Branch and Bound via the Alternating Direction Method of Multipliers for the Quadratic Assignment Problem

by

Zhenyu (Alister) Liao

A research paper
presented to the University of Waterloo
in partial fulfillment of the
requirement for the degree of
Master of Mathematics
in
Computational Mathematics

Supervisor: Prof. Henry Wolkowicz

Waterloo, Ontario, Canada, 2016

Abstract

The quadratic assignment problem (**QAP**) is one of the most difficult NP-hard problems, and so far there is no efficient exact algorithm that can solve large **QAP** instances in a reasonable amount of time. Prior studies have shown that the semidefinite programming (**SDP**) relaxation is able to provide extremely tight lower bound to the **QAP**. Preliminary results suggest that the alternating method of multipliers (**ADMM**) can solve the **SDP** relaxation much faster than the previous primal-dual interior-point method, which has difficulty solving the **SDP** relaxation efficiently and accurately. Such promising results motivated us to implement a branch and bound algorithm, the most successful exact algorithm for the **QAP**, based on the **ADMM** method for the **SDP** relaxation.

Our empirical results show a dramatic reduction in the number of visited nodes by either breadth first search or depth first search in our branch and bound algorithm. This reduction demonstrates the effectiveness of the **ADMM** method for the **SDP** relaxation, and enables us to apply our algorithm to large **QAP** instances. In fact, our results are comparable to some of the best existing branch and bound algorithms using different bounds, indicating the potential of a new efficient and robust exact algorithm for the **QAP**.

Acknowledgments

First, I would like to express my gratitude to my supervisor, Prof. Henry Wolkowicz, for his support to my study and research. Thank you for guiding me throughout the research and the writing of this paper with patience, enthusiasm, and broad knowledge. This work is not possible without your dedicated contribution. My thanks also go to my second reader, Prof. Thomas F. Coleman, for his nice comments regarding this paper.

I thank all my friends in the Computational Mathematics program. Together we have accomplished so much in such a short period! It is my great honor to have known all of the following inspiring individuals: Nathan Braniff, Gurpreet Gosal, Yueshan He, Xinghang Ye, Zhongheng Yuan, Xiaochen Zhang, and Xixuan Yu.

Last but not the least, I would like to thank the Administrative Coordinator, Stephanie Martin, for hosting fantastic group gatherings every month, and the Director, Kevin Hare, for his help and advice throughout the entire program.

Dedication

To my beloved parents, Ling Cheng and Xianmin Liao

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

In this paper, we study the quadratic assignment problem (**QAP**), and propose a new branch and bound algorithm that is efficient and able to solve large **QAP** instances. The **QAP** was introduced by Koopmans and Beckmann in [18] to model the facility location problem. A **QAP** instance usually contains the flows between n facilities, the distances between n locations, and the costs of putting facilities at specific locations. The task is to find the best planning such that the sum of flows multiplied by distances plus location costs is minimized. It is well known that solving the **QAP** itself and finding an ϵ-approximation are both NP-hard problems. The detailed proof can be found in [9]. Problems with size $n = 30$ are still considered hard to solve as Burkard et al. commented in [8], "All main algorithm techniques for the exact solution of NP-hard problems have been used for attacking the **QAP**: decomposition, branch-and-bound, and branch-and-cut. The **QAP**, however, bravely resisted."

Recent progress in exact algorithms is largely due to branch and bound approaches [9]. However, the performance of such method depends on efficient bounding procedures. The trade off between a tight bound and fast calculation has been the most intriguing topic in this field. Among all relaxations, the semidefinite programming (**SDP**) relaxation for the **QAP** proposed in [29] has been verified to provide competitive or even best bounds for a few **QAPLIB** [7] instances such as the Hadley [17] problems and the Taillard [28] problems [9]. Nevertheless, the popular approach, namely the primal-dual interior-point method, has difficulty solving large problems and obtaining high accuracy results. Therefore, implementing branch and bound based on such an approach would fail inevitably in terms of computation time, despite the benefit of getting a tight lower bound from the **SDP** relaxation.

Some weaknesses of the above method have been addressed in a new study of the alternating direction method of multipliers (**ADMM**) for solving the **SDP** relaxation. By adding non-negativity constraint, they manage to show that **ADMM** can obtain high accuracy results in a significantly less amount of time than the p-d, i-d approach. In fact, the new method provides the best known lower bound for almost all test instances from **QAPLIB** [24]. The promising results from their study motivate us to implement a branch and bound algorithm based on the new **ADMM** approach.

The **ADMM** method is updated in this paper to include a new stopping criterion that takes advantege of the branch and bound setting. Several combinations of the maximum number of iterations and the tolerance are tested in a grid search for optimal branch and bound performance. The empirical results in Chapter 4 suggest that our new branch and bound algorithm is able to take advantage of the new **ADMM** bounding strategy, and provides a possible way of solving large **QAP** instances. In particular, the number of visited nodes is dramatically reduced from previous branch and bound algorithms using classical bounds and is comparable to the best existing algorithms in [1, 3, 4].

The rest of this chapter includes a detailed introduction to the **QAP**, branch and bound, and the **ADMM**. Many mathematical operators and definitions used by this paper is introduced in Section 1.4. We provide a derivation of the new **ADMM** algorithm for the **SDP** relaxation in Chapter 2, which is the underlying theoretical motivation for this paper. Then we introduce our new branch and bound algorithm in Chapter 3, in which we define our node selection strategy in Section 3.1, branching strategy in Section 3.2, and our full algorithm in Section 3.4. Chapter 4 provides implementation details and numerical results of our mew branch and bound algorithm. Chapter 5 gives a brief summary of our conclusions and some potential improvements of our algorithm.

1.1 The Quadratic Assignment Problem

The earliest formulation of the **QAP** by Koopmans and Beckmann [18] is

$$\min_{p\in\Pi} \sum_{i=1}^{n}\sum_{j=1}^{n} f_{ij} d_{p(i)p(j)} + \sum_{i=1}^{n} c_{ip(i)}, \tag{1.1}$$

where p is a permutation of $\{1 \dots n\}$, f_{ij} is the "flow" from facility i to facility j, d_{kl} is the "distance" from location k to l, and c_{ik} is the "fixed cost" of putting facility i in position k. If we simultaneously assign facility i to location k and facility j to location l, we incur the cost $f_{ij}d_{kl} + f_{ji}d_{lk} + c_{ik} + c_{jl}$. If all c's are zeros or equal, as in many instances in the **QAPLIB**, we call the problem the homogeneous **QAP**.

The original settings in [18] lead to the famous facility location problem. Dickey and Hopkins proposed a similar campus planning problem in [12]. Another famous engineering problem formulated as **QAP** is the turbine balancing problem [8]. The goal of that problem is to minimize the distance between the center of gravity and the axis of the runner, which has also been proved to be NP-hard. The most renowned combinatorial optimization problem, the traveling salesman problem (**TSP**), can also be formulated as a **QAP**. In this formulation, $D = (d_{kl})$ is the distances between cities and $F = (F_{ij})$ forms a cyclic unit flow, e.g., $F = I$. Then the Koopmans-Beckmann formulation for the **TSP** can be written as

$$\min_{p\in\Pi} \sum_{i=1}^{n}\sum_{j=1}^{n} f_{ij} d_{p(i)p(j)}. \tag{1.2}$$

This result can be used to prove that the **QAP** is an NP-hard problem since it contains the **TSP** as a special case. Some other applications of the **QAP**, including computer manufacturing, scheduling, parallel and distributed computing, and statistical data analysis, just to name a few, can be found in [8, 9, 26].

In this paper, we consider the equivalent trace formulation of the **QAP** [13] with symmetric and 0-diagonal flow and distance matrices:

$$p_X^* := \min_{X\in\Pi_n} \operatorname{tr}\left(FXDX^\top + 2CX^\top\right), \tag{1.3}$$

where $F = (f_{ij}), D = (d_{kl}) \in \mathbb{S}^n$ are real symmetric $n \times n$ matrices, $C = (c_{ik})$ is a real $n \times n$ matrix, and Π_n denotes the set of $n \times n$ permutation matrices. This formulation was used to derive the eigenvalue related bound in [14], which is still one of the tightest lower bounds for the **QAP**. For other formulations of the **QAP**, see [8, 9, 26].

Example 1.1. Suppose we have the following **QAP** instance.

$$F = \begin{bmatrix} 0 & 2 & 3 & 4 \\ 2 & 0 & 7 & 8 \\ 3 & 7 & 0 & 12 \\ 4 & 8 & 12 & 0 \end{bmatrix}, \quad D = \begin{bmatrix} 0 & 15 & 14 & 13 \\ 15 & 0 & 10 & 9 \\ 14 & 10 & 0 & 5 \\ 13 & 9 & 5 & 0 \end{bmatrix}, \quad \text{and } C = \begin{bmatrix} 16 & 15 & 14 & 13 \\ 9 & 10 & 11 & 12 \\ 8 & 7 & 6 & 5 \\ 1 & 2 & 3 & 4 \end{bmatrix}.$$

Given a permutation matrix, $X = \begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$, we can compute the corresponding objective function by (1.1). The assignments here are facility 1 to location 2, facility 2 to location 3, facility 3 to location 1, and facility 4 to location 4. The objective value is

$$\begin{aligned} p^* = {} & (2\times 10 + 3\times 15 + 4\times 9) + (2\times 10 + 7\times 14 + 8\times 5) + (3\times 15 + 7\times 14 + 12\times 13) \\ & + (4\times 9 + 8\times 5 + 12\times 13) + 2\times(15 + 11 + 8 + 4) = 866. \end{aligned}$$

Alternatively, we compute the objective function by (1.3).

$$\begin{aligned} p^*_X = {} & \operatorname{tr}\left(\begin{bmatrix} 0 & 2 & 3 & 4 \\ 2 & 0 & 7 & 8 \\ 3 & 7 & 0 & 12 \\ 4 & 8 & 12 & 0 \end{bmatrix}\begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}\begin{bmatrix} 0 & 15 & 14 & 13 \\ 15 & 0 & 10 & 9 \\ 14 & 10 & 0 & 5 \\ 13 & 9 & 5 & 0 \end{bmatrix}\begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}^\top\right) \\ & + 2\operatorname{tr}\left(\begin{bmatrix} 16 & 15 & 14 & 13 \\ 9 & 10 & 11 & 12 \\ 8 & 7 & 6 & 5 \\ 1 & 2 & 3 & 4 \end{bmatrix}\begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}^\top\right) \\ = {} & \operatorname{tr}\left(\begin{bmatrix} 101 & 62 & 80 & 49 \\ 177 & 158 & 134 & 109 \\ 178 & 90 & 299 & 62 \\ 260 & 208 & 172 & 232 \end{bmatrix} + \begin{bmatrix} 30 & 28 & 32 & 26 \\ 20 & 22 & 18 & 24 \\ 14 & 12 & 16 & 10 \\ 4 & 6 & 2 & 8 \end{bmatrix}\right) \\ = {} & 101 + 158 + 299 + 232 + 30 + 22 + 16 + 8 = 866. \end{aligned}$$

The results are consistent with the fact that the trace formulation (1.3) is equivalent to the Koopmans-Beckmann formulation (1.1).

The optimal value of this example is 724 with $X^* = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix}$. □

It is well known that the set of permutation matrices Π_n is equivalent to the following two intersections [26]

$$\Pi_n = \mathcal{O}_n \cap \mathcal{E}_n \cap \mathcal{N}_n = \mathcal{Z}_n \cap \mathcal{E}_n \cap \mathcal{O}_n, \tag{1.4}$$

where $\mathcal{O}_n = \{X \in \mathbb{R}^{n\times n} : X^\top X = I\}$ is the set of orthogonal matrices, $\mathcal{Z}_n = \{X : X_{ij}^2 - X_{ij} = 0, \forall i,j = 1,\ldots,n\}$ is the set of $(0,1)$ matrices, $\mathcal{E}_n = \{X \in \mathbb{R}^{n\times n} : Xe = X^\top e = e\}$ is the set of doubly stochastic matrices whose row and column sums equal to one, and $\mathcal{N}_n = \{X \in \mathbb{R}^{n\times n} : X \geq 0\}$ is the set of non-negative matrices.

In the next chapter, we derive the **SDP** relaxation of the **QAP** as shown in [24, 29], starting with the intersection of those sets.

1.2 Branch and Bound

Even though 54 years have passed since the branch and bound for the **QAP** was first introduced by Gilmore in [16], this algorithm remains the most effective exact algorithm for the **QAP** [9]. Nearly all successful methods for the **QAP** can be categorized as branch and bound algorithms [8]. It is a tree-based model wherein each node represents a subproblem generated in the branching phase. The upper and lower bound of the subprobelm are calculated according to some relaxation in the bounding phase. The algorithm then decides whether to prune the current branch or subdivide the problem even further by assessing the lower bound.

In the branching phase, the algorithm needs to select a subproblem to process and then subdivide the problem into even smaller subproblems. The selection strategies have been explored for many years, including problem-independent, instance related branching, max lower bound branching, and min number of nodes strategies. However, there is no clear winner in practice [8].

The oldest and probably the simplest branching rule is single assignment branching proposed by Gilmore in [16]. This rule is still the most efficient one, and some development has been proposed on node selection methods based on different bounding strategies. As the name suggests, this method allocates an unassigned facility i to an available location j according to some selection rule. Then two branches (leaves) of the node (root) are created – one assigning i to j (a) and another preventing the assignment (u). The subproblem at (a) is a smaller **QAP**, whereas the one at (u) is the original **QAP** with some modification to prevent the assignment. This modification can be easily done by bumping $C(i, j)$ up to a large number and thus making the assignment i to j come with an unbearable fixed cost.

The pair assignment branching puts two facilities at two locations each time, but its performance in practice is not as good as the single one. Mirchandani proposed the relative positioning method in [21], in which the partial permutation at each node is determined by distances between facilities instead of assignments. Empirical results show that this method may be appropriate for sparse **QAP**s, but in general single assignment branching still outperforms this method [8, 26].

In addition to the above three classic branching rules, Roucairol proposed another powerful branching rule called polytomic or k-partite branching [27]. This method assigns an available facility to all possible locations at one level, or alternatively, uses an available location to host all possible facilities. For a **QAP** of size n, if we choose facility i in the beginning, then the method creates n branches, in which i is assigned to location $1, 2, \ldots, n$ respectively. The size of all subprobelms at this level is $n - 1$, and thus those problems are easier to solve. The depth of the branching tree is at most $n-1$ and the number of nodes is at most $\sum_{k=1}^{n-1} \frac{n!}{(n-k)!}$. The branching tree created in this way is naturally suited for parallel computing since all branches of a level can be computed independently and pruned after all computation for one level is done. In fact, the parallel computing implementation of the polytomic branching solved nug30, a Nugent probelm [22] of size 30, for the first time in 2000 [6].

The success of any branch and bound algorithm depends on its bounding technique, especially the effectiveness of the lower bound. The calculation of the lower bound often involves solving relaxations of the original **QAP**. The ideal lower bound should be tight but computationally inexpensive. However, such bound rarely exists in practice, resulting in the failure of many branch and bound algorithms [29]. Due to computation limits, the Gilmore-Lawler bound (**GLB**) have been the only bound implemented in branch and bound algorithms for a long time [9]. The **GLB** is efficient for **QAP** instances of size up to 24, but the growth rate of the branching tree, together with the poor quality of the bound, eventually eats up the computational advantage. Therefore, it is impossible to use **GLB** in a branch and bound algorithm to solve large **QAP** instances such as nug27/28/30 [2]. Many new bounds have been proposed since the **GLB** came out, including eigenvalue based, linear programming based, and reformulation based bounds. For a survey of the lower bounds, see [2, 9].

In this paper, we use the lower bounds from the **SDP** relaxation. This type of bounds have been verified to be one of the tightest bounds up to date, but it is generally too expensive for branch and bound algorithms [9]. Recently, this disadvantage is addressed in [24], in which Oliveira et al. proposed a new **ADMM** method for calculating the lower bounds. Their empirical tests show that the computation time has been dramatically reduced comparing to the original interior point method. This improvement makes it possible for us to implement a branch and bound algorithm based on their new method.

The last step in a branch and bound algorithm is pruning. We use the global minimum upper bound (*the incumbent value*) as the cutoff point, which are updated in the bounding phase. Then any nodes with their lower bounds greater than the incumbent solution are deleted. To speed up branch and bound algorithm, we would want to prune a branch as early as possible, and that is why the tightness of the lower bounds are crucial in the algorithm. In general, as the size of the problem gets larger, it is impossible to visit all nodes within a reasonable time frame.

Among the existing implementations, parallel computing implementation seems to be most successful in recent development [9]. nug30 was first solved with a computing grid of more than 1000 machines over a one-week period, witch is equivalent to seven years of computation on a single fast workstation [6]. Even with the improvements done in [1], nug30 still requires 2.5 CPU years to solve. Although hardware advances play an important role behind the solution of large **QAP** instances, we want to point out that the improvement is not possible without good new algorithms.

1.3 The Alternating Direction Method of Multipliers

The **ADMM** was first introduced in the mid-1970s with some early ideas emerged in the mid-1950s. Most theoretical results for this method have already been established in the last century. However, **ADMM** remained in theory for a long time due to hardware constraints. Recent improvements in parallel and distributed computing systems have brought **ADMM** back to light. We introduce the basic idea of the **ADMM** as shown in [5] using the following minimization problem.

Example 1.2.

$$\begin{aligned} &\min_{x,y}\ f(x) + g(y) \\ &\text{s. t.}\ \ Ax + By = c \end{aligned} \tag{1.5}$$

where $x \in \mathbb{R}^n, y \in \mathbb{R}^m, A \in \mathbb{R}^{k\times n}, B \in \mathbb{R}^{k\times m}$ and $c \in \mathbb{R}^k$. We further assume that f and g are convex. Next, we introduce a multiplier $z \sim Ax + By = c$. Then we can write the augmented Lagrangian for (1.5) as

$$\mathcal{L}_\rho(x, y, z) = f(x) + g(y) + z^\top(Ax + By - c) + \frac{\rho}{2}||Ax + By - c||^2$$

The **ADMM** then performs the following update iteratively:

$$\begin{aligned} x_+ &= \arg\min_x\ \mathcal{L}_\rho(x, y, z), \\ y_+ &= \arg\min_y\ \mathcal{L}_\rho(x_+, y, z), \\ z_+ &= z + \rho(Ax_+ + By_+ - c). \end{aligned}$$

□

As we can see above, The **ADMM** borrows some ideas from dual ascent method and the method of multipliers. However, it is different from the other two methods that the two primal variables, x and y, are updated in an alternating fashion. While the joint minimization may be very difficult, the separate update is often much easier to carry out. It is also worth mentioning that **ADMM** does need separable objective function. we show in Section 2.2 that we can manipulate the constraints if the original function is non-separable. For a survey of the **ADMM**, see [5].

1.4 Preliminaries

We denote the set of $n \times n$ symmetric matrices by $\mathbb{S}^n$; symmetric positive semidefinite matrices by $\mathbb{S}^n_+$; symmetric positive definite matrices by $\mathbb{S}^n_{++}$. Π is the set of permutations, and Π_n is the set of $n \times n$ permutation matrices. For $X \in \mathbb{R}^{n\times n}$, $\operatorname{vec}(X) \in \mathbb{R}^{n^2}$ is a vector constructed by stacking the columns of X, and $\operatorname{diag}(X) \in \mathbb{R}^n$ is the vector of the diagonal entries of X. For $x \in \mathbb{R}^{n^2}$, $\operatorname{Mat}(x) \in \mathbb{R}^{n\times n}$ is a matrix whose columns are made up of every n elements in x, and $\operatorname{Diag}(x) \in \mathbb{R}^{n\times n}$ is a diagonal matrix formed from the corresponding entries of x.

Definition 1.1. Let $\mathbb{E}$ and $\mathbb{F}$ denote some Hilbert spaces. Let $\mathcal{A} : \mathbb{E} \to \mathbb{F}$ be a linear mapping between the two spaces. The *adjoint operator*, $\mathcal{A}^* : \mathbb{F} \to \mathbb{E}$, is defined such that

$$\langle \mathcal{A}x, y\rangle = \langle x, \mathcal{A}^* y\rangle \qquad \forall x \in \mathbb{E}, y \in \mathbb{F},$$

where $\langle \cdot, \cdot \rangle$ is the inner product. ■

Note that $\operatorname{vec}(\cdot)$ and $\operatorname{Mat}(\cdot)$ are adjoint operators to each other, as well as $\operatorname{diag}(\cdot)$ and $\operatorname{Diag}(\cdot)$. As we have defined previously, $\mathcal{O}_n = \{X \in \mathbb{R}^{n\times n} : X^\top X = I\}$ is the set of orthogonal matrices, $\mathcal{Z}_n = \{X : X_{ij}^2 - X_{ij} = 0, \forall i, j = 1, \dots, n\}$ is the set of $(0, 1)$ matrices, $\mathcal{E}_n = \{X \in \mathbb{R}^{n\times n} : Xe = X^\top e = e\}$ is the set of doubly stochastic matrices whose row and column sums equal to one, and $\mathcal{N}_n = \{X \in \mathbb{R}^{n\times n} : X \geq 0\}$ is the set of non-negative matrices. $X \circ Y$ is the Hadamard product, i.e., element-wise product, of two matrices of the same size.

The trace of a square matrix X, $\operatorname{tr}(X)$, is the sum of the elements on the main diagonal of X. We use the following properties of the trace operator in our paper.

- $\operatorname{tr}(ABCD) = \operatorname{tr}(BCDA) = \operatorname{tr}(CDAB) = \operatorname{tr}(DABC)$
- $\operatorname{tr}(Y^\top X) = \operatorname{tr}(X^\top Y) = \sum_{i,j} X_{ij}Y_{ij} = \sum_{i,j}(X \circ Y)_{ij}$
- $\langle X, Y\rangle = \operatorname{tr}(Y^\top X)$

- $\mathrm{tr}(X^\top Y) = \mathrm{vec}(Y)^\top \mathrm{vec}(X) = \mathrm{vec}(X)^\top \mathrm{vec}(Y)$

Suppose $X \in \mathbb{R}^{m\times n}$ and $Y \in \mathbb{R}^{p\times q}$. The Kronecker product of those two matrices, denoted $X \otimes Y$, is defined as:

$$X \otimes Y = \begin{bmatrix} X_{11}Y & \cdots & X_{1n}Y \\ \vdots & \ddots & \vdots \\ X_{m1}Y & \dots & X_{mn}Y \end{bmatrix}.$$

We use the following properties of the Kronecker product in our paper.

- $(A \otimes B)(X \otimes Y) = AX \otimes BY$
- $\mathrm{vec}(FXD) = (D^\top \otimes F)\,\mathrm{vec}(X)$
- $(X \otimes Y)^\top = X^\top \otimes Y^\top$
- $\mathrm{tr}(X \otimes Y) = \mathrm{tr}(X)\,\mathrm{tr}(Y)$

For $Y \in \mathbb{S}$, $Y \succeq 0$ refers to the Löwner partial order, that is, Y is positive semidefinite. Similarly, $Y \preceq 0$ indicates Y is negative semidefinite. The usual $Y \geq 0$ is used to suggest all entries of Y are non-negative.

Chapter 2

The New ADMM Algorithm for the QAP

In this chapter, we introduce a new bounding strategy for the **QAP** based on the **SDP** relaxation and the **ADMM** algorithm in [24]. This strategy relaxes the searching space from Π_n to $\mathbb{S}_+^{n^2+1}$, and then reduces to $\mathbb{S}_+^{(n-1)^2+1}$ after facial reduction. If the solution of the **SDP** relaxation is also in the original search space, then the original **QAP** is solved. Since the relaxation usually does not solve the original **QAP**, we get a lower bound from solving the relaxation and an upper bound from approximating the closest feasible solution.

An efficient and tight lower bound is crucial to the success of any branch and bound algorithm. The **SDP** relaxation related bounds in [29] have been shown to be among the best existing bounds. However, it is not suitable for a branch and bound method when solved using the primal-dual interior point method. In Section 2.1, we derive the **SDP** relaxation following a similar procedure as shown in [29]. Then in Section 2.2, we introduce the new **ADMM** algorithm. The numerical experiments in [24] have shown that this new algorithm is much faster than the p-d i-d approach, while still maintains decent accuracy and tightness. Finally, the last two sections are dedicated to the actual lower and upper bound used in our branch and bound algorithm.

We make extensive use of mathematical operators and symbols in this chapter. Please refer to Section 1.4 for notations and definitions.

2.1 A Derivation for the SDP Relaxation

We start with the trace formulation (1.3) and replace the set of permutation matrices using (1.4).

$$\begin{aligned} p_X^* := \min_X \;& \operatorname{tr}(FXDX^\top + 2CX^\top) && \\ \text{s.t. } & XX^\top = X^\top X = I && (\mathcal{O}) \\ & \|Xe - e\|^2 + \|X^\top e - e\|^2 = 0 && (\mathcal{E}) \\ & X_{ij}^2 - X_{ij} = 0, \; \forall i,j && (\mathcal{Z}) \end{aligned} \tag{2.1}$$

Then we introduce the following Lagrange multipliers:

$$\begin{aligned} S_b &\sim XX^\top = I, \\ S_o &\sim X^\top X = I, \\ u_0 &\sim \|Xe - e\|^2 + \|X^\top e - e\|^2 = 0, \\ W_{ij} &\sim X_{ij}^2 - X_{ij} = 0, \; \forall i,j. \end{aligned}$$

The Lagrangian for (2.1) is

$$\begin{aligned} \mathcal{L}_0(X, S_b, S_o, u_0, W) = & \operatorname{tr}(FXDX^\top + 2CX^\top) + S_b(XX^\top - I) + S_o(X^\top X - I) \\ & + u_0(\|Xe - e\|^2 + \|X^\top e - e\|^2) + \sum_{i=1}^n \sum_{j=1}^n W_{ij}(X_{ij}^2 - X_{ij}) \\ = & \operatorname{tr}(FXDX^\top + 2CX^\top) + S_b(XX^\top - I) + S_o(X^\top X - I) \\ & + u_0(\|Xe\|^2 + \|X^\top e\|^2 - 2e^\top(X + X^\top)e + 2n) \\ & + W[(X \circ X)^\top - X^\top]. \end{aligned}$$

Next, we homogenize the X terms in $\mathcal{L}_0$ by x_0 s. t. $x_0^2 = 1$ and introduce another multiplier $\omega_0 \sim x_0^2 = 1$. Then we can rewrite $\mathcal{L}_0$ as

$$\begin{aligned} \mathcal{L}_1(X, S_b, S_o, u_0, W, \omega_0) = & \operatorname{tr}(FXDX^\top) + S_b XX^\top + S_o X^\top X + u_0(\|Xe\|^2 + \|X^\top e\|^2) \\ & + W(X \circ X)^\top + \omega_0 x_0^2 - \operatorname{tr}(S_b) - \operatorname{tr}(S_o) - 2x_0 u_0 e^\top(X + X^\top)e \\ & + 2x_0^2 u_0 n - \operatorname{tr}[x_0(W - 2C)X^\top] - \omega_0. \end{aligned}$$

Let $y = \begin{bmatrix} x_0 \\ \operatorname{vec}(X) \end{bmatrix} \in \mathbb{R}^{n^2+1}$ and $\omega = \begin{bmatrix} \omega_0 \\ \operatorname{vec}(W) \end{bmatrix} \in \mathbb{R}^{n^2+1}$. Then

$$\begin{aligned} \mathcal{L}_2(y, S_b, S_o, u_0, W, \omega) \; = \; & \operatorname{tr}\left(y^\top \left(L_Q + \mathrm{B}^0 \operatorname{Diag}(S_b) + \mathrm{O}^0 \operatorname{Diag}(S_o) + u_0 K + \operatorname{Arrow}(\omega)\right) y\right) \\ & -\omega_0 - \operatorname{tr} S_b - \operatorname{tr} S_o, \end{aligned}$$

where

$$
\begin{aligned}
L_Q &= \begin{bmatrix} 0 & \operatorname{vec}(C)^\top \\ \operatorname{vec}(C) & D \otimes F \end{bmatrix}, \\
B^0 \operatorname{Diag}(S_b) &= \begin{bmatrix} 0 & 0 \\ 0 & I \otimes S_b \end{bmatrix}, \\
O^0 \operatorname{Diag}(S_o) &= \begin{bmatrix} 0 & 0 \\ 0 & S_o \otimes I \end{bmatrix}, \\
\mathit{Arrow}(\omega) &= \begin{bmatrix} \omega_0 & -\frac{1}{2}\omega_{1:}^\top \\ -\frac{1}{2}\omega_{1:} & \operatorname{Diag}(\omega_{1:}) \end{bmatrix}, \\
K &= \begin{bmatrix} n & -e^\top \otimes e^\top \\ -e \otimes e & I \otimes E \end{bmatrix} + \begin{bmatrix} n & -e^\top \otimes e^\top \\ -e \otimes e & E \otimes I \end{bmatrix}.
\end{aligned}
$$

The dual problem of (2.1) can then be written using $\mathcal{L}_2$:

$$
\begin{aligned}
d_y^* &:= \max_{S_b, S_o, u_0, W, \omega} \min_y \mathcal{L}_2(y, S_b, S_o, u_0, W, \omega) \\
&:= \max_{S_b, S_o, \omega_0} -\omega_0 - \operatorname{tr} S_b - \operatorname{tr} S_o \\
&\quad \text{s. t. } L_Q + \mathrm{B}^0 \operatorname{Diag}(S_b) + \mathrm{O}^0 \operatorname{Diag}(S_o) + u_0 K + \operatorname{Arrow}(\omega) \succeq 0.
\end{aligned}
\tag{2.2}
$$

Now, we introduce a new matrix of multipliers Y for the only constraint in (2.2), and write yet another Lagrangian as:

$$
\begin{aligned}
\mathcal{L}_3(Y, S_b, S_o, u_0, W, \omega) = Y^\top \big(&L_Q + \mathrm{B}^0 \operatorname{Diag}(S_b) + \mathrm{O}^0 \operatorname{Diag}(S_o) + u_0 K + \operatorname{Arrow}(\omega)\big) \\
&- \omega_0 - \operatorname{tr} S_b - \operatorname{tr} S_o.
\end{aligned}
$$

We can now obtain the final **SDP** relaxation by taking the dual of (2.2):

$$
p_{R0}^* := \min_Y \max_{S_b, S_o, u_0, W, \omega} \mathcal{L}_3(Y, S_b, S_o, u_0, W, \omega). \tag{2.3}
$$

Before we write the final **SDP** relaxation, we need the following adjoint operators:

$$
\begin{aligned}
b^0 \operatorname{Diag}(Y) &= \sum_{k=1}^n Y_{(k,:)(k,:)}, \\
o^0 \operatorname{Diag}(Y) &= \sum_{k=1}^n Y_{(:,k)(:,k)}, \\
\mathit{arrow}(Y) &= \operatorname{diag}(Y) - \begin{bmatrix} 0 \\ Y_{(2:,1)} \end{bmatrix}.
\end{aligned}
$$

Then the **SDP** relaxation of (1.3) is

$$\begin{aligned}\min_{Y}\ & \operatorname{tr}(L_Q Y)\\ \text{s. t. }\ & \mathrm{b}^0\operatorname{Diag}(Y)=I\\ & \mathrm{o}^0\operatorname{Diag}(Y)=I\\ & \operatorname{tr}(KY)=0\\ & \operatorname{arrow}(Y)=E_{00}\\ & Y\succeq 0.\end{aligned}\tag{2.4}$$

Theorem 2.1. *[29, Theorem 2.1] Suppose that Y is restricted to be rank-one in (2.4), i.e., $Y=\begin{bmatrix}1\\x\end{bmatrix}\begin{bmatrix}1\\x\end{bmatrix}^\top=\begin{bmatrix}1 & x^\top\\ x & xx^\top\end{bmatrix}$ for some $x\in\mathbb{R}^{n^2}$. Then the optimal solution of (2.4) provides the permutation matrix $X=\operatorname{Mat}(x)$ that solves the **QAP**.* ▲

The above theorem establishes the link between the **SDP** relaxation and the original **QAP**, enabling us to find the closest feasible solution using the relaxed solution. Note that the **SDP** relaxation is very tight since $d_y^*\le p_{R0}^*\le p_X^*$, where p_{R0}^* is the dual of the dual d_y^*.

Proposition 2.1. *[29, Proposition 2.1] Suppose the Y is feasible for the **SDP** relaxation (2.4). Then Y is singular.* ◁

Since all feasible solutions are singular, the set of the feasible solutions for (2.4) has no interior. This may cause instability when implementing any kind of interior-point method. The **SDP** relaxation of **QAP** presented in [29] uses *facial reduction* to guarantee strict feasibility. The so-called *minimal face* of the semidefinite cone is defined by a full-rank matrix $\hat{V}\in\mathbb{R}^{(n^2+1)\times[(n-1)^2+1]}$ given as

$$\hat{V}=\begin{bmatrix}1 & 0\\ \frac{1}{n}e\otimes e & V\otimes V\end{bmatrix},\tag{2.5}$$

where V is an $n\times(n-1)$ matrix containing a basis of the orthogonal complement of e, i.e. $V^\top e=0$. The feasible solutions for (2.4) can then be characterized as $Y=\hat{V}R\hat{V}^\top$, where $R\in\mathbb{S}_+^{(n-1)^2+1}$. It has been shown in [29] that the block-0-diagonal and off-0-diagonal operators have many redundant constraints. Therefore, the gangster operator is introduced to remove all redundant constraints and provide a tighter **SDP** relaxation.

Definition 2.1. [24, Definition 2.1] The gangster operator, $\mathcal{G}_J : \mathbb{S}^{n^2+1} \to \mathbb{S}^{n^2+1}$, is defined as

$$\mathcal{G}_J(Y)_{ij} = \begin{cases} 0 & \text{if } (i,j) \in J \text{ or } (j,i) \in J \\ Y_{ij} & \text{otherwise} \end{cases}$$

where the gangster index set, J, is the set of of indices $i < j$ of entries in Y (given in Theorem 2.1) corresponding to:

- the off-diagonal elements in the n diagonal blocks;
- the diagonal elements in the off-diagonal blocks except for the last column of off-diagonal blocks and also not the $(n-2, n-1)$ off-diagonal block (These latter off-diagonal block constraints are redundant after the facial reduction).

■

Before we write the final **SDP** relaxation, we need the following theorem to show that all constraints in (2.4), except for the positive semidefinite requirement, are redundant after adding the gangster operator.

Theorem 2.2. *[29, Theorem 4.1] Let $Y = \hat{V} R \hat{V}^\top$ be written in block matrix form as given in Theorem 2.1. Then*

1. *$\mathcal{G}_J(Y) = 0$ implies that $\operatorname{diag}(Y^{1,n}) = 0, \ldots, \operatorname{diag}(Y^{1,(n-1)}) = 0$, and $\operatorname{diag}(Y^{(n-2),(n-1)}) = 0$;*
2. *Let $\bar{J} = J \cup (0,0)$. Then $\mathcal{G}_{\bar{J}}(\hat{V} \cdot \hat{V}^\top)$ has range space equal to $\mathcal{S}_{\bar{J}} := \{X \in \mathbb{S}^{n^2+1} : X_{ij} = 0 \text{ if } (i,j) \notin \bar{J}\}$.*

▲

Now, the final **SDP** relaxation is

$$\begin{aligned} p^*_{R1} := \min_R \ & \operatorname{tr}\left(\left(\hat{V}^\top L_Q \hat{V}\right) R\right) \\ \text{s.t. } & \mathcal{G}_J\left(\hat{V} R \hat{V}^\top\right) = E_{00} \\ & R \succeq 0. \end{aligned} \tag{2.6}$$

Note that we reduce the search space from $\mathbb{S}^{n^2+1}_+$ to $\mathbb{S}^{(n-1)^2+1}_+$.

2.2 The New ADMM Algorithm

To apply **ADMM** to the **SDP** relaxation (2.6), we need an additional primal variable since the objective function is non-separable. In the previous section, we facially reduce Y to $\hat{V}R\hat{V}^\top$ and obtain a smaller problem. Here, we can add Y back as a primal variable and write (2.6) equivalently as

$$\begin{aligned} p^*_{RY} := \min_{R,Y} \;\; & \operatorname{tr}(L_Q Y) \\ \text{s. t. } & \mathcal{G}_J(Y) = E_{00} \\ & Y = \hat{V}R\hat{V}^\top \\ & R \succeq 0. \end{aligned} \tag{2.7}$$

Now, let Z be the matrix of multipliers for $Y = \hat{V}R\hat{V}^\top$. The augmented Lagrangian of (2.7) can then be written as

$$\mathcal{L}_\beta(R, Y, Z) = \operatorname{tr}(L_Q Y) + Z^\top(Y - \hat{V}R\hat{V}^\top) + \frac{\beta}{2}\|Y - \hat{V}R\hat{V}^\top\|^2. \tag{2.8}$$

We denote R, Y, and Z to be the primal reduced, primal, and dual variables before the **ADMM** updates. The new **ADMM** algorithm for the **SDP** relaxation (2.7) performs the following updates for R_+, Y_+, and Z_+:

$$R_+ = \operatorname*{arg\,min}_{R \in \mathbb{S}_+^{(n-1)^2+1}} \mathcal{L}_\beta(R, Y, Z), \tag{2.9a}$$

$$Y_+ = \operatorname*{arg\,min}_{Y \in \mathcal{P}_i} \mathcal{L}_\beta(R_+, Y, Z), \tag{2.9b}$$

$$Z_+ = Z + \gamma\beta(Y_+ - \hat{V}R_+\hat{V}^\top), \tag{2.9c}$$

where $\mathcal{P}_i$ is the polyhedral constraints for Y, e.g., $\mathcal{P}_i$ can be the linear manifold from the gangster constraints:

$$\mathcal{P}_1 = \{Y \in \mathbb{S}^{n^2+1} : \mathcal{G}_J(Y) = E_{00}\}.$$

Let $\hat{V}$ be normalized so that $\hat{V}^\top\hat{V} = I$. The explicit solution for the R update is

$$\begin{aligned} R_+ &= \operatorname*{arg\,min}_{R \in \mathbb{S}_+^{(n-1)^2+1}} \operatorname{tr}(L_Q Y) + Z^\top(Y - \hat{V}R\hat{V}^\top) + \frac{\beta}{2}\|Y - \hat{V}R\hat{V}^\top\|^2 \\ &= \operatorname*{arg\,min}_{R \in \mathbb{S}_+^{(n-1)^2+1}} \left\| Y - \hat{V}R\hat{V}^\top + \frac{1}{\beta}Z \right\|^2 \end{aligned}$$

$$
\begin{aligned}
&= \underset{R\in\mathbb{S}_+^{(n-1)^2+1}}{\arg\min} \left\| R - \hat{V}^\top\big(Y + \frac{1}{\beta}Z\big)\hat{V} \right\|^2 \\
&= \mathcal{P}_{\mathbb{S}_+^{(n-1)^2+1}}\left(\hat{V}^\top\big(Y + \frac{1}{\beta}Z\big)\hat{V}\right), \qquad (2.10)
\end{aligned}
$$

where $\mathcal{P}_{\mathbb{S}_+^{(n-1)^2+1}}(\cdot)$ is the projection onto $\mathbb{S}_+^{(n-1)^2+1}$. We can use $\mathcal{P}_1$ to write an explicit solution for the Y update as well:

$$
\begin{aligned}
Y_+ &= \underset{Y\in\mathcal{P}_1}{\arg\min}\ \mathrm{tr}(L_QY) + Z^\top(Y - \hat{V}R_+\hat{V}^\top) + \frac{\beta}{2}\|Y - \hat{V}R_+\hat{V}^\top\|^2 \\
&= \underset{Y\in\mathcal{P}_1}{\arg\min} \left\| Y - \hat{V}R_+\hat{V}^\top + \frac{L_Q+Z}{\beta} \right\|^2 \\
&= E_{00} + \mathcal{G}_{J^c}\left(\hat{V}R_+\hat{V}^\top - \frac{L_Q+Z}{\beta}\right). \qquad (2.11)
\end{aligned}
$$

Note that Y_+ is not necessarily positive semidefinite since we do not impose such condition in $\mathcal{P}_1$. Hence, Y_+ is usually infeasible for (2.7). We could have maintained the positive semidefinite condition by projecting Y_+ onto $\mathbb{S}_+^{n^2+1}$ at each iteration. Unfortunately, the **SDP** projection is not cheap since it involves calculating eigenvalues. Besides, we show later that we actually do not need $Y_+ \in \mathbb{S}_+^{n^2+1}$ to get the lower and upper bound.

We can tighten (2.7) further by adding the constraint $0 \le Y \le 1$:

$$
\begin{aligned}
p^*_{RY1} := \min_{R,Y}\ & \mathrm{tr}(L_QY) \\
\text{s. t. }\ & \mathcal{G}_J(Y) = E_{00} \\
& 0 \le Y \le 1 \\
& Y = \hat{V}R\hat{V}^\top \\
& R \succeq 0.
\end{aligned} \qquad (2.12)
$$

Then the Y update becomes

$$
Y'_+ = \underset{Y\in\mathcal{P}_2}{\arg\min}\ \mathcal{L}_\beta(R_+, Y, Z),
$$

where $\mathcal{P}_2 = \mathcal{P}_1 \cap \{0 \le Y \le 1\}$, and the corresponding explicit solution is

$$
Y'_+ = \min\left(1, \max\left(0, Y_+\right)\right).
$$

We observe that the above non-negativity constraint only slightly increases the complexity, which is a huge advantage of the **ADMM**. Although the less-than-one constraint is redundant, the algorithm converges faster when both constraints are present. We postpone the discussion of stopping criteria to Section 3.5 in a branch and bound setting.

2.3 Lower Bound

A successful branch and bound algorithm requires the lower bound to be tight and computationally efficient. Since the **SDP** relaxation is very tight, we would get a decent lower bound if we solve (2.12) to optimality. Nevertheless, the obsession with exact solution or even high accuracy would lead to horrible runtime in practice. Although it is not common in the bounding phase, we can actually control the quality of the lower bound by modifying the stopping criteria in the **ADMM**. While we do not discuss such stopping criteria in this section, we provide a way to get the lower bound after the **ADMM** is terminated.

Theorem 2.3. *[24, Lemma 3.1] Let $\mathcal{Y} := \{Y : \mathcal{G}_J(Y) = E_{00},\, 0 \leq Y \leq 1\}$, and $\mathcal{Z} := \{Z : \hat{V}^\top Z \hat{V} \preceq 0\}$. The dual function of (2.12) is*

$$g(Z) := \min_{Y \in \mathcal{Y}} \langle L_Q + Z, Y \rangle.$$

Then the dual problem of (2.12) is defined as follows and satisfies weak duality:

$$d_Z^* := \max_{Z \in \mathcal{Z}} g(Z) \leq \; p_{RY1}^*.$$

▲

Using the above theorem, we can take $g(Z)$ as the lower bound for (2.12) as well as for (1.3). However, the output from the **ADMM**, Z^{out}, does not necessarily belong to $\mathcal{Z}$ since we do not impose such condition in the Z update. Therefore, we need to project Z^{out} to $\mathcal{Z}$ and use $g\big(\mathcal{P}_{\mathcal{Z}}(Z^{out})\big)$ as the lower bound. Fortunately, Z is symmetric since both Y and R are symmetric, and so we can get $\mathcal{P}_{\mathcal{Z}}(Z^{out})$ using the following method.

Let $\hat{V}_\perp$ be the orthonormal basis of null$(\hat{V})$. Then $\bar{V} = (\hat{V}, \hat{V}_\perp)$ is an orthogonal matrix. Let $\bar{V}^\top Z^{out} \bar{V} = W = \begin{bmatrix} W_{11} & W_{12} \\ W_{21} & W_{22} \end{bmatrix}$, where $W_{11} \in \mathbb{S}^{(n-2)^2+1}$. Now, we have

$$\hat{V}^\top Z^{out} \hat{V} \preceq 0 \Leftrightarrow \hat{V}^\top Z^{out} \hat{V} = \hat{V}^\top \bar{V} W \bar{V}^\top \hat{V} = W_{11} \preceq 0.$$

Hence,

$$\begin{aligned}\mathcal{P}_{\mathcal{Z}}(Z^{out}) &= \arg\min_{Z\in\mathcal{Z}} \|Z - Z^{out}\|^2 \\ &= \arg\min_{W_{11}\preceq 0} \|\bar{V}W\bar{V}^\top - Z^{out}\|^2 \\ &= \arg\min_{W_{11}\preceq 0} \|W - \bar{V}^\top Z^{out}\bar{V}\|^2 \\ &= \begin{bmatrix} \mathcal{P}_{\mathbb{S}_-^{(n-2)^2+1}}(W_{11}) & W_{12} \\ W_{21} & W_{22} \end{bmatrix}.\end{aligned}$$

Note that $\mathcal{P}_{\mathbb{S}_-^{(n-2)^2+1}}(W_{11}) = -\mathcal{P}_{\mathbb{S}_+^{(n-2)^2+1}}(-W_{11})$, and we do not require $Y \in \mathbb{S}_+^{n^2+1}$ to get the lower bound.

2.4 Upper Bound (Feasible Solution)

The upper bound is used in a branch and bound algorithm to update the incumbent value until it reaches optimality. Let Y^{out} be the output from the **ADMM**; λ and v be the largest eigenvalue and its eigenvector. Although it is likely that $Y^{out} \notin \mathbb{S}_+^{n^2+1}$, Y^{out} should only be slightly infeasible with few small negative eigenvalues due to the Y update in the **ADMM**. We can then recover X^{out} from Y^{out} using the block matrix form in Theorem 2.1, in which $X^{out} = \mathrm{Mat}(x)$. We obtain x by taking the second through the last element of the first column of $\lambda vv^\top$. The recovered X^{out} is usually not a permutation matrix, and so we need to find its closest feasible counterpart.

$$\min_{X^\Pi\in\Pi_n} \|X^{out} - X^\Pi\|^2 \tag{2.13}$$

Using the *Frobenius norm*, we can write the objective function as $\|X^{out}\|_F^2 + \|X^\Pi\|_F^2 - 2\langle X^{out}, X^\Pi\rangle$, and write (2.13) equivalently as

$$\max_{X^\Pi\in\Pi_n} \langle X^{out}, X^\Pi\rangle. \tag{2.14}$$

Theorem 2.4. *(Birkhoff's Theorem) The set of doubly-stochastic matrices is identical with the convex hull of the set of permutation matrices.* ▲

We can replace the permutation matrix constraint in (2.14) by (1.4) and Theorem 2.4. Then we can solve the following linear programming problem using simplex method, and get the approximated feasible solution.

$$\begin{aligned} &\max_{X^{\Pi}} \langle X^{out}, X^{\Pi} \rangle \\ &\text{s. t. } X^{\Pi} e = e \\ &\qquad {X^{\Pi}}^{\top} e = e \\ &\qquad X^{\Pi} \geq 0. \end{aligned} \tag{2.15}$$

Instead of finding a feasible solution through (2.15), we can add the rank-one constraint to the R update and modify the update as follows:

$$R_{+} = \mathcal{P}_{\mathbb{S}_{+}^{(n-1)^2+1} \cap \mathcal{R}_1} \left(\hat{V}^{\top} (Y + \frac{Z}{\beta}) \hat{V} \right), \tag{2.16}$$

where $\mathcal{R}_1 = \{R : \operatorname{rank}(R) = 1\}$ denotes the set of rank-one matrices. Since $\operatorname{rank}(AB) \leq \min(\operatorname{rank}(A), \operatorname{rank}(B))$, $Y = \hat{V} R \hat{V}^{\top}$ should also be rank-one. For $Y^{out} \in \mathbb{S}^{n^2+1}$ with largest eigenvalue $\lambda > 0$ and corresponding eigenvector w, we have $\mathcal{P}_{\mathbb{S}_{+}^{n^2+1} \cap \mathcal{R}_1}(Y^{out}) = \lambda w w^{\top}$. Then $X^{out} = \operatorname{Mat}(x)$, where x is obtained from $\lambda w w^{\top}$, is a permutation matrix up to machine epsilon. In our implementation, we get two upper bounds using the above two methods and use the smaller one to update the incumbent value.

Chapter 3

Branch and Bound for the QAP

A branch and bound algorithm for the **QAP** recursively divides the current unsolved problem into a few subproblems, which are then solved or divided again. Depending on the branching strategy, some subproblems start with a partial permutation matrix that results in a smaller **QAP**. When the algorithm begins, the original **QAP** are solved by the **ADMM** algorithm using the **SDP** relaxation. Either the optimal value is reached, in which case we stop and get the solution for the **QAP**, or we have an upper and lower bound of the **QAP**. If at any given node, the lower bound exceeds the global optimal upper bound (the incumbent value), then further branching on that node could not lead to a solution that is as good as the incumbent value. Therefore, the node is pruned from the search tree. The algorithm gradually visits all nodes of the search tree that are not pruned, trying to find a better solution and prove optimality.

The algorithm is guaranteed to find the optimal solution if both the branching and bounding phases are correct.

Lemma 3.1. *[15, 2.1] Let $F(p_X^*)$ denote the feasible set of a **QAP**, and $p_{X1}^*, \dots, p_{Xk}^*$ denote the subproblems of p_X^*. The branching is valid if :*

1. *Every feasible solution of p_X^* is a feasible solution of at least one of the subproblems $p_{X1}^*, \dots, p_{Xk}^*$.*
2. *Every feasible solution of any of the subproblems $p_{X1}^*, \dots, p_{Xk}^*$ is a feasible solution of p_X^*.*

$\triangle$

The above two conditions ensure that $F(p_X^*) = F(p_{X1}^*) \cup \cdots \cup F(p_{Xk}^*)$, and so we should find the the same set of solutions at the end of the algorithm. We show in Section 3.2 that our branching strategy fulfills those conditions.

The credibility of the lower bound depends on the relaxation.

Lemma 3.2. *[15, 2.2] A **SDP** relaxation p_{RY}^* of the original **QAP** p_X^* is only valid when the following requirements are met:*

1. *If p_{RY}^* has no feasible solutions, then the same is true of p_X^*.*
2. *The minimal value of p_X^* is no less than the minimal value of p_{RY}^*.*
3. *If an optimal solution of p_{RY}^* is feasible in p_X^*, then it is an optimal solution of p_X^*.*

$\triangle$

The first requirement does not necessarily apply here since the relaxation and the original **QAP** always have feasible solutions unless a subproblem starts with an infeasible matrix, e.g., a facility is mistakenly assigned to two locations. The **SDP** relaxation is a lower bound to the **QAP**, and so it satisfies the second requirement. Besides, the **QAP** is a minimization problem, and if a lower bound solution is feasible for the **QAP**, it is also the optimal solution.

In the next section, we introduce some existing strategies to select subproblems. Although the way we choose subproblems does not affect the correctness of the algorithm, it affects the runtime and efficiency of the algorithm. We discuss the branching rules in Section 3.2, and provide a way to get the bounds from a reduced problem in Section 3.3. Our complete branch and bound algorithm is presented in Section 3.4. Then we discuss how the stopping criteria in **ADMM** affect the runtime of the algorithm in Section 3.5, where we also propose a new rule using the incumbent value. In Section 3.6, We conduct a grid search on combinations of some parameters in **ADMM**, and we end this chapter with an experiment on node selection strategies.

3.1 Node Selection Strategy

When the algorithm decides to branch on a particular node, a few subprobelms are added to the queue and waiting for inspection. As the size of the problem grows larger, or the depth of the current node becomes larger, the number of subproblems would be enormous. The order of processing those nodes turns into a non-trivial matter at that point. Although the order does not affect the correctness of the algorithm, it directly affects the incumbent value and ultimately the efficiency of the algorithm. Like other tree-based models, branch and bound for the **QAP** has two commonly used and problem-independent strategies – breadth first search and depth first search. In fact, most branch and bound implementations use depth first search to control the size of the queue. We are aware of other strategies such as instance related, max lower bound, and min number of nodes strategies, but as Burkard et al. point out in [8], there is no dominant strategy. Therefore, we decide to implement breadth first search and depth first search, and briefly discuss other possibilities at the end of this section.

In *breadth first search*, the node with a smaller depth is always visited before others that are deep in the branch and bound tree. The search rule is usually enforced using a first-in-last-out (FIFO) queue. When the algorithm decides to further divide a problem into a few subproblems, all of those subproblems are added to the queue immediately. Then we pop one subproblem from the queue and get its bounds by using the **ADMM** algorithm. If that subproblem is divided again, all of its sub-subproblems are added to the queue without any processing. As the name FIFO suggests, the node who comes first is processed first. Therefore, problems of smaller depth are processed before others.

We notice that the above strategy is not very popular among existing implementations because the size of the queue grows very fast. Nevertheless, the problem is ultimately due to the poor quality of lower bounds, especially the **GLB** that are used by many algorithms. Breath first search is more efficient when the time spent on a solved or pruned node is less than the time spent on processing the subtree originated from that node. Since we are able to control the quality of the lower bound obtained by the **ADMM**, we have the opportunity to balance the time spent on one node and the tightness of its lower bound.

In *depth first search*, the algorithm always processes node with the largest depth first. The search rule is usually enforced using a last-in-first-out (LIFO) queue. When a node is processed, if it is not solved and its lower bound is less than the incumbent value, the algorithm first puts the node back to the queue, and then add one of its subproblems to the queue. The LIFO queue sends the subproblem to the **ADMM** algorithm and so forth. In this way, some deep nodes are visited early on in branch and bound. Those nodes contain smaller problems that are easy to solve, and thus the incumbent value is more likely to be improved very fast. In addition, the size of the queue are relatively smaller since it only contains visited nodes and one additional node.

We understand that controlling the size of the queue is a big problem for many previous implementations of branch and bound. The algorithm in [6] produces 239,449 to 360,148,026 nodes for nug20 depending on different bounding methods. Our implementation, on the other hand, produces fewer than 1,000 nodes for nug20. Hence, depth first search does not provide us with the advantage of a smaller queue as it does to previous branch and bound algorithms, which is confirmed by our experiment in Section 3.7.

Other node selection strategies generally involve some node ranking criteria in the hope of finding the optimal solution as soon as possible. If the incumbent value reaches optimality early in the algorithm, the nodes would be pruned more often at small depths, reducing the size of the queue as well as the runtime. However, such strategies bring some overhead to node processing. The accumulation of the overhead is an inevitable drawback of this kind of strategies and often contradicts the intention to find the optimal solution quickly. Since the size of the queue is not a pressing matter in our implementation, we probably would not benefit that much from ranking nodes according to some rule. Therefore, we only consider breadth first search and depth first search in our implementation.

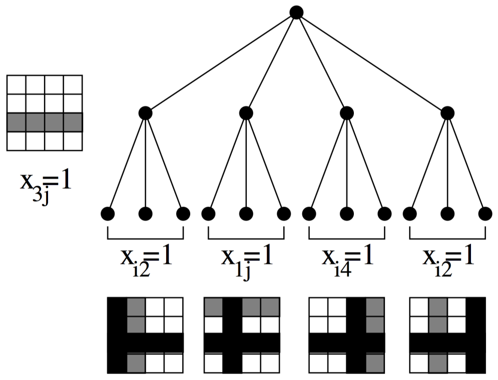
3.2 Branching Strategy

In the branching phase, a subproblem is further divided into a few sub-subproblems. Ideally, the work required to process those sub-subproblems should be substantially less than that for the subproblem. However, bad branching strategy might lead to sub-subproblems that are as difficult as the subproblem. Empirical tests in [8, 26] have shown that single assignment branching and polytomic branching are more efficient than other branching strategies.

Branching strategies generally answer two questions – how many subproblems to create and how to divide the search space [6]. Single assignment branching creates two branches by assigning a facility to a location and preventing such assignment. The assignment branch is an one-size smaller **QAP** while the other one is a similar problem with some modification. Therefore, the repeated assignment branch would be much easier to solve than others deep in the tree. Although the work load is terribly skewed, this branching strategy was used in [25] to solve some previously unsolved **QAPLIB** instances and enjoyed general applause over others for a long time. Depending on the assignment prevention method used to create a subproblem, that subproblem might not be feasible if a facility happens to be banned from all locations, or vice versa. Many branch and bound algorithms using single assignment branching have to keep track of the assignment in order to guarantee feasibility.

Two major drawbacks of single assignment branching, the infeasibility problem and the imbalance workload, are addressed by polytomic branching. This branching strategy was first used in [27] and later in [20], and now it is the most commonly used strategy for algorithms capable of solving large scale **QAP**s [6]. The implementation can either be row branching, assigning an available facilities to all possible locations, or column branching, using an available location to host all facilities. Based on previous successes, we decide to use polytomic branching in our branch and bound algorithm. The conditions for a valid branching rule in Lemma 3.1 are satisfied by polytomic branching since the search space is divided equally among all branches. We demonstrate polytomic branching rule in Figure 3.1. Suppose we have a **QAP** of size 4 and we branch on facility 3 (row branching) at the first level. Then four subproblems are created at this level where facility 3 is assigned to

Figure 3.1: [6, Figure 3.5] **Polytomic Branching Strategy**



location 1 to 4 (The gray bar in the top-left grid illustrates the corresponding entries in X). We present a mix of row and column branching at the second level. The black bars represent the forbidden location, e.g., if facility 3 is assigned to location 1 before, then row 3 and column 1 are fixed with 1 at $(3, 1)$ and 0 at other places. The nodes spawn from node 1, 3 and 4 perform column branching, e.g., location 2 is chosen at node 1 and now facility 1, 2 and 4 are assigned to that location, resulting in three children. On the other hand, node 2 at the first level performs row branching again and assigns facility 1 to location 1, 3 and 4. Note that all child problems are exactly one size smaller than their parent problems.

3.3 Bounding Reduced QAP

Polytomic branching always creates subproblems that are smaller than the original **QAP**. Although they are easier to solve, we cannot use the **ADMM** directly on them partially due to the connection between the assigned facilities and the unassigned ones. We present a way to modify the inputs of the reduced **QAP** and to restore the lower and upper bound of the reduced problem to their counterparts in the corresponding full-sized **QAP** using the example below.

Example 3.1. Suppose we have obtained a partial solution $X = \left[\begin{array}{c|c} I_r & 0 \\ \hline 0 & X' \end{array}\right]$ where I_r is the reordered solved part and is an identity matrix of size $r < n-1$, and $X' \in \Pi_{n-r}$ is the unsolved part. We reorder F, D, and C in a similar fashion, and our objective function is

$$\begin{aligned}
p_X^* &:= \min_{X \in \Pi_n} \operatorname{tr}\left(FXDX^\top + 2CX^\top\right) \\
&:= \min_{X' \in \Pi_{n-r}} \operatorname{tr}\left(\left[\begin{array}{c|c} F_r & F_{r:} \\ \hline F_{r:}^\top & F' \end{array}\right]\left[\begin{array}{c|c} I_r & 0 \\ \hline 0 & X' \end{array}\right]\left[\begin{array}{c|c} D_r & D_{r:} \\ \hline D_{r:}^\top & D' \end{array}\right]\left[\begin{array}{c|c} I_r & 0 \\ \hline 0 & X' \end{array}\right]^\top\right) \\
&\quad + 2\operatorname{tr}\left(\left[\begin{array}{c|c} C_r & C_{r:} \\ \hline C_{r:}^\top & C' \end{array}\right]\left[\begin{array}{c|c} I_r & 0 \\ \hline 0 & X' \end{array}\right]^\top\right) \\
&:= \min_{X' \in \Pi_{n-r}} \operatorname{tr}\left(\left[\begin{array}{c|c} F_rD_r + F_{r:}X'D_{r:}^\top & F_rD_{r:}X'^\top + F_{r:}X'D'X'^\top \\ \hline F_{r:}^\top D_r + F'X'D_{r:}^\top & F_{r:}^\top D_{r:}X'^\top + F'X'D'X'^\top \end{array}\right]\right) \\
&\quad + 2\operatorname{tr}\left(\left[\begin{array}{c|c} C_r & C_{r:}X'^\top \\ \hline C_{r:}^\top & C'X'^\top \end{array}\right]\right) \\
&:= \min_{X' \in \Pi_{n-r}} \operatorname{tr}\left(F'X'D'X'^\top + 2\left(F_{r:}^\top D_{r:} + C'\right)X'^\top + F_rD_r + C_r\right).
\end{aligned}$$

We can now solve $\min_{X' \in \Pi_{n-r}} \operatorname{tr}\left(F'X'D'X'^\top + 2\left(F_{r:}^\top D_{r:} + C'\right)X'^\top\right)$ using the **ADMM** algorithm from the previous chapter. After we obtain the lower and upper bound of the reduced problem, we can restore them to the corresponding bounds of the full-sized problem by adding $\operatorname{tr}(F_rD_r + C_r)$. □

Note that the above example can be easily generalized to any **QAP** with existing assignments through index manipulation. Suppose we have a **QAP** of size n with r existing assignments. Let $(Fa, La) = \{(Fa, La) : X(Fa_i, La_i) = 1 \ \forall i = 1, \ldots, r\}$ be the set of indices in which facility Fa_i is assigned to location La_i for all $i = 1, \ldots, r$. Let $Fu = \{1, \ldots, n\} \setminus Fa$ and $Lu = \{1, \ldots, n\} \setminus La$. Note that those four sets are used as row/column indices instead of indices of an entry in X. We now define the reduced **QAP** as

$$p_{X_r}^* := \min_{X_r \in \Pi_{n-r}} \operatorname{tr}\left(F_{Fu,Fu}X_rD_{Lu,Lu}X_r^\top + 2\left(F_{Fa,Fu}^\top D_{La,Lu} + C_{Fu,Lu}\right)X_r^\top\right). \tag{3.1}$$

The constant to be added to the reduced lower and upper bound is $\operatorname{tr}(F_{Fa,Fa}D_{La,La} + C_{Fa,La})$.

3.4 Algorithm

We present two algorithms below. The first one uses breadth first search with a FIFO queue. The original **QAP** is wrapped in a node and appended to the queue in the beginning, and the incumbent value is set to infinity. Then the algorithm visits the first node in the queue and gets the appropriate lower and upper bound on that node. If the lower bound is less than or equal to the incumbent value, then we update the incumbent value use the upper bound and proceed to decide if further branching is needed. If the node is solved by **ADMM**, i.e., the lower bound equals the upper bound, we add it to the solution list; otherwise, we add all of its subproblems to the queue. No matter which case this node falls into, it is deleted from the queue when the processing finishes. When the queue becomes empty, we use the incumbent value to remove previously solved but not optimal solutions from the list. Then the solution list contains only optimal solutions to the original **QAP**.

Algorithm 1: Branch and Bound with Breadth First Search

```
$FIFOq = \{\mathbf{QAP}\,(F, D, C)\}$;
$incumbent = \infty$;
while $FIFOq$ is not empty do
    $p$ = the first node of $FIFOq$;
    modify $F, D$, and $C$ if there is any existing assignment in $p$;
    get lower bound ($lb$) and upper bound ($ub$) from $\mathbf{ADMM}\,(p)$;
    restore $lb$ and $ub$ if there is any existing assignment in $p$;
    if $lb \leq incumbent$ then
        $incumbent = \min(incumbent, ub)$;
        if $lb = ub$ then
            add $p$ to the solution list;
        else
            create all subproblems and add them to $FIFOq$;
        end
    end
    remove $p$ from $FIFOq$;
end
delete non-optimal solution from the solution list.
```

The second algorithm uses depth first search with a LIFO queue. The steps are mostly the same as the above algorithm. However, only one subproblem is selected to visit next instead of all subproblems, and a node can only be deleted if either it fails the incumbent value test or all of its subproblems are visited.

Algorithm 2: Branch and Bound with Depth First Search

```
$LIFOq = \{\textbf{QAP}\,(F, D, C)\}$;
$incumbent = \infty$;
while $LIFOq$ is not empty do
    $p$ = the last node of $LIFOq$;
    modify $F, D$, and $C$ if there is any existing assignment in $p$;
    get lower bound ($lb$) and upper bound ($ub$) from ADMM ($p$);
    restore $lb$ and $ub$ if there is any existing assignment in $p$;
    if $lb > incumbent$ or all subproblems of $p$ are created then
        remove $p$ from $LIFOq$;
    else
        $incumbent = \min(incumbent, ub)$;
        if $lb = ub$ then
            add $p$ to the solution list;
            remove $p$ from $LIFOq$;
        else
            create one subproblem and add it to $LIFOq$;
        end
    end
end
delete non-optimal solution from the solution list.
```

3.5 Stopping Criteria of the ADMM

The implementation of the **ADMM** for the **SDP** relaxation in [24] has three stopping conditions for the **ADMM** updates – limiting the maximum number of iterations, controlling the Y update, and controlling the feasibility of Y. Limiting the number of iterations is the simplest way to control the amount of time spent on one node. If a problem has difficulty converging after running for a very long time, it is probably more efficient to move on to its subproblems. However, a small allowance may lead to poor lower bounds and eventually slows down the branch and bound algorithm due to the inability to prune any branch. We explore the effect of the limit on the performance of our algorithm in Section 3.6.

The other two stopping criteria are based on the change of variables after the **ADMM** updates. Let $r_p = Y_+ - \hat{V}R_+\hat{V}^\top$ denote the primal residual and $r_d = Y_+ - Y$ the dual residual. Note that r_p corresponds to the feasible condition of Y and r_d is the change of Y at each iteration. The tolerance is set on $\|r_p\|$ and $\beta\|r_d\|$. The **ADMM** terminates if both norms falls below the tolerance for five consecutive times. It is, however, difficult to interpret those two norms in a branch and bound setting. The branch and bound algorithm generally does not require high precision. In particular, obtaining better bounds on one node is a terrible goal that forces the algorithm to spend too much time on difficult problems. We show in Section 3.6 that smaller tolerance dramatically slows down our branch and bound algorithm.

Besides the above three stopping criteria, we propose another check that can potentially terminate the **ADMM** much earlier. We pause the **ADMM** update every N iterations and calculate the lower bound from the newly updated Z_+ using Lemma 2.3. Instead of using Y_+, we generate $Y(\mathcal{P}_{\mathcal{Z}}(Z_+))$ using the following rule:

$$Y(\mathcal{P}_{\mathcal{Z}}(Z_+))_{ij} = \begin{cases} 1 & \text{if } (i,j) \notin J, (j,i) \notin J, \text{ or } (L_Q + Z_+)_{ij} < 0 \\ 0 & \text{otherwise} \end{cases}$$

Note that $Y(\mathcal{P}_{\mathcal{Z}}(Z_+)) \in \mathcal{Y}$, and the lower bound is $g(\mathcal{P}_{\mathcal{Z}}(Z_+)) = \langle L_Q + \mathcal{P}_{\mathcal{Z}}(Z_+), Y(\mathcal{P}_{\mathcal{Z}}(Z_+))\rangle$. The **ADMM** can then compare the incumbent value and terminates if the lower bound is already larger than that value. The early termination not only saves us the remaining iterations to reach one of the previous three stopping criteria, but also relieves us from calculating any upper bound at all. This new stopping criterion is actually triggered very often in our branch and bound algorithm since we are able to obtain tight lower bound from the **SDP** relaxation, and the incumbent value reaches optimal value very fast in general.

3.6 Parameter Tuning for the ADMM

We conduct grid search on a few combinations of the maximum number of iterations and the tolerance using nug15. Note that parameter tuning is only applicable to the high-rank **ADMM**, while the parameters for the low-rank case can be set based on the fact that Y is rank-one with only $(0, 1)$ entries. We use computer 2 (see Appendix A) and fix the maximum number of iterations to 500 and the tolerance to 0.1 for the low-rank case. Since breadth first search and depth first search have similar performance for nug15, we only use depth first search in our experiment.

The results in Table 3.1 are consistent with our expectations in Section 3.5. The optimal maximum number of iterations is 800 and the optimal tolerance is 0.01. In general, a small maximum number of iterations forces the stopping criterion based on that number to be triggered more frequently, while a large one essentially voids that stopping criterion. If the tolerance is too small, then all lower bounds are not very tight, causing the algorithm to visit a larger portion of the tree. Note that the optimal maximum number of iterations are more dependent on specific instances, but the tolerance can be a more universal criterion. The former essentially sets a limit to the maximum amount of time the algorithm can spend on a particular node. This limit should be adaptable to size and depth for optimal performance. However, due to resource and time constraints, we do not incorporate such idea in our algorithm and simply use the above optimal combination for all experiments in this paper.

Table 3.1: Runtime comparison for parameter tuning

Tol. Max. Num. of Iter.	$1e-1$	$1e-2$	$1e-3$	$1e-4$	$1e-5$	$1e-6$
500	3,327.57	**1,274.21**	1,269.73	1,275.86	1,310.49	1,312.14
600	3,290.97	**927.78**	966.95	878.73	1,017.87	869.29
700	3,133.40	**725.85**	730.44	715.27	713.45	724.38
800	**3,002.75**	**449.66**	**468.10**	**520.24**	**498.94**	**487.99**
900	3,093.60	**476.35**	498.81	522.51	522.56	542.85
1,000	3,107.68	**506.74**	568.56	576.50	573.05	518.61
2,000	3,074.21	**546.60**	709.45	753.74	750.15	762.94
3,000	3,061.69	**607.36**	881.35	963.78	981.25	1,004.60
4,000	2,926.89	**584.86**	950.19	1,001.46	1,114.57	1,093.87
5,000	3,099.89	**581.44**	997.61	1,095.88	1,257.40	1,312.19
6,000	3,136.70	**597.26**	1,111.89	1,225.33	1,492.08	1,418.40
8,000	3,155.58	**581.68**	1,133.15	1,209.60	1,543.59	1,625.89
10,000	3,135.98	**629.11**	1,107.37	1,268.23	1,466.63	1,699.88
20,000	3,049.96	**577.05**	1,107.32	1,228.51	1,484.13	2,025.60
40,000	3,168.35	**598.35**	1,080.00	1,230.47	1,491.18	2,108.75

3.7 Experiment on Node Selection Strategies

We test breadth first search and depth first search on a few **QAPLIB** instances of size up to 15. We use computer 5 (see Appendix A) and set the maximum number of iterations to 800 and tolerance to 0.01 for the high-rank **ADMM**; 500 and 0.1 for the low-rank case. The parameters for the high-rank case are explored in Section 3.6, in which we use nug15 to conduct a grid search, while those for the low-rank case are determined based on the fact that Y is rank-one with only $(0, 1)$ entries.

We compare the number of solutions, runtime, and the number of nodes between those two strategies. In Table 3.2, the two strategies are able to find the same number of solutions except for scr12, in which depth first search find one fewer solution. The runtime of the two strategies are also similar except for rou15. Breadth first search terminates at the first level, while depth first search visits 195 nodes before termination, 13 times the number of nodes as the other method does. Again, due to the good quality of our lower bounds and fast convergence, it appears that depth first search is not better than breadth first search in our algorithm.

Table 3.2: Breadth first search v. depth first search

Dataset	Number of Solutions (BF)	Time (s) (BF)	Number of Nodes (BF)	Number of Solutions (DF)	Time (s) (DF)	Number of Nodes (DF)
nug12	4	48.54	12	4	39.29	23
nug14	1	55.62	14	1	37.85	14
nug15	4	184.91	15	4	104.02	15
rou12	1	34.49	68	1	35.16	68
rou15	1	91.21	15	1	244.88	195
scr12	8	128.68	218	7	126.41	294
scr15	2	269.02	79	2	348.81	222
tai15	1	549.79	326	1	694.92	576

Next, we examine closely the number of nodes by depth for tai15a in Table 3.3. Both strategies are able to prune most branches early on in the algorithm due to the advantage of the tight lower bound. Depth first search goes deeper in the search tree and visits a few more nodes, resulting in a slightly longer runtime. The results, however, do not necessarily indicate that depth first search works less effective than breadth first search. The latter performs surprisingly well because first, the **ADMM** is able to find the optimal value at the first level of the search tree, and second, the lower bounds on all nodes are very tight. Unfortunately, as the size of the problems grows, it is less likely for breadth first search to achieve either condition at smaller depth. In such case, depth first search would work better by finding the optimal value deep in the tree.

Table 3.3: Number of nodes by depth for tai15a

Depth	Maximum	Number of Nodes (BF)	% Nodes Visited (BF)	Number of Nodes (DF)	% Nodes Visited (DF)
1	15	15	1.00E+02	15	1.00E+02
2	210	210	1.00E+02	210	1.00E+02
3	2,730	26	9.52E-01	91	3.33E+00
4	32,760	24	7.33E-02	60	1.83E-01
5	360,360	11	3.05E-03	44	1.22E-02
6	3,603,600	10	2.78E-04	40	1.11E-03
7	32,432,400	9	2.78E-05	36	1.11E-04
8	259,459,200	8	3.08E-06	32	1.23E-05
9	1,816,214,400	7	3.85E-07	21	1.16E-06
10	10,897,286,400	6	5.51E-08	18	1.65E-07
11	54,486,432,000	0	0.00E+00	5	9.18E-09
12	217,945,728,000	0	0.00E+00	4	1.84E-09
Total	285,441,552,075	326	1.14E-07	576	2.02E-07

Chapter 4

Numerical Experiments

We test our two branch and bound algorithms in Section 3.4 on a selection of **QAP** instances of size up to 28, all of which have known optimal values and are shown in the second column. Our implmentation uses some MATLAB files from [24]. The **ADMM** algorithm (ADMM_QAP.m) is modified to add a new stopping criterion proposed in Section 3.5. We check the new criterion every 200 iterations in **ADMM** ($N = 200$) since it evokes *qr* and *eig* in MATLAB, which does incur some overhead. We write our own function for calculating lower and upper bounds based on the original test script (quicktest.m). We keep $\gamma = 1.618$ and $\beta = \frac{n}{3}$ in **ADMM**, which is previously used in [24]. The maximum number of iterations are set to 800 and the tolerance to 0.01 for the high-rank **ADMM** as determined by the grid search in Section 3.6. Those parameters for the low-rank case are kept at 500 and 0.1. All results in this paper are produced using *seed* 39 to randomly select a facility in the branching phase, and the verbose option in our test scripts is turned off.

Halfway through our initial experiment, we suffered from occasional failure of the *eig* function in MATLAB when updating R for the high-rank case. We try to avoid such situation by smoothing the matrix if *eig* does not converge for the original one. Nevertheless, the smoothing method does not always work. The error seems to happen exclusively on Linux servers running MATLAB R2015a (computer 1 to 4, see Appendix A), since such error did not occur in a trial run using nug15 on a PC with MATLAB R2016a. Unfortunately, the majority of our tests are done using those Linux servers. Since we do not completely understand why the *eig* function fails in our test, we have no choice but to give up on a particular node if such error happens. Instead of solving the bad node, we simply set its upper bound to infinity and lower bound to negative infinity so that all of its subproblems are inspected by our algorithm.

The number of solutions obtained by the algorithm is given in column 3. The runtime, presented in column 4, is timed from the moment after the test problem is loaded to the termination of the branch and bound algorithm after the solution list is finalized. The total number of nodes visited by the algorithm is presented in column 5. Although the number of nodes by depth is also recorded, we do not show such information due to space limit. The last column indicates the computer used in solving a particular problem (see Appendix A). Note that computer 1 to 4 are shared servers, and so there are some inconsistencies in the runtime across all tests. Experiments that take longer are especially prone to sudden burst of jobs on the server. Readers are advised to regard the runtime as a rough idea of the performance instead of a metric for cross comparison.

Our results demonstrate that our bounding strategy, the **ADMM** algorithm for the **SDP** relaxation, provides very strong bounds that are extremely effective in the pruning phase of the branch and bound algorithm. When comparing our results with [1, 6], we observe that our algorithm produces similar results, in terms of the number of nodes, as the **RLT1/RLT2** method in [1]. The quadratic-programming-bound based branch and bound algorithm in [4] generates more than 2 billion nodes for nug28; the **RLT1/RLT2** method generates 202,295 nodes, while our algorithm generates only 31,463 nodes using breadth first search with parallel computing. This dramatic reduction in the number of nodes makes it possible to apply our algorithm to large **QAP** instances.

4.1 Results for Depth First Search

We present the results in Table 4.1, all of which are produced on the two newest servers of the fastlinux.math pool of the MFCF. The runtime grows exponentially as the size becomes larger. This growth rate is mostly due to the expansion of the search space, which is the set of all feasible solutions. A problem of size 10 has $10! = 3,628,800$ feasible solutions; of size 15 has more than a trillion; of size 20 has more than 2×10^{18}; of size 25 has more than 10^{25} – a poorly designed algorithm would take an insane amount of time enumerating all possible solutions. In our experiment, we were able to test problems of size up to 25 in a reasonable amount of time. Due to the growth of the runtime, parallel computing is the only possible way to solve large **QAP** instances of size greater than 25.

Table 4.1: Numerical results for branch and bound with depth first search

Dataset	Optimal Value	Num. of Opt. Solns Found	Time (s)	Num. of Nodes Visited	Computer
had16	3,720	1	289.22	16	1
had18	5,358	2	2,588.88	178	1
had20	6,922	7	9,517.94	832	1
nug12	578	4	43.74	23	1
nug14	1,014	1	49.56	14	1
nug15	1,150	4	147.85	15	1
nug16a	1,610	1	144.84	16	1
nug16b	1,240	8	419.06	31	1
nug17	1,732	1	1,151.46	188	1
nug18	1,930	2	5,071.32	805	1
nug20	2,570	4	12,272.13	793	1
nug21	2,438	4	13,346.15	788	1
nug24	3,488	8	98,094.79	3,637	1
nug25	3,744	8	724,262.47	19,916	1
rou12	235,528	1	40.26	68	1
rou15	354,210	1	303.89	195	1
rou20	725,522	1	74,114.27	8,942	1
scr12	31,410	7	131.46	294	1
scr15	51,140	2	1,607.98	745	1
scr20	110,030	4	11,217.48	1,356	1
tai15a	388,214	1	874.64	576	1
tai17a	491,812	1	5,789.18	1,370	1
tai20a	703,482	1	143,114.56	12,883	1

4.2 Results for Breadth First Search with Parallel Computing

Instead of running the breadth first search sequentially, we decide to implement a relatively simple *parallel computing* version using the parallel computing toolbox in MATLAB. In the original algorithm in Section 2, we only examine the first node in the FIFO queue. By replacing the for-loop with parfor-loop, we are able to simultaneously examine all nodes in the FIFO queue and then append all subproblems to the queue after finishing processing all nodes at the same depth. This is, however,not a complete parallel computing implementation, in which a supervisor distribute the node in the queue to an empty worker while the worker can add new nodes to the queue concurrently. Most of our results are produced on the newest server in the biglinux.math pool of the MFCF using 23 workers. Note the surprising results from esc16h. Our algorithm works terribly on this problem probably due to the inability to explore the hidden *group symmetry* feature as noted in [10] – a subset of facilities can be put in any location of a subset of locations to achieve optimality.

Our results indicate that parallel computing is more effective for large **QAP** instances, while its performance is similar or even worse than sequential computing for small ones. In Table 4.1, had16, had18, and had20 take 289, 2589, and 9518 seconds to finish execution, while in parallel computing they take 184, 291, and 738 seconds. We expect breadth first search produce fewer nodes as shown in Section 3.7, and our results here are consistent with the previous experiment. This is largely due to our powerful bounding strategy, which is able to prune nodes of small depth in the beginning of the algorithm.

It does not make much sense, however, to compare the runtime from the parallel computing implementation directly to that from the sequential version. Some runtime conversion methods are proposed by [1, 6, 19], but they are still case dependent and not easily adaptable in a different setting. A notable reason that prevents such direct comparison is that parallel computing involves large I/O transmission that is much less effective than the I/O within one process. As the size of the problem grows, the I/O operation contributes a non-trivial amount of time to the actual runtime of the algorithm.

Table 4.2: Numerical results for branch and bound with breadth first search and parallel computing

Dataset	Optimal Value	Num. of Opt. Solns Found	Time (s)	Num. of Nodes Visited	Computer
esc16h	996	6,121,489	656,088.55	22,444,642	4
had16	3,720	1	183.56	16	3
had18	5,358	1	290.76	102	3
had20	6,922	11	738.22	817	3
nug12	578	4	29.43	12	3
nug14	1,014	1	58.92	14	3
nug15	1,150	4	91.72	15	3
nug16a	1,610	1	205.14	16	3
nug16b	1,240	8	118.78	16	3
nug17	1,732	1	210.45	97	3
nug18	1,930	2	417.59	435	3
nug20	2,570	4	939.63	724	3
nug21	2,438	4	929.66	667	3
nug22	3,596	4	2,448.12	939	3
nug24	3,488	8	8,973.21	3,086	3
nug25	3,744	8	47,397.03	18,572	3
nug27	5,234	8	120,750.28	9,936	4
nug28	5,166	4	496,225.14	31,463	4
rou12	235,528	1	76.96	50	3
rou15	354,210	1	85.19	15	3
rou20	725,522	1	3,418.73	6,187	3
scr12	31,410	7	74.39	172	3
scr15	51,140	2	108.89	57	3
scr20	110,030	4	1,274.70	1,094	3
tai15a	388,214	1	86.79	225	3
tai17a	491,812	1	441.64	304	3
tai20a	703,482	1	4,448.48	7,342	3
tai25a	1,167,256	1	896,627.35	554,701	3

Chapter 5

Conclusion and Future Work

In this paper, we proposed a new branch and bound algorithm based on the new **ADMM** method for solving the **SDP** relaxation of the **QAP**. We used both depth first search and breadth first search in our node selection process. Our experiment on node selection strategies show that the depth first search does not have a relative advantage in our algorithm, which contradicts with many previous works in favor of such strategy. We believe the advantage of the depth first search is not evident due to the sharp lower bound obtained from the **ADMM**. Since the algorithm is able to prune nodes at smaller depth early on in the process, breadth first search actually visits fewer nodes using a comparable amount of time than the depth first search does. We added a new stopping criterion in the **ADMM** that is able to use the incumbent value from the branch and bound algorithm. This early termination improves the efficiency of the overall algorithm by avoiding unnecessary iterations in the **ADMM**. We modified two parameters, the maximum number of iterations and the tolerance, according to our grid search on a few combinations.

Our numerical experiments demonstrate that our new branch and bound algorithm provides comparable results as do some of the best existing algorithms. This is largely due to the tight lower bounds that are extremely effective in the pruning phase of the branch and bound algorithm. In particular, the dramatic reduction in the number of visited nodes makes it possible to apply our algorithm to large **QAP** instances.

We are unable, however, to compare the runtime of our algorithm directly with others due to the difficulty in runtime conversion. Ideally, all algorithms should be tested on the same machine or computational grid. Due to time and resource constraints, the runtime in our numerical experiments are not consistent, let alone to compare them with others.

Our work can be extended in many directions. Heuristics in [2, 8, 23] can be used as a warm start for the **ADMM** algorithm. The group symmetry structure of the **QAP** can be exploited to improve the efficiency of the algorithm, see [10, 11]. The idea of strong branching in [2] can potentially boost our branch and bound algorithm, in which cheap bounds for potential subproblems are used to make the branching decision. Last but not the least, a proper parallel computing implementation with depth first search, as described in [6, 8], can push the limit of our algorithm and enable us to test our algorithm on even larger **QAP** instances.

APPENDICES

Appendix A

Computer Specifications

Index	Name	Make/Model	CPUs	Memory	MATLAB version
1	cpu135.math	Dell PowerEdge M630	Intel Xeon E5-2637 v3 4-core 3.5 GHz (Haswell) $\times$ 2	64 GB	R2015a
2	cpu137.math	Dell PowerEdge M630	Intel Xeon E5-2637 v3 4-core 3.5 GHz (Haswell) $\times$ 2	64 GB	R2015a
3	cpu139.math	Dell PowerEdge M830	Intel Xeon E5-4660 v3 14-core 2.1 GHz (Haswell) $\times$ 4	256 GB	R2015a
4	cpu131.math	Dell PowerEdge R815	AMD Opteron 6276 16-core 2.3 GHz (Interlagos) $\times$ 4	512 GB	R2015a
5	Personal Mac	Apple MacBook Pro 15	Intel Core i7-3635QM 4-core 2.4 Ghz (Ivy Bridge) $\times$ 1	8 GB	R2015b

Index

References

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

[2] K. Anstreicher, *Recent advances in the solution of quadratic assignment problems*, Math. Program., 97 (2003), pp. 27–42. ISMP, 2003 (Copenhagen).

[3] K. Anstreicher and N. Brixius, *A new bound for the quadratic assignment problem based on convex quadratic programming*, Math. Program., 89 (2001), pp. 341–357.

[4] K. Anstreicher, N. Brixius, J.-P. GOUX, and J. LINDEROTH, *Solving large quadratic assignment problems on computational grids*, Math. Program., 91 (2002), pp. 563–588.

[5] S. Boyd, N. Parikh, E. Chu, B. Peleato, and J. Eckstein, *Distributed optimization and statistical learning via the alternating direction method of multipliers*, Foundations and Trends® in Machine Learning, 3 (2011), pp. 1–122.

[6] N. W. Brixius, *Solving large-scale quadratic assignment problems*, The University of Iowa, 2000.

[7] R. Burkard, S. Karisch, and F. Rendl, *QAPLIB – a quadratic assignment problem library*, European J. Oper. Res., 55 (1991), pp. 115–119. www.opt.math.tu-graz.ac.at/qaplib/.

[8] R. E. Burkard, M. Dell'Amico, and S. Martello, *Assignment Problems, Revised Reprint*, Siam, 2009.

[9] E. Çela, *The quadratic assignment problem*, vol. 1 of Combinatorial Optimization, Kluwer Academic Publishers, Dordrecht, 1998. Theory and algorithms.

[10] E. de Klerk and R. Sotirov, *Exploiting group symmetry in semidefinite programming relaxations of the quadratic assignment problem*, Math. Program., 122 (2010), pp. 225–246.

[11] E. de Klerk, R. Sotirov, and U. Truetsch, *A new semidefinite programming relaxation for the quadratic assignment problem and its computational perspectives*, INFORMS Journal on Computing, 27 (2015), pp. 378–391.

[12] J. W. Dickey and J. W. Hopkins, *Campus building arrangement using topaz*, Transportation Research, 6 (1972), pp. 59–68.

[13] C. S. Edwards, *A branch and bound algorithm for the koopmans-beckmann quadratic assignment problem*, Mathematical Programming Study, 13 (1980), pp. 35–52.

[14] G. Finke, R. Burkard, and F. Rendl, *Quadratic assignment problems*, Annals of Discrete Mathematics, 31 (1987), pp. 61–82.

[15] A. M. Geoffrion and R. E. Marsten, *Integer programming algorithms: A framework and state-of-the-art survey*, Management Science, 18 (1972), pp. 465–491.

[16] P. Gilmore, *Optimal and suboptimal algorithms for the quadratic assignment problem*, SIAM Journal on Applied Mathematics, 10 (1962), pp. 305–313.

[17] S. Hadley, F. Rendl, and H. Wolkowicz, *A new lower bound via projection for the quadratic assignment problem*, Math. Oper. Res., 17 (1992), pp. 727–739.

[18] T. C. KOOPMANS and M. J. BECKMANN, *Assignment problems and the location of economic activities*, Econometrica, 25 (1957), pp. 53–76.

[19] P. M HAHN, W. L HIGHTOWER, T. AnneJOHNSON, M. Guignard-Spielberg, C. Roucairol, et al., *Tree elaboration strategies in branch-and-bound algorithms for solving the quadratic assignment problem*, Yugoslav Journal of Operations Research ISSN: 0354-0243 EISSN: 2334-6043, 11 (2001).

[20] T. Mautor and C. Roucairol, *A new exact algorithm for the solution of quadratic assignment problems*, Discrete Applied Mathematics, 55 (1994), pp. 281–293.

[21] P. Mirchandani and T. Obata, *Locational decisions with interactions between facilities: the quadratic assignment problem a review*, Working Paper Ps-79-1, Rensselaer Polytechnic Institute, Troy, New York, May 1979.

[22] C. Nugent, T. Vollman, and J. Ruml, *An experimental comparison of techniques for the assignment of facilities to locations*, Operations Research, 16 (1968), pp. 150–173.

[23] A. Nyberg and T. Westerlund, *A new exact discrete linear reformulation of the quadratic assignment problem*, European Journal of Operational Research, 220 (2012), pp. 314–319.

[24] D. Oliveira, H. Wolkowicz, and Y. Xu, *ADMM for the SDP relaxation of the QAP*, tech. rep., University of Waterloo, Waterloo, Ontario, 2015. submitted Dec. 15, 2015, 12 pages.

[25] P. Pardalos, K. Ramakrishnan, M. Resende, and Y. Li, *Implementation of a variance reduction-based lower bound in a branch-and-bound algorithm for the quadratic assignment problem*, SIAM J. Optim., 7 (1997), pp. 280–294.

[26] P. Pardalos, F. Rendl, and H. Wolkowicz, *The quadratic assignment problem: a survey and recent developments*, in Quadratic assignment and related problems (New Brunswick, NJ, 1993), P. Pardalos and H. Wolkowicz, eds., Amer. Math. Soc., Providence, RI, 1994, pp. 1–42.

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

[28] E. Taillard, *Robust tabu search for the quadratic assignment problem*, 17 (1991), pp. 443–455.

[29] Q. Zhao, S. Karisch, F. Rendl, and H. Wolkowicz, *Semidefinite programming relaxations for the quadratic assignment problem*, J. Comb. Optim., 2 (1998), pp. 71–109. Semidefinite programming and interior-point approaches for combinatorial optimization problems (Fields Institute, Toronto, ON, 1996).