Accelerating RLT2-DA Algorithm Using a GPU Cluster

The RLT2-DA algorithm described in the previous section was shown to outperform the Lagragian subgradient search and many other algorithms in a branch-and-bound scheme, in terms of lower bound strength and the number of nodes fathomed, for problems with $n \leq 30$. However, for solving a QAP of size n using RLT2-DA, we need to solve $O(n^4)$ LAPs of size $O(n^2)$, and update $O(n^6)$ dual multipliers. The overall solution complexity of sequential RLT2-DA is $O(n^7)$. An important observation about RLT2-DA is that the $O(n^4)$ LAPs can be solved independently of each other and similarly, the $O(n^6)$ Lagrangian multipliers can be updated independently of each other. Therefore, with the help of a correct parallel programming architecture, it is possible to achieve significant speedup over the sequential algorithm.

In the parallel algorithm that we have implemented, both phases of the sequential algorithm are executed on the GPU(s) by one or more CUDA kernels. We have chosen CUDA enabled NVIDIA GPUs as our primary architecture, because a GPU offers a large number of processor cores which can process a large number of threads in parallel. This is extremely useful for the dual update phase of RLT2-DA, since one CUDA thread can be assigned to each multiplier, and a host of multipliers can be updated at a time. Additionally, our efficient GPU-accelerated algorithm for the LAP can be used to speed up the LAP solution phase of RLT2-DA. Both these algorithms can be combined into a GPU-accelerated RLT2-DA solver engine, which can obtain strong lower bounds on the QAP, in an efficient manner.

In the single GPU implementation of RLT2-DA solver, it may become challenging to store the $O(n^6)$ cost coefficients in the GPU memory, especially for larger problems. One of the alternatives to overcome this problem is to store the matrices in the CPU memory and copy them into the

GPU memory as required. However, this approach requires $O(n^6)$ transfer between the CPU and GPU, which may introduce severe communication overhead. A better alternative is to split the cost coefficients (or LAP matrices) across multiple GPUs (if available), which also allows us to solve several LAPs concurrently. In this work, we have used grid architecture with multiple processing elements (PE), each containing one CPU-GPU pair. Communication between the different CPUs is accomplished using *message passing interface* (MPI). The overall algorithmic architecture is shown in Fig. 5.2 and the details of our implementation are described in the following sections.

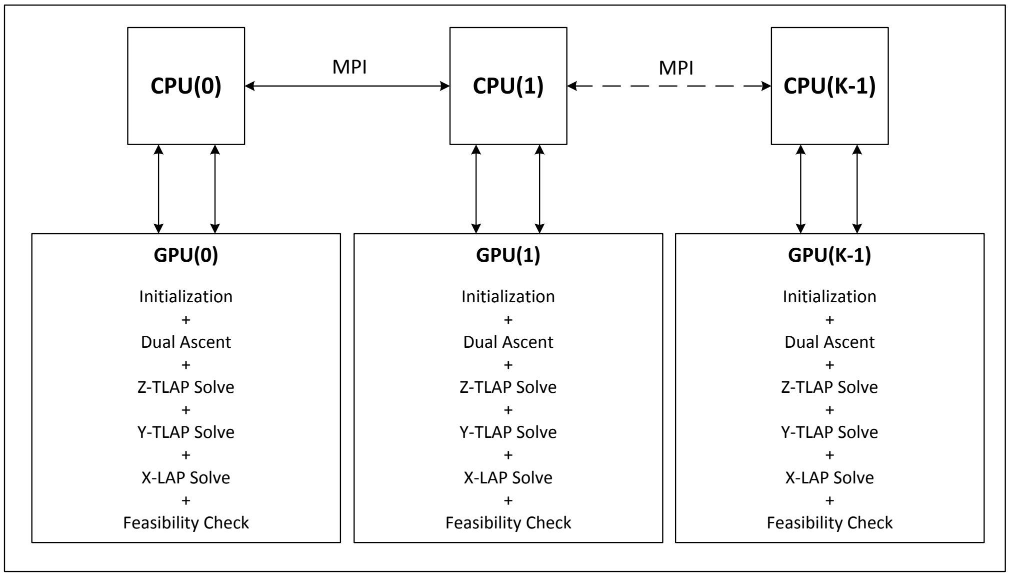


Figure 5.2: Parallel/accelerated RLT2-DA

5.3.1 Initialization

The program is initialized with K MPI processes, equal to the number of PEs in the grid. It is assumed that one MPI process gets allocated to exactly one CPU. The CPU with rank 0 is chosen as the root. The cost matrices for the Y and Z-LAPs are split evenly across all the GPUs in the grid, i.e., each device owns $M_z = \left\lceil \frac{n^2(n-1)^2}{K} \right\rceil$ number of Z-LAP matrices and $M_y = \left\lceil \frac{n^2}{K} \right\rceil$ number of Y-LAP matrices. The X-LAP matrix is owned only by the root GPU.

5.3.2 LAP Solution

All the LAPs are solved on the GPU using the *alternating-tree variant* of our accelerated Hungarian algorithm. We know that our GPU-accelerated Hungarian algorithm is extremely efficient in solving large LAPs, rather than small LAPs. Therefore, all the LAPs owned by a particular GPU are combined and solved as a *tiled* LAP (or TLAP). For example, if we have M_z number of Z-LAP matrices (of size $(n-2)\times(n-2)$) on a particular GPU, then instead of solving them one at a time, we stack these LAP matrices and solve a single TLAP of size $M_z \times (n-2) \times (n-2)$. The solution complexity for a TLAP is $O(M^{1.5}n^3)$, which is asymptotically worse than $O(Mn^3)$. However, in practice, we found that a single TLAP solves much faster than solving individual LAPs one by one. We suspect that this happens because of the execution overhead incurred due to repeated invocation of the CUDA kernels in the one-at-a-time approach versus invoking the kernels only once in the tiled approach.

5.3.3 Dual Ascent

As mentioned earlier, Lagrangian multiplier update is quite straightforward to parallelize on a GPU. There is exactly one dual multiplier associated with one of z variables, and we can easily assign one GPU thread per element of the LAP cost matrices residing on a particular GPU. It is important to note that we do not need any additional data structures to store the dual multipliers $\mathbf{v}$, since we only need the updated cost coefficients $\mathbf{b}'$, $\mathbf{C}'$, and $\mathbf{D}'$ during each iteration of RLT2-DA. Therefore, during the multiplier update step, these cost coefficients can be updated in-place with the specified ascent rule(s).

For updating the dual multipliers (or cost coefficients), we need the dual slacks $\pi(x)$, $\pi(y)$, and $\pi(z)$ of the complementary variables during each iteration, which might not be native to a particular GPU. In short, before each iteration, we need to transfer the arrays of dual variables $\boldsymbol{\alpha}, \boldsymbol{\beta}, \boldsymbol{\gamma}, \boldsymbol{\delta}, \boldsymbol{\xi}, \boldsymbol{\psi}$ and the modified cost matrices $\mathbf{b}'$, $\mathbf{C}'$, and $\mathbf{D}'$ between the various CPUs/GPUs, using MPI. This might incur a significant MPI communication overhead, since the matrix $\mathbf{D}'$ contains $O(n^6)$ elements. To alleviate this overhead, local copies of the complementary cost coefficients are stored on a GPU, for each of the $M_z(n-2)^2$ number of D_{ijkpqr} cost coefficients owned by that GPU. Therefore, we only need to transfer the dual variables $\boldsymbol{\alpha}, \boldsymbol{\beta}, \boldsymbol{\gamma}, \boldsymbol{\delta}, \boldsymbol{\xi}, \boldsymbol{\psi}$ and the cost matrices $\mathbf{b}', \mathbf{C}'$. During the dual ascent phase, the dual slack calculation and the dual multiplier update is performed locally, however we have to make sure that ascent rules are consistent for complementary variables spread across the various GPUs. This approach involves some duplication of work, however the communication complexity is reduced from $O(n^6)$ to $O(n^5)$.

5.3.4 Accelerated RLT2-DA and Variants

The parallel algorithm for RLT2-DA is depicted in Algorithm 16. Most of the steps in this algorithm are same as that of the sequential algorithm, with the exception that the LAP solution and the dual update phases are performed on the GPU, and there are additional MPI communication steps.

Two variants of the accelerated RLT2-DA algorithm are implemented, namely *Slow* RLT2-DA (S-RLT2-DA) and *Fast* RLT2-DA (F-RLT2-DA), which are based on Equation (5.53). In the S-RLT2-DA variant, the LAPs with updated cost coefficients $\mathbf{b}'$, $\mathbf{C}'$, and $\mathbf{D}'$ are solved during each iteration, which corresponds to the left hand side of Equation (5.53). The steps mentioned in Algorithm 16 are essentially that of S-RLT2-DA. In the F-RLT2-DA variant, the LAPs are solved with the incremental cost coefficients (the fractional dual slacks $\boldsymbol{\pi}(\mathbf{x}), \boldsymbol{\pi}(\mathbf{y}), \boldsymbol{\pi}(\mathbf{z})$) from the right hand side of Equation (5.53), and the result is added to the lower bound obtained during

Algorithm 16: Accelerated RLT2-DA.

```
1. Initialization:
    (a) Initialize $m \leftarrow 0$, $\mathbf{v}^m \leftarrow \mathbf{0}$, $\bar{\nu}(\text{LDRLT2}) \leftarrow -\infty$, and GAP $\leftarrow \infty$.
    (b) Initialize $\mathbf{b}'$ on GPU(0). Initialize $\mathbf{C}'$ and $\mathbf{D}'$ on respective GPUs.
2. Termination: Stop if $m >$ ITN_LIM or GAP $<$ MIN_GAP or Feasibility check $= true$.
3. Z-LAP solve (parallely on $K$ GPUs):
    (a) Update $D'_{ijkpqr} \leftarrow D'_{ijkpqr} - v^m_{ijkpqr}, \forall(i \neq j \neq k, p \neq q \neq r)$
    (b) Solve Z-TLAP of size $M_z \times (n-2) \times (n-2)$ and cost coefficients $\mathbf{D}'$.
    (c) Let $\Theta_{ijpq}(\mathbf{v}^m) \leftarrow \nu(\text{Z-LAP}(i,j,p,q))$, $\forall(i \neq j, p \neq q)$.
    (d) Broadcast $\Theta_{ijpq}(\mathbf{v}^m)$, $\boldsymbol{\xi}_{ijpq}$, and $\boldsymbol{\psi}_{ijpq}$ to all CPUs/GPUs using MPI_Bcast directive.
4. Y-LAP solve (parallely on $K$ GPUs):
    (a) Update $C'_{ijpq} \leftarrow C_{ijpq} + \Theta_{ijpq}(\mathbf{v}^m), \forall(i \neq j, p \neq q)$.
    (b) Solve Y-TLAP of size $M_y \times (n-1) \times (n-1)$ and cost coefficients $\mathbf{C}'$.
    (c) Let $\Delta_{ip}(\mathbf{v}^m) \leftarrow \nu(\text{Y-LAP}(i,p))$, $\forall(i,p)$.
    (d) Broadcast $\mathbf{C}'$, $\Delta_{ip}(\mathbf{v}^m)$, $\boldsymbol{\gamma}_{ip}$, and $\boldsymbol{\delta}_{ip}$ to all CPUs/GPUs using MPI_Bcast directive.
5. X-LAP solve (only on GPU(0)):
    (a) Update $b'_{ip} \leftarrow b_{ip} + \Delta_{ip}(\mathbf{v}^m)$, $\forall(i,p)$.
    (b) Solve a single X-LAP of size $n \times n$ and cost coefficients $\mathbf{b}'$.
    (c) Update $\nu(\text{LRLT2}(\mathbf{v}^m)) \leftarrow \nu(\text{X-LAP})$.
    (d) If $\bar{\nu}(\text{LDRLT2}) < \nu(\text{LRLT2}(\mathbf{v}^m))$, update $\bar{\nu}(\text{LDRLT2}) \leftarrow \nu(\text{LRLT2}(\mathbf{v}^m))$ and GAP.
    (e) Broadcast $\bar{\nu}(\text{LDRLT2})$, GAP, $\mathbf{b}'$, $\boldsymbol{\alpha}$, and $\boldsymbol{\beta}$ to all CPUs/GPUs using MPI_Bcast
        directive.
6. (Parallely on $K$ GPUs) Update the dual multipliers $v^{m+1}_{ijkpqr} \leftarrow v^m_{ijkpqr} + \lambda_{ijkpqr}$, according to
   some combination of rules from Equations (5.30)-(5.32) (and SA if applicable).
7. Update $m \leftarrow m + 1$. Return to Step 2.
```

the previous iteration. This variant has smaller execution time (hence the name *fast*), since the incremental cost coefficient matrices are sparser as compared to the actual cost coefficient matrices.[1] However, as the inequality suggests, the lower bound of S-RLT2-DA is stronger than that of F-RLT2-DA. Another advantage of using S-RLT2-DA is that during each iteration, we have the updated dual multipliers (in the form of cost coefficients $\mathbf{b}', \mathbf{C}', \mathbf{D}'$) which can be used as a starting solution for the children nodes in a branch-and-bound scheme. However, in F-RLT2-DA, recovering the actual dual multipliers is not so straightforward.

For both the above variants, a stronger lower bound can be obtained by adopting a two-phase approach. In the first phase (Step 3b-1), Z-TLAPs with the cost coefficients $\mathbf{D}'$ are solved. During the second phase, initially (Step 3b-2), for each $(i < j < k, p \neq q \neq r)$, the six complementary z variables are partitioned into two sets based on their dual slacks: $S_B = \{z : \pi(z) = 0\}$ and $S_N = \{z : \pi(z) > 0\}$. Then, the dual slacks of the complementary variables from S_N are added; their cost coefficients are reduced by the corresponding $\pi(z)$; and the sum is evenly distributed across the cost coefficients of the complementary variables from S_B. Finally (Step 3b-3), the Z-TLAPs are re-solved and the algorithm continues to Step 3c. Since we are solving the TLAPS two times, this two-phase approach takes almost twice the time of the one-phase approach, but it provides the strongest lower bounds.

A third variant is to use Simulated Annealing with some temperature schedule, in which the algorithm is allowed to redistribute a random fraction of the current lower bound among some of the z variables (see Type 4 ascent rule in Section 5.2). This deteriorating step provides the algorithm with an opportunity to get out of a local maximum, which further improves the lower bound.

In Section 5.5, we will compare the lower bounds and execution times for each of the above variants, which will provide significant insight to researchers to make careful selection. We will refer to these variants as F1, F2, S1, S2, where the first letter is used for distinction between fast and slow variants, while the number indicates whether it is a single-phase or two-phase algorithm.

5.4 Parallel Branch-and-bound with Accelerated RLT2-DA

Although the objective function value of the LP relaxation of RLT2 was shown to provide tight lower bound (equal to the integer optimal) for the small QAPs ($n \leq 12$), the LP relaxation is expected to have a duality gap for medium and large QAPs. Also, due to the total unimodularity of LRLT2($\mathbf{v}$), ν(LDRLT2) can only ever reach the LP relaxation bound. Therefore, the LDRLT2 objective function value provides a lower bound on both LPRLT2 and QAP objective function values, and RLT2-DA cannot be used on its own to find exact solutions to large QAPs. To this end, we need to use the branch-and-bound (B&B) procedure to solve medium and large QAPs to optimality.

[1]This phenomenon was observed in large scale computational tests of GPU-accelerated Hungarian algorithm presented in Section 4.6.2.

B&B is a standard procedure used for solving integer optimization problems, and it is implemented in the similar fashion as any search tree. Each node in this B&B tree corresponds to a subproblem in which a single variable is assigned a specific value. This assignment partitions the solution space into two or more disjoint subspaces. Solving an LP relaxation of the subproblem at a particular node provides a lower bound on that node. A node and its children are fathomed if any one of the following three conditions are satisfied: (1) The lower bound at that node exceeds or equals the incumbent objective value; (2) The subproblem is infeasible; or (3) The subproblem has an integer solution. Fathomed nodes are not considered for further exploration and the whole branch is removed from the tree. Therefore, quality of the lower bound is of utmost importance, so as to explore as few nodes as possible. Another important consideration in B&B scheme is the search strategy to be employed for exploring the tree. The tree could be explored using Breadth-First-Search (BFS), or Depth-First-Search (DFS), or Best-First-Search (BstFS), each of which has its own pros and cons. We used a hybrid BstFS+DFS approach, since it was more suitable for the problem under study.

The specifics of our parallel B&B procedure can be explained as follows. At the root level of the B&B tree, none of the facility locations are fixed. At each subsequent level ℓ of the B&B tree, the locations of the first $(\ell - 1)$ facilities are fixed. The ℓ^{th} facility is assigned to each one of the remaining $(n - \ell + 1)$ locations, giving rise to $(n - \ell + 1)$ children nodes at that level. This type of branching is called "polytomic" branching and it has been used for solving QAPs by Roucairol (1987), Pardalos and Crouse (1989), Clausen and Perregaard (1997), and Anstreicher et al. (2002) using other formulations and lower bounding techniques.

For each node of the B&B tree, the lower bound is obtained using our accelerated RLT2-DA executed by a bank of PEs (CPU-GPU pairs). Performing RLT2-DA on the root node produces the root lower bound (reported in Section 5.5). Parallelizing the B&B procedure is quite straightforward since we can allocate multiple banks of PEs, such that each bank is responsible for a subset of unexplored nodes and corresponding sub-trees. Load balancing is a critical aspect of parallel B&B so as to improve processor utilization. Load balancing can be achieved by precisely managing the queue of unexplored nodes and redistributing them on the idle PE banks as required. Figure 5.3 shows the architecture used in our parallel B&B scheme.

1. We begin the parallel B&B with N banks containing K PEs each. CPU0 is designated as the "Master Processor" (MP), which maintains a "master list" of unexplored nodes such that the node with the smallest lower bound is at the top of the list (essentially a "heap"). This makes sure that the first PE Bank is always working on the nodes with the best bound (essentially a BstFS). The master list is seeded with some initial set of nodes from some level ℓ_{init}. As an example, if $\ell_{\text{init}} = 0$, then the list contains only the root node in which none of the facilities are assigned to any of the locations. For $\ell_{\text{init}} = 1$, there will be n nodes in which facility 0 is assigned to all the n locations; etc.

2. The nodes from the master list are equally distributed among all the PEs, which explore the

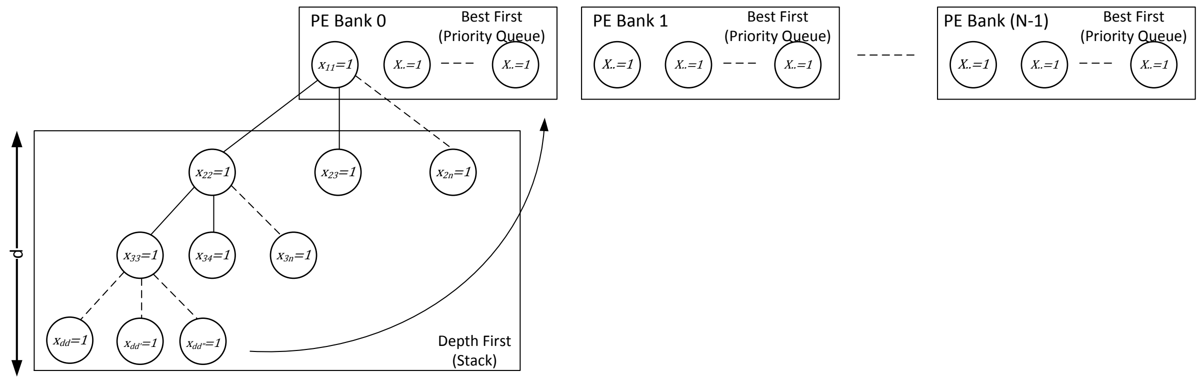


Figure 5.3: Parallel branch-and-bound

respective sub-trees in a DFS manner. In some of the QAP instances (e.g., Nugent instances) the locations are present on a grid, which allows us to apply symmetry elimination rules and eliminate a number of assignments which will have the same objective value.

3. For each of the nodes allocated to a particular PE bank, a lower bound is calculated using our accelerated RLT2-DA with SA. An "adaptive" strategy is used for the lower bound calculation. Initially, for a particular node at level ℓ, a quick lower bound is obtained by executing a maximum of 250 iterations of "fast" RLT2-DA with SA. If the node is not fathomed, then a maximum of 500 iterations of the "slow" RLT2-DA with SA are performed, which not only provides a tighter bound, but also provides the best dual multipliers for that sub-problem. The fast RLT2-DA acts as a filter which allows us to eliminate the "easy" nodes quickly, before reverting to the slow RLT2-DA and branching.

4. If a node at level ℓ cannot be fathomed even after performing "slow" RLT2-DA, then it is branched upon by placing the $(\ell + 1)^{th}$ facility at all the available locations, generating $(n - \ell)$ children. For all these children, the dual multipliers of the parent node can be used as an initial solution (warm start), which saves us from calculating the dual multipliers from scratch. This significantly speeds up RLT2-DA execution.

5. An important aspect of B&B is to distinguish between the amount of time spent in improving the lower bound versus branching, which will produce a number of children with improved lower bounds. For this purpose, some early termination criteria are used for RLT2-DA. In these criteria, the RLT2-DA iterations are stopped and the node is branched upon, if the optimality gap does not improve by a total of 0.0002 points within the last 15 iterations.

6. While the DFS strategy keeps all the PE banks fairly busy, it is possible that some PE banks may obtain "easy" nodes which can fathomed fairly quickly. In such cases, those PE banks

will remain idle, which is detrimental for the system utilization. Therefore, a load balancing scheme can be implemented similar to the one implemented by Anstreicher et al. (2002). In this scheme, the idle PE bank sends a request to the MP. The MP checks the request queue after every 300 seconds. If there is at least one processor that is idle, then the unexplored nodes from all the PE banks are collected by the MP and redistributed evenly across all the PEs.

7. Finally, it may be beneficial to limit the DFS exploration up to a maximum depth d. This is because, we need to save the $O(n^6)$ dual multipliers associated with each node at a particular level, to be able to perform warm start on its children nodes. These multipliers are saved on the CPU memory of the corresponding PEs from the bank, and for depth d, the space complexity becomes $O(d \cdot n^6)$. Any unexplored nodes beyond the depth d are collected and redistributed by the MP, and the memory is reset.

The computational results for this parallel B&B scheme coupled with the accelerated RLT2-DA procedure are presented in the next section.

5.5 Computational Experiments

The accelerated RLT2-DA algorithm was coded in C++ and CUDA C programming languages and deployed on the Blue Waters Supercomputing Facility at the University of Illinois at Urbana-Champaign. The computational resources had the following specifications. Each CPU was an AMD Interlagos model 6276 processor, with 8 cores, 2.3GHz clock speed, and 32GB memory. Each GPU was an NVIDIA GK110 "Kepler" K20X GPU, with 2688 processor cores, and 6GB memory.

Various computational studies were conducted on different variants of our parallel/accelerated RLT2-DA, with respect to the bound strength, scaling behavior, and performance in branch-and-bound procedure. For testing purposes, we used various solved and unsolved instances of size $18 \leq n \leq 40$ from the QAPLIB (Burkard et al., 1997). These computational tests are documented in the following sections.

For the dual ascent variants, the following parameter values were used in the ascent rules: $\kappa^z = \frac{5}{6}$, $\kappa^y = 1$, $\kappa^x = 1$ and $\phi^z_{(\cdot)} = \phi^y_{(\cdot)} = \phi^x_{(\cdot)} = \frac{1}{5}$. For the SA based variants, the following annealing schedule was used. The initial temperature was set to 100, and the reduction factor was set to 0.99. The fractions κ^{lb}_i and κ^{lb}_p were generated randomly $\forall i, p$; with a constraint that $\sum_i \kappa^{lb}_i + \sum_p \kappa^{lb}_p \leq 0.25$. This means that at most 25% of the current lower bound was made available for redistribution using Type 4 ascent rule.

Since accelerated RLT2-DA algorithm is memory intensive, instances of specific size requires a certain minimum number of GPUs to be able to fit all the necessary data structures. Table 5.2 lists the minimum number of PEs required to be able to comfortably solve the QAP instances of different sizes. Note that these numbers are derived according to the specifications of GPUs that we used for testing. These numbers might change if we use GPUs with specifications other than the ones mentioned earlier.

Table 5.2: Minimum number of PEs for various problem sizes

n	≤ 20	22	25	27	30	35	40
Minimum # of PEs	1	2	4	7	15	55	500

5.5.1 Computational Results for Bound Strength

To compare the strength of lower bounds, we used the Nug20 instance (which has $n = 20$ facilities and locations) from the QAPLIB. On this instance, we ran 2000 iterations of the different variants. The tests were performed with only a single PE. All the Z-LAPs were tiled into a single Z-TLAP, and all the Y-LAPs were tiled into a single Y-TLAP. For these tests, we noted the lower bounds and execution times. For the first iteration, i.e., for $\boldsymbol{v} = \mathbf{0}$, we obtain the Gilmore-Lawler bound of 2057. After that, RLT2-DA obtains an increasing sequence of lower bounds during the subsequent iterations, in accordance with Theorem 8. The results are shown in Table 5.3, Fig. 5.4, and Fig. 5.5.

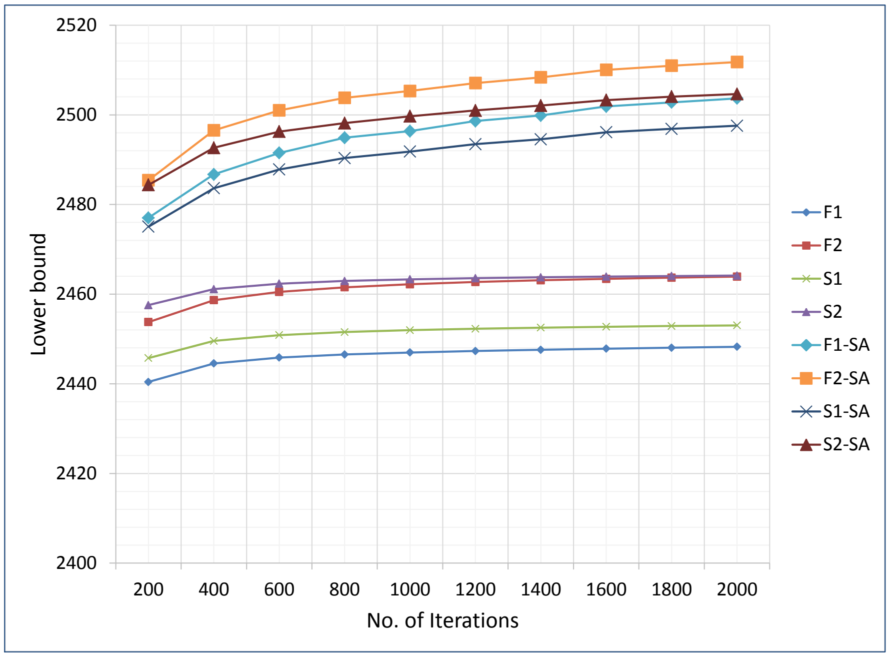


Figure 5.4: Lower bounds of accelerated RLT2-DA variants

In general, for non-SA variants, we can see that the lower bounds obtained using the “slow” variants are stronger than the ones obtained using the “fast” variants. Additionally, the lower bounds obtained using the “2-phase” variants are stronger than the ones obtained using the “1-phase” variants. The lower bounds obtained using SA are significantly stronger than their non-SA counterparts. However, the lower bounds obtained using the “fast” variants with SA are much

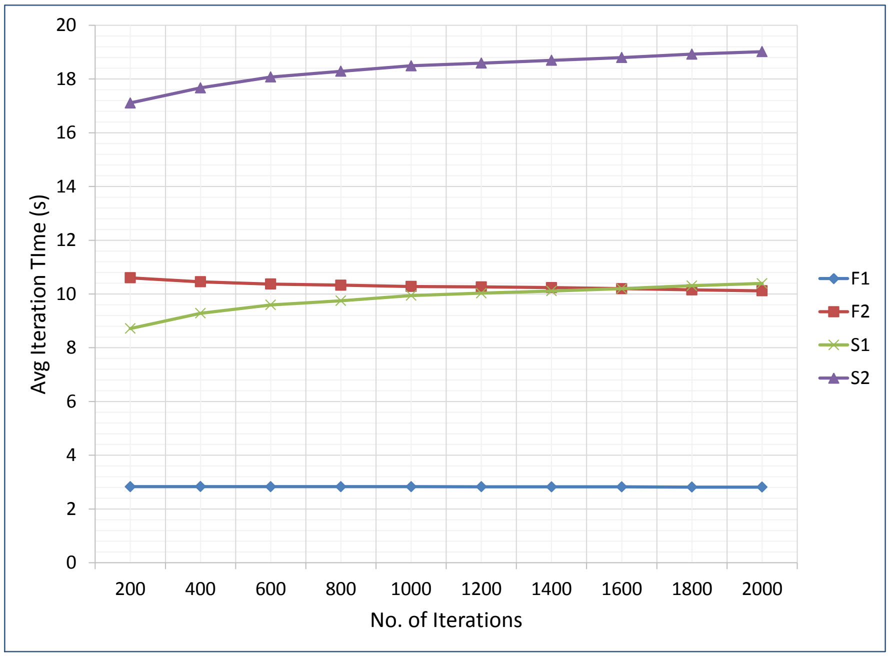


Figure 5.5: Iteration times of accelerated RLT2-DA variants

stronger than the ones obtained using the "slow' variants with SA, contrary to Equation (5.53). The reason behind this behavior is not completely understood but we suspect that the "fast" variants allow for a better redistribution of the fractional ν(LDRLT2) among the cost coefficients of the complementary z variables. The iteration times of "fast" variants are much shorter than the "slow" methods, and we can see that "1-phase" variants are at least twice as fast as "2-phase" variants. Additionally, the average iteration time for the "fast" variants remains more or less the same with increasing number of iterations. However, for the "slow" variants, the average iteration time increases with the number of iterations. The reason for this phenomenon is that as we update the dual multipliers, cost coefficients are spread further apart, thereby increasing the time spent in "augmenting path search" and "dual update" steps of the Hungarian algorithm (refer to Section 4.6 for more details). In general, we can see that S2-RLT2-DA is dominated both in terms of execution time and lower bound strength. The primary bottleneck in the iteration time RLT2-DA is the Z-TLAP solution phase, however, we can increase the number of PEs (up to a certain limit) and solve more TLAPs in parallel to further reduce the iteration time. We will see these scaling results in the next section.

In addition to Nug20, we performed 2000 iterations of SA-based F2-RLT2-DA on some of the other well-known instances from the QAPLIB. The results are listed in Table 5.4. The percentage optimality gap is calculated as $100 \times \frac{\text{Best UB}-\text{LB}}{\text{LB}}$. We can see that accelerated RLT2-DA can be used to find strong lower bounds on large problem instances.

Finally, we compared the parallel RLT2-DA method with a state-of-the-art method based

Table 5.3: Bound strength of RLT2-DA variants on Nug20

Itn	F1			F2			S1			S2		
	w/o SA	w/ SA	Time (s)	w/o SA	w/ SA	Time (s)	w/o SA	w/ SA	Time (s)	w/o SA	w/ SA	Time (s)
200	2440.4	2477.02	566.126	2453.77	2485.41	2120.63	2445.74	2475.06	1743.26	2457.55	2484.36	3421.64
400	2444.53	2486.71	1133.48	2458.65	2496.54	4181.65	2449.57	2483.62	3714.44	2461.12	2492.64	7068.8
600	2445.86	2491.51	1698.15	2460.49	2501.01	6221.61	2450.86	2487.82	5757.08	2462.34	2496.28	10845.3
800	2446.54	2494.9	2265.94	2461.52	2503.8	8262.34	2451.53	2490.36	7801.03	2462.94	2498.15	14629.6
1000	2446.99	2496.36	2830.11	2462.21	2505.33	10281.1	2451.96	2491.82	9944.8	2463.3	2499.68	18491.5
1200	2447.31	2498.61	3394.96	2462.7	2507.09	12316	2452.28	2493.46	12036.6	2463.56	2500.98	22311.7
1400	2447.58	2499.85	3953.12	2463.09	2508.37	14332.8	2452.53	2494.54	14158.7	2463.75	2502.06	26172.7
1600	2447.83	2501.87	4517.88	2463.41	2510.04	16319.5	2452.72	2496.11	16318.5	2463.91	2503.29	30077.4
1800	2448.05	2502.78	5068.61	2463.67	2510.97	18274.6	2452.89	2496.88	18554.3	2464.04	2504.07	34062.8
2000	2448.26	2503.67	5627.15	2463.91	2511.79	20235.6	2453.03	2497.57	20786.3	2464.15	2504.64	38039.2

Table 5.4: F2-RLT2-DA lower bounds for various instances from QAPLIB (2000 iterations)

Problem	LAP Counts (X, Y, Z)	# of PEs	LB	OPT	% GAP	Itn time (s)
Nug18	(1, 324, 93636)	1	1909.29	1930	1.08	5.07
Nug20	(1, 400, 144400)	1	2511.79	2570	2.32	10.12
Nug22	(1, 484, 213444)	2	2603.84	2650	1.77	10.64
Nug25	(1, 625, 360000)	4	3582.83	3744	4.50	12.23
Nug27	(1, 729, 492804)	7	5000.78	5234	4.66	12.52
Nug30	(1, 900, 518400)	15	5755.35	6124	6.41	12.86

on Semi-definite Programming (SDP) relaxation, proposed by Peng et al. (2015). Lower bound strength is compared in terms of $\%R_{gap}$, which is defined as $100 \times \frac{\text{Best UB}-\text{LB}}{\text{Best UB}}$. Table 5.5 shows the $\%R_{gap}$ and execution times reported by Peng et al. (2015), for three different variants of their SDP based formulation, on Nugent problem instances. In the last two columns, we have reported the number of iterations and the execution time required for F2-RLT2-DA to reach the $\%R_{gap}$ of the SDRMS-SUM formulation, since it is the best formulation among the three. We can see that the SDRMS-SUM formulation provides strong lower bounds extremely quickly. Ultimately, RLT2-based lower bounds are a lot stronger, as seen in Table 5.4. If implemented in a B&B scheme, SDP based methods will end up exploring more nodes than RLT2-DA, but time spent per node will be significantly smaller.

Table 5.5: Comparison with SDP relaxation based lower bounds with RLT2-DA

Problem	SDRMS-SUM		SDRMS-SVD		SDRMS-ONE		F2-RLT2-DA	
	$\%R_{gap}$	Time (s)	$\%R_{gap}$	Time (s)	$\%R_{gap}$	Time (s)	Itns	Time (s)
Nug18	9.17	15	9.38	17	9.74	8	8	46.31
Nug20	9.03	19	9.3	24	9.69	10	10	105.64
Nug22	8.68	26	9.15	33	9.54	12	12	148.2
Nug25	9.05	36	9.19	46	9.75	17	23	308.94
Nug27	7.91	48	8.41	60	8.79	20	42	591.98
Nug30	8.43	67	8.54	79	8.93	28	87	1219.64

5.5.2 Computational Results for Parallel Scalability

Although, there is a minimum required number of PEs for executing accelerated RLT2-DA on a QAP of specific size, the number of PEs can be increased and the LAPs can be solved parallely on multiple PEs. This allows us to achieve significant parallel speedup. We performed strong scalability study of our accelerated RLT2-DA algorithm (specifically the S2 variant) on Nug20 problem instance. For this study, the PEs were increased from 1 to 32 in geometric increments of 2. For each PE category, the Y-TLAPs and Z-TLAPs were split evenly across all the GPUs and solved parallely during each iteration. The results for the parallel scalability study are shown in Table 5.6 and Fig. 5.6. We can see that, as we continue to increase the number of PEs in the system, we get diminishing returns in the execution times. In other words, doubling the number of PEs does not necessarily reduce the execution time by half. This happens because increasing the number of PEs also increases the MPI communication. At some point, adding more PEs in the system will actually increase the execution time, because the communication will start to dominate.

Table 5.6: Scalability results for S2-RLT2-DA on Nug20 (200 iterations)

LAP Counts	Execution Time (s)						
(X, Y, Z)	1 PE	2 PE	4 PE	8 PE	16 PE	20 PE	32 PE
(1, 400, 144400)	17.11	10.90	7.05	5.43	3.79	3.58	3.78

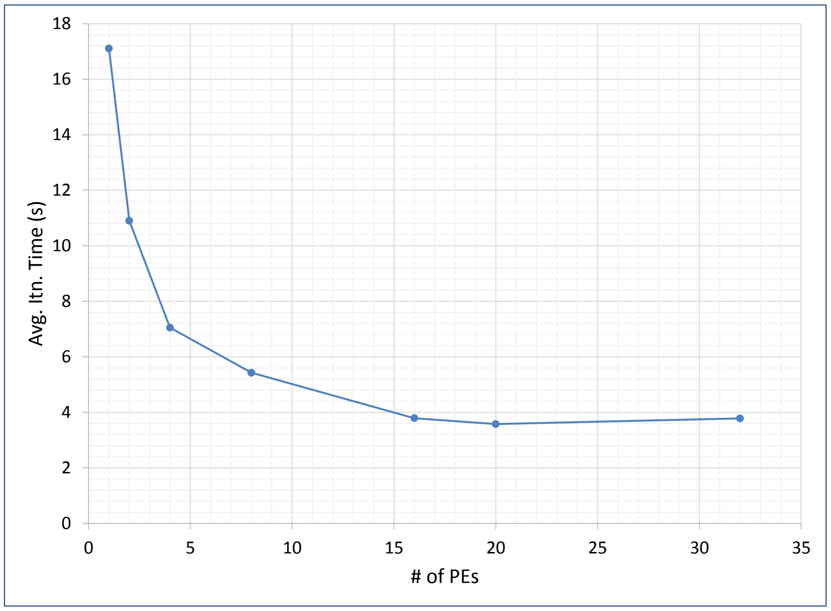


Figure 5.6: Strong scalability tests for S2-RLT2-DA on Nug20

5.5.3 Computational Results for Addition of Valid Inequalities to RLT2

The lower bounds obtained using RLT2-DA can be improved further by introducing valid inequalities to the RLT2 formulation. These valid inequalities connect the y_{ijpq} variables (and their complements) with the pair (x_{ip}, x_{jq}); and the z_{ijkpqr} variables (and their complements) with the triplet (x_{ip}, x_{jq}, x_{kr}). These constraints can be written as follows:

$$x_{ip} + x_{jq} \leq y_{ijpq} + 1;\ \forall (i \neq j, p \neq q); \tag{5.55}$$

$$x_{ip} + x_{jq} + x_{kr} \leq z_{ijkpqr} + 2;\ \forall (i \neq j \neq k, p \neq q \neq r). \tag{5.56}$$

Constraint (5.55) enforces that, if $x_{ip} = x_{jq} = 1$, then y_{ijpq} should be set to 1. Similarly, constraint (5.56) enforces that, if $x_{ip} = x_{jq} = x_{kr} = 1$, then z_{ijkpqr} should be set to 1.

These constraints can be incorporated in the RLT2-DA in the same spirit as the feasibility check. Specifically, for some $(i \neq j, p \neq q)$, if $x_{ip} = x_{jq} = 1$, then we know that the dual slacks $\pi(x_{ip}) = \pi(x_{jq}) = 0$. In that case, we can write the following dual ascent rule:

$$b_{ip} \leftarrow b_{ip} + \frac{\pi(y_{ijpq})}{2}; \tag{5.57}$$

$$b_{jq} \leftarrow b_{jq} + \frac{\pi(y_{ijpq})}{2}; \tag{5.58}$$

$$C_{ijpq} \leftarrow C_{ijpq} - \pi(y_{ijpq}). \tag{5.59}$$

Similarly, for some $(i \neq j \neq k, p \neq q \neq r)$, if $x_{ip} = x_{jq} = x_{kr} = 1$, then we know that the dual slacks $\pi(x_{ip}) = \pi(x_{jq}) = \pi(x_{kr}) = 0$. In that case, we can write the following dual ascent rule:

$$b_{ip} \leftarrow b_{ip} + \frac{\pi(z_{ijkpqr})}{3}; \tag{5.60}$$

$$b_{jq} \leftarrow b_{jq} + \frac{\pi(z_{ijkpqr})}{3}; \tag{5.61}$$

$$b_{kr} \leftarrow b_{kr} + \frac{\pi(z_{ijkpqr})}{3}; \tag{5.62}$$

$$D_{ijkpqr} \leftarrow D_{ijkpqr} - \pi(z_{ijkpqr}). \tag{5.63}$$

After applying these rules, the X-LAP is re-solved to obtain a new objective function value. Since $C_{ijpq} - \pi(y_{ijpq}) \geq 0$ and $D_{ijkpqr} - \pi(z_{ijkpqr}) \geq 0$, dual feasibility is maintained. Additionally, since we are adding a non-negative fraction to the cost coefficients of the basic x variables, the objective function will experience a non-negative increase.

Table 5.7 depicts the results for this augmented RLT2-DA (S1 variant with SA). We can clearly see that there is at least 6-8 point improvement in the lower bounds when compared against the F2-RLT2-DA results with SA, and since it is a 1-phase variant, it is faster. To the best of our knowledge, we have improved upon the lower bounds that were published in the literature, specifically for the RLT2 formulation.

Table 5.7: Lower bound comparison for augmented RLT2-DA

Itn	NUG18		NUG20		NUG22		NUG25	
	F2-LB	AUG-S1-LB	F2-LB	AUG-S1-LB	F2-LB	AUG-S1-LB	F2-LB	AUG-S1-LB
200	1889.83	1900.69	2485.41	2493.56	2575.69	2587.13	3529.46	3531.98
400	1899.58	1909.07	2496.54	2503.96	2583.03	2593.48	3547.01	3547.94
600	1902.7	1911.93	2501.01	2508.07	2590.25	2600.57	3560.19	3561.51
800	1904.5	1913.47	2503.8	2510.51	2593.95	2604.02	3564.91	3565.61
1000	1905.64	1914.43	2505.33	2512.02	2595.97	2605.88	3569	3569.54
1200	1906.6	1915.29	2507.09	2513.6	2598.68	2608.53	3572.63	3573.37
1400	1907.75	1916.41	2508.37	2514.69	2600.71	2610.58	3575.95	3576.53
1600	1908.6	1917.06	2510.04	2516.17	2601.89	2611.53	3578.84	3579.07
1800	1908.94	1917.29	2510.97	2516.93	2603.11	2612.56	3581.08	3581.42
2000	1909.29	1917.59	2511.79	2517.6	2603.84	2613.1	3582.83	3582.72

5.5.4 Computational Results for Parallel Branch-and-bound

We also tested our parallel B&B scheme with accelerated RLT2-DA. The master list was seeded with nodes from $\ell_{init} = 2$, i.e., nodes obtained by fixing the locations of the first two facilities. The maximum dive depth d was set to 5. The RLT2-DA procedure was terminated, in favor of branching, if the optimality gap did not improve by 0.0002 within the last 15 iterations. To calculate an initial upper bound, randomized version of the steepest descent pairwise interchange heuristic was used, which provides descent upper bounds on the QAP (which are optimal in many cases).

For each problem, we noted the total execution time, the number of nodes explored, and the utilization factor of each PE bank. The utilization factor is equal to the ratio of clock time for which a particular PE bank was busy with productive work, such as processing a node, to the total clock time for which the resources were requested. Low utilization indicates that the PE bank spent most of its time in idle state. The results for the B&B tests are shown in Table 5.8. We can see that the number of nodes explored and the completion times increase exponentially with the problem size. The PE bank utilization also increases, because there is more work available for each processor. The number of nodes explored in each problem are comparable to those reported by Adams et al. (2007). Since our parallelization scheme is scalable across multiple GPUs, theoretically, we can obtain RLT2-based lower bounds on QAPs of any size, by simply requesting more GPUs.

Table 5.8: Branch-and-bound results on Nugent problems

Problem	OPT	PE Banks	PEs per bank	Nodes	Total Time (min)	PE Bank Utilization		
						Min	Avg	Max
Nug18	1930	18	1	306	9.26	0.524	0.805	0.986
Nug20†	2570	20	1	216	56.51	0.155	0.549	0.994
Nug22	2650	22	2	462	105.05	0.651	0.729	0.998
Nug25†	3744	200	4	19419	436.12	0.688	0.864	0.975

†Symmetry elimination rules were used for these problem instances.

5.6 Application to M Facility Dominance Procedure

As explained in Section 3.5, the problem of locating M infinitesimal facilities at the corners of M cells, can be formulated as a Quadratic Semi-Assignment Problem (QSAP). Equations (3.13)-(3.15) represent this basic QSAP formulation. The optimal objective value for this problem provides a lower bound on any placement of M finite-size facilities within those M cells, and it can be used to prune the feasible space. Since QSAP is an NP-hard problem, it may not easy to solve it optimally, however, we can certainly convert this QSAP into the following RLT2-based MILP:

$$\text{QSAP-RLT2:} \quad \min \sum_i \sum_p b_{ip} x_{ip} + \sum_i \sum_{j \neq i} \sum_p \sum_q C_{ijpq} y_{ijpq}$$

$$+ \sum_i \sum_{j \neq i} \sum_{k \neq i,j} \sum_p \sum_q \sum_r D_{ijkpqr} z_{ijkpqr}; \tag{5.64}$$

$$\text{s.t.} \quad \sum_p x_{ip} = 1, \qquad \forall i; \tag{5.65}$$

$$\sum_q y_{ijpq} = x_{ip}, \qquad \forall (i \neq j, p); \tag{5.66}$$

$$\sum_r z_{ijkpqr} = y_{ijpq}, \qquad \forall (i \neq j \neq k; p, q); \tag{5.67}$$

$$z_{ijkpqr} = z_{ikjprq} = z_{jikqpr}$$

$$= z_{jkiqrp} = z_{kijrpq} = z_{kjirqp}, \qquad \forall (i < j < k; p, q, r); \tag{5.68}$$

$$x_{ip} \in \{0, 1\}, \qquad \forall (i, p); \tag{5.69}$$

$$y_{ijpq} \geq 0, \qquad \forall (i \neq j; p, q); \tag{5.70}$$

$$z_{ijkpqr} \geq 0, \qquad \forall (i \neq j \neq k; p, q, r). \tag{5.71}$$

We can use our accelerated RLT2-DA to obtain lower bounds on the QSAP-RLT2. In this procedure, the LAP solution stage will be extremely simple because there are only one-sided assignment constraints. These Linear Semi-Assignment Problems (LSAP) can be solved in $O(n^2)$ time by simply scanning for the min-cost matching in each row, and making that assignment. In the parallel framework, one thread can be used to process one row of the matrix and the complexity can be further reduced to $O(n)$. The multiplier update procedure remains the same. The lower bounds obtained for the QSAP-RLT2 can be used as valid lower bounds for eliminating sub-optimal candidates within the subset of m cells. A viable future research direction is to test the efficacy of the QSAP-RLT2 lower bounds for the problem of placing $M \geq 2$ new facilities in a layout with N existing facilities.

5.7 Conclusion

To summarize, we developed a Graphics Processing Units (GPU) based version of the Lagrangian dual ascent procedure (RLT2-DA), for obtaining lower bounds on the RLT2 formulation of the

Quadratic Assignment Problem (QAP), using multiple GPUs in a grid setting. The sequential procedure has two main stages: Linear Assignment Problem (LAP) solution stage and multiplier update stage. In the LAP solution stage, we have to solve $O(n^6)$ LAPs of size $n \times n$, which can be solved independently of each other. We can use our GPU-accelerated Hungarian algorithm to solve a group of LAPs on each GPU, which provides additional speed up. For the multiplier update stage, we leveraged on the fact that each multiplier can be updated independently of the others, and this can be done parallely by a host of CUDA threads. Our main contribution is a novel GPU-based parallelization of the RLT2-DA, in which we used redundant matrices for the complementary cost coefficients. This approach allows us to avoid communicating $O(n^6)$ cost coefficients through MPI, and achieve superior parallel scalability.

We conducted several tests on different variants of our accelerated RLT2-DA procedure, and compared them based on their lower bound strengths, execution times, and "warm start" capability. We concluded that simulated annealing based approaches provide significantly stronger bounds as compared to non-simulated annealing based approaches. Although it is counter-intuitive, simulated annealing based fast 2-phase (F2) variant provides the strongest lower bound of all the variants. Therefore, it is best suited for settings in which the primary goal is to find strong lower bounds on the QAPs. The fast 1-phase (F1) and slow 1-phase (S1) variants play an important role in a branch-and-bound scheme, in which we need to consider the tradeoff between the bound strength and iteration time. The S1 variant allows us to save the dual multipliers, which can be used to jump start the RLT2-DA after branching, which saves quite a lot of iterations. The F1 variant can be used as a "filter" to quickly eliminate easy nodes from further consideration. Our parallel branch-and-bound scheme is able to comfortably solve problems with $n \leq 30$, with the required number of GPUs. We also tested an augmented RLT2-DA procedure by incorporating valid inequalities, which further improves the lower bounds of benchmark problem instances. A comparison with the state-of-the-art SDP-relaxation based method reveals that our parallel RLT2-DA is superior in terms of lower bounds. A part of this credit goes to the RLT2 formulation, for its strong LP-relaxation bounds and ease of decomposition. However, the SDP methods are extremely fast, which might prove to be beneficial in a branch-and-bound scheme. A hybrid method could be proposed as a part of the future work.

The most lucrative feature of our GPU-accelerated algorithm is that it can be implemented on a desktop computer simply equipped with a few gaming graphics cards, to obtain strong lower bounds on comparatively large QAPs. Future directions of research include adaptation of RLT2-DA for solving other difficult problems such as the facility location, graph association, traveling salesman problem, vehicle routing problem, etc.

Chapter 6

Epilogue

In summary, this work addresses both theoretical and computational aspects of the facility-placement problem and computational aspects of assignment problems. These two problems are interconnected in a subtle fashion, in that, a computationally efficient solution procedure for the assignment problems provides means for solving the placement problems in an efficient manner. By far, the theories of facility location and facility layout analyses have largely evolved independently of each other. The theory of finite-size facility placement can be seen a successful unification of these disparate theories. Through this research, we expect to contribute to the scientific community by bringing these two fields of location theory and layout analysis close to each other. In due time, the theory of FPP will yield its benefits to the industry in the form of efficient methods for layout design, which will result in direct savings in operational costs. This unified theory of facility placement will also enrich the theories of facility location and layout, and introduce new challenges, thereby instigating further research in all three fields.

The problem under consideration is a generalization of the Quadratic Assignment Problem (QAP) on the continuous space. The computational advances in the "omnipresent" assignment problems will be extremely valuable to the researchers dealing with large instances of these problems appearing in many other science and engineering applications, including the theories of facility location, facility layout, and facility placement. The parallel and high performance computing methods developed for solving the Linear and the Quadratic Assignment Problems could be adapted for solving large instances of the related problems, such as the Facility Location, Graph Association, Traveling Salesman Problem, Vehicle Routing Problem, etc., which are some of the well-known assignment problems that are difficult to solve.

Bibliography

Adams, W. P., Guignard, M., Hahn, P. M., and Hightower, W. L. (2007). A level-2 reformulation–linearization technique bound for the quadratic assignment problem. *European Journal of Operational Research*, 180(3):983–996.

Adams, W. P. and Johnson, T. A. (1994). Improved linear programming-based lower bounds for the quadratic assignment problem. *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, 16:43–77.

Ahuja, R. K., Magnanti, T. L., and Orlin, J. B. (1993). *Network flows: theory, algorithms, and applications.* Prentice Hall, Upper Saddle River, NJ, USA.

Anstreicher, K., Brixius, N., Goux, J.-P., and Linderoth, J. (2002). Solving large quadratic assignment problems on computational grids. *Mathematical Programming*, 91(3):563–588.

Baker, R. (2011). *CMOS: Circuit Design, Layout, and Simulation.* IEEE Press Series on Microelectronic Systems. Wiley.

Balas, E., Miller, D., Pekny, J., and Toth, P. (1991). A parallel shortest augmenting path algorithm for the assignment problem. *Journal of the ACM (JACM)*, 38(4):985–1004.

Batta, R., Ghose, A., and Palekar, U. (1989). Locating facilities on the Manhattan metric with arbitrarily shaped barriers and convex forbidden regions. *Transportation Science*, 23(1):26–36.

Bazaraa, M. S., Jarvis, J. J., and Sherali, H. D. (2011). *Linear programming and network flows.* John Wiley & Sons.

Bertsekas, D. P. (1990). The Auction algorithm for assignment and other network flow problems: A tutorial. *Interfaces*, 20(4):133–149.

Bertsekas, D. P. and Castañon, D. A. (1991). Parallel synchronous and asynchronous implementations of the Auction algorithm. *Parallel Computing*, 17(6):707–732.

Bertsekas, D. P. and Castañon, D. A. (1993). Parallel asynchronous Hungarian methods for the assignment problem. *ORSA Journal on Computing*, 5(3):261–274.

Blelloch, G. E. (1990). Prefix sums and their applications. Technical report, Synthesis of Parallel Algorithms.

Burkard, R. E. (2002). Quadratic Assignment Problems. *Handbook of combinatorial optimization*, pages 2741–2814.

Burkard, R. E. and Çela, E. (1999). Linear assignment problems and extensions. In Du, D.-Z. and Pardalos, P., editors, *Handbook of Combinatorial Optimization*, pages 75–149. Springer US.

Burkard, R. E., Karisch, S. E., and Rendl, F. (1997). QAPLIB–A Quadratic Assignment Problem library. *Journal of Global Optimization*, 10(4):391–403.

Buš, L. and Tvrdík, P. (2009). Towards Auction algorithms for large dense assignment problems. *Computational Optimization and Applications*, 43(3):411–436.

Castillo, I., Westerlund, J., Emet, S., and Westerlund, T. (2005). Optimization of block layout design problems with unequal areas: A comparison of MILP and MINLP optimization methods. *Computers & Chemical Engineering*, 30(1):54 – 69.

Clausen, J. and Perregaard, M. (1997). Solving large quadratic assignment problems in parallel. *Computational Optimization and Applications*, 8(2):111–127.

Date, K., Makked, S., and Nagi, R. (2014). Dominance rules for the optimal placement of a finite-size facility in an existing layout. *Computers & Operations Research*, 51:182–189.

Date, K. and Nagi, R. (2016). GPU-accelerated Hungarian algorithms for the Linear Assignment Problem. *Parallel Computing*, 57:52–72.

DeMone, P. (2004). The incredible shrinking CPU: Peril of proliferating power.

Dijkstra, E. (1959). A note on two problems in connexion with graphs. *Numerische Mathematik*, 1(1):269–271.

Drezner, Z. (1986). Location of regional facilities. *Naval research logistics quarterly*, 33(3):523–529.

Frieze, A. and Yadegar, J. (1983). On the quadratic assignment problem. *Discrete applied mathematics*, 5(1):89–98.

Geoffrion, A. M. (1974). Lagrangean relaxation for integer programming. In *Approaches to Integer Programming*, pages 82–114. Springer.

Gonçalves, A. D., Pessoa, A. A., de Assumpção Drummond, L. M., Bentes, C., and Farias, R. C. (2015). Solving the quadratic assignment problem on heterogeneous environment (cpus and gpus) with the application of level 2 reformulation and linearization technique. *CoRR*, abs/1510.02065.

Hahn, P. and Grant, T. (1998). Lower bounds for the Quadratic Assignment Problem based upon a dual formulation. *Operations Research*, 46(6):912–922.

Hahn, P. M., Zhu, Y.-R., Guignard, M., Hightower, W. L., and Saltzman, M. J. (2012). A level-3 reformulation-linearization technique-based bound for the quadratic assignment problem. *INFORMS J. on Computing*, 24(2):202–209.

Hakimi, S. (1964). Optimum locations of switching centers and the absolute centers and medians of a graph. *Operations Research*, 12(3):450–459.

Hamacher, H. and Nickel, S. (1995). Restricted planar location problems and applications. *Naval Research Logistics (NRL)*, 42(6):967–992.

Hamacher, H. and Schöbel, A. (1997). A note on center problems with forbidden polyhedra. *Operations Research Letters*, 20(4):165–169.

Harris, M., Sengupta, S., and Owens, J. D. (2007). Parallel prefix sum (scan) with CUDA. *GPU Gems*, 3(39):851–876.

Heragu, S. S. (2008). *Facilities Design*. CRC Press.

Hoberock, J. and Bell, N. (2010). Thrust: A parallel template library. Version 1.7.0.

Huang, E. and Korf, R. E. (2013). Optimal rectangle packing: An absolute placement approach. *Journal of Artificial Intelligence Research*, 46:89–127.

Jonker, R. and Volgenant, A. (1986). Improving the Hungarian assignment algorithm. *Operations Research Letters*, 5(4):171–175.

Jonker, R. and Volgenant, A. (1987). A shortest augmenting path algorithm for dense and sparse linear assignment problems. *Computing*, 38(4):325–340.

Jonker, R. and Volgenant, A. (1999). Linear assignment procedures. *European Journal of Operational Research*, 116(1):233–234.

Juel, H. and Love, R. (1976). An efficient computational procedure for solving the multi-facility rectilinear facilities location problem. *Operations Research Quarterly*, pages 697–703.

Kaufman, L. and Broeckx, F. (1978). An algorithm for the quadratic assignment problem using Bender's decomposition. *European Journal of Operational Research*, 2(3):207–211.

Kelachankuttu, H., Batta, R., and Nagi, R. (2007). Contour line construction for a new rectangular facility in an existing layout with rectangular departments. *European Journal of Operational Research*, 180(1):149–162.

Kennington, J. L. and Wang, Z. (1991). An empirical analysis of the dense assignment problem: Sequential and parallel implementations. *ORSA Journal on Computing*, 3(4):299–306.

King, R. (2007). Averting the IT energy crunch.

Koopmans, T. C. and Beckmann, M. (1957). Assignment problems and the location of economic activities. *Econometrica: Journal of the Econometric Society*, pages 53–76.

Kuhn, H. W. (1955). The Hungarian method for the assignment problem. *Naval Research Logistics Quarterly*, 2(1-2):83–97.

Kusiak, A. and Heragu, S. S. (1987). The facility layout problem. *European Journal of Operational Research*, 29(3):229 – 251.

Larson, R. and Li, V. (1981). Finding minimum rectilinear distance paths in the presence of barriers. *Networks*, 11(3):285–304.

Larson, R. and Sadiq, G. (1983). Facility locations with the Manhattan metric in the presence of barriers to travel. *Operations Research*, 31(4):652–669.

Lawler, E. L. (1963). The quadratic assignment problem. *Management science*, 9(4):586–599.

Lawler, E. L. (1976). *Combinatorial optimization: networks and matroids*. Courier Corporation.

Loiola, E. M., De Abreu, N. M. M., Boaventura-Netto, P. O., Hahn, P., and Querido, T. (2007). A survey for the quadratic assignment problem. *European Journal of Operational Research*, 176(2):657–690.

Love, R. and Kraemer, S. (1973). A dual decomposition method for minimizing transportation costs in multifacility location problems. *Transportation Science*, 7(4):297–316.

Love, R. and Morris, J. (1975). Technical note–solving constrained multi-facility location problems involving lp distances using convex programming. *Operations Research*, 23(3):581–587.

Love, R. and Yerex, L. (1976). An application of a facilities location model in the prestressed concrete industry. *Interfaces*, 6(4):45–49.

Luo, L., Wong, M., and Hwu, W.-m. (2010). An effective GPU implementation of breadth-first search. In *Proceedings of the 47th Design Automation Conference*, DAC '10, pages 52–55, New York, NY, USA. ACM.

Meller, R. D. and Gau, K.-Y. (1996). The facility layout problem: Recent and emerging trends and perspectives. *Journal of Manufacturing Systems*, 15(5):351 – 366.

Meller, R. D., Narayanan, V., and Vance, P. H. (1998). Optimal facility layout design. *Operations Research Letters*, 23(3–5):117 – 127.

Merrill, D., Garland, M., and Grimshaw, A. (2012). Scalable GPU graph traversal. In *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '12, pages 117–128, New York, NY, USA. ACM.

Montreuil, B. (1991). A modelling framework for integrating layout design and flow network design. In *Material Handling '90*, volume 2 of *Progress in Material Handling and Logistics*, pages 95–115. Springer Berlin Heidelberg.

Munkres, J. (1957). Algorithms for the assignment and transportation problems. *Journal of the Society for Industrial & Applied Mathematics*, 5(1):32–38.

Naiem, A. and El-Beltagy, M. (2009). Deep greedy switching: A fast and simple approach for linear assignment problems. In *7th International Conference of Numerical Analysis and Applied Mathematics.*

Naiem, A., El-Beltagy, M., and Rasmy, M. (2010). A centrally coordinated parallel Auction algorithm for large scale assignment problems. In *Informatics and Systems (INFOS), 2010 The 7th International Conference on*, pages 1–4. IEEE.

Nering, E. D. and Tucker, A. W. (1993). *Linear Programs and Related Problems*, volume 1. Access Online via Elsevier.

NVIDIA (2012). *CUDA C Programming Guide 5.0.* NVIDIA Corporation.

Pardalos, P. M. and Crouse, J. V. (1989). A parallel algorithm for the quadratic assignment problem. In *Proceedings of the 1989 ACM/IEEE conference on Supercomputing*, pages 351–360. ACM.

Peng, J., Zhu, T., Luo, H., and Toh, K.-C. (2015). Semi-definite programming relaxation of quadratic assignment problems based on nonredundant matrix splitting. *Computational Optimization and Applications*, 60(1):171–198.

Pitsoulis, L. (2009). *Encyclopedia of Optimization*, chapter Quadratic semi-assignment problem, pages 3170–3171. Springer US, Boston, MA.

Ramachandran, B. and Pekny, J. (1996). Dynamic matrix factorization methods for using formulations derived from higher order lifting techniques in the solution of the quadratic assignment problem. *Nonconvex Optimization and its Applications*, 7:75–92.

Ramakrishnan, K., Resende, M., Ramachandran, B., and Pekny, J. (2002). Tight QAP bounds via linear programming. *Combinatorial and Global Optimization*, pages 297–303.

Roucairol, C. (1987). A parallel branch and bound algorithm for the quadratic assignment problem. *Discrete Applied Mathematics*, 18(2):211 – 225.

Roverso, R., Naiem, A., El-Beltagy, M., El-Ansary, S., and Haridi, S. (2010). A GPU-enabled solver for time-constrained linear sum assignment problems. In *Informatics and Systems (INFOS), 2010 The 7th International Conference on*, pages 1–6. IEEE.

Sahni, S. and Gonzalez, T. (1976). P-complete approximation problems. *Journal of the ACM (JACM)*, 23(3):555–565.

Sarkar, A., Batta, R., and Nagi, R. (2005). Planar area location/layout problem in the presence of generalized congested regions with the rectilinear distance metric. *IIE Transactions*, 37(1):35–50.

Sarkar, A., Batta, R., and Nagi, R. (2007). Placing a finite size facility with a center objective on a rectangular plane with barriers. *European Journal of Operational Research*, 179(3):1160–1176.

Sarrafzadeh, M., Wang, M., and Yang, X. (2003). *Modern Placement Techniques.* Springer Science & Business Media.

Sathe, M., Schenk, O., and Burkhart, H. (2012). An Auction-based weighted matching implementation on massively parallel architectures. *Parallel Computing*, 38(12):595 – 614.

Savaş, S., Batta, R., and Nagi, R. (2002). Finite-size facility placement in the presence of barriers to rectilinear travel. *Operations Research*, 50(6):1018–1031.

Sengupta, S., Lefohn, A. E., and Owens, J. D. (2006). A work-efficient step-efficient prefix sum algorithm. In *Proceedings of the Workshop on Edge Computing Using New Commodity Architectures*, pages 26–27.

Sherali, H. D., Fraticelli, B. M. P., and Meller, R. D. (2003). Enhanced model formulations for optimal facility layout. *Operations Research*, 51(4):629–644.

Sherwani, N. (1999). *Algorithms for VLSI physical design automation.* Springer Science & Business Media.

Storøy, S. and Sørevik, T. (1997). Massively parallel augmenting path algorithms for the assignment problem. *Computing*, 59(1):1–16.

Vasconcelos, C. N. and Rosenhahn, B. (2009). Bipartite graph matching computation on GPU. In *Energy Minimization Methods in Computer Vision and Pattern Recognition*, pages 42–55. Springer.

Volgenant, A. (1996). Linear and semi-assignment problems: a core oriented approach. *Computers & Operations Research*, 23(10):917–932.

Wang, S., Bhadury, J., and Nagi, R. (2002). Supply facility and input/output point locations in the presence of barriers. *Computers & Operations Research*, 29(6):685–699.

Wein, J. M. and Zenios, S. (1990). Massively parallel Auction algorithms for the assignment problem. In *Frontiers of Massively Parallel Computation, 1990. Proceedings., 3rd Symposium on the*, pages 90–99. IEEE.

Wesolowsky, G. and Love, R. (1971). Location of facilities with rectangular distances among point and area destinations. *Naval Research Logistics Quarterly*, 18(1):83–90.

Zhang, M., Savas, S., Batta, R., and Nagi, R. (2009). Facility placement with sub-aisle design in an existing layout. *European Journal of Operational Research*, 197(1):154 – 165.