SOLVING LARGE-SCALE QUADRATIC ASSIGNMENT PROBLEMS

by

Nathan William Brixius

An Abstract

Of a thesis submitted in partial fulfillment of the
requirements for the Doctor of Philosophy
degree in Computer Science
in the Graduate College of
The University of Iowa

December 2000

Thesis supervisor: Professor Kurt Anstreicher

ABSTRACT

The quadratic assignment problem (QAP) is the problem of finding an assignment of an equal number of facilities to locations that minimizes the transportation cost between facilities. The large number of possible assignments along with the interdependence of the cost on the flows and distances causes QAP to be an extremely difficult problem to solve in practice.

Branch-and-bound algorithms are typically used to find exact solutions to QAP. A branch-and-bound algorithm solves a QAP by assigning some facilities to locations, and computing lower bounds on these smaller subproblems. The use of lower bounds allows the algorithm to eliminate from consideration partial assignments that cannot lead to a solution, and via this process of elimination efficiently determine an optimal assignment.

This dissertation proposes a new branch-and-bound implementation capable of solving previously unsolved quadratic assignment problems. The first ingredient is a new continuous relaxation of QAP based on convex quadratic programming, resulting in an efficiently computed, effective lower bounding procedure. The new bound is a key component in an improved branch-and-bound algorithm that uses information provided by the continuous relaxation to intelligently extend the search for optimal solutions.

Even though the performance of our branch-and-bound algorithm surpasses previous implementations on a range of standard test problems, large computational resources are required. The solution of QAPs as small as thirty facilities and locations requires years of sequential computation. Grid computing resources allow hundreds of workstations to work on the solution of a QAP at once. The MW grid computing tool was used to produce an efficient, fault-tolerant parallel branch-and-bound implementation. This implementation was used to solve the Nugent 30 QAP, unsolved since 1968, over a period of seven days using 1000 machines.

Abstract approved:_____

Thesis supervisor

_____

Title and department

_____

Date

SOLVING LARGE-SCALE QUADRATIC ASSIGNMENT PROBLEMS

by

Nathan William Brixius

A thesis submitted in partial fulfillment of the
requirements for the Doctor of Philosophy
degree in Computer Science
in the Graduate College of
The University of Iowa

December 2000

Thesis supervisor: Professor Kurt Anstreicher

Graduate College
The University of Iowa
Iowa City, Iowa

CERTIFICATE OF APPROVAL

PH.D. THESIS

This is to certify that the Ph.D. thesis of

Nathan William Brixius

has been approved by the Examining Committee for the thesis requirement for the Doctor of Philosophy degree in Computer Science at the December 2000 graduation.

Thesis committee: _____
Thesis supervisor

_____
Member

_____
Member

_____
Member

_____
Member

TABLE OF CONTENTS

Page

LIST OF TABLES v

LIST OF FIGURES viii

CHAPTER

1 THE QUADRATIC ASSIGNMENT PROBLEM 1

1.1 Introduction 1
1.2 A Survey of the Quadratic Assignment Problem 5
1.2.1 Standard Forms for QAP 6
1.2.2 Difficulty of QAP 8
1.3 Applications 10
1.4 Solution Methods 14
1.4.1 Heuristics for QAP 15
1.4.2 Branch-and-Bound Algorithms 19
1.5 Lower Bounds for QAP 23
1.5.1 Gilmore-Lawler Bound 24
1.5.2 Bounds Based on Linear Programming Relaxations 26
1.5.3 Hahn-Grant Bound 29
1.5.4 Eigenvalue Bounds 31
1.5.5 Semidefinite Programming Bounds 34
1.5.6 Dynamic Programming Bound 37
1.5.7 Comparison of Lower Bounds 39
1.6 Other Solution Methods 41
1.6.1 Cutting Plane Methods 41
1.7 Software for QAP 43
1.7.1 Parallel Algorithms for QAP 45
1.8 Summary 47

2 A CONVEX QUADRATIC PROGRAMMING BOUND FOR QAP . 48

2.1 The Quadratic Programming Bound 48

2.2 Derivation of QPB 50
2.3 Finding a Dual Solution 56
2.3.1 Finding an Initial Dual Solution 57
2.3.2 Improving the Dual Solution 63
2.4 A Long-step Path Following Algorithm for QP 67
2.5 Solution Using the Frank-Wolfe Algorithm 71
2.6 Performance of QPB 77

3 BRANCH AND BOUND ALGORITHMS FOR QAP 86

3.1 Branch-and-Bound Algorithms 86
3.2 A Detailed Introduction to Branch-and-Bound 89
3.3 Node Selection Strategies 91
3.4 Heuristics 93
3.5 Branching Strategies for QAP 94
3.5.1 Single Assignment Branching 94
3.5.2 Polytomic Branching 96
3.5.3 Branching Rules for QAP 97
3.6 Exploiting QAP Symmetry 106
3.7 Specifying a Complete Branching Strategy 108
3.8 Branching-Related Experiments and Results 111

4 ESTIMATING THE PERFORMANCE OF BRANCH-AND-BOUND ALGORITHMS 121

4.1 Introduction 121
4.2 Knuth's Estimation Procedure 123
4.3 Importance Sampling 126
4.4 Results 131

5 COMPUTATIONAL RESULTS 141

5.1 Introduction 141
5.2 Parallel Implementation of Branch-and-Bound 141
5.2.1 Grid Computing 141
5.2.2 MW: An Framework for Grid Computing 143
5.2.3 MW Algorithmic Details 146
5.3 Computational Results on QAPLIB Problems 151
5.4 Results Using MWQAP on Large Problems 157
5.4.1 Solution of the Nug30 QAP 157
5.4.2 MWQAP results on other large problems 168

6 MODIFICATIONS AND EXTENSIONS 170

6.1 An Alternate Quadratic Programming Bound 170
6.2 A Bound Improvement Procedure 172
6.3 The Parametric Eigenvalue Bound and QPB 176
6.4 Improvements to QPB 179
6.4.1 Caching Search Directions 179
6.4.2 Warmstarting QPB 181
6.4.3 Computing QPB using an Integer LAP Solver 182
6.4.4 Warmstarting LAP 184
6.5 Alternate Branching Strategies 185
6.5.1 Making Branching Decisions using GLB 185
6.5.2 Lower Bounds Using GLB 186
6.5.3 Non-polytomic Branching Strategies 187
6.6 A New Heuristic for QAP Based on QPB 189
6.7 Future Work and Extensions 190

APPENDIX A NOTATION AND DEFINITIONS 193

A.1 Notation 193
A.2 Optimization Problems 194
A.2.1 Linear Programming 194
A.2.2 Quadratic Programming 195
A.2.3 Semidefinite Programming 195
A.2.4 Integer Programming 196
A.3 Linear Assignment Problems 197
A.4 Machine Characteristics 199

APPENDIX B MWQAP USER'S GUIDE 200

B.1 Solving QAPs using MWQAP 200
B.1.1 Providing Problem Data to MWQAP 200
B.1.2 Interpreting the Output 206
B.2 Additional Features 210
B.2.1 Command-line Options 210
B.2.2 Using the Estimator 211

REFERENCES 213

LIST OF TABLES

Table Page

1.1 Comparison of heuristic methods measured in percent above best known value (BKV) 18

1.2 Comparison of different lower bounds on QAPLIB test set 40

2.1 Bounds for QAPLIB problems 80

2.2 A comparison of QPB-IP and QPB-FW 81

2.3 Parametric improvement procedure applied to QPB-IP 82

2.4 Performance of QPB-FW on nug20 using dual update every NUPDATE iterations 83

3.1 Summary of branching rules 106

3.2 Branching Strategy A 110

3.3 Branching Strategy B 110

3.4 Comparison of branching strategies on scr15 114

3.5 Comparison of Rules 2 and 4 114

3.6 Comparison of Rules 1 and 2 116

3.7 Comparison of Rules 3 and 4 117

3.8 Branching Strategy C 119

3.9 Branching Strategy D 119

3.10 Effects of gap-based branching on nug24 120

4.1 Actual vs. estimated performance on nug20 127

4.2 Actual vs. estimated performance on nug25 128

4.3 Actual vs. estimated performance on QAPLIB problems 132

4.4 Nug30 branching strategy 1 135

4.5 Nug30 estimate 1: Overall statistics 135

4.6 Nug30 estimate 1: Breakdown of strategies used 136

4.7 Nug30 estimate 1: Relative gap information 136

4.8 Nug30 branching strategy 2 137

4.9 Nug30 estimate 2: Overall statistics 137

4.10 Nug30 estimate 2: Breakdown of strategies used 138

4.11 Nug30 estimate 2: Relative gap information 138

5.1 MWQAP parameters 150

5.2 Branch-and-bound performance on QAPLIB problems 152

5.3 Branching rules for QAPLIB problems 153

5.4 Branching strategies for QAPLIB problems (Depth/Gap) 154

5.5 Number of nodes to solve QAPLIB problems 155

5.6 Equivalent CPU time (m) to solve QAPLIB problems 155

5.7 Nug30 run statistics 159

5.8 Nug30 computational pool 160

5.9 Contribution of each location during nug30 run 160

5.10 Contribution of each architecture during nug30 run 161

5.11 Nug30 branching strategy 161

5.12 Nug30: Overall statistics 162

5.13 Nug30: Breakdown of strategies used 163

5.14 Nug30: Relative gap information 164

6.1 QP-S bounds on QAPLIB problems 173

6.2 QPB-GLB/imp on QAPLIB problems 176

6.3 QPB-EVB3 bound on QAPLIB problems 177

6.4 Effect of different scaling factors for QAPLIB problems 184

6.5 Performance of limited enumeration procedure 190

A.1 Machine and compiler characteristics 199

B.1 Lower bounds supported by MWQAP 204

B.2 Summary of MWQAP parameters 205

B.3 Command-line options 211

LIST OF FIGURES

Figure		Page
1.1	A quadratic assignment problem, $n = 4$	2
1.2	Optimal assignment for previous problem	2
1.3	Distance and flow matrices for previous example	6
1.4	Graph with bandwidth 2	10
1.5	Eye movement link values between aircraft instruments during climbing maneuver with constant heading	13
1.6	Binary branching strategy	22
1.7	Polytomic branching strategy	22
1.8	Distances for the nug06 QAP	23
1.9	A QAP subproblem	38
1.10	Example of a cutting plane	42
2.1	High-level algorithm for computing QPB	55
2.2	Structure of optimal basis	59
2.3	Algorithm to find an initial dual basis	62
2.4	Algorithm to compute S', T' needed by QPB	63
2.5	Algorithm to update dual basis	66

2.6 Frank-Wolfe algorithm 72

2.7 Solving QP(A, B, C) using the FW algorithm 74

2.8 Comparison of solution procedures for QPB on nug20 75

2.9 Computing QPB using the FW algorithm 77

2.10 Convergence of FW algorithm as problem size increases 81

2.11 Convergence of FW algorithm by depth of problem 84

2.12 Lower bounds for QAP by depth of problem 85

3.1 Branch-and-bound tree 87

3.2 Generic branch-and-bound algorithm 89

3.3 Single assignment strategy for QAP 94

3.4 Binary branching may produce infeasible subproblems 97

3.5 Polytomic branching strategy for QAP 98

3.6 Branching Rule 1 99

3.7 Branching using reduced cost matrix U 102

3.8 Bounds are computed only for NBEST rows and columns 103

3.9 Branching using prospective bound computation 104

3.10 Branching Rule 4: a "look-ahead" rule 105

3.11 Symmetry of nug06: $J_1 = \{1, 2\}$ 107

3.12 Symmetry of nug06: $J_2 = \{2, 5\}, J_3 = \{1, 2, 4, 5\}$ 107

3.13 Branch-and-bound algorithm for QAP 112

3.14 Relative gaps of level 3 problems for nug20 115

3.15 Ranks of rows/columns selected for nug20 117

4.1 Knuth's Estimation Procedure 122

4.2 A simplified branch-and-bound algorithm 124

4.3 An estimation procedure for branch-and-bound 125

4.4 Distribution of dives for nug25 using $q = 0, q = 2$ 132

4.5 Actual and estimated number of nodes on nug25, $q = 2$ 133

5.1 A Master-Worker algorithm for branch-and-bound 145

5.2 Size of task pool using lazy best-first search 148

5.3 Equivalent CPU time (m) to solve nugxx problems 156

5.4 History of solution of Nugent QAPs 158

5.5 Number of workers participating in nug30 solution 166

5.6 Thousands of LAPs solved during nug30 solution 166

5.7 Size of master task pool during nug30 solution 167

6.1 Bound improvement procedure 175

6.2 Caching search directions for nug15 180

6.3 Convergence of FW algorithm using different starting points 181

6.4 QAP heuristic based on QPB 189

B.1 MWQAP parameters file 202

B.2 Symmetry file for nug06 206

B.3 Estimator parameters file 211

CHAPTER 1
THE QUADRATIC ASSIGNMENT PROBLEM

1.1 Introduction

This dissertation examines new techniques for solving large quadratic assignment problems. The quadratic assignment problem (QAP) is an extremely difficult type of discrete optimization problem with applications in economics, circuit design, and ergonomics. To those in the scientific computing community, QAP is a challenge problem requiring massive amounts of computational resources. The solution of increasingly larger QAP instances requires both algorithmic and parallel computing advances. Two new algorithmic techniques are presented in this thesis. The first is a new continuous relaxation of QAP that is used to reduce the search for an optimal solution to a more manageable size. The second is an improved branch-and-bound algorithm that uses information provided by the continuous relaxation to intelligently extend the search for optimal solutions. Even with these advances, the solution of the largest QAPs requires years of sequential computation. Grid computing resources allow hundreds of workstations to work on the solution of a QAP at once. Algorithmic advances in conjunction with the MW grid computing system have resulted in an implementation capable of solving previously unsolved quadratic assignment problems.

The quadratic assignment problem is the problem of finding an assignment

Figure 1.1: A quadratic assignment problem, $n = 4$

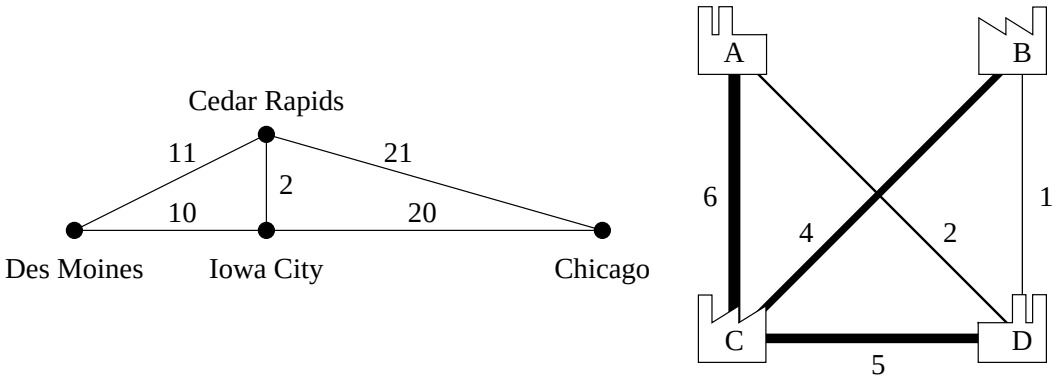
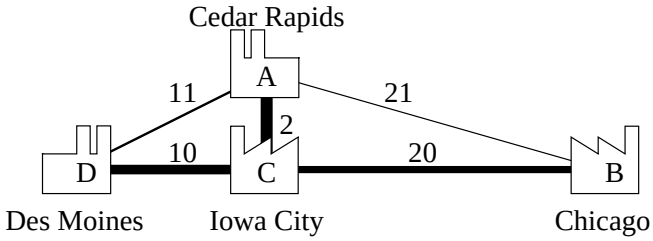


Figure 1.2: Optimal assignment for previous problem



of an equal number of facilities to locations that minimizes the transportation cost between facilities. A small quadratic assignment problem with four facilities and locations is given in Figure 1.1. The known quantities are the flow of materials between facilities, and the distances between locations, and the question is where to locate each facility. The cost of transporting materials between two facilities is the product of the amount of material to be shipped, and the distance between the two facilities. To minimize the transportation cost, it seems that facilities that ship large

quantities of material to each other should be placed as close together as possible. The minimum cost assignment in our example has cost 378 and is shown in Figure 1.2.

There are 24 different possible assignments of facilities to locations in the problem of Figure 1.1, and in general a QAP with n facilities and locations has $n!$ different solutions. The large number of solutions along with the interdependence of the cost on the flows and distances causes QAP to be an extremely difficult problem to solve in practice. A QAP with 20 facilities and locations is considered challenging to solve to optimality.

Branch-and-bound algorithms are to date the most effective exact solution procedure for QAP. A branch-and-bound algorithm solves QAPs by assigning some of the facilities to locations, and computing lower bounds on the partial assignments. By using these lower bounds, the algorithm can hopefully eliminate from consideration most of the possible assignments, and via this process of elimination obtain a solution to the problem. For example, in the problem of Figure 1.2 a branch-and-bound algorithm may quickly rule out the assignment of Facility C to Chicago since the flows out of Facility C and the distances to all other locations are relatively high.

The effectiveness of a branch and bound algorithm depends fundamentally on the procedure used to obtain lower bounds. A lower bound is used by a branch-and-bound algorithm to answer the question "Can this partial assignment lead to an improved solution to the problem?" The bounding procedure must be fast, and

produce bounds that are as high as possible. A bounding procedure that is fast but reports low bounds will be ineffective because it will not be able to reject many partial assignments. Even if the bounding procedure produces good bounds, to solve a large problem it will be necessary to use it millions of times, and powerful computational platforms become necessary. Grid computing environments, which consist of very large collections of computers connected by a network such as the Internet, provide such a platform.

The branch-and-bound implementation described in this thesis uses a new lower bounding procedure, improved search strategies, and grid computing resources to solve QAPs of unprecedented size. Among the newly solved problems is a QAP with thirty facilities and locations proposed by Nugent et al. in 1968 (see [71]), long considered a challenge problem. Until recently it was thought that this problem was too difficult to be solved using current techniques. The solution of nug30 required the use of over 1000 machines over a one-week period, and would have required nearly seven years of computation on a single fast workstation.

This dissertation examines the components of our successful branch-and-bound implementation. After a survey of QAP and current techniques for its solution, in Chapter 2 we propose a new lower bound for QAP based on convex quadratic programming. Chapter 3 embeds the new lower bound into a branch-and-bound algorithm with new branching strategies. Chapter 4 presents a framework for determining which problems can likely be solved by our algorithm, and how best to solve them.

Chapter 5 describes the grid computing framework used and challenges in producing an efficient distributed implementation of our algorithm, and presents detailed computational results on a set of standard test problems.

Throughout our discussion we present the results of various computational experiments. Unless otherwise stated, the experiments were performed on an HPC3000 workstation, whose characteristics are given in more detail in Appendix A.4. Appendix A also provides a summary of the notation used, and proofs of some basic results.

1.2 A Survey of the Quadratic Assignment Problem

Next we consider a more rigorous formulation of QAP. We follow with a discussion of the difficulty of QAP and with some descriptions of the applications of QAP. We then turn our attention to solution methods for QAP, first examining the effectiveness of heuristics such as simulated annealing and genetic algorithms, as well as other heuristics which have been tailored specifically for QAP. The exact solution of QAP, that is, finding an optimal assignment and proving its optimality, has long been considered a "challenge problem" in combinatorial optimization and computer science. Problems of size $n = 36$ proposed over 30 years ago (see [82]) remain open. We survey innovations both in branch-and-bound algorithms used to solve QAP, and on lower bounds based on relaxing the constraints of QAP.

Earlier QAP surveys include those of Pardalos, Rendl and Wolkowicz [74] and

Burkard, Çela, Pardalos and Pitsoulis [18]. Our primary focus is branch-and-bound algorithms for QAP, and interested readers may want to refer to [74, 18] for different perspectives.

1.2.1 Standard Forms for QAP

Figure 1.3: Distance and flow matrices for previous example

$$D = \begin{bmatrix} 0 & 0 & 6 & 2 \\ 0 & 0 & 4 & 1 \\ 6 & 4 & 0 & 5 \\ 2 & 1 & 5 & 0 \end{bmatrix}, \quad F = \begin{bmatrix} 0 & 21 & 11 & 2 \\ 21 & 0 & 30 & 20 \\ 11 & 20 & 0 & 10 \\ 2 & 20 & 10 & 0 \end{bmatrix}$$

There are several standard forms for QAP in the literature. The earliest of these is the Koopmans-Beckmann formulation of QAP (1957):

$$\min_{p \in \Pi} \sum_{i=1}^{n} \sum_{j=1}^{n} a_{ij} b_{p(i)p(j)} + \sum_{i=1}^{n} c_{ip(i)}. \tag{1.1}$$

Sometimes instead of A and B, one sees F and D for "flow" and "distance". The flow and distance matrices for the QAP shown in Figure 1.1 are given in Figure 1.3. QAP is a search for an optimal permutation p of $\{1 \dots n\}$ in the space of permutations Π, which has $n!$ members. Figure 1.1 is an example of a **homogeneous QAP** because the linear term C is zero. C is usually interpreted as a matrix of fixed costs for making assignments. We will assume throughout that the number of facilities and locations are the same, and n will always refer to this quantity. (In [50] Kaibel examines QAPs with fewer facilities than locations.)

The **trace formulation** of a Koopmans-Beckmann QAP is obtained by writing permutations as permutation matrices:

$$\min_{X \in \Pi} \operatorname{tr}\, AXBX^T + CX^T \tag{1.2}$$

where X is a permutation matrix with $x_{ij} = 1$ if $p(i) = j$, and $\operatorname{tr} M$, the **trace** of M is the sum of the diagonal elements of M. It is easily seen that the trace formulation in (1.2) is equivalent to the original Koopmans-Beckmann formulation in (1.1).

A more general form for QAP was introduced by Lawler in 1963:

$$\min_{p \in \Pi} \sum_{i=1}^{n} \sum_{j=1}^{n} d_{ijp(i)p(j)} + \sum_{i=1}^{n} c_{ip(i)}. \tag{1.3}$$

The Lawler formulation is more general than Koopmans-Beckmann because it does not require that d_{ijkl} be obtained as a product of entries of matrices A and B. Again characterizing permutations as permutation matrices, a **quadratic integer programming formulation** of Lawler's QAP is obtained:

$$\begin{aligned} \min \quad & \sum_{i=1}^{n} \sum_{j=1}^{n} \sum_{k=1}^{n} \sum_{l=1}^{n} d_{ijkl} x_{ij} x_{kl} + \sum_{i,j=1}^{n} c_{ij} x_{ij} \\ \text{s.t.} \quad & \sum_{i=1}^{n} x_{ij} = 1, \quad j = 1, 2, \ldots, n, \\ & \sum_{j=1}^{n} x_{ij} = 1, \quad i = 1, 2, \ldots, n, \\ & x_{ij} \in \{0, 1\}, \quad i, j = 1, 2, \ldots, n. \end{aligned} \tag{1.4}$$

In this formulation a quadratic function is minimized subject to linear constraints, where the variables x_{ij} are either zero or one. A Koopmans-Beckmann form QAP may be written as a Lawler QAP by letting $d_{ijkl} = a_{ik} b_{jl}$.

In many cases it is useful to represent permutations as permutation matrices. It is well known that the set of permutation matrices Π can be characterized as the intersection of the following sets of matrices:

$$\Pi = \mathcal{O} \cap \mathcal{E} \cap \mathcal{N} = \mathcal{S} \cap \mathcal{E} \cap \mathcal{N} = \mathcal{S} \cap \mathcal{D}$$

where $\mathcal{O}$ represents the set of orthogonal matrices, $\mathcal{S}$ is the set of matrices where $\operatorname{tr} X^T X = n$, $\mathcal{E}$ denotes the set of matrices with row and column sums equal to one, $\mathcal{N}$ is the set of nonnegative matrices, and the members of $\mathcal{D} = \mathcal{E} \cap \mathcal{N}$ are called doubly stochastic matrices.

As we will see in Section 1.5, various lower bounds on QAP are obtained by considering constraint sets which are supersets of Π.

1.2.2 Difficulty of QAP

QAP is an NP-hard problem, meaning that it is not likely that a polynomial time algorithm exists for its solution. As Graves and Whinston note in [40], the traveling salesperson problem (TSP), which is known to be NP-hard, can be written as a QAP. If facility i ships one unit to facility $i+1$, for $i = 1, \ldots, n-1$ and facility n ships one unit to facility 1, and the distances are the distances between cities the salesperson wants to travel to, then a solution to the resulting QAP is also a solution to the TSP.

Further, QAP belongs to a core of especially difficult NP-hard problems, in that finding an ϵ-approximate solution to QAP is also NP-hard. Let us define an algorithm to be an **ϵ-approximate algorithm** for a minimization problem P iff for

every instance of P,

$$(\hat{F} - F^*)/F^* \leq \epsilon, \epsilon > 0,$$

where F^* is the optimal solution (assumed greater than 0) and $\hat{F}$ is the approximate solution obtained by the algorithm.

Sahni and Gonzalez [79] proved that ϵ-approximation of QAP is NP-hard, for any $\epsilon > 0$ via a reduction from the Hamiltonian cycle problem. The Hamiltonian cycle problem is the problem of finding a cycle in a graph containing each vertex exactly once. The reduction is as follows.

Let $G(V, E)$ be an undirected graph with $n = |V|$. The following size n QAP is constructed from G:

$$A_{ij} = \begin{cases} 1 & \text{if } j = i + 1 \text{ and } i < m \text{ or if } i = m \text{ and } j = 1, \\ 0 & \text{otherwise.} \end{cases}$$

$$B_{ij} = \begin{cases} 1 & \text{if } (i, j) \in E, \\ \omega & \text{otherwise,} \end{cases}$$

where ω is appropriately chosen. The total cost $f(p)$ of an assignment p of facilities to locations is $\sum_{i=1}^{n} A_{ij(i)} B_{p(i),p(j(i))}$ where $j(i) = (i \bmod m) + 1$, and $p(i)$ is the index of the location assigned to facility i. If G has a Hamiltonian cycle, then there is an assignment with cost $f(p) = n$. In case G has no Hamiltonian cycle, then at least one of the values $B_{p(i),p(j(i))}$ must be ω and so the cost becomes greater than or equal to $n + \omega - 1$. Choosing $\omega > (1 + \epsilon)n$ results in optimal solutions with a value of n if G has a Hamiltonian cycle and value greater than $(1 + \epsilon)n$ if G has no Hamiltonian

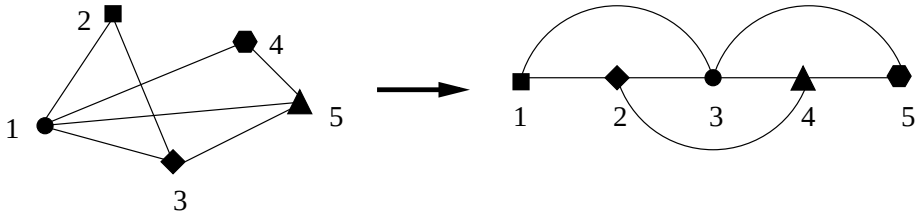
cycle. Thus from an ϵ-approximate solution it can be determined whether or not G has a Hamiltonian cycle.

Unlike some other NP-hard problems, QAP has been exceptionally difficult to solve in practice. While TSPs with thousands of cities have been solved, in general QAPs of size $n \geq 35$ cannot be solved to optimality. The solution of a size 30 QAP, described in Chapter 5 required over seven years of CPU time.

1.3 Applications

A surprising number of problems from different areas can be formulated as QAPs. As we saw in the last section, in addition to facility-location problems, graph problems such as the traveling salesperson problem can be formulated as QAPs. Unfortunately, in most cases such problems lose much of their special structure and are not efficiently solved as QAPs. A notable exception to this general rule is the bandwidth minimization problem.

Figure 1.4: Graph with bandwidth 2



Suppose a graph G with n vertices $1, 2, \ldots, n$ is given, and the vertices are

labeled $p(1), p(2), \ldots, p(n)$. The **bandwidth of** p is defined to be

$$b_G(p) = \max_{ij \in E} |p(i) - p(j)|,$$

and the **bandwidth of** G is defined to be

$$\min_{p \in \Pi} b_G(p).$$

Figure 1.4 shows a graph with bandwidth 2 – it is clear that the bandwidth is at least 2 because vertex 1 is adjacent to all other vertices. The solution of a particular QAP can be used to find a lower bound on the bandwidth of a graph, so the solution of a QAP could be one step of a procedure for solving bandwidth minimization problems; see [74] for details.

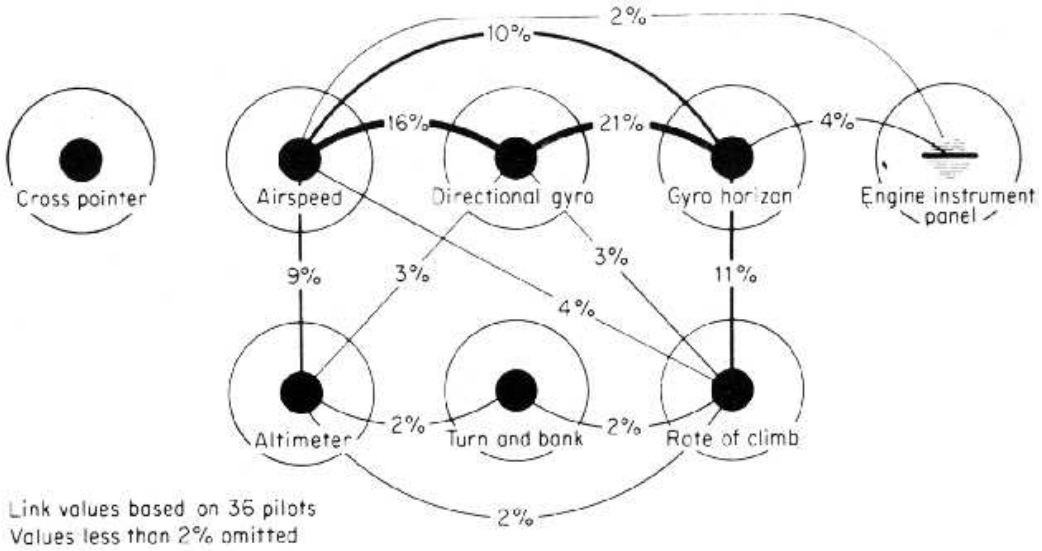
The turbine balancing problem, as described by Pitsoulis et al. in [75], is an NP-hard problem arising in the manufacturing and maintenance of turbine engines. A turbine is a collection of fans, where each fan is a disk with blades attached at equally spaced intervals. The blades are of different weights due to wear and errors in manufacture. Due to these differences in weight, when the turbine spins there is some imbalance, and the turbine balancing problem is to place the blades so that this imbalance is minimized. Again the problem is combinatorial in nature – each of the n blades must be placed in one of n available slots on the disk. The authors utilized a heuristic procedure, GRASP, to obtain approximate solutions to the resulting QAPs. Mason and Rönnqvist also used heuristics to solve a QAP formulation of jet turbine balancing in [66].

In [82], Steinberg discusses the formulation of the backboard wiring problem as a quadratic assignment problem. A set of elements are given, and certain elements must be connected to others by differing numbers of wires. Given a set of points where elements may be placed, the problem is to place each element on a point so that the total amount of wire used is minimized. This is a problem which is quite naturally expressed as a QAP, and is of obvious utility. Interestingly enough, in the original paper by Steinberg from 1960, three sample problems were given of size $n = 36$ and all three remain unsolved to this day, highlighting the difficulty of finding exact solutions to QAP.

An interesting application dating from 1976 comes from Geoffrion and Graves in [33]. The authors consider scheduling parallel production lines via a QAP formulation. More specifically, it is given that each production line l can produce R_{pl} units of each product p, and that there is a transition cost associated with converting a production line from one product to another. The problem is to obtain a production schedule that satisfies a number of orders, where each order specifies a quantity of a particular product, to be produced by a given time. Under some very mild assumptions concerning the nature of the production lines, this problem also has a QAP formulation.

Assignment problems also naturally model many problems coming from ergonomic design. A few interesting ergonomic applications involving QAP are found in [80]. For example, the placement of aircraft instruments in a cockpit so that eye

Figure 1.5: Eye movement link values between aircraft instruments during climbing maneuver with constant heading



Source: [80, Figure 13-3]

movement is minimized can be formulated as a QAP (see Figure 1.5).

Vector quantization is an image compression technique. An image is divided into rectangular sections, each represented by a vector in k-dimensional space. The vectors are quantized by matching them with vectors in a codebook of a relatively small size, say n. Each section of the image is then characterized by an index into the codebook, rather than a vector. The primary goal is to select a codebook that minimizes the distortion of the compressed image. A secondary issue arises when the indices are sent over a communications channel subject to transmission errors. In such instances, a codevector c_i may be sent, but the receiver mistakes it for c_j. Therefore, one wants to reorder the codevectors in a codebook so that distortion due

to channel noise is minimal. It is not known in advance what transmission errors will be made, but we know that certain indices will be mistaken for others more frequently because their bit patterns are similar.

The authors of [22] use a QAP formulation to minimize the distortion of a vector quantized image due to channel noise. More formally, given a codebook $C = \{c_1, \ldots, c_n\}$ with n vectors in k-dimensional space, and the probabilities of occurrence $p(i)$ of c_i, the task is to assign the codevectors to $(\log_2 n)$-bit binary codes in order to minimize the average distortion

$$\frac{1}{k} \sum_{i=1}^{n} p(i) \sum_{j=1}^{n} p(j|i) d(c_i, c_j)$$

where $d(c_i, c_j)$ is the euclidean distance between codevectors c_i and c_j, and $p(j|i)$ is the probability that c_j is mistaken for c_i. The problem can be written as a QAP, and in [22] a heuristic method is proposed for its solution.

1.4 Solution Methods

Finding optimal solutions to QAPs of size as small as $n = 20$ can be quite computationally intensive. Therefore, in many situations heuristics are used to generate good suboptimal solutions, although as noted in the previous section such performance is not guaranteed. We explore some of the proposed heuristics for QAP in the next subsection.

1.4.1 Heuristics for QAP

A simple local search algorithm for QAP is presented in [74]. The algorithm starts with an initial feasible solution and moves to neighboring permutations until no further improvement is possible.

GRASP (Greedy Randomized Adaptive Search Procedure, see [60, 72, 77]) is an iterative procedure that generates a series of approximate solutions to a discrete optimization problem such as QAP. Each iteration consists of two stages: a **construction phase** whereby a feasible solution (i.e. permutation) is generated, and an **improvement phase** that searches in the neighborhood of the initial solution. In many implementations of GRASP [60] the best solution over all GRASP iterations is kept, and it is possible to use information from previous iterations to bias the construction phase of later iterations.

By varying the construction and improvement phases, different variants of GRASP are obtained. For example, in [75] the construction phase creates an initial permutation p by first making two assignments simultaneously so as to minimize the cost of the assignment, and then randomly making the last $n-2$ assignments, being biased toward low cost assignments. The improvement phase searches within the 2-exchange neighborhood of p (denoted $N_2(p)$), i.e. the space of permutations which differ by two exchanges from p. For example, $(2,1,4,3) \in N_2(1,2,3,4)$, but $(4,1,2,3) \notin N_2(1,2,3,4)$.

The hybrid ant system of [32] is the foundation of an effective heuristic proce-

dure for QAP. Ant systems model the behavior of ants to find solutions to combinatorial optimization problems. A general description of ant systems is found in [28]; let us focus specifically on ant systems and QAP. An ant system for QAP consists of a number of ants, with each ant in control of a permutation. The ants update a shared "pheromone matrix" T where t_{ij} is a measure of how desirable the assignment $p(i) = j$ is. Ants are likely to be drawn to high pheromone areas in the space of permutations.

Each ant in the ant system is initially given a random permutation. The permutation is first modified by performing a series of swaps determined in part by T, then by a local search. At the end of each phase, each ant updates the pheromone matrix T – the better the solution, the more pheromone left behind. In subsequent iterations, areas of the search space yielding high quality solutions will likely be explored further. When the search stagnates, ants can be given new random permutations, or T can be erased.

Tabu search [47] is a search technique that iteratively searches in the neighborhood of the current solution. The main characteristic of a tabu search algorithm is that it stores a list of prohibited ("tabu") moves that are to be avoided. This list can be used to prevent the algorithm from being stuck at a local minimum, or from entering a cycle. Typically moves are kept on a list for a specified period of time t. In a reactive tabu search t is determined through feedback from the algorithm. In sophisticated implementations of tabu search algorithms there is also an aspiration

function which sometimes replaces the normal objective. The aspiration function serves to either intensify or diversify the search, depending on what is required. For a description of tabu search as applied to QAP, see [84].

Simulated annealing is another heuristic that attempts to prevent a neighborhood search algorithm from getting stuck at a local minimum. The term "simulated annealing" is a reference to the annealing of metals, whereby a metal is cooled gradually over time according to a schedule, so as to avoid an undesirable final state. Similarly, a simulated annealing algorithm seeks to decrease the objective of the solution gradually, and occasionally permits moves that increase the objective. The original application of simulated annealing to QAP is [86]. Connolly proposed an improved simulated annealing algorithm in [25].

Genetic algorithms have been used to solve a number of difficult optimization problems, see [36]. In a genetic algorithm, each element of the search space is mapped to a chromosome, typically represented as a string of bits. The operations of reproduction, crossover and mutation are applied to successive generations of a population of chromosomes. By choosing the fittest members of each generation to be the most likely to reproduce, it is hoped that the population eventually converges to an optimal solution to the problem at hand. A first application of genetic algorithm techniques is found in [31]. A greedy genetic algorithm for QAP of Ahuja, Orlin and Tirwari [2] gave particularly good results on the QAPLIB [19] test set.

In Table 1.1 we compare several heuristic methods for QAP on problems from

Table 1.1: Comparison of heuristic methods measured in percent above best known value (BKV)

problem	BKV	TT	RTS	SA	GH	HAS
nug20	2570	0.0	91.1	7.0	0.0	0.0
nug30	6124	3.2	87.2	12.1	0.7	9.8
sko42	15812	3.9	111.6	11.4	0.3	7.6
sko49	23386	6.2	97.8	13.3	4.0	14.1
tai20a	703482	21.1	24.6	71.6	26.8	67.5
tai25a	1167256	51.0	34.5	100.2	62.9	118.9
tai30a	1818146	34.0	28.6	90.7	43.9	131.1
tai35a	2422002	75.7	35.5	134.5	69.8	176.2
tai40a	3139370	100.6	62.3	130.7	88.4	198.9
tai50a	4941410	114.5	83.4	153.9	104.9	280.0
wil50	48816	4.1	50.4	6.1	3.2	6.1
chr25a	3796	696.5	988.9	1249.7	269.2	308.2
els19	17212548	0.0	9.0	1853.9	0.0	0.0
kra30a	88900	47.0	200.8	146.6	13.4	63.0
kra30b	91420	5.9	71.2	19.5	5.4	7.1
tai20b	122455319	0.0	-	673.0	0.0	9.1
tai25b	344355646	0.7	-	112.2	0.0	0.0
tai30b	637117113	5.5	-	440.8	0.1	0.0
tai35b	283315445	17.8	-	317.5	10.7	2.6
tai40b	637250948	20.8	-	456.5	21.1	0.0
tai50b	458821517	29.4	-	81.1	21.4	19.2

Source: Tables 2 and 4 of [32]

QAPLIB: the genetic hybrid method of Fleurent and Ferland [31] (GH), the reactive tabu search of Battiti and Tecchiolli [8] (RTS), the tabu search of Taillard [84] (TT), the simulated annealing approach from Connolly [25] (SA), and the hybrid ant/local search (HAS) technique in [32]. The table shows that for some problems, it is not too difficult to obtain good quality suboptimal solutions for QAP, while for others finding a solution anywhere close to optimal can be quite difficult.

The structure of the flow and distance matrices A and B need to be considered when selecting a heuristic. As [32] notes, and is readily seen from the results in Table 1.1, heuristics often perform differently on real world problems with irregular data (below the line) than on randomly generated problems (above the line). The irregularity of a problem can be measured by a statistic known as flow dominance – the ratio between the standard deviation and mean of the values in each matrix.

1.4.2 Branch-and-Bound Algorithms

The fundamental difficulty in solving QAP is that it is too hard to search for an optimal solution in the space of permutations, due to the interdependence of the flows and distances in the cost.

In a more general situation, one may be asked to optimize over a search space H which is difficult to work with. Perhaps there exists a larger space $E \supseteq H$ which is more easily characterized and easier to work with. Since it is too difficult to search in H, a branch-and-bound algorithm searches for an optimal solution in E, which is a lower bound on the solution to H. If the solution in E also happens to be in H,

the original problem is solved. Otherwise, the algorithm divides H into pieces, the union of which contains H, and applies the bounding procedure to each piece.

The idea of relaxing the constraints of a difficult problem is quite common. For example in computer graphics, in a raytracing or radiosity algorithm a complex, wavy surface may be decomposed into a collection of triangles, and computations performed with respect to the triangles instead of the complicated surface. The best results are obtained when the relaxed problem E is as close to H as possible, yet is still easy to work with. Sometimes we may even lift the problem into a higher dimensional space to find such a search space E.

More specifically, a branch-and-bound algorithm begins with a minimization problem P to solve, and an upper bound on its solution v. A lower bound z on the solution to P is found via a relaxation of P, which is presumably simple to calculate. If the lower bound z equals the solution value v, then no better solution can exist. Otherwise, a better solution may lie in the search space of P. In such a case, P is divided into subproblems, which are then processed by the algorithm. The main phases of a branch-and-bound algorithm are to select a subproblem to process, to compute lower bounds, and to subdivide problems (or "branch").

A branch-and-bound algorithm generates a tree, where each node represents a subproblem to be solved. The activity at each node consists of computing a lower bound on the subproblem, and then generating child subproblems ("branching"). Computing lower bounds allows us to prune (or "fathom") the tree along paths that

cannot lead to an optimal solution.

In the case of QAP, the initial, root problem is to find a permutation (assignment) that minimizes a quadratic objective function. To branch, the space of permutations is divided by either disallowing or forcing assignments to be made in each child subproblem. Typically, branch-and-bound algorithms spend most of their time computing lower bounds, and thus the bounding phase is considered the most important phase of the algorithm. There have been numerous articles written concerning bounding techniques for QAP, which are surveyed later. For now we concentrate on the other parts of the branch-and-bound algorithm.

The search space for QAP is the set of permutations, which as noted earlier has $n!$ members. A permutation p corresponds to an assignment of facilities to locations, for example, facility 1 is assigned to location $p(1)$. One common technique for creating child subproblems is to utilize **single assignment branching**, pictured in Figure 1.6. One child is obtained by selecting an unassigned facility and fixing it to an unassigned location: $p(i) = j$, and the other disallows this assignment. An alternative to single assignment branching is polytomic branching, introduced by Mautor and Roucairol [67]. Its distinguishing feature is that a given node can generate more than two child nodes. In polytomic branching, we choose a facility and assign it to all possible locations, see Figure 1.7. Therefore the root problem will have at most n children, those children will have at most $n - 1$ children, and so on. It is clear that the depth of a branch-and-bound tree can be no more than n when using this scheme. In either

Figure 1.6: Binary branching strategy

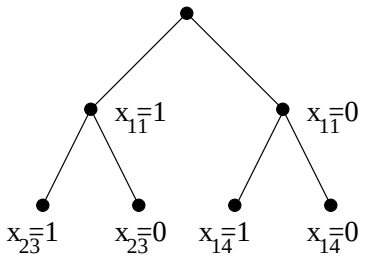
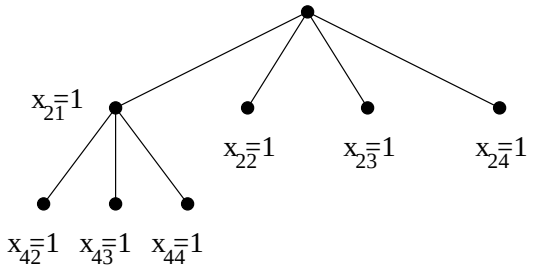


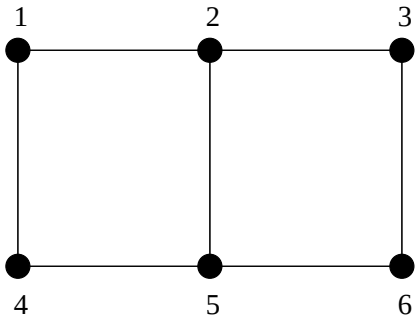
Figure 1.7: Polytomic branching strategy



single assignment or polytomic branching, nodes of the branch-and-bound tree are partial permutations, where some of the facilities have been assigned to some of the locations, and other assignments may have been disallowed.

In many cases, the QAP distance matrix has a special structure that can be exploited in a branch-and-bound algorithm. For example, if the locations are points on a rectangular grid then different partial permutations may be equivalent. The distance matrix of the QAPLIB test problem Nugent 6 corresponds to ℓ_1 distances for the graph shown in Figure 1.8. Partial permutations assigning only one facility i

Figure 1.8: Distances for the nug06 QAP



to locations 1,3,4, or 6 are all equivalent in the above example. Symmetry like this can be taken advantage of in a branch-and-bound algorithm regardless of the lower bound or branching strategy used, and often provides a significant benefit. Mautor and Roucairol [67], Brüngger et al. [13] among others embedded logic in their branch and bound implementations to detect such symmetries and take advantage of them to avoid creating redundant subproblems.

1.5 Lower Bounds for QAP

Tight, efficiently computed lower bounds are the key to an effective branch-and-bound algorithm. The various lower bounds for QAP differ in three major areas:

1. Which QAP formulation is used?

2. How is the formulation relaxed?

3. How is the relaxation solved?

We begin with lower bounds that use linear programming relaxations of QAP.

1.5.1 Gilmore-Lawler Bound

The Gilmore-Lawler lower bound (GLB) for QAP was proposed independently by Lawler [57] and by Gilmore [35] in the early 1960s, and since that time has found extensive use in branch-and-bound algorithms. It has the advantage of being cheap to compute but the bounds have proved to be too weak to be used to solve larger problems. The GLB can be obtained by relaxing an equivalent integer linear programming formulation of QAP, which we now present.

Using the more general Lawler formulation of QAP, we let $y_{ijkl} = x_{ij}x_{kl}$. Gathering the y_{ijkl} in a matrix, we have

$$Y = X \otimes X = \begin{bmatrix} x_{11}X & x_{12}X & \dots & x_{1n}X \\ x_{21}X & x_{22}X & \dots & x_{2n}X \\ \vdots & & & \vdots \\ x_{n1}X & x_{n2}X & \dots & x_{nn}X \end{bmatrix}. \tag{1.5}$$

So we write QAP as

$$\begin{aligned} \min \sum_{i=1}^{n}\sum_{j=1}^{n}\sum_{k=1}^{n}\sum_{l=1}^{n} d_{ijkl}y_{ijkl} + \sum_{i=1}^{n}\sum_{j=1}^{n} c_{ij}x_{ij} & \\ \text{s.t.} \sum_{i=1}^{n} x_{ij} &= 1, \quad j = 1, 2, \dots, n, \\ \sum_{j=1}^{n} x_{ij} &= 1, \quad i = 1, 2, \dots, n, \\ Y &= X \otimes X, \\ x_{ij} &\in \{0, 1\}, \quad i, j = 1, 2, \dots, n. \end{aligned} \tag{1.6}$$

Since $X \in \Pi$ implies $Y \in \Pi$, (1.6) would be a linear assignment problem in

n^2 variables if not for the constraint (1.5). The Gilmore-Lawler bound, and indeed several other lower bounds for QAP, are motivated by solving an LAP while trying as best as possible to impose (1.5). Looking at the (i, j) submatrix of size $n \times n$ in (1.5), we notice that each contains either n 1's or no 1's at all, depending on whether x_{ij} is 1 or 0. The Gilmore-Lawler bound tries to produce a matrix Y which is not necessarily the Kronecker product of a permutation matrix with itself, but which does satisfy at least the condition just described.

Let us define LAP(M) to refer to the objective value of the solution of the linear assignment problem with cost matrix M, and $D^{(i,j)}$ to be the (i, j) submatrix of D. That is, $D^{(i,j)}$ is the $n \times n$ matrix with $D^{(i,j)}_{kl} = d_{ijkl}$. Now consider the matrix F, where $f_{ij} = \text{LAP}(D^{(i,j)}) + c_{ij}$, and let $X^{(i,j)}$ be the solution matrix of LAP($D^{(i,j)}$). The Gilmore-Lawler bound is obtained by solving LAP(F). Notice that

$$Y = \begin{bmatrix} z_{11}X^{(1,1)} & z_{12}X^{(1,2)} & \dots & z_{1n}X^{(1,n)} \\ z_{21}X^{(2,1)} & z_{22}X^{(2,2)} & \dots & z_{2n}X^{(2,n)} \\ \vdots & & & \vdots \\ z_{n1}X^{(n,1)} & z_{n2}X^{(n,2)} & \dots & z_{nn}X^{(n,n)} \end{bmatrix}, \tag{1.7}$$

where Z is the solution matrix of LAP(F), satisfies the condition that each submatrix $z_{ij}X^{(i,j)}$ is either the zero matrix or a permutation matrix, and therefore $D \bullet Y + C \bullet Z = F \bullet Z$ is a lower bound for QAP. An interpretation of the f_{ij} is that f_{ij} is a lower bound on the cost incurred from setting $x_{ij} = 1$.

In the case where the QAP is in Koopmans-Beckmann form, we can compute GLB by solving only one LAP, resulting in an $O(n^3)$ algorithm. Define the **minimal**

vector product:

$$\langle x, y \rangle_- = \min_{P \in \Pi} \langle x, Py \rangle .$$

(It is easy to show that to compute $\langle x, y \rangle_-$, it suffices to sort x and y in opposite orders and take the dot product.) Also, let $\hat{a}_i$ be the elements in row i of A, excluding a_{ii}, and let $\hat{b}_j$ be the elements in column j of B, excluding b_{jj}. Define the matrix F as follows:

$$f_{ij} = a_{ii}b_{jj} + \langle \hat{a}_i, \hat{b}_j \rangle_- + c_{ij}, \quad i, j = 1, \ldots, n.$$

The Gilmore-Lawler bound is then exactly LAP(F).

1.5.2 Bounds Based on Linear Programming

Relaxations

We have already seen one linearization of QAP, namely the Lawler linearization (1.6) used to derive the Gilmore-Lawler bound. It turns out that a number of bounds for QAP may be obtained from the dual of a different linearization. Among these bounds are the Gilmore-Lawler bound [35, 57], the bound of Carraresi and Malucelli [21], the bound of Assad and Xu [6], the bound of Adams and Johnson [1], and the bound of Hahn and Grant [45].

Let us refocus on the constraint in (1.6) that $Y = X \otimes X$, or equivalently $y_{ijkl} = x_{ij}x_{kl}$. Since the (i, j) block of Y is $x_{ij}X$, the row and columns of block (i, j) must sum to x_{ij}. Using this fact, Lawler's linearization (1.6) is equivalent to:

$$
\begin{aligned}
\min \quad & \sum_{i=1}^{n}\sum_{j=1}^{n}\sum_{k=1}^{n}\sum_{l=1}^{n} d_{ijkl}y_{ijkl} + \sum_{i=1}^{n}\sum_{j=1}^{n} c_{ij}x_{ij} \\
\text{s.t.} \quad & \sum_{l=1}^{n} y_{ijkl} = x_{ij}, \qquad i,j,k = 1,\ldots,n, \\
& \sum_{k=1}^{n} y_{ijkl} = x_{ij}, \qquad i,j,l = 1,\ldots,n, \\
& \sum_{i=1}^{n} x_{ij} = 1, \qquad j = 1,\ldots,n, \\
& \sum_{j=1}^{n} x_{ij} = 1, \qquad i = 1,\ldots,n, \\
& x_{ij} \in \{0,1\}, \quad i,j = 1,\ldots,n, \\
& y_{ijkl} \in \{0,1\}, \quad i,j,k,l = 1,\ldots,n.
\end{aligned}
\tag{1.8}
$$

Using the fact that $y_{ijkl} = x_{ij}x_{kl}$ allows us to eliminate some of the constraints in (1.8). Since permutations are 0-1 matrices, $y_{ijij} = x_{ij}x_{ij} = x_{ij}$. This fact allows us to assume that $d_{ijij} = 0$, otherwise we could move these costs to c_{ij}. Also $y_{ijkl} = x_{ij}x_{kl} = x_{kl}x_{ij} = y_{klij}$, the so-called **complementary constraints**, see [52]. Lastly, consider $y_{ijil} = x_{ij}x_{il}$, where $j \neq l$. Since X is a permutation matrix, one of x_{ij} and x_{il} must be zero, so $y_{ijil} = 0$ if $j \neq l$. Therefore without loss of generality, we may assume $d_{ijil} = 0$ if $j \neq l$, and similarly $d_{ijkj} = 0$ if $i \neq k$. In [52], the elements of D that are assumed to be zero, and the corresponding entries of Y are called **disallowed**. Adding the complementary constraints, and removing disallowed variables, we obtain the following mixed integer linear programming (MILP) formulation for QAP proposed

by Adams and Johnson in [1] and analyzed further in [52]:

$$
\begin{aligned}
\min \quad & \sum_{i=1}^{n}\sum_{j=1}^{n}\sum_{k=1}^{n}\sum_{l=1}^{n} d_{ijkl}y_{ijkl} + \sum_{i=1}^{n}\sum_{j=1}^{n} c_{ij}x_{ij} \\
\text{s.t.} \quad & \sum_{l=1,l\neq j}^{n} y_{ijkl} = x_{ij} \qquad i,j,k = 1,\ldots,n, i \neq k, \\
& \sum_{k=1,k\neq i}^{n} y_{ijkl} = x_{ij} \qquad i,j,l = 1,\ldots,n, j \neq l, \\
& \sum_{i=1}^{n} x_{ij} = 1 \qquad j = 1,\ldots,n, \\
& \sum_{j=1}^{n} x_{ij} = 1 \qquad i = 1,\ldots,n, \\
& y_{ijkl} = y_{klij} \qquad i,j,k,l = 1,\ldots,n, i < k, l \neq j, \\
& x_{ij} \in \{0,1\} \qquad i,j = 1,\ldots,n, \\
& y_{ijkl} \geq 0 \qquad i,j,k,l = 1,\ldots,n, i \neq k, l \neq j.
\end{aligned}
\tag{1.9}
$$

A linear program is obtained by relaxing the integrality constraints on the x_{ij}. By inspection we see that (1.9) has $2n^2(n-1) + n^2(n-1)^2/2 + 2n$ constraints and $n^2[(n-1)^2+1]$ variables. For $n = 20$ this comes to 87440 constraints and 144800 variables, which is too large an LP to solve in a reasonable amount of time using current methods. Resende et al. [78] used both the simplex method and an interior-point method to solve the linear program (1.9). For a particular QAPLIB [19] test problem of size 20, the simplex method was unable to solve (1.9) in under a day, and the interior-point method took over an hour to solve (1.9). (Adams and Johnson [1] overcome this difficulty by using a subgradient procedure to find suboptimal solutions to the Lagrangian dual of (1.9), which results in lower bounds for QAP.)

The approach taken in [52] is to form the dual of (1.9):

$$
\begin{aligned}
& \max \sum_{i=1}^{n} \lambda_i + \sum_{j=1}^{n} \mu_j \\
\text{s.t.} \quad & \sigma_{ijk} + \theta_{ijl} + \alpha_{ijkl} + \beta_{ijkl} = d_{ijkl} && i,j,k,l = 1,\ldots,n, i < k, l \neq j, \\
& \sigma_{ijk} + \theta_{ijl} - \alpha_{klij} + \beta_{ijkl} = d_{ijkl} && i,j,k,l = 1,\ldots,n, i > k, l \neq j, \\
& \lambda_i + \mu_j - \sum_{k \neq i}^{n} \sigma_{ijk} - \sum_{l \neq j}^{n} \theta_{ijl} + \gamma_{ij} = c_{ij} && i,j = 1,\ldots,n, \\
& \beta_{ijkl} \geq 0 && i,j,k,l = 1,\ldots,n, k \neq i, l \neq j \\
& \gamma_{ij} \geq 0 && i,j = 1,\ldots,n.
\end{aligned}
\tag{1.10}
$$

Finding an exact solution to (1.10) is also difficult, but at least we know that feasible solutions give lower bounds on QAP. Karisch et al. show in [52] that many known bounds are variations on a simple algorithmic framework for solving (1.10). Each step of the iterative procedure consists of solving n^2 LAPs (one for each $n \times n$ block of D) and then using these results to form and solve one more LAP, which provides an updated lower bound. We refer to [52] for the details. Since each of the above bounds is obtained by an approximate solution procedure for (1.10), there is obviously no way any of those bounds can be better than the exact solution to (1.9) or (1.10).

1.5.3 Hahn-Grant Bound

Although as shown in [52] the bound of Hahn and Grant [45] fits into the framework described in the last section, it is worth looking at from a different angle. The bound of Hahn and Grant is motivated by trying to extend the Hungarian method for LAP to QAP. The Hungarian method applies a series of reductions to the cost

matrix D, transforming it into a permutation matrix which is the solution of the LAP. In the spirit of these reductions, Hahn and Grant propose two classes of operations on D.

Class 1: Addition (or subtraction) of a constant to all allowed elements of a submatrix ($D^{(i,j)}$) row or column and the corresponding subtraction (or addition) of this constant from either another row or column of the submatrix or from the submatrix linear cost element c_{ij}.

Class 2: Addition or subtraction of a constant to all allowed elements of any row or column in matrix D.

Class 1 operations simply transform the QAP into an equivalent QAP. Class 2 operations change all solutions by the same amount, so the order of these solutions remains the same.

The second key to the HGB is the use of complementary constraints:

$$y_{ijkl} = y_{klij}.$$

Since these pairs of values are guaranteed to be equal, we can shift costs between the corresponding costs d_{ijkl} and d_{klij} without fear of changing the solution of the problem.

These operations are used to obtain a dual-based bounding procedure. If the operations on D decrease the cost of all assignments by some amount v, and D remains nonnegative, then the solution to the QAP must be at least v. Therefore the goal is to apply Class 1 and 2 operations to maximize the lower bound v. Through careful

application of the above operations along with judicious shifting of the complementary costs, Hahn and Grant have been able to obtain bounds that are close to the optimal solution of (1.10). Moreover, the bound is efficiently obtained. The bound is also well-suited for a branch-and-bound implementation because sometimes the algorithm terminates with a permutation matrix X that attains the reported lower bound, in which case no further exploration of the current subproblem is required. In [43] the Hahn-Grant bound is used in a branch-and-bound algorithm to efficiently solve the Nugent 25 problem.

1.5.4 Eigenvalue Bounds

Several bounds for Koopmans-Beckmann QAP with symmetric A and B are obtained by relaxing $X \in \Pi$ to $X \in \mathcal{O}$ and working with the eigenvalues of A and B. The following result is well known [30]. For symmetric matrices A and B,

$$\min_{X \in \mathcal{O}} \operatorname{tr} AXBX^T = \langle \lambda(A), \lambda(B) \rangle_-$$

where $\lambda(A)$ denotes the vector of eigenvalues of A. The basic eigenvalue bound (EVB) for QAP is

$$\mathrm{EVB}(A, B, C) = \langle \lambda(A), \lambda(B) \rangle_- + \mathrm{LAP}(C).$$

The bound is cheaply computed, but too weak to be of computational use.

Several improvements to EVB have been proposed. One improvement is obtained by performing perturbations on A, B, and C in an effort to increase the bound. Let e be the vector of ones, and let E be a square matrix of ones. Consider the fol-

lowing perturbations of the matrices A, B, and C:

$$\begin{aligned} A' &= A + eg^T + ge^T + \mathrm{Diag}(r), \\ B' &= B + eh^T + he^T + \mathrm{Diag}(s), \\ C' &= C - 2[Aeh^T + ge^T B + gs^T + rh^T + ngh^T + (e^T g)eh^T] \\ &\quad -[as^T + rb^T + rs^T]. \end{aligned} \tag{1.11}$$

where $a = \mathrm{diag}(A), b = \mathrm{diag}(B)$, and g, h, r and s are vectors of size n. It can be verified that $\mathrm{QAP}(A, B, C) = \mathrm{QAP}(A', B', C')$. One choice for g, h, r and s is based on minimizing the spectral variance of A' and B' [30]. An even better bound can be obtained by choosing g, h, r and s so that the eigenvalue bound

$$\mathrm{EVB}(A', B', C') = \langle \lambda(A'), \lambda(B') \rangle_- + \mathrm{LAP}(C')$$

is maximized. This bound, known as the parametric eigenvalue bound [30], is one of strongest known bounds for QAP. However, it is costly to compute because the bound, considered as a function of the perturbations, is a nondifferential, nonconcave function. Finding a maximum of this function may be as difficult as solving the original QAP.

The authors of [41] took a different approach to improving the basic eigenvalue bound. The goal is to try to enforce $X \in \mathcal{E}$ as well as $X \in \mathcal{O}$. This is accomplished by projecting the problem into a lower dimensional space. There is a one-to-one correspondence between orthogonal matrices in this lower dimensional space, and matrices that are in $\mathcal{O} \cap \mathcal{E}$ in the original space, as stated by the following theorem:

Theorem 1.5.1 [41, Lemma 3.1] *Let X be an $n \times n$ matrix with $X \in \mathcal{O} \cap \mathcal{E}$. Then*

there is an $(n-1)\times(n-1)$ *orthogonal matrix* $\hat{X}$ *such that* $X = V\hat{X}V^T + (1/n)E$, *where* V *is an* $n\times(n-1)$ *matrix whose columns are an orthonormal basis for the nullspace of* e^T. *Conversely, if* $\hat{X}$ *is an* $(n-1)\times(n-1)$ *orthogonal matrix, then* $X = V\hat{X}V^T + (1/n)E \in \mathcal{O}\cap\mathcal{E}$.

It follows from Theorem 1.5.1 that if $X \in \mathcal{O}\cap\mathcal{E}$, then

$$\begin{aligned} AXBX^T &= AV\hat{X}V^TBV\hat{X}^TV^T + \frac{1}{n}(AEBV\hat{X}^TV + AV\hat{X}V^TBE) + \frac{1}{n^2}AEBE \\ &= AV\hat{X}V^TBV\hat{X}^TV^T + \frac{1}{n}(AEBX^T + AXBE) - \frac{1}{n^2}AEBE, \end{aligned}$$

and therefore

$$\operatorname{tr} AXBX^T = \operatorname{tr}\hat{A}\hat{X}\hat{B}\hat{X}^T + \frac{2}{n}\operatorname{tr} Aee^TBX^T - \frac{1}{n^2}s(A)s(B), \tag{1.12}$$

where $\hat{A} = V^TAV$, $\hat{B} = V^TBV$. Using (1.12), we can then write

$$\operatorname{tr} AXBX^T + C\bullet X = \operatorname{tr}\hat{A}\hat{X}\hat{B}\hat{X}^T + D\bullet X - \frac{1}{n^2}s(A)s(B), \tag{1.13}$$

where $D = C + (2/n)r(A)r(B)^T$. Since

$$\min_{\hat{X}\in\mathcal{O}} \quad \operatorname{tr}\hat{A}\hat{X}\hat{B}\hat{X}^T = \langle\lambda(\hat{A}),\lambda(\hat{B})\rangle_-, \tag{1.14}$$

it follows from (1.13) that the **projected eigenvalue bound** [41]

$$\mathrm{PB}(A,B,C) = \langle\lambda(\hat{A}),\lambda(\hat{B})\rangle_- + \mathrm{LAP}(D) - \frac{1}{n^2}s(A)s(B), \tag{1.15}$$

satisfies $\mathrm{PB}(A,B,C) \le \mathrm{QAP}(A,B,C)$. The bounds produced by the projected eigenvalue bound are close to those produced by the parametric eigenvalue bound, yet PB is quite efficiently computed.

1.5.5 Semidefinite Programming Bounds

Semidefinite programming relaxations for QAP resulting in lower bounds were first presented in [87], and later in [61]. To obtain the first, simplest SDP bound, we make use of the trace formulation of QAP. To keep things less cluttered we let $x = \mathrm{vec}(X)$, i.e. we stack the columns of X on top of one another to obtain x. Then we can rewrite the objective function:

$$\begin{aligned} f(X) &= \operatorname{tr} AXBX^T + CX^T \\ &= x^T(B \otimes A)x + c^T x \\ &= \operatorname{tr} xx^T(B \otimes A) + c^T x \\ &= \operatorname{tr} L_Q Y_X, \end{aligned}$$

where $c = \mathrm{vec}(C)$ and we define the $(n^2+1) \times (n^2+1)$ matrices

$$L_Q = \begin{bmatrix} 0 & c^T/2 \\ c/2 & B \otimes A \end{bmatrix}, Y_X = \begin{bmatrix} 1 & x^T \\ x & xx^T \end{bmatrix}. \tag{1.16}$$

If we do not require Y_X to have exactly the form (1.16), then we have a relaxation of QAP, and $\min L_Q \bullet Y_X$ is a lower bound on QAP. Notice that Y_X is a positive semidefinite, rank one matrix. In the SDP relaxation it is too hard to enforce the rank one constraint, so it is discarded.

Since $Y = Y_X$ is supposed to result from a permutation matrix X, it must be a 0-1 matrix, so $X_{ij}^2 = X_{ij}$. But this means that the diagonal of Y is equal to its first column (or row), which we write as $Y_0 = \mathrm{diag}(Y)$.

Next consider the orthogonality constraint $XX^T = I$. Letting X_i denote the ith column of X, $XX^T = \sum_{i=1}^{n} X_i X_i^T$. Since $X_i X_i^T = Y^{(i,i)}$, the i^{th} diagonal block of Y (ignoring the first row and column), if one defines as in [87] the linear operator

$$\text{bdiag}(Y) = \sum_{n=1}^{n} Y^{(i,i)} \tag{1.17}$$

then $\text{bdiag}(Y) = I$ is a necessary condition for Y to be of the form (1.16).

The authors of [87] consider the redundant orthogonality constraint $X^T X = I$ to get a stronger relaxation. Constraints that are redundant in the original problem can be used to strengthen a relaxation. Note that $(X^T X)_{ij} = X_i^T X_j = \text{tr}\, X_i X_j^T$, which is simply the trace of the (i, j) block of Y. The linear operator odiag is defined as follows:

$$(\text{odiag}(Y))_{ij} = \text{tr}\, Y^{(i,j)}, \tag{1.18}$$

and $\text{odiag}(Y) = I$ is another valid constraint on Y.

Lastly, consider the assignment constraints – the sums of each row and column of X must equal one. The condition that all row sums are equal to one implies

$$\begin{aligned} (E \otimes I)x &= e \\ (E \otimes I)xx^T &= ex^T \\ (E \otimes I) \bullet (xx^T) &= n. \end{aligned}$$

Similarly, the constraint that column sums are equal to one is equivalent to $(I \otimes E) \bullet$

$(xx^T) = n$. Therefore, if Y is of the form (1.16) it must be that $D \bullet Y = 0$, where

$$D = \begin{bmatrix} n & -e^T \otimes e^T \\ -e \otimes e & I \otimes E \end{bmatrix} + \begin{bmatrix} n & -e^T \otimes e^T \\ -e \otimes e & E \otimes I \end{bmatrix}.$$

Putting all of these constraints together, we obtain a first semidefinite programming relaxation of QAP:

$$\begin{aligned} \min \quad & \operatorname{tr} L_Q Y \\ \text{s.t.} \quad & \operatorname{bdiag}(Y) = I, \quad \operatorname{odiag}(Y) = I \\ & Y_0 = \operatorname{diag}(Y), \quad D \bullet Y = 0, \\ & Y \succeq 0. \end{aligned} \tag{1.19}$$

Theorem 1.5.2 [87, Proposition 2.1] *Suppose that Y is feasible for the SDP relaxation above. Then Y is singular.*

As noted in [87], Theorem 1.5.2 means that the feasible set of SDP (1.19) has no interior, and hence cannot be solved in a numerically stable way. Zhao et al. express the feasible set of (1.19) in a lower dimensional space to overcome this difficulty, but unfortunately the bounds obtained by this relaxation are not very tight. The relaxation can be strengthened by adding more constraints; for example the diagonals of the off-diagonal blocks of Y must be zero. The addition of constraints such as these results in bounds which are among the tightest known, but are very expensive computationally. The computation of one SDP bound for the Nugent 20 problem required over six hours on a DEC Alpha [87].

Another semidefinite programming relaxation for QAP was given by Lin and Saigal in [61]. Their formulation ignores the bdiag and odiag constraints (1.17) and

(1.18), and uses a different representation for the row and column sum constraints. As a result, the SDP relaxation of [61] does not suffer from the numerical difficulties of the simplest SDP relaxation in [87], though it is not as strong. An interior-point method for the solution of the proposed SDP is introduced, and a preconditioned conjugate gradient method is used to solve the linear system generated at each iteration of the interior-point algorithm. The bounds obtained are the best known for some problems, but remain computationally very expensive.

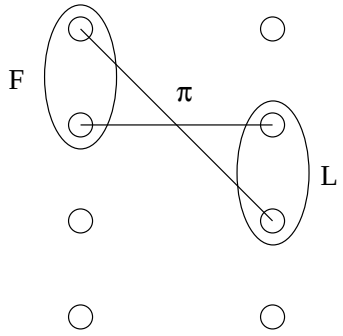
Anstreicher and Wolkowicz [5] proved the interesting result that $\mathrm{EVB}(A, B) = \langle \lambda(A), \lambda(B) \rangle_-$ has a semidefinite programming representation. The SDP characterization of EVB provides the basis for the derivation of a convex quadratic programming bound described in Chapter 2.

1.5.6 Dynamic Programming Bound

The dynamic programming bound (DPB) of Marzetta and Brüngger [65] has been used in a branch-and-bound implementation that was the first to solve the Nugent 25 test problem. Dynamic programming is a general technique used to solve many different types of sequential decision problems. Typically a table of the results of a number of subproblems is created, and these results are used to construct the solution of the problem at hand. A more in-depth description of dynamic programming can be found in any algorithms text, e.g. Cormen [26].

To motivate the bound, consider what happens in the course of a branch-and-bound algorithm (see Section 1.4.2 for more details). At a given node in the

Figure 1.9: A QAP subproblem



tree, a subset of facilities $F \subseteq \{1 \ldots n\}$ have been assigned to a subset of locations $L \subseteq \{1 \ldots n\}$ via the partial permutation $\pi : F \to L$. If $m = |F| = |L|$ then the subproblem is a QAP of size $n - m$ with

$$\begin{aligned} A'_{ij} &= A_{ij} && i, j \notin F \\ B'_{kl} &= B_{kl} && k, l \notin L \\ C'_{ik} &= C_{ik} + \sum_{j \in F} (A_{ij} B_{k\pi(j)} + A_{ji} B_{\pi(j)k}) && i \notin F, k \notin L \end{aligned} \tag{1.20}$$

Notice that A' and B' depend only on F and L, but not on π, and therefore many problems in the tree may have the same A' and B' (see Figure 1.9). So we define

$$\mathrm{dpb}(A', B', C') = \mathrm{QAP}(A', B', 0) + \mathrm{LAP}(C').$$

Solving $\mathrm{QAP}(A, B, 0)$ is just as hard as solving the original problem. However, if the $|F|$ and $|L|$ are large, A' and B' are small and we can quickly solve $\mathrm{QAP}(A', B', 0)$. With this in mind, a dynamic programming algorithm for QAP can be formulated. Starting with $|F| = |L| = n - 1$, the solution values to all possible

QAP$(A', B', 0)$ are stored in a lookup table. The algorithm proceeds to level k once values for all subproblems of level greater than k are computed. A branch-and-bound algorithm is executed to compute dpb for each level k subproblem. Bound computation requires only the solution of an LAP, and a table lookup. Once k is small, it may become prohibitive to find exact solutions to all subproblems. In this case, the branch-and-bound algorithm may be stopped prematurely at any point, and the smallest bound in the list of pending nodes Q may be used as a lower bound for the subproblem. In this way, time may be traded for accuracy. If the distance matrix of the problem has symmetries, they can be exploited to reduce the size of the lookup table required.

1.5.7 Comparison of Lower Bounds

Table 1.2 compares representatives of several types of lower bounds for QAP. GLB is the standard Gilmore-Lawler bound [35, 57], KCCEB [52] is based on the linear relaxation presented in Section 1.5.2, and is very similar to the Hahn-Grant bound, PB is the projected eigenvalue bound of [41], EVB3 is the considerably more expensive parametric eigenvalue bound [30], and SDPB1 is the computationally least expensive SDP bound presented in [87]. BKV denotes the best known value for each QAP, which has been proved optimal for all problems, except tai25a and tai30a. Though the relative tightness of the bounds varies over the type of test problem, in most cases the tightest bound is either KCCEB or SDPB1. However, the bounds KCCEB, EVB3, and SDP1 are considerably more expensive to compute than GLB and

Table 1.2: Comparison of different lower bounds on QAPLIB test set

Problem	BKV	GLB	KCCEB	PB	EVB3	SDPB1
esc16a	68	38	41	47	50	47
esc16b	292	220	274	250	276	250
esc16c	160	83	91	95	113	95
esc16d	16	3	4	-19	-12	-19
had16	3720	3358	3553	3560	3601	3612
had18	5358	4776	5078	5104	5176	5174
had20	6922	6166	6567	6625	6702	6713
kra30a	88900	68360	75566	63717	n.a.	69736
kra30b	91420	69065	76235	63818	n.a.	70324
nug12	578	493	521	472	498	486
nug15	1150	963	1033	973	1001	1009
nug20	2570	2057	2173	2196	2290	2281
nug25	3744	2869	3064	3190	3287	3305
nug30	6124	4539	4785	5266	5448	5413
rou12	235528	202272	223543	200024	201337	208685
rou15	354210	298548	323589	296705	297958	306833
rou20	725520	599948	641425	597045	n.a.	615549
scr12	31410	27858	29538	4727	n.a.	11117
scr15	51140	44737	48547	10355	n.a.	17046
scr20	110030	86766	94489	16113	n.a.	28535
tai20a	703482	580674	616644	575831	n.a.	591994
tai25a	1167256	962417	1005978	956657	n.a.	974004
tai30a	1818146	1504688	1565313	1500407	n.a.	1529135

PB. It is known that GLB is too weak to be used to solve the most difficult problems [24], but bounds like EVB3 and SDP1 probably take too long to compute to be used in a branch-and-bound implementation. Not surprisingly, an active area of research is to derive lower bounds which are stronger than GLB but not too demanding in terms of computational effort.

1.6 Other Solution Methods

1.6.1 Cutting Plane Methods

Cutting plane methods are another technique to exactly solve combinatorial optimization problems. A high level description of a cutting plane method for the solution of an integer program IP is the following:

1. Formulate a linear program LP from the integer program, typically by relaxing the integrality constraints and perhaps ignoring other constraints.

2. Solve LP to obtain solution x.

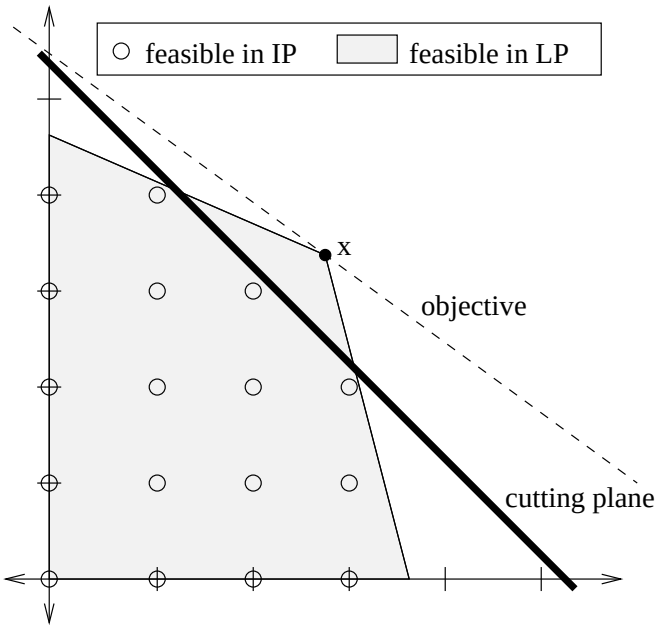
3. If x satisfies the constraints of the original problem (including integrality), stop.

4. Otherwise, call a separation procedure to find inequalities violated by x, but satisfied by all integer solutions to IP.

5. Add these inequalities to LP and go back to step 1.

The inequalities that separate, or cut off x from the rest of the search space are called **cutting planes**, hence the name "cutting plane method". A cutting plane

Figure 1.10: Example of a cutting plane



for a simple example is shown in Figure 1.10. Recall that the constraints of a linear program describe a polytope in some high dimensional space. The effectiveness of a cutting plane method depends in part on how closely the initial LP polytope approximates the solution space of the original problem. It also depends on subsequent cuts made by the method, which are determined by the separation procedure. Also note that terminating the procedure before an optimal solution is found can still provide useful information; in particular each linear program provides successively better lower bounds on the solution to the problem. Cutting plane procedures differ from branch-and-bound algorithms in that branching is avoided as much as possible by exerting more effort at each node. In particular, LP typically has a very large

number of variables, making branching costly.

To obtain an effective cutting plane algorithm, careful study of the problem's polyhedral structure is required. For QAP, there are many different integer programming formulations – see [51] for a more detailed presentation. We have already presented some of these linearizations in the context of lower bound computation, see (1.6) and (1.9). Both linearizations have $O(n^4)$ variables y_{ijkl}. Using the fact that the y_{ijkl} are supposed to be binary allows for various cuts to be made.

Kaibel [50] used a cutting plane procedure to solve three of the easier instances of size 32 from QAPLIB to optimality. However more difficult problems of size $n \geq 25$ cannot be solved using these methods at present.

1.7 Software for QAP

Many people have written software packages implementing exact solution methods or heuristics for QAP. Some of these packages are publicly available, but some of the codes obtaining the best results in practice are not. The best source for information about software codes for QAP is the QAPLIB [19] homepage on the internet. This page contains a set of commonly used test problems and solutions, links to surveys and dissertations concerning QAP, links to software, and a listing of QAP researchers. As a result of this effort, different techniques for solving QAP can be compared in a manner that is both fair and simple.

The most commonly examined set of QAP test problems are the so-called Nugent problems, found in [71]. The problems in this set range from size $n = 5$ to

$n = 30$. The article [42] gives an in-depth description of the history of the problems and their solution.

Several codes implementing heuristic methods of QAP are available. An implementation for the GRASP heuristic [77] is available from the QAPLIB home page. Implementations of the fast ant system, tabu search and simulated annealing procedures for QAP are available from É. Taillard's homepage:

`http://sunst50.einev.ch/prive/pro/taillard`

QAPPack is a Java package implementing a branch-and-bound algorithm for QAP. The package is based on the linearization of QAP proposed in [52]. Recall that a number of other bounds fall into this framework, among them the Hahn-Grant bound and Gilmore-Lawler bound. Since implementations of all these bounds are provided, QAPPack provides a framework for comparison between the different methods.

An early implementation of a branch-and-bound algorithm for QAP by Burkard and Derigs [16] is available on the homepage of H. Wolkowicz. In 1980, the Burkard-Derigs implementation first solved the Nugent 15 problem.

The algorithm of Hahn, Grant and Hall [46] is one of the best general purpose branch-and-bound QAP codes – it was the first code to solve several QAPLIB problems and solved many of the larger instances substantially faster than any other implementation at the time. It combines the Hahn-Grant bound [45] with several unique strategies for constructing the branch-and-bound tree, described in [43]. Work continues on a parallel implementation of the algorithm.

1.7.1 Parallel Algorithms for QAP

Branch-and-bound algorithms typically examine millions of nodes in the course of solving a large problem. Therefore a coarse-grained algorithm where the nodes are divided among processors is usually used to solve large problems. Such an algorithm is not hard to implement if the amount of information required to compute each bound and make branching decisions is small. In a shared memory system, each processor can simply look up the required information. In a message passing environment, typically one processor (the master) is responsible for distributing work among the others (the workers).

In 1992, Mautor and Roucairol [67] implemented a branch and bound algorithm on a shared memory machine, and were able to solve some previously unsolved QAPLIB instances. The implementation used the Gilmore-Lawler bound, and took advantage of symmetry in the problems where it existed.

The Nugent 17 problem was first solved by Laursen [56] in 1993, using a 16 processor system and the Gilmore-Lawler bound. Each processor ran a sequential branch-and-bound algorithm. The best processor utilization (98%) was obtained by dynamically distributing subproblems in a synchronous fashion.

In 1994, Clausen and Perregaard [24] used the Gilmore-Lawler bound in a parallel branch-and-bound algorithm to solve the Nugent 20 problem. The algorithm was run on 16 Intel processors connected in a ring topology. Each processor kept a pool of unsolved subproblems, and periodically compared the size of its pool to

those of its neighbors to balance the load among processors. In addition to exploiting symmetry, a simulated annealing algorithm was used to generate an initial incumbent solution. The bounding scheme of Carraresi and Malucelli [21] was also employed, but the improvement in the quality of the bound was not enough to offset the additional cost required.

Later, Clausen and Perregaard joined with Brüngger and Marzetta [13] to implement a parallel branch-and-bound algorithm using the Gilmore-Lawler bound. The code made use of ZRAM ([14], [15]), a portable library of parallel search algorithms and data structures. Their code was the first to solve ten different QAPLIB instances, including the Nugent 21 and Nugent 22 problems.

The dynamic programming bound [65] was used in a parallel implementation (also using ZRAM) that first solved the Nugent 25 test problem. Since each processor needs to access the lookup table required by the algorithm, the communication requirements of this parallel algorithm are quite high.

The authors of [13, 65] also made use of a tree-size estimation procedure first proposed by Knuth in [53]. The procedure calls for a sample of random paths in the tree to estimate its size, and is useful for forecasting the resources required by an algorithm to solve a large problem. Chapter 4 considers the application of the Knuth procedure to estimating computations on large QAPs.

1.8 Summary

We have concluded our description of solution methods for QAP, the problem of assigning facilities to locations so as to minimize transportation cost. Theory tells us that even finding a solution guaranteed to be close to the solution is difficult. In practice, QAP has proved to be much more difficult than related problems such as the traveling salesperson problem.

For most instances, any of a number of heuristics can be used to get a good suboptimal solution, however these heuristics provide no guarantees. Branch and bound algorithms have been used to find optimal solutions to QAPs of size less than 30. The key ingredient in such algorithms is the lower bounding procedure. The Gilmore-Lawler bound is the oldest known, and has been used the most in practice. To solve more difficult problems, tighter, more computationally expensive bounds are required. The use of information gathered during bound computation can be used to make branching decisions, and it is likely that successful implementations will make intelligent use of this information.

Lastly, most of the best results obtaining exact solutions have been obtained with the use of parallel computing resources in one form or another, and to attack problems of the scale now on the threshold of feasibility it is virtually certain that these resources will need to be utilized. Fortunately, branch and bound lends itself well to the implementation of efficient parallel algorithms with high CPU utilization.

CHAPTER 2
A CONVEX QUADRATIC PROGRAMMING BOUND FOR QAP

2.1 The Quadratic Programming Bound

In this chapter we describe a new lower bounding technique for the quadratic assignment problem. Efficiently computed, tight lower bounds are the key to an effective branch-and-bound algorithm. The quadratic programming bound, or QPB, is related to the projected eigenvalue bound, and relies on a semidefinite programming (SDP) representation of the basic eigenvalue bound of [30], from [5]. This chapter begins with a presentation of QPB and some empirical results that indicate that QPB may be a good candidate for use in a branch-and-bound implementation.

Difficult combinatorial optimization problems are often solved using procedures that provide bounds on their solution values. Most bounding procedures first form a relaxed version of the given problem that is easier to solve – typically a relaxed problem is obtained by considering a larger search space. Consider a generic optimization problem where the objective is to find a point in region S satisfying some given criteria. A relaxation instead searches a region R, which includes S. If the solution to the search in R is also in S, the original problem is solved. For each type of optimization problem there are many possible relaxations. QAP is a search problem over the set of permutations Π, which can be characterized as the intersec-

tion of orthogonal matrices $\mathcal{O}$, nonnegative matrices $\mathcal{N}$, and the set $\mathcal{E}$ of matrices with row and column sums equal to one. Our new bound QPB searches over the set of matrices $\mathcal{E} \cap \mathcal{N}$, whereas other bounds search over different regions.

The derivation of the new bound is complicated. It is difficult to formulate relaxations of the set of permutations that produce bounding procedures that are computationally efficient, yet provide tight lower bounds and insight as to the solution of the given QAP. Useful bounding procedures are obtained by studying carefully the structure of both the objective and constraints of QAP.

In the case of QPB, a given QAP is relaxed to form a convex quadratic program. Forming the quadratic program requires the solution of a particular semidefinite programming problem – in Section 2.3 we describe an efficient algorithm for its solution. We present two methods for solving the quadratic programs that arise. The first is an interior-point algorithm that is able to find high-accuracy solutions. However, since we are concerned with quickly finding approximate solutions that are not necessarily optimal, in Section 2.5 we present a steepest-descent based approach that converges more slowly but is computationally less expensive. Finally, we provide some examples and experiments that show that the QPB is a good candidate for use in a branch-and-bound algorithm.

In our discussion we make extensive use of results concerning the Kronecker product $\otimes$. See Appendix A for notation, definitions and basic results.

2.2 Derivation of QPB

The basic eigenvalue bound (EVB) was described in the previous chapter and is computed by relaxing the constraint that X is a permutation matrix to an orthogonality constraint:

$$EVB(A, B, C) := \min_{X \in \mathcal{O}} \operatorname{tr} AXBX^T + \mathrm{LAP}(C).$$

It is proved in [30] that

$$\min_{X \in \mathcal{O}} \operatorname{tr} AXBX^T = \langle \lambda(A), \lambda(B) \rangle_- \tag{2.1}$$

where $\lambda(A)$ denotes the vector of eigenvalues of A. Although the basic eigenvalue bound can be computed efficiently, it is not very strong for most problems.

The projected eigenvalue bound of [41] improves upon the basic eigenvalue bound by enforcing $X \in \mathcal{E}$ as well as $X \in \mathcal{O}$. This is accomplished by projecting the problem into a lower dimensional space. There is a one-to-one correspondence between orthogonal matrices in this lower dimensional space, and matrices that are in $\mathcal{O} \cap \mathcal{E}$ in the original space, as stated by the following theorem:

Theorem 2.2.1 [41, Lemma 3.1] *Let X be an $n \times n$ matrix with $X \in \mathcal{O} \cap \mathcal{E}$. Then there is an $(n-1) \times (n-1)$ orthogonal matrix $\hat{X}$ such that $X = V\hat{X}V^T + (1/n)E$, where V is an $n \times (n-1)$ matrix whose columns are an orthonormal basis for the nullspace of e^T. Conversely, if $\hat{X}$ is an $(n-1) \times (n-1)$ orthogonal matrix, then $X = V\hat{X}V^T + (1/n)E \in \mathcal{O} \cap \mathcal{E}$.*

It follows from Theorem 2.2.1 that if $X \in \mathcal{O} \cap \mathcal{E}$, then

$$\begin{aligned} AXBX^T &= AV\hat{X}V^TBV\hat{X}^TV^T + \frac{1}{n}(AEBV\hat{X}^TV + AV\hat{X}V^TBE) + \frac{1}{n^2}AEBE \\ &= AV\hat{X}V^TBV\hat{X}V^T + \frac{1}{n}(AEBX^T + AXBE) - \frac{1}{n^2}AEBE, \end{aligned}$$

allowing us to write

$$\operatorname{tr} AXBX^T + C \bullet X = \operatorname{tr} \hat{A}\hat{X}\hat{B}\hat{X}^T + D \bullet X - \frac{1}{n^2}\operatorname{tr} AEBE, \tag{2.2}$$

where $D = C + (2/n)Aee^TB$, $\hat{A} = V^TAV, \hat{B} = V^TBV$. The term $\operatorname{tr} AEBE$ is equal to $s(A)s(B)$, the product of the sums of the elements of A and B. Since minimizing $D \bullet X$ over $X \in \mathcal{E} \cap \mathcal{N}$ is simply a linear assignment problem, and since

$$\min_{\hat{X} \in \mathcal{O}} \quad \operatorname{tr} \hat{A}\hat{X}\hat{B}\hat{X}^T = \langle \lambda(\hat{A}), \lambda(\hat{B}) \rangle_-, \tag{2.3}$$

it follows from (2.2) that the **projected eigenvalue bound** [41]

$$\mathrm{PB}(A, B, C) := \langle \lambda(\hat{A}), \lambda(\hat{B}) \rangle_- + \mathrm{LAP}(D) - \frac{1}{n^2}s(A)s(B), \tag{2.4}$$

satisfies $\mathrm{PB}(A, B, C) \leq \mathrm{QAP}(A, B, C)$. PB is significantly stronger than EVB as shown in Table 1.2 of the previous chapter. For many problems, PB is a competitive bound, especially in consideration of the modest computational effort needed to obtain it. However, PB does not produce $X \in \mathcal{E} \cap \mathcal{N}$ attaining the bound, which would be useful for making branching decisions in a branch and bound algorithm. In a branch-and-bound algorithm, certain variables x_{ij} are fixed to one, resulting in smaller QAP instances. Unfortunately, in many cases PB increases slowly as variables are fixed,

and in some cases PB even decreases. For these reasons, PB is not an ideal candidate for use in a branch-and-bound algorithm.

Our new convex quadratic programming bound for QAP is derived by considering a semidefinite programming representation of the basic eigenvalue bound. To represent EVB as an SDP, it is necessary to represent the orthogonality of X in terms of linear constraints. The basic SDP bound (1.19) of the previous chapter uses the bdiag$(\cdot)$ and odiag$(\cdot)$ operators defined in (1.17) and (1.18). It can be shown that bdiag$(\cdot)$ and odiag$(\cdot)$ are the adjoints of the operators $S \to I \otimes S$ and $T \to T \otimes I$, from $\Re^{n\times n}$ to $\Re^{n^2\times n^2}$, respectively. Consider the following pair of SDPs:

$$\begin{aligned} \mathrm{SDP}(A,B): \quad \min \quad & (B\otimes A)\bullet Y \\ \text{s.t.} \quad & \mathrm{bdiag}(Y) = I \\ & \mathrm{odiag}(Y) = I \\ & Y \succeq 0, \end{aligned}$$

$$\begin{aligned} \mathrm{SDD}(A,B): \quad \max \quad & \operatorname{tr} S + \operatorname{tr} T \\ \text{s.t.} \quad & (I\otimes S) + (T\otimes I) \preceq (B\otimes A) \\ & S = S^T, T = T^T. \end{aligned} \tag{2.5}$$

The constraints $\mathrm{bdiag}(Y) = I, \mathrm{odiag}(Y) = I$ are satisfied for $Y = \mathrm{vec}(X)\,\mathrm{vec}(X)^T$ with orthogonal X. In light of the discussion from the previous chapter, it should be seen that SDP(A,B) is a semidefinite programming relaxation of QAP(A,B).

$\mathrm{SDD}(A, B)$ is derived as the Lagrangian dual of the minimization problem from (2.1) in [5].

Theorem 2.2.2 *If Y is feasible in $\mathrm{SDP}(A, B)$, and (S, T) are feasible in $\mathrm{SDD}(A, B)$, then $(B \otimes A) \bullet Y = \operatorname{tr} S + \operatorname{tr} T + Y \bullet [(B \otimes A) - (I \otimes S) - (T \otimes I)] \geq \operatorname{tr} S + \operatorname{tr} T$. Moreover $\mathrm{SDP}(A, B) = \mathrm{SDD}(A, B) = \langle \lambda(A), \lambda(B) \rangle_-$.*

Proof: The first result is simply the weak duality relationship between $\mathrm{SDP}(A, B)$ and $\mathrm{SDD}(A, B)$. $\mathrm{SDP}(A, B) = \mathrm{SDD}(A, B)$ follows from the fact that these are dual semidefinite programming problems, both of which have interior solutions; see for example [69, Theorem 4.2.1]. Finally, $\mathrm{SDD}(A, B) = \langle \lambda(A), \lambda(B) \rangle_-$ is proved in [5, Theorem 3.2]. $\square$

From (2.4) and Theorem 2.2.2 it is natural to consider $\mathrm{SDD}(\hat{A}, \hat{B})$. Let us assume that we are able to find $\hat{S}, \hat{T}$ that solve $\mathrm{SDD}(\hat{A}, \hat{B})$. Then we can use Theorem 2.2.2 to rewrite $\operatorname{tr} \hat{A}\hat{X}\hat{B}\hat{X}^T$ as:

$$\begin{aligned} \operatorname{tr} \hat{A}\hat{X}\hat{B}\hat{X}^T &= \operatorname{vec}(\hat{X})^T (\hat{B} \otimes \hat{A}) \operatorname{vec}(\hat{X}) \\ &= \hat{Y} \bullet (\hat{B} \otimes \hat{A}) \\ &= \langle \lambda(\hat{A}), \lambda(\hat{B}) \rangle_- + \operatorname{vec}(\hat{X})^T \hat{Q} \operatorname{vec}(\hat{X}), \end{aligned} \tag{2.6}$$

where

$$\hat{Q} = (\hat{B} \otimes \hat{A}) - (I \otimes \hat{S}) - (\hat{T} \otimes I) \succeq 0.$$

Substituting (2.6) into (2.2) then results in

$$\operatorname{tr} AXBX^T + C \bullet X = \langle \lambda(\hat{A}), \lambda(\hat{B}) \rangle_- + D \bullet X + \operatorname{vec}(\hat{X})^T \hat{Q} \operatorname{vec}(\hat{X}) - \frac{1}{n^2} s(A) s(B). \tag{2.7}$$

Comparing (2.4) with (2.7), it is clear that the projected eigenvalue bound $\mathrm{PB}(A, B, C)$ corresponds to *ignoring* the term $\operatorname{vec}(\hat{X})^T \hat{Q} \operatorname{vec}(\hat{X})$ in (2.7), and then minimizing the remaining term $D \bullet X$ over $X \in \mathcal{E} \cap \mathcal{N}$. However, since $\hat{Q} \succeq 0$, a better bound can be obtained by solving the convex quadratic program that includes this term. To get the simplest possible formulation for this program it is convenient to express the quadratic term using X rather than $\hat{X}$. Let $S' = V \hat{S} V^T$, and $T' = V \hat{T} V^T$. From the definition of $\hat{A}$ and $\hat{B}$ we then have

$$\begin{aligned} \operatorname{vec}(\hat{X})^T \hat{Q} \operatorname{vec}(\hat{X}) &= \operatorname{vec}(\hat{X})^T (V^T \otimes V^T)[(B \otimes A) - (I \otimes S') - (T' \otimes I)](V \otimes V) \operatorname{vec}(\hat{X}) \\ &= (\operatorname{vec}(X) - \frac{1}{n} e)^T [(B \otimes A) - (I \otimes S') - (T' \otimes I)](\operatorname{vec}(X) - \frac{1}{n} e) \\ &= \operatorname{vec}(X)^T Q \operatorname{vec}(X) - \frac{2}{n} (e^T B \otimes e^T A) \operatorname{vec}(X) + \frac{1}{n^2} s(A) s(B), \end{aligned} \tag{2.8}$$

where $Q = (B \otimes A) - (I \otimes S') - (T' \otimes I)$, and we are using the fact that

$$\begin{aligned} (I \otimes S')e &= (I \otimes V \hat{S} V^T)(e \otimes e) = 0, \\ (T' \otimes I)e &= (V \hat{T} V^T \otimes I)(e \otimes e) = 0. \end{aligned}$$

Substituting (2.8) into (2.7), and noting that $(e^T B \otimes e^T A) \operatorname{vec}(X) = e^T A X B e = \operatorname{tr} A e e^T B X^T$, we obtain

$$\operatorname{tr} AXBX^T + C \bullet X = \langle \lambda(\hat{A}), \lambda(\hat{B}) \rangle_- + \operatorname{vec}(X)^T Q \operatorname{vec}(X) + C \bullet X. \tag{2.9}$$

Figure 2.1: High-level algorithm for computing QPB

$z = \text{QPB}(A, B, C)$

$(\lambda(\hat{A}), \lambda(\hat{B})) = \mathsf{eigenvalues}(\hat{A}, \hat{B})$

$(S', T') = \mathsf{dual_solution}(A, B, C)$

$z_{QP} = \mathsf{solve_QP}(A, B, C, S', T')$

$z = \langle \lambda(\hat{A}), \lambda(\hat{B}) \rangle_- + z_{QP}$

Motivated by (2.9), we define the quadratic programming problem

$$\begin{aligned} \text{QP}(A, B, C) = \quad & \min \quad \text{vec}(X)^T Q \, \text{vec}(X) + C \bullet X \\ & \text{s.t.} \quad Xe = X^T e = e \\ & \qquad X \geq 0, \end{aligned} \tag{2.10}$$

and the quadratic programming bound

$$\text{QPB}(A, B, C) = \langle \lambda(\hat{A}), \lambda(\hat{B}) \rangle_- \; + \text{QP}(A, B, C). \tag{2.11}$$

A high-level procedure for computing QPB is given in Figure 2.1. We will describe how to solve $\text{QP}(A, B, C)$ and find the dual solution that provides S', T' in subsequent sections.

By construction $\text{PB}(A, B, C) \leq \text{QPB}(A, B, C) \leq \text{QAP}(A, B, C)$. In the next lemma we show that $\text{QPB}(A, B, C)$ strictly dominates $\text{PB}(A, B, C)$ under a mild assumption.

Lemma 2.2.3 *Assume that* $\text{PB}(A, B, C) < \text{QAP}(A, B, C)$*, and the solution of* $\text{LAP}(D)$ *is unique. Then* $\text{PB}(A, B, C) < \text{QPB}(A, B, C)$.

Proof: Let X^* be the solution of LAP(D). Since $X^* \in \mathcal{O} \cap \mathcal{E}$, by Theorem 2.2.1 we can write $X^* = V\hat{X}V^T + (1/n)E$, from which it follows that $\hat{X} = V^T X^* V$. From (2.7) we must then have $\mathrm{vec}(\hat{X})^T \hat{Q}\,\mathrm{vec}(\hat{X}) > 0$, since otherwise $\mathrm{PB}(A, B, C) = \mathrm{QAP}(A, B, C)$. Note that $\mathrm{QPB}(A, B, C)$ corresponds exactly to minimizing the right-hand side of (2.7) over $X \in \mathcal{E} \cap \mathcal{N}$. If X^* is also a solution of $\mathrm{QP}(A, B, C)$, then $\mathrm{PB}(A, B, C) < \mathrm{QPB}(A, B, C)$ follows from $\mathrm{vec}(\hat{X})^T \hat{Q}\,\mathrm{vec}(\hat{X}) > 0$. If X^* is not optimal for $\mathrm{QP}(A, B, C)$, let X^{**} be any optimal solution. From (2.7) we then have

$$\mathrm{QPB}(A, B, C) \geq \langle \lambda(\hat{A}), \lambda(\hat{B}) \rangle_- + D \bullet X^{**} - \frac{1}{n^2} s(A)s(B) > \mathrm{PB}(A, B, C),$$

by the uniqueness of X^*. □

When $\mathrm{PB}(A, B, C) < \mathrm{QAP}(A, B, C)$, but the solution of LAP(D) is not unique, we cannot prove that $\mathrm{QPB}(A, B, C) > \mathrm{PB}(A, B, C)$ for the following reason. For each X^* which is an *extreme-point* solution of LAP(D) we might have $\mathrm{vec}(\hat{X})^T \hat{Q}\,\mathrm{vec}(\hat{X}) > 0$ for $\hat{X} = V^T X^* V$, but there could be a non-extreme-point solution X^* having $\mathrm{vec}(\hat{X})^T \hat{Q}\,\mathrm{vec}(\hat{X}) = 0$. The former implies that $\mathrm{QAP}(A, B, C) > \mathrm{PB}(A, B, C)$, but the latter implies that $\mathrm{QPB}(A, B, C) = \mathrm{PB}(A, B, C)$.

2.3 Finding a Dual Solution

The computation of QPB requires the solution of the dual semidefinite programming problem $\mathrm{SDD}(\hat{A}, \hat{B})$ to obtain matrices $\hat{S}, \hat{T}$. It can be shown that for any $\hat{A}, \hat{B}$, $\mathrm{SDD}(\hat{A}, \hat{B})$ is equivalent to a linear programming problem with many optimal solutions, and any one of them can be used to construct $\hat{S}, \hat{T}$. In what follows we derive an efficient solution procedure for $\mathrm{SDD}(\hat{A}, \hat{B})$ by exploiting the special struc-

ture of the linear programs that arise. We also show that we can use an approximate solution X to QP(A, B, C) to choose alternate $\hat{S}, \hat{T}$ that may improve QPB. To simplify notation, in our presentation we consider the problem SDD(A, B) instead of SDD$(\hat{A}, \hat{B})$.

2.3.1 Finding an Initial Dual Solution

The objective is to find S, T solving (2.5). Since A, B are symmetric, there exist orthogonal W, U so that $A = W\Sigma W^T, B = U\Lambda U^T$, where $\Sigma = \mathrm{Diag}(\sigma)$, $\Lambda = \mathrm{Diag}(\lambda)$, and σ and λ are the vectors of eigenvalues of A and B, respectively. Let us assume λ, σ are sorted so that $\lambda_1 \leq \lambda_2 \leq \ldots \leq \lambda_n$, $\sigma_1 \geq \sigma_2 \geq \ldots \geq \sigma_n$. As in the proof of [5, Theorem 3.2], note that for any S and T,

$$\begin{gathered}(B \otimes A) - (I \otimes S) - (T \otimes I) = \\ (U \otimes W)\left[(\Lambda \otimes \Sigma) - (I \otimes \bar{S}) - (\bar{T} \otimes I)\right](U^T \otimes W^T),\end{gathered} \tag{2.12}$$

where $\bar{S} = W^T S W$, $\bar{T} = U^T T U$. Since $U \otimes W$ is nonsingular, $\operatorname{tr} S = \operatorname{tr} \bar{S}$ and $\operatorname{tr} T = \operatorname{tr} \bar{T}$, the problem SDD$(A, B)$ is equivalent to

$$\begin{aligned}\max \quad & \operatorname{tr} \bar{S} + \operatorname{tr} \bar{T} \\ \text{s.t.} \quad & (\Lambda \otimes \Sigma) - (I \otimes \bar{S}) - (\bar{T} \otimes I) \succeq 0.\end{aligned} \tag{2.13}$$

However, since Λ and Σ are diagonal matrices, (2.13) is equivalent to the linear programming problem:

$$\begin{aligned}\max \quad & e^T \bar{s} + e^T \bar{t} \\ \text{s.t.} \quad & \bar{t}_i + \bar{s}_j \leq c_{ij}, \qquad i, j = 1, \ldots, n.\end{aligned} \tag{2.14}$$

where $c_{ij} = \lambda_i \sigma_j$. The problem (2.14) is the dual of the linear assignment problem:

$$\begin{aligned} \min \quad & \textstyle\sum_{i,j} c_{ij} x_{ij} \\ \text{s.t.} \quad & \textstyle\sum_{j=1}^{n} x_{ij} = 1, \quad i = 1, \ldots, n \\ & \textstyle\sum_{i=1}^{n} x_{ij} = 1, \quad j = 1, \ldots, n \\ & x_{ij} \geq 0, \quad i, j = 1, \ldots, n, \end{aligned} \tag{2.15}$$

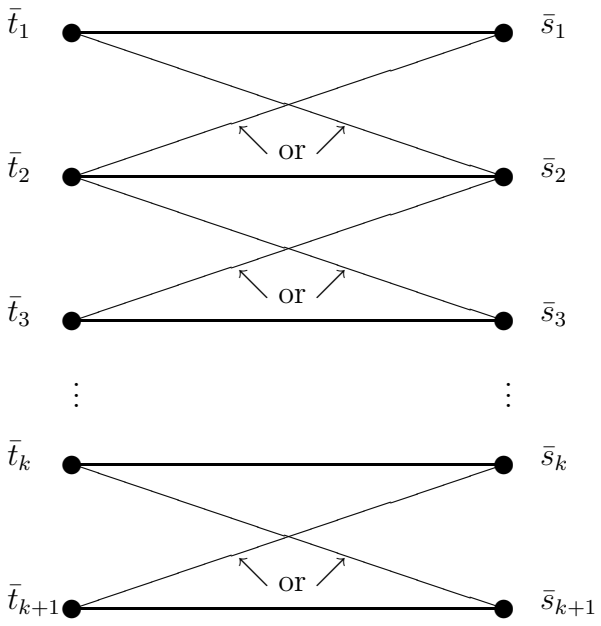
Lemma 2.3.1 *The optimal solution to* (2.15) *is* $x_{ii} = 1$, $i = 1, \ldots, n$.

Proof: From Theorem A.3.1, there exists an optimal solution to LAP corresponding to a permutation matrix. Therefore we may write (2.15) as $\min_{p \in \Pi} c_{ip(i)} = \min_{p \in \Pi} \lambda_i \sigma_{p(i)} = \langle \lambda, \sigma \rangle_-$. By Theorem A.3.2 in Appendix A, the minimizing p sorts the elements of σ in nondecreasing order. Thus $p(i) = i, i = 1, \ldots, n$ or equivalently $x_{ii} = 1, i = 1, \ldots, n$. □

It is well known that a basis for LAP corresponds to a tree in a complete bipartite graph on $2n$ nodes. Such a tree contains $2n - 1$ arcs, and the associated basis is highly degenerate. Our concern is in obtaining an *optimal* basis for (2.15); in other words a basis containing the arcs (i, i), $i = 1, \ldots n$ such that the associated dual basic solution is feasible for (2.14). In the next theorem we show that such bases can be very easily characterized.

Theorem 2.3.2 *Let* (2.15) *have* $c_{ij} = \lambda_i \sigma_j$, $\lambda_1 \leq \lambda_2 \leq \ldots \leq \lambda_n$, $\sigma_1 \geq \sigma_2 \geq \ldots \geq \sigma_n$. *Then any basis consisting of the arcs* (i, i), $i = 1, \ldots, n$, *and for each* $i = 1, \ldots n - 1$ *either the arc* $(i, i + 1)$ *or* $(i + 1, i)$, *is an optimal basis.*

Figure 2.2: Structure of optimal basis



Proof: The proof is by induction on n. If $n = 1$ there is nothing to show. Assume that such a basis is given for $n = k$, and that $\bar{t}_i$, $\bar{s}_i$, $i = 1, \ldots, k$ are associated values having $\bar{c}_{ij} = c_{ij} - \bar{t}_i - \bar{s}_j \geq 0$, $i, j = 1, \ldots, k$. (It is well known that there is one degree of freedom for the variables, so for example $\bar{t}_1$ can be set to an arbitrary value and the remaining variables solved for.) See Figure 2.2 for a graphical depiction.

Assume that the arcs $(k, k+1)$ and $(k+1, k+1)$ are added to the basis. Since from Lemma 2.3.1 (k, k) is already in the basis, we can conclude that

$$\bar{s}_k = \lambda_k \sigma_k - \bar{t}_k \tag{2.16a}$$

$$\bar{s}_{k+1} = \lambda_k \sigma_{k+1} - \bar{t}_k \tag{2.16b}$$

$$\bar{t}_{k+1} = \lambda_{k+1}\sigma_{k+1} - \bar{s}_{k+1} = \sigma_{k+1}(\lambda_{k+1} - \lambda_k) + \bar{t}_k. \tag{2.16c}$$

Then $\bar{c}_{k,k+1} = 0$, $\bar{c}_{k+1,k+1} = 0$, and

$$\begin{aligned} \bar{c}_{k+1,k} &= \lambda_{k+1}\sigma_k - (\bar{t}_{k+1} + \bar{s}_k) \\ &= (\sigma_k - \sigma_{k+1})(\lambda_{k+1} - \lambda_k) \\ &\geq 0, \end{aligned}$$

by the assumptions on the ordering of the vectors λ and σ. If $k = 1$ we are finished. Otherwise, it remains to show that for each $i, j < k$, $\bar{c}_{i,k+1} \geq 0$, and $\bar{c}_{k+1,j} \geq 0$. Let $i < k$. By definition we have

$$\begin{aligned} \bar{c}_{i,k+1} &= \lambda_i \sigma_{k+1} - (\bar{t}_i + \bar{s}_{k+1}) \\ \bar{c}_{i,k} &= \lambda_i \sigma_k - (\bar{t}_i + \bar{s}_k), \end{aligned}$$

and therefore

$$\begin{aligned}\bar{c}_{i,k+1} &= \bar{c}_{i,k} + \lambda_i(\sigma_{k+1} - \sigma_k) + (\bar{s}_k - \bar{s}_{k+1}) \\ &= \bar{c}_{i,k} + (\lambda_k - \lambda_i)(\sigma_k - \sigma_{k+1}) \\ &\geq 0,\end{aligned}$$

where the second equality uses (2.16), and the inequality follows from the inductive hypothesis and the orderings of λ and σ. Similarly, for $j < k$ we have

$$\begin{aligned}\bar{c}_{k+1,j} &= \lambda_{k+1}\sigma_j - (\bar{t}_{k+1} + \bar{s}_j) \\ \bar{c}_{k,j} &= \lambda_k\sigma_j - (\bar{t}_k + \bar{s}_j),\end{aligned}$$

so

$$\begin{aligned}\bar{c}_{k+1,j} &= \bar{c}_{k,j} + (\lambda_{k+1} - \lambda_k)\sigma_j + (\bar{t}_k - \bar{t}_{k+1}) \\ &= \bar{c}_{k,j} + (\lambda_{k+1} - \lambda_k)(\sigma_j - \sigma_{k+1}) \\ &\geq 0,\end{aligned}$$

again using (2.16), the inductive hypothesis, and the orderings of λ and σ. This completes the inductive step if $(k, k+1)$ and $(k+1, k+1)$ are added to the basis. The analysis using $(k+1, k)$ in place of $(k, k+1)$ is very similar. □

Any dual basic solution $\bar{s}, \bar{t}$ corresponding to an optimal basis for LAP provides an optimal solution for SDD(A, B) of the form $S = W\bar{S}W^T$, $T = U\bar{T}U^T$, where $\bar{S} = \text{Diag}(\bar{s})$, $\bar{T} = \text{Diag}(\bar{t})$. Note that from (2.12), the corresponding "reduced costs" $\bar{c}_{ij} = \lambda_i\sigma_j - \bar{t}_i - \bar{s}_j \geq 0$ are then exactly the eigenvalues of $(B \otimes A) - (I \otimes S) - (T \otimes I)$.

Figure 2.3: Algorithm to find an initial dual basis

$\bar{s}, \bar{t}$ = INITIAL_BASIS(σ, λ)

$\bar{s}_1 = 0, \bar{t}_1 = \lambda_1 \sigma_1$

FOR $i = 1, \ldots, n-1$

IF i IS EVEN

$\bar{t}_{i+1} = \lambda_i \sigma_{i+1} - \bar{s}_i$

$\bar{s}_{i+1} = \lambda_{i+1} \sigma_{i+1} - \bar{t}_{i+1}$

IF i IS ODD

$\bar{s}_{i+1} = \lambda_{i+1} \sigma_i - \bar{t}_i$

$\bar{t}_{i+1} = \lambda_{i+1} \sigma_{i+1} - \bar{s}_{i+1}$

Recall that for the construction of QP(A, B, C) we are actually working with SDD$(\hat{A}, \hat{B})$, where $\hat{A} = \hat{W}\hat{\Sigma}\hat{W}^T$, $\hat{B} = \hat{U}\hat{\Lambda}\hat{U}^T$, so the procedure described here is applied with $\hat{\lambda}$ and $\hat{\sigma}$ in place of λ and σ, to obtain $\bar{s} \in \Re^{n-1}$, $\bar{t} \in \Re^{n-1}$, $\hat{S} = \hat{W}\bar{S}\hat{W}^T$, $\hat{T} = \hat{U}\bar{T}\hat{U}^T$.

The algorithm given in Figure 2.3 computes an initial optimal basis for LAP by alternately choosing arcs $(k, k+1)$ and $(k+1, k)$. Since the eigenvalues λ, σ are already required to compute the first term of (2.11), initial_basis has $O(n)$ time complexity. The initial_basis algorithm is used in the dual_solution procedure (Figure 2.4). In Figure 2.4, svd denotes a function that finds eigenvalues and eigenvectors of a symmetric matrix as described at the beginning of this section. For the problems of the size of interest ($n \leq 36$), such computations are not computationally intensive.

Figure 2.4: Algorithm to compute S', T' needed by QPB

$[S', T'] = \text{DUAL_SOLUTION}(A, B)$

$\hat{A} = V^T A V, \hat{B} = V^T B V$

$[\hat{W}, \hat{\sigma}] = \mathsf{svd}(\hat{A})$

$[\hat{U}, \hat{\lambda}] = \mathsf{svd}(\hat{B})$

$[\bar{s}, \bar{t}] = \mathsf{initial_basis}(\hat{\sigma}, \hat{\lambda})$

$\hat{S} = \hat{W}\,\mathrm{Diag}(\bar{s})\hat{W}^T$

$\hat{T} = \hat{U}\,\mathrm{Diag}(\bar{t})\hat{U}^T$

$S' = V\hat{S}V^T$

$T' = V\hat{T}V^T$

2.3.2 Improving the Dual Solution

Theorem 2.3.2 of the previous section showed that there are 2^{n-2} different optimal bases for LAP, and consequently in general there are many different optimal solutions to SDD(A, B). In this section we show that the solution value of QP(A, B, C) depends on the choice of $\bar{s}, \bar{t}$. If an optimal or approximate solution to QPB(A, B, C) is known, it can be used to choose a new optimal basis for LAP, providing $\bar{s}, \bar{t}$ that lead to an improved lower bound for QAP.

Let us calculate the part of the bound that depends explicitly on the choice of $\bar{s}, \bar{t}$. From (6),

$$\begin{aligned} \hat{Q} &= (\hat{B} \otimes \hat{A}) - (I \otimes \hat{S}) - (\hat{T} \otimes I) \\ &= (\hat{B} \otimes \hat{A}) - (I \otimes \hat{W}\bar{S}\hat{W}^T) - (\hat{U}\bar{T}\hat{U}^T \otimes I). \end{aligned} \tag{2.17}$$

Let X be an approximate solution of QP, and $\hat{X} = V^T X V$. From (2.17), the part of QPB that depends on $\bar{s}, \bar{t}$ is

$$\begin{aligned}\operatorname{vec}(\hat{X})^T \hat{Q} \operatorname{vec}(\hat{X}) &= \operatorname{vec}(\hat{X})^T (\hat{B} \otimes \hat{A}) \operatorname{vec}(\hat{X}) - \operatorname{vec}(\hat{X})^T (I \otimes \hat{W}\bar{S}\hat{W}^T) \operatorname{vec}(\hat{X}) \\ &\quad - \operatorname{vec}(\hat{X})^T (\hat{U}\bar{T}\hat{U}^T \otimes I) \operatorname{vec}(\hat{X}).\end{aligned}$$

Using the fact that $(I \otimes \hat{W}\bar{S}\hat{W}^T) \operatorname{vec}(\hat{X}) = \operatorname{vec}(\hat{W}\bar{S}\hat{W}^T\hat{X})$, we calculate the part of the bound that depends on $\bar{s}$:

$$\begin{aligned}\operatorname{vec}(\hat{X})^T (I \otimes \hat{W}\bar{S}\hat{W}^T) \operatorname{vec}(\hat{X}) &= \hat{X} \bullet \hat{W}\bar{S}\hat{W}^T\hat{X} \\ &= \operatorname{tr} \hat{X}\hat{X}^T\hat{W}\bar{S}\hat{W}^T \\ &= \operatorname{tr} \hat{W}^T\hat{X}\hat{X}^T\hat{W}\bar{S} \\ &= \sum_{i=1}^{n-1} g_i \bar{s}_i,\end{aligned}$$

where $g_i = (\hat{W}^T\hat{X}\hat{X}^T\hat{W})_{ii}$. Then $g \geq 0$, and $\hat{X}\hat{X}^T = I$ implies $g = e$. Similarly, $(\hat{U}\bar{T}\hat{U}^T \otimes I) \operatorname{vec}(\hat{X}) = \operatorname{vec}(\hat{X}\hat{U}\bar{T}\hat{U}^T)$, so

$$\begin{aligned}\operatorname{vec}(\hat{X})^T (\hat{U}\bar{T}\hat{U}^T \otimes I) \operatorname{vec}(\hat{X}) &= \hat{X} \bullet \hat{X}\bar{U}\hat{T}\hat{U}^T \\ &= \operatorname{tr} \hat{X}\hat{U}\hat{T}\bar{U}^T\hat{X}^T \\ &= \operatorname{tr} \hat{U}^T\hat{X}^T\hat{X}\hat{U}\bar{T} \\ &= \sum_{i=1}^{n-1} h_i \bar{t}_i,\end{aligned}$$

where $h_i = (\hat{U}^T\hat{X}^T\hat{X}\hat{U})_{ii}$. Then $h \geq 0$, and note that $\hat{X}^T\hat{X} = I$ implies $h = e$.

Also from the orthogonality of $\hat{U}, \hat{W}$, $e^T g = \operatorname{tr}(\hat{W}^T\hat{X}\hat{X}^T\hat{W}) = \operatorname{tr}(\hat{X}\hat{X}^T) =$

$\mathrm{tr}(\hat{X}^T\hat{X}) = e^T h$. To get a better bound, this suggests finding $\bar{s}, \bar{t}$ solving

$$\min\ g^T\bar{s} + h^T\bar{t}. \tag{2.18}$$

If we keep the requirement that $\bar{s}, \bar{t}$ be dual optimal, then $\bar{s}_i + \bar{t}_i = \lambda_i\sigma_i$ and the problem becomes

$$\min\ (g-h)^T\bar{s}. \tag{2.19}$$

The minimizing $\bar{s}, \bar{t}$ can be found via one pass through the variables by monitoring the quantity $d_i = \sum_{k=1}^{i}(g_k - h_k)$. By Theorem 2.3.2, for a given i either arc $(i, i+1)$ or $(i+1, i)$ must be in the basis. If $(i, i+1)$ is in the basis,

$$\bar{s}_{i+1} = \hat{\lambda}_i\hat{\sigma}_{i+1} - t_i,$$

and if $(i+1, i)$ is in the basis,

$$\bar{s}_{i+1} = \hat{\lambda}_{i+1}(\hat{\sigma}_{i+1} - \hat{\sigma}_i) + s_i.$$

The difference of the two possible values for $\bar{s}_{i+1}$ is $(\hat{\lambda}_i - \hat{\lambda}_{i+1})(\hat{\sigma}_{i+1} - \hat{\sigma}_i) \geq 0$. Let us suppose that $d_i > 0$, arc $(i+1, i)$ is chosen, and the remaining arcs are selected so as to minimize $\sum_{j=i+1}^{n}(g_j - h_j)s_j$. Notice that if $(i, i+1)$ is chosen instead, the same choice of remaining arcs will cause *each* $s_j, j > i$ to be increased by $(\hat{\lambda}_i - \hat{\lambda}_{i+1})(\hat{\sigma}_{i+1} - \hat{\sigma}_i)$. Since the sums of g and h are equal, for a given $i < n$ if $d_i > 0$ the remaining sum $\sum_{j=i+1}^{n}(g_j - h_j)$ is negative, and increasing each s_j by a constant decreases the objective (2.19). Therefore if $d_i > 0$, the minimizing basis cannot contain the arc $(i+1, i)$. The resulting algorithm update_basis is given in Figure 2.5, and a corresponding update_dual_solution algorithm follows easily.

Figure 2.5: Algorithm to update dual basis

s, t = UPDATE_BASIS(g, h, λ, σ)

$t_1 = 0, s_1 = \lambda_1\sigma_1, d_0 = 0$

FOR $i = 1, \ldots, n-1$

$d_i = d_{i-1} + g_i - h_i$

IF $d_i > 0$,

$s_{i+1} = \lambda_i\sigma_{i+1} - t_i,$

$t_{i+1} = \lambda_{i+1}\sigma_{i+1} - s_{i+1}$

ELSE $(d_i \leq 0)$,

$t_{i+1} = \lambda_{i+1}\sigma_i - s_i,$

$s_{i+1} = \lambda_{i+1}\sigma_{i+1} - t_{i+1}$

Another way of deriving the above algorithm is to write (2.18) in the equivalent form

$$\begin{aligned} \max \quad & (e - \alpha g^k)^T \bar{s} + (e - \alpha h^k)^T \bar{t} \\ \text{s.t.} \quad & \bar{t}_i + \bar{s}_j \leq \hat{\lambda}_i \hat{\sigma}_j, \quad i, j = 1, \ldots, n-1, \end{aligned} \tag{2.20}$$

for all sufficiently small positive α. Clearly then $\bar{s}, \bar{t}$ are dual optimal (i.e. $e^T\bar{s} + e^T\bar{t} = \langle \lambda(\hat{A}), \lambda(\hat{B}) \rangle_-$), and minimize $g^T s + h^T t$ among dual optimal solutions. (2.20) is the

dual of a linear assignment problem:

$$\begin{aligned} \text{LAP} \qquad \min \quad & \sum_{i,j} \hat{c}_{ij}\hat{x}_{ij} \\ \text{s.t.} \quad & \sum_{j=1}^{n-1} \hat{x}_{ij} = 1 - \alpha h_i, \quad i = 1, \ldots, n \\ & \sum_{i=1}^{n-1} \hat{x}_{ij} = 1 - \alpha g_j, \quad j = 1, \ldots, n \\ & x_{ij} \geq 0, \quad i, j = 1, \ldots, n, \end{aligned}$$

for sufficiently small positive α. Logic similar to that used in Theorem 2.3.2 leads to the same update_basis algorithm.

For optimal solutions to (2.20) $\bar{s}^*, \bar{t}^*$, the difference $\Delta^k = (\bar{s}^k - \bar{s}^*)^T g^k + (\bar{t}^k - \bar{t}^*)^T h^k$ can be shown to be the *maximum* amount that QPB can possibly be increased by varying $\bar{s}$ and $\bar{t}$. In a branch-and-bound algorithm, this quantity can be monitored to decide whether it is worth the effort to obtain an improved bound. In general we have found that QPB is not very sensitive to the choice of $\bar{s}, \bar{t}$, but a few steps of the above procedure may give a worthwhile improvement in the bound.

2.4 A Long-step Path Following Algorithm for QP

To obtain the new bound QPB we require the solution, or approximate solution, of the quadratic programming problem QP(A, B, C). There are many algorithmic approaches that can be applied to such a problem. In the application at hand we do *not* require a highly accurate minimization, but if the problem is not solved exactly we *do* require a valid lower bound on the optimal value. In this section we describe a

simple "long-step path following" interior-point algorithm that approximately solves $\mathrm{QP}(A, B, C)$, and generates the required lower bound. The complexity analysis for such a method applied to a quadratic programming problem was considered in [4], and was extended to the case of a more general convex objective in [54]. The long-step path following algorithm is closely related to the classical SUMT technique of [29].

Consider a nonlinear programming problem (NLP)

$$\begin{array}{lrl} \text{NLP} & z^* & = \min f(x) \\ & \text{s.t.} & Hx = d \\ & & x \geq 0, \end{array}$$

where $y \in \Re^k$, H is an $m \times k$ matrix with independent rows, and $f(\cdot)$ is a convex function. We assume that the feasible region of NLP is compact, and contains a point having $x > 0$. $\mathrm{QP}(A, B, C)$ is clearly a problem of this form, with $x = \mathrm{vec}(X)$, $m = 2n - 1$, and $k = n^2$. The KKT conditions are necessary and sufficient for optimality in NLP, and can be written:

$$\nabla f(x)^T - H^T\nu - u = 0 \tag{2.21a}$$

$$u \geq 0,\ x \geq 0,\ Hx = d, \tag{2.21b}$$

$$u^T x = 0. \tag{2.21c}$$

It is also very well known that if x, ν, u satisfy (2.21a) and (2.21b), then

$$f(x) \geq z^* \geq f(x) - x^T u.$$

The **logarithmic barrier function** for NLP is

$$F(x,\mu) = f(x) - \mu \sum_{i=1}^{k} \ln(x_i),$$

where $\mu \geq 0$ is the barrier parameter. Since $F(\cdot,\mu)$ is strictly convex and the feasible region of NLP is compact, $F(\cdot,\mu)$ has a unique minimizer $x(\mu)$ on $\{x > 0 \,|\, Hx = d\}$ for each μ. Let $u(\mu) = \mu x^{-1}(\mu)$, where $x^{-1}(\mu)$ is the vector whose ith component is $1/x_i(\mu)$. It is then very easy to show that there is a $\nu(\mu)$ so that $x(\mu), u(\mu), \nu(\mu)$ satisfy (2.21a) and (2.21b), and therefore

$$f(x(\mu)) - k\mu \leq z^* \leq f(x(\mu)). \tag{2.22}$$

The long-step path following strategy for NLP is based on approximately minimizing $F(\cdot,\mu)$ using damped Newton steps, reducing μ, and repeating the process until a suitable approximate solution of NLP is obtained. It is clear from (2.22) that if a tolerance $\epsilon > 0$ is chosen, and $\mu \leq \epsilon/k$, then the exact minimizer $x(\mu)$ provides a lower bound within ϵ of the true solution value z^*. However such an exact minimizer is impractical or impossible to obtain, thus we require a methodology for obtaining a lower bound based on an *approximate* minimizer x of $F(\cdot,\mu)$. This can be done in several ways. One method, which is based on the known structure of $x(\mu), u(\mu)$, is to obtain u, ν by solving the auxiliary problem

$$\begin{aligned} \min \quad & \| \operatorname{Diag}(x)u - \mu e \| \\ \text{s.t.} \quad & \nabla f(x)^T - H^T\nu - u = 0, \end{aligned} \tag{2.23}$$

where x and μ are fixed. It is easy to show that ν solving (2.23) is given by $\nu = \bar{H}^T(\bar{H}\bar{H}^T)^{-1}\bar{H}b$, where $\bar{H} = H \operatorname{Diag}(x)$, and $b = \mu e - \operatorname{Diag}(x)\nabla f(x)^T$. For this ν, if

$u = \nabla f(x)^T - H^T\nu \geq 0$, then $f(x) - x^T u$ is a valid lower bound on z^*. In practice one can choose a tolerance ϵ, reduce μ until $\mu \leq \epsilon/(2k)$, and for this final value of μ approximately minimize $F(\cdot, \mu)$ until the above procedure generates $u \geq 0$ with $x^T u \leq \epsilon$.

In addition to providing a lower bound $\underline{v} = f(x) - x^T u$, the multiplier vector u is potentially useful for fixing variables when NLP is the continuous relaxation of an underlying discrete problem. Let $\bar{x}$ be an *integer* feasible solution for NLP, and assume that $f(\bar{x}) \leq \bar{v}$. From the convexity of $f(\cdot)$ we have $f(\bar{x}) \geq f(x) + \nabla f(x)(\bar{x} - x)$, and therefore

$$\begin{aligned} \bar{v} &\geq f(x) + \nabla f(x)(\bar{x} - x) \\ &= f(x) + (\bar{x} - x)^T(u + H^T\nu) \\ &= \underline{v} + \bar{x}^T u, \end{aligned}$$

where the last equality uses the fact that $Hx = H\bar{x} = d$. Since $\bar{x} \geq 0$, $u \geq 0$, and $\bar{x}$ has integer components, we can conclude that

$$\underline{v} + u_i > \bar{v} \implies \bar{x}_i = 0. \tag{2.24}$$

The fixing logic in (2.24) is a convex programming generalization of the well-known technique of variable fixing based on reduced costs in linear programming relaxations of discrete optimization problems; see for example [27]. Fixing logic based on reduced costs also exists for the GLB bound [67] and the HGB bound [45]. Our branch-and-bound implementation makes extensive use of this fixing logic.

Computational results indicate that the interior-point approach easily obtains good quality bounds for QAP, but the expense needed to obtain them may too great. A comparison of the IP approach with an alternative solution method is presented in Section 2.6.

2.5 Solution Using the Frank-Wolfe Algorithm

The previous two sections have described how to first formulate $\mathrm{QP}(A, B, C)$ by finding a dual solution to $\mathrm{SDD}(\hat{A}, \hat{B})$, and then solve it using an interior-point algorithm. Preliminary computational results in [3] indicate that QPB is a promising bound for many problems, but repeatedly solving $\mathrm{QP}(A, B, C)$ using an interior-point method within a branch-and-bound algorithm may take too long to solve QAPs of the size of interest. To address this concern we now present an iterative procedure that quickly gives approximate solutions to $\mathrm{QP}(A, B, C)$ of sufficient quality for use in a branch-and-bound algorithm. The Frank-Wolfe (or conditional gradient) algorithm is a technique for solving constrained optimization problems similar to steepest descent methods for unconstrained optimization. In the remainder of this section we show that though the Frank-Wolfe algorithm converges slowly, the particular structure of QPB can be exploited to produce an efficient solution procedure.

The standard Frank-Wolfe algorithm, shown in Figure 2.6, solves constrained optimization problems of the form $z^* = \min_{x \in S} f(x)$, where S is a convex set. Each iteration of the Frank-Wolfe (FW) algorithm finds a search direction in the feasible region that provides the largest possible decrease in the linearized objective func-

Figure 2.6: Frank-Wolfe algorithm

x = FW(f, S, x_0)

  FOR $i = 0, \ldots$,MAXIT:

    $g_i = \nabla f(x_i)$

    FIND d_i MINIMIZING $g_i^T d_i$, s.t.$d_i \in S$

    $\alpha_i = \arg\min_\alpha f((1-\alpha)x_i + \alpha d_i), 0 \leq \alpha \leq 1$

    $x_{i+1} = (1-\alpha_i)x_i + \alpha_i d_i$

tion. The Frank-Wolfe algorithm is guaranteed to converge to a global minimum but converges slowly in practice, like most algorithms based on steepest descent

Applying the Frank-Wolfe algorithm to QPB results in an efficient bound computation procedure. For simplicity we will use matrix notation in our presentation, with $f(X) = \mathrm{tr}[AXBX^T - SX - XT + CX^T]$, and $S = \mathcal{E} \cap \mathcal{N}$. Let

$$G = G(X) = 2(AXB - SX - XT) + C \tag{2.25}$$

denote the gradient of f at X; that is

$$G_{ij}(X) = \frac{\partial f(X)}{\partial X_{ij}}.$$

The most attractive feature of the FW algorithm in our application is that the minimization problem to be solved at each iteration is a linear assignment problem (LAP), which can be solved in $O(n^3)$ time. (Although our implementation uses a floating point LAP solver, many LAP solvers require integer input. In Chapter 6 we show how to adapt our solution procedure for use with an integer LAP solver.)

The steplength computation can also be calculated efficiently. The algorithm requires a steplength α minimizing $f(X_{i+1}) = f(X_i + \alpha D_i)$. Writing this condition as $G_{i+1} \bullet D_i = 0$ leads to

$$\alpha = \frac{-G_i \bullet D_i}{2(AD_iB - SD_i - D_iT) \bullet D_i}.$$

In Algorithm 2.6, the search direction D_i is given by $D_i = X_i^* - X_i$, where X_i^* is the solution of LAP(G_i). In this case, the gradient at the next iterate is

$$\begin{aligned} G_{i+1} &= G_i + 2\alpha(AD_iB - SD_i - D_iT) \\ &= G_i + 2\alpha(A(X_i^* - X_i)B - S(X_i^* - X_i) - (X_i^* - X_i)T) \\ &= G_i + \alpha(2(AX_i^*B - SX_i^* - X_i^*T) - G_i + C) \\ &= G_i + \alpha\Delta G_i, \end{aligned} \tag{2.26}$$

where $\Delta G_i = 2(AX_i^*B - SX_i^* - X_i^*T) - G_i + C$. The steplength is then given by

$$\alpha = (G_i \bullet X_i - G_i \bullet X_i^*)/(\Delta G_i \bullet X_i^* - \Delta G_i \bullet X_i). \tag{2.27}$$

Since it takes $O(n^2)$ time to multiply a matrix with a permutation matrix, if G_i is known only one $O(n^3)$ matrix multiplication (to compute AX_i^*B) is required to compute G_{i+1} and α. The solve_qp_fw algorithm of Figure 2.7 solves QP(A, B, C) using the FW algorithm with modifications (2.26) and (2.27).

Observe that each iteration requires the solution of one linear assignment problem, one matrix multiplication, and some $O(n^2)$ work. Given X_i, an upper bound $\bar{z}_i$

Figure 2.7: Solving QP(A, B, C) using the FW algorithm

X = SOLVE_QP_FW$(A, B, C, S, T, X_0, G_0)$

FOR $i = 0, \ldots$,MAXIT:

$X_i^* = \mathrm{LAP}(G_i)$

$\Delta G_i = 2(AX_i^*B - SX_i^* - X_i^*T) - G_i + C$

$\alpha_i = \min((G_i \bullet X_i - G_i \bullet X_i^*)/(\Delta G_i \bullet X_i^* - \Delta G_i \bullet X_i), 1)$

$X_{i+1} = (1 - \alpha_i)X_i + \alpha_i X_i^*$

$G_{i+1} = (1 - \alpha_i)G_i + \alpha_i \Delta G_i$

for QPB is computed efficiently as follows:

$$\begin{aligned} \bar{z}_i &= \mathrm{tr}(AX_iB - SX_i - X_iT + C)X_i^T + \langle \lambda(\hat{A}), \lambda(\hat{B}) \rangle_- \\ &= \frac{1}{2}(G_i \bullet X_i + C \bullet X_i) + \langle \lambda(\hat{A}), \lambda(\hat{B}) \rangle_- \end{aligned}$$

In addition, on each iteration we obtain the associated lower bound for QPB, which is also a lower bound for QAP:

$$z_i = \bar{z}_i + G_i \bullet (X_i^* - X_i). \tag{2.28}$$

Using (2.28), the FW algorithm can be terminated with a valid lower bound after any iteration.

The dual update procedure described in Section 2.3 can be applied using any feasible X. An additional parameter NUPDATE can be introduced so that the update_basis is called every NUPDATE iterations using the current X; finding new S, T that may result in an improved bound.

Figure 2.8: Comparison of solution procedures for QPB on nug20

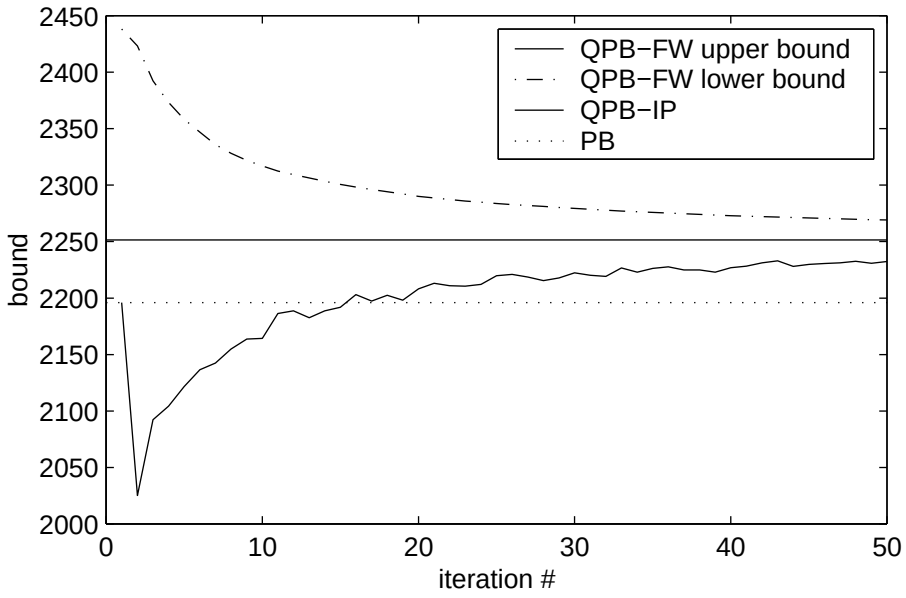


Figure 2.8 shows the convergence behavior of FW-QPB on the nug20 QAPLIB instance, which is representative of the behavior of the algorithm on most QAPs. The algorithm minimizes the upper bound $\bar{z}_i$, though our interest is actually in the lower bound z_i. Note that the lower bound dips considerably after the first iteration before recovering after about 15 iterations. For the nug20 problem the initial bound z_0 generated by the FW algorithm is 2178.3, which is precisely the value of the projected eigenvalue bound for the same problem, as shown in the figure. The following lemma shows that this is no coincidence.

Lemma 2.5.1 *Suppose that the Frank-Wolfe algorithm is applied to* $\mathrm{QP}(A, B, C)$ *with the initial solution* $X_0 = (1/n)ee^T$. *Then* $z_0 = \mathrm{PB}(A, B, C)$.

Proof: From (2.25),

$$G_0 = \frac{2}{n}\left(Aee^TB + See^T + ee^TT\right) + C = \frac{2}{n}(Aee^TB) + C,$$

because $S = V\hat{S}V^T$, $T = V\hat{T}V^T$, and $e^TV = 0$. Similarly

$$\begin{aligned} f(X_0) &= \operatorname{tr}(AX_0BX_0^T + SX_0X_0^T + X_0TX_0^T) + C \bullet X_0 \\ &= \frac{1}{n^2}\operatorname{tr}(Aee^TBee^T) + C \bullet X_0 \\ &= \frac{1}{n^2}(e^TAe)(e^TBe) + C \bullet X_0. \end{aligned}$$

Let $D = G_0 = C + (2/n)Aee^TB$. From (2.28) we then have

$$\begin{aligned} z_0 &= \langle\lambda(\hat{A}), \lambda(\hat{B})\rangle_- + f(X_0) + \mathrm{LAP}(D) - G_0 \bullet X_0 \\ &= \langle\lambda(\hat{A}), \lambda(\hat{B})\rangle_- + \mathrm{LAP}(D) + \frac{1}{n^2}(e^TAe)(e^TBe) - \frac{2}{n}(Aee^TB) \bullet X_0 \\ &= \langle\lambda(\hat{A}), \lambda(\hat{B})\rangle_- + \mathrm{LAP}(D) - \frac{1}{n^2}(e^TAe)(e^TBe) \\ &= \mathrm{PB}(A, B, C). \quad \square \end{aligned}$$

Note that since Aee^TB is a rank-one matrix, G_0 can be computed using $O(n^2)$ arithmetic operations.

The behavior illustrated in Figure 2.8 is typical in our experience. When initialized at $X_0 = (1/n)ee^T$ the bound z_1 drops sharply from the initial value $z_0 = \mathrm{PB}(A, B, C)$, and the bound sequence z_k, $k \geq 1$ then increases relatively steadily. In Chapter 6 we examine alternate initializations for the FW algorithm, and schemes for enforcing monotonicity on the bound sequence, however none of these efforts have produced reliable improvements in the overall performance of the algorithm.

Figure 2.9: Computing QPB using the FW algorithm

$X = \text{FW_QPB}(A, B, C)$

$(\lambda(\hat{A}), \lambda(\hat{B})) = \mathsf{eigenvalues}(\hat{A}, \hat{B})$

$(S', T') = \mathsf{dual_solution}(A, B)$

$X_0 = E/n$

$G_0 = \frac{2}{n}(Aee^T B) + C$

$z_{QP} = \mathsf{solve_QP}(A, B, C, S', T')$

$z = \langle \lambda(\hat{A}), \lambda(\hat{B}) \rangle_- + z_{QP}$

The complete algorithm for computing QPB using the FW algorithm (denoted QPB-FW) is given in Figure 2.9. In our implementation we use the Meschach [83] package to compute eigenvalues and the LAP solver of Jonker and Volgenant [49] to solve linear assignment problems. In the remainder of this chapter we compare QPB-FW to other bounding procedures, in terms of bound quality and CPU time.

2.6 Performance of QPB

The previous three sections have introduced a complete lower bounding procedure for QAP based on convex quadratic programming. We have given two different algorithms for solving the quadratic programs that arise: the more sophisticated, accurate interior-point approach of Section 2.4, and the conceptually simpler, iterative Frank-Wolfe procedure of Section 2.5. In this section we analyze the performance of QPB on a suite of test problems from QAPLIB [19]. In order to facilitate a com-

parison with other bounding methods we examine the same test problems used in [87]. In Table 2.1 for each problem we give the optimum value and lower bounds GLB, KCCEB, PB, QPB-IP, QPB-FW, EVB3, and SDPB1. All problems are homogenous instances of QAP ($C = 0$). For tai25a the optimum value is not known, and the value reported is the best known. GLB is the well-known Gilmore-Lawler bound. KCCEB is the dual LP-based bound from [52], obtained after 256 iterations. KCCEB is closely related to the dual bound of [45], and is a good approximation of the continuous LP bound computed in [78] (see [52] for details). PB is the projected eigenvalue bound, and QPB-IP is the quadratic programming bound of Section 2.4, using tolerance $\epsilon = 10^{-4}$. QPB-FW is the QP bound computed using the solution procedure of the previous section, using 150 FW iterations. EVB3 is the parametric eigenvalue bound of [76], and SDPB1 is the "basic" semidefinite programming bound μ_{R1}, from [87]. More elaborate SDP bounds for QAP are described in [61] and [87], but computationally these bounds are currently too costly for use in a branch-and-bound setting. The values of GLB, PB, EVB3, and SDPB1 are taken from [87], and those of KCCEB are taken from [52]; "n.a." denotes that a bound is not available for a particular problem. All bounds are rounded up to the next largest integer, since the data for these problems is integral and the objective value associated with any permutation matrix must therefore also be integral. (In fact it can be shown that the objective value must be an *even* integer for all of these problems, so bounds in Table 2.1 that are odd could be further increased by one.)

From Table 2.1 we see that for most problems the quality of the QP bounds is generally better than PB and worse than SDPB1. On some classes of problems, such as the esc and scr problems, the bounds produced by QPB are poor compared to GLB or KCCEB.

The speed of the FW algorithm more than compensates for the decreased bound quality. Table 2.2 compares bound quality and CPU times for QPB-IP, QPB-FW, and GLB on a subset of the QAPLIB problems. As the problem size is increased, the execution time of QPB-FW becomes better and better relative to the IP approach, with only a small degradation in bound quality. QPB-IP is coded in Matlab, whereas QPB-FW is coded in C++, however this difference accounts for only a small constant factor. Therefore, in the remainder of our experiments the FW algorithm is used. Although the bounds produced by GLB are obtained much more quickly relative to QPB-FW, the bound quality is much worse, particularly for the larger instances.

The accuracy of the FW algorithm is controlled by varying the number of FW iterations. Figure 2.10 shows how the gap between the QPB-FW and QPB-IP bounds decreases as more FW iterations are performed, for varying problem sizes. After 40-80 FW iterations the convergence of the algorithm slows considerably, but a bound within 2% of the interior-point solution is attained after only 20-30 iterations. Notice also that the convergence of the algorithm does not degrade considerably as the problem size increases.

The dual update procedure of Section 2.3.2 uses a suboptimal solution to

Table 2.1: Bounds for QAPLIB problems

Name	BKV	GLB	KCCEB	PB	QPB-IP	QPB-FW	EVB3	SDPB1
esc16a	68	38	41	47	55	50	50	47
esc16b	292	220	274	250	250	250	276	250
esc16c	160	83	91	95	95	95	113	95
esc16d	16	3	4	-19	-19	-19	-12	-19
esc16e	28	12	12	6	6	6	13	6
esc16g	26	12	12	9	9	9	11	9
esc16h	996	625	704	708	708	708	708	708
esc16i	14	0	0	-25	-25	-25	-21	-25
esc16j	8	1	2	-6	-6	-6	-4	-6
had12	1652	1536	1619	1573	1592	1583	1595	1604
had14	2724	2492	2661	2609	2630	2617	2643	2651
had16	3720	3358	3553	3560	3594	3581	3601	3612
had18	5358	4776	5078	5104	5141	5126	5176	5174
had20	6922	6166	6567	6625	6674	6654	6702	6713
kra30a	88900	68360	75566	63717	68257	67894	n.a.	69736
kra30b	91420	69065	76235	63818	68400	68013	n.a.	70324
nug12	578	493	521	472	482	473	498	486
nug14	1014	852	n.a.	871	891	879	898	903
nug15	1150	963	1033	973	994	982	1001	1009
nug16a	1610	1314	1419	1403	1441	1428	1455	1461
nug16b	1240	1022	1082	1046	1070	1057	1081	1082
nug17	1732	1388	1498	1487	1523	1509	1521	1548
nug18	1930	1554	1656	1663	1700	1681	1707	1723
nug20	2570	2057	2173	2196	2252	2233	2290	2281
nug21	2438	1833	2008	1979	2046	2027	2116	2090
nug22	3596	2483	2834	2966	3049	3024	3174	3140
nug24	3488	2676	2857	2960	3025	3002	3074	3068
nug25	3744	2869	3064	3190	3268	3243	3287	3305
nug30	6124	4539	4785	5266	5362	5328	5448	5413
rou12	235528	202272	223543	200024	205461	204128	201337	208685
rou15	354210	298548	323589	296705	303487	298232	297958	306833
rou20	725520	599948	641425	597045	607362	601995	n.a.	615549
scr12	31410	27858	29538	4727	8223	8024	n.a.	11117
scr15	51140	44737	48547	10355	12401	11909	n.a.	17046
scr20	110030	86766	94489	16113	23480	21873	n.a.	28535
tai12a	224416	195918	220804	193124	199378	199001	195673	203595
tai15a	388214	327501	351938	325019	330205	327459	327289	333437
tai17a	491812	412722	441501	408910	415576	414764	410076	419619
tai20a	703482	580674	616644	575831	584938	583356	n.a.	591994
tai25a	1167256	962417	1005978	956657	981870	965444	n.a.	974004
tho30	149936	90578	99855	119254	124286	123970	n.a.	125972

Table 2.2: A comparison of QPB-IP and QPB-FW

	QPB-IP		QPB-FW		GLB	
Problem	Gap	Time	Gap	Time	Gap	Time
nug12	0.166	1.28	0.174	0.014	0.147	0.0001
nug15	0.136	4.07	0.138	0.022	0.163	0.0002
nug18	0.119	10.34	0.123	0.035	0.195	0.0003
nug21	0.161	19.54	0.162	0.048	0.248	0.0004
nug24	0.133	47.36	0.136	0.067	0.233	0.0005
nug27	0.120	104.61	0.122	0.097	0.293	0.0006
nug30	0.124	177.76	0.127	0.130	0.259	0.0009

Figure 2.10: Convergence of FW algorithm as problem size increases

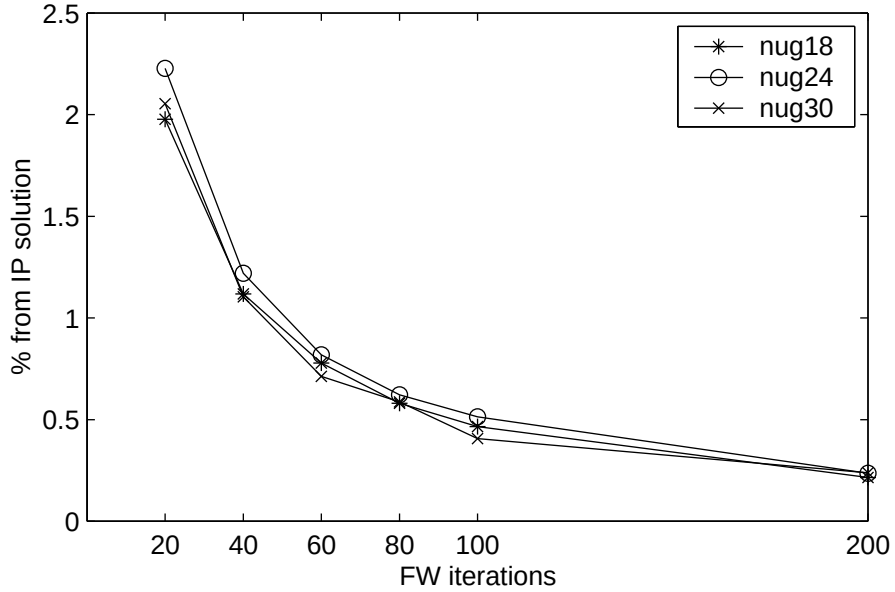


Table 2.3: Parametric improvement procedure applied to QPB-IP

Problem	BKV	QPB-IP	QPB-IP1
esc16a	68	55	55
had16	3720	3594	3595
had18	5358	5141	5143
had20	6922	6674	6677
nug16b	1240	1070	1071
nug18	1930	1700	1705
nug21	2438	2046	2055
nug24	3488	3025	3028
nug30	6124	5362	5365
rou15	354210	303487	303777
rou20	725520	607362	607822
scr15	51140	12401	12479
scr20	110030	23480	23960
tai17a	491812	415576	416033
tai20a	703482	584938	585139
tai25a	1167256	981870	983456
tho30	149936	124286	124684

$\mathrm{QP}(A, B, C)$ to obtain improved S, T. Repeatedly applying this procedure results in a parametrically improving lower bound. To obtain the results in Table 2.3, we used the interior-point algorithm to compute the initial QP bound QPB-IP0, and then updated the dual basis according to the procedure described in Section 2.3.2 to obtain QPB-IP1. On most problems, subsequent steps produce very small increases (or even decreases) in QPB, and the quantity Δ^k rapidly decreases. For many of the problems considered in Table 2.3 we obtained $\Delta^1 = 0$, indicating that no further increase in QPB is possible.

The parametric bound improvement procedure is better suited for use with the

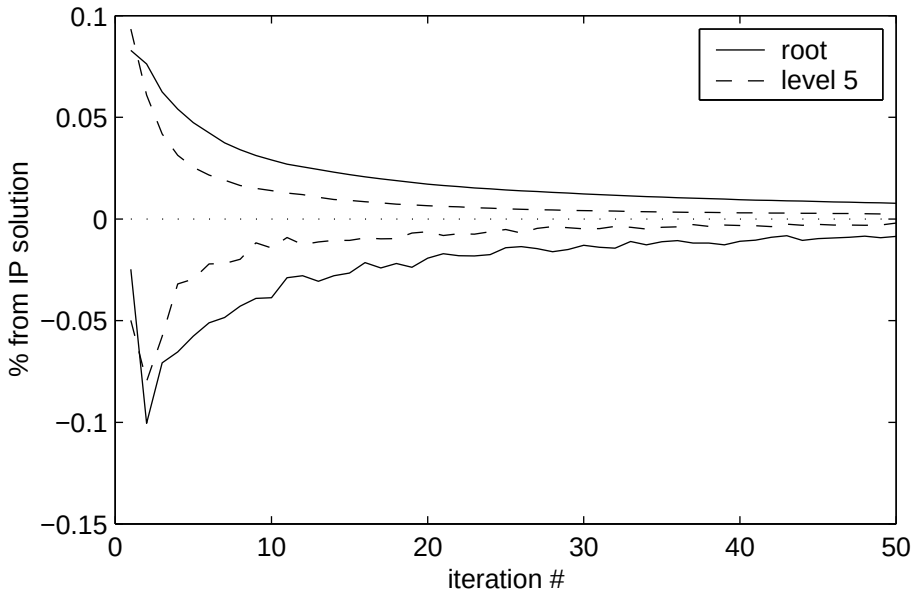
Table 2.4: Performance of QPB-FW on nug20 using dual update every NUPDATE iterations

	NUPDATE				
Iter	10	20	30	40	50
5	0.0579	0.0579	0.0579	0.0579	0.0579
15	0.0243	0.0267	0.0267	0.0267	0.0267
25	0.0154	0.0141	0.0143	0.0143	0.0143
35	0.0125	0.0111	0.0109	0.0114	0.0114
45	0.0094	0.0089	0.0078	0.0096	0.0098
55	0.0068	0.0071	0.0075	0.0057	0.0080
65	0.0057	0.0064	0.0062	0.0065	0.0049
75	0.0056	0.0057	0.0049	0.0052	0.0058
85	0.0040	0.0045	0.0052	0.0041	0.0049
95	0.0045	0.0033	0.0045	0.0044	0.0030
105	0.0047	0.0031	0.0036	0.0037	0.0031
115	0.0036	0.0040	0.0031	0.0034	0.0036
125	0.0031	0.0025	0.0026	0.0031	0.0027
135	0.0027	0.0028	0.0030	0.0023	0.0033
145	0.0023	0.0028	0.0027	0.0026	0.0025
Time	0.056500	0.051000	0.049000	0.048000	0.047000

Frank-Wolfe algorithm, because a dual update can be applied after every k Frank-Wolfe iterations with minimal cost. Table 2.4 shows that updating S, T every 20-40 iterations appears to give the best tradeoff in terms of time and bound quality.

A branch and bound algorithm repeatedly computes lower bounds on subproblems by assigning some of the facilities to locations, resulting in QAPs of smaller dimension. Assigning a facility to a location reduces the dimension of the flow and distance matrices A and B by one, and adds an additional term to the linear cost matrix C. Therefore, as more assignments are made, the linear term C becomes more dominant, making the quadratic programs easier to solve. Figure 2.11 shows how the

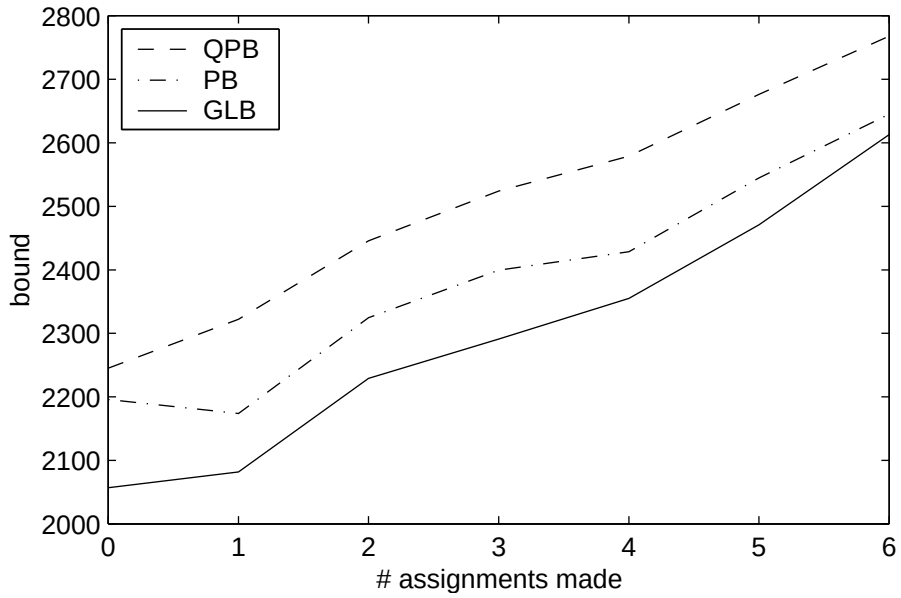
Figure 2.11: Convergence of FW algorithm by depth of problem



convergence of the FW algorithm improves as assignments are made. To measure how quickly the bound produced by the FW procedure approaches the bound produced by the IP algorithm, we plot on the y-axis the gap between QPB-FW and QPB-IP. After five assignments have been made, the gap decreases more quickly. As convergence improves, fewer FW iterations are required, and QPB-FW becomes progressively cheaper to compute.

In Section 2.2 we mentioned that a drawback of the projected eigenvalue bound is that in many cases the bound quality does not improve very much as assignments are made. Figure 2.12 gives a comparison of the GLB, PB, and QPB bounds as assignments are made on the nug20 QAPLIB problem. Facility i is assigned to location

Figure 2.12: Lower bounds for QAP by depth of problem



i for $i = 1, \ldots, 5$ to obtain successively smaller problems. Notice that PB actually decreases after the first assignment is made, but in general the difference in bound quality remains about the same. However, note that QPB becomes less expensive to compute relative to PB and GLB as assignments are made since fewer FW iterations are required to obtain an accurate bound.

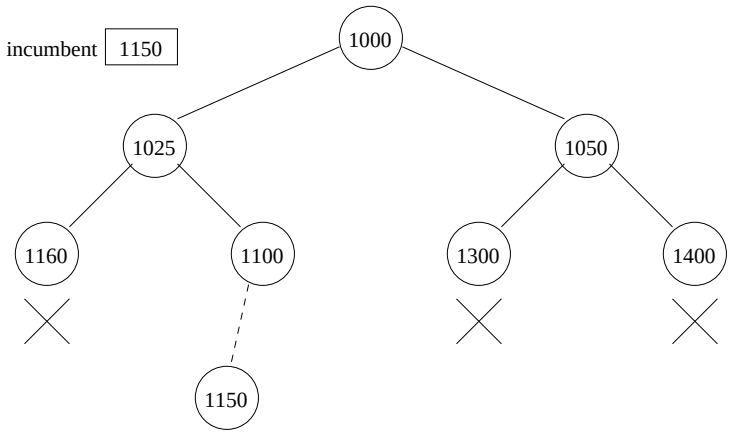
CHAPTER 3
BRANCH AND BOUND ALGORITHMS FOR QAP

3.1 Branch-and-Bound Algorithms

The quadratic programming lower bound described in the previous chapter is an important component of the branch-and-bound algorithm we use to solve large quadratic assignment problems. Branch-and-bound algorithms are enumerative procedures that have been used to solve a variety of difficult discrete optimization problems. In particular, branch-and-bound algorithms are the most used exact solution method for QAP. A few of the many implementations are [57, 40, 13, 24, 46, 65, 67].

A branch-and-bound algorithm repeatedly divides a problem into simpler subproblems of the same type, which are then solved or further subdivided. The search space of a problem is partitioned into search spaces for its subproblems. The algorithm generates a search tree with the root corresponding to the original problem to be solved. A generic branch-and-bound tree is shown in Figure 3.1. At each node in the search tree, a relaxed version of the subproblem is solved, typically by loosening the constraints of the subproblem. The solution to the relaxation constitutes a lower bound to the original subproblem. If a lower bound at a given node exceeds the value of a previously known discrete solution (the **incumbent value**), then continued searching on this path cannot lead to an improved solution, and the node is

Figure 3.1: Branch-and-bound tree



fathomed. In Figure 3.1 the nodes are labeled with their lower bound values, and fathomed nodes are marked with an X. If a node is not fathomed, the algorithm generates child subproblems and possibly searches for a better discrete solution to the problem. Eventually the algorithm reaches a subproblem which is easy enough to be solved exactly (perhaps by enumeration of all possible solutions), and at this point an improved solution to the problem may be found. A branch-and-bound algorithm finds the optimal solution to an optimization problem and proves that no other solution can be better.

The effectiveness of a branch-and-bound algorithm depends mainly on:

- How lower bounds are computed (**bounding**), and
- How child subproblems are created (**branching**).

Bounding and branching go hand-in-hand – information from the bound computation

is usually used to make branching decisions. Some early branching approaches for mixed integer linear programming (MILP) are described in [34] and a more recent study is given by Linderoth and Salvelsbergh in [62]. Branching strategies for QAP are investigated in detail in [67, 43]. In the case of QAP subproblems are usually created by assigning unassigned locations to unassigned facilities. An intelligent branching strategy tries to make assignments which result in subproblems that fathom as quickly as possible. Research in the area of branch-and-bound algorithms for QAP has focused primarily on bounding strategies, but branching strategies also play an important role in determining the effectiveness of the algorithm.

In the remainder of this chapter we describe in detail a branch-and-bound algorithm for QAP based on the QP bound of the previous chapter. Each component of the algorithm allows for a tradeoff between efficiency and quality of results produced. For example, increasing the number of Frank-Wolfe iterations in QPB produces better lower bounds. Branching decisions are made either by examining dual information provided by QPB, or by obtaining more information by prospectively computing bounds on subproblems. Several different branching rules are introduced in Section 3.3. A distinction between our algorithm and others is that different branching rules can be used in different parts of the tree. The algorithm uses sophisticated logic for selecting an appropriate branching rule for each subproblem. This logic allows the algorithm to concentrate more effort on difficult subproblems while keeping the total time spent under control.

Figure 3.2: Generic branch-and-bound algorithm

v = BRANCH-AND-BOUND(P)

1. $Q = \{P\}$
2. WHILE $Q \neq \{\}$:

   $S = \mathsf{next_problem}(\mathsf{Q})$

   IF $\mathsf{is_easy}(S), \bar{v} = \mathsf{solve}(S)$, IF $\bar{v} < v$, $v \leftarrow \bar{v}$, BACK TO 2.

   $R = \mathsf{relax}(S)$

   $z = \mathsf{solve}(R)$

   IF $z \leq v$

      $\bar{v} = \mathsf{heuristic}(R, z)$, IF $\bar{v} < v$, $v \leftarrow \bar{v}$.

      $\{C_1 \ldots C_k\} = \mathsf{subdivide}(S)$

      $Q = Q \cup \{C_1 \ldots C_k\}$

3.2 A Detailed Introduction to Branch-and-Bound

A generic branch-and-bound algorithm is given in Figure 3.2. The algorithm takes as input a minimization problem P to solve and an initial suboptimal solution with value v, and eventually finds the optimal solution to P by examining or ruling out all possible solutions.

The branch-and-bound algorithm first chooses a subproblem S to solve from a list or queue of unsolved subproblems Q. If S is not easily solved, the bounding phase begins by computing a lower bound on the solution to S via a relaxation R. The relaxed problem should be easy to solve, yet still retain as much of the structure of S as possible. The algorithm presented later on in this chapter uses the QP relaxation

and solves it using the efficient Frank-Wolfe algorithm of the previous chapter. If the resulting lower bound is larger than the incumbent solution value, the subproblem is fathomed. To keep the total number of considered nodes manageable, nodes should be fathomed as often as possible. Whether a node fathoms or not depends both on the bounding procedure and the difficulty of the subproblems encountered. If S does not fathom, a heuristic procedure can use the solution of R to try to find an improved incumbent value v. Next, S is subdivided into child subproblems $C_1, \ldots, C_k$. The branching phase ends when the subproblems are added to the queue.

The correctness of a branch-and-bound algorithm critically depends on the correctness of its bounding and branching phases. A lower bounding procedure is valid if it produces relaxations R with the following properties [34]:

- If R has no feasible solutions, neither does S.
- The minimum value of S is no less than the minimum value of R.
- If an optimal solution of R is feasible in S, then it is an optimal solution of S.

Unlike other problems such as general MILP, QAP subproblems almost always have feasible solutions, so for QAP the first property is not of great concern. Bounding procedures relax the set of permutations Π into some larger set, for example QPB relaxes Π into the set of doubly stochastic matrices. The derivation of QPB in the previous chapter showed that the second and third properties are satisfied; namely that $\text{QPB}(A, B, C) \leq \text{QAP}(A, B, C)$, and if a permutation solves QPB, then it is also optimal for QAP.

The branching phase is correct if it divides subproblems into child subproblems such that:

- Every feasible solution of S is a feasible solution of at least one of $C_1, \ldots, C_k$.
- Every feasible solution of $C_1, \ldots, C_k$ is a feasible solution of S.

Typically assignments of facilities to locations are made to produce child subproblems, though other methods are possible. The design of correct branching strategies is discussed in more detail in Section 3.5.

3.3 Node Selection Strategies

At any point during a branch-and-bound algorithm, there are several pending nodes waiting to be processed. The correctness of the algorithm is not affected by the choice; it is permissible to choose any node in the queue. However the overall efficiency of the algorithm *is* affected by the node selection procedure. Three common tree search strategies are breadth-first, depth-first and best-first. Most QAP implementations use a depth- or best-first strategy.

A FIFO (First In, First Out) strategy chooses the node that has been in the queue the longest. This strategy results in a breadth-first search of the branch-and-bound tree. FIFO strategies are rarely used in branch-and-bound algorithms because the size of the queue quickly becomes unmanageable. Due to the difficulty in obtaining tight lower bounds for larger problems, the number of nodes at each level of the tree grows exponentially for the first few levels.

A LIFO (Last In, First Out) strategy selects the node most recently added to the queue, resulting in a depth-first search of the tree. LIFO strategies are often used in branch-and-bound algorithms because they minimize the size of the queue. Deep nodes in the tree are explored early on in the search, allowing for the possibility of an improved incumbent solution to be found. An additional feature is that such a strategy is efficient and very simply coded.

A best-first strategy orders the nodes according to a given criterion, and chooses the node maximizing (or minimizing) the criterion. The goal of a best-first strategy is to find an optimal solution to the problem as quickly as possible. If an optimal solution is found quickly, the algorithm will fathom as many as nodes as possible as quickly as possible, minimizing the size of the search tree. Therefore, the selection criteria used in best-first branch-and-bound algorithms often relate to the estimated solutions to subproblems. If the estimates are accurate, choosing the node with the smallest estimated solution value will quickly lead to finding the optimal solution value. The drawback of this approach is that the nodes in the queue must be ordered, causing insertion of new nodes to the list to be more time consuming. For some problems this extra cost is not of great concern, but for QAP it may not be worth the effort required to keep the list ordered, since heuristics are often able to find optimal or nearly optimal solutions to the problem at the start of the algorithm.

Node selection becomes more interesting in a parallel setting where several different processors may be working on several different subproblems $S^1, \ldots, S^p$ and

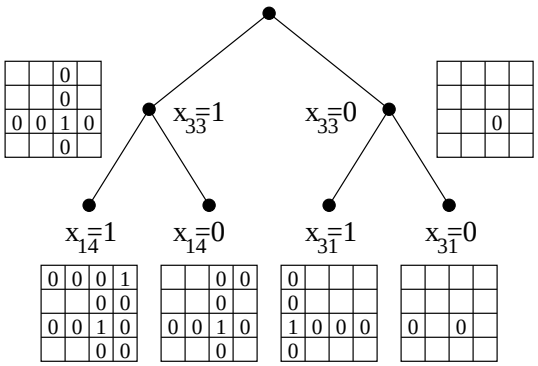
will return sets of child subproblems $C^1, \ldots, C^p$. In such a case, load balancing and starvation become potential hazards. Further discussion of parallel implementation of branch and bound algorithms for QAP is delayed until Chapter 5.

3.4 Heuristics

The solution to a relaxed subproblem can be used by a heuristic procedure in an attempt to find an improved incumbent solution to QAP. The type of heuristic used, and the quality of the upper bounds produced depend on the information provided by the bounding procedure. In our implementation, we have chosen not to implement a heuristic step since for the problems of interest, the initial incumbent value is often close to the optimal solution value. Since a future implementation may include such a heuristic, we describe one possibility briefly.

QPB searches over nonnegative matrices with row and column sums equal to one. Therefore a simple (primal) heuristic is to find the permutation matrix $\bar{X}$ most closely matching the doubly stochastic matrix X solving $\mathrm{QP}(A, B, C)$, and check if the permutation improves on the incumbent value. Solving the linear assignment problem $\mathrm{LAP}(-X)$ provide such an $\bar{X}$. In fact, this technique can be applied in conjunction with any lower bounding procedure that produces a continuous solution X, for example bounds based on linear programming [52]. This idea is pursued further to construct a simple heuristic for QAP in Chapter 6.

Figure 3.3: Single assignment strategy for QAP



3.5 Branching Strategies for QAP

The branching phase is a crucial step in the design of a branch-and-bound algorithm. In this step, a subproblem S is subdivided into child subproblems $C = \{C_1 \ldots C_k\}$. QAP is an optimization problem over the set of all permutations Π of $\{1, \ldots, n\}$, and deeper subproblems represent searches over successively smaller subsets of Π. To completely specify the branching phase it is necessary to specify how many child subproblems to create, and how to determine the search spaces of each child. These choices have a great impact in determining the overall efficiency of the branch-and-bound algorithm; a poor branching decision at a node may produce child nodes that are nearly as difficult to solve as the parent node.

3.5.1 Single Assignment Branching

The first issue is to determine how many child subproblems to create at each node. Most previous implementations have chosen either single assignment or poly-

tomic strategies. A single assignment branching strategy divides each subproblem into two child subproblems by assigning or disallowing the assignment of one facility to one location. Such a strategy was used in [73] which solved previously unsolved QAPLIB instances. Figure 3.3 shows an example of single assignment branching–to form the left child of the root node, facility 3 has been assigned to location 3 (denoted $3 \rightarrow 3$), and to form the right child the assignment $3 \rightarrow 3$ has been disallowed. Viewing the solution as a permutation matrix, the assignment $3 \rightarrow 3$ is equivalent to assigning $x_{33} = 1$, and disallowing $3 \rightarrow 3$ becomes $x_{33} = 0$. A key observation is that making an assignment is a much more powerful operation than disallowing an assignment; that is, if $x_{33} = 1$, then $x_{i3} = 0$ for $i \neq 3$ and $x_{3j} = 0$ for $j \neq 3$. Setting $x_{ij} = 1$ determines $2n-1$ entries of the permutation matrix X. After several branchings subproblems at the same level of the tree vary in difficulty. Regardless of the bound used, it is reasonable to expect that at any given node in the tree the bound values in the right subtree will often be much lower than those in the left subtree.

Assigning one facility to a location results in a QAP which is of dimension one smaller than its parent. Let $A_{(ij)}$ denote matrix A with row i and column j removed. Also, let $\hat{a}_i$ be the elements in row i of A, excluding a_{ii}, and let $\hat{b}_j$ be the elements in column j of B, excluding b_{jj}. If for a symmetric QAP(A, B, C) $x_{ij} = 1$, then we

obtain the smaller QAP(A', B', C') with

$$\begin{aligned} A' &= A_{(ii)} \\ B' &= B_{(jj)} \\ C' &= C_{(ij)} + 2\hat{a}_i\hat{b}_j, \end{aligned} \tag{3.1}$$

and a constant term $d' = a_{ii}b_{jj} + c_{ij}$ added to the objective.

Disallowing an assignment by setting $x_{ij} = 0$ for some $i, j \in \{1, \ldots, n\}$ results in a QAP of the same dimension as its parent. One can prohibit the assignment $i \to j$ from being made by setting $c_{ij} = M$ for sufficiently large positive M. When all but one value in a row or column of X has been disallowed, the dimension of the problem can be reduced by one. After a few variables have been disallowed, it may become difficult to tell whether a given subproblem actually has a solution or not: once x_{32} is set to 0, the left subproblem in Figure 3.4 has no solution, but the right subproblem has a unique solution. The zero entries in Figure 3.4 denote entries of X_1, X_2 that have already been disallowed, and question marks indicate undetermined entries. A branch-and-bound algorithm using a binary branching strategy should be able to recognize and handle properly subproblems with no feasible solution. This problem is equivalent to determining whether a bipartite graph between two sets of n nodes has a complete matching.

3.5.2 Polytomic Branching

Polytomic branching was introduced by Mautor and Roucairol in [67] and is the most commonly used strategy for algorithms capable of solving large scale QAPs. A

Figure 3.4: Binary branching may produce infeasible subproblems

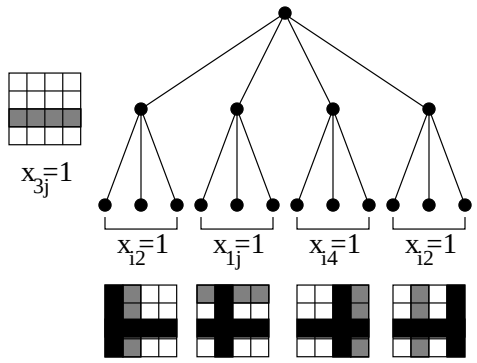
$$X_1 = \begin{bmatrix} ? & ? & ? & 0 & 0 \\ ? & 0 & ? & 0 & 0 \\ 0 & ? & 0 & ? & 0 \\ 0 & 0 & 0 & ? & ? \\ 0 & 0 & 0 & ? & ? \end{bmatrix} \qquad X_2 = \begin{bmatrix} ? & 0 & ? & ? & 0 \\ ? & 0 & ? & 0 & 0 \\ 0 & ? & 0 & ? & 0 \\ 0 & ? & 0 & 0 & ? \\ 0 & ? & 0 & 0 & ? \end{bmatrix}$$

polytomic branching strategy assigns an unassigned facility to each available location, or all unassigned facilities to one available location. We refer to the former as **row branching**, and the latter as **column branching**. If a polytomic strategy is used, the maximum depth of the tree is n, and the depth of a node is simply the number of assignments made. Figure 3.5 depicts the first two levels of a branch-and-bound tree using a polytomic branching rule. The root node produces four child nodes using column branching, and these subproblems in turn each produce three children. As Figure 3.5 indicates, it is permissible to use both row and column branching at the same level in the tree – branching decisions are usually made locally without regard to decisions made elsewhere in the tree.

3.5.3 Branching Rules for QAP

Our implementation exclusively uses polytomic branching strategies, so we focus on designing branching strategies for a polytomic framework. However, many of the techniques described here can be adapted for use in a binary branching framework. The task is to choose a facility (or location) that when fixed to all remaining locations

Figure 3.5: Polytomic branching strategy for QAP

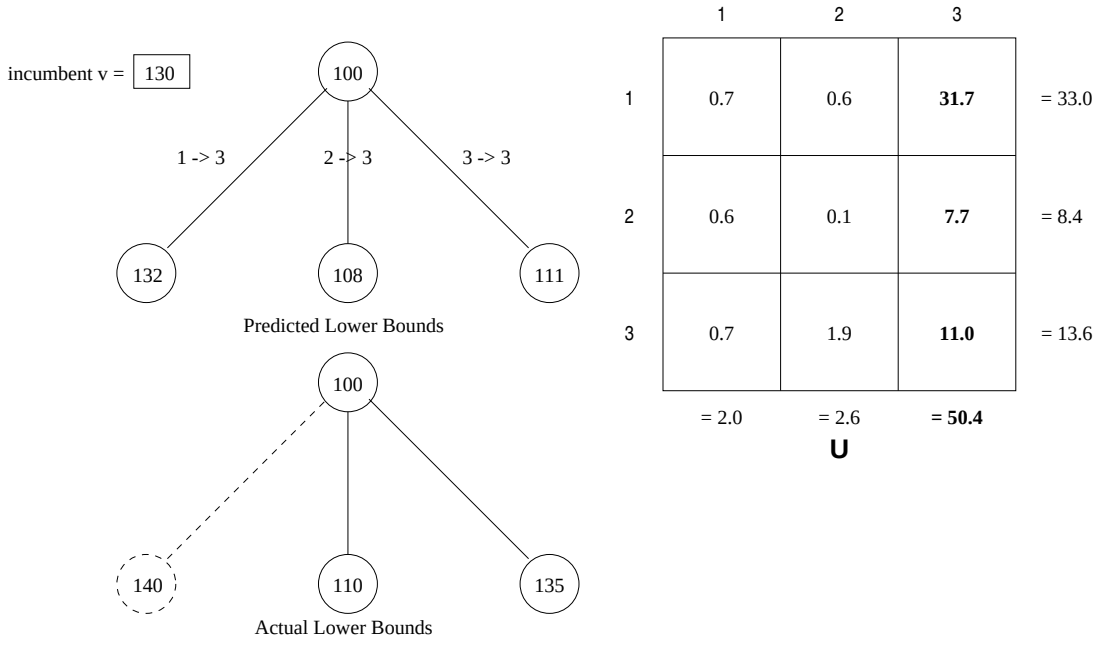


(facilities) results in child subproblems that are fathomed as quickly as possible. The ability to make such a choice depends on two factors:

- How is the difficulty of a subproblem estimated?
- How are the estimates of all the possible subproblems combined to make a choice of facility or location to branch on?

Our first set of branching rules is based on the dual matrix U provided by the QPB. In what follows, we assume that the algorithm is making a branching decision at the root of the tree. Specifically, the problem to solve is QAP(A, B, C) of size n, and a lower bound on its solution is z, that is, the lower bound at the root of the tree is z. An incumbent solution v is also known, i.e. v is an upper bound on the solution to QAP(A, B, C). Let us introduce notation for the possible children of the root problem – let S_{ij} be the QAP of size $n-1$ where facility i is assigned to

Figure 3.6: Branching Rule 1



location j, whose construction is given in formula (3.1). Recall from the discussion in the previous chapter that QPB produces a dual matrix U such that if $z + u_{ij} > v$, then the assignment $i \rightarrow j$ cannot possibly lead to a solution better than v. Thus U can be used to eliminate subproblems and predict which subproblems are likely to be easiest to solve. Other bounds such as GLB and HGB also produce dual matrices with similar properties, see [46, 67].

The first of our four branching rules is fairly intuitive. The rule is to select the row (or column) that maximizes the row (or column) sum of U. Such a choice will produce subproblems with predicted lower bounds that are as high as possible. Figure 3.6 shows a simple example of how the dual matrix is used to make such a

choice. The sum of the entries in column 3 of U is greatest, so each facility is assigned to location 3 to generate child subproblems. Notice that since $z + u_{13} > 130$ there is no need to consider the first subproblem.

An extension of the basic idea of Rule 1 is to weight each u_{ij} via a weighting function w. To give more weight to subproblems with higher u_{ij} one can compute the **gap reduction factor** (grf):

$$\text{grf} = \min(1, \frac{u_{ij}}{v - z})$$

and then a corresponding weighting function

$$w(u_{ij}) = [(1 - W)\,\text{grf} + W]u_{ij} \tag{3.2}$$

where W is a parameter to be chosen. If $W = 1$ then $w(u_{ij}) = u_{ij}$, and if $W = 0$, $w(u_{ij}) = \text{grf}\, u_{ij}$, which gives more weight to nodes with a high gap reduction factor.

The second rule is to try to minimize the number of child nodes created. The analog of Rule 2 was previously used for the traveling salesman problem by Lawler et al. in [59], and is quite similar to the branching rule used in [67]. Define the set $N_i' = \{j \mid z + u_{ij} < v, j = 1, \ldots, n\}$, for a fixed row i. N_i' consists exactly of the child problems with $X_{ij} = 1$ that *cannot* be eliminated. For some nodes, there may be several rows or columns minimizing the number of child subproblems created – in this case the row or column sum of the remaining children can be used as a tiebreaker. There is reason to believe that minimizing the number of children created is a better branching rule than maximizing the row sum. Since any bound exceeding

v fathoms, all subproblems with bounds greater than v should be treated the same. The computational results in Section 3.8 show that Rule 2 is effective in reducing the size of the tree for small problems.

Rules 1 and 2 are similar in the respect that they both scan the dual matrix U to make a branching decision. In fact, Rules 1 and 2, as well as many of those proposed by other researchers are special cases of a more general branching rule. The rule, given in figure 3.7 takes as input a matrix U and a weighting function f. Rule 1 corresponds to $\mathsf{weak}(U, w)$, where U is the dual matrix and w is given by (3.2). Rule 2 corresponds to $\mathsf{weak}(F, i)$ for the identity function i and a matrix F with entries

$$f_{ij} = \begin{cases} M, & \text{if } z + u_{ij} \geq v \\ u_{ij}, & \text{if } z + u_{ij} < v \end{cases} \tag{3.3}$$

where M is a large constant. Defining F in this manner places highest priority on subproblems that fathom, and a secondary priority on subproblems with high u_{ij}.

Researchers have examined several other branching rules using the U matrix. A description of some of these rules, as well as a comparison of their performance using the HGB bound is given by Hahn, et al. in [43]. An example of such a rule is to choose the facility with the largest number of predicted bounds above the median.

Our final two branching rules obtain more information by "prospectively" setting $x_{ij} = 1$, and computing QPB for the associated QAP problem of dimension $n-1$, before making the final decision of where to branch. In our experience, branching rules using U are valuable once we are deep enough in the tree so that a non-trivial fraction of the subproblems can be eliminated. Near the root of the tree we desire

Figure 3.7: Branching using reduced cost matrix U

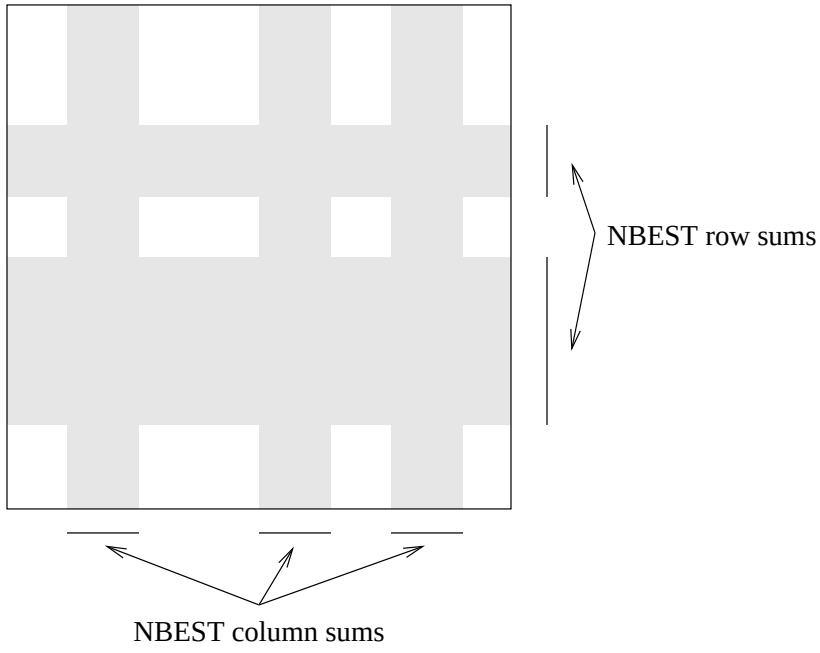
WEAK(U, f)

1. $r_i = \sum_{j=1}^{n} f(u_{ij}), \quad i = 1, \ldots, n$
2. $c_j = \sum_{i=1}^{n} f(u_{ij}), \quad j = 1, \ldots, n$
3. $i_{\max} = \arg\max r_i$
4. $j_{\max} = \arg\max c_j$
5. IF $\max r > \max c$ BRANCH ON $i_{\max}$, ELSE BRANCH ON $j_{\max}$

branching rules that produce more accurate information about the difficulty of the possible subproblems. In this case, we prospectively set $X_{ij} = 1$ and compute lower bounds z_{ij} on the resulting S_{ij}. This type of branching is analogous to the well-known technique of "strong branching" for integer and mixed-integer linear programming, see for example [62]. Hahn, et al. were the first to compute lower bounds for the purpose of making branching decisions in a QAP branch-and-bound algorithm. The z_{ij} can be gathered in a matrix Z which can be used in the same manner as the dual matrix U to make branching decisions.

Computing all possible z_{ij} requires n^2 bound computations, which may be too costly. Hahn, et. al. proposed a sampling procedure HGBSMPL-k where only every k-th entry of the Z matrix is computed. We propose computing z_{ij} only if row i or column j is one of the k most promising rows and columns as determined by the U matrix. More formally, we define the sets I_S, J_S having the k highest row and

Figure 3.8: Bounds are computed only for NBEST rows and columns



column sums, respectively. For each (i, j) where either $i \in I_S$ or $j \in J_S$ we compute a lower bound on S_{ij}. A picture of this technique is given in Figure 3.8 – bounds are computed only for the shaded entries of U.

Another technique for reducing the cost of strong branching is to compute the z_{ij} bounds using less computation than is used to compute z. The cost of computing QPB is easily reduced by decreasing the maximum number of FW iterations performed. Lemma 2.5.1 shows that performing one FW iteration produces the projected eigenvalue bound; however PB is known to be ineffective as a basis for making branching decisions. Also recall that for the first few FW iterations, QPB typically decreases. As a result, when computing bounds to be used for branching purposes

Figure 3.9: Branching using prospective bound computation

STRONG(Z, U, w)

1. PERFORM STEPS 1-2 OF weak(U, w).
2. LET I DENOTE THE SET OF ROWS HAVING THE NBEST HIGHEST VALUES OF r_i, $i = 1, \ldots, n$.
3. LET J DENOTE THE SET OF COLUMNS HAVING THE NBEST HIGHEST VALUES OF c_i, $i = 1, \ldots, n$.
4. $\bar{z}_{ij} = \begin{cases} z_{ij} & \text{IF } i \in I \text{ OR } j \in J, \\ 0 & \text{OTHERWISE.} \end{cases}$
5. weak$(\bar{\mathsf{Z}}, \mathsf{i})$

we usually perform between 25 and 50 FW iterations.

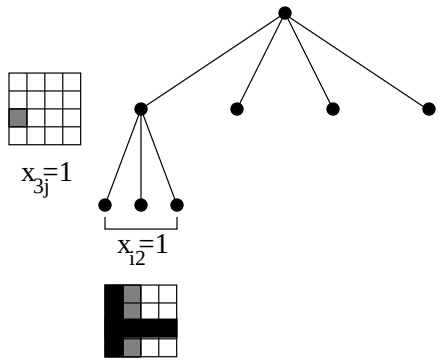
Our third rule is analogous to Rule 1 in that it chooses the row or column with the highest sum of bounds. A general technique for branching rules that compute prospective bounds is given in Figure 3.9. Rule 3 is simply strong(Z, U, w) where U, w are defined as in Rule 1 and

$$z_{ij} = \text{QPB}(S_{ij}, \text{FW3})$$

where $\text{QPB}(S_{ij}, \text{FW3})$ is obtained by computing QPB on the reduced problem S_{ij} using a maximum of FW3 Frank-Wolfe iterations.

A related idea is to use a different bound altogether for branching, for example the cheaply computed Gilmore-Lawler bound. Unfortunately, this rule did not improve the performance of the algorithm. Variations on this idea are examined further in Chapter 6.

Figure 3.10: Branching Rule 4: a "look-ahead" rule



Rules based on the dual U matrix essentially make branching decisions based on inexact information about the subproblems one level down in the tree. Rules based on computing z_{ij} use actual bounds on the child subproblems to make better decisions. The final branching rule uses information about the subproblems two levels down the tree. When a prospective bound z_{ij} is computed using the previous rule, there is an associated dual matrix U^{ij}, of dimension $n-1$. Then it is clear that $z_{ij} + U^{ij}_{kl}$ is a lower bound on the solution value of the QAP obtained by assigning both $i \to j$ and $k \to l$.

Rule 4 is based on applying Rule 1 to each of these matrices U^{ij}. Rule 4 can be thought of as a "look-ahead" branching rule that tries to maximize the total increase in the bounds after 2 levels of branching. Define

$$\hat{z}_{ij} = (|N| - 1)z_{ij} + v^{ij}$$

where v^{ij} is the maximal row sum of U^{ij}. Rule 4 is then $\mathsf{strong}(\hat{Z}, U, w)$.

Table 3.1: Summary of branching rules

Rule 1	$\mathsf{weak}(U, w)$
Rule 2	$\mathsf{weak}(F, i)$
Rule 3	$\mathsf{strong}(Z, U, w)$
Rule 4	$\mathsf{strong}(\hat{Z}, U, w)$

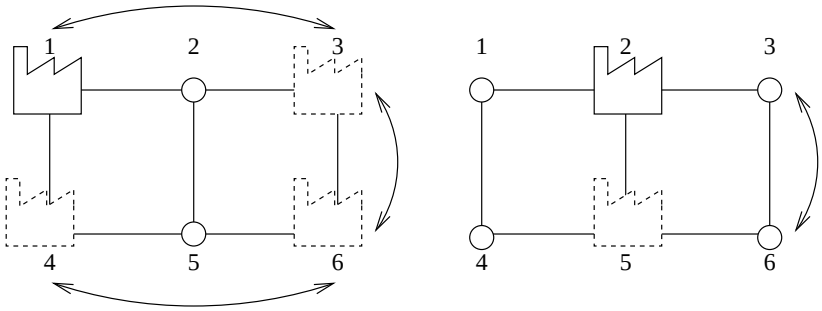
Table 3.1 summarizes the four branching rules used in our branch-and-bound algorithm. The first two scan U to make a branching decision, using slightly different criteria. The last two compute prospective bounds using a small number of FW iterations, but only for the most promising subproblems. Rule 4, the "look-ahead" rule, goes even further than Rule 3 by examining the dual matrices U^{ij} produced by the prospective bound computations to predict the lower bounds on subproblems two levels down in the tree.

3.6 Exploiting QAP Symmetry

Many QAPs arise from applications where the distance matrix B exhibits symmetries that can be exploited to reduce the number of child problems generated in the branching process. In a problem with symmetries there is a subset of locations $J_1 \in \{1, \ldots, n\}$ such that at the root of the branch-and-bound tree, one need only consider assigning a facility i to locations $j \in J_1$. As an example consider the problem nug06, where the distance matrix corresponds to the l_1 distances on a 2×3 rectangular grid, see Figure 1.8.

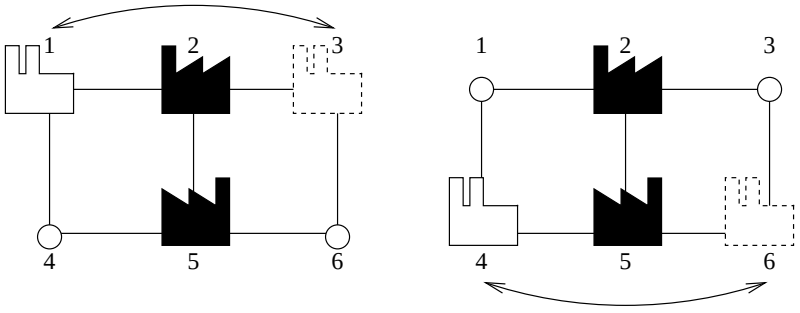
In this case we take $J_1 = \{1, 2\}$, as indicated by Figure 3.11. Additionally,

Figure 3.11: Symmetry of nug06: $J_1 = \{1, 2\}$



there may be symmetries remaining in the problem, even after assignments have been made. There may be one or more subsets of locations J_2 so that if at any node in the tree the set of fixed assignments $\bar{J} = \{j \mid x_{ij} = 1, i = 1, \ldots, n\}$ satisfies $\bar{J} \subset J_2$, then the children of the node can be restricted to be of the form $X_{ij} = 1, j \in J_3$, regardless of the choice of $i \in I$. For nug06, $J_2 = \{2, 5\}, J_3 = \{1, 2, 4, 5\}$ as indicated by Figure 3.12. The fixed locations are darkened, and the dashed locations are those eliminated by symmetry. Larger problems may have more than one pair of J_2/J_3 sets.

Figure 3.12: Symmetry of nug06: $J_2 = \{2, 5\}, J_3 = \{1, 2, 4, 5\}$



A different approach to handling symmetry [9, 67] is to test for it at each node by analyzing the distance matrix of each subproblem. Such a procedure requires $O(n^2)$ computation, whereas our procedure is $O(n)$.

The branching rules of the previous section are also affected when symmetry exists. As described, symmetry only applies when row branching is performed. Therefore at each node, it is first determined whether symmetry remains in the subproblem, that is whether the set of locations under consideration can be reduced to J_1 or J_3. If symmetry remains, then row branching is performed using only the locations J_1 or J_3. For example, if Rule 1 is applied at the root, the row sums are computed using only u_{ij} for $j \in J_1$, $i = 1, \ldots, n$.

3.7 Specifying a Complete Branching Strategy

In Section 3.5 four branching rules were introduced. We now give a framework for combining these rules into a branching strategy. To completely specify the branching rules a number of parameters must be chosen. These are:

NFW1. Maximum number of FW iterations used.

NFW2. Maximum number of FW iterations used if node cannot be fathomed.

NFW3. Number of FW iterations used for prospective bound computations.

NBEST. Number of rows/columns in which to compute prospective bounds.

NUPDATE. Number of FW iterations between update of dual matrices S, T.

We now give some details regarding these parameters. The QP bound of Chapter 2 is an iterative procedure that produces an upper bound v_k and lower bound z_k at each iteration k. If $z_k > v$ the current node can be fathomed, and the bound computation is terminated. On the other hand if $v_k < v$ then we know that the lower bound from QPB will *not* be high enough to fathom the current node. By setting NFW2<NFW1 we allow for earlier termination in the latter case. (Note however that even when a node cannot be fathomed it is desirable to compute a reasonably accurate bound z for branching purposes.) The complete logic for the number of FW iterations is then that we terminate on iteration k if $z_k > v$, or $v_k < v$ and $k \geq$ NFW2, or k=NFW1.

As described in the previous chapter, QPB(A, B, C) involves the choice of a dual basis depending on the optimal solution of a particular semidefinite programming problem. Such an optimal solution is not in general unique, and different choices of a dual basis may produce different values of QPB(A, B, C). Given a choice of dual basis, and an approximate solution X of QP(A, B, C), a simple procedure is described in Chapter 2 that attempts to generate a new dual optimal solution that will increase QP(A, B, C). We apply one step of this procedure every NUPDATE FW iterations, using the current primal solution X_k as the basis for the update.

In general we combine the four different branching rules given above, with suitable parameter choices, to obtain a complete branching strategy. In our implementation the choice of branching rule is based on depth in the tree and relative gap.

Table 3.2: Branching Strategy A

Gap	Depth	Rule	NFW1	NFW2	NFW3	NBEST	NUPDATE
0.0	50	2	150	100	–	–	30

Table 3.3: Branching Strategy B

Gap	Depth	Rule	NFW1	NFW2	NFW3	NBEST	NUPDATE
0.5	2	4	150	100	50	10	30
0.0	50	2	150	100	–	–	30

The depth is simply the number of fixed assignments $|\bar{I}|$. The relative gap, as in [23], is defined as

$$g(z) = \frac{v - z}{v - z_0}. \tag{3.4}$$

where z_0 is the lower bound at the root node of the branch-and-bound tree. In our experience, the relative gap is a more accurate predictor of the difficulty of a subproblem than the depth. Taking into account the relative gap of a node allows us to spend less time on easier subproblems and spend more time on more difficult ones.

To select an appropriate branching rule, we associate with each parameter set a depth and gap cutoff. For example, consider the branching strategies in Tables 3.2, 3.3. Notice that the parameters can be divided into bound-related parameters (NFW1, NFW2, NFW3, NUPDATE), branching parameters (NBEST,Rule), and parameters that determine when the rule is used (Gap, Depth). Strategy A uses Rule 2,

with NFW1=150, NFW2=100, updating the dual matrices every 30 FW iterations, for all nodes in the tree. Strategy B uses Rule 4 on all nodes on levels 0-2 that have a relative gap less than 0.5, and rule 2 elsewhere. For strategies with more than one branching rule, the rules are scanned top-to-bottom to find a rule where the relative gap is less than or equal to the "Gap" entry, and the depth of the node is less than or equal to the "Depth" entry. If no such rule is found, the last rule is used by default.

Although the relative gap is a good indicator of problem difficulty, a rule selection scheme based solely on relative gaps would be quite difficult to use on larger problems. A small change in a relative gap may cause an expensive branching rule to be used many more times, since the number of nodes at each level of the branch-and-bound tree can grow quite quickly. The Depth parameter can be used to ensure that the use of an expensive branching rule does not become excessive.

3.8 Branching-Related Experiments and Results

The bounding strategy of Chapter 2, the basic branch-and-bound framework of Section 3.2, and the branching rules and selection strategies of the previous subsections were assembled to produce a branch-and-bound algorithm for QAP, outlined in Figure 3.13. (Appendix A describes implementation details, and Chapter 5 describes the distributed implementation and reports computational results on large and previously unsolved problems.)

Let us point out a few details of the algorithm. An additional parameter to the

Figure 3.13: Branch-and-bound algorithm for QAP

```
v = BRANCH-AND-BOUND(P, Δ)
  1. Q = {P}.
  2. WHILE Q ≠ {}:
        S = pop(Q)
        IF parent_bound(S) ≤ v − Δ, BACK TO 2.
        IF SIZE(S) = 3, v̄ = enumerate(S), IF v̄ < v, v ← v̄, BACK TO
        2.
        [RULE,PARAMS] = select_rule(S)
        z = QPB(S,PARAMS)
        IF z ≤ v − Δ
           {C_1 ... C_k} = subdivide(S,RULE)
           push(Q, {C_1 ... C_k})
```

algorithm is a fathoming tolerance Δ. For example, if the input problem is integer, the solution value must be integer, and we set $\Delta = 1 - \epsilon$, where ϵ is a small positive number that allows for roundoff errors in the bound computation. If the solution value to the problem is known to be even, $\Delta = 2 - \epsilon$. Increasing Δ further results in a suboptimal solution procedure for QAP. The algorithm keeps track of the lower bound of a node's parent, so that when the incumbent is updated, no wasteful bound computation is performed. When the subproblem is a QAP of size three, all six possible solutions are enumerated and the incumbent value is updated as necessary. The select_rule routine picks a branching rule and parameters for QPB based on the relative gap and depth of the subproblem, using the logic of the previous subsection.

Rules 1, 3, and 4 use a weighting factor to guide the branching decision. The performance of rules 1,3,4 is not very sensitive to the value of the weighting parameter W. $W = 0.5$ appears to be a good choice over a range of problem instances, so in our experiments we use this value.

A small example illustrates why the use of different branching strategies is important. We consider the problem scr15, and implement our B&B algorithm using branching strategies A and B, as described in the previous section. The results are given in Table 3.4. In the table the entries in the "Fathom" column give the fraction of the nodes at each level that were fathomed. For the remaining unfathomed nodes, the entries in the "Elim." column give the fraction of the potential child nodes that were eliminated. For example, using strategy A, 24% of the nodes on level 3 were fathomed, and 18% of the potential children of the unfathomed nodes were eliminated. Note that for both strategies the number of level 1 nodes is 9, rather than 15, due to symmetry in the distance matrix for the problem. Using rule 4 for the top few levels of the tree in Strategy B incurs extra time at those levels (particularly level 1) compared to Strategy A, but pays off handsomely in greatly reducing the size of the tree. Note that the total number of nodes required using Strategy A is higher by a factor of over 56, and the total time is higher by a factor of about 28. In our implementation subproblems of size $n = 3$ are solved by enumeration; therefore there are no nodes below level 12 in this example.

The branching rules presented in Section 3.5 can be divided into two classes –

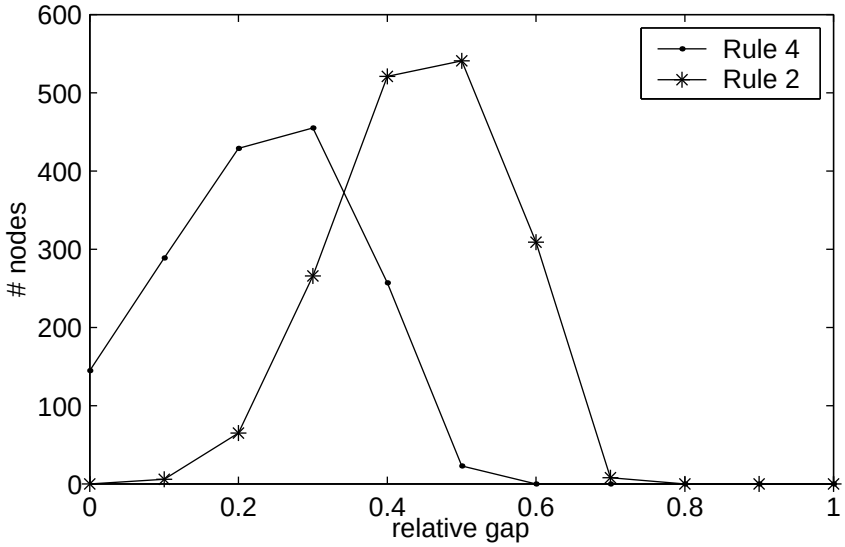
Table 3.4: Comparison of branching strategies on scr15

	Branching Strategy A				Branching Strategy B			
Level	Nodes	Fathom	Elim.	Time (s)	Nodes	Fathom	Elim.	Time (s)
0	1	0.00	0.00	0.03	1	0.00	0.00	0.92
1	9	0.00	0.00	0.21	9	0.00	0.17	9.51
2	108	0.06	0.08	1.82	90	0.07	0.60	1.24
3	1,181	0.24	0.18	13.94	424	0.52	0.63	3.24
4	8,747	0.49	0.32	67.61	907	0.73	0.48	4.12
5	33,422	0.65	0.42	170.85	1388	0.86	0.67	4.08
6	67,089	0.78	0.54	236.96	667	0.89	0.70	1.61
7	62,360	0.86	0.59	150.90	203	0.96	0.77	0.28
8	29,127	0.90	0.61	53.90	15	0.87	0.79	0.03
9	7,830	0.97	0.68	8.90	3	0.67	0.83	0.01
10	468	1.00	0.70	0.43	1	0.00	0.80	0.00
11	3	0.67	0.75	0.00	1	0.00	0.75	0.00
12	1			0.00	1			0.00
Total	210,346			705.55	3,710			25.04

Table 3.5: Comparison of Rules 2 and 4

	Rule 2		Rule 4	
Problem	Time	Nodes	Time	Nodes
nug12	5.57	1807	10.95	141
nug15	63.17	16064	119.61	968
nug16b	106.59	25376	311.34	2190
nug18	2067.78	387434	10342.30	49426
had16	43.57	7846	306.42	2410
had18	475.08	86519	3631.95	21589

Figure 3.14: Relative gaps of level 3 problems for nug20



those that perform additional bound computations (Rules 3 and 4) and those that do not (Rules 1 and 2). To examine the effect of these additional computations we solved a set of QAPLIB problems twice, using Rule 2 and Rule 4 at all nodes. As shown by Figure 3.5, Rule 4 is effective in reducing the size of the tree, however the extra time incurred by using Rule 4 at all nodes is too great. To get better performance a combination of rules should be used – expensive rules at the top of the tree, where there are fewer nodes, and less expensive rules for the bulk of the nodes in the middle of the tree.

Figure 3.14 measures the relative gaps of all level 3 subproblems for nug20 using two different branching strategies. The first strategy uses Rule 4 for all nodes up to and including level 2, and the second strategy uses Rule 2 throughout. The

Table 3.6: Comparison of Rules 1 and 2

	Rule 1		Rule 2	
Problem	Time	Nodes	Time	Nodes
nug12	5.68	1833	5.57	1807
nug15	63.29	16465	63.17	16064
nug16b	135.55	32890	106.59	25376
nug18	2096.01	401373	2067.78	387434
had16	35.33	7121	43.57	7846
had18	454.60	91536	475.08	86519

relative gaps for both rules roughly fit a normal distribution, the difference being that the use of the more expensive Rule 4 has shifted the center of the distribution well to the left. The use of Rule 4 results in fewer nodes at level 3, and those that remain are significantly easier – evidence that using more accurate information to make branching decisions pays off.

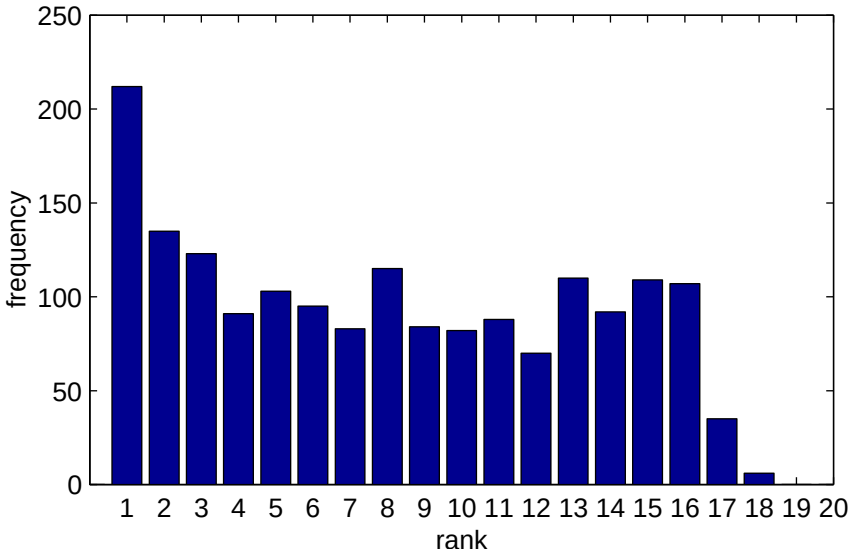
Comparing Rules 1 and 2 on small QAPLIB instances (Figure 3.6), we see that in most cases the rules perform similarly. At the top of the tree, it is likely that few children will be eliminated and in these cases Rules 1 and 2 often make the same branching decisions. As the problem size increases, however, Rule 2 becomes a slightly better choice.

Similarly, we also compare the performance of Rules 3 and 4 by solving the same set of QAPLIB problems. To obtain the results in Figure 3.7, for the top three levels of the tree either Rule 3 or 4 is used, and Rule 2 thereafter. The performance of Rule 4 is slightly better, both in nodes and execution time. The computational

Table 3.7: Comparison of Rules 3 and 4

	Rule 3		Rule 4	
Problem	Time	Nodes	Time	Nodes
nug12	10.38	148	10.95	141
nug15	113.28	1024	119.61	968
nug16b	303.45	2868	301.34	2190
nug18	11342.11	51528	10342.30	49426
had16	294.25	2824	306.42	2410
had18	3535.27	29362	3631.95	21589

Figure 3.15: Ranks of rows/columns selected for nug20



results of Chapter 5 are obtained using a combination of Rules 2 and 4 to construct a complete branching strategy.

The cost of Rules 3 and 4 is controlled in part by the NBEST parameter. Prospective bounds are computed for a fixed facility (or location) only if the facility

is determined to be one of the NBEST most promising, as determined by the row (column) sum of U. Figure 3.15 records the frequency with which rows or columns with a given rank were chosen during the solution of nug20, when prospective bounds were computed for all possible subproblems at the first two levels of the tree. The row/column with the best row/column sum of U ended up being the final branching choice over 200 times. Although the most highly ranked rows are chosen most frequently, the overall distribution is much flatter than one might expect. From this data, we draw the conclusion that computing prospective bounds as in Rules 3 and 4 will often result in different branching choices than Rules 1 and 2, and that keeping the NBEST parameter close to n at the top of the tree is potentially worthwhile.

The Gap and Depth parameters determine the branching rule used at each node of the tree. For smaller problem instances, the Gap parameter is not really necessary, and good results can be obtained simply by using different branching rules at different depths of the tree. For larger problems, selecting a strategy solely based on the depth of a problem becomes unwieldy. Table 3.10 compares the solution of nug24 using two different strategies, first using the parameters in Table 3.8, and secondly using the parameters in Table 3.9. Notice that when using a branching rule based solely on depth, the choice is either to spend a considerable amount of time at level 4 using an expensive rule, or much less using a cheaper rule, whereas when the gap is also considered, the amount of time at each level changes more gradually.

Table 3.8: Branching Strategy C

Gap	Depth	Rule	NFW1	NFW2	NFW3	NBEST	NUPDATE
0	2	4	150	150	50	20	30
0	4	3	100	100	25	10	30
0	50	2	75	50	–	–	30

Table 3.9: Branching Strategy D

Gap	Depth	Rule	NFW1	NFW2	NFW3	NBEST	NUPDATE
0.42	3	4	150	150	100	30	30
0.32	4	4	150	150	50	30	30
0.15	5	4	150	100	25	5	30
0.09	8	2	150	100	–	–	30
0.04	9	2	100	75	–	–	30
0	50	2	75	50	–	–	30

Table 3.10: Effects of gap-based branching on nug24

	Branching Strategy C		Branching Strategy D	
Level	Nodes	Time	Nodes	Time
0	1	2.67	1	5.25
1	6	60.65	6	100.09
2	138	1158.96	138	1918.46
3	2679	7719.55	2805	14606.94
4	37553	81116.99	42489	41108.37
5	267012	8309.16	373688	74296.30
6	2249827	33330.29	1627203	24186.75
7	6011981	65238.72	3946341	38624.77
8	9062304	74108.93	5286344	39687.33
9	7997331	50329.95	4785547	25755.83
10	4329027	21297.09	2817310	10656.64
11	1489263	5843.68	1100831	3243.20
12	349857	1100.38	256200	612.03
13	58551	155.65	42000	81.88
14	8489	19.04	5817	9.51
15	1183	2.38	879	1.16
16	180	0.27	151	0.20
17	44	0.07	32	0.05
18	11	0.01	10	0.03
19	1	0.00	2	0.01
20	1	0.00	1	0.00
21	1	0.00	1	0.00
Total	31865440	349794.00	20287796	274894.72

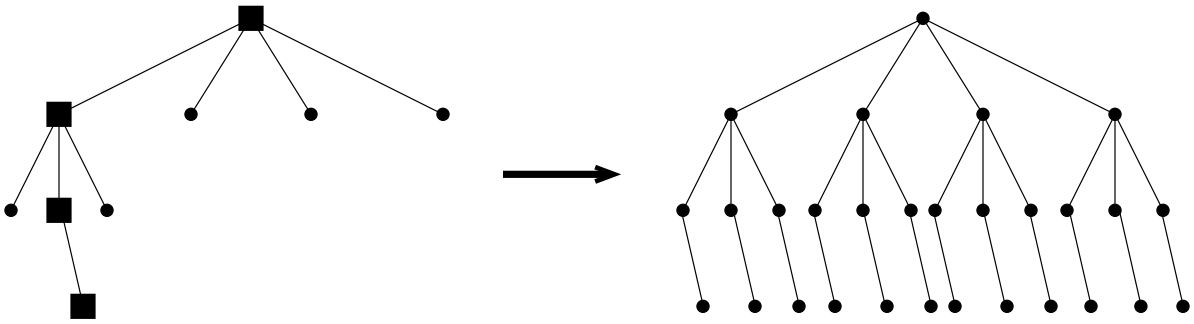
CHAPTER 4
ESTIMATING THE PERFORMANCE OF BRANCH-AND-BOUND ALGORITHMS

4.1 Introduction

In this chapter we consider a simple estimation procedure for branch-and-bound algorithms that allows us to adapt the search strategies of our algorithm to solve larger problems. The behavior of branch-and-bound algorithms, and backtracking programs in general is notoriously difficult to predict. Different problem instances of the same size may respond quite differently to the same algorithmic strategy. Even worse, small changes in the various parameters of an algorithm such as ours may also dramatically affect performance, as in Figure 3.4 of Chapter 3. Combining these facts with the fact that solving the largest problems may take weeks or years of sequential CPU time, performance estimation is essential. Even rough estimates of execution time and other statistics allow us to determine which problems can be solved. More accurate estimates provide information that can be used to find the best strategy to solve a particular problem.

In [53] Knuth presented a simple estimation procedure for backtracking programs. The procedure obtains estimates by making random walks down the search tree the backtracking procedure generates, see Figure 4.1. The procedure can pro-

Figure 4.1: Knuth's Estimation Procedure



duce estimates of the size of the tree, CPU time, and any other measurable statistic concerning the performance of the backtracking program. It is proved in [53] that the expected value of the estimates is equal to the actual value being estimated, and therefore a sample average over many trials provides an unbiased estimate. Knuth points out several potential pitfalls in the procedure, however he found that the procedure provides good performance estimates on a variety of different backtracking applications. Indeed, the procedure was used by Brüngger et al. to estimate the performance of their branch-and-bound algorithm for QAP [13, 14].

The estimates produced by the procedure on our branch-and-bound algorithm become progressively less accurate when applied to larger instances. In particular the deeper levels of the branch-and-bound tree are often left unexplored, even though an increasingly greater proportion of the nodes are at deeper levels of the tree. As a result, it is necessary to modify the basic procedure so that deeper parts of the tree

are explored more thoroughly. This is accomplished by biasing the random walks the estimator takes to select more difficult subproblems more frequently.

The basic estimation procedure of Knuth is presented in Section 4.2. Next, we describe the importance sampling technique used to obtain more accurate estimates, and we show how the technique is applied to our branch-and-bound algorithm for QAP. Finally, some numerical results are presented, including a case study of how the estimator can be used to tune the parameters of our branch-and-bound algorithm.

4.2 Knuth's Estimation Procedure

Knuth's procedure estimates the performance of backtracking algorithms. Since we are interested in estimating branch-and-bound algorithms, the discussion is focused on this particular application. A simplified version of the branch-and-bound algorithm presented in Chapter 3 is given in Figure 4.2, consisting of selection, bounding, and branching phases.

Knuth's procedure produces estimates by making random walks down the tree. A path is extended by choosing only *one* of the possible child subproblems $C_1 \ldots C_k$. Estimates are generated by assuming that nodes at the same level as a given node S have the same properties as S. For example, all nodes at the same level as S are assumed to have the same number of children. Each random path, or "dive" produces a new estimate, and the individual estimates are averaged to obtain a final estimate.

More formally, suppose that the task is to estimate the value of some function

Figure 4.2: A simplified branch-and-bound algorithm

```
v = BRANCH-AND-BOUND(P)
  1. Q = {P}
  2. WHILE Q ≠ {}:
        S = next_problem(Q)
        z = lower_bound(S)
        IF z ≤ v
            {C_1 ... C_k} = subdivide(S)
            Q = Q ∪ {C_1 ... C_k}
```

f over all nodes in the search tree, that is

$$\text{cost}(T) = \sum_{S \in T} f(S), \tag{4.1}$$

where T denotes the set of all nodes in the search tree. For example, if $f(S)$ measures the time to process node S, then cost(T) is the total execution time of the branch-and-bound algorithm. If $f(S) = 1$, then cost(T) is the number of nodes in the tree. Figure 4.3 is the basic estimation procedure of Knuth applied to a branch-and-bound algorithm. The procedure calculates as a result c, an estimate of (4.1). The quantity d_i can be interpreted as the estimated number of nodes at level i of the tree. The procedure $\mathsf{random}(1, k)$ selects a random integer between 1 and k with uniform probability, and is used to select a single subproblem to extend the path generated by the algorithm.

Theorem 4.2.1 *[Theorem 1, [53]] The expected value of c as computed in Figure 4.3*

Figure 4.3: An estimation procedure for branch-and-bound

$c =$ ESTIMATE(P)

1. $Q = \{P\}, d_0 \leftarrow 1, c_0 \leftarrow 0, i = 0.$
2. WHILE $Q \neq \{\}$:

   $S =$ next_problem(Q)

   $z =$ lower_bound(S)

   IF $z \leq v$

   $\{C_1 \ldots C_k\} =$ subdivide(S)

   $j =$ random$(1, k)$

   $Q = Q \cup \{C_j\}$

   $c_{i+1} \leftarrow c_i + f(S)d_i,$

   $d_{i+1} \leftarrow kd_i$

   $i = i + 1$
3. RETURN c_i

is $\text{cost}(T)$ *as defined by* (4.1).

Proof: Observe that for any subproblem $S \in T$ at level i of the tree the term

$$d_1 d_2 \dots d_i f(S) \tag{4.2}$$

occurs in the computation of c with probability $1/d_1 d_2 \dots d_i$ since this is the chance the algorithm will consider the subproblem S. □

4.3 Importance Sampling

The basic estimation technique of the previous section is quite effective for many problems. Brüngger et al. used the procedure to estimate branch-and-bound for QAP in [13], and reported that they were able to obtain reasonably accurate estimates. For small to medium-sized QAPs we were also able to obtain accurate estimates of the running time of our implementation. In Table 4.1 we compare estimated vs. actual performance on the nug20 QAPLIB problem, producing an estimate based on 10,000 random dives in the tree. Although the estimator produced no dives deeper than level 8 of the tree, the estimate of the total number of dives in the tree has an error of only 2.2%.

For larger problems, the chance that a dive will reach deeper levels of the tree becomes quite small. As a consequence, deeper levels of the tree may remain unexplored, and as stated by Knuth, "there is clearly a danger that our estimates will almost always be low, except for rare occasions when they will be much too high." When we began to solve larger problems ($n \geq 25$) we observed exactly this type of

Table 4.1: Actual vs. estimated performance on nug20

	Actual		Estimated	
Level	Nodes	Time	Nodes	Time
0	1	3.20	1	3.24
1	6	43.18	6	43.22
2	97	219.89	97	217.84
3	1591	601.91	1598	612.46
4	18521	776.27	18763	863.05
5	102674	921.26	106975	944.13
6	222900	1208.09	245746	1459.19
7	221873	795.82	270000	924.53
8	124407	317.92	94878	287.22
9	47930	97.81	0	0.00
10	11721	20.85	0	0.00
11	2509	3.67	0	0.00
12	450	0.65	0	0.00
13	73	0.06	0	0.00
14	5	0.00	0	0.00
15	3	0.00	0	0.00
16	1	0.00	0	0.00
17	1	0.00	0	0.00
Total	754763	5010.58	738066	5354.89

Table 4.2: Actual vs. estimated performance on nug25

	Actual		Estimated	
Level	Nodes	Time	Nodes	Time
0	1	7.61	1	6.57
1	6	102.53	6	92.59
2	94	2052.28	94	1383.47
3	1862	18961.55	1851	13054.79
4	34363	58878.66	33510	47080.76
5	429444	187471.00	410903	129255.54
6	2884391	62903.08	2700124	46146.53
7	10080284	147214.89	9382946	107607.90
8	17167548	172055.93	17050185	137610.51
9	19719449	135290.46	14894555	44413.69
10	18097486	95185.75	0	0.00
11	9772308	41428.72	0	0.00
12	4162521	14205.36	0	0.00
13	1468724	4049.11	0	0.00
14	454263	1023.28	0	0.00
15	125167	225.34	0	0.00
16	30929	45.28	0	0.00
17	7156	8.49	0	0.00
18	1584	1.53	0	0.00
19	368	0.27	0	0.00
20	86	0.05	0	0.00
21	20	0.01	0	0.00
22	5	0.00	0	0.00
Total	84438059	941111.00	44474177	526652.00

behavior. The performance of the estimation procedure on nug25, as shown in Table 4.2, is representative of the behavior of the estimator for larger problems. Out of 10,000 dives none were deeper than level 10 of the tree and the error in both the time and node estimates is nearly 50%.

We introduce bias into the sampling to obtain more deep dives. The technique, called **importance sampling**, increases the chance that nodes leading to long paths are selected. The idea of using importance sampling in the estimation procedure was suggested by Knuth in [53]. The uniform sampling procedure $\mathsf{random}(1, k)$ is replaced with a more general procedure that selects a subproblem i with a given probability p_i, and the update $d_{i+1} \leftarrow kd_i$ is replaced with $d_{i+1} \leftarrow d_i/p_i$. As long as the probabilities p_i are nonzero, it can be proved that the expected value of c remains $\text{cost}(T)$, and therefore the probabilities can be chosen with problem-specific information in mind to obtain better estimates. In our application deeper dives are desired, and the estimated lower bounds of the child subproblems can be used to generate the probabilities. A high lower bound indicates a path that is likely to soon end in a fathomed node, so to obtain longer paths nodes with low bounds should receive higher probabilities. An estimated lower bound is obtained by combining the lower bound of a parent node with dual information, or by prospectively computing a lower bound for branching purposes, as described in Chapter 3. The relative gap $g(z)$ defined by (3.4) measures the difficulty of a subproblem using the lower bound of a subproblem z along with the incumbent value and root lower bound z_0. Letting g_i denote the relative gap of

the ith child subproblem, we define sampling probabilities as follows:

$$p_i = \frac{g_i^q}{\sum_j g_j^q}, \tag{4.3}$$

where the exponent q is a parameter to be chosen. If $q = 0$ the sampling procedure reduces to the original estimation procedure. If $q = 1$, the probability a node is chosen is proportional to the gap. The larger the exponent q, the more heavily weighted are nodes with high gaps (and low bounds). In the next section an appropriate choice for q is determined empirically.

A second modification to the basic procedure improves both the accuracy of the estimates and the speed with which they are obtained. Recall from Chapter 3 that our branch-and-bound algorithm uses logic that selects an appropriate branching strategy that depends on the estimated difficulty of a node. In particular, on harder problems substantial time is spent making branching decisions at nodes high in the tree. Therefore a typical dive will consist of several nodes that take a great deal of time, followed by the remaining nodes that are relatively inexpensive. Since there are few nodes at the upper levels of the tree, and since many dives (typically 10,000) are required to generate an accurate estimate, many dives would repeat the same computations at the upper levels. To avoid this duplication of effort we first run the algorithm in "breadth-first mode" for the top NBFS levels. That is, a list of all level NBFS nodes is obtained, and all dives start from a randomly selected node at level NBFS, chosen according to the rule (4.3). If NBFS is too large, however, there are an enormous number of nodes to be saved and the storage requirements become

prohibitive; NBFS=3 is usually a good choice for larger problems.

4.4 Results

The branch-and-bound tree estimator presented in the previous section obtains more accurate estimates by using importance sampling to obtain a greater number of long paths. To measure the importance of a node we calculate the relative gap at the node and raise it to the power q. Increasing q gives more weight to nodes with poorer lower bounds, which tends to lead to longer paths. To compare the effects of increasing q we performed 10,000 dives of the estimator on the nug25 problem using $q = 0$ and $q = 2$. Figure 4.4 shows that indeed more deep dives are obtained when q is increased. For $q = 0$, the average depth was 5.25, and the longest dive was to level 9. For $q = 2$, the average depth was 6.62, and the longest dive was to level 12. Figure 4.5 plots the estimated and actual number of nodes for nug25 on a logarithmic scale, showing that the shape of the tree is estimated accurately through level 10, past the peak number of nodes.

Table 4.3 compares actual and estimated nodes and execution times for a range of small to medium-sized QAPLIB problems. For all estimates, 10,000 dives were performed, and $q = 1.5$. The NBFS parameter was set to level 3. For these instances the estimates are generally within 10% of the actual values.

The estimation procedure can be used to estimate statistics besides nodes and time. By estimating other problem statistics we can use the estimator to tune the parameters of our branch-and-bound algorithm for a specific problem. As an

Table 4.3: Actual vs. estimated performance
on QAPLIB problems

	Actual		Estimated	
Problem	Nodes	Time	Nodes	Time
nug12	608	4.23	537	4.05
nug15	5277	23.97	5773	25.51
nug16b	6268	25.17	6580	26.50
nug18	267233	1173.97	266302	1119.46
nug20	754763	5010.58	738066	5354.89
nug21	1216444	8818.46	1137360	7903.38
nug22	712597	5360.95	755130	5645.64

Figure 4.4: Distribution of dives for nug25 using $q = 0, q = 2$

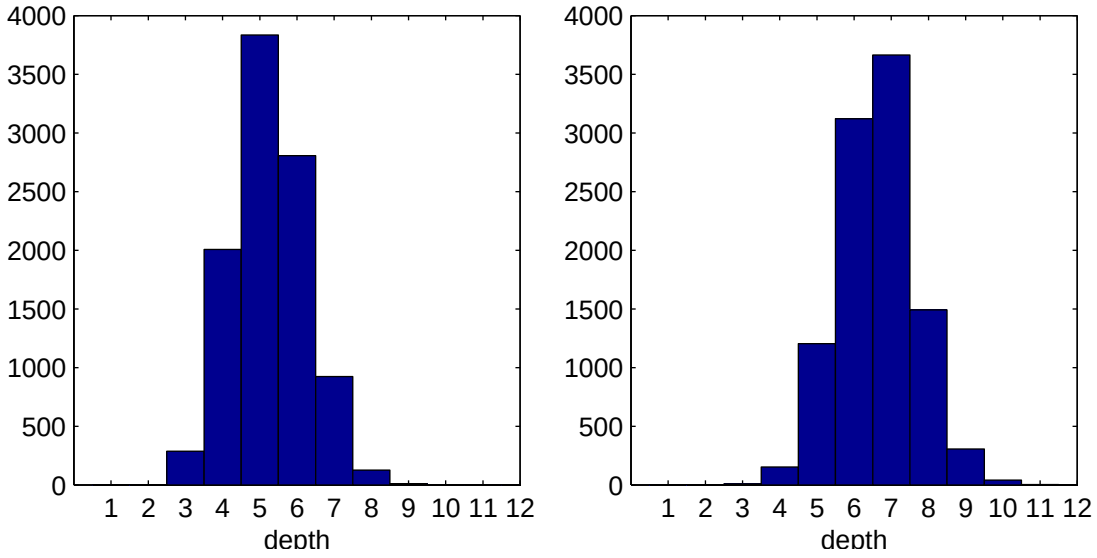
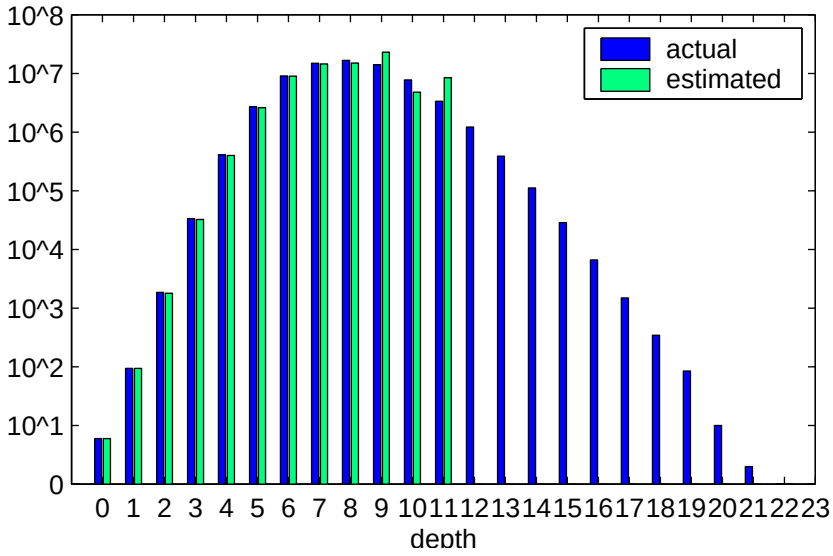


Figure 4.5: Actual and estimated number of nodes on nug25, $q = 2$



example, we describe how we used estimates to choose a good set of parameters for the previously unsolved nug30 problem. Using the parameters of Figure 4.4, we obtained the estimated statistics (using $q = 2$ and 10,000 dives) shown in Figures 4.5 - 4.7.

Table 4.5 provides information about each level of the tree. At each level, the number of nodes, CPU time, percentage of nodes fathomed (Fathom) and fraction of children eliminated during branching (Elim.) are estimated. Table 4.6 gives estimates of which branching strategies are used at each level of the tree. For example, at level 3 of the tree, it is estimated that 79% of the nodes will be processed using the first strategy. Lastly, Table 4.7 provides information about the relative gap, an estimate of the difficulty of subproblems at each level of the tree. Of course, deeper nodes are

generally easier than those higher in the tree.

Tables 4.5 - 4.7 provide considerable of information about the expected behavior of the algorithm. To show how this information is interpreted to derive an improved set of parameters, statistics of particular interest are boldfaced. For example, notice that the amount of time spent at level 7 is approximately times that spent at level 8, even though there are far fewer nodes at level 7. Looking at Table 4.6, one sees the explanation: the third strategy, using the computationally expensive Rule 4, is used on almost 13% of the nodes at level 7, but it is not used at all at level 8. In fact, over one third of the estimated time is spent using the third strategy, which seems disproportionately high. The estimate indicates that it would be better to raise the gap setting for the third strategy, so that it is used less frequently. Looking at the relative gap information in Table 4.7 helps to determine how to change the relative gap parameter in the branching strategy. Using the mean and standard deviation estimates, and assuming approximately a normal distribution for the upper tail of the relative gap distribution, if the gap parameter is set to .12 then the number of nodes using strategy 3 at level 7 should be cut roughly in half.

We next produced an estimate based on the parameters in Table 4.8. The gap parameters for the third and fourth strategies were changed, and the NBEST parameter for the second strategy was increased in the hope that better branching decisions would be made high in the tree. The number of FW iterations were increased for the last three strategies as well, in an effort to decrease the number of nodes at

Table 4.4: Nug30 branching strategy 1

Gap	Depth	Rule	NFW1	NFW2	NFW3	NBEST	NUPDATE
0.34	3	4	150	150	100	30	30
0.22	6	4	150	150	50	15	30
0.1	7	4	150	100	25	5	30
0.05	8	2	100	75	–	–	30
0.03	10	2	75	50	–	–	30
0	50	2	50	25	–	–	30

Table 4.5: Nug30 estimate 1: Overall statistics

Level	Nodes	Fathom	Elim.	Time
0	1	0.0000	0.0000	19.55
1	9	0.0000	0.0000	423.26
2	225	0.0000	0.0000	10215.78
3	6128	0.0000	0.0385	212684.07
4	159943	0.0099	0.1843	1576238.67
5	3358292	0.1416	0.4003	12912491.54
6	43770858	0.3385	0.5673	48396502.64
7	293151653	0.5494	0.6591	**87971649.10**
8	1048235387	0.7045	0.6380	21120277.85
9	2416570568	0.7916	0.5872	34751544.94
10	3650580684	0.7820	0.6747	49571802.42
11	4780270305	0.8323	0.7620	47741802.68
12	3284837699	1.0000	N/A	30259117.90
Total	15520941757			3.34525×10^8

Table 4.6: Nug30 estimate 1: Breakdown of strategies used

Level	Frequency strategy used						Nodes
0	1.0000	0.0000	0.0000	0.0000	0.0000	0.0000	1
1	1.0000	0.0000	0.0000	0.0000	0.0000	0.0000	9
2	1.0000	0.0000	0.0000	0.0000	0.0000	0.0000	225
3	0.7936	0.1674	0.0352	0.0034	0.0000	0.0000	6128
4	0.0000	0.6018	0.2816	0.0744	0.0190	0.0233	159943
5	0.0000	0.2089	0.4045	0.2048	0.0816	0.1002	3358292
6	0.0000	0.0386	0.2618	0.2958	0.1473	0.2565	43770858
7	0.0000	0.0000	**0.1298**	0.2811	0.2159	0.3732	**293151653**
8	0.0000	0.0000	0.0000	0.2761	0.2024	0.5215	1048235387
9	0.0000	0.0000	0.0000	0.0000	0.5302	0.4698	2416570568
10	0.0000	0.0000	0.0000	0.0000	0.6479	0.3521	3650580684
11	0.0000	0.0000	0.0000	0.0000	0.0000	1.0000	4780270305
12	0.0000	0.0000	0.0000	0.0000	0.0000	1.0000	3284837699
nodes	5.10e3	2.49e6	5.09e7	3.85e8	3.93e9	1.12e10	
time	2.05e5	2.88e7	**1.14e8**	**1.37e7**	7.03e7	1.07e8	

Table 4.7: Nug30 estimate 1:
Relative gap information

Level	Mean	Std.Dev.
0	1.000000	0.000000
1	0.834814	0.020960
2	0.626952	0.080834
3	0.431031	0.109989
4	0.251731	0.118108
5	0.142286	0.094283
6	0.079428	0.063888
7	**0.052725**	**0.044555**
8	0.038910	0.036427
9	0.046048	0.042371
10	0.049190	0.033078
11	0.030287	0.018243
12	0.013146	0.003150

Table 4.8: Nug30 branching strategy 2

Gap	Depth	Rule	NFW1	NFW2	NFW3	NBEST	NUPDATE
0.34	3	4	150	150	100	30	30
0.22	6	4	150	150	50	30	30
0.12	7	4	150	100	25	5	30
0.06	8	2	150	100	–	–	30
0.03	10	2	100	75	–	–	30
0	50	2	75	50	–	–	30

Table 4.9: Nug30 estimate 2: Overall statistics

Level	Nodes	Fathom	Elim.	Time
0	1	0.0000	0.0000	19.32
1	9	0.0000	0.0000	414.78
2	225	0.0000	0.0000	10075.13
3	6147	0.0000	0.0369	203996.00
4	155942	0.0099	0.1751	1519100.63
5	3207917	0.1401	0.3848	9996550.46
6	43459091	0.4155	0.5374	33074698.16
7	291949859	0.6184	0.6281	52876831.61
8	898342396	0.7277	0.6409	19044520.83
9	1872380648	0.8093	0.6413	25696088.29
10	2483496566	0.8282	0.6821	27121079.49
11	2099400200	0.9123	0.7147	21081742.17
12	**6221220189**	1.0000	N/A	**508597.56**
Total	13913619194			1.91134×10^8

Table 4.10: Nug30 estimate 2: Breakdown of strategies used

Level	Frequency strategy used						Nodes
0	1.0000	0.0000	0.0000	0.0000	0.0000	0.0000	1
1	1.0000	0.0000	0.0000	0.0000	0.0000	0.0000	9
2	1.0000	0.0000	0.0000	0.0000	0.0000	0.0000	225
3	0.7804	0.1589	0.0534	0.0072	0.0000	0.0000	6147
4	0.0000	0.5388	0.3265	0.1038	0.0309	0.0000	155942
5	0.0000	0.1582	0.3851	0.2666	0.1214	0.0687	3207917
6	0.0000	0.0241	0.2066	0.3004	0.2265	0.2424	43459091
7	0.0000	0.0000	**0.0851**	0.2594	0.2969	0.3585	291949859
8	0.0000	0.0000	0.0000	0.2486	0.3224	0.4290	898342396
9	0.0000	0.0000	0.0000	0.0000	0.5513	0.4487	1872380648
10	0.0000	0.0000	0.0000	0.0000	0.5884	0.4116	2483496566
11	0.0000	0.0000	0.0000	0.0000	0.0000	1.0000	2099400200
12	0.0000	0.0000	0.0000	0.0000	0.0000	1.0000	6221220189
nodes	5.03e3	1.64e6	3.51e7	3.13e8	2.88e9	**1.07e10**	
time	1.94e5	2.03e7	6.89e7	1.26e7	4.77e7	**4.13e7**	

Table 4.11: Nug30 estimate 2:
Relative gap information

Level	Mean	Std.Dev.
0	1.000000	0.000000
1	0.831303	0.020894
2	0.623652	0.080865
3	0.426593	0.111252
4	0.254427	0.112663
5	0.145497	0.090607
6	0.080032	0.063287
7	0.054059	0.043766
8	**0.044216**	0.034855
9	0.044779	0.038112
10	0.043890	0.030593
11	0.038787	0.024393
12	0.011957	0.002500

the deeper levels of the tree.

The estimates in Tables 4.9 – 4.11 show that the estimated time using branching strategy 2 decreased by over 42%, compared with the estimate using branching strategy 1. Again we have highlighted a few of the most interesting statistics. Note that the number of nodes at level 12 is very high compared to previous levels. The actual peak of the node distribution is probably at level 10, or perhaps at level 11 as in our previous estimate. The node figure for level 12 is likely a gross overestimate produced by a single "rogue" dive. On the other hand, the time for level 12 is almost certainly underestimated, because all the nodes encountered by the estimator on level 12 were easy enough to fathom.

Looking at the last line of Table 4.10, we see that the time is more evenly distributed among the six branching strategies. The time for the last rule is certainly an underestimate, because there were no dives below level 12. However, since the peak number of nodes should be at level 10 or 11, and only the cheapest branching rule is in use past level 10, the additional time spent processing nodes deeper than level 12 is not likely to be significant. Finally, we note that since expensive strategies are used less frequently at level 7, the mean estimated gap at level 8 is somewhat higher than in the previous estimate. However, the mean relative gaps at lower levels of the tree remain similar to those in Estimate 1. In Table 4.10 it is estimated that the third rule will be used by about 8.5% of nodes at level 7, compared to the original estimate of about 13% in Table 4.6.

The extended example of nug30 shows that estimation of many different statistics allows for a more accurate assessment of the likely performance of the algorithm. The amount of "guesswork" needed to produce parameters that lead to an efficient solution is greatly reduced. However, even when many dives are performed and importance sampling is used, for large problems there are inevitably cases where an estimated statistic makes little sense. Information such as the overall shape of the branch-and-bound tree and the actual performance of the algorithm on smaller instances must be taken into account when interpreting the output of the estimator on large problems.

Even with an efficient branch-and-bound implementation and an estimation procedure to guide parameter selection, the time to solve increasingly larger QAP instances grows exponentially. The execution time of our implementation roughly triples each time n is increased by one – nug24 can be solved in well under a million seconds, but an estimated CPU time of approximately 2×10^8 seconds corresponds to over 6 years of sequential computation for nug30. It is clear that to solve larger instances additional computational resources must be brought to bear. The next chapter describes a grid computing system capable of harnessing hundreds of workstations in different geographical locations to efficiently solve large QAPs in parallel.

CHAPTER 5
COMPUTATIONAL RESULTS

5.1 Introduction

The nug30 QAP solved by our branch-and-bound implementation ranks as one of the most difficult discrete optimization problems ever solved. Its solution was made possible by the combination of an algorithm that yields the best sequential results with a distributed computing system that provides tremendous computing power over extended periods of time. In this chapter we first describe the grid computing system allowing us to parallelize our branch-and-bound algorithm. Next we report on the solution of nug30 as well as other QAPLIB problems, comparing the performance of our branch-and-bound implementation to others.

5.2 Parallel Implementation of Branch-and-Bound

5.2.1 Grid Computing

A **computational grid** is a collection of distributed, possibly heterogeneous resources which can be used together to perform computations. Computational grids (or metacomputers) are analogous to power grids in the sense that the provided resource is ubiquitous and that grid users need not know the source of the provided

resource. The concept of grid computing is becoming a reality now that large collections of fast workstations are being linked by increasingly faster networks.

Grid computing environments possess several advantages compared to traditional parallel computing environments. The components of the system are typically standard, inexpensive PCs and workstations that are already in use. By linking geographically distributed computational resources, resources can be utilized to their fullest since many more users can make use of them. Further, since the grid has no fixed physical makeup, new resources can be introduced, and old resources withdrawn, at any time without requiring upgrades or modifications to application software. Lastly, and perhaps most importantly for our application, potentially far more resources can be used to solve a particular problem, as there are no physical or architectural constraints on the resources.

Despite the above advantages, the dynamic nature of grid environments raises a new set of obstacles that are not as problematic in conventional parallel computing environments. Applications using grids are able to assume very little about their computing environment – resources may vary in location, latency, availability, and performance. By its very nature, a grid is a failure-prone environment, where machines may crash, or be removed from the grid at any moment. Even with advances in networking technologies, the available bandwidth between resources is much lower than that found in a traditional parallel computing environment, where a relatively small number of machines are connected by very high bandwidth connections. High

latency often exists between resources, so applications requiring a great deal of synchronicity are currently ill-suited for use in grid environments. Grid computing tools aim to overcome these difficulties and provide a reliable, fault-tolerant environment for applications.

5.2.2 MW: An Framework for Grid Computing

The Master-Worker (MW) system is an enabling tool for grid computing, created at Argonne National Laboratory and the University of Wisconsin. A detailed description of MW is found in [38, 39]. Our branch-and-bound algorithm uses the MW API to tap into the computational grid and perform tree searches in parallel. MW is suited to so-called master-worker parallel algorithms whereby a master machine distributes work to a number of worker machines, who report results back to the master. In a master-worker algorithm there is little or no communication between workers. Branch and bound algorithms fit perfectly into the master-worker framework – the master keeps track of unexplored nodes in the search tree and distributes them among the workers, who perform independent searches in the branch-and-bound tree.

MW is a software framework that provides an interface between the designer of a parallel application and the underlying communication and resource management facilities. Computational resources are managed by Condor [64, 63], a software package that handles the acquisition and release of resources on the grid. A unique feature of Condor is that the resources it manages are nondedicated, making use of computational resources that would otherwise be wasted. Communication between resources is

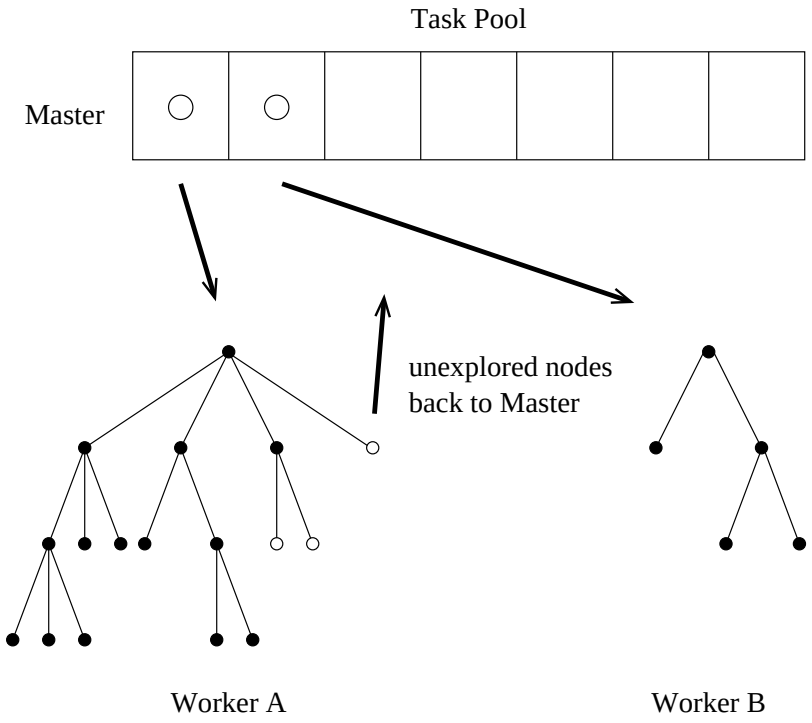
ultimately handled by message passing interfaces such as PVM or MPI. MW provides high-level abstractions that simplify communication between master and workers, allowing the application designer to think in terms of a high-level, application-specific computational task rather than individual messages.

In our discussion we will consider MW from the point of view of the branch-and-bound application. That is, we do not concentrate on how MW acquires and releases resources, or the underlying details of communication. Instead, we concern ourselves with how to best use MW to produce an efficient, stable parallel implementation of our branch-and-bound algorithm. We refer to [38] for a more complete description of MW, and to [64] for an introduction to Condor.

A user of MW implements a parallel algorithm by defining a computational task to be performed by workers, the actions the master takes to send tasks to workers, and the actions performed by workers upon receiving tasks. In the application at hand, a task corresponds to a single node in the branch-and-bound tree. The master selects nodes from its list of subproblems (or **task pool**) and sends them to workers, and the workers receive nodes and execute the sequential branch-and-bound algorithm on them for a given period of time, as shown in Figure 5.1. Workers explore some portion of the branch-and-bound tree (the filled-in circles in Figure 5.1) and return unexplored nodes to the master task pool.

The MW API provides a set of C++ interfaces that are instantiated by the application, among them `MWDriver`, `MWWorker`, and `MWTask`. Writing a parallel applica-

Figure 5.1: A Master-Worker algorithm for branch-and-bound



tion using MW amounts to implementing these classes. An `MWTask` in our application corresponds to a node in the branch-and-bound tree. The `MWDriver` and `MWWorker` communicate by sending and receiving `MWTask` objects. `MWTasks` also contain statistical information that can be used by the master to measure the performance of the branch-and-bound algorithm and parallel efficiency. In the presence of failures at the worker level, the master resends tasks as necessary.

5.2.3 MW Algorithmic Details

An advantage of MW is that it allows users to exploit the algorithmic characteristics of a specific application to produce an efficient parallel algorithm. In what follows we describe a few of the algorithmic details that do not exist in the sequential version of our branch-and-bound implementation.

5.2.3.1 Controlling Grainsize

An efficient parallel algorithm running on a computational grid tries to minimize the amount of communication and keeps all workers busy doing useful work in the presence of high latency and varying resource availability. The **grainsize** is a measure of the duration of each computational task. A large grainsize helps ensure that the master will not be overwhelmed by workers asking for more tasks. If the grainsize is too large, however, workers will infrequently return new tasks and other workers may be forced to wait to receive new tasks. In our application, the grainsize is controlled via a parameter $t_{\max}$. A worker executes the sequential version

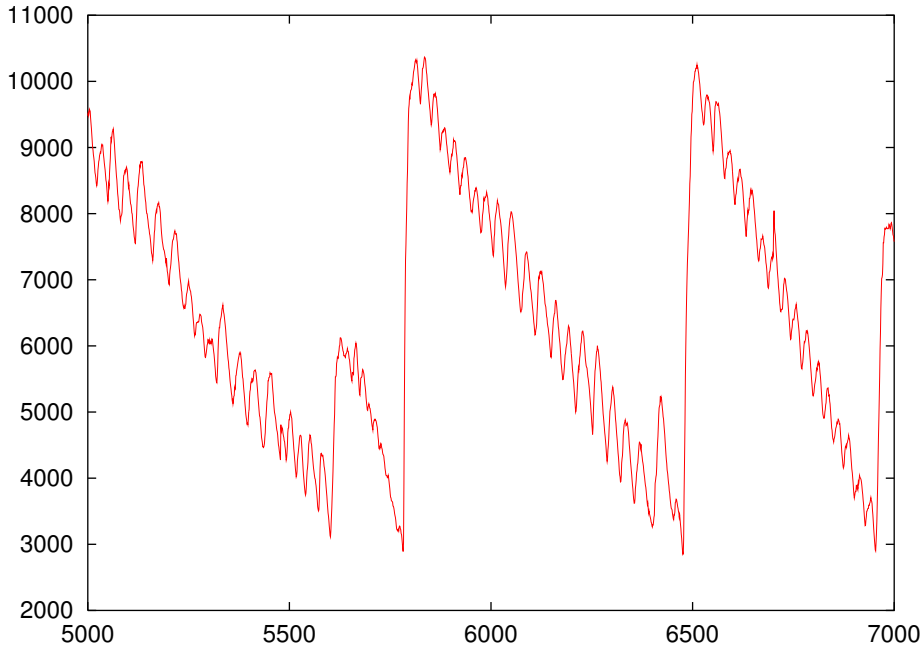
of branch-and-bound for $t_{\max}$ seconds, then sends its remaining nodes back to the master, who in turn distributes them to other workers. The master reschedules tasks that correspond to failed or unresponsive worker machines.

Communication costs are reduced when the workers return short lists of subproblems. Furthermore, workers should avoid returning subproblems that will fathom immediately – in such cases the time to send the task is far greater than the time to process it. The sequential branch-and-bound algorithm of Chapter 3 performs a depth-first search of the subtree, but a modification of depth-first search results in fewer "easy" subproblems being reported back to the master. After the $t_{\max}$ second time limit expires for a given worker, the worker enters a "finish up" phase. The worker is given a small period of time to select and process subproblems deeper than a specified depth $d_{\max}$ and/or relative gap greater than $g_{\min}$. It is hoped that many of these easier subproblems fathom and the overall size of the worker task pool is reduced. A series of short finish up phases are performed with progressively deeper $d_{\max}$ to weed out as many of the easier tasks as possible.

5.2.3.2 Managing the Master Task Pool

Keeping the master task pool small allows the master to conserve memory and manage its tasks as efficiently as possible. Normally the master's task pool is ordered by depth, ensuring that workers requesting tasks receive the deepest nodes available. However, when the task pool is small and there is a risk of workers having to wait for nodes the depth-first policy is abandoned. When the master task pool

Figure 5.2: Size of task pool using lazy best-first search



is small, workers are asked to report more frequently by lowering $t_{\max}$. Additionally, workers are given more difficult tasks in an effort to get them to return more tasks to repopulate the task pool. The master accomplishes this by switching to a best-first search when there are fewer than 3000 tasks in the task pool. That is, the tasks are ordered by their relative gaps, a measure of task difficulty. Figure 5.2 shows how the "lazy best-first" strategy keeps the size of the master task pool in a desirable range near the end of the solution of nug30.

5.2.3.3 Fault Tolerance

Because the `MWDriver` reschedules tasks when the processors running these tasks fail, applications running on top of MW are fault tolerant in the presence of all processor failures – except the master processor. In order to make computations fully reliable, MW offers features to logically checkpoint the state of the computation on the master process. The entire state of the master is saved every 15 minutes in our application. As a result, the master process can be reinitialized after a failure with the loss of at most 15 minutes of computation.

5.2.3.4 Normalized Performance Measurement

The heterogeneous and dynamic nature of a computational grid makes application performance difficult to assess. Consistent, reliable performance measurements are important because they allow for comparisons of different algorithmic techniques, and comparisons with other parallel implementations. Since the number and quality

Table 5.1: MWQAP parameters

Parameter	Value
checkpoint	15 minutes
$t_{\max}$	400 seconds
finish up phase 1	use $g_{\min}$ and $d_{\max}$ for 2*$t_{\max}$ seconds
finish up phase 2	drop $g_{\min}$ constraint,use $d_{\max}$ for $t_{\max}$ seconds
finish up phase 3	drop $g_{\min}$ constraint,use $d_{\max}$+1 for $t_{\max}$ seconds
lazy best-first	activated when task pool size is smaller than 3000

of resources vary between trials, statistics such as cumulative CPU time or wall-clock time have little value. Normalizing the length of each performed task by the performance of the corresponding worker allows for useful, experimentally repeatable performance measurement.

The different resources participating in a run have varying characteristics such as processor speed, amount of memory, etc. To tabulate a normalized set of statistics, workers joining the computation are asked to run a benchmark task. In our application the benchmark task is to run a small, specific portion of the branch-and-bound tree. When workers complete tasks they report the time it took to complete the benchmark task, enabling the master to take the worker's benchmark time into account when adding in the task statistics to the overall statistics. To determine how long a run would take on a given machine, it suffices to run the benchmark job on the machine and scale the times by the provided benchmark factor.

The parameters controlling grainsize, the length of the finish up phases, lazy best-first searching, and checkpointing were determined empirically via a series of

experiments on the QAP instances nug25, nug27 and nug28. The resulting parameters, given in Table 5.1, help to ensure high parallel efficiency over the length of the execution of MWQAP.

5.3 Computational Results on QAPLIB Problems

We now present results for QAPLIB problems solvable by our algorithm. The most difficult instances were solved in parallel using MWQAP, the rest sequentially using a HPC3000 workstation whose characteristics are given in Appendix A.4. The normalized execution times and number of nodes in the branch-and-bound tree are given in Table 5.2. Previously unsolved instances are boldfaced.

The execution time of the algorithm clearly increases exponentially with the problem size, although certain sets of problems are more efficiently solved than others. Unsurprisingly, the best results are obtained for problems for which the root QPB bound is tightest, for example the "hadxx" and "nugxx" problems. The "taixxa" problems are particularly challenging due to poor root QPB bounds and lack of problem symmetry.

The branching strategies used for the QAPLIB problems are composed of six basic rules, shown in Table 5.3. The full algorithm parameters for each problem are a combination of the six rules, along with depth and gap parameters that control when each rule is used. A complete specification of the branching strategies used is given in Table 5.4. The larger instances ($n \geq 20$) are far more sensitive to changes

Table 5.2: Branch-and-bound performance on QAPLIB problems

Problem	Nodes	Time	SEQ/MW
had12	265	< 1	S
had14	791	3	S
had16	7249	34	S
had18	84172	391	S
had20	105869	614	S
kra30b	5136036412	84089376	MW
nug12	608	4	S
nug14	6413	34	S
nug15	5277	24	S
nug16b	6268	25	S
nug17	82716	350	S
nug18	267233	1174	S
nug20	701026	4861	S
nug21	1216534	7936	S
nug22	710665	4903	S
nug24	20287796	274896	MW
nug25	70981238	720162	MW
nug27	399258758	7037780	MW
nug28	2547412140	57432518	MW
nug30	11892208412	218366867	MW
rou12	2360	11	S
rou15	36918	415	S
rou20	53869874	420129	S
scr12	612	4	S
scr15	3904	23	S
scr20	3256229	30780	S
tai12a	399	1	S
tai15a	116782	823	S
tai17a	747890	5107	S
tai20a	50694572	364040	S

Table 5.3: Branching rules
for QAPLIB problems

	Rule	NFW1	NFW2	NFW3	NBEST	NUPDATE
I	4	150	150	100	30	30
II	4	150	150	50	30*	30
III	4	150	100	25	5	30
IV	2	150	100	–	–	30
V	2	100	75	–	–	30
VI	2	75	50	–	–	30

* 15 for nug27,nug28

in parameters than the smaller instances, for example in Chapter 4 we showed that a small change in branching strategies resulted in a 50% improvement in the estimated time to solve nug30.

In Table 5.5, 5.6 we give the nodes required, and normalized CPU time (in minutes) required by our algorithm on a smaller subset of the QAPLIB problems, as well as results for the following B&B algorithms:

- CP/BMCP: Brüngger, Marzetta, Clausen, and Perregaard, from [24, 14],
- HGH: Hahn, Grant and Hall, from [46],
- HHJGR: Hahn, Hightower, Johnson, Guignard-Spielberg, and Roucairol, from [43].

The times reported by the above were normalized to the speed of an HP-C3000 workstation.

Table 5.4: Branching strategies for QAPLIB problems (Depth/Gap)

	Branching Strategy					
Problem	I	II	III	IV	V	VI
had12	–	–	–	all	–	–
had14	–	–	–	all	–	–
had16	–	–	–	all	–	–
had18	–	–	–	all	–	–
had20	–	–	–	all	–	–
nug12	–	2/0.65	5/0.40	–	–	rest
nug14	–	2/0.65	5/0.40	–	–	rest
nug15	–	2/0.65	5/0.40	–	–	rest
nug16b	–	2/0.65	5/0.45	–	–	rest
nug17	–	2/0.65	4/0.44	–	8/0.15	rest
nug18	–	2/0.65	4/0.44	–	8/0.15	rest
nug20	1/0.60	3/0.50	5/0.30	–	8/0.07	rest
nug21	1/0.50	3/0.40	5/0.28	–	8/0.08	rest
nug22	1/0.46	3/0.30	5/0.25	–	8/0.07	rest
nug24	3/0.42	4/0.32	5/0.15	8/0.09	9/0.04	rest
nug25	3/0.42	5/0.32	5/0.18	7/0.09	8/0.04	rest
nug27	3/0.42	5/0.32	6/0.16	8/0.09	9/0.04	rest
nug28	3/0.34	5/0.24	6/0.12	8/0.09	10/0.01	rest
nug30	3/0.34	6/0.24	7/0.12	8/0.06	10/0.03	rest
rou12	–	–	1/1.00	–	–	rest
rou15	–	2/0.60	5/0.40	–	7/0.15	rest
rou20	1/0.60	3/0.55	5/0.25	8/0.10	10/0.05	rest
scr12	–	1/0.40	–	–	–	rest
scr15	–	2/0.40	–	–	–	rest
scr20	–	2/0.40	5/0.12	–	9/0.04	rest
tai12a	–	–	–	all	–	–
tai15a	–	2/0.45	4/0.30	–	6/0.12	rest
tai17a	3/0.70	4/0.45	5/0.30	–	8/0.12	rest
tai20a	2/0.70	3/0.50	4/0.42	6/0.10	8/0.05	rest

Table 5.5: Number of nodes to solve QAPLIB problems

Problem	MWQAP	CP/BMCP	HGH	HHJGR
had16	7,249	18,770,885	13,808	3,069
had18	84,172	761,452,218	197,487	53,224
had20	105,869	7,616,968,110		
nug16b	6,268	320,556		
nug18	267,233	114,948,381		202,021
nug20	701,026	360,148,026	724,289	239,449
nug21	1,216,534	3,631,929,368	3,192,565	
nug22	710,665	48,538,844,41	10,768,366	988,302
nug24	20,287,796			11,674,950
nug25	70,981,238			108,738,131
rou20	53,869,874	2,161,665,137	2,090,862	
tai17a	747,890	20,863,039		
tai20a	50,694,572	2,215,221,637		

Table 5.6: Equivalent CPU time (m) to solve QAPLIB problems

Problem	MWQAP	CP/BMCP	HGH	HHJGR
had16	0.6	15.8	2.8	0.4
had18	6.5	871.1	70.1	9.1
had20	19.2	50,992.4		
nug16b	0.4	0.3		
nug18	19.6	123.1		33.5
nug20	81.0	483.0	333.1	134.1
nug21	132.3	6,331.9	1,296.9	
nug22	81.7	113,336.1	3,775.9	1,663.4
nug24	4,581.6			6,437.7
nug25	12,002.7			47,490.2
rou20	7,002.1	3,787.7	2,062.6	
tai17a	85.1	23.6		
tai20a	6,067.3	12,131.8		

Figure 5.3: Equivalent CPU time (m) to solve nugxx problems

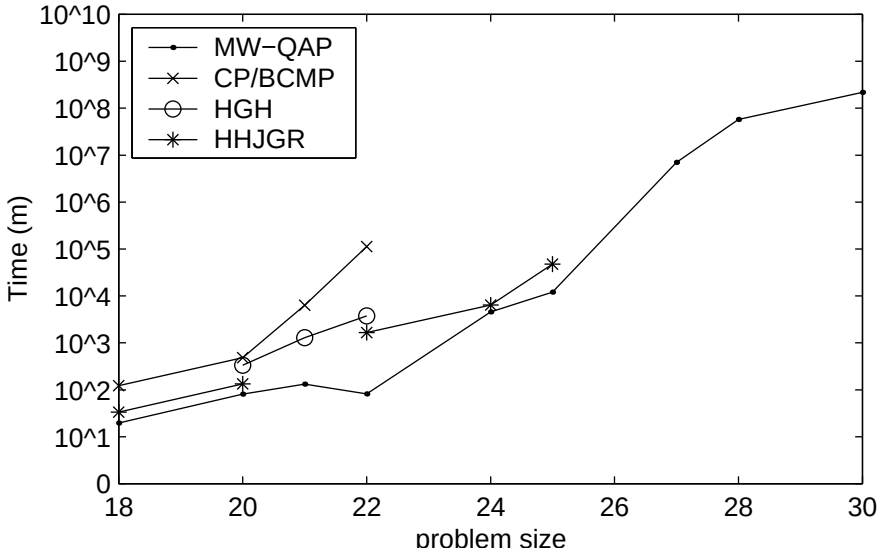


Table 5.6 indicates that the performance of our algorithm on larger problems is far superior to the GLB-based results of CP/BMCP, and is competitive with the state-of-the-art dual LP-based results of HHJGR. The number of nodes explored by the CP/BCMP algorithm is much greater since the weaker Gilmore-Lawler bound was used. Figure 5.3 shows how the execution times change for the Nugent problems as the problem size is increased. Clearly all the algorithms exhibit exponential time performance; the difference is the factor by which the execution time increases – the slope of the lines in Figure 5.3. The growth rates of MWQAP and HHJGR are similar, however the CP/BMCP approach based on GLB grows much more quickly.

5.4 Results Using MWQAP on Large Problems

5.4.1 Solution of the Nug30 QAP

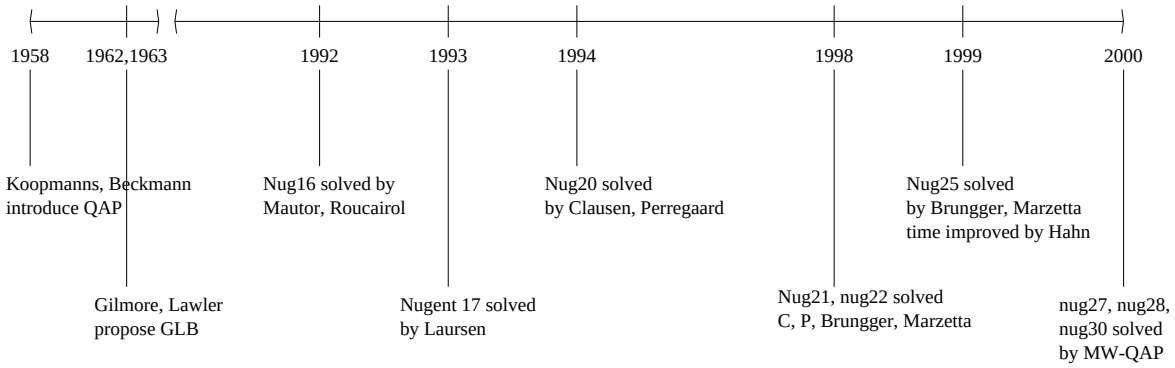
The Nugent QAP of size 30 (or simply nug30) is the largest of a set of QAPs posed by Nugent, Vollman and Ruml in 1968 [71]. The original set were instances of size 5,6,7,8,12,15,20 and 30. Instances of size 14,16,17,18,21,22,24,25,27,28 have been introduced over the years by removing facilities and locations from larger instances. The distances of these problems correspond to Manhattan distances on rectangular grids. The Nugent problems are the most commonly used set of benchmark QAPs, and the solution of the various instances have marked advances in both processor speeds and QAP solution methods. Peter Hahn relates the history of the solution of the Nugent problems in [42], and Figure 5.4 summarizes this progress. Until the solution of nug25 the best implementations used GLB as their lower bounding procedure. Nugent 25 was first solved using the dynamic programming lower bounding approach of [14]. Subsequently nug25 was solved more efficiently using the branch-and-bound implementation of Hahn, et al. [43].

The optimal solution to the nug30 QAP instance is:

$$14, 5, 28, 24, 1, 3, 16, 15, 10, 9, 21, 2, 4, 29, 25, 22, 13, 26, 17, 30, 6, 20, 19, 8, 18, 7, 27, 12, 11, 23$$

and the corresponding objective value is 6124. Three other solutions with the same solution value are equivalent via symmetry. A permutation with an objective value of 6124 was first obtained by Skorin-Kapov using tabu search, see [81]. In order to prove the optimality of this solution using MWQAP, 11,892,208,412 nodes of a branch and

Figure 5.4: History of solution of Nugent QAPs



bound tree were explored. Solving the associated node subproblems and computing the branching information required 574,254,156,532 Frank-Wolfe iterations.

On average, there were 653 machines participating in the nug30 solution process, with a maximum of 1009. One of the most remarkable features of the run was that almost 1 million linear assignment problems (LAPs) were solved each second during the course of the run. (One LAP must be solved for each Frank-Wolfe iteration). Table 5.7 shows a number of other interesting statistics about the nug30 run and the computational pool. The machine speeds have been normalized to an HP-C3000 workstation using the performance normalization methodology of Section 5.2.3.4. (Thus the "average" machine used in the nug30 computation was 56% as fast as an HP-C3000).

Machines from all over the world participated in the nug30 solution. Table 5.8 shows the composition of the computational pool. The 1024 SGI/Irix processors at

Table 5.7: Nug30 run statistics

Average number of available workers	652.7
Maximum number of available workers	1009
Running wall clock time (sec)	597,872
Total cpu time (sec)	346,640,860
Average machine speed	0.560
Minimum machine speed	0.045
Maximum machine speed	1.074
Equivalent CPU time (sec) on an HP-C3000	218,823,577
Parallel Efficiency	93%
Number of times a machine joined the computation	19,063

NCSA currently rank that supercomputer as the 51st fastest in the world, according to Top500.org. However, this is a very heavily used machine, and we were able to acquire at most 41 processors at any one time. The machines at Georgia Tech are part of the Interactive High Performance Computing Lab, and the computers in Italy are part of the Italian "Computational Grid" and as such, were spread throughout the entire country : Rome, Bologna, Padova, Milan, and Naples.

Table 5.9 shows the percentage of the work done at each participating location, and Table 5.10 shows the percentage of the work done by machines of each operating system and architecture type.

The branch-and-bound parameters are given in Table 5.11. Section 3.7 describes in detail the purpose of each of these parameters. The strategy at the top of the tree is to compute accurate prospective bounds to make the best branching choices possible. Thereafter the amount of effort to obtain prospective bounds is

Table 5.8: Nug30 computational pool

Number	Arch/OS	Location
414	Intel/Linux	Argonne
96	SGI/Irix	Argonne
1024	SGI/Irix	NCSA
16	Intel/Linux	NCSA
45	SGI/Irix	NCSA
246	Intel/Linux	Wisconsin
146	Intel/Solaris	Wisconsin
133	Sun/Solaris	Wisconsin
190	Intel/Linux	Georgia Tech
94	Intel/Solaris	Georgia Tech
54	Intel/Linux	Italy (INFN)
25	Intel/Linux	New Mexico
12	Sun/Solaris	Northwestern
5	Intel/Linux	Columbia U.
10	Sun/Solaris	Columbia U.

Table 5.9: Contribution of each location during nug30 run

Location	Percentage
Argonne	42.27
Wisconsin	33.69
Gatech	11.90
INFN	5.65
NCSA	2.74
New Mexico	1.42
Columbia	1.23
NW	1.10

Table 5.10: Contribution of each architecture during nug30 run

Arch/OS	Percentage
Intel/Linux	79.57
SGI/Irix	8.76
Sun/Solaris	6.17
Intel/Solaris	5.50

Table 5.11: Nug30 branching strategy

Gap	Depth	Rule	NFW1	NFW2	NFW3	NBEST	NUPDATE
0.34	3	4	150	150	100	30	30
0.24	6	4	150	150	50	30	30
0.12	7	4	150	100	25	5	30
0.06	8	2	150	100	–	–	30
0.03	10	2	100	75	–	–	30
0.00	50	2	75	50	–	–	30

scaled back until levels 5-7 when less expensive branching rules begin to be used at the majority of nodes.

The nug30 computation was started on June 8, 2000 at 11:05, with the master machine located at the Condor pool at the University of Wisconsin-Madison. The computation completed on June 15, at 21:20, and in the interim the nug30 computation was stopped five times for various reasons, and resumed using the checkpointing technique of Section 5.2.3.3. The progress of the computation was monitored via the Internet as it progressed using the iMW-QAP environment described in [37].

Table 5.12: Nug30: Overall statistics

Level	Nodes	Fathom	Elim.	Time
0	1	0.0000	0.0000	25.08
1	9	0.0000	0.0000	608.67
2	225	0.0000	0.0000	17611.96
3	6120	0.0000	0.0400	398354.05
4	157947	0.0173	0.1818	2934065.38
5	3300091	0.1575	0.3885	19752634.78
6	42497088	0.3952	0.5355	59055117.10
7	286494566	0.6135	0.6245	82612511.42
8	956340170	0.7428	0.6421	31244385.17
9	1937015498	0.8212	0.6342	41471871.87
10	2660681036	0.8540	0.6635	45069685.33
11	2614933119	0.8742	0.6825	32951941.62
12	1984262473	0.9091	0.7108	19435198.26
13	938579161	0.9206	0.7328	7602716.45
14	338657371	0.9265	0.7514	2287052.36
15	99017742	0.9295	0.7685	573502.34
16	24225224	0.9320	0.7834	118175.81
17	4998989	0.9337	0.7944	20571.61
18	885554	0.9355	0.8028	3043.65
19	135201	0.9367	0.8082	389.95
20	18058	0.9311	0.8139	43.97
21	2317	0.9094	0.8180	5.07
22	344	0.8605	0.7995	0.83
23	77	0.8312	0.7912	0.23
24	19	0.7368	0.7333	0.04
25	8	0.7500	0.8000	0.02
26	2	0.0000	0.7500	0.00
27	2	0.0000	N/A	0.00
Total	11892208412			3.46e8

Table 5.13: Nug30: Breakdown of strategies used

	Frequency strategy used						Nodes
Level	I	II	III	IV	V	VI	
0	1.0000	0.0000	0.0000	0.0000	0.0000	0.0000	1
1	1.0000	0.0000	0.0000	0.0000	0.0000	0.0000	9
2	1.0000	0.0000	0.0000	0.0000	0.0000	0.0000	225
3	0.7894	0.1668	0.0374	0.0060	0.0003	0.0000	6120
4	0.0000	0.5939	0.2514	0.0992	0.0331	0.0224	157947
5	0.0000	0.1972	0.3177	0.2616	0.1257	0.0978	3300091
6	0.0000	0.0357	0.1809	0.3202	0.2387	0.2245	42497088
7	0.0000	0.0000	0.0803	0.2715	0.3029	0.3453	286494566
8	0.0000	0.0000	0.0000	0.2351	0.3201	0.4447	956340170
9	0.0000	0.0000	0.0000	0.0000	0.5456	0.4544	1937015498
10	0.0000	0.0000	0.0000	0.0000	0.5374	0.4626	2660681036
11	0.0000	0.0000	0.0000	0.0000	0.0000	1.0000	2614933119
12	0.0000	0.0000	0.0000	0.0000	0.0000	1.0000	1984262473
13	0.0000	0.0000	0.0000	0.0000	0.0000	1.0000	938579161
14	0.0000	0.0000	0.0000	0.0000	0.0000	1.0000	338657371
15	0.0000	0.0000	0.0000	0.0000	0.0000	1.0000	99017742
16	0.0000	0.0000	0.0000	0.0000	0.0000	1.0000	24225224
17	0.0000	0.0000	0.0000	0.0000	0.0000	1.0000	4998989
18	0.0000	0.0000	0.0000	0.0000	0.0000	1.0000	885554
19	0.0000	0.0000	0.0000	0.0000	0.0000	1.0000	135201
20	0.0000	0.0000	0.0000	0.0000	0.0000	1.0000	18058
21	0.0000	0.0000	0.0000	0.0000	0.0000	1.0000	2317
22	0.0000	0.0000	0.0000	0.0000	0.0000	1.0000	344
23	0.0000	0.0000	0.0000	0.0000	0.0000	1.0000	77
24	0.0000	0.0000	0.0000	0.0000	0.0000	1.0000	19
25	0.0000	0.0000	0.0000	0.0000	0.0000	1.0000	8
26	0.0000	0.0000	0.0000	0.0000	0.0000	1.0000	2
27	0.0000	0.0000	0.0000	0.0000	0.0000	1.0000	2
nodes	5.07e3	2.26e6	3.18e7	3.17e8	2.89e9	8.65e9	
time	3.78e5	4.59e7	1.05e8	2.01e7	7.57e7	9.83e7	

Table 5.14: Nug30: Relative gap information

Level	Mean	Std.Dev.
0	1.000000	0.000000
1	0.831554	0.020814
2	0.624186	0.081081
3	0.427808	0.110772
4	0.250402	0.117602
5	0.139766	0.092794
6	0.079827	0.061827
7	0.054073	0.042902
8	0.042495	0.035038
9	0.044746	0.038556
10	0.042124	0.034589
11	0.036580	0.029799
12	0.033484	0.026720
13	0.029598	0.023569
14	0.026529	0.021050
15	0.023947	0.018907
16	0.021481	0.016842
17	0.019074	0.014764
18	0.016843	0.012807
19	0.014817	0.010948
20	0.012890	0.009399
21	0.010861	0.007465
22	0.009697	0.006477
23	0.008641	0.005549
24	0.007687	0.002250
25	0.005699	0.001703
26	0.003387	0.000178
27	0.003387	0.000178

The statistics of the branch-and-bound tree are given in Tables 5.12, 5.13, 5.14. The largest number of nodes are at levels 10 and 11 of the tree, although the most time was spent at level 7. The bulk of the computation was performed on nodes at levels 6-11, and the branching strategy used was relatively effective in distributing the work among these levels of the tree. Table 5.13 indicates that around 43.8% of the time was spent using more expensive branching rules that compute prospective bounds to make branching decisions, a higher percentage than for smaller problems.

From Table 5.14 it is seen that the bounds on subproblems are quickly approach the objective value 6124. An average relative gap of 5.4% at level 7 corresponds to a gap of about 42 between the average subproblem bound and solution. Even modest improvements in the lower bound would greatly reduce the size of the tree; whether a corresponding decrease in computation time would be realized depends on the additional cost of the improvements.

Finally, Figures 5.5 – 5.7 highlight various aspects of the performance of MW throughout the solution of nug30. Figure 5.5 plots the number of workers participating throughout the one week run. The downward spikes indicate the five occasions when the solution process was interrupted. An average of 653 workers participated in the run, with a peak of 1009. Figure 5.6 shows that in general the LAP solution rate varies in relation to the number of workers participating. Lastly, Figure 5.7 shows how the lazy best-first strategy of Section 5.2.3.2 was effective in keeping the master task pool well-stocked with subproblems. Near the end of the run, the best-first

Figure 5.5: Number of workers participating in nug30 solution

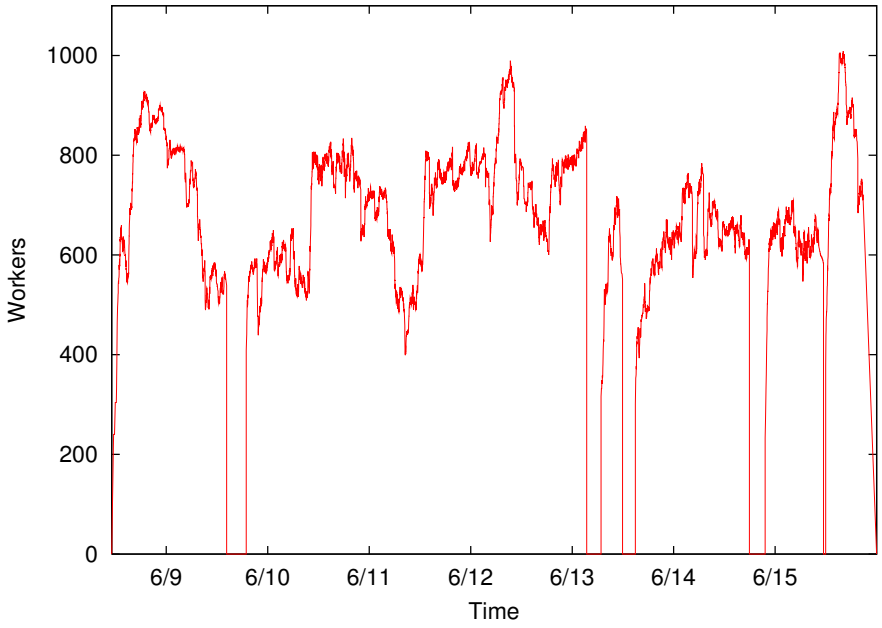


Figure 5.6: Thousands of LAPs solved during nug30 solution

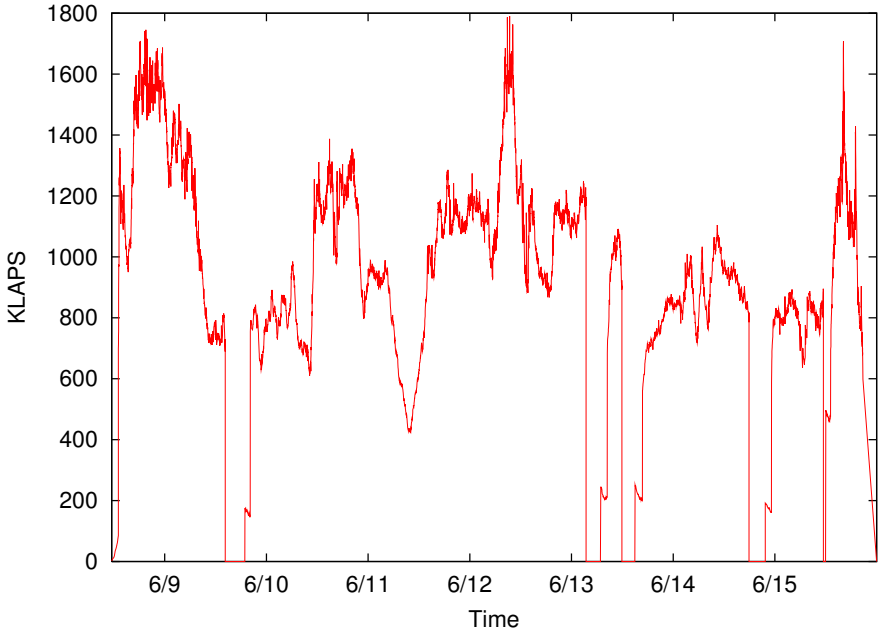
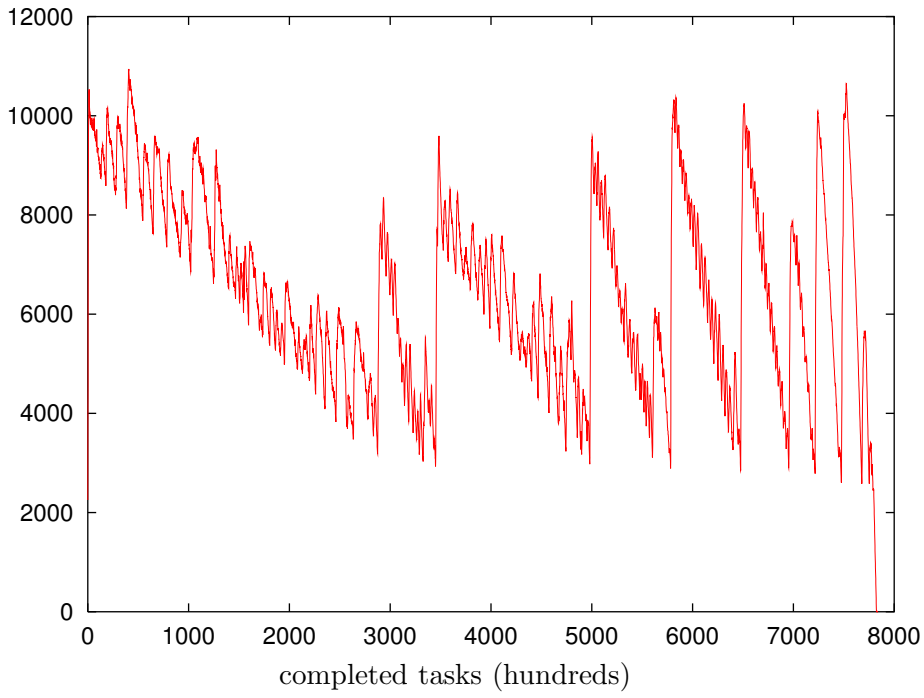


Figure 5.7: Size of master task pool during nug30 solution



strategy is adopted more frequently, due to the fact that the last high-level nodes to be processed have better lower bounds.

5.4.2 MWQAP results on other large problems

MWQAP was also used to solve the QAPLIB instances nug24, nug25, nug27, nug28, and kra30b. The latter three instances were previously unsolved. We created the nug27 and nug28 instances by removing the last two/three facilities and locations from nug30, and ran several heuristics, including GRASP [77] and simulated annealing to obtain a good feasible solution. For both problems the simulated annealing code of Taillard (see Section 1.7 for details) produced the best solution, and this was used as the BKV to initialize the branch-and-bound procedure. In both cases this BKV was proved optimal.

The problem kra30b arose from a hospital planning application in 1972 [55]. The related kra30a problem was first solved by Hahn et al. [43]. The kra30b problem uses the same flow matrix as kra30a, but has a distance matrix derived from a 5 by 3 by 2 grid. See [44] for an interesting discussion of these problems. The total wall-clock time required by MWQAP to solve kra30b was approximately 3.8 days. The number of worker machines averaged approximately 417, and peaked at 780. A total of 1.36×10^8 CPU seconds was expended by worker machines. The equivalent computation time on a single HP-C3000 workstation would be approximately 2.7 years. Even though the root gap between QPB and the optimal solution to kra30b is nearly twice the root gap of nug30, kra30b is clearly much easier to solve. One

explanation is that the distance matrix to kra30b has an eight-fold symmetry, as opposed to the four-fold symmetry of nug30.

CHAPTER 6
MODIFICATIONS AND EXTENSIONS

6.1 An Alternate Quadratic Programming Bound

The quadratic programming bound of Chapter 2 was obtained via a semidefinite programming relaxation of QAP. The relaxation provides matrices S', T' that make the objective function convex. A different convexification scheme is motivated by simply relaxing the integrality constraints of QAP, and adding diagonal perturbations to A and B. This approach was suggested in [74] but to our knowledge has not been pursued. Letting $Q = B \otimes A$, the QAP formulation (1.1) can be written as

$$\begin{aligned} \min \quad & x^T Q x + c^T x \\ \text{s.t.} \quad & \sum_{i=1}^{n} x_{ij} = 1 \\ & \sum_{j=1}^{n} x_{ij} = 1 \\ & x_{ij} \in \{0, 1\}, \end{aligned} \tag{6.1}$$

where $x = \text{vec}(X)$. If the 0-1 constraint is relaxed to $x_{ij} \geq 0$, (6.1) is a quadratic program. Unfortunately, since in general A and B are not positive semidefinite the QP is not convex and hence difficult to solve to optimality.

In Chapter 1 we showed that one can make row, column, and diagonal perturbations of QAP(A, B, C) producing QAP(A', B', C') with the same solution value, see

(1.11). Adding sufficiently large diagonal perturbations produces positive semidefinite A', B', resulting in a convex quadratic program that can be solved to obtain a lower bound.

To get the best bound, we seek the "smallest" possible diagonal perturbations making A' and B' positive semidefinite. A reasonable choice is to find perturbations d_A, d_B with the smallest sum. The perturbations d_A are found by solving the following semidefinite program:

$$\begin{array}{ll} \min & e^T d_A \\ \text{s.t.} & A + \operatorname{Diag}(d_A) \succeq 0, \end{array} \tag{6.2}$$

and a similar problem is used to obtain d_B. To put (6.2) in the standard form of Appendix A, let $b = -e, F_0 = A, F_i = -e_i e_i^T$. Solving the QP (6.1) with A', B', C' results in a lower bound, which we call QPB-S1(A, B, C).

A better bound is obtained by taking into account the constraints of (6.1). The feasible set of 6.1 is the set of matrices X with row and column sums equal to one, so Q need only be convex over this set. Since $Q = (B \otimes A)$ and $(V \otimes V)$ is a basis for the nullspace of the row and column sum constraints, this condition is equivalent to $V^T(A + \operatorname{Diag}(d_A))V \succeq 0$, $V^T(B + \operatorname{Diag}(d_B))V \succeq 0$. A semidefinite program to obtain d_A is then

$$\begin{array}{ll} \min & \operatorname{tr} V^T(A + \operatorname{diag}(d))V \\ \text{s.t.} & V^T(A + \operatorname{diag}(d))V \succeq 0. \end{array} \tag{6.3}$$

and a similar problem is used to obtain d_B. The bound QPB-S2 is obtained by solving (6.1) using the perturbations from (6.3). If the same procedure is performed using

$-A, -B$ in place of A, B, a valid lower bound is also obtained. The lower bound obtained using negated A, B is called QPB-S3.

Table 6.1 shows that the QPB-S bounds vary considerably in quality, but in general the bounds produced are not much better than PB. To compute the bounds, the procedures QPB-S1, QPB-S2, QPB-S3 were coded in Matlab. The SDPs that arise are solved by SDPHA [12], and the QPs are solved using an interior-point algorithm based on [4]. The SDPs that arise in the computation of the QPB bound have closed form solutions (see Section 2.3.1), but the SDPs (6.2), (6.3) do not, therefore the QPB-S bounds cannot be computed as efficiently. Although on some problems the QPB-S bounds are better than QPB, for example QPB-S3 on nug25, for other problems such as rou20 none of the QPB-S bounds is as good as PB. In short, the increased cost and widely varying quality of the QPB-S bounds prevent them from being good candidates for use in a branch-and-bound algorithm.

6.2 A Bound Improvement Procedure

Bounds such as QPB and GLB produce dual matrices U such that if the assignment $i \rightarrow j$ is made, the bound on S_{ij} is at least $z + U_{ij}$. It follows that $z + U \bullet X$ is also a lower bound on QAP, where X is a solution of LAP(U). Normally $U \bullet X = 0$, so no improvement in the lower bound is gained by solving LAP(U). Now consider the situation where two different lower bounds z_1, z_2 have been obtained along with associated dual matrices U_1, U_2. For example, z_1, U_1 might come from GLB, and z_2, U_2 might come from QPB. Consider the objective value $v(X)$ of any

Table 6.1: QP-S bounds on QAPLIB problems

Problem	BKV	QPB-S1	QPB-S2	QPB-S3	PB
esc16a	68	37	38	-14	47
esc16b	292	252	253	234	250
esc16c	160	66	67	-63	95
had16	3720	544	712	3588	3560
had18	5358	618	818	5222	5104
had20	6922	-338	328	6799	6625
nug12	5780	-85	-52	482	472
nug14	1014	-367	-277	897	871
nug15	1150	-581	-483	980	973
nug16a	1610	-843	-693	1453	1403
nug16b	1240	-518	-459	1086	1046
nug17	1732	-1237	-1087	1523	1487
nug18	1930	-1187	-1079	1728	1663
nug20	2570	-1432	-1315	2269	2196
nug21	2438	-4514	-3759	2228	1979
nug22	3596	-9312	-8094	3389	2966
nug24	3488	-5680	-5063	3199	2960
nug25	3744	-4704	-4306	3413	3190
rou12	235528	124031	137714	104552	200024
rou15	354210	95090	119079	134239	296705
rou20	725520	147570	196702	311083	597045
scr12	31410	-50233	-45709	12124	4727
scr15	51140	-123503	-108812	20500	10355
scr20	110030	-498104	-455747	55129	16113
tai20a	703482	277608	316196	242695	575831

discrete solution to QAP. It must be the case that:

$$v(X) \geq z_1 + U_1 \bullet X, \tag{6.4}$$

$$v(X) \geq z_2 + U_2 \bullet X. \tag{6.5}$$

The minimum $v(X)$ is also a valid lower bound for QAP so we consider the problem

$$\begin{aligned} \min \quad & \theta \\ \text{s.t.} \quad & \theta \geq z_1 + U_1 \bullet X \\ & \theta \geq z_2 + U_2 \bullet X \\ & Xe = X^T e = e \\ & X \geq 0. \end{aligned} \tag{6.6}$$

The dual problem with multipliers λ_1, λ_2 is as follows:

$$\begin{aligned} \max_{\lambda_1,\lambda_2 \geq 0} \min_{\theta,X} \quad & (\theta - \lambda_1\theta - \lambda_2\theta) + (\lambda_1 U_1 + \lambda_2 U_2) \bullet X + \lambda_1 z_1 + \lambda_2 z_2 \\ \text{s.t.} \quad & Xe = X^T e = e \\ & X \geq 0. \end{aligned} \tag{6.7}$$

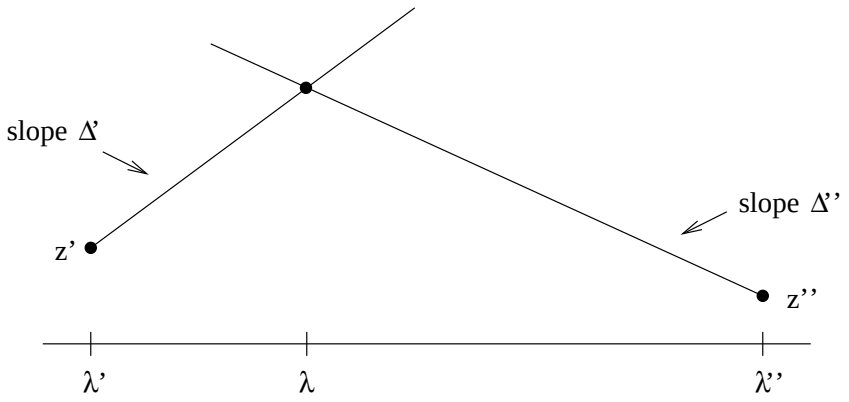
Since feasible solutions to the dual problem (6.7) are lower bounds on the solution value of (6.6), any feasible solution to (6.7) is a lower bound on QAP. Noting that $\lambda_1, \lambda_2 \geq 0$ and $\lambda_1 + \lambda_2 = 1$ is implied by the objective in (6.7), the dual problem is rewritten as:

$$\max_{\lambda, 0 \leq \lambda \leq 1} \lambda z_2 + (1-\lambda) z_1 + \text{LAP}(\lambda U_2 + (1-\lambda) U_1) \tag{6.8}$$

Taking the subdifferential of (6.8) shows that for there to be a possibility of improving the bound by increasing λ, it must be the case that

$$\Delta(\lambda) = (U_2 - U_1) \bullet X(\lambda) \geq z_1 - z_2 \geq 0,$$

Figure 6.1: Bound improvement procedure



where $X(\lambda)$ is a solution of $\mathrm{LAP}(\lambda U_2 + (1-\lambda)U_1)$. In particular, one can check if a bound better than z_1 and z_2 may exist by examining $\Delta(0), \Delta(1)$.

More generally, if for some $0 \leq \lambda' < \lambda'' \leq 1$ it is the case that $\Delta(\lambda') > 0$ and $\Delta(\lambda'') < 0$ then there exists a $\lambda \in [\lambda', \lambda'']$ that may produce an improved bound. Figure 6.1 shows that a good candidate for such a λ is given by the intersection of the lines $z' + (\lambda - \lambda')\Delta'$ and $z'' + (\lambda - \lambda'')\Delta''$. Solving for λ,

$$\lambda = \frac{z'' - z' + \lambda'\Delta' - \lambda''\Delta''}{\Delta' \Delta''}. \tag{6.9}$$

Replacing λ' or λ'' with λ, the process can be repeated as desired.

The above procedure requires two valid lower bounds and two valid dual matrices U. Two interesting variants are:

- QPB-GLB: z_1, U_1 from GLB, z_2, U_2 from QPB.
- QPB-imp: z_1, U_1 from iteration k of QPB, z_2, U_2 from iteration $k-1$ of QPB.

Table 6.2: QPB-GLB/imp on QAPLIB problems

Problem	QPB-FW	QPB-GLB	QPB-imp
nug12	477.49	493.00	477.97
nug15	991.33	991.33	992.05
nug16b	1066.33	1066.33	1067.54
nug18	1693.26	1693.26	1695.14
nug20	2246.87	2246.87	2248.55
nug21	2043.54	2043.54	2045.25
nug24	3014.44	3014.44	3017.31
nug27	4590.15	4590.15	4596.09
nug30	5344.96	5344.96	5350.18

These two variants were implemented and compared on a set of QAPLIB problems. Three iterations of the bound improvement procedure were performed. The results in Table 6.2 show that for problems where QPB and GLB differ considerably in bound quality, QPB-GLB is simply the larger of QPB and GLB (QPB, on all problems but nug12). Using the last two iterations of the FW procedure gives better results, although the improvement in the bound is quite small. In general the cost of forming and solving the extra linear assignment problems required is not worth the rather small improvement in the bound.

6.3 The Parametric Eigenvalue Bound and QPB

The parametric eigenvalue bound of [30], presented in Chapter 1 is based on choosing perturbations g, h, r and s that maximize the eigenvalue bound of the perturbed problem (1.11). Unfortunately, the problem of finding perturbations that maximize the bound is a difficult nonlinear optimization problem. Nevertheless, in

Table 6.3: QPB-EVB3 bound
on QAPLIB problems

Problem	QPB-FW	QPB-EVB3
nug12	477.493	499.62
nug15	991.326	1007.49
nug16b	1066.33	1076.75
nug18	1693.26	1720.83
nug20	2246.87	2292.80
nug21	2043.54	2144.22
nug24	3014.44	3107.55
nug27	4590.15	4733.72
nug30	5344.96	5495.58

[30] it is shown that the bounds produced are often of very high quality. Since QPB is related to an eigenvalue bound, the idea of trying to combine EVB3 and QPB in some way is an attractive one.

The authors of [30] have provided us with a Fortran implementation of the EVB3 procedure, which has been converted into C++ and modified so that it can be used as a callable procedure. The result is a routine that takes as input A, B, C and produces as output perturbed A', B', C' that approximately maximize EVB. We then compute QPB using the data A', B', C', producing the bound QPB-EVB3.

Table 6.3 shows that the bounds on the perturbed problems QPB-EVB3 are significantly better than the conventional QP bound QPB-FW. The cost to obtain the QPB-EVB3 bound however is several times the cost of QPB-FW.

There are several ways QPB-EVB3 can be used in a branch-and-bound algorithm. The simplest option is to simply replace the root problem QAP(A, B, C) with

the perturbed problem QAP(A', B', C'). Another option is to apply the EVB3 procedure to some or all subproblems. Optionally, these perturbations can be propagated to child subproblems where they can be reused.

The cost of simply perturbing the root problem is minimal, however our results show that using the perturbed root problem throughout the entire algorithm results in poorer branching decisions. The reasons for this behavior are not entirely clear. The time and number of nodes required to solve nug20 increase by nearly a third when the root problem is perturbed, despite the fact that the root bound is substantially better. Similar results hold for other problems.

Perturbing some or all individual nodes in the tree does not seem to improve the performance of our algorithm either, because the improvement in the bound does not seem to justify the considerable extra cost. For example, for the nug16b problem, if EVB3 is used through level 4 of the branch-and-bound tree then the total CPU time increases by nearly 10%, and the total number of nodes in the tree decreases from 6268 to 6065. Using EVB3 only at the top of the tree improves the lower bounds, but not enough to fathom many more subproblems.

A third option is to compute the perturbations at higher level nodes, and then propagate these perturbations down to all of their descendants. A perturbation that is optimal for a problem is suboptimal for its child subproblems, but still may result in improved bounds for the subproblems. Since the EVB3 procedure is called at a smaller number of nodes, the additional cost is not as significant. However, the

bounds at some or all subproblems have the possibility of being improved. There are two potential drawbacks to this approach. The first is that an optimal perturbation for a parent problem may not provide improved bounds for its descendants. In fact, perturbations that increase the parent bound may lead to a decrease in the child bound, since costs are shifted to the linear term C as assignments are made. Secondly, perturbations must be stored along with the subproblems in order for them to be propagated down the tree. As a result, the branch-and-bound algorithm requires additional storage. This last perturbation scheme was not evaluated to see if the potential lower bound improvement was enough to offset the potential difficulties described here.

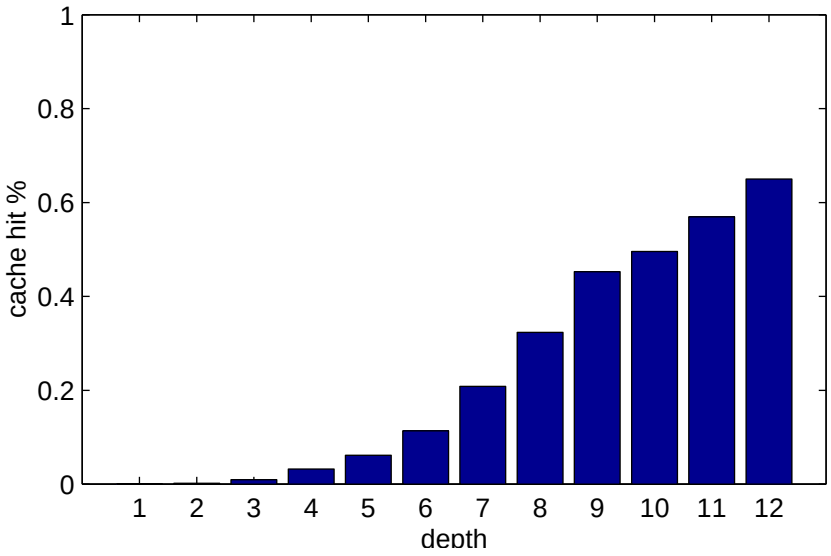
6.4 Improvements to QPB

6.4.1 Caching Search Directions

One of the most time-consuming parts of the computation of QPB is computing AX_k^*B at each iteration, see Figure 2.7. Since X_k^* is a permutation matrix, the time to compute AX_k^*B is the same as one matrix-matrix multiplication. It is reasonable to expect that over the course of 50-150 FW iterations, some of the permutation matrices X_i^* will repeat. One strategy to improve performance is to cache the matrices AX_k^*B and reuse them whenever possible, avoiding matrix multiplications.

For the nug15 problem, we recorded FW iterations for which solutions were repeated, and saved the corresponding AX_k^*B in a hash table. Figure 6.2 shows the interesting result that the percentage of iterations where X^* was reused for the nug15

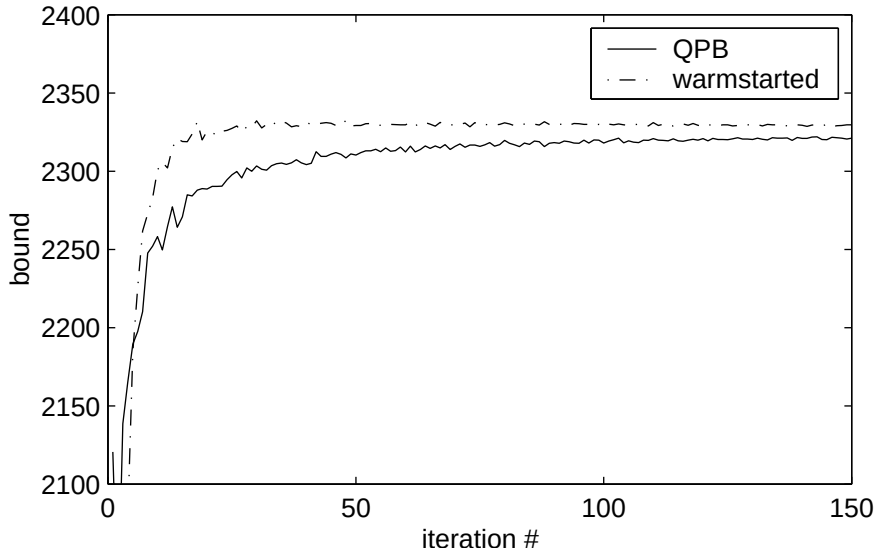
Figure 6.2: Caching search directions for nug15



problem increases with the depth of the problem, eventually exceeding 50%. The increased cache hit rate can be attributed to the fact that QPB solution matrices become closer to permutation matrices, since the linear terms of subproblems deeper in the tree are more dominant.

However, to get this high cache hit rate, all previous X^* need to be stored, which requires extra time and storage. If solutions repeat after only a few iterations, then a smaller hash table could be used, and saving AX_k^*B becomes potentially worthwhile. Unfortunately, in our limited testing the average number of iterations between cache hits was around 30 iterations for the critical portion of the branch-and-bound tree. This means that to be useful, the cache would have to retain at least the last 30 AX_i^*B matrices. For larger problems the cache hit rate is even lower, and

Figure 6.3: Convergence of FW algorithm using different starting points



hence the time saved by reusing AX_k^*B is not enough to compensate for the extra time and storage required.

6.4.2 Warmstarting QPB

In the FW algorithm for QPB (see Figure 2.9), the initial iterate X_0 is always set to E/n. The solution matrix X of a parent subproblem could be used to initialize the X_0 matrices of child subproblems in an attempt to improve the convergence of the FW algorithm. The technique of using the solution of a parent node to initialize child subproblems is commonly referred to as **warmstarting**.

If a subproblem was obtained from its parent by assigning $i \to j$, then the initial solution matrix X_0 is obtained by removing row i and column j from the solution matrix to the parent problem. It is likely that X_0 is then infeasible – its row

and column sums are not equal to one. Therefore, in this case the FW algorithm is modified to take a full step at the first iteration, that is, $\alpha = 1$. This ensures that all subsequent iterates will be feasible.

A plot of the convergence of the conventional FW algorithm and the warmstarted FW algorithm for the root problem of nug20 is shown in Table 6.3. After a few iterations, the warmstarted bound is better than the conventional QP bound, although after 150 iterations the improvement is only 0.3%.

The use of warmstarting requires the use of the parent bound's solution matrix X, therefore $O(n^2)$ extra storage is required for each branch-and-bound node. However, the warmstarting procedure can be used in branching rules 3 and 4 to compute prospective bounds, without additional storage. Although the use of warmstarting using Rules 3 and 4 provides better lower bounds, since the increase is relatively uniform warmstarting does not cause substantially different branching decisions to be made.

6.4.3 Computing QPB using an Integer LAP Solver

The QPB-FW lower bound is an iterative process that requires the solution of linear assignment problems (LAP). The previously mentioned LAP solver of Jonker and Volgenant [49] was modified to handle floating point data, however most commonly used LAP solvers such as [11, 16] require integer cost matrices. To use an integer LAP solver we can make a small modification of the FW procedure described in Section 2.5 based on the use of a matrix $\tilde{G}_k \leq G_k$. Since $X \geq 0$ for any feasible

X, it follows that

$$\begin{aligned}\langle\lambda(\hat{A}),\lambda(\hat{B})\rangle_- + f(X) &\geq \langle\lambda(\hat{A}),\lambda(\hat{B})\rangle_- + f(X_i) + \tilde{G}_i \bullet X - G_i \bullet X_i \\ &= \langle\lambda(\hat{A}),\lambda(\hat{B})\rangle_- + f(X_i) - G_i \bullet X_i + \mathrm{LAP}(\tilde{G}_i) + \tilde{U}_i \bullet X \\ &= \tilde{z}_i + \tilde{U}_i \bullet X,\end{aligned}$$

where $\tilde{U}_i \geq 0$ is the matrix of reduced costs from $\mathrm{LAP}(\tilde{G}_i)$,

$$\tilde{z}_i = \langle\lambda(\hat{A}),\lambda(\hat{B})\rangle_- + f(X_i) - G_i \bullet X_i + \tilde{G}_i \bullet \tilde{X}_i,$$

and $\tilde{X}_i$ is an optimal solution of $\mathrm{LAP}(\tilde{G}_i)$. By using a matrix $\tilde{G}_i$ in place of G_i we can restrict the LAP solved on each FW iteration to have integer data. In particular, if

$$\tilde{G}_i = \frac{1}{\theta_i}\lfloor\theta_i G_i\rfloor,$$

where θ_i is a positive scaling factor, then $\tilde{G}_i \leq G_i$, $\mathrm{LAP}(\tilde{G}_i) = (1/\theta)\,\mathrm{LAP}(\theta\tilde{G}_i)$, and $\theta\tilde{G}_i$ is an integer matrix. By using a larger value of θ_i, $\mathrm{LAP}(\tilde{G}_i)$ becomes a better approximation of $\mathrm{LAP}(G_i)$, but the time to solve $\mathrm{LAP}(\theta\tilde{G}_i)$ typically increases. (It is well known that the time to solve a LAP with integer data is sensitive to the scale of the data.)

In Table 6.4 we have computed root bounds for the Nugent problems using several scaling factors θ ranging from 1 to 100. For the smallest scaling factors, the bound is roughly 5% worse for most instances, and for high scaling factors the bounds produced are very close to those computed using a floating-point LAP solver,

Table 6.4: Effect of different scaling factors for QAPLIB problems

	Scaling factor			
Problem	1	5	10	100
nug12	469.992	476.19	477.58	477.50
nug15	983.175	989.42	990.27	991.70
nug16b	1055.41	1064.98	1065.48	1066.33
nug18	1683.47	1690.71	1692.12	1693.04
nug20	2232.86	2244.53	2245.39	2247.19
nug21	2030.33	2041.38	2042.97	2043.45
nug24	2999.82	3011.6	3012.88	3014.09
nug27	4579.49	4587.41	4589.52	4589.55
nug30	5326.69	5341.68	5343.23	5344.82
nug20 time (ms)	41.1	41.1	41.2	46.6

for example the bounds in Table 6.3. The time to obtain the nug20 root bound is also given in Table 6.4, showing that the speed of the bound computation does indeed depend on the scaling factor; in particular the use of $\theta = 100$ increased the time by about 13%. The times reported are the average of 100 bound computations. A simple modification that gives good results is to use a larger scaling factor such as 10 for only the last 5 or 10 FW iterations, and a scaling factor of 1 elsewhere. However, in our experience the best results were obtained using a floating-point LAP solver.

6.4.4 Warmstarting LAP

The major step in each iteration of the FW procedure is to solve an LAP to find a search direction. In Section 6.4.1 we saw that the solutions to these LAPs often repeat. Here we investigate the idea of using the solution of the previous LAP(G_{i-1})

as a starting point to LAP(G_i). Since G_{i-1} and G_i are likely to be close, perhaps their solutions X^*_{i-1}, X^*_i are also similar.

The difficulty is that the LAP code used by QPB-FW does not use an initial primal solution X_i. To warmstart LAP, a different LAP algorithm must be used. We implemented the alternating basis algorithm of [7], which requires only an initial primal solution, and as such is a good candidate for warmstarting. Although LAP algorithms of this type are less efficient in practice as evidenced by the computational results in [49], our hope was that by providing an initial primal solution close to the solution of the LAP better results would be obtained. Unfortunately we found that this was not the case, and the idea was abandoned.

6.5 Alternate Branching Strategies

6.5.1 Making Branching Decisions using GLB

The branching rules of Chapter 3 were divided into two categories: those that compute prospective bounds ("strong branching") and those that do not. The reasoning behind the first class of rules is that computing lower bounds for branching purposes provides more accurate information, resulting in better branching decisions. It is generally unwise to use strong branching strategies at all nodes in a branch-and-bound tree as the time computing prospective bounds becomes prohibitive. In Chapter 3 two techniques for reducing the extra cost of strong branching were proposed: computing bounds for only some of the possible subproblems, and computing bounds more quickly by reducing the number of FW iterations.

An attractive alternative idea is to use an altogether different, less expensive bounding procedure to compute prospective bounds. Two obvious candidates are GLB and PB, both of which are computed much more quickly than QPB. Unfortunately, the use of these bounds for branching purposes was not effective. Using GLB or PB to compute prospective bounds at the top of the tree gave poor performance, as the branching decisions made were poorer, and the size of the tree became much larger. We solved nug20 twice, once using the parameters listed in Chapter 5 and a second time where the third strategy used GLB to compute prospective bounds instead of QPB. Even though the time used by the third strategy was cut nearly in half, the overall time increased by over 30% because of poorer branching decisions. However, an interesting compromise may be to use strong branching using GLB or PB for subproblems in the middle of the tree.

6.5.2 Lower Bounds Using GLB

The idea of using GLB or PB to make branching decisions has been rejected. However, GLB can also be used to compute lower bounds for subproblems that are close to fathoming. (PB is already computed after a single iteration of QPB, see Theorem 2.5.1.) The intuition is if the relative gap at a node is quite small, then even though GLB is weaker for most problems perhaps the subtree will not fathom quite as quickly as QPB, but quickly enough so that an overall savings in CPU time is realized. The philosophy is to trade one QPB computation for (hopefully) a small number of GLB computations.

Once again our computational results showed that such an approach did not produce better results. To the contrary, using GLB increased both the CPU time and number of nodes required to solve a problem. For example, on the nug20 problem if GLB is used only for nodes with relative gaps less than 2%, the execution time increases from 4132.1 to 10448.1 seconds, and the number of nodes increases by a factor of 30. Even if the QPB bound is relatively tight for a particular subproblem, there is no guarantee that another bound such as GLB will also be tight. Moreover, the assignments made to obtain subproblems high in the tree are made with the objective of increasing QPB, not GLB, so there is reason to believe that switching to a different bound deep in the tree may be counterproductive.

For some of the problems in Table 1.2 the root GLB bound is significantly stronger than QPB, for example the "scrxx" problems. For these problems, it appears that using GLB at all nodes in the tree is the most effective solution method.

6.5.3 Non-polytomic Branching Strategies

Our branch-and-bound implementation uses strictly polytomic strategies, that is, a fixed facility is chosen and assigned to all possible locations. Other strategies are possible.

Single assignment branching, as described in Section 3.5.1, considers the assignment of a single facility to a location. To generate one subproblem, the assignment is made, and to generate the other it is disallowed. An analogue of Rule 1 in Chapter 3 is to choose the assignment with maximal U_{ij}. Unfortunately, such a branching

strategy does not seem to be very effective for QPB. The reason is that disallowing an assignment by setting $C_{ij} = M$ for large M does not usually improve QPB substantially, and so the resulting subproblem is nearly as difficult as the original problem. However, it is known that deeper subproblems in a branch-and-bound tree have larger linear terms C, so it is possible that a binary strategy may be more useful deeper in the tree. An interesting modification of our algorithm would be to use polytomic strategies at the top of the tree, and binary strategies for nodes deep in the tree. Since for large problems there are a large number of nodes 10 to 12 levels deep in the tree, there is a potential for great improvement.

A variation on the conventional binary branching scheme is to try to generate two child subproblems of approximately equal difficulty. Suppose that a row i has been chosen, and can be assigned to any of the locations in $J = \{1, \ldots, n\}$. Then to generate two subproblems, J can be divided into two subsets J_L, J_R. For the first subproblem, i must be assigned to one of the locations in J_L, which is enforced by setting $C_{ij} = L$ for $j \in J_R$. Similarly, for the second subproblem i must be assigned to one of the locations in J_R. To ensure that the subproblems are of approximately equal difficulty, J_L, J_R can be chosen so that for example, $\sum_{j \in J_L} U_{ij} \approx \sum_{j \in J_R} U_{ij}$. Some preliminary testing on small QAP instances indicated that such a strategy is not as effective as a polytomic strategy, hence this idea was not pursued.

A final variation on the conventionally proposed branching strategies is to generate several different types of subproblems. For locations j that will improve

the lower bound the most, a subproblem corresponding to the assignment $i \rightarrow j$ is generated. For example, suppose $\bar{u}$ is the mean of $U_{ij}, j \in \{1, \ldots, n\}$. Then if $U_{ij} \geq \bar{u}$, generate a subproblem S_{ij}. To generate the last child subproblem, the locations $\{j \,|\, U_{ij} \geq \bar{u}\}$ are disallowed. The motivation is to generate only one difficult subproblem, instead of several.

6.6 A New Heuristic for QAP Based on QPB

Any lower bound procedure LB for QAP can be used in the following constructive procedure that produces a suboptimal permutation p.

Figure 6.4: QAP heuristic based on QPB

```
p = DIVE(A, B, C)
    A' = A, B' = B, C' = C, d' = 0
    k = 0, p = {}
    WHILE k < n
        FIND ASSIGNMENT i → j MINIMIZING d' + LB(S_ij).
        p(i) ← j.
        COMPUTE A', B', C', d' ACCORDING TO (3.1)
        k = k + 1
```

The procedure in Figure 6.4 essentially follows one path down a branch and bound tree without fathoming. A path is extended by choosing the assignment that minimizes the lower bound. The performance of this heuristic using QPB is given in Table 6.5. Note that for most of the problems tested, we were able to obtain a

Table 6.5: Performance of limited enumeration procedure

Problem	BKV	Solution	Gap
had14	2724	2724	0.0000
had16	3720	3766	0.0124
nug12	578	578	0.0000
nug15	1150	1152	0.0017
nug20	2570	2628	0.0226
nug25	3744	3774	0.0080
rou12	235528	238134	0.0111
rou15	354210	363432	0.0260
scr20	110030	113234	0.0291
tai20a	703482	715658	0.0173

solution within 3% of the optimum. Surprisingly, the heuristic works well even on problems where QPB is not very strong; for example the root gap for scr20 is over 78%, but the resulting solution is within 3% of optimal. We can easily extend the procedure to look at more permutations; in this case we have a limited enumeration procedure for QAP.

6.7 Future Work and Extensions

The exact solution of QAP is extremely challenging. Solving QAP using branch-and-bound requires tight, efficiently computed lower bounds and intelligent branching strategies. In this chapter we have surveyed some possible improvements to our basic branch-and-bound framework, however other possible extensions remain.

An improved lower bounding scheme has the possibility of improving the performance of our algorithm dramatically. Even a decrease in the gap between root

bound and solution from 15% to 10% on a problem would have a dramatic effect in reducing the size of the branch-and-bound tree. If such an improvement could be obtained without significantly increasing the bound computation time, overall performance would be improved. There several possible extensions to QPB that are worth considering. The first is to more efficiently implement the interior-point solution method described in Section 2.4. In addition to providing better bounds, perhaps better branching information could also be obtained. A second possible extension is to combine QPB with another bound, along the lines of Section 6.2. A more modest improvement would be to improve the convergence behavior of the FW algorithm currently used to compute lower bounds. The possibility of a theoretical strengthening or extension to QPB is also a worthy topic of research. Bounds that allow for a tradeoff between bound quality and the computational effort to obtain them are of particular interest. In any case, new lower bounding techniques should be evaluated in terms of their performance as part branch-and-bound algorithm, over a range of problem sizes and classes.

Improved branching strategies may also lead to better QAP solution methods. To a large extent the effectiveness of a branching strategy is determined by the quality of information the bounding procedure provides. For example, the reduced costs matrix U was used to great effect to construct simple but effective branching strategies for QPB. At the first few levels of the tree, even more expensive strategies could be employed in an effort to make the best branching decisions possible. For example, an

extension of the look-ahead branching rule that computes bounds on subproblems two levels deep may be worthwhile. The middle portion of the tree is difficult to handle because the only available alternatives are to use either the reduced costs matrix U, or a much more computationally expensive strong branching rule. A branching rule that provides more accurate information than the U matrix, yet is not as expensive as the strong branching rules could be of great value.

Lastly, an improved branch-and-bound implementation would provide both direct and indirect benefits. A direct benefit would be a modest improvement in the overall performance of the algorithm. However the real value of a good branch-and-bound implementation is that it provides a general framework that is easily adapted for use with different lower bounding and branching strategies. Careful implementations provide the opportunity for experimentation and evaluation of new ideas, and using the MW framework allows for these new ideas to be tested on large QAPs without having to wait days or weeks for results. Given the exponential behavior of branch-and-bound algorithms for QAP, significant advances are achieved not by code optimization but by algorithmic improvements.

APPENDIX A
NOTATION AND DEFINITIONS

A.1 Notation

$\bullet$	$A \bullet B = \mathrm{tr} B^T A = \sum_{i=1}^{n} \sum_{j=1}^{n} A_{ij} B_{ij}$
$X \succeq 0$	X is positive semidefinite
$\lambda(X)$	vector of eigenvalues of X
$\mathrm{diag}(X)$	a vector consisting of the diagonal elements of X
$\mathrm{Diag}(x)$	a diagonal matrix whose elements are the entries of x
e, E	vector, square matrix consisting of ones
Kronecker product	$A \otimes B = \begin{bmatrix} a_{11}B & a_{12}B & \dots & a_{1n}B \\ a_{21}B & a_{22}B & \dots & a_{2n}B \\ \vdots & & & \vdots \\ a_{m1}B & a_{m2}B & \dots & a_{mn}B \end{bmatrix}$.
minimal vector product	$\langle x, y \rangle_- = \min_{P \in \Pi} \langle x, Py \rangle$
trace	$\mathrm{tr}\, A = \sum_{i=1}^{n} A_{ii}$
$\mathrm{vec}(X)$	vector containing columns of X stacked on each other

A.2 Optimization Problems

A.2.1 Linear Programming

Linear programming is to minimize (or maximize) a linear function subject to linear constraints on the variables. Any linear program can be written in the standard form:

$$\begin{aligned} \min \quad & c^T x \\ \text{s.t.} \quad & Ax = b \\ & x \geq 0. \end{aligned} \tag{A.1}$$

A vector x that satisfies the constraints of (A.1) is called **feasible**. If some of the constraints on x are inequalities, or if x is not restricted to be nonnegative, we can add additional variables to put the problem into standard form.

For a linear program in the form (A.1), we can define the **dual problem**

$$\begin{aligned} \max \quad & b^T y \\ \text{s.t.} \quad & A^T y + s = c \\ & s \geq 0. \end{aligned} \tag{A.2}$$

The (primal) problem (A.1) and its dual (A.2) are closely related. It is well known (see [70]) that if both the primal variable x and the dual variables (y, s) are feasible, then $c^T x \geq b^T y$. That is, a feasible solution to the dual problem provides a lower bound on the objective value of the primal problem. Additionally, $c^T x = b^T y$ if and only if x and (y, s) are the optimal solutions to (A.1) and (A.2) respectively.

Linear programs with many thousands of variables and constraints can be solved efficiently by using the simplex algorithm, or by interior-point methods.

A particular case of LP is the linear assignment problem (LAP), discussed in Section A.3.

A.2.2 Quadratic Programming

Quadratic programming (QP) is the optimization of a convex quadratic function subject to linear constraints. We write QP in the following standard form:

$$\begin{aligned} \min \quad & x^T Q x + c^T x \\ \text{s.t.} \quad & Ax = b \\ & x \geq 0, \end{aligned} \tag{A.3}$$

where Q is an $n \times n$ symmetric positive semidefinite matrix, that is, $x^T Q x \geq 0$ for any vector x. Of course, if $Q = 0$ we are left with a linear program in standard form. A variety of methods are used to solve nontrivial QPs, see [70] for more information. In general it is not too difficult to solve QPs where $m, n \leq 100$.

A.2.3 Semidefinite Programming

Semidefinite programming (SDP) is an extension of linear programming where the nonnegativity constraints on a vector x have been replaced by positive semidefiniteness constraints on a matrix variable X. We write SDP in the following form:

$$\begin{aligned} \min \quad & C \bullet X \\ \text{s.t.} \quad & A_i \bullet X = b_i, \quad i = 1, \ldots, m \\ & X \succeq 0, \end{aligned} \tag{A.4}$$

where $X \succeq 0$ indicates that X is positive semidefinite. The matrices C, X and $A_i, i = 1, \ldots, m$ are all $n \times n$ and symmetric. The dual problem for SDP is:

$$\begin{aligned} \max \quad & b^T y \\ \text{s.t.} \quad & \sum_{i=1}^{m} y_i A_i + S = C \\ & S \succeq 0. \end{aligned} \tag{A.5}$$

The primal variable is X, the dual variables are (y, S). Weak duality holds between (A.4) and (A.5), that is $b^T y \leq C \bullet X$ if y and X are feasible, and strong duality holds under an interior point assumption for both problems. Recently developed interior point methods can be used to efficiently obtain solutions to semidefinite programs of reasonable dimension ($m, n \leq 1000$, depending on the sparsity of the input matrices). See [85] for a survey.

A.2.4 Integer Programming

An integer linear program is a linear program where the variables are all integer:

$$\begin{aligned} \min \quad & c^T x \\ \text{s.t.} \quad & Ax = b \\ & x \geq 0, x \in Z^n. \end{aligned} \tag{A.6}$$

In many combinatorial optimization applications, x is restricted to be binary: $x \in \{0, 1\}$. A mixed integer linear program (MILP) is an LP where only some of the components of x are required to be integer. Nemhauser and Wolsey's text [68] provides a comprehensive introduction to integer programming. Nonlinear integer programs

are also of interest; QAP is a quadratic integer programming problem.

A.3 Linear Assignment Problems

Many lower bounds for QAP require the solution of one or more Linear Assignment Problems (LAPs). In particular, the solution of linear assignment problems dominates the time to compute bounds such as QPB and GLB. We briefly describe the formulation of LAP, methods for its solution, and survey available computer codes. An excellent survey of LAP is [17].

As the name suggests, in a linear assignment problem some number of objects in one set are to be assigned to an equal number of objects in a second set. The cost of assigning objects in the first set to objects in the second set are given, and the objective is to find the assignment that minimizes the total assignment cost. For example, the problem may be to assign workers to jobs so that the total time spent working on jobs is minimal. LAP is written as follows:

$$\min_{p \in \Pi} \sum_{i=1}^{n} c_{ip(i)} \tag{A.7}$$

where the c_{ij} are the given costs, and p is an assignment where $p(i) = j$ indicates that worker i has been assigned to job j. LAP can also be posed as the problem of finding the minimum cost matching on a bipartite graph.For this reason, in algorithms texts this problem is also sometimes referred to as "minimum cost weighted bipartite matching".

LAP can also be written in terms of permutation matrices:

$$\begin{aligned} \min \quad & \sum_{i=1}^{n}\sum_{j=1}^{n} c_{ij}x_{ij} \\ \text{s.t.} \quad & \sum_{i=1}^{n} x_{ij} = 1, \quad j = 1, 2, \ldots, n, \\ & \sum_{j=1}^{n} x_{ij} = 1, \quad i = 1, 2, \ldots, n, \\ & x_{ij} \in \{0, 1\}, \quad i, j = 1, 2, \ldots, n. \end{aligned} \tag{A.8}$$

As written, LAP is an integer program. However, the following result of Birkhoff motivates more efficient solution methods for LAP.

Theorem A.3.1 (Birkhoff) *The vertices of the assignment polytope correspond uniquely to permutation matrices.*

It follows from A.3.1 that the integrality constraint on the x_{ij} can be relaxed to $x \geq 0$, turning (A.8) into a linear program. In other words, the linear objective is optimized over the set of doubly stochastic matrices, rather than permutation matrices.

Solution methods for LAP include the Hungarian algorithm [58], auction algorithms [10, 11], and algorithms based on the simplex method for LP such as [7]. An early Fortran implementation of the Hungarian algorithm is given in [16]. Several improvements for the basic Hungarian algorithm have been proposed, for example [20]. Jonker and Volgenant [48] provided an implementation of the algorithm in [20] which is extremely efficient for small, dense LAPs.

Certain special cases of LAP are solved even more efficiently. In particular, consider the solution of LAP(C) where $c_{ij} = a_i b_j$.

Table A.1: Machine and compiler characteristics

model	HP9000/785/C3000 workstation
RAM	512 MB
SPECint95	31.8
SPECfp95	26.0
OS	HP-UX 10.20
compiler	g++ version 2.95.2
compiler options	`+O4`

Theorem A.3.2 *Let $a, b \in \Re^n$, and let $c_{ij} = a_i b_j$. Suppose further that the elements of a are sorted in nonincreasing order. Then* $\mathrm{LAP}(C) = \langle a, b \rangle_- = \min_{p \in \Pi} \sum_{i=1}^{n} a_i b_{p(i)}$, *where the minimizing p sorts the elements of b in nondecreasing order.*

A.4 Machine Characteristics

All results, except for those using MW-QAP, were obtained using an HP-C3000 workstation. The characteristics of the workstation and compiler are summarized in Table A.1.

APPENDIX B
MWQAP USER'S GUIDE

B.1 Solving QAPs using MWQAP

Our branch and bound implementation provides a powerful platform for solving QAPs, whether sequentially or in parallel. The distributed version, MWQAP, uses the MW framework of Chapter 5. In what follows we assume that the sequential version of the implementation is used, but most of what is described also applies to MWQAP.

Our purpose is to describe the steps a user goes through to solve a QAP, beginning with specification of a QAP and corresponding branching strategy, and ending with how to interpret the provided output.

B.1.1 Providing Problem Data to MWQAP

MWQAP is implemented in ANSI C++ and has been ported to HP, Sun, Linux, Windows and SGI platforms, among others. To solve a QAP, a user of MWQAP needs to provide the following information:

- problem data,
- symmetry information,
- specification of solution method (branching strategy).

In what follows, the format of each is specified.

B.1.1.1 QAPLIB

MWQAP accepts input in the standard QAPLIB format [19] consisting of the problem size, distance and flow matrices A and B, and optionally a linear term C. The QAPLIB format is quite simple: the first entry is be the number of facilities and locations n, followed by the entries of A in row-major order, and finally the entries of B. Lastly, MWQAP will read the entries of C, if they are given, otherwise the QAP is assumed to be homogeneous and $C = 0$. The standard QAPLIB problems can be found at the QAPLIB web site:

`http://www.opt.math.tu-graz.ac.at/qaplib/`

The user may also provide a file containing symmetry information in the format of Section B.1.1.3. Lastly, the user provides a parameters file that specifies how the branch-and-bound algorithm is to be applied. (Certain parameters, such as the fathoming tolerance Δ from Figure 3.13 and weighting parameter from (3.2), can only be changed by recompiling the code.)

B.1.1.2 Specifying a branching strategy

The parameters in our algorithm are divided into three classes – branching parameters, bounding parameters, and when a particular branching rule is to be used. Let us look at a parameters file and examine it in detail.

Each line of the parameters file begins with a numerical value (with one ex-

Figure B.1: MWQAP parameters file

```
1    has the location of a symmetry file been provided?
/space/brixius/mw-qap/SYM/nug12.sym
-1   restriction on tree depth (-1 means none)
2    number of branching rules

0.5
2
   4
 150  Frank-Wolfe iterations, for bound
 150  Frank-Wolfe iterations, when we know we can't fathom
  50  Frank-Wolfe iterations, for row selection
   2  which bound?
  30  step for parametric bound
  30  compute B for best _ rows of U

   2
   0  Frank-Wolfe iterations, for bound
   0  Frank-Wolfe iterations, when we know we can't fathom
   0  Frank-Wolfe iterations, for row selection
   4  which bound?
   0  step for parametric bound
   0  compute B for best _ rows of U
```

ception), and is optionally followed by a comment which is ignored by MWQAP. The order in which parameters is fixed and may not be changed by the user. The first line indicates whether the user has supplied a file containing symmetry information. If the value is nonzero, the second line contains the path to the symmetry file. The next line contains the maximum depth to which the algorithm is to be run. A negative value indicates simply that the entire branch-and-bound tree is to be generated. The fourth line of the file indicates how many different branching rules are to be used.

Then, each of the branching rules is given, one at a time. The first two parameters for a strategy determine exactly when the strategy is to be used. Relative gap and depth parameters are given. The relative gap was defined in (3.4) and is a measure of the difficulty of a subproblem between 0 and 1. Nodes with smaller relative gaps are generally easier than those with higher relative gaps. In our example, the relative gap entry 0.5 and depth 2 indicate that the first strategy is to be used for all nodes with gap less than 0.5 whose depth is no greater than 2.

The remaining parameters indicate how the branching and bounding phases are to be performed. The third parameter defines the branching rule to be used. There are four branching rules, described in Chapter 3. Rules 3 and 4 are more computationally expensive but make better branching decisions than Rules 1 and 2. Next the parameters FW1, FW2, FW3 as defined in Chapter 3 are given, which control how many Frank-Wolfe iterations are to be performed to obtain bounds.

The next parameter indicates which lower bounding technique is to be used.

Table B.1: Lower bounds supported by MWQAP

Value	Lower bound procedure
1	QPB
2	QPB-GLB
3	QPB-imp
4	GLB
5	PB
6	QPB-EVB3

The available lower bounding procedures are given in Table B.1. QPB-GLB computes both QPB and GLB, and uses the procedure of Section 6.2 to find a bound which may be better than both. QPB-imp uses the same procedure, except it uses the last two lower bounds computed by the QPB-FW algorithm. QPB-EVB3 first finds a perturbed problem according to the scheme of Section 6.3, and then computes QPB on the perturbed problem. For most problems, using QPB (or GLB in some cases) for all strategies will give the best results. Of course, if a bound that does not use QPB is used, such as GLB, the FW parameters are ignored. Next, the NUPDATE parameter is given, which is again ignored if the bound does not involve QPB. Lastly, the NBEST parameter is given, which controls the cost of branching rules 3 and 4, see Figure 3.15.

As our example indicates, the format of the parameters file is quite rigid, and a particular field must have an entry, even if it does not apply. We have summarized the parameters in Table B.2. Parameter files for many QAPLIB instances that have

Table B.2: Summary of MWQAP parameters

#	Description	Values
1	symmetry file given?	$\{0, 1\}$
2	symmetry file path	text
3	tree depth	integer
4	number of branching rules	integer
S1	gap	[0,1]
S2	depth	integer
S3	branching rule	$\{0, 1, 2, 3, 4\}$
S4	FW1	pos. integer
S5	FW2	pos. integer
S6	FW3	pos. integer
S7	bound	pos. integer
S8	NUPDATE	pos. integer
S9	NBEST	pos. integer

been found to give good results are provided with MWQAP.

B.1.1.3 Providing symmetry information

Section 3.6 describes how the branch-and-bound algorithm upon which MWQAP is based exploits symmetry in the distance matrix to decrease the size of the search tree. The symmetry of a QAP instance is described by three sets, J_1, J_2, J_3. J_1 indicates that at the root of the tree, one need only consider assigning a facility i to locations $j \in J_1$. Together, J_2, J_3 indicate symmetries deeper in the tree. If the set of fixed locations at a node all belong to the set J_2, one need only consider assigning the next facility to locations $j \in J_3$. In Section 3.6 it was shown that for nug06, where the distance matrix corresponds to a 2×3 grid, $J_1 = \{1, 2\}$, $J_2 = \{2, 5\}$, $J_3 = \{1, 2, 4, 5\}$.

Figure B.2: Symmetry file for nug06

```
2
1 2

1

2 4
2 5
1 2 4 5
```

The corresponding symmetry file for nug06 is listed in Figure B.2.

The first entry indicates the size of J_1, and is followed by the entries in J_1. Then, the number of J_2/J_3 sets is given. Some problems have more than one pair of J_2/J_3 sets; nug06 only has one. Next, the size of J_2 and J_3 are given, followed by the entries of J_2 and finally J_3. MWQAP provides symmetry files for many of the QAPLIB problems with symmetry.

B.1.2 Interpreting the Output

MWQAP provides a simple command line interface and detailed output information. Here is a snapshot of the solution of the nug20 problem using our code.

```
% bnb ../QAPLIB/nug20.dat -u 2572

input file: ../QAPLIB/nug20.dat
incumbent: 2572
parameters file: ../bnb-aug15/PARAM/nug20.par

gap     depth   strat       FW          best    update
0.6     1       4       150/150/100     30      30
0.5     3       4       150/150/50      30      30
0.3     5       4       150/100/25      5       30
```

```
0.07    8       2       100/ 75/**      **      30
-1      1000    2        75/ 50/**      **      30

weight = 0.5
read symmetry info from ./SYM/nug20.sym
Thu Aug 17 15:21:04 nodes = 100000  stack size = 34  cpu time = 675.39
Thu Aug 17 15:32:43 nodes = 200000  stack size = 46  cpu time = 1369.14
New objective value: 2570
Thu Aug 17 15:43:42 nodes = 300000  stack size = 48  cpu time = 2022.7
Thu Aug 17 15:55:00 nodes = 400000  stack size = 16  cpu time = 2695.47
Thu Aug 17 16:06:57 nodes = 500000  stack size = 49  cpu time = 3407.04
Thu Aug 17 16:19:05 nodes = 600000  stack size = 51  cpu time = 4129.45
Thu Aug 17 16:31:15 nodes = 700000  stack size = 7  cpu time = 4854.28
```

The user supplies the path to an input file and indicates any additional information – in this case that the initial upper bound is to be 2572. The parameters file `nug20.par` and symmetry file `nug20.sym` are automatically located and read. MWQAP prints the parameters to be used and then begins the solution process. Occasionally a status message is printed summarizing the progress of the algorithm. Once the algorithm is complete, a detailed statistical summary is printed. This information is extremely useful for tuning the algorithm for maximum efficiency.

When the algorithm halts, detailed information about the solution process is provided:

```
  ---- RESULTS ----

    Total Time: 4861.05 seconds
         Nodes: 701026 / 560581 / 1
    root bound: 2246.87
 fw iterations: 27894552

level  nodes      %fathom %c.elim     time
 0         1      0.0000  0.0000      3.26
 1         6      0.0000  0.0102     43.83
```

```
 2         97      0.0000  0.0689    210.98
 3       1581      0.0734  0.2541    493.34
 4      18505      0.3563  0.4660    756.65
 5     101759      0.6452  0.5975    887.55
 6     217991      0.7899  0.6743   1184.87
 7     208897      0.8607  0.7194    837.10
 8     106131      0.8921  0.7472    333.88
 9      34750      0.9006  0.7632     87.01
10       8996      0.9058  0.7752     18.51
11       1904      0.9139  0.7710      3.44
12        338      0.9112  0.7708      0.57
13         55      0.8727  0.7551      0.05
14         12      0.8333  0.9167      0.01
15          1      0.0000  0.8000      0.00
16          1      0.0000  0.7500      0.00
17          1      0.0000  N/A         0.00

Breakdown of strategies used:
 0 |   1.0000   0.0000   0.0000   0.0000   0.0000  | 1
 1 |   1.0000   0.0000   0.0000   0.0000   0.0000  | 6
 2 |   0.0000   0.5773   0.3402   0.0722   0.0103  | 97
 3 |   0.0000   0.0044   0.4105   0.5161   0.0689  | 1581
 4 |   0.0000   0.0000   0.0502   0.7370   0.2128  | 18505
 5 |   0.0000   0.0000   0.0019   0.6264   0.3718  | 101759
 6 |   0.0000   0.0000   0.0000   0.4569   0.5431  | 217991
 7 |   0.0000   0.0000   0.0000   0.3043   0.6957  | 208897
 8 |   0.0000   0.0000   0.0000   0.1982   0.8018  | 106131
 9 |   0.0000   0.0000   0.0000   0.0000   1.0000  | 34750
10 |   0.0000   0.0000   0.0000   0.0000   1.0000  | 8996
11 |   0.0000   0.0000   0.0000   0.0000   1.0000  | 1904
12 |   0.0000   0.0000   0.0000   0.0000   1.0000  | 338
13 |   0.0000   0.0000   0.0000   0.0000   1.0000  | 55
14 |   0.0000   0.0000   0.0000   0.0000   1.0000  | 12
15 |   0.0000   0.0000   0.0000   0.0000   1.0000  | 1
16 |   0.0000   0.0000   0.0000   0.0000   1.0000  | 1
17 |   0.0000   0.0000   0.0000   0.0000   1.0000  | 1
nodes       7       63     1800   262413   436743
time    47.09   200.53  1132.87  2180.83  1299.73

Relative gap information:
      mean      std. dev
 0| 1.000000    0.000000
```

```
 1| 0.770907    0.055469
 2| 0.496791    0.130989
 3| 0.262795    0.117476
 4| 0.151131    0.086881
 5| 0.103614    0.065947
 6| 0.075118    0.052414
 7| 0.056325    0.041305
 8| 0.045106    0.033506
 9| 0.038056    0.028171
10| 0.033526    0.023454
11| 0.028730    0.020333
12| 0.026741    0.017411
13| 0.020542    0.009882
14| 0.014478    0.009743
15| 0.011140    0.000000
16| 0.006151    0.000000
17| 0.006151    0.000000

Objective: 2570
incumbent solution: 18 6 3 5 16 19 17 13 4 2 8 7 14 1 11 9 15 0 10 12
```

The first section of output gives general information about the branch-and-bound tree. The number of nodes, and CPU time spent at each level of the tree is given. The `%fathom` column indicates the percentage of nodes at the given level fathomed by the lower bounding procedure. The entries in `%c.elim` give the fraction of the potential child nodes that were eliminated during the branching phase.

The second section of the output reports how often each branching strategy was used at each level. For example, it can be seen from the output that strategy 4 was used 51.61% of the time at level 3. At the bottom of the section, the total number of times the strategy was used along with the total cpu time spent by the strategy is reported. This information is extremely useful in adjusting parameters, as demonstrated in Section 4.4.

The third section of the output gives some sense of the relative difficulties of the nodes at each level of the tree. The distribution of relative gaps of the nodes at a given level of the tree is usually close to a normal distribution, see for example Figure 3.14. The `mean` column gives the mean relative gap, and `std.dev` gives the standard deviation at each level. This information can be used to change parameters, as in Chapter 4, and can be compared in successive runs or estimates to see how changing branching strategies affects the performance of the algorithm. Finally the optimal assignment and corresponding objective value is printed.

B.2 Additional Features

B.2.1 Command-line Options

The user of MWQAP can also control some of the details of the solution process by using one or more of the command-line options given in Table B.3. The `-b` option is used to compute a benchmark time for the purpose of comparing machine performance, see Chapter 5. The `-c` option causes MWQAP to enter console mode, where the user can compute bounds on subproblems, fix assignments, and investigate the branch-and-bound tree. The console mode can be used to ensure the correctness of the algorithm. The `-e` switch causes the code to estimate the solution of the given problem, see the next section for more details. The `-h` uses the heuristic of Section 6.6 to find a suboptimal solution to the given QAP. The last three options allow the user to use a different parameters file, symmetry file, or initial incumbent value for the problem.

Table B.3: Command-line options

Option	Description
`-b`	benchmark
`-c`	enter console (do not run branch and bound)
`-d X`	run to depth X (overrides parameter file)
`-e`	estimate (do not run branch and bound)
`-f X`	scale all times by X
`-h`	use heuristic (do not run branch and bound)
`-p F`	use parameters file P
`-s F`	use symmetry file F
`-u X`	set initial upper bound X

Figure B.3: Estimator parameters file

```
3         depth
10000     total dives
1000      print estimate every
100       print update every
1.5       importance sampling exponent
30        throw out dives greater than
```

B.2.2 Using the Estimator

MWQAP also includes the estimation procedure of Chapter 4. The procedure makes random walks down the branch-and-bound tree to generate estimates of all the statistics normally provided by the code. The user can control how the estimation is obtained by editing the `estimate.par` file in the home directory of MWQAP.

Figure B.3 shows what an estimator parameters file looks like. Most of the parameters will be familiar from the discussion in Chapter 4. The third parameter

determines how often a running estimate of all statistics will be printed. The fourth parameter prints a one line message indicating the current time, date, and estimated nodes and CPU time, and is meant to simply signal that the estimate procedure is still running! The last parameter instructs the estimator to ignore any dive reaching beyond a certain depth. This parameter can be used to prevent overestimates.

REFERENCES

[1] W.P. Adams and T. Johnson. Improved linear programming based lower bounds for the quadratic assignment problem. In P. Pardalos and H. Wolkowicz, editors, *Quadratic Assignment and Related Problems*, volume 16 of *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, pages 43–77. AMS, 1994.

[2] R.K. Ahuja, J.B. Orlin, and A. Tiwari. A greedy genetic algorithm for the quadratic assignment problem. Technical report, Sloan School of Management, Massachusetts Institute of Technology, Cambridge, MA, 02139 USA, 1997.

[3] K.M Anstreicher and N. Brixius. A new bound for the quadratic assignment problem based on convex quadratic programming. Technical report, Department of Management Sciences, University of Iowa, Iowa City, Iowa, May 1999.

[4] K.M. Anstreicher, D. den Hertog, C. Roos, and T. Terlaky. A long-step barrier method for convex quadratic programming. *Algorithmica*, 10:365–382, 1993.

[5] K.M Anstreicher and H. Wolkowicz. On lagrangian relaxation of quadratic matrix constraints. Research report CORR 98-24, Department of Combinatorics and Optimization, University of Waterloo, Waterloo, Ontario, Canada, 1998.

[6] A. A. Assad and W. Xu. On lower bounds for a class of quadratic 0,1 programs. *Operations Research Letters*, 4(4):175–180, 1985.

[7] R.S. Barr, F. Glover, and D. Klingman. The alternating basis algorithm for assignment problems. *Mathematical Programming*, 13:1–13, 1977.

[8] R. Battiti and G. Tecchiolli. The reactive tabu search. *ORSA Journal on Computing*, 6(2):126–140, 1994.

[9] M.S. Bazaraa and O. Kirca. A branch-and-bound-based heuristic for solving the quadratic assignment problem. *Naval Research Logistics Quarterly*, 30:287–304, 1983.

[10] D.P. Bertsekas. *Linear network optimization : algorithms and codes*. MIT Press, Cambridge, Mass., 1991.

[11] D.P. Bertsekas. Auction algorithms for network flow problems: A tutorial introduction. *Computational Optimization and Applications*, 1:7–26, 1992.

[12] N. Brixius, F. A. Potra, and R. Sheng. SDPHA: a MATLAB implementation of homogeneous interior-point algorithms for semidefinite programming. *Optimization Methods and Software*, 11&12:583–596, 1999.

[13] A. Brüngger, A. Marzetta, J. Clausen, and M. Perregaard. Joining forces in solving large-scale quadratic assignment problems in parallel. Technical report, Department of Computer Science, University of Copenhagen, 1998.

[14] A. Brüngger, A. Marzetta, J. Clausen, and M. Perregaard. Solving large-scale qap problems in parallel with the search library ZRAM. *Journal of Parallel and Distributed Computing*, 50:157–169, 1998.

[15] A. Brüngger, A. Marzetta, K. Fukuda, and J. Nievergelt. The parallel search bench ZRAM and its applications. *Annals of Operations Research*, 90:45–63, 1999.

[16] R.E. Burkard and U. Derigs. *Assignment and matching problems: Solution methods with Fortran programs*, volume 184 of *Lecture Notes in Economics and Mathematical Systems*. Springer, Berlin, 1980.

[17] R.E. Burkhard and E. Çela. Linear assignment problems and extensions. In D.-Z. Du and P.M Pardalos, editors, *Handbook of Combinatorial Optimization*, volume Supplement Volume A, pages 75–149. Kluwer, 1999.

[18] R.E. Burkhard, E. Çela, P.M. Pardalos, and L.S. Pitsoulis. The quadratic assignment problem. In D.-Z. Du and P.M Pardalos, editors, *Handbook of Combinatorial Optimization*, volume 3, pages 241–337. Kluwer, 1998.

[19] R.E. Burkhard, S.E. Karisch, and F. Rendl. QAPLIB – a quadratic assignment problem library. *Journal of Global Optimization*, 10:391–403, 1997.

[20] G. Carpaneto and P. Toth. Solution of the assignment problem. *ACM Transactions on Mathematical Software*, 6(1):104–111, 1980.

[21] P. Carraresi and F. Malucelli. A new lower bound for the quadratic assignment problem. *Operations Research*, 40(Supplement 1):S22–S27, 1992.

[22] D.-M. Chiang and L. C. Potter. Minimax non-redundant channel coding for vector quantization. In *Proceedings of the International Conference on Acoustics, Speech, and Signal Processing*, volume V, pages 617–620, Minneapolis, MN, 1993.

[23] J. Clausen, S. Karisch, M. Perregaard, and F. Rendl. On the applicability of lower bounds for solving rectilinear quadratic assignment problems in parallel. *Computational Optimization and Applications*, 10:127–147, 1998.

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

[25] D.T. Connolly. An improved annealing scheme for the qap. *European Journal of Operational Research*, 46:93–100, 1990.

[26] T.H. Cormen, C.E. Leiserson, and R.L. Rivest. *Introduction to Algorithms*. MIT Press and McGraw-Hill, Cambridge, MA, 1990.

[27] H. Crowder, E. L. Johnson, and M. W. Padberg. Solving large-scale zero-one linear programming problems. *Operations Research*, 31:803–834, 1983.

[28] M. Dorigo, V. Maniezzo, and A. Colorni. The ant system: Optimization by a colony of cooperating agents. *IEEE Transactions on Systems, Man, and Cybernetics-Part B*, 26:29–41, 1996.

[29] A.V. Fiacco and G.P. McCormick. *Nonlinear Programming, Sequential Unconstrained Minimization Techniques*. Wiley, New York, 1968. Reprinted as *Classics in Applied Mathematics Vol. 4*, SIAM, Philadelphia, 1990.

[30] G. Finke, R.E. Burkhard, and F. Rendl. Quadratic assignment problems. *Annals of Discrete Mathematics*, 31:61–82, 1987.

[31] C. Fleurent and J. A. Ferland. Genetic hybrids for the quadratic assignment problem. *Quadratic assignment and related problems, P. Pardalos and H. Wolkowicz, editors, DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, 16:173–187, 1994.

[32] L. M. Gambardella, É. D. Taillard, and M. Dorigo. Ant colonies for the quadratic

assignment problem. *Journal of the Operational Research Society*, 50:167–176, 1999.

[33] A.M. Geoffrion and G.W. Graves. Scheduling parallel production lines with changeover costs: Practical applications of a quadratic assignment/LP approach. *Operations Research*, 24:595–610, 1976.

[34] A.M. Geoffrion and R.E Marsten. Integer programming algorithms: A framework and state-of-the-art survey. *Management Science*, 18(9):465–490, 1972.

[35] P.C. Gilmore. Optimal and suboptimal algorithms for the quadratic assignment problem. *SIAM Journal on Applied Mathematics*, 10:305–313, 1962.

[36] D.E. Goldberg. *Genetic Algorithms in Search, Optimization, and Machine Learning*. Addison-Wesley Publishing Company, Inc., New York, 1989.

[37] M. Good and J.-P. Goux. iMW : A web-based problem solving environment for grid computing applications. Technical report, Department of Electrical and Computer Engineering, Northwestern University, 2000.

[38] J.-P. Goux, S. Kulkarni, J. Linderoth, and M. Yoder. An enabling framework for master-worker applications on the computational grid. In *Proceedings of the Ninth IEEE International Symposium on High Performance Distributed Computing*, 2000.

[39] J.-P. Goux, J. Linderoth, and M. Yoder. Metacomputing and the master-worker paradigm. Technical report, Mathematics and Computer Science Division, Argonne National Laboratory, 2000.

[40] G.W. Graves and A.B. Whinston. An algorithm for the quadratic assignment problem. *Management Science*, 17:453–471, 1970.

[41] S.W. Hadley, F. Rendl, and H. Wolkowicz. A new lower bound via projection for the quadratic assignment problem. *Mathematics of Operations Research*, 17(3):727–739, 1992.

[42] P. M. Hahn. Progress in solving the nugent instances of the quadratic assignment problem. Technical report, Systems Engineering, University of Pennsylvania, 2000.

[43] P. M. Hahn, W. L. Hightower, T. A. Johnson, M. Guignard-Spielberg, and

C. Roucairol. Tree elaboration strategies in branch and bound algorithms for solving the quadratic assignment problem. Technical report, Systems Engineering, University of Pennsylvania, 1999.

[44] P. M. Hahn and J. Krarup. A hospital facility layout problem finally solved. Technical report, Systems Engineering, University of Pennsylvania, 2000.

[45] P.M. Hahn and T. Grant. Lower bounds for the quadratic assignment problem based upon a dual formulation. *Operations Research*, 46(5):912–922, 1998.

[46] P.M. Hahn, T. Grant, and N. Hall. A branch-and-bound algorithm for the quadratic assignment problem based on the hungarian method. *European Journal of Operational Research*, 108:629–640, 1998.

[47] A. Hertz, É. D. Taillard, and D. de Werra. Tabu search. In E. Aarts and J.K. Lenstra, editors, *Local search in combinatorial optimization*, pages 121–136. John Wiley & Sons, Inc., 1997.

[48] R. Jonker and A. Volgenant. Improving the hungarian assignment algorithm. *OR Letters*, pages 171–175, 1986.

[49] R. Jonker and A. Volgenant. A shortest augmenting path algorithm for dense and sparse linear assignment problems. *Computing*, pages 325–340, 1987.

[50] V. Kaibel. Polyhedral combinatorics of quadratic assignment problems with less objects than locations. In R.E. Bixby, E.A. Boyd, and R.Z. Ríos-Mercado, editors, *Integer Programming and Combinatorial Optimization*, volume 1412 of *Lecture Notes in Computer Science*, pages 409–422. Springer, 1998.

[51] V. Kaibel. Polyhedral methods for the QAP. In P.M. Pardalos and L. Pitsoulis, editors, *Nonlinear Assignment Problems (???)*, pages 1–34. Kluwer Academic Publishers, 1999.

[52] S.E. Karisch, E. Çela, J. Clausen, and T. Espersen. A dual framework for lower bounds of the quadratic assignment problem based on linearization. Technical report, Department of Mathematics, Technical University Graz, 1998.

[53] D.E. Knuth. Estimating the efficiency of backtrack programs. *Mathematics of Computation*, 29:121–136, 1975.

[54] K.O. Kortanek and J. Zhu. A polynomial barrier algorithm for linearly constrained convex programming problems. *Mathematics of Operations Research*, 18:116–127, 1993.

[55] J. Krarup and P.M. Pruzan. Computer-aided layout design. *Mathematical Programming Study*, 9:75–94, 1978.

[56] P.S. Laursen. Simple approaches to parallel branch and bound. *Parallel Computing*, 19:143–142, 1993.

[57] E.L. Lawler. The quadratic assignment problem. *Management Science*, 9:586–599, 1963.

[58] E.L. Lawler. *Combinatorial Optimization: Networks and Matroids*. Holt, Rinehart and Winston, New York, 1976.

[59] E.L. Lawler, J. Lenstra, A. Rinnooy Kan, and D. Shmoys. *The Traveling Salesman Problem: A Guided Tour of Combinatorial Optimization*. Wiley, New York, 1985.

[60] Y. Li, P.M. Pardalos, and M.G.C. Resende. A greedy randomized adaptive search procedure for the quadratic assignment problem. In P.M. Pardalos and H. Wolkowicz, editors, *Quadratic assignment and related problems*, volume 16 of *DIMACS Series on Discrete Mathematics and Theoretical Computer Science*, pages 237–261. American Mathematical Society, 1994.

[61] C.-J. Lin and R. Saigal. On solving large-scale semidefinite programming problems – a case study of quadratic assignment problem. Technical report, Department of Industrial and Operations Engineering, University of Michigan, Ann Arbor, MI, 1999.

[62] J.T. Linderoth and M.W.P. Savelsbergh. A computational study of branch and bound search strategies for mixed integer programming. *INFORMS Journal on Computing*, 11:173–187, 1999.

[63] M. Litzkow, M. Livny, and M.W. Mutka. Condor - a hunter of idle workstations. In *Proceedings of the 8th International Conference of Distributed Computing Systems*, pages 104–111, 1988.

[64] M. Livny, J. Basney, R. Raman, and T. Tannenbaum. Mechanisms for high throughput computing. *SPEEDUP Journal*, 11(1), 1997.

[65] A. Marzetta and A. Brüngger. A dynamic-programming bound for the quadratic assignment problem. In *Computing and Combinatorics: 5th Annual International Conference, COCOON'99*, volume 1627 of *Lecture Notes in Computer Science*, pages 339–348. Springer, 1999.

[66] A.J. Mason and M. Rönnqvist. Solution methods for the balancing of jet turbines. *Computers and Operations Research*, 24(2):153–167, 1997.

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

[68] G.L. Nemhauser and L.A. Wolsey. *Integer and Combinatorial Optimization*. John Wiley & Sons, Inc., New York, 1988.

[69] N. Nesterov and A. Nemirovskii. *Interior-Point Polynomial Algorithms in Convex Programming*. SIAM, Philadelphia, 1994.

[70] J. Nocedal and S.J. Wright. *Numerical Optimization*. Springer Series in Operations Research. Springer, New York, 1999.

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

[72] P.M. Pardalos, L.S. Pitsoulis, and M.G.C. Resende. A parallel GRASP implementation for the quadratic assignment problem. In A. Ferreira and J. Rolim, editors, *Parallel Algorithms for Irregularly Structured Problems – Irregular'94*, pages 111–130. Kluwer Academic Publishers, 1995.

[73] P.M. Pardalos, K.G Ramakrishnan, M.G.C Resende, and Y. Li. Implementation of a variance reduction-based lower bound in a branch-and-bound algorithm for the quadratic assignment problem. *SIAM Journal on Optimization*, 7(1):280–294, 1997.

[74] P.M. Pardalos, F. Rendl, and H. Wolkowicz. The quadratic assignment problem: A survey and recent developments. In P.M. Pardalos and H. Wolkowicz, editors, *Quadratic assignment and related problems*, volume 16 of *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, pages 1–42. AMS, 1994.

[75] L.S. Pitsoulis, P.M. Pardalos, and D.W Hearn. Approximate solutions to the tur-

bine balancing problem. *Submitted to European Journal of Operational Research*, 1998.

[76] F. Rendl and H. Wolkowicz. Applications of parametric programming and eigenvalue maximization to the quadratic assignment problem. *Mathematical Programming*, 53:63–78, 1992.

[77] M.G.C. Resende, P.M. Pardalos, and Y. Li. Algorithm 754: Fortran subroutines for approximate solution of dense quadratic assignment problems using GRASP. *ACM Transactions on Mathematical Software*, 22:104–118, 1996.

[78] M.G.C. Resende, K.G. Ramakrishnan, and Z. Drezner. Computing lower bounds for the quadratic assignment problem with an interior point algorithm for linear programming. *Operations Research*, 43:781–791, 1995.

[79] S. Sahni and T. Gonzalez. P-complete approximation problems. *Journal of the Association of Computing Machinery*, 23:555–565, 1976.

[80] M. Sanders and E. McCormick. *Human Factors in Engineering and Design*. McGraw-Hill, New York, sixth edition, 1987.

[81] J. Skorin-Kapov. Tabu search applied to the quadratic assignment problem. *ORSA J. Computing*, 2:33–45, 1990.

[82] L. Steinberg. The backboard wiring problem: A placement algorithm. *SIAM Review*, 3:37–50, 1961.

[83] D.E. Stewart and Z. Leyk. Meschach: Matrix computations in c. In *Proceedings of the Center for Mathematics and its Applications*, volume 32, The Australian National University, 1994.

[84] É. D. Taillard. Robust taboo search for the quadratic assignment problem. *Parallel Computing*, 17:443–455, 1995.

[85] L. Vandenberghe and S. Boyd. Semidefinite programming. *SIAM Review*, 38:49–95, 1996.

[86] M.R. Wilhelm and T.L. Ward. Solving quadratic assignment problems by simulated annealing. *IEEE Transactions*, 19:107–119, 1987.

[87] Q. Zhao, S.E. Karisch, F. Rendl, and H. Wolkowicz. Semidefinite programming relaxations for the quadratic assignment problem. CORR Report 95-27, University of Waterloo, February 1998.