Spring 2018

Efficient local optimization for low-rank large-scale instances of the quadratic assignment problem

Cole Stiegler
University of Iowa

EFFICIENT LOCAL OPTIMIZATION FOR LOW-RANK LARGE-SCALE INSTANCES OF THE QUADRATIC ASSIGNMENT PROBLEM

by

Cole Stiegler

A thesis submitted in partial fulfillment of the
requirements for the Doctor of Philosophy
degree in Applied Mathematical and Computational Sciences
in the Graduate College of
The University of Iowa

May 2018

Thesis Supervisor: Professor David E. Stewart

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

Cole Stiegler

has been approved by the Examining Committee for the thesis requirement for the Doctor of Philosophy degree in Applied Mathematical and Computational Sciences at the May 2018 graduation.

Thesis committee: _____
David Stewart, Thesis Supervisor

_____
Ann Campbell

_____
Palle Jorgensen

_____
Amaury Lendasse

_____
Jeffrey Ohlmann

ACKNOWLEDGEMENTS

The path through grad school has many pitfalls and challenges and I am lucky to have many people that have helped me through the entire process.

First and foremost I want to thank my family: Mom, Dad, Keir, and Betsy. Your unwavering belief in me and my work here been crucial from my first semester all the way through my defense. I am so fortunate to have such an amazing and fun family.

To my friends, Maria, Kathryn, Julia, Gordon, Rolando, Kaitlin, Alex, Neha, and Nick; I cannot overstate how much I appreciate the bonds we've developed over schoolwork, life decisions, and dancing. Ken and Greg, thank you for our many illuminating discussions, there is a direct connection between those conversations and the quality of this

A big thank you to my advisor, David Stewart. I would not have been able to do this without your guidance and patience. Also, I am eternally grateful for your encouragement to seek and take part in summer internships.

I can't forget to thank PanIC, my ultimate team. You have become some of my closest friends and are the part of Iowa City that I will miss the most.

Finally, thank you to the Graduate College and the Math and AMCS departments for their support. My thesis work has been partially supported through the AMCS summer fellowship, Post-Comprehensive Exam Fellowship, and Ballard-Seashore Fellowship.

ABSTRACT

The quadratic assignment problem (QAP) is known to be one of the most computationally difficult combinatorial problems. Optimally solvable instances of the QAP remain of size $n \leq 40$ with heuristics used to solve instances in the range $40 \leq n \leq 256$. In this thesis we develop a local optimization algorithm called GradSwaps (GS). GS uses the first-order Taylor approximation (FOA) to efficiently determine improving swaps in the solution. We use GS to locally optimize instances of the QAP of size $1000 \leq n \leq 70000$ where the data matrices are given in factored form, enabling efficient computations. We give theoretical background and justification for using the FOA and bound the error inherent in the approximation. A strategy for extending GS to larger scale QAPs using blocks of indices is described in detail.

Three novel large-scale applications of the QAP are developed. First, a strategy for data visualization using an extreme learning machine (ELM) where the quality of the visualization is measured in the original data space instead of the projected space. Second, a version of the traveling salesperson problem (TSP) with the squared Euclidean distance metric; this distance metric allows the factorization of the data matrix, a key component for using GS. Third, a method for generating random data with designated distribution and correlation to an accuracy surpassing traditional techniques.

PUBLIC ABSTRACT

The reordering of objects or tasks is an everyday problem. What order should we do the chores? Which is the best arrangement of pictures on the wall? If we can assign different costs to different arrangements, this becomes an assignment problem. Assignment problems seek to minimize the total cost depending on the ordering of tasks or objects. In this thesis, we examine a specific type of assignment problem called the quadratic assignment problem (QAP). The QAP is special because the cost for each placement depends on every other item in addition to the object currently being placed.

If we designed an university campus to minimize the total walking done by students, this would depend on two things: the average number of students traveling between every pair of buildings and the distances between those buildings. Since the total walking from a single building depends on the location of every other building, this is an example of a QAP.

The QAP is computationally a very difficult problem, where solving small instances exactly is difficult. In this thesis, we focus on quickly finding good solutions (as opposed to the best solution) to very large instances of the QAP. This will be done in a variety of previous unexplored applications involving machine learning, the traveling salesperson problem, and statistical correlations.

TABLE OF CONTENTS

LIST OF TABLES

Table

LIST OF FIGURES

Figure

LIST OF ALGORITHMS

Algorithm

CHAPTER 1
QUADRATIC ASSIGNMENT PROBLEM

The quadratic assignment problem (QAP) is a classic combinatorial optimization problem. Originally introduced in 1957 by Koopmans and Beckmann [82], it has proven to be one of the most difficult problems to solve computationally, with provably optimal solutions having been obtained only for relatively modestly-sized instances; for larger instances, the use of heuristics is the current state of the art. This chapter will outline the history and development of the quadratic assignment problem, with the format taking inspiration from Burkard [26].

1.1 Formulations of the QAP

The QAP can be stated in several different ways depending on the purpose the author had in mind while working on the problem. In this section we outline a number of the original, most common, and most useful formulations.

1.1.1 Quadratic Integer Program

The QAP was introduced by Koopmans and Beckmann [82] as a way to increase the flexibility of the linear assignment problem. Their application was the determination of plant locations, where every location had a profit associated with the plant assigned to that location (a linear assignment) as well as a cost associated with the distance and flow between plants (the quadratic assignment). If a_{ki} is the profit associated with plant k in location i, f_{kl} is the flow between plant k and plant

l, and c_{ij} is the cost of transportation per unit flow from location i to location j, the QAP can be formulated as

$$\max_{P \in S_n} \sum_{k,i} a_{ki} p_{ki} - \sum_{k,l} \sum_{i,j} f_{kl} p_{ki} c_{ij} p_{lj} \tag{1.1}$$

where S_n is the set of permutation matrices. Thus, before we further discuss the QAP, we first define permutations and permutation matrices.

1.1.1.1 Permutations and Matrices

A permutation is the reordering of a given set while a permutation matrix is a matrix representation of that reordering. We will encode a permutation $p \in S_n$ as an ordering of the numbers $1 \dots n$, where the number at a given location indicates which item would be assigned to the location. For example,

$$p = [1, 3, 4, 2, 5]$$

indicates the third item will be placed in the second location, the second item will be placed in the fourth location, etc. Additionally, we write $p(1) = 1,\ p(2) = 3,\ p(3) = 4,$ etc. Note that we use S_n to indicate the group of permutations and their corresponding matrices interchangeably, with context sufficient to distinguish between the two.

Permutations p can be encoded in a matrix P as follows:

$$P_{ij} = \begin{cases} 1 & \text{if } p(i) = j \\ 0 & \text{else} \end{cases} \tag{1.2}$$

The equally important translation of permutation matrices into permutation vectors can be done with the above relationship as well. This enables us to move between

permuting subscripts and matrix multiplication, as seen through the following example:

Example 1.1. *We are given data* $X = (x_1, x_2, x_3, x_4)^T$*, permutation* $p = [2, 4, 1, 3]$*, and the corresponding permutation matrix*

$$P = \begin{pmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{pmatrix}$$

We apply the permutation as a reordering and as matrix multiplication to show they yield the same result:

$$\begin{aligned} p(X) &= \begin{pmatrix} x_{p(1)} \\ x_{p(2)} \\ x_{p(3)} \\ x_{p(4)} \end{pmatrix} \\ &= \begin{pmatrix} x_2 \\ x_4 \\ x_1 \\ x_3 \end{pmatrix} \\ &= \begin{pmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{pmatrix} \\ &= PX \end{aligned}$$

Note that premultiplication of data X by a permutation matrix P, PX, permutes the rows of the matrix, while postmultiplication by the transpose, XP^T, will permute the columns of the matrix.

It is easy to notice that permutation matrices obey the following constraints:

$$\sum_{i=1}^{n} P_{ij} = 1 \qquad\qquad j = 1, \ldots n$$

$$\sum_{j=1}^{n} P_{ij} = 1 \qquad\qquad i = 1, \dots n$$

$$P_{ij} \in \{0, 1\} \qquad\qquad i, j = 1, \dots n$$

Returning to our discussion of the Koopmans-Beckman formulation of the QAP from section 1.1, we see that the formulation is a sum of linear and quadratic terms. This is often written as

$$\min_{P \in S_n} \sum_{k,l=1}^{n} \sum_{i,j=1}^{n} f_{kl} p_{ki} d_{ij} p_{lj} + \sum_{k,i=1}^{n} b_{ki} p_{ki} \tag{1.3}$$

where instead of a_{ki} representing the profit of plant k at location i, we instead have b_{ki} representing the price of building plant k at location i. Also, we have substituted c_{ij} for d_{ij} as this entry is often referred to as the distance matrix.

An equivalent formulation uses the Frobenius inner product between two matrices.

Definition 1.1. The **Frobenius inner product** $\langle\ ,\ \rangle$ between two matrices $A, B \in \mathbb{R}^{m \times n}$ is defined to be

$$\langle A, B \rangle = \sum_{i}^{m} \sum_{j}^{n} a_{ij} b_{ij}$$

where a_{ij} and b_{ij} are the entries of matrices A and B, respectively.

In order to move from indices to matrices, we note

$$\sum_{i,j=1}^{n} p_{ki} d_{ij} p_{lj} = PDP^T$$

With that in mind, we can write (1.3) as follows:

$$\min_{P \in S_n} \langle F, PDP^T \rangle + \langle B, P \rangle \tag{1.4}$$

Another popular method of notating this formulation of the QAP absorbs the permutations into subscripts instead of as matrix multiplication.

$$\min_{p \in S_n} \sum_{i,j=1}^{n} f_{ij} d_{p(i)p(j)} + \sum_{i=1}^{n} b_{ip(i)} \tag{1.5}$$

This notation emphasizes the costs inherent in the problem: $f_{ij}d_{p(i)p(j)}$ represents the flow-distance cost from assigning plant i to location $p(i)$ and plant j to location $p(j)$, while $b_{ip(i)}$ represents the location cost of assigning plant i to location $p(i)$.

1.1.2 Trace Formulation

The inner product formulation of the QAP is very similar to the following formulation using the matrix trace

Definition 1.2. Matrix Trace The trace of a matrix $A \in \mathbb{R}^{n \times n}$ is the sum of the diagonal values.

$$\operatorname{tr}(A) = \sum_{i=1}^{n} a_{ii}$$

This was first used by Edwards [42, 41]. If $A, B \in \mathbb{R}^{m \times n}$, then the Frobenius inner product and matrix trace are related as follows

$$\begin{aligned} \langle A, B \rangle &= \sum_{i=1}^{m} \sum_{j=1}^{n} a_{ij} b_{ij} \\ &= \sum_{i=1}^{n} (AB^T)_{ii} \\ &= \operatorname{tr}(AB^T) \end{aligned}$$

From here it is easy to see that (1.4) can be written using the trace as well:

$$\langle F, PDP^T \rangle + \langle B, P \rangle = \text{tr}(FPD^T P^T) + \text{tr}(BP^T) \tag{1.6}$$

$$= \text{tr}(FPDP^T) + \text{tr}(BP^T) \tag{1.7}$$

The last equality is true when D is symmetric, which is usually the case and will always be true in the applications found later in the thesis.

Before moving forward, there are a few useful facts about the trace of a matrix to remember. The trace possesses symmetric invariance for $A \in \mathbb{R}^{n \times n}$,

$$\text{tr}(A) = \text{tr}(A^T) \tag{1.8}$$

linearity for $\alpha, \beta \in \mathbb{R}$ and $A, B \in \mathbb{R}^{m \times n}$,

$$\text{tr}(\alpha A + \beta B) = \sum_{i=1}^{n} (\alpha a_{ii} + \beta b_{ii}) = \alpha \sum_{i=1}^{n} a_{ii} + \beta \sum_{i=1}^{n} b_{ii} = \alpha \text{tr}(A) + \beta \text{tr}(B) \tag{1.9}$$

and finally cyclic invariance for $A \in \mathbb{R}m \times n$ and $B \in \mathbb{R}^{n \times m}$

$$\text{tr}(AB) = \text{tr}(BA) \tag{1.10}$$

We show below that if D or F is symmetric, then we can make a substitution in order to assume they are both symmetric. Assume $D = D^T$ and $F \neq F^T$.

$$\text{tr}(FPDP^T) = \frac{1}{2}\text{tr}(FP(D + D^T)P^T) \tag{1.11}$$

$$= \frac{1}{2}\text{tr}(FPDP^T) + \frac{1}{2}\text{tr}(FPD^T P^T) \tag{1.12}$$

$$= \frac{1}{2}\text{tr}(FPDP^T) + \frac{1}{2}\text{tr}(F^T PDP^T) \tag{1.13}$$

$$= \text{tr}(\frac{1}{2}(F + F^T)PDP^T) \tag{1.14}$$

where the equality from (1.12) to (1.13) is due to the cyclical and symmetric invariance of the trace. Thus, if $D = D^T$, we can replace F with its symmetric projection, $G = \frac{1}{2}(F + F^T)$, and formulate the QAP with two symmetric matrices, G and D.

For the bulk of this thesis, we will use the trace formulation of the QAP as it lends itself well to calculus techniques as well as other computations. However, we will now conclude our survey with one final distinct QAP formulation.

1.1.3 Kronecker Product

This final formulation of the QAP generalizes the Koopmans-Beckmann QAP very slightly, allowing for additional flexibility. As shown by Lawler [84], instead of the cost coming from the product of two different terms $f_{kl}d_{ij}$, this is generalized to a single term c_{kilj}.

$$\min_{X \in S_n} \sum_{k,i=1}^{n} \sum_{l,j} c_{kilj} x_{ki} x_{lj} \tag{1.15}$$

It is easy to see the additional generality of equation (1.15) since the Koopmans-Beckman variant (1.3) is a special case with $c_{kilj} = f_{kl}d_{ij}$ (for purely quadratic terms) and $c_{ijij} = t_{ii}d_{jj} + b_{ij}$ (when linear terms are incorporated). Lawler also formulated (1.15) as a linear assignment problem with a total of n^4 variables and solution of the form $X \in S_{n^2} \subset \mathbb{R}^{n^2 \times n^2}$. In order to formulate the problem in this way, Lawler also introduced an additional constraint, $X = Y \otimes Y$, where $\otimes$ indicates the Kronecker product of two matrices.

Definition 1.3. The **Kronecker product** of matrices $A \in \mathbb{R}^{n \times m}$ and $B \in \mathbb{R}^{k \times l}$,

$A \otimes B$, is

$$A \otimes B = \begin{pmatrix} a_{11}B & a_{12}B & \dots & a_{1m}B \\ a_{21}B & a_{22}B & \dots & a_{2m}B \\ \vdots & & & \vdots \\ a_{n1}B & a_{n2}B & \dots & a_{nm}B \end{pmatrix}$$

Therefore $A \otimes B \in \mathbb{R}^{nk \times ml}$.

Lawler's formulation of the QAP is

$$\min_X \langle C, X \rangle \tag{1.16}$$

$$\text{s.t.} X = Y \otimes Y \tag{1.17}$$

$$Y \in S_n \tag{1.18}$$

where (1.16) is the objective function of a $n^2 \times n^2$ dimension linear assignment problem (LAP) and (1.17)-(1.18) add the additional constraint forcing the decision variable to be a "Kronecker second power" [84]. Lawler used this formulation to compute lower bounds for the QAP, computing the solution to the more efficient LAP and then imposing additional constraints *a posteriori*.

We now continue our discussion of the QAP with an overview of its different applications.

1.2 Applications of the QAP

We divide the discussion of QAP applications into two subsections. First, we discuss applications originally developed for the QAP; second, we discuss other combinatorial problems which may be formulated as QAP.

1.2.1 QAP Applications

The first and original application of the QAP came from [82] and has been outlined above in section (1.1.1). In that application, the authors sought to maximize profit in the assignment of facilities to an equal number of locations; the linear term in the QAP represented the revenue of a given building at a given location and the quadratic term represented the costs associated with transporting goods between different locations and buildings.

A similar problem is presented by Elshafei [44]. Elshafei considered a hospital in Cairo suffering from overcrowding which sought to improve the arrangement of different clinics within the hospital. The hospital wanted to minimize the total travel done by patients; the yearly flow and the distance between facilities is known, thus it is easy to see how these quantities correspond to f_{kl} and d_{ij} respectively in equation (1.3) above.

In [115] Steinberg describes the backboard wiring problem for computers. While this specific application may be obsolete, the general idea is still useful to consider: a computer backboard is to be wired where every element is connected by a given (nonnegative) number of wires to every other element. Additionally, a distance between the elements depending on placement can be measured; Steinberg describes a variety of different metrics to calculate this distance. The problem is to minimize the total amount of wiring on the backboard under the given metric by assigning elements to locations on the backboard. Since this is an assignment problem where the cost of placement for every element depends on the placement of every other element,

this is another example of the QAP.

One special case of the QAP occurs when the nonzero entries in the flow matrix make up a tree (an undirected, acyclic connected graph) and the distance matrix is complete. This case is examined in [32] in the context of intermachine transportation costs on an assembly line. Clearly assembly lines tend to be acyclic and connected while the choice of arrangement of the machines can make a significant difference in efficiency, hence this is another example of a QAP.

A different type of problem arises in [24], where the arrangement of keys on a stenotype machine is examined to improve the efficiency and speed of the user. The distance matrix corresponds to the typing time of the letters of an average stenotypist while the flow matrix corresponds to the frequency of pairs of letters occuring in different languages.

Many applications of the QAP can be described as "layout" problems. In [83] a general description of such problems is given, with the necessary assumptions made in order to formulate a problem as a "layout" problem. As with [44], the authors were approached initially to find a better planning system aimed at improving the design and construction of hospitals.

The applications later in this thesis, which still involve the "layout" of mathematical objects, tend to be more abstract than those presented in the preceding paragraphs.

1.2.2 Combinatorial Problems as QAP

There are many other well known combinatorial problems that can be treated as special cases of the QAP. Each of these have specialized algorithms that are much more effective at solving them, but it is worthwhile to examine how they are related to the QAP.

One of the best known and most widely studied combinatorial optimization problems is the traveling salesperson problem (TSP) [35, 71, 88, 103, 43]. The problem is to find an optimal tour between a collection of cities that visits each city exactly once and minimizes the total distance traveled. To formulate the TSP as a QAP, we leave the distance matrix D the same (as it represents distance in both cases) and we modify the flow matrix. The flow matrix F becomes the adjacency matrix of a cycle on n vertices, i.e. a square matrix with ones on the first super diagonal and the bottom left corner. This represents the restriction of the TSP to visit every location exactly once in a tour. It is well known that the TSP is a strongly NP-hard problem [43].

The linear arrangement problem seeks to find the optimal ordering of the vertices of a graph in order to minimize the total distance between vertices connected by edges on the graph. Also a heavily studied NP-hard problem [102, 3, 111, 62], it too can be formulated as a special case of the QAP. The objective function is

$$\min_{p \in S_n} \sum_{i,j} |p(i) - p(j)| d_{ij} \tag{1.19}$$

where the distance matrix D is the adjacency matrix between the vertices of the given graph, i.e. $d_{ij} = 1$ indicates an edge between vertices i and j. The first term in the

objective function, $|p(i) - p(j)|$ can be represented by the permutation of the flow matrix F, with $f_{ij} = |i - j|$.

The maximum clique [100, 21], graph partitioning [23, 86], and graph packing problems [20, 129] can be formulated as special cases of the QAP, with outlines for these formulations given in [26]. Each are NP-hard problems and thus are difficult in their own right. As the above examples are special cases of the QAP, it is no surprise that the QAP is itself an extremely difficult NP-hard problem. We continue with a discussion of the computational complexity of the QAP.

1.2.3 Computational Complexity of the QAP

The QAP is one of the most computationally difficult combinatorial optimization problems. The largest instances of the QAP without special structure that have been proven to be solved to optimality are $n = 36$, a fact alone which should indicate the difficulty of the problem. Additionally, a popular repository for QAP problem instances, the QAPLIB [27], contains only three instances with $n > 100$, the largest of which is $n = 256$, all with structure built into the problem data. Thus, for the large instances ($n \geq 1000$) that this thesis will discuss, even local optimality is a difficult task.

As the special cases of the QAP above are all NP-hard problems, it should come as no surprise that the general QAP is itself an NP-hard problem. The proof of this is given in [106], where the QAP is converted to a Hamiltonian cycle problem, thus showing that they are equivalent and both NP-hard. Even in the special case

where the distance matrix D obeys the triangle inequality, Arkin [12] demonstrates an algorithm of $O(n^3)$ complexity which finds an approximate solution no less than one fourth of the optimal solution. Hence, even special cases of the QAP require relatively slow algorithms to determine an approximate solution.

1.3 Solution Methods of the QAP

As with any optimization problem, there will be a variety of different strategies used in order to solve the QAP. As mentioned above, the QAP is a very difficult problem so the determination of provably optimal solutions is only tractable for relatively small instances. Thus, heuristics, such as local search, must be considered in order to find relatively good solutions in a reasonable amount of time. This section will review the different types of solution methods and corresponding bounds used to solve the QAP.

1.3.1 Bounds and Exact Methods

As with any problem, a first approach is to reformulate the QAP in order to use known solution methods. Specifically, many attempts to solve the QAP came from reformulating it as a mixed integer linear program (MILP). The linearization process has many different techniques. As described above, Lawler [84] linearized the problem by transforming the QAP into a much larger LAP by examining the products $y_{ijkl} = x_{ik}x_{jl}$, resulting in n^4+n^2 binary variables and n^4+2n^4+1 constraints. Another linearization by Frieze and Yadegar [50] reduces the number of integer variables to $\frac{1}{2}(n^4 - 2n^3 + 3n^2)$ with $n^2 + 2n$ constraints. Finally, the widely used reformulation-

linearization technique (RLT) originally suggested by Sherali and Adams [110, 2] can be applied to the QAP to generate a formulation with more variables and constraints but ultimately a tighter relaxation, giving better computational results.

In order to examine branch-and-bound techniques, first the calculation of bounds for the QAP must be discussed. One type of lower bound comes directly from the MILP methods described above. Any solution to a relaxation of the QAP is immediately a lower bound for the QAP itself, thus tighter relaxations yield better lower bounds. Gilmore [55] and Lawler [84] developed bounds based on the Kronecker product LAP formulation of the QAP (1.16)-(1.17). These are found by solving n^2 LAP subproblems, each of size n, then collecting elements of a modified C matrix into another LAP whose solution is a lower bound on the original QAP. This can be done in $O(n^5)$ time since each LAP can be solved in $O(n^3)$ time. For the Koopmans-Beckmann version, equation (1.3), the computational work can be reduced to $O(n^3)$ [26].

Eigenvalue-based bounds also exist for the QAP, first computed by Finke et al. [48] based on the QAP trace formulation (1.7). Their bound sandwiches the possible values of the purely quadratic portion of the QAP using the eigenvalues of the flow and distance matrices F and D: if $\lambda_1 \leq \cdots \leq \lambda_n$ are the eigenvalues of F and $\mu_1 \leq \cdots \leq \mu_n$ are the eigenvalues of D, then

$$\sum_{i=1}^{n} \lambda_i \mu_{n-i+1} \leq \operatorname{tr}(FPDP^T) \leq \sum_{i=1}^{n} \lambda_i \mu_i \tag{1.20}$$

In order to accommodate a QAP with a linear term, the LAP was solved to

optimality separately and added to these bounds. These bounds can also be found by relaxing the set of permutation matrices S_n to the set of orthogonal matrices O_n. This was shown first in [104].

It is well known that permutation matrices $S_n = O_n \cap D_n \cap N_n$ where

$$O_n := \{X \in \mathbb{R}^{n\times n} | X^T X = I\} \tag{1.21}$$

$$D_n := \{X \in \mathbb{R}^{n\times n} | Xe = X^T e = e\} \tag{1.22}$$

$$N_n := \{X \in \mathbb{R}^{n\times n} | X_{ij} \geq 0 \ \ \forall i, j\} \tag{1.23}$$

Note O_n is the set of orthogonal matrices, D_n the set of matrices with row and column sums equal to one, and N_n the set of entry-wise nonnegative matrices. The bound in (1.20) was improved in [64, 63] by tightening the relaxation to orthogonal matrices with row and column sum equal to one. Hadley [63] extended these results to include cases for nonsymmetric matrices of F and D.

Other types of bounds on the QAP which improve on the above techniques have been found using semidefinite programming [75, 131], quadratic programming [10], and decomposition methods [29, 76]. However, in general, lower bounds tend to become looser as the size of the problem increases, thus beyond relatively modestly-sized instances many bounding techniques lose some usefulness.

Branch-and-bound is a common technique for solving integer or mixed-integer programs, so naturally it has been applied to instances of the QAP. Pardalos et al. [98] apply bounds developed in [87] and use a greedy randomized adaptive search procedure (GRASP, see section 1.3.3.4) to improve on results using the Gilmore-

Lawler bound. Anstreicher and Brixius[11] implemented in a computational grid approach, taking advantage of ideas from [10]. Adams [2] extends the reformulation linearization technique from [110] to dramatically reduce the number of nodes in the branch-and-bound algorithm and solve relatively large instances of the QAP.

Exact solution methods using Bender's decomposition are of limited use for the QAP. While there has been some investigation into using Bender's decomposition for the QAP, it has been applied to small instances of the problem and seems to have limited potential for expansion to larger instances [78, 16, 17]. The convergence in the method is generally too slow with the QAP to be successfully scaled up to even moderate instance sizes.

Similar to Bender's decomposition, branch-and-cut is not heavily used to solve the QAP. Padberg and Rijal [96] as well as Erdogan and Tansel [45] have developed branch-and-cut algorithms for solving the QAP, though are still restricted to relatively modest instance sizes. Much of the work on branch-and-cut algorithms for the QAP involves describing the appropriate polytope [95, 74].

Ultimately, while only exact solution methods are able to guarantee optimality in the final solution, they are not the primary method for solving the QAP. Constrained to $n < 40$ in most cases, with much of the literature focusing on $n \leq 30$, the potential for scaling these techniques is limited barring some significant breakthroughs in the field. Therefore, most of the state of the art QAP solving methods use heuristics to find good (and possibly, though unprovably, optimal) solutions. We discuss this in the following sections.

1.3.2 Local Search

A local search method finds a locally optimal solution to the QAP with respect to a specified neighborhood of solutions. This local solution could be the global solution, but in all likelihood is not. Local search methods are useful because they can improve on an existing solution relatively quickly and thus are used extensively in heuristics. The primary thrust of this thesis is the description of a novel local search algorithm for extremely large instances of the QAP; however before we survey previous local search techniques, we must first describe what is meant by a locally optimal solution for the QAP.

A locally optimal solution means that no other solution in the neighborhood of the current solution is superior. We know that a solution to the QAP is a permutation of size n, so we define what is meant by the neighborhood of a permutation. Probably the most commonly used permutation neighborhood is the *k-swap* neighborhood [119, 51, 4, 34, 108, 116].

Definition 1.4. K-Swap Neighborhood A **k-swap neighbor** of a permutation $\hat{p}$ is any permutation p such that p differs from $\hat{p}$ by at most k indices. The k-swap neighborhood of $\hat{p}$ is the set of its k-swap neighbors. That is,

$$N_k(\hat{p}) = \{p \in S_n | H(p, \hat{p}) \leq k\} \tag{1.24}$$

where $H(p, \hat{p})$ is the Hamming distance. The Hamming distance [65] is the number of coordinates that differ between a pair of permutations.

Since the size of the $k-$swap neighborhood is $\binom{n}{k}$, due to computational difficulties in checking an entire k-swap neighborhood, k is generally restricted to $k = 2$. This 2-swap neighborhood will be referred to simply as the swap neighborhood or the set of swaps.

Another possible neighborhood is the insertion neighborhood, introduced by Ahuja et al. [5]. For this type of neighborhood, a single index is removed and inserted elsewhere in the permutation with all the intervening indices shifting to accomodate the insertion. For example, for $p = [1, 2, 3, 4, 5, 6, 7]$ the third index, 3, could be inserted in the sixth position, yielding the permutation $[1, 2, 4, 5, 6, 3, 7]$. Ahuja also explored possibilities to expand beyond 2-swap neighborhoods through improvement graphs.

The general idea of a local search algorithm is to find an improving solution in the neighborhood of the current solution, making this improving solution the new current solution, and iterating this process until converging to a current solution with no improving neighbors. However, the choice of improving solution is nontrivial. A greedy local search algorithm may search the entire neighborhood to find the neighbor that yields the best improvement while a speed-focused algorithm may search the neighborhood and accept the first neighbor that yields any improvement. It is easy to see how each approach may have advantages and drawbacks: the best improvement rule may take fewer iterations to converge on an optimal solution while the first improvement will require less calculation per iteration due to the limiting the search to only a portion of the entire neighborhood.

The primary advantage of using a neighborhood-based local search technique is due to the fact that feasiblity can be maintained at every point of the search. Many exact optimization strategies (branch-and-bound, Bender's decomposition, branch-and-cut, etc.) do not ensure feasible solutions at every step of the process. Thus, local search algorithms are efficient for finding better (though usually not globally optimal) solutions quickly.

One major advantage to using the 2-swap neighborhood for local search algorithms with the QAP is the improvement for a given swap can be computed fairly easily. As this formula will become relevant later in the thesis, it will be derived here. Since this thesis focuses on the Koopmans-Beckmann version of the QAP, that is the difference formula which will be calculated, though the derivation will start with the inner product form (1.4). We first compute the difference in the linear term followed by the more complicated quadratic term.

Without loss of generality, assume the current solution is the identity matrix. A new solution P corresponding to a swap in the indices k and l will be identical to the identity matrix except $P_{kk} = P_{ll} = 0$ and $P_{kl} = P_{lk} = 1$. Therefore

$$(\Delta P)_{ij} = (P - I)_{ij} = \begin{cases} 0 & i, j \neq k, l \\ 1 & (i, j) \in \{(k, l), (l, k)\} \\ -1 & (i, j) \in \{(k, k), (l, l)\} \end{cases} \tag{1.25}$$

With this we can easily calculate the difference due to the linear component of the QAP:

$$\langle B, P\rangle - \langle B, I\rangle = \langle B, P - I\rangle \tag{1.26}$$

$$= B_{kl} + B_{lk} - B_{kk} - B_{ll} \tag{1.27}$$

We calculate $\langle F, D^*\rangle - \langle F, D\rangle$ where $D^* = PDP^T$ and P remains the permutation corresponding to a swap of the entries k and l. Note that

$$(D^* - D)_{ij} = \begin{cases} 0 & i, j \neq k, l \\ D_{lj} - D_{kj} & i = k,\ j \neq k, l \\ D_{kj} - D_{lj} & i = l,\ j \neq k, l \\ D_{il} - D_{ik} & i \neq k, l\ j = k \\ D_{ik} - D_{il} & i \neq l, k\ j = l \\ D_{ll} - D_{kk} & i, j = k \\ D_{kk} - D_{ll} & i, j = l \\ D_{lk} - D_{kl} & i = k,\ j = l \\ D_{kl} - D_{lk} & i = l,\ j = k \end{cases} \tag{1.28}$$

We start with the difference in the inner product formulation.

$$\begin{aligned} \langle F, D^*\rangle - \langle F, D\rangle &= \langle F, D^* - D\rangle \\ &= \sum_{ij} F_{ij}\hat{D}_{ij} \\ &= \sum_{j\neq k,l} F_{kj}(D_{lj} - D_{kj}) + F_{lj}(D_{kj} - D_{lj}) \end{aligned}$$

$$
\begin{aligned}
&+ \sum_{i \neq k,l} F_{ik}(D_{il} - D_{ik}) + F_{il}(D_{ik} - D_{il}) \\
&+ F_{kk}(D_{ll} - D_{kk}) + F_{ll}(D_{kk} - D_{ll}) \\
&+ F_{kl}(D_{lk} - D_{kl}) + F_{lk}(D_{kl} - D_{lk}) \\
&= \sum_{j \neq k,l} (F_{kj} - F_{lj})(D_{lj} - D_{kj}) \\
&+ \sum_{i \neq k,l} (F_{ik} - F_{il})(D_{il} - D_{ik}) \\
&+ (F_{kk} - F_{ll})(D_{ll} - D_{kk}) + (F_{kl} - F_{lk})(D_{lk} - D_{kl})
\end{aligned}
$$

Recognizing that the index variables i and j can be treated identically above, we combine the expression with (1.27) to arrive at the final expression for the difference formula $\Delta\phi$ for the swap of entries k and l in the solution to the QAP:

$$
\begin{aligned}
\Delta\phi = &\sum_{j \neq k,l} ((F_{kj} - F_{lj})(D_{lj} - D_{kj}) + (F_{jk} - F_{jl})(D_{jl} - D_{jk})) \\
&+ (F_{kk} - F_{ll})(D_{ll} - D_{kk}) + (F_{kl} - F_{lk})(D_{lk} - D_{kl}) \\
&+ B_{kl} + B_{lk} - B_{kk} - B_{ll}
\end{aligned}
\tag{1.29}
$$

Clearly this can be computed in $O(n)$ time, with similar formula found in the literature [53, 119, 51]. In fact Taillard [119] and Frieze [51] go further to show that if all possible swaps in a neighborhood are computed once, instead of recomputing all $\binom{n}{2}$ of them after a swap is performed (an $O(n^3)$ task), it is possible to update the values instead in $O(n^2)$ time. This method is limited by the fact that the updates only hold for those swaps which contain indices that have not been modified; all swaps that include modified indices (k and l in the calculation above) must be recomputed

from scratch. Below we compute this update formula.

Assume that indices p and q are swapped first; we compute the difference due to a subsequent swap in the indices k and l. Denote $\Delta\phi^{pq}$ the difference when k and l are swapped after p and q, while $\Delta\phi$ will continue to denote the difference due to swapping k and l with no prior swap. Immediately we notice that only the summation in (1.29) is affected, as the correction terms have no p or q indices.

$$\Delta\phi^{pq} = \Delta\phi - ((F_{kp} - F_{lp})(D_{lp} - D_{kp}) + (F_{pk} - F_{pl})(D_{pl} - D_{pk}) \tag{1.30}$$

$$+ (F_{kq} - F_{lq})(D_{lq} - D_{kq}) + (F_{qk} - F_{ql})(D_{ql} - D_{qk})) \tag{1.31}$$

$$+ (F_{kp} - F_{lp})(D_{lq} - D_{kq}) + (F_{pk} - F_{pl})(D_{ql} - D_{qk}) \tag{1.32}$$

$$+ (F_{kq} - F_{lq})(D_{lp} - D_{kp}) + (F_{qk} - F_{ql})(D_{pl} - D_{pk}) \tag{1.33}$$

$$= \Delta\phi + (F_{kp} - F_{lp} - F_{kq} + F_{lq})(D_{lq} - D_{kq} - D_{lp} + D_{kp}) \tag{1.34}$$

$$+ (F_{pk} - F_{pl} - F_{qk} + F_{ql})(D_{ql} - D_{qk} - D_{pl} + D_{pk}) \tag{1.35}$$

Above, we subtract (1.30)-(1.31) as they no longer reference the correct indices due to the swap in indices p and q, while we add (1.32)-(1.33) to compensate. It is easy to see looking at the above equation that the difference update $\Delta\phi^{pq}$ can be computed in $O(1)$ time for $O(n^2)$ swaps, thus giving us the $O(n^2)$ update complexity. It bears mentioning again that all swaps where $k, l \in \{p, q\}$ need to be recomputed from scratch, but there are $O(n)$ of those and each takes $O(n)$ time, seen in (1.29), maintaining the $O(n^2)$ complexity.

A local search is completely dependent on the initial solution; two similar

initial solutions can result in local optima that are very different in quality. Hence, it is common to run a local search algorithm multiple times from different initial solutions.

1.3.3 Heuristics

For optimization problems that are intractable for exact methods, heuristic techniques are frequently used. In general, a heuristic does not provide an optimality gap on the solutions it obtains, so the comparison of heuristic quality is usually done between the best found solutions. This section will describe some commonly used heuristic techniques applied to the QAP.

1.3.3.1 Genetic Algorithms

Some of the most widely used heuristic techniques are genetic algorithms (GAs). Work on GAs began in 1954 by Barricelli [14, 13] with books being published by Fraser [49] and Crosby [33] in the 1970s. GAs take inspiration from natural evolution, producing individuals which can generate offspring and mutate, with weaker offspring being removed from the population. The main operations of a GA are mutation, crossover, and selection of a population of solutions. A general GA may look like Algorithm 1.1.

It is important to have an effective method of describing the individuals and their fitness in order to facilitate the steps of the GA. For the QAP the population is a collection of permutations, encoded as we describe in Section 1.1.1.1. The fitness is measured by the objective function of the QAP. For each of the major operators in

Algorithm 1.1 Genetic Algorithm

Input: Fitness function, crossover operator, mutation scheme, replacement strategy

Output: Population of solutions

```
1: Generate random initial population of solutions
2: while Not done do
3:     Select parent solutions
4:     Generate offspring
5:     Replace a portion of next generation with offspring
6:     Mutate solutions in population
7: end while
```

the algorithm, there are a variety of ways to perform them. Below is an overview of the approaches found in [5, 93, 121].

Parent selection can be done in a number of ways. Simplest is just randomly choosing two parents from the existing population. However, many algorithms choose to bias the parent selection towards parents which are more fit, weighting the probability parents are chosen by their rank among the population or directly by their fitness measure itself.

The crux of many GA is the offspring operation, or "reproduction." Once two parents are chosen, their offspring must share traits of each parent. In terms of a GA for the QAP, this means that any indices that occupy the same position in the parents' solution will be passed on to the offspring. The remaining positions must be

filled in with the remaining indices. Depending on the source, this is done randomly [121], through an iterative procedure [5, 93], or via graph-theoretic optimization [5], though there are many other possible approaches.

Diversity in the population is important for GA because if too much diversity is lost, the algorithm will converge prematurely to a suboptimal solution. Mutation is a process by which diversity is introduced into the population. Usually this is done by simply perturbing some proportion of the population in a random method; for a permutation this means choosing a subset of the permutation and randomly permuting those entries. The choice of which individuals to mutate is also generally random, though the fittest individual in the population is typically preserved. Mutation is either done immediately following reproduction of each offspring or after a large number of offspring have been generated and the population has converged.

After one or more offspring solutions are created, the next generation of the population is constructed by replacing current members of the population with offspring. Culling the population is usually done at the reproduction phase of the GA. If the offspring were only added to the population without any sort of culling, the population size would balloon considerably, diluting the efficacy of the GA and slowing the convergence to an optimal solution.

Finally, a local search algorithm can be incorporated at any point in a GA, although usually it is done following reproduction: an offspring is produced then locally optimized. As long as the local search can be done efficiently, this can greatly accelerate the convergence of the GA. Inclusion of a local search into a GA is sometimes

called a "memetic algorithm" [19].

1.3.3.2 Simulated Annealing

Simulated annealing (SA) is another popular heuristic with origins outside mathematics. Annealing is a metallurgy technique where the cooling of a metal is done slowly, allowing for inferior intermediate material, so the final product has minimal defects and increased workability. In the context of optimization, the fundamental idea is that a move or swap in the solution may be accepted - with a probability that is controlled by a temperature parameter - even if it is detrimental to the objective function.

The idea originates with Metropolis et al. [91] when they were designing a method to find the lowest energy state possible for a number of particles. The central idea was to make a random small adjustment to the position of a particle and calculate the change in energy ΔE of the system. If $\Delta E < 0$, the adjustment was accepted; if $\Delta E > 0$, the adjustment was accepted with probability $\exp(-\Delta E/kT)$, a formula from Boltzmann's law, where k is a constant determined by the user and T is the temperature parameter. Thus, as temperature decreases, the chance of accepting a detrimental move is reduced. This is referred to as the "Metropolis algorithm."

First applied to the travelling salesperson problem by Kirkpatrick et al. [79] and Cerny [28], SA has had considerable success solving the QAP as well [128, 108, 92, 25]. The critical elements of a SA algorithm are the neighborhood search, temperature schedule, and termination criterion. A basic SA algorithm is described in Algorithm

1.2.

Algorithm 1.2 Simulated Annealing

Input: Cooling schedule, initial solution

Output: (Locally) optimal solution

1: **while** Not done **do**
2: Select candidate move
3: Calculate ΔE
4: **if** $\Delta E < 0$ **then**
5: Accept move
6: **else if** $\Delta E > 0$ **then**
7: Accept move with probability $\exp(-\Delta E/kT)$, otherwise reject
8: **end if**
9: Update temperature T according to schedule
10: **end while**

Similar to local optimization, there are multiple types of neighborhoods to choose from, but in most applications the simple 2-swap neighborhood is used. Unlike in a best-improvement local optimization procedure, the entire neighborhood will not be examined every iteration in an SA algorithm as this would greatly diminish efficiency. Instead, the candidate swap is often chosen at random from the neighborhood. It is also possible for there to be a pre-selected order to the candidate swaps.

The important aspect of the neighborhood search is that the entire neighborhood is thoroughly examined as the algorithm progresses.

Probably the most important aspect of a SA algorithm is the temperature parameter and its cooling schedule. The schedule is generally weakly decreasing, though some algorithms choose to include sequences of re-heating which briefly increases the temperature parameter. Anily and Federgruen [8] proved that SA is guaranteed to converge to a global optimum if a cooling schedule of $T_k = C/\log(k+2)$ for $k = 0, 1, \ldots$ is used. This is usually too slow for practical application, so often a geometric schedule, $T_{k+1} = \alpha T_k$ with $\alpha \in (0, 1)$, is used instead. Literature contains other options as well, such as intermittant decreases [92] or rational decreases [89].

The termination criterion for SA is has a number of conceptually simple possiblities. Some ways to terminate the algorithm are after a predetermined number of iterations, when the temperature is below a given value, the objective function has reached a desired value, or the frequency of accepted moves has diminshed past a given rate.

1.3.3.3 Tabu Search

In tabu search (TS) heuristic, as a local search through the solution space, attributes of executed moves are recorded to prohibit the search from reversing recent moves for a number of iterations. Glover [57, 58, 59] was among the first to investigate the potential of TS for optimization, with Skorin-Kapov [113, 114] and Taillard [119] among the first to apply the technique to the QAP.

The defining ingredients of TS are the neighborhood structure, the attributes defining the tabu status, the tabu list length, and the aspiration criterion. The behavior and success of TS, possibly more than the GA and SA, is intrinsically linked to the choices made for these ingredients; it is possible with poor choices a TS algorithm could completely fail while the same algorithm tweaked slightly could have considerable success in finding good local or global optima. An overview of TS is given in Algorithm 1.3.

Algorithm 1.3 Tabu search

Input: Tabu list type and length, aspiration criterion, neighborhood, attribute-based definition of tabu status

Output: (Locally) optimal solution

1: **while** Not done **do**
2: Perform best available, non-tabu move or tabu move satisfying the aspiration criterion
3: Record attributes of executed move
4: **end while**

As with GAs, SA, and local search, there are many choices of neighborhood but the most common is the 2-swap neighborhood. The advantages for TS are similar to other heuristics; the steps are easily computable with an efficient update formula and TS is empirically effective. Unlike in SA, usually a TS algorithm will search the

entire neighborhood and the best possible move is performed. However, if there are no moves which improve the objective function, either due to the current solution being located at a local optimum or all improving moves are currently in the tabu list, the least detrimental move is chosen. Additionally, for TS a simple neighborhood facilitates the maintenance of the tabu list, reinforcing the choice to use the 2-swap neighborhood.

The type and length of the tabu list are important decisions to be made when designing a TS algorithm. Taillard [119] describes a technique where instead of storing the entire solution, only the positions and indices involved in the swap itself are recorded, preventing only those indices from taking those locations again. This way, each entry in the tabu list prohibits a larger class of possible solutions and encourages more diversity in the search space.

The length of the tabu list is also crucial to the performance of the TS algorithm. Too short and the solution will cycle, preventing any progress. Too long and the search can be unnecessarily restricted and high quality solutions will not be obtained. Unfortunately, list length appears to need to be tuned based on the each problem instance. Taillard [119] describes an instance where cycling is observed with list length 30 but not 26-29; to avoid cycling, Taillard makes the list length variable within the algorithm. In [15], Battiti and Tecchiolli design their list size to react to the frequency of improvement of the best found solution.

Finally, the aspiration criterion for TS determines when a solution may overcome its presence on the tabu list and be accepted. The trivial aspiration criterion

is accepting the best move in the neighborhood if it produces a better solution than the best solution obtained in the current search history. Taillard [119] modifies this by adding an additional diversity requirement to new solutions in order to avoid premature convergence or insufficient diversity of solutions.

TS lends itself well to parallelization and thus additional computational efficiency [34, 114, 73]. Additionally, TS is occasionally incorporated as a intermediate step within a larger scheme, for example getting paired with SA [92] or GA [38]. Comparisons of the previous three techniques can be found in [107, 108, 94].

1.3.3.4 Other Techniques

There are many other types of heuristics that have been applied to the QAP that are too numerous to describe in detail here. The following heuristics are the most next most frequently encountered in the literature.

Ant colony optimization (ACO) uses the idea of previous solutions as pheremones to guide the current solutions (ants). The pheremones are updated as new solutions are produced, encouraging the ants to search more carefully in the area of the current best solution. As with any heuristic, diversification in the pheremones (the search space) must occur occasionally to prevent premature convergence to a suboptimal solution. Ant colony optimization as applied to the QAP is examined in [120, 53] with the origins of ACO in Dorigo [40, 39].

Greedy randomized adaptive search procedures (GRASP) are constructive multi-start algorithms which are designed to find good solutions very quickly. Origi-

nally designed for the set covering problem [47], GRASP has been adapted to many different applications, including the QAP [99, 97]. The main idea is to greedily construct an initial solution by starting with a pair of indices and adding index by index to the solution. To ensure the process doesn't return the same solution every time, there is an element of randomness to the indices added. Finally, a local search procedure is done after the solution is constructed in order to further improve the quality. Since each iteration of GRASP has no effect on another, it is a prime candidate for parallelization.

Iterated local search (ILS) is the final heuristic to be described here, though this is not an exhaustive investigation of all possible heuristics. An detailed overview of the technique by Stutzle and Ruiz can be found in [117] with application to the QAP by Stutzle [116]. Benlic and Hao developed the similar breakout local search for the QAP [18]. ILS starts with a solution, performs a local search, and (depending on the implementation), either perturbs the initial solution or the local optimum and repeats the local search. This allows a better quality in solutions than a simple repeated local search with different starting points.

This concludes the description of background material into the QAP and finishes chapter 1. In the following chapter, a new local optimization technique for very large scale QAP, GradSwaps, will be described in detail. The chapter will include theoretic framework for the technique followed by examples of implementation.

CHAPTER 2
GRADSWAPS

This chapter will describe a novel algorithm designed to quickly find local minima for very large instances of the quadratic assignment problem. We call this algorithm GradSwaps (GS), as it is based off the calculation of the gradient of the objective function. The theory of GS is first developed, followed by demonstrations of its effectiveness on very large ($n > 1000$) instances of the QAP.

2.1 General Description

In this section we give a description of the GS algorithm with theoretical background. At the highest level, GS uses the first order approximation (FOA) of every possible swap in the 2-swap neighborhood to predict which swaps would be beneficial to the objective function. Next, the candidate swaps are collected and swaps with overlapping indices are removed so there are no duplicate indices in the potential swaps. Finally, all remaining swaps are performed at once, instead of the usual method of a single swap per iteration. A high level overview for GS is given in algorithm 2.1

To find the first order approximation we first need a gradient and thus we need an objective function. We define our objective function as the trace version of the QAP (1.7). Let $F, D, B \in \mathbb{R}^{n\times n}$

$$\phi(P) = \operatorname{tr}\left(FPDP^T\right) + \operatorname{tr}\left(BP^T\right) \tag{2.1}$$

Note this objective function does not require a permutation matrix as its argument;

Algorithm 2.1 High Level GradSwaps

Input: Initial solution P

Output: (Locally) optimal solution P^*

1: **while** No convergence **do**

2: Calculate $\nabla\phi(P)$

3: Calculate first order approximation

4: Identify candidate improving swaps

5: Remove all swaps with redundant indices

6: Perform remaining swaps on current solution

7: **end while**

in fact, P only needs to be conformable with F, D, and B, thus $P \in \mathbb{R}^{n\times n}$. Since we have no restriction on P, $\phi(P)$ is a differentiable function.

With the objective function defined, the next step is to calculate the gradient. The process below follows the techniques described in [90].

$$\begin{aligned} \mathrm{d}\phi(P) &= \ \mathrm{dtr}\left(FPDP^T\right) + \ \mathrm{dtr}\left(BP^T\right) && (2.2)\\ &= \mathrm{tr}\left(F\ \mathrm{d}PDP^T + FPD\ \mathrm{d}P^T + B\ \mathrm{d}P^T\right) && (2.3)\\ &= \mathrm{tr}\left(F^TPD^T\ \mathrm{d}P^T + FPD\ \mathrm{d}P^T + B\ \mathrm{d}P^T\right) && (2.4)\\ &= \mathrm{tr}\left((F^TPD^T + FPD + B)\ \mathrm{d}P^T\right) && (2.5)\\ &= \langle F^TPD^T + FPD + B,\ \mathrm{d}P\rangle && (2.6) \end{aligned}$$

Therefore, from the first identification theorem in [90], the gradient is

$$\nabla\phi(P) = F^T P D^T + F P D + B \tag{2.7}$$

In the next section, we describe the theory and calculate the first order approximation for all swaps.

2.1.1 First Order Approximation

As examined in Section 1.3.2, the 2-swap neighborhood is a common choice for local optimization techniques. The difficulty arises in determining which swaps to perform at every iteration of a local search algorithm. Popular choices are the best swap or the first improvement found. However, the process of calculating the exact cost of a swap can be expensive, especially if the change in objective function for all swaps in the neighborhood is calculated. Therefore, we propose that instead of calculating the exact cost for every swap, the first order approximation is used to predict the quality of swaps.

The first order approximation of a general, once-differentiable function $f \in C_1(\mathbb{R})$ about the point a is

$$f(x) \approx f(a) + f'(a)(x - a) = f(a) + f'(a)\Delta x \tag{2.8}$$

where $\Delta x = x - a$, the change in the variable x. Considered for a matrix-argument, scalar function $\phi \in C_1(\mathbb{R}^{n\times n})$ we have a similar expression for the first order approximation about P_0.

$$\phi(P) \approx \phi(P_0) + \langle \nabla\phi(P_0), \Delta P \rangle \tag{2.9}$$

where the inner product $\langle , \rangle$ above is the Frobenius inner product from (1.1). As we have calculated $\nabla\phi$ for our objective function, it remains to calculate ΔP. However, ΔP is the same here as in (1.25) from Section 1.3.2.

With this, we can calculate the first order approximation of the difference in the objective function due to a swap in the indices k and l. Let $G = \nabla\phi(P_0)$.

$$\phi(P) - \phi(P_0) \approx \langle G, \Delta P \rangle \tag{2.10}$$

$$= G_{kl} + G_{lk} - G_{kk} - G_{ll} \tag{2.11}$$

Thus if we have already calculated the gradient $\nabla\phi(P_0)$ to calculate all first order predictions takes $3\binom{n}{2}$ flops. This calculation for all possible swaps can be generalized into matrix form. Let F be the matrix where $F_{ij} = G_{kl} + G_{lk} - G_{kk} - G_{ll}$ as we see above in (2.11). Let $\mathbf{g} \in \mathbb{R}^n$ be the vector of the diagonal entries of $G = \nabla\phi(P_0)$ and $\mathbf{e} \in \mathbb{R}^n$ is the vector of ones.

$$F = G + G^T - \mathbf{e}\mathbf{g}^T - \mathbf{g}\mathbf{e}^T \tag{2.12}$$

This operation will become useful in the future aside from only calculating the first order approximation, so we define it as a function.

Definition 2.1. The **First Order Function** $\Phi(X) : \mathbb{R}^{n\times n} \to \mathbb{R}^{n\times n}$ is defined as follows: let $\mathbf{x} \in \mathbb{R}^n$ be the vector of the diagonal entries of X and $\mathbf{e} \in \mathbb{R}^n$ be the vector of ones. Then

$$\Phi(X) = X + X^T - \mathbf{e}\mathbf{x}^T - \mathbf{x}\mathbf{e}^T$$

is the First Order Function.

We prove a few items about $\Phi(X)$. First, $\Phi(X)$ is symmetric.

$$\begin{aligned}\Phi(X) &= X + X^T - \mathbf{e}\mathbf{x}^T - \mathbf{x}\mathbf{e}^T \\ &= X^T + X - \mathbf{x}\mathbf{e}^T - \mathbf{e}\mathbf{x}^T \\ &= \Phi(X)^T\end{aligned}$$

In addition to the output being symmetric itself, $\Phi(X) = \Phi(X^T)$ since the first two terms always generate a symmetric matrix and $\mathbf{x}$ is the diagonal and thus does not change with the transposition. Finally, we show $\Phi(X)$ is linear. Let $Y \in \mathbb{R}^{n\times n}$ and $\alpha, \beta \in \mathbb{R}$. Let y be the vector of diagonal entries of Y.

$$\begin{aligned}\Phi(\alpha X + \beta Y) &= (\alpha X + \beta Y) + (\alpha X + \beta Y)^T \\ &\quad - \mathbf{e}(\alpha\mathbf{x} + \beta\mathbf{y})^T - (\alpha\mathbf{x} + \beta\mathbf{y})\mathbf{e}^T \\ &= \alpha(X + X^T - \mathbf{e}\mathbf{x}^T - \mathbf{x}\mathbf{e}^T) + \beta(Y + Y^T - \mathbf{e}\mathbf{y}^T - \mathbf{y}\mathbf{e}^T) \\ &= \alpha\Phi(X) + \beta\Phi(Y)\end{aligned}$$

One final note about $\Phi(X)$. Though useful for the definition and proving properties of the first order function, forming the matrices $\mathbf{e}\mathbf{x}^T$ and $\mathbf{x}\mathbf{e}^T$ would be computationally inefficient. Instead see Algorithm 2.2 for a more efficient computational strategy.

The first order function Φ is $O(n^2)$ in complexity. The assignment of x takes $O(n)$ operations, then every entry in the matrix X is modified exactly twice, once during the subtraction step and then once in the addition step. It would be impossible to further improve the complexity as every entry in the matrix must be modified.

Algorithm 2.2 First Order Function Φ

Input: Matrix X

Output: $\Phi(X)$

1: $x \in \mathbb{R}^{n\times 1} \leftarrow$ diagonal entries of X

2: $\hat{X} \leftarrow$ subtract x from every column of X

3: $\Phi(X) \leftarrow \hat{X} + \hat{X}^T$

Having developed the FOA, it is important to assess the quality of the approximation. We investigate this in the following section.

2.1.2 Relation to Exact Swap Difference

For the 2-swap neighborhood, it is natural to wonder exactly how close the FOA is to the exact difference calculation. The calculation of the FOA is computationally faster, but if the FOA is not reasonably close to the exact difference then this approach will not be very effective. In this section we calculate the difference between the exact difference and the FOA, use this calculation to produce a bound on the accuracy of the FOA, and give empirical evidence justifying the use of the FOA.

Recall the exact difference formula (1.29) from Section 1.3.2:

$$\Delta\phi = \sum_{j\neq k,l} ((F_{kj} - F_{lj})(D_{lj} - D_{kj}) + (F_{jk} - F_{jl})(D_{jl} - D_{jk})) \tag{2.13}$$

$$+ (F_{kk} - F_{ll})(D_{ll} - D_{kk}) + (F_{kl} - F_{lk})(D_{lk} - D_{kl}) \tag{2.14}$$

$$+ B_{kl} + B_{lk} - B_{kk} - B_{ll} \tag{2.15}$$

Looking to the future for a simple expression, we rewrite (2.13) using matrix multiplication:

$$(2.13) = (FD^T)_{kl} + (FD^T)_{lk} - (FD^T)_{kk} - (FD^T)_{ll} \tag{2.16}$$

$$+ (F^TD)_{kl} + (F^TD)_{lk} - (F^TD)_{kk} - (F^TD)_{ll} \tag{2.17}$$

$$- (F_{kk} - F_{lk})(D_{lk} - D_{kk}) - (F_{kl} - F_{ll})(D_{ll} - D_{kl}) \tag{2.18}$$

$$- (F_{kk} - F_{kl})(D_{kl} - D_{kk}) - (F_{lk} - F_{ll})(D_{ll} - D_{lk}) \tag{2.19}$$

Note that line (2.18) and (2.19) are correction terms that must be included when rewriting line (2.13) using matrix multiplication since we no longer skip indices k and l. We combine these correction terms with (2.14):

$$(2.14) + (2.18) + (2.19) = (F_{kk} - F_{ll})(D_{ll} - D_{kk}) \tag{2.20}$$

$$+ (F_{kl} - F_{lk})(D_{lk} - D_{kl}) \tag{2.21}$$

$$- (F_{kk} - F_{lk})(D_{lk} - D_{kk}) - (F_{kl} - F_{ll})(D_{ll} - D_{kl}) \tag{2.22}$$

$$- (F_{kk} - F_{kl})(D_{kl} - D_{kk}) - (F_{lk} - F_{ll})(D_{ll} - D_{lk}) \tag{2.23}$$

$$= (F_{kl} + F_{lk} - F_{kk} - F_{ll})(D_{kl} + D_{lk} - D_{kk} - D_{ll}) \tag{2.24}$$

$$\tag{2.25}$$

We are almost to the point where the comparison of the FOA and exact difference calculation is effortless. First, we aggregate the above results for the exact difference in the objective function due to a swap in the indices k and l:

$$\Delta\phi = (FD^T)_{kl} + (FD^T)_{lk} - (FD^T)_{kk} - (FD^T)_{ll} \tag{2.26}$$

$$+ (F^TD)_{kl} + (F^TD)_{lk} - (F^TD)_{kk} - (F^TD)_{ll} \tag{2.27}$$

$$+ (F_{kl} + F_{lk} - F_{kk} - F_{ll})(D_{kl} + D_{lk} - D_{kk} - D_{ll}) \tag{2.28}$$

$$+ B_{kl} + B_{lk} - B_{kk} - B_{ll} \tag{2.29}$$

One difficulty with the inner product formulation (1.4) versus the trace formulation (1.7) is that D in the former corresponds to D^T in the latter. Due to this, D^T in our $\Delta\phi$ above is the same as D in our gradient calculation due to their respective origins in the different formulations. To remedy this, we rename D in (2.29) to be D^T.

Before we continue, we need to define the Hadamard product of matrices.

Definition 2.2. The **Hadamard Product** $A \odot B$ for two matrices of equal size $A, B \in \mathbb{R}^{m \times n}$ is the entry-wise product of the matrices.

$$(A \odot B)_{ij} = A_{ij} B_{ij}$$

With the Hadamard product and the renaming of D in mind, we now have all the tools needed to relate the FOA with the exact difference calculation.

$$\begin{aligned}
\Delta\phi &= (FD)_{kl} + (FD)_{lk} - (FD)_{kk} - (FD)_{ll} \\
&\quad + (F^T D^T)_{kl} + (F^T D^T)_{lk} - (F^T D^T)_{kk} - (F^T D^T)_{ll} \\
&\quad + (F_{kl} + F_{lk} - F_{kk} - F_{ll})(D_{kl} + D_{lk} - D_{kk} - D_{ll}) \\
&\quad + B_{kl} + B_{lk} - B_{kk} - B_{ll} \\
&= \big(\Phi(FD) + \Phi(F^T D^T) + \Phi(F) \odot \Phi(D) + \Phi(B)\big)_{kl} \\
&= \big(\Phi(FD + F^T D^T + B) + \Phi(F) \odot \Phi(D)\big)_{kl}
\end{aligned}$$

The first expression in the final line above looks very familiar: $\nabla\phi(P) = FPD + F^TPD^T + B$. Therefore, by evaluating the gradient at the identity we have our final relation between the FOA and the exact difference:

$$\Delta\phi = \Phi(\nabla\phi) + \Phi(F) \odot \Phi(D) \tag{2.30}$$

Next, we find the exact difference correction through a different method, using calculus.

2.1.3 Second Order Correction

The derivation of the exact difference correction can also be found through improving the first order approximation by calculating the second order approximation. Since the objective function $\phi(P)$ is quadratic, the second order approximation will be exact, thus the second order correction to the FOA is equal to the correction term in (2.30). To find the second order approximation, we must calculate the Hessian matrix of the objective function. Note the size of the Hessian matrix, $H\phi \in \mathbb{R}^{n^2\times n^2}$. We define $H\phi(P)_{ij} = d^2\phi/(d\text{vec}P_i d\text{vec}P_j)$ where $\text{vec}(P)$ is the vectorization of P, found by stacking each column on top of each other. With the Hessian, we can write down the second order approximation.

$$\phi(P) \approx \phi(P_0) + \langle\nabla\phi(P_0), \Delta P\rangle + \frac{1}{2}\langle H\phi(P_0)\text{vec}\Delta P, \text{vec}\Delta P\rangle \tag{2.31}$$

This is the same as the first order approximation (2.9) with the quadratic term added at the end. Thus, this quadratic term should be the same correction as in (2.30). To verify this, we first calculate the Hessian matrix of $\phi(P)$. Beginning

toward the end of the gradient derivation (2.4), again we follow the technique from [90] and apply the d operator a second time.

$$\mathrm{d}^2\phi(P) = \ \mathrm{dtr}\left(F^T P D^T \ \mathrm{d}P^T + FPD \ \mathrm{d}P^T + B \ \mathrm{d}P^T\right) \tag{2.32}$$

$$= \mathrm{tr}\left(F^T \ \mathrm{d}PD^T \ \mathrm{d}P^T + F \ \mathrm{d}PD \ \mathrm{d}P^T\right) \tag{2.33}$$

From the second identification table in Chapter 10 of [90], there is the statement

$$\mathrm{d}^2\phi(P) = \mathrm{tr}\left(B \ \mathrm{d}PC \ \mathrm{d}P^T\right) \longleftrightarrow H\phi(P) = \frac{1}{2}\left(C^T \otimes B + C \otimes B^T\right) \tag{2.34}$$

Using this statement, we are able to identify our Hessian matrix for $\phi(P)$.

$$H\phi(P) = D \otimes F^T + D^T \otimes F \tag{2.35}$$

From here we wish to calculate the second order correction from (2.31). Recall the definition of ΔP corresponding to a swap of indices k and l from equation (1.25):

$$(\Delta P)_{ij} = (P - I)_{ij} = \begin{cases} 0 & i, j \neq k, l \\ 1 & (i, j) \in \{(k, l), (l, k)\} \\ -1 & (i, j) \in \{(k, k), (l, l)\} \end{cases} \tag{2.36}$$

Note $\Delta P = (\Delta P)^T$. Paired with the fact that

$$\mathrm{tr}\,(ABCD) = (\mathrm{vec}D^T)^T(C^T \otimes A)\mathrm{vec}B \tag{2.37}$$

(also from [90]), we evaluate the second order correction.

$$\frac{1}{2}\langle H\phi(P_0)\text{vec}\Delta P, \text{vec}\Delta P\rangle = \frac{1}{2}(\text{vec}\Delta P)^T H\phi(P_0)\text{vec}\Delta P \tag{2.38}$$

$$= \frac{1}{2}(\text{vec}\Delta P)^T (D \otimes F^T + D^T \otimes F)\text{vec}\Delta P \tag{2.39}$$

$$= \frac{1}{2}\text{tr}\left(F^T \Delta P D^T \Delta P + F\Delta P D \Delta P\right) \tag{2.40}$$

$$= \text{tr}\left(F\Delta P D \Delta P\right) \tag{2.41}$$

$$= \sum_{i=1}^{n}\sum_{j,p,q} F_{ij}\Delta P_{jp} D_{pq} \Delta P_{qi} \tag{2.42}$$

$$= F_{kk}(D_{kk} - D_{kl} - D_{lk} + D_{ll}) + F_{kl}(-D_{kk} + D_{kl} + D_{lk} - D_{ll}) \tag{2.43}$$

$$+ F_{lk}(-D_{kk} + D_{kl} + D_{lk} - D_{ll}) + F_{ll}(D_{kk} - D_{kl} - D_{lk} + D_{ll}) \tag{2.44}$$

$$= \Phi(F) \odot \Phi(D) \tag{2.45}$$

As expected, the second order correction to the FOA is the same as the exact difference correction in (2.30). Now that we have calculated the correction to the FOA, we will evaluate the accuracy of the FOA compared to the exact difference.

2.1.4 Accuracy of the FOA

From (2.30) we see the $\Delta\phi - \Phi(\nabla\phi) = \Phi(F) \odot \Phi(D)$. It may seem computationally inconsequential to calculate this correction to the FOA, but we will see that for low-rank F and D the gradient can be computed very efficiently without an explicit representation of either matrix; therefore the computation of the correction is in fact a substantial burden. To justify this computational shortcut, section demon-

strates the accuracy of the FOA by deriving a formal error bound. This error bound will later be incorporated into the GS algorithm.

As we wish to bound the error in an individual swap and each entry of the FOA corresponds to a swap, we find our bound using the matrix max norm.

Definition 2.3. The **matrix max norm**, $\|\cdot\|_{\max}$ of a matrix $A \in \mathbb{R}^{m\times n}$ is the maximum absolute value over the entries of the matrix.

$$\|A\|_{\max} = \max_{i,j} |A_{ij}|$$

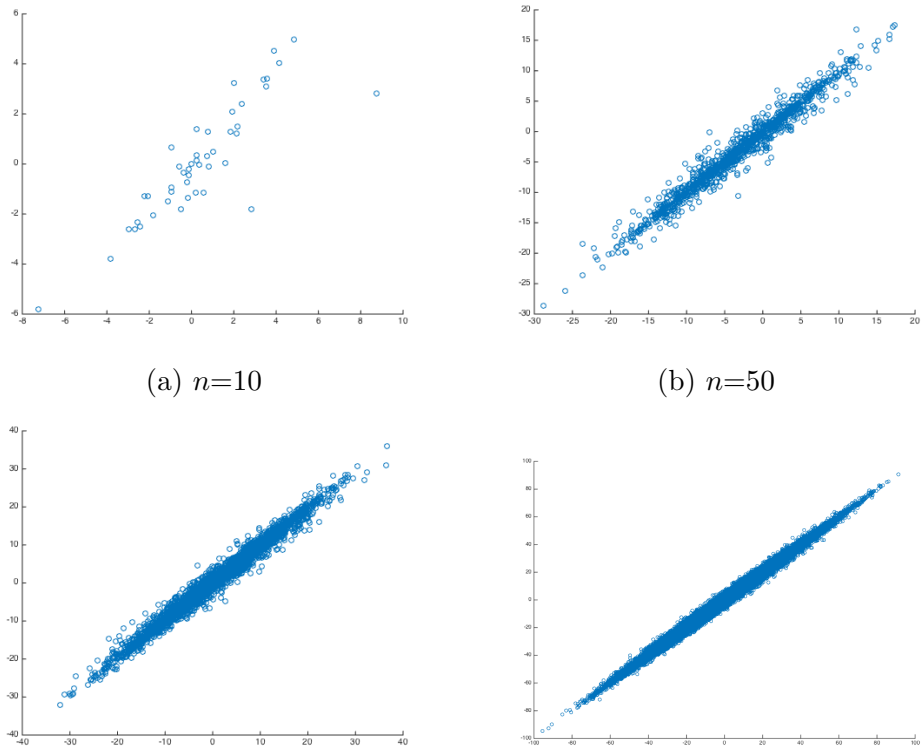
We calculate the bound below.

$$\|\Delta\phi - \Phi(\nabla\phi)\|_{\max} = \|\Phi(F) \odot \Phi(D)\|_{\max} \tag{2.46}$$

$$\leq 4\|F\|_{\max} \cdot 4\|D\|_{\max} = 16\|F\|_{\max}\|D\|_{\max} \tag{2.47}$$

Notice that while (2.47) scales with the entries of F and D, it does not scale with the size of the matrices; however, the values of the FOA scale linearly with the number of rows of the matrices. Therefore for extremely large instance of the QAP, this error bound will become insignificant in comparison to the value of the FOA itself. Additionally, if the entries of D or F are nonnegative or the matrices have a null diagonal this error bound tightens to $4\|F\|_{\max}\|D\|_{\max}$ as the subtracted terms from Φ can be ignored and the factor of 4 from each evaluation of Φ can be reduced to 2.

In Figure 2.1 we present four scatter plots with the exact difference on the vertical axis and the FOA on the horizontal axis. Data matrices F and D were generated randomly from a uniform $[-1, 1]$ distribution and the four plots correspond



(a) n=10 (b) n=50

(c) n=100 (d) n=500

Exact Difference on Vertical Axis, FOA on Horizontal Axis

Figure 2.1: Accuracy of the First Order Approximation

to different sizes of the problem. It is easy to see that the two values are highly correlated; this validates the concept of using the FOA instead of the exact difference due to the reduction in computation required to calculate the FOA. Additionally, the most extreme values for the FOA tend to be amongst the most extreme for the exact difference, indicating that a greedy local search method using FOA instead of exact differences would return similar, if not identical, results.

Figure 2.2 shows the proportion of the total number of FOA predictions with

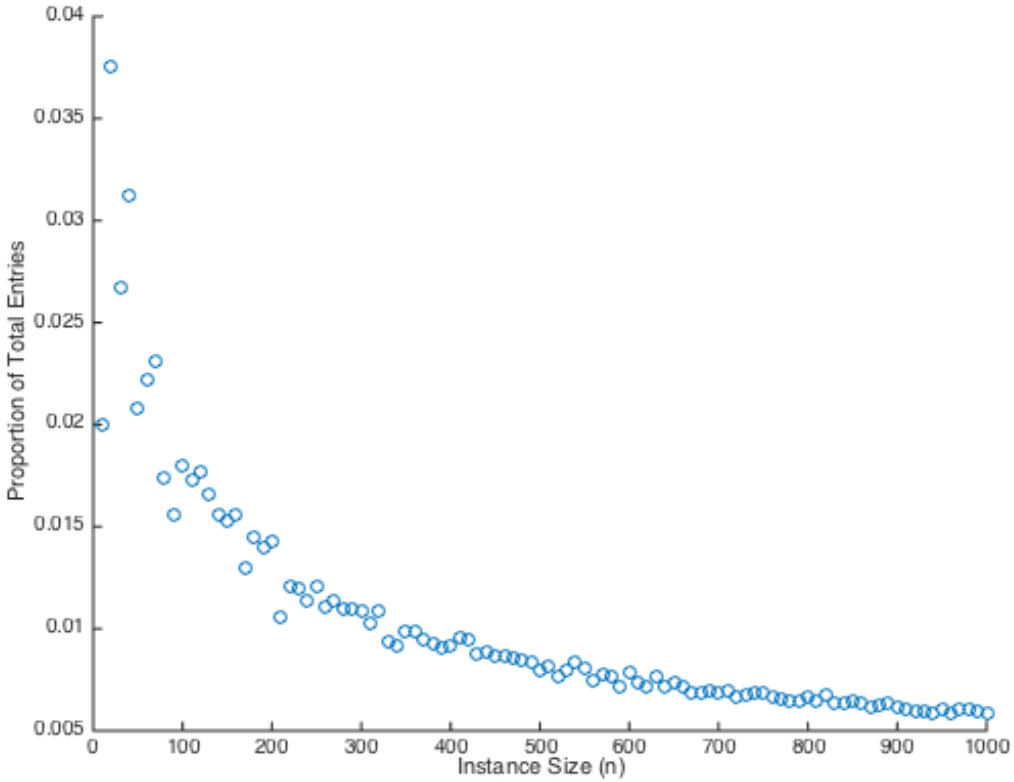


Figure 2.2: Wrong Predictions by Instance Size

the incorrect sign over the total number of predictions. While error in the magnitude of a FOA prediction is a concern, a sign error is more important as an incorrect sign could cause a favorable swap to be designated unfavorable or vice-versa. One can see that as the size of the instance increases, the proportion of sign errors diminishes rapidly. Also note that those swaps which may have the wrong sign likely are relatively small in magnitude (see Figure 2.1) and thus would be unlikely to be chosen in any sort of greedy algorithm.

Now that the use of the FOA has been justified, we turn to additional ideas

to speed up the local optimization algorithm.

2.1.5 Swap Selection

Another defining feature of the GS algorithm is the fact that multiple swaps are performed every iteration instead of a single swap as done by most greedy local search algorithms. This is due to the increased computational efficiency that is inherent in this method. There are a number of considerations in choosing which swaps to perform based on the FOA.

Note that every swap performed modifies the FOA of the other swaps; we choose to ignore this relatively minor effect at each iteration as this collective error tends to be insignificant in practice. However, if the number of indices swapped per iteration increases, this minor effect can compound to the point that the FOA is no longer accurate. Thus any method to choose the swaps must take this into consideration.

One way to choose which swaps to perform is to formulate the decision as a set packing problem. Given a collection of subsets $\mathcal{S}$ of a universe $\mathcal{U}$, the goal of the maximum set packing problem is to maximize the number of subsets with no overlap between the selected subsets. This problem has been studied extensively [54, 67, 118] and is known to be a NP-complete problem [77]. Since our goal is not to make as many swaps as possible but instead improve the objective function as much as possible, our formulation will be as a weighted 2-set packing problem, where the 2 refers to the

largest possible subset in $\mathcal{S}$. We formulate this problem as a integer linear program.

$$\max \sum_{S \in \mathcal{S}} w(S) x_S \tag{2.48}$$

$$\text{subject to } \sum_{S : e \in S} x_S \leq 1 \quad \forall e \in \mathcal{U} \tag{2.49}$$

$$x_S \in \{0, 1\} \quad \forall S \in \mathcal{S} \tag{2.50}$$

It remains to describe $\mathcal{U}$, the subsets S, and the weights $w(S)$. The universe of elements $\mathcal{U}$ is the indices of the solution, the integers 1 to n. The 2-sets are the two indices in a given swap with weights corresponding to the FOA approximation for the swap. Also included are 1-sets, which indicate a given index is not changed from one iteration to the next and thus have a weight of 0.

There are a couple of problems if we choose the swaps to perform via solving the above integer program. First, without an additional constraint on the total number of swaps performed, it is conceivable too many swaps will be done and the FOA loses its accuracy. Second, as we are already using a heuristic prediction (the FOA), exerting extra computational effort to compute the best FOA swaps to use is unnecessary. Thus, to overcome these problems we propose a simple greedy heuristic in Algorithm 2.3. As the QAP is traditionally formulated as a minimization problem the algorithm is designed accordingly; lines 2 and 4 are the only steps that would need modification for a maximization-based QAP.

First, we take only the upper triangular portion of $\Phi(\nabla\phi)$ since the (i, j) entry corresponds to the same swap as the (j, i) entry, thus reducing the search space of candidate swaps in half. Next, we limit the candidate swaps to those that are less

Algorithm 2.3 SwapSelect Algorithm

Input: FOA $\Phi(\nabla\phi)$, $S_f = \emptyset$

Output: Collection of chosen swaps S_f

1: $U \leftarrow$ upper triangular portion of $\Phi(\nabla\phi)$

2: $S \leftarrow \{(i,j)|\ U_{ij} \leq -16\|F\|_{\max}\|D\|_{\max}\}$

3: $W \leftarrow$ weights (entries of U) corresponding to the swaps S

4: $W, S \leftarrow W$ sorted in increasing order, S reordered accordingly

5: **for** $k = 1 \ldots$ max swaps **do**

6: **if** $i_k \notin S_f$ and $j_k \notin S_f$, $(i_k, j_k) = S_k$ **then**

7: $S_f \leftarrow S_f \cup (i_k, j_k)$

8: **end if**

9: **end for**

than the error bound between exact differences and the FOA (2.47). This removes any possible weights with incorrect signs as well as further reducing the search space. Finally, before any swaps are actually chosen, the weights are sorted in increasing order with the swaps similarly reordered.

Since we have already sorted the candidate swaps, each swap will be examined only once during the selection process as no further comparison in quality must be done. Thus at each iteration, we add the next candidate swap to our list of chosen swaps if it doesn't overlap with an already chosen swap. Note that the check for overlapping indices can be done by keeping a binary n-vector, with 1 indicating existence of a given index in a chosen swap and 0 indicating its absence; this way it is unnecessary to look through all chosen swaps for potential overlap with a candidate swap. Finally, this process continues until the maximum number of chosen swaps have been found, at which point the algorithm terminates and returns the list of chosen swaps.

We show the computational complexity of Algorithm 2.3 is $O(n^2 \log(n))$. If $F, D \in \mathbb{R}^{n \times n}$, then $\Phi(\nabla\phi) \in \mathbb{R}^{n \times n}$ as well, with n^2 total entries. In the worst case scenario, after steps 1 and 2 there still remain $n^2/2$ entries to be sorted in step 4, resulting in $O(n^2 \log(n))$ complexity. The loop complexity is proportional to the number of sorted entries, $O(n^2)$, as in the worst case scenario every entry is examined is the maximum number of swaps is never attained.

Even with the inclusion of a maximum number of possible chosen swaps, there is still the potential that the minor adjustments to the FOA from each swap accumulate and cause the objective function to increase once all chosen swaps are performed.

To combat this, a simple backtracking strategy is implemented: if the objective function increases upon performing all the chosen swaps, reduce the number of chosen swaps by a factor of $\alpha \in (0, 1)$ and re-perform the swaps on the original solution. This can be done repeatedly if necessary, though if the number of chosen swaps becomes very small relative to the size of the problem, the current solution is likely approaching a local optimum.

With the description of Algorithm 2.3, we are now prepared to give a complete description of the GS algorithm.

2.1.6 Detailed GS algorithm

In this section, we synthesize all the elements of the GS algorithm into an effective and efficient local search algorithm for the QAP. We give a detailed overview in Algorithm 2.4. Note that this algorithm assumes the QAP is strictly quadratic, i.e. there is no linear term. We will see that this assumption is valid for all proposed applications, though the adaptation to include the linear term is conceptually and algorithmically simple. The remainder of this section will carefully describe those steps not previously discussed.

One of the most important implementation steps in the entire algorithm is found in steps 1 and 14. It would be tremendously wasteful of computational time and memory to explicitly represent the permutation matrix P and perform the matrix multiplication PDP^T. Instead, P can be stored as a vector which we use to reorder the rows and columns of the matrix rather than performing matrix multiplication.

Algorithm 2.4 GradSwaps

Input: Data matrices F, D, Initial solution P_0, Convergence Criterion, Max swaps per iteration, Backtracking parameter α

Output: Locally optimal solution P^*

$D \leftarrow P_0 D P_0^T$

$f \leftarrow \phi(I)$

$P^* \leftarrow P_0$

while Convergence criterion not attained **do**

  $G \leftarrow \nabla\phi(I) = FD + F^T D^T$

  $A \leftarrow \Phi(G)$

  $S \leftarrow \text{SwapSelect}(A)$

  $P \leftarrow$ permutation corresponding to swaps S

  while $\phi(P) > f$ **do**

    $l \leftarrow \alpha \cdot \text{length}(S)$

    $S \leftarrow \{S_1 \dots S_l\}$

    $P \leftarrow$ permutation corresponding to swaps S

  end while

  $D \leftarrow PDP^T$

  $f \leftarrow \phi(I)$

  $P^* \leftarrow PP^*$

end while

Recall that pre-multiplication, PD, permutes rows of D while post-multiplication by the transpose, DP^T, permutes columns; thus by reordering the rows and columns of D instead of multiplying we save an $O(n^3)$ operation every iteration.

Steps 1-3 use the initial solution P_0 to reorder D, determine the initial objective value f, and record the current solution P^*. Note that throughout the algorithm, we reorder the matrix D according to the current solution (steps 1 and 14). Therefore, we need to assign the initial objective value using the identity permutation in step 2, otherwise we would apply the initial permutation twice.

The convergence criterion has, until now, been largely ignored. There are a few different choices for this, with the most obvious being to simply stop the algorithm when no candidate swaps exceed the error bound calculated in (2.47). This takes too long to converge, often with a small number of swaps performed each iteration. In practice, it is much more effective to stop the algorithm when the relative change in objective function, $|(f_k - f_{k-1})/f_k|$, is below a given threshold. This value must be modified depending on the data, but a value of 10^{-5} has proven to be a reasonable threshold for convergence in many cases.

Steps 5-7 compute the gradient G, the first order approximation A, and select the candidate swaps S as discussed in Section 2.1.5. Step 8 takes the selected swaps and records them as a single permutation.

As discussed above, even if every selected swap corresponds to an improvement in the objective function, when performed together they may not have the desired effect due to the accumulated minor adjustments of the selected swaps on the FOA.

Thus we incorporate a simple backtracking technique in steps 9-12. If there is no improvement in the objective function, a set proportion (1-α) of the selected swaps are removed and the objective function is re-computed with the more limited set of swaps. This process can be repeated as necessary. Note that in practice $\alpha = .5$ is a good starting choice, though this can be tweaked according to application.

The final steps 14-16 are similar to the initial steps as they permute the data matrix D, calculate the new objective value (though in practice the value is already known from the inner while loop), and update the current solution.

One significant factor that has yet to be mentioned is the benefit of symmetric, low-rank, and positive semi-definite data matrices. If D and F are positive semi-definite (all eigenvalues ≥ 0), then they have a Cholesky decomposition.

Definition 2.4. If $A \in \mathbb{R}^{n \times n}$ is a symmetric positive (semi-)definite matrix, it has a **Cholesky decomposition**

$$A = CC^T$$

where $C \in \mathbb{R}^{n \times n}$ is a lower triangular matrix. If A has rank $r < n$, then $C \in \mathbb{R}^{n \times r}$ can be found instead.

If $F = CC^T$ is rank r and $D = BB^T$ is rank s, then by performing the Cholesky factorization (a one time $O(n^3)$ operation) it is possible to reduce the computation involved in evaluating the gradient substantially. First, the gradient calculation (step 5) is immediately faster for symmetric F and D since $G \leftarrow 2FD$ and only a single matrix multiplication needs to be done. However, if we group the multiplication as in

equation (2.51) or (2.52), we can reduce the total computation from $O(n^3)$ to $O(n^2 r)$ or $O(n^2 s)$, respectively.

$$G = 2FD = 2CC^T BB^T = 2C((C^T B)B^T) \tag{2.51}$$

$$= 2(C(C^T B))B^T \tag{2.52}$$

We will see during the discussion of specific applications that often the data matrices come pre-factored and are very low rank ($r, s \ll n$), thus this is a significant computational speedup in the algorithm.

Finally, a broad discussion of the complexity of each iteration of the algorithm is appropriate. The possible bottlenecks in the algorithm are the first order approximation $\Phi(G)$, the gradient evaluation, the function evaluation, the reordering $D \leftarrow PDP^T$, the sorting of swap weights, and the swap selection. As discussed above, $\Phi(G)$ can be computed in $O(n^2)$ time. Implementing the function evaluation we would not use the trace but instead the inner product form of the QAP; this is also an $O(n^2)$ operation. The reordering of D can be done very efficiently, in $O(n)$ time, and it was shown above that the weight sorting and swap selection can be done in $O(n^2 \log(n))$ and $O(n)$ time, respectively. Therefore the bottleneck of the GS algorithm comes from the gradient calculation and weight sorting, hence the total computational complexity of GS is $O(n^2(s + r + \log(n)))$.

Thus far, the development has strictly focused on the 2-swap neighborhood for the local search as this is the most common neighborhood in the literature (see Section 1.3.2) due to its ease of computation and effectiveness in providing good local solutions. In the next section, we examine two other possible neighborhood

structures, the 3-swap neighborhood and the insertion neighborhood.

2.1.7 Other Neighborhoods

The first alternative to the 2-swap neighborhood is the 3-swap neighborhood. It is similar to the 2-swap neighborhood except instead of swapping two indices, three indices can be rearranged. Thus, when indices are selected there are a total of five possible results instead of the one present in the 2-swap neighborhood: for indices i, j, and k originally in the order ijk, the five possible arrangments are ikj, kji, jik, kij, and jki. Note that three of these are 2-swaps and only two are actually novel. Additionally, there are a total of $(n^3 + 3n^2 + 2n)/6$ total combinations of three indices, substantially more than 2-swaps. The number of rearrangements in the s-swap neighborhood is $s! - 1$ and the total combinations is $\binom{n}{s}$, thus the total rearrangements would be $\binom{n}{s}(s! - 1) \sim n^s$ for $n \gg s \gg 1$ and so we are limited to very small s.

In fact, even $s = 3$ is infeasible to use for local optimization. While the statement of the 3-swap neighborhood is relatively simple, the computation is not. For a 3-swap of indices ijk to jki the first order approximation would be $F_{jki} = G_{ki} + G_{ij} + G_{jk} - G_{ii} - G_{jj} - G_{kk}$ where $G = \nabla\phi(I)$. Thus the actual computation of the approximation, at five flops, is only a little more expensive than the 2-swap FOA (three flops). However, as mentioned above, there will be five neighborhoods per combination of three indices and $\binom{n}{3}$ combinations of indices. Thus for even a modest (by the standards of the applications later in the thesis) $n = 1000$, the

computation of all 2-swaps requires $3\binom{1000}{2} \approx 1.50 \cdot 10^6$ flops (an $O(n^2)$ operation), while the computation of all 3-swaps requires $5\binom{1000}{3}(3! - 1) \approx 4.15 \cdot 10^9$ flops (an $O(n^3)$ operation). This is an increase of over 2500 times as much computation for only two more possible types of swaps. Additionally, in computational experiments, these new cyclical 3-swaps did not make any difference in the final quality of the local optimum (Table 2.1). Thus, the 3-swap neighborhood is not remotely useful in local optimization for the QAP as it is an order of n more expensive to compute and the resulting quality is no better.

Another possible neighborhood to discuss is the "insertion" neighborhood introduced by Ahuja in [5]. The idea is to insert index i in the current location of index j and move all intervening indices (including j) to fill in the gaps created. For example, if we have $p = [1, 3, 4, 5, 2, 6]$, an insertion of $3 \to 2$ would result in $\hat{p} = [1, 4, 5, 2, 3, 6]$, where index 3 has taken the location of index 2 and indices 4, 5, and 2 have moved to the left towards the former location of index 3. Immediately, it is a little doubtful this will be as effective as the 2-swap neighborhood since any non-adjacent insertion will disrupt many interceding indices which may be in ideal positions.

There are exactly twice as many possible insertion neighbors as 2-swap neighbors since one chooses a starting and ending location but unlike the 2-swap neighborhood, order now matters. The computation of the first order approximation for inserting index i into position j with $i < j$ is $F_{i\to j} = G_{ji} + \sum_{k=i+1}^{j} G_{k-1,k} - \sum_{k=1}^{j} G_{kk}$. The case where $j < i$ is similar, $F_{i\to j} = G_{ij} + \sum_{k=i+1}^{j} G_{k,k-1} - \sum_{k=1}^{j} G_{kk}$. Worst case, this involves $2n - 1$ flops for the insertion $1 \leftrightarrow n$ so a naive evaluation of the entire

$\epsilon = 10^{-6}$	$n = 150,\ f = 10,\ d = 10$			$n = 150,\ f = 40,\ d = 15$		
Neighborhood	Time	Function	Iterations	Time	Function	Iterations
2-Swap	0.010	0.997	10.4	0.010	0.994	11.5
3-Swap	0.405	0.738	7.8	1.486	0.818	29.8
Insertion	0.042	0.843	16.4	0.057	0.813	22.6
$\epsilon = 10^{-6}$	$n = 1000,\ f = 100,\ d = 100$			$n = 2000,\ f = 200,\ d = 100$		
2-Swap	0.294	0.997	7.6	1.275	0.998	7
Insertion	0.951	0.852	20.9	3.099	0.856	16.9

Function = $(f_{start} - f_{trial})/(f_{start} - f_{best})$, Time in seconds

Table 2.1: Comparison of Three Different Neighborhoods

neighborhood requires $O(n^3)$ operations. However, this can be done much more efficiently using cumulative sums, reducing the demand to $O(n^2)$ operations. Numerical results can be seen in Table 2.1.

All numerical calculations in this thesis were performed on a MacBook Pro with 2.3 GHz Intel Core i7 processor and 16 GB memory. The algorithms were implemented in Matlab version 2014b.

The data matrices were created as follows: for $D \in \mathbb{R}^{n\times n}$ with rank d, a matrix $\hat{D} \in \mathbb{R}^{n\times d}$ was created using uniformly random numbers in the interval $[0, 10]$ and $D = \hat{D}\hat{D}^T$. Similar methods were used for creating the data matrix F with rank f. Values for n and the ranks of D and F are given for each trial. Time is in seconds, iterations are the number of GS algorithm loops performed, and function value is relative to the best improvement among all trials, i.e. $(f_{start} - f_{trial})/(f_{start} - f_{best})$.

As one can see from Table 2.1, the 2-swap neighborhood is clearly the best.

The 3-swap neighborhood is substantially slower and becomes intractable for larger n, hence why it is not present for trials with large n. The memory requirements become much too large to hold an entire $n \times n \times n$ first order approximation matrix in memory and subsequently search for improving swaps.

The insertion neighborhood fairs better in comparison with the 2-swap neighborhood, but still is inferior. The insertion neighborhood is three to six times slower and the quality of solution is substantially lower as well. Thus, these numerical experiments align with what is seen in the literature, i.e. the 2-swap neighborhood is the best choice of neighborhood for heuristic algorithms.

2.1.8 GradSwaps vs. Greedy Local Search

In this section we compare the proposed GS algorithm with a greedy local search (LS) technique. The LS technique calculates the exact difference for every swap and chooses the best swap at every iteration. Also, instead of recalculating the exact differences, LS updates the difference calculation as outline in Section 1.3.2), resulting in a significant speedup ($O(n^3)$ to $O(n^2)$) of the LS algorithm.

In Table 2.2 we compare the performance of GS with a greedy exact local search technique. The data matrices were created as $D = \hat{D}\hat{D}^T$ where $\hat{D} \in \mathbb{R}^{n \times d}$ is generated from a uniform random distribution $[0, 1]$, and similarly with $F = \hat{F}\hat{F}^T$, $\hat{F} \in \mathbb{R}^{n \times f}$. The function values given are $(\phi_{LS} - \phi_{GS})/(\phi_0 - \phi_{GS})$. This represents the proportion of relative improvement of GS over LS with higher values indicating a wider gap between the two techniques (in favor of GS) and a maximum of 1.

100 LS iterations, $\epsilon = 1e-5$ GS convergence tolerance					
n	(rank F, rank D)$\rightarrow$	(50,10)	(100,25)	(250,50)	(500,100)
500	Function Comparison $\frac{\phi_{LS}-\phi_{GS}}{\phi_0-\phi_{GS}}$	0.118	0.132	0.135	0.131
1000		0.339	0.347	0.360	0.353
2500		0.610	0.619	0.621	0.619
5000		0.755	0.762	0.762	0.761
500	LS Time (Seconds)	0.849	0.821	0.815	0.845
1000		4.78	4.79	4.82	4.81
2500		36.9	35.6	35.6	35.6
5000		195	194	194	198
500	GS Time (Seconds)	0.0737	0.0374	0.0348	0.0473
1000		0.217	0.182	0.145	0.171
2500		2.11	0.749	0.884	1.02
5000		10.7	6.59	3.36	3.55

Function Comparison $= \frac{\phi_{LS}-\phi_{GS}}{\phi_0-\phi_{GS}}$, Time in seconds

Table 2.2: Comparison of Local Search vs. GradSwaps

In terms of function improvement, GS outperforms LS in every case. In terms of time, GS is faster than LS by roughly 20x or more. Therefore, GS is the clearly superior technique to perform efficient local optimization. If the best possible solution is desired, LS often outperforms GS eventually. This would occur as the solution approaches a local optimum since the accuracy of the FOA becomes more suspect when the value of the predicted swaps approaches zero whereas the LS calculation will always be correct no matter the magnitude. This allows LS to fine tune a solution in a way that GS is not always able. However this case is not examined above as the number of iterations required to achieve this result is at least 5-10$\times$ longer and the objective value is barely superior to the GS solution.

The computational time for GS above remains reasonable for $n < 5000$ sized matrices. However, for very large instances ($n > 10000$) – as can be found in the applications to be discussed – this becomes somewhat intractable both in terms of time and memory requirements. To ensure larger problems remain tractable, we develop a method for optimizing a given block of indices at a time.

2.2 Block Method

As mentioned above, even for low-rank data matrices, there are memory and speed problems that we encounter as the size of the problem instance grows. So as a practical solution with any large problem, we break a single very large QAP into many subsets and optimize over each subset individually.

The process described below is very scalable in the sense that the amount of computational effort per iteration does not increase dramatically with the size of the problem instance; this is true as long as we maintain the size of the subsets over which we optimize. Computation time will still increase with larger problem instances due to the larger number of subsets which must be optimized.

We will refer to this partitioning and optimizing over subsets as the "block method" despite the fact that in practice contiguous blocks are not used. In Algorithm 2.5 we see an outline for the implementation of the block method of optimization.

In the following sections, we give details on the block method optimization technique.

Algorithm 2.5 Block method

Input: Data matrices F, D, Initial solution P_0, Convergence Tolerance ϵ, Max iterations

Output: Locally optimal solution P^*

$D \leftarrow P_0 D P_0^T$

$f \leftarrow \phi(I)$

$P^* \leftarrow P_0$

while $(f_i - f_{i-1})/f_i < \epsilon$ **do**

    $P_i \leftarrow$ select subset

    $P_i^* \leftarrow$ GradSwaps over P_i

    $P^* \leftarrow P_i^* P^*$ update current solution

    $D_i \leftarrow P_i^* D_i P_i^{*T}$ update data matrix D

end while

2.2.1 Block Function Computations

In order to perform GS on a subset of indices, we first must compute the gradient with respect to the subset. Without loss of generality, we assume the indices over which we will optimize all lie in the upper left block of the permutation matrix, though the following computations hold for any subset. We will compute the objective function and gradient with respect to P_1 in (2.53). Through some minor abuse of notation, we partition the data matrices into appropriately sized blocks as well.

$$P = \begin{pmatrix} P_1 & \\ & P_2 \end{pmatrix} \tag{2.53}$$

$$F = \begin{pmatrix} F_{11} & F_{12} \\ F_{21} & F_{22} \end{pmatrix} \tag{2.54}$$

$$D = \begin{pmatrix} D_{11} & D_{12} \\ D_{21} & D_{22} \end{pmatrix} \tag{2.55}$$

Note that the upper right and lower left blocks of P are zero blocks and in practice we assume $P_2 = I$, the identity matrix. We compute the block objective function.

$$\phi(P_1) = \operatorname{tr}\left(FPDP^T\right) \tag{2.56}$$

$$= \operatorname{tr}\left[\begin{pmatrix} F_{11} & F_{12} \\ F_{21} & F_{22} \end{pmatrix}\begin{pmatrix} P_1 & \\ & P_2 \end{pmatrix}\begin{pmatrix} D_{11} & D_{12} \\ D_{21} & D_{22} \end{pmatrix}\begin{pmatrix} P_1^T & \\ & P_2^T \end{pmatrix}\right] \tag{2.57}$$

$$= \operatorname{tr}\left[\begin{pmatrix} F_{11} & F_{12} \\ F_{21} & F_{22} \end{pmatrix}\begin{pmatrix} P_1D_{11}P_1^T & P_1D_{12}P_2^T \\ P_2D_{21}P_1^T & P_2D_{22}P_2^T \end{pmatrix}\right] \tag{2.58}$$

$$= \operatorname{tr}\left[\begin{pmatrix} F_{11}P_1D_{11}P_1^T + F_{12}P_2D_{21}P_1^T & * \\ * & F_{21}P_1D_{12}P_2^T + F_{22}P_2D_{22}P_2^T \end{pmatrix}\right] \tag{2.59}$$

$$= \operatorname{tr}\left(F_{11}P_1D_{11}P_1^T + F_{12}P_2D_{21}P_1^T + F_{21}P_1D_{12}P_2^T + F_{22}P_2D_{22}P_2^T\right) \tag{2.60}$$

We now compute the gradient with respect to P_1, again following the techniques from

[90]:

$$\mathrm{d}\phi(P_1) = \ \mathrm{dtr}\left(F_{11}P_1D_{11}P_1^T + F_{12}P_2D_{21}P_1^T + F_{21}P_1D_{12}P_2^T + F_{22}P_2D_{22}P_2^T\right) \tag{2.61}$$

$$= \mathrm{tr}\left(F_{11}\ \mathrm{d}P_1D_{11}P_1^T + F_{11}P_1D_{11}\ \mathrm{d}P_1^T + F_{12}P_2D_{21}\ \mathrm{d}P_1^T + F_{21}\ \mathrm{d}P_1D_{12}P_2^T\right) \tag{2.62}$$

$$= \mathrm{tr}\left((F_{11}^TP_1D_{11}^T + F_{11}P_1D_{11} + F_{12}P_2D_{21} + F_{21}^TP_2D_{12}^T)\ \mathrm{d}P_1^T\right) \tag{2.63}$$

$$\nabla_{P_1}\phi(P_1) = F_{11}^TP_1D_{11}^T + F_{11}P_1D_{11} + F_{12}P_2D_{21} + F_{21}^TP_2D_{12}^T \tag{2.64}$$

As mentioned above, since we fix P_2 in the block method and update only the indices in P_1, we can substitute $P_2 = I$, the identity matrix, giving us the final form of the block gradient (2.65).

$$\nabla_{P_1}\phi(P_1) = F_{11}^TP_1D_{11}^T + F_{11}P_1D_{11} + F_{12}D_{21} + F_{21}^TD_{12}^T \tag{2.65}$$

With the block function and gradient computed, we now have the tools to perform GS on a specific block of the permutation matrix. The block method also works exceedingly well for factorable F or D matrix, as discussed above in terms of the Cholesky factorization. Assume F and D are symmetric positive semi-definite with $F = \hat{F}\hat{F}^T$ and $D = \hat{D}\hat{D}^T$. If $\hat{F} = \begin{bmatrix} \hat{F}_1 \\ \hat{F}_2 \end{bmatrix}$ and $\hat{D} = \begin{bmatrix} \hat{D}_1 \\ \hat{D}_2 \end{bmatrix}$, we see that $F_{ij} = \hat{F}_i\hat{F}_j^T$ for $i, j \in \{1, 2\}$ and similarly for D_{ij}. Additionally, note that $F_{12} = F_{21}^T$, $D_{12} = D_{21}^T$, $D_{11}^T = D_{11}$, and $F_{11}^T = F_{11}$ in the case where F and D are symmetric. Using these ideas, we rewrite (2.65) as

$$\nabla_{P_1}\phi(P_1) = 2\hat{F}_1\hat{F}_1^TP_1\hat{D}_1\hat{D}_1^T + 2\hat{F}_1\hat{F}_2^T\hat{D}_2\hat{D}_1^T \tag{2.66}$$

$$= 2\hat{F}_1(\hat{F}_1^T\hat{D}_1 + \hat{F}_2^T\hat{D}_2)\hat{D}_1^T \tag{2.67}$$

Note that (2.67) has been evaluated at the identity, as will be the case in the GS algorithm. There are a few substantial computational benefits to this formulation of the gradient. The first is demonstrated in the factored form above (2.67); immediately this saves two matrix multiplications. Following the convention from the numerical experiments in Table 2.1, let $\text{rank}(D) = d$ and $\text{rank}(F) = f$. Also, let the number of indices selected for the block be m, so $\hat{D}_1 \in \mathbb{R}^{m\times d}$, $\hat{D}_2 \in \mathbb{R}^{(n-m)\times d}$ and similarly for F. For now, assume that $d < f$ with $f < m \ll n$ always true. Then the most efficient method for grouping the gradient products (2.67) can be seen in (2.68):

$$\nabla_{P_1}\phi(P_1) = (\hat{F}_1(\hat{F}_1^T\hat{D}_1 + \hat{F}_2^T\hat{D}_2))\hat{D}_1^T \tag{2.68}$$

Below we compute the computational effort of each step one at a time to arrive at a total complexity of this product.

$$\hat{F}_1^T\hat{D}_1 + \hat{F}_2^T\hat{D}_2 \in \mathbb{R}^{f\times d} \to O(fmd) + O(f(n-m)d) \tag{2.69}$$

$$\hat{F}_1(\hat{F}_1^T\hat{D}_1 + \hat{F}_2^T\hat{D}_2) \in \mathbb{R}^{m\times d} \to O(mfd) + O(fmd) + O(f(n-m)d) \tag{2.70}$$

$$(\hat{F}_1(\hat{F}_1^T\hat{D}_1 + \hat{F}_2^T\hat{D}_2))\hat{D}_1^T \in \mathbb{R}^{m\times m} \to O(mfd) + O(fmd) + O(f(n-m)d) + O(m^2d) \tag{2.71}$$

We see the total complexity of this is $O((n-m)fd + m^2d)$. However, for a given set of indices the product $\hat{F}_2^T\hat{D}_2$ only needs to be computed once. Thus, once a subset has been chosen and the first gradient evaluation has taken place,

every remaining iteration of the block method GS algorithm requires only $O(m^2 d)$ operations. If $f < d$, a similar result with f substituted for d can be attained with a different grouping of products.

The method for choosing the subset (block) of indices still needs to be discussed. This is a non-trivial choice and the success of the algorithm heavily depends upon it.

2.2.2 Subset Selection

The choice of subset for each iteration of the block method is crucial. If disjoint subsets are chosen, the point of diminishing returns is found very quickly. However, if non-disjoint subsets are chosen, the resulting local optimum is always superior. The question remains: how does one choose the best subsets over which to optimize?

One idea is to set the block size m, partition the entire n-set of indices into $2n/m$ disjoint blocks of size $m/2$, and develop a schedule to pair the blocks together so each iteration of GS optimizes over a block of size m. A possible schedule is to choose random pairs of blocks and optimize over those. Another schedule is to use a kind of hypercube scheduling, which will first pair adjacent blocks, then once-separated blocks, then thrice separated blocks, and so on, with the separation following a $2^i - 1$ scheme. A more detailed description can be found in Algorithm 2.6. In the algorithm, $\mathcal{M}$ is the schedule of block pairings where b is the index of a block, thus $(b, b + 2^i)$ is the pairing of two blocks of indices.

A far simpler scheme than block pairing is to choose m random indices every

Algorithm 2.6 Hypercube Scheduling

Input: Number of indices n, Maximum iterations k_{max}

Output: Hypercube Pairing Schedule $\mathcal{M}$

$\mathcal{M} \leftarrow \emptyset$
for $i = 0 \dots \lceil \log_2(n) \rceil - 1$ **do**
  $b_0 \leftarrow 0$
  for $j = 0 \dots i$ **do**
    $b_0, b \leftarrow b_0 + 1$
    while $b + 2^i \leq n$ **do**
      $\mathcal{M} \leftarrow \mathcal{M} \cup (b, b + 2^i)$
      $b \leftarrow b + 2^{i+1}$
      $k \leftarrow k + 1$
      if $k = k_{max}$ **then**
        Break all loops
      end if
    end while
  end for
end for

iteration of the block method. This approach requires almost no overhead computation aside from the block GS algorithm. A comparison of these three schemes will be shown after a discussion of the convergence criterion for the block method.

2.2.3 Convergence Criterion

As with any algorithm, we need a way to determine acceptable convergence for the block method. A naive choice is to wait until no beneficial swaps are performed in GS for k different subset selections; in numerical experiements, this choice takes 10 or more times longer without attaining superior results. Instead, we use the same convergence criterion as with GS itself. If the relative improvement of the objective function has slowed sufficiently, we stop the algorithm. In the following section, Table 2.5 demonstrates the quality of different values for ϵ, but ultimately this parameter must be tuned for the given application.

2.2.4 Block Method Results

We experiment with a variety of large QAP to explore varying the settings for the block method described above. The data matrices were created in the same manner as in Section 2.1.7. We vary the size of the data matrices, the size of the subset selected, the type of index selection, and the convergence criterion. Each value given is the average over 10 runs made at the appropriate settings.

Time is measured in seconds, iterations are the number of block-method GS performed, and improvement is the function value relative to the best improvement among all trials, i.e. $(f_{start} - f_{trial})/(f_{start} - f_{best})$.

$n = 1e4,\ d = 100,\ f = 500, \epsilon = 10^{-5}$						
Subset size m		250	500	1000	2000	2500
Hypercube	Time	3.22	2.09	2.12	3.49	4.46
	Improvement	0.979	0.980	0.987	0.990	0.984
	Iterations	41	21	11	6	5
Random Block Matching	Time	1.41	1.32	1.52	3.20	3.64
	Improvement	0.343	0.479	0.543	0.703	0.614
	Iterations	17.5	12.9	8.0	5.8	4.4
Random Indices	Time	5.47	4.60	5.22	8.64	10.87
	Improvement	0.808	0.901	0.957	0.978	0.974
	Iterations	67.8	50.0	32.4	19.6	16.2

Improvement $= (f_{start} - f_{trial})/(f_{start} - f_{best})$, Time in seconds

Table 2.3: Comparison of Block Method Subset Sizes, $n = 10000$

Table 2.3 compares the time to convergence with varied subset size. As one can see in the table, the hypercube scheduling strategy isn't quite as fast as the random block matching, but the quality of solutions is much better (especially for the smaller m). Additionally, the random indices scheduling strategy approaches the hypercube strategy in quality, but is substantially slower. As mentioned earlier, the block method is practically infinitely scalable (up to memory constraints), so in the interest of demonstrating scalability, we increase to $n = 3e4$ and repeat the simulation.

The tendencies we witnessed in Table 2.3 become even more pronounced in Table 2.4, with a few different trends emerging as well. Again, the hypercube scheduling performs the best in terms of objective function improvement, with random block and random indices matching further deteriorating in comparison. The time trends

$n = 3e4,\ d = 100,\ f = 500, \epsilon = 10^{-5}$						
Subset size m		250	500	1000	2000	2500
Hypercube Schedule	Time	24.34	13.59	9.80	11.28	13.07
	Improvement	0.978	0.985	0.987	0.986	0.987
	Iterations	120	61	31	16	13
Random Block Matching	Time	6.32	5.21	4.19	6.45	9.15
	Improvement	0.226	0.319	0.359	0.446	0.563
	Iterations	31.4	23.6	13.5	9.7	9.8
Random Indices	Time	16.86	20.08	18.65	24.38	29.20
	Improvement	0.491	0.762	0.871	0.940	0.949
	Iterations	83.2	87.6	63.3	44.3	37.5

Improvement $= (f_{start} - f_{trial})/(f_{start} - f_{best})$, Time in seconds

Table 2.4: Comparison of Block Method Subset Sizes, $n = 30000$

are mostly maintained, with random blocks being the fastest followed by hypercube scheduling; the one exception is for the very smallest subset size, where hypercube scheduling has the most iterations as well as slowest time. However, $m = 250$ gives the worst function values for all three techniques, thus it would be a poor choice to use in practice.

One trend that became much more obvious with the larger instance is the decrease then increase in time for both hypercube scheduling and random block matching as m increases. The middle size, $m = 1000$ is the fastest for both schedules, reinforcing the idea that subset size is a parameter that should be tuned to get the best results.

In Table 2.5 we restrict ourselves to only the hypercube scheduling technique and vary the subset size as well as the convergence criterion. Note that the con-

$n = 1e4,\ d = 100,\ f = 500$					
Time (seconds)					
Subset size m	250	500	1000	2000	2500
$\epsilon = 10^{-4}$	0.14	0.15	0.33	1.05	1.56
$\epsilon = 10^{-5}$	0.13	1.53	1.68	2.92	3.64
$\epsilon = 10^{-6}$	2.67	1.80	2.02	3.51	4.39
$\epsilon = 10^{-7}$	3.24	2.15	2.38	4.07	4.73
Improvement					
$\epsilon = 10^{-4}$	0.035	0.084	0.182	0.387	0.485
$\epsilon = 10^{-5}$	0.035	0.983	0.987	0.989	0.990
$\epsilon = 10^{-6}$	0.978	0.984	0.988	0.990	0.990
$\epsilon = 10^{-7}$	0.978	0.985	0.988	0.990	0.990
Iterations					
$\epsilon = 10^{-4}$	2	2	2	2	2
$\epsilon = 10^{-5}$	2	21	11	6	5
$\epsilon = 10^{-6}$	41	21	11	6	5
$\epsilon = 10^{-7}$	41.1	21.1	11.1	6.1	5.1

Improvement $= (f_{start} - f_{trial})/(f_{start} - f_{best})$, Time in seconds

Table 2.5: Comparison of Hypercube Schedule Subset Size and Stop Tolerance, $n = 10000$

vergence criterion is varied for both the individual block GS and the overall block method algorithm. It is important to recognize that "iterations" still measures the number of block GS algorithm iterations, not the number of iterations within a single execution of the block GS.

We notice immediately that convergence tolerance has a significant impact on solution quality as measured by relative function value. For small subsets and $\epsilon = 10^{-4}$, the relative function value is not even a tenth as good as the best solution. As m increases this effect is diminished somewhat, but evidently an insufficiently

small tolerance results in poor solutions. Unsurprisingly, stricter tolerance forces the algorithm to take longer. However, since the number of iterations does not increase significantly (aside from jumps when ϵ is too large), we must conclude the time difference between $\epsilon = 10^{-6}$ and $\epsilon = 10^{-7}$ is due to additional iterations within a single execution of block GS.

Clearly there is a tradeoff between quality and speed. However, as quality does not significantly diminish between trials, aside from the gigantic drops due to insufficient strictness in convergence criterion, the most efficient choice for the block method appears to be some happy medium in terms of convergence strictness and subset size. Specifically, in Table 2.5 the ideal parameter choices appear to be $m = 1000$ with $\epsilon = 10^{-5}$.

In this chapter we gave a theoretic description of the GS algorithm, including theoretic and empirical justification for using the FOA instead of exact difference, using the 2-swap neighborhood, and the optimal method for swap selection. Additionally, we develop techniques to scale the GS algorithm to instances much larger than currently appear in the literature by optimizing over blocks of indices instead of the entire problem. In the following chapter, we examine three applications that justify this exploration into very-large-scale QAP.

CHAPTER 3
LARGE-SCALE QAP APPLICATIONS

Thus far, we gave an extensive overview of the QAP and developed the block GS algorithm, an algorithm designed to locally optimize large-scale, low-rank QAP. However, as all commonly studied instances of the QAP are relatively small, with the largest in the well-studied QAPLIB [27] only at $n = 256$, it is necessary to develop large-scale applications of the QAP. This chapter describes three real-world applications to demonstrate the effectiveness of GS in practice.

The first application uses extreme learning machines (ELM), a type of machine learning, to visualize high dimensional data. This application comes from Akusok et al. [6] with GS improving on the computation time considerably.

The second application deals with a less studied type of the well-known traveling salesperson problem (TSP). We examine the version in which the distance metric is the squared euclidean distance; this allows for use of the low-rank GS techniques.

The third and final application is the generation of random data with a specified correlation structure. Inspired by [30], our formulation of this problem as a QAP and subsequent solution with GS yields better results than the usual correlated data generation techniques.

3.1 Data Visualization Via Extreme Learning Machine

Data visualization is very important in today's world of ubiquitous data. Often, it is difficult to understand or gain intuition with data; by visualizing large

amounts of data in an effective way, we are able to take advantage of humans' inherent ability to quickly understand visual information. Thus, the ability to effectively visualize data is crucial in the process of data analysis. The importance of data visualization is underscored by the fact that it has been the topic of many texts in recent years [31, 122, 130].

Popular techniques for performing data visualization include principal component analysis (PCA) [109, 61], self-organizing maps (SOM) [126, 112], and other dimensionality reduction techniques [56]. Applications of data visualization occur in any field that uses large amounts of data, which in today's world consists of most analytic fields. Data visualization can be seen in biological applications, such as genetics [123, 124] and neuroscience [60], as well as in business decision making [81, 66, 80].

The following discussion uses an extreme learning machine to visualize the MNIST data set. As mentioned above, this application and general idea comes from Akusok et. al. [6] though the formulation as a QAP and subsequent use of GS is novel with this thesis.

3.1.1 Theory

Extreme learning machines (ELM) are a type of single layer feed-forward neural network (SLFN) suggested by Huang et al. in [72]. Neural networks (NN) are a type of machine learning algorithm that have existed for over 50 years, dating back at least to Rosenblatt's perceptron [105]; like many machine learning techniques, they seek to accurately approximate existing data and predict future results. The reader is

directed to [37] for a complete description of NN theory, construction, and application.

NN can be divided into three sections: input neurons, output neurons, and hidden neurons. The input and output neurons are each a single layer while the hidden neurons may be organized into many layers. These many layers of neurons need to be trained on the given data, often through a process called back-propogation [68] which can be computationally intensive and very slow. SLFN require training like any neural network though they possess only a single hidden layer of neurons (where many neural networks will have many layers) and thus less computation is required to fully train the network.

A single neuron in any type of NN consists of weights from a neuron in the previous layer $\mathbf{w}_i \in \mathbb{R}^{d_I}$ (where d_I is the dimension of the input data), a bias to the inputs $b_i \in \mathbb{R}$, an activation function $g(x) : \mathbb{R} \to \mathbb{R}$, and output weights to the next layer of neurons $\beta_i \in \mathbb{R}^{d_F}$ (where d_F is the dimension of the output or target data). Together, the output of a single neuron is modeled in (3.1),

$$\boldsymbol{\beta}_i g(\mathbf{w}_i \cdot \mathbf{x}_j + b_i) \tag{3.1}$$

where the index i corresponds to the ith neuron in that layer and the index j indicates the effect on the jth input from the previous layer.

In a SLFN, there are a total of three layers, the input, output, and single hidden layer. The input layer is the original data and the output layer is the result of the hidden layer, thus there is only one layer of neurons that have an effect on the data. If $\mathbf{t}_j \in \mathbb{R}^{d_F}$ is the target data, then (3.2) shows the SLFN approximation

where$k$ is the number of neurons and n is the number of target samples.

$$\mathbf{t}_j \approx \sum_{i=1}^{k} \boldsymbol{\beta}_i g(\mathbf{w}_i \cdot \mathbf{x}_j + b_i) \quad j = 1, \ldots n \tag{3.2}$$

Ideally this approximation will be close or even exact. In an SLFN, an optimization procedure would take place to train the weights $\mathbf{w}_i$, b_i, and $\boldsymbol{\beta}_i$ to improve the accuracy of the approximation. The key difference between a general SLFN and an ELM is that in an ELM, $\mathbf{w}_i$ and b_i are generated randomly and only $\boldsymbol{\beta}_i$ is trained. With $\mathbf{w}_i$ and b_i fixed, we define a matrix $H \in \mathbb{R}^{n \times k}$, the hidden layer output matrix.

$$H_{ji} = g(\mathbf{w}_i \cdot \mathbf{x}_j + b_i) \tag{3.3}$$

Below we define B and T with the goal of writing equation (3.1) as a matrix equation. The rows of B are the transposed column vectors $\boldsymbol{\beta}_i$.

$$B = \begin{pmatrix} \boldsymbol{\beta}_1^T \\ \vdots \\ \boldsymbol{\beta}_{N_1}^T \end{pmatrix} \in \mathbb{R}^{k \times d_F} \qquad T = \begin{pmatrix} \mathbf{t}_1 \\ \vdots \\ \mathbf{t}_{N_2} \end{pmatrix} \in \mathbb{R}^{n \times d_F} \tag{3.4}$$

With these definitions, we write our approximation equation, seen in (3.5).

$$HB = T \tag{3.5}$$

In [72], Huang et al. show that as long as the activation function g is infinitely differentiable, then provided H is square (i.e. $n = k$ so there are sufficient hidden neurons), H is invertible with probability one and $\|HB - T\| = 0$. That is, if the number of hidden neurons are equal to the number of samples in the target data, the SLFN can approximate the target data exactly.

As of yet, there is no reason why the randomization of $\mathbf{w}_i$ and b_i in an ELM is useful or necessary; this comes to light in the training process. In a SLFN, these

weights as well as $\boldsymbol{\beta}_i$ would have to be trained; this process can be very slow and computationally expensive. In the ELM, only the $\boldsymbol{\beta}_i$ need to be trained and we will see that this is a computationally easy task.

It is a well known fact that if A is invertible, the solution to $\min_x \|A\mathbf{x} - \mathbf{b}\|_2$, the linear least-squares problem, is $\mathbf{x} = A^{-1}\mathbf{b}$. In the case where A is not invertible, usually due to the being rectangular, the solution is instead $\mathbf{x} = A^{\dagger}\mathbf{b}$, where $A^{\dagger}$ is the Moore-Penrose pseudo-inverse of A. A similar result holds in our case.

$$B^* = \text{argmin}_B \|HB - T\|_F = H^{\dagger}T \tag{3.6}$$

Since $H^{\dagger} = (H^T H)^{-1} H^T$, assuming $H^T H$ is invertible, the training of the ELM is deterministic and the solution is computable in $O(k^2(n+k))$ time.

3.1.2 ELMVIS Application

The application of an ELM to the visualization problem was first suggested by Akusok et al. [6]. They describe what they call the ELMVIS+ algorithm. Most visualization algorithms incorporate some sort of projection from the original, (presumably) higher dimensional space to two or three dimensions, allowing humans to see the data; usually the quality of the visualization is measured in the lower dimensional projection space. ELMVIS+ approaches this a little differently by changing the visualization problem from a projection problem to an assignment problem. To clarify, "ELMVIS" will refer to the idea of using ELM in data visualization; "ELMVIS+" refers to the specific algorithm Akusok et. al developed to perform the maximization.

The idea is to start with a random assignment $p \in S_n$ of the original data

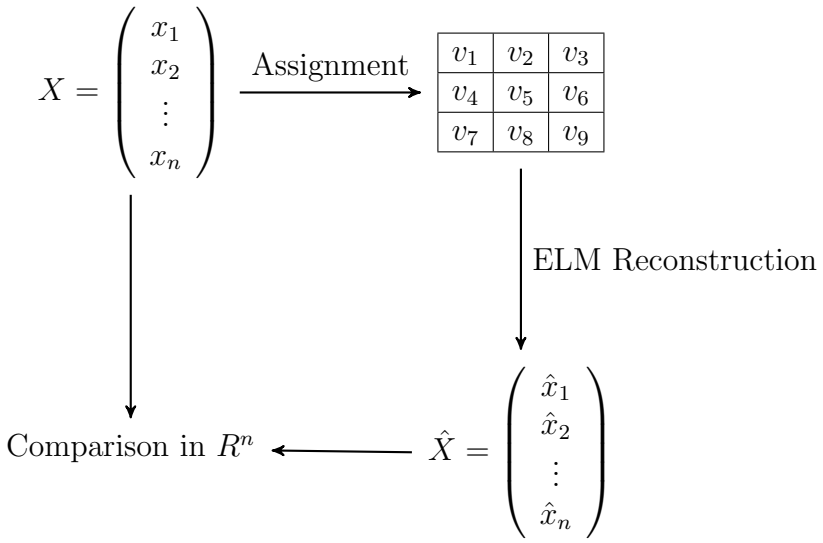


Figure 3.1: ELMVIS+ Diagram

$X \in \mathbb{R}^{n \times d}$ to a fixed visualization space $V \in \mathbb{R}^{n \times v_d}$ (v_d= 2 or 3), reconstruct the data $\hat{X} \in \mathbb{R}^{n \times d}$ in the original space via an ELM as a de-projection, and measure the quality of the visualization by comparing the similarity of X with $\hat{X}$ in the higher dimension space. Here n is the number of data points and d is the dimension of the data. This process can be seen in Figure 3.1.

Instead of retraining the ELM to improve the reconstruction, the assignment of the data to visualization space is reordered to improve the similarity of the reconstructed and original data. The reordering is done by repeatedly choosing two random data points, swapping them, and keeping the swap if it improves the similarity. This is summarized in Algorithm 3.1. There are a few implementation details they use to improve the algorithm. Specifically, swaps are not actually performed every iteration; instead a list of improving swaps is kept and periodically multiple swaps are done at

once as the actual swapping of indices can be computationally slow.

Algorithm 3.1 ELMVIS+

Input: Data X, Visualization space V

Output: Assignment of X to V

```
Train ELM on visualization space
p ← initial assignment of X to V
Calculate initial similarity
while Convergence not satisfied do
    Choose two random data points
    Swap two data points and calculate new similarity
    if Similarity improves then
        Keep the swap
    end if
end while
```

Thus far, the discussion of ELMVIS+ has been entirely qualitative. At this point, we will quantitatively describe ELMVIS+ and subsequently formulate the problem as a low-rank QAP. There is little more to say regarding the assignment of the X to V; the assignment is a permutation which is initiated randomly and the permutation is modified to improve the similarity of X and $\hat{X}$. The meaning of "similarity" will be discussed in detail below. The visualization space, which does need to be

decided upon prior to the initial assignment, is a collection of vectors $v_i \in \mathbb{R}^2$, which each indicate the position of the visualization point. It would also be possible to perform visualization in three dimensions, although only two will be discussed in this thesis. For a rectangular space, a possible assignment for $\mathbf{v}_i$ with n samples and length of rectangle l is seen in (3.7).

$$\mathbf{v}_i = \left(\left\lceil \frac{i}{\ell} \right\rceil, (i-1) \text{mod } l + 1 \right) \tag{3.7}$$

The next step is the initialization of the neuron weight matrix $H \in \mathbb{R}^{n \times k}$ and the training of the ELM. First we choose the number of neurons k. As will be quantified later, this choice has great impact on both quality and speed of visualization. Next, we generate the input weights ($\mathbf{w}_i$) and biases (b_i) from a uniform random distribution, with our choice of distribution taking $\mathbf{w}_i \in [-1, 1]$ and $b_i \in [0, 1]$. Additionally, the activation function $g(x)$ must be infinitely differentiable, so we choose $g(x) = 1/(1 + \exp(-x))$. With these choices, we initialize our neuron weight matrix H using visualization space V:

$$H_{ij} = g(\mathbf{w}_j \cdot \mathbf{v}_i + b_j) \qquad \forall i \in 1 \ldots k \quad j \in 1, \ldots n \tag{3.8}$$

We now train the ELM to approximate $\hat{X} = HB$ (3.9)-(3.13). Recall that the training of the ELM means finding the output weight matrix B and from (3.6) we know $B = H^{\dagger}X$. Here X is used as the target data as we wish to reconstruct X as accurately as possible. It is well known that $H^{\dagger} = (H^T H)^{-1} H^T$, so we can simplify our approximation. Also, in (3.12) we introduce the QR decomposition of H, so $H = QR$, where $Q \in \mathbb{R}^{n \times k}$ is a matrix of orthonormal columns and $R \in \mathbb{R}^{k \times k}$

is an upper triangular invertible matrix.

$$\hat{X} = HB \tag{3.9}$$

$$= HH^{\dagger}X \tag{3.10}$$

$$= H(H^T H)^{-1} H^T X \tag{3.11}$$

$$= QR(R^T Q^T QR)^{-1} R^T Q^T X \tag{3.12}$$

$$= QQ^T X \tag{3.13}$$

Note that it is unnecessary to actually compute the full QR decomposition of H. We only need the first k orthonormal columns of Q (since H has rank $\leq k$) and computing the R matrix is entirely unnecessary.

At this point we need to define what is meant by "similarity" between X and $\hat{X}$ as this is our measure of quality, the objective function. The objective function will be the mean cosine similarity between the matrices X and $\hat{X}$, making this a maximization problem. Note that a maximization problem is the negative of a minimization problem, thus our subsequent comparison to the minimization QAP is relevant. In d-dimensional space, $\cos(\theta) = (\mathbf{a} \cdot \mathbf{b})/(\|\mathbf{a}\|\|\mathbf{b}\|)$ where $\mathbf{a}, \mathbf{b} \in \mathbb{R}^m$. Since $X \in \mathbb{R}^{n \times d}$ is n d-dimensional data points, we normalize X by dividing each row by its Euclidean norm. Through a minor abuse of notation, assume X and $\hat{X}$ are normalized from here on, allowing us to write the cosine similarity for a single sample $\cos(\theta_i) = X_i^T \hat{X}_i$. Using matrix notation, we divide by n to write the mean cosine similarity among all samples:

$$\cos(\theta) = \frac{1}{n} \langle X, \hat{X} \rangle \tag{3.14}$$

$$= \frac{1}{n}\text{tr}(X^T\hat{X}) \tag{3.15}$$

$$= \frac{1}{n}\text{tr}(X^TQQ^TX) \tag{3.16}$$

In order to write this as a QAP, recall that at every iteration the ELM de-projection doesn't change but instead the order of the samples in X changes. Thus we write the ELMVIS+ objective function as a function of the permutation matrices P:

$$\phi(P) = \frac{1}{n}\text{tr}((PX)^TQQ^TPX) = \frac{1}{n}\text{tr}(X^TP^TQQ^TPX) \tag{3.17}$$

If we let $F = QQ^T$ and $D = XX^T$ we recover the original Koopmans-Beckmann QAP objective function $\phi(P) = \text{tr}(FPDP^T)$. Notice that these are factorizations of F and D, thus saving the computational work of computing them up front.

Additionally, the rank of QQ^T is k, the number of neurons in our ELM. Thus, we have control over this parameter which has a significant role in the computational time. However, there is a tradeoff: the greater the number of neurons in the ELM, the better the reconstruction $\hat{X}$ when the visualization space is de-projected. As with any NN, the number of neurons is a parameter to be tuned for the best performance of the algorithm. The rank of XX^T is the dimensionality of each data point, in this case d; $d \ll n$ in most cases, an encouraging fact for the efficiency and efficacy of the GS algorithm.

The bulk of the computational effort comes from the gradient evaluation, a process which was covered in Section 2.2.1. However, note that the objective function

can be evaluated very efficiently as well (3.18):

$$\phi(P) = \text{tr}(X^T P^T Q Q^T P X) = \|Q^T P X\|_F^2 \tag{3.18}$$

In (3.18) we see that the objective function can be written as the Frobenius norm of $Q^T P X \in \mathbb{R}^{k \times d}$, thus the objective function can be computed very quickly without forming the entire matrix product $X^T P^T Q Q^T P X$. The product PX is done through reindexing as opposed to matrix multiplication, so the entire objective function can be computed in $O(dkn)$ as the matrix product is much more expensive than calculating the Frobenius norm.

Finally, in practice we divide $\phi(P)$ by n, thus calculating the mean cosine similarity of all the samples. This allows us to more easily compare the objective value to 1, the maximum of cosine. However, we can do a little better than this, as it is relatively simple to compute an upper bound based on singular values of X. This bound comes from relaxing constraints on P to orthogonal matrices instead of permutation matrices. Let $X = U \Sigma V^T$ be the singular value decomposition, so $U \in \mathbb{R}^{n \times n}$ and $V \in \mathbb{R}^{d \times d}$ are orthogonal and $\Sigma \in \mathbb{R}^{n \times d}$ is diagonal with the singular values on the diagonal. Also let $Q = \hat{Q} R$ be the QR-decomposition so $\hat{Q} \in \mathbb{R}^{n \times n}$ is orthogonal and $R \in \mathbb{R}^{n \times k}$ is upper triangular. Since Q has orthonormal columns, we can choose $\hat{Q}$ such that R is just the diagonal of ones.

We make these substitutions in the objective function to develop the bound. Assume $k < d$ as this will be the case in our numerical experiments. Recall k is the

number of neurons and d is the size of a single sample of data.

$$\|Q^T PX\|_F^2 = \|R^T \hat{Q}^T PU\Sigma V^T\|_F^2 \tag{3.19}$$

$$= \|R^T \hat{P}\Sigma\|_F^2 \tag{3.20}$$

Above we assigned $\hat{P} = \hat{Q}^T PU$, so $\hat{P}$ is an orthogonal matrix. Also, note that the Frobenius norm is invariant under pre- or post-multiplication of orthogonal matrices, thus we can remove V^T from the function. If we choose $P = \hat{Q}U^T$ then $\hat{P} = \hat{Q}^T \hat{Q}U^T U = I$, leaving the product $\hat{R}^T\Sigma$ inside the norm. This product is a diagonal $k \times d$ matrix where the diagonal is the largest k singular values of X. This gives the result in (3.21) where σ_i is the ith largest singular value of X.

$$\min_{P \in O(n)} \phi(P) = \sum_{i=1}^{k} \sigma_i^2 \tag{3.21}$$

This bound is a special case of the eigenvalue bounds discussed in Section 1.3.1, specifically in (1.20); however, the derivation is original. Similar to the objective function, in practice this bound is divided by n to normalize it to a cosine value. As this bound is based on orthogonal matrices, it is not a tight upper bound. However, it is still useful to develop a tighter upper bound than $\phi(P) \leq 1$.

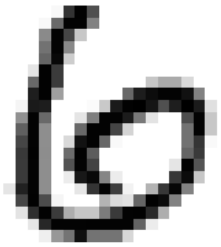
This concludes the theoretical discussion of the ELMVIS+ algorithm and formulation of the ELMVIS problem as a QAP. In the next section we demonstrate the technique using the MNIST data set.

3.1.3 ELMVIS Results

This section will apply the block GS algorithm to the ELMVIS application. We will use the MNIST database of handwritten digits [85] to demonstrate this procedure.

(a) Three

(b) Five



(c) Six

(d) Nine

Figure 3.2: Four Samples of MNIST Handwritten Digits

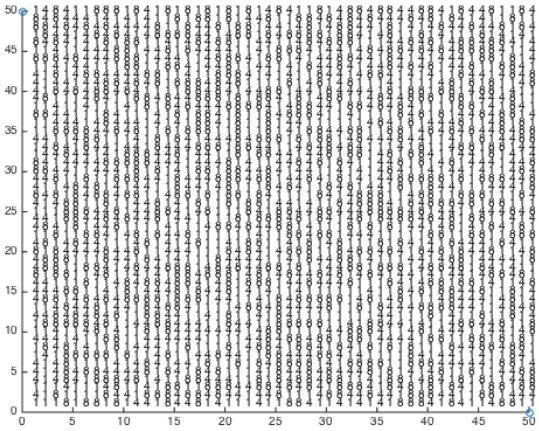
The data consists of two sets, training and testing samples, with 60000 and 10000 samples in each, respectively. An individual sample is a 28x28 pixel image where every pixel has a greyscale value from 0 to 255. There is no difference between the testing and training data aside from the size of the data set. In addition to the handwritten samples themselves, there are labels for every sample indicating the digit that was written as originally the data set was meant to test classification machine learning techniques. In Figure 3.2 we see four examples of the handwritten digits.

In Figure 3.3 we see a sample of the before (Figure 3.3a) and after (Figure

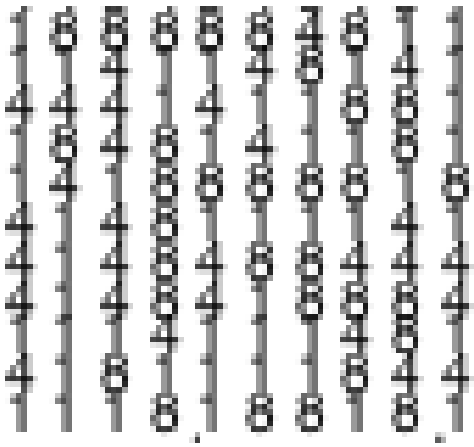
3.3c) of the visualization process. A magnified view of the lower left 11x10 digits is shown as well. Our sample visualization shows 2500 samples instead of the entire 10k or 60k data set. To visualize those, it is necessary to have a very large figure or to take a majority sample; in either case the end result appears very much as in Figure 3.3. The figures do not actually show the handwritten digits themselves but instead a typeset version of the number. Additionally, only the digits 1, 4, and 8 were chosen for this visualization due to their dissimilarity in appearance, thus aiding the eye in discerning the differences in this example. The corresponding cosine similarities, i.e. objective value, are given as well. The visualization is done on a 50x50 rectangular grid with 25 neurons in the ELM.

It is easy to see that this technique produces an organized visualization from a random assortment of digits. There is definite grouping to be seen with few stray digits. Additionally, there is substantial improvement in the objective value, though we have no reason to believe we have found a global maximum. The bound from (3.21) bounds the objective value for this set of indices at 0.8571. Just by looking at the magnified section, Figure 3.3d, we see there are some improvements that could be made. However, GS is a local optimization procedure, thus if we had found a global maximum it would be due to extraordinary luck.

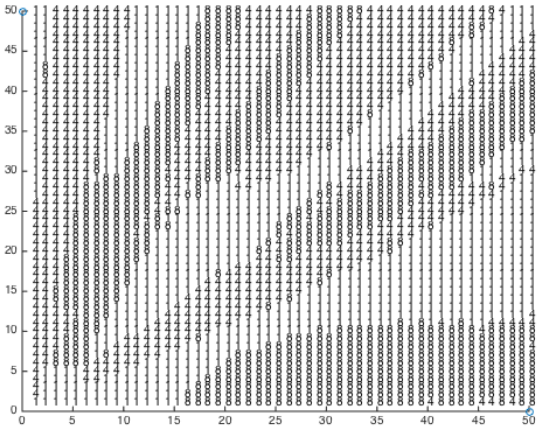
Before applying block GS to this problem, we first look at the upper bounds developed above in (3.21). From Table 3.1 we see that the upper bounds are less than 1. This makes sense, as a cosine similarity of 1 would indicate a perfect reconstruction which would require as many neurons as samples (or at least as many as the dimension



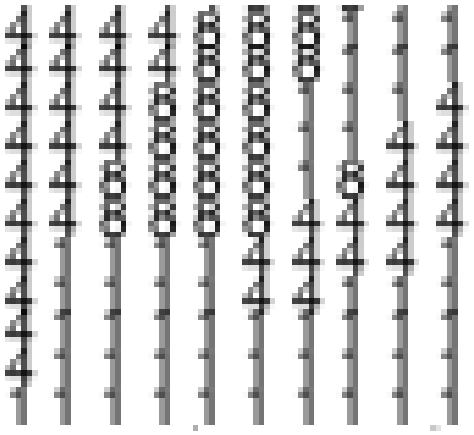
(a) Objective Value = 0.4526



(b) Lower Left Corner of 3.3a



(c) Objective Value = 0.6828



(d) Lower Left Corner of 3.3c

Figure 3.3: Sample Visualization with $n = 2500$

Neurons	10	25	50	100
Upper Bound 10k	0.6838	0.8106	0.8922	0.9481
Upper Bound 60k	0.6732	0.8012	0.8857	0.9447

Table 3.1: Upper Bounds by Number of Neurons for 10k and 60k Data Sets

of the samples). To properly evaluate the quality of the local maxima we must keep these bounds in mind instead of the number 1.

Once again, there are two parameters which must be tuned for this problem: the number of neurons (tradeoff between accuracy and speed) and the size of the block subset. We show results for both the 10k test data set and the entire 60k training data set in Tables 3.2 and 3.3, respectively. These computations were done on the same computer as those in Chapter 2: a MacBook Pro with 2.3 GHz Intel Core i7 processor and 16 GB memory. We use the block method with the hypercube scheduling scheme for subset selection. Convergence tolerance is set $\epsilon = 1e-5$, the data set is the MNIST data set described above, and the dimensions of the rectangular visualization spaces are 100x100 (for the 10k data set) and 240x250 (for the 60k data set).

Tables 3.2 and 3.3 show results of block GS applied to ELMVIS. The number of neurons $k \in \{10, 25, 50, 100\}$ and the subset block size $m \in \{250, 500, 1000, 2000\}$ are varied. The upper left quadrant gives the objective function (cosine similarity), the upper right gives the total number of gradient evaluations during block GS, the lower left gives the total time to convergence of block GS, and the bottom right gives the number of iterations of GS. To clarify, the number of iterations means the number

	Objective Function				Gradient Evaluations			
Subset size m	250	500	1000	2000	250	500	1000	2000
10 neurons	0.4620	0.4747	0.4775	0.4807	478.5	1103.2	779.3	50.5
25 neurons	0.4902	0.5192	0.5223	0.5203	483.7	1177.3	788.7	488.8
50 neurons	0.5809	0.6044	0.6075	0.6124	2734.8	2222.0	1273.5	695.3
100 neurons	0.5936	0.6170	0.6232	0.6252	2271.6	2069.7	1200.5	700.8
	Time (seconds)				Iterations			
10 neurons	4.25	12.84	18.53	35.33	63.5	152.3	101.8	50.5
25 neurons	4.26	13.92	20.48	42.14	61.0	154.9	100.8	59.2
50 neurons	24.87	26.77	32.38	60.08	333.9	274.7	152.6	78.9
100 neurons	25.53	30.51	34.69	63.77	282.0	267.6	146.9	82.5

Table 3.2: MNIST 10k Dataset Comparison of Subset Size and Number of Neurons

of hypercube pairings made during the block GS algorithm.

From the Table 3.2 we can see that with more neurons (k) the objective value increases substantially. With large enough k, we know the reconstruction would be perfect, thus it makes sense with increasing k that the reconstruction would steadily improve. An increase in k also increases the computation time.

The size of the subset (m) has an effect on the objective value as well, though this effect is somewhat dependent on the number of neurons. If $k = 10$, then $m = 250$ is significantly worse than all other results, though much faster as well. To a lesser extent, the same is true for $k = 10$ and $m = 500$. However, once $k \geq 25$, there is relatively little difference in objective value based on subset size. The primary effect of m is on the time to convergence: time is monotonic with m, so the time to convergence increases as the subset size increases.

The number of gradient evaluations and iterations of the block GS algorithm

	Objective Function				Gradient Evaluations			
Subset size m	250	500	1000	2000	250	500	1000	2000
10 neurons	0.4568	0.4598	0.4588	0.4852	2333	1113	616	902
25 neurons	0.4879	0.4976	0.5448	0.5456	2791	2467	4427	2110
50 neurons	0.5168	0.5296	0.5691	0.5769	2969	2591	4371	2064
100 neurons	0.5719	0.6310	0.6339	0.6444	5022	8917	5085	3746
	Time (seconds)				Iterations			
10 neurons	177	84.9	56.8	126	361	164	90	108
25 neurons	177	157	309	272	348	295	466	216
50 neurons	183	169	338	278	335	295	486	216
100 neurons	333	640	431	549	540	1000	556	3746

Table 3.3: MNIST 60k Dataset Comparison of Subset Size and Number of Neurons

are clearly related. On average, every iteration uses between 6-10 gradient evaluations, depending on k and m. Also, the number of iterations and evaluations behave inversely with respect to m, since with a larger subset fewer iterations would be necessary to arrive at a local maximum for the entire problem. Finally, there does not seem to be a strong relationship between iterations and m, indicating that the number of neurons makes little difference in the number of iterations required for convergence.

Overall, the results for the 60k data set, seen in Table 3.3, are fairly similar to those of the 10k data set. Increasing k improves the objective function but also requires additional computational time. The primary difference between the two tables is that time no longer increases monotonically with m. We see the fastest average time to convergence occurs for $m = 250,\ 500$, and 1000 depending on k. Finally, there are 6-10 gradient evaluations on average per iteration of the block GS algorithm.

We have not examined the effect of different visualizations spaces on the performance of GS with regard to the ELMLVIS technique. However, the underlying effects described above are expected to remain the same, thus we will not delve into multiple visualizations at this time. For other examples of visualizations using ELMVIS+, the reader is encouraged to read the original paper [6].

This concludes our discussion of the ELMVIS application for GS. We have shown that GS can be effectively applied to the visualization problem and is an efficient local maximization solver. In the next section, we introduce another application, the special case of the traveling salesperson problem where the distance metric is the squared Euclidean distance.

3.2 TSP with Squared Euclidean Distance

The traveling salesperson problem (TSP) is possibly the best known combinatorial optimization problem. The TSP finds the minimum cost route through a given set of locations such that each location is visited exactly once and the traveler returns to the original location at the end of the tour. Despite the ease of statement, this is an NP-hard problem with a tremendous body of research devoted to all aspects of the topic.

As evidence to the difficulty of the QAP, the TSP can be formulated as a special case of the QAP. The next section will give the detailed formulation as well as additional theory. However, since this special case is so well known, there are many approaches to solve the TSP that are far superior than using QAP techniques.

With that in mind, we examine a single special case of the TSP, where the distance metric used is the squared Euclidean distance, and apply GS to this special case. This special case is not extensively studied and has the low-rank properties that allow the GS algorithm to be its most effective.

The most common application in the literature for the squared Euclidean distance is a wireless network. In a wireless network, the power needed to reach a distance r from a transmitter is roughly proportional to R^α, with $2 \leq \alpha \leq 6$. Funke et al. [52] suggest the idea of a network passing around a virtual token as a method to exchange information through the network, creating a TSP. They also give a 6-approximation algorithm to this non-metric version of the TSP, bounding the optimal value by six times the value of the minimum spanning tree (MST). This bound is improved by de Berg et al. [36] to a 5-approximation in the case that $\alpha = 2$, which is the case examined here. Van Nijnatten [125] did substantial theoretic analysis for $\alpha = 2$ as well. Finally, much of the analysis in this literature is based on the T^3 algorithm of Andreae and Bandelt [7] which made significant contributions to the TSP in the case that the distances follow a relaxed triangle inequality.

3.2.1 TSP Formulation

We will examine the symmetric traveling salesperson problem, thus the direction of travel does not matter (the distance from node i to j is equal to the distance from j to i). This is often formulated as a binary integer program [71] with decision variables $x_j \in \{0, 1\}$ indicating whether edge j is in the tour and c_j the cost of

traveling that edge. The complete formulation is

$$\min \qquad \frac{1}{2}\sum_{j=1}^{m}\sum_{k\in J(j)} c_k x_k \tag{3.22}$$

$$\text{s. t.} \qquad \sum_{k\in J(j)} x_k = 2 \qquad\qquad j = 1,\dots m \tag{3.23}$$

$$\sum_{j\in E(K)} x_j \leq |K| - 1 \qquad\qquad \forall K \subset \{1,\dots m\} \tag{3.24}$$

$$x_j \in \{0,1\} \qquad\qquad \forall j \in E \tag{3.25}$$

where $J(j)$ is the set of all edges connected to node j and $E(k)$ is the subset of edges connecting any proper nonempty subset of nodes K. Constraint (3.23) forces every node to have two edges connecting to it, one in and one out. Constraint (3.24) removes the possibility of subtours, one of the most difficult aspects of solving the TSP.

It is intuitively easy to see how the TSP might be formulated as a permutation problem. For permutation p, if index $p(i)$ in position i corresponds to visiting node $p(i)$ at timestep i, solving the TSP is equivalent to identifying the permutation that minimizes total distance. Written more formally,

$$\min \qquad d(p(n), p(1)) + \sum_{i=1}^{n-1} d(p(i), p(i+1)) \tag{3.26}$$

$$\text{s. t.} \qquad p \in S_n \tag{3.27}$$

where $d(i,j)$ is the distance from node i to node j. One can see that this summation looks very similar to the trace of PDP^T, where P is the permutation matrix corresponding to p and D is the matrix of distances where $D_{ij} = d(i,j)$. However, the

second indices are offset by one, thus if we premultiply by the adjacency matrix for an $n-$cycle, i.e. the square matrix of ones on the first superdiagonal and a one in the bottom left corner, we recover the above sum. If we call that shift matrix S, this formulation is identical to the trace form of the Koopmans-Beckmann QAP [82].

Recall that here we will examine the case of the TSP when the distance between nodes is the squared Euclidean distance. While there are quality approximation algorithms and theory for TSP when the distances satisfy the triangle inequality ($d(i,j) + d(j,k) \geq d(i,k)$), these cannot be used in this case. A simple example demonstrating the failure of the triangle inequality occurs on a line: if there are three nodes at $x_1 = 1$, $x_2 = 3$, and $x_3 = 5$, $d(x_1, x_2) + d(x_2, x_3) = 2^2 + 2^2 < 4^2 = d(x_1, x_3)$. Thus we are unable to use any conclusions as a result of the triangle inequality.

3.2.2 Computations

As mentioned above, instead of examining the general symmetric TSP, we look at the specific case where the distance between nodes is the squared Euclidean distance. This case of the TSP is ripe for GS as the distance matrix is only rank 4. Also, instead of using matrix multiplication, the shift matrix pre-multiplication can be done through much more efficient re-indexing instead; this is true of any type of TSP. Recall that the squared Euclidean distance between two points $\mathbf{x}, \mathbf{y} \in \mathbb{R}^p$ is

$$\|\mathbf{x} - \mathbf{y}\|_2^2 = \sum_{i=1}^{p} (x_i - y_j)^2 \tag{3.28}$$

With this in mind, we show below that the distance matrix is only rank 4. Let

X be the matrix of node locations, so $X = \begin{pmatrix} \mathbf{x}_1^T \\ \vdots \\ \mathbf{x}_n^T \end{pmatrix}$ where $\mathbf{x}_i$ is the location of the ith node.

$$D_{ij} = \|\mathbf{x}_i - \mathbf{x}_j\|_2^2 \tag{3.29}$$

$$= (\mathbf{x}_i - \mathbf{x}_j)^T(\mathbf{x}_i - \mathbf{x}_j) \tag{3.30}$$

$$= \mathbf{x}_i^T\mathbf{x}_i - 2\mathbf{x}_i^T\mathbf{x}_j + \mathbf{x}_j^T\mathbf{x}_j \tag{3.31}$$

If we define $(X * X)_i = \mathbf{x}_i^T\mathbf{x}_i$ so $X * X \in \mathbb{R}^n$, then the full distance matrix D can be written

$$D = (X * X)\mathbf{e}^T - 2XX^T + \mathbf{e}(X * X)^T \tag{3.32}$$

$$= ((X * X), X, \mathbf{e}) \begin{pmatrix} \mathbf{e}^T \\ -2X^T \\ (X * X)^T \end{pmatrix} \tag{3.33}$$

where $\mathbf{e} = (1, \ldots 1)^T$. Note that $p = 2$ as we will be looking at this problem in the plane. Since $X * X, \mathbf{e} \in \mathbb{R}^n$ and $X \in \mathbb{R}^{n\times 2}$, the expression in (3.33) is the product of two rank 4 matrices (as they are size $n \times 4$ and $4 \times n$, respectively) and thus D is also rank 4. Let the first matrix be denoted by Y and the second by Z^T, so $D = YZ^T$. This factorization enables us to use block GS. Also, we must only store two $n \times 4$ matrices instead of a full $n \times n$ matrix, a huge improvement in memory requirements. As instances of the TSP can be rather large, this is an important aspect of the algorithm.

Unfortunately, the shift matrix S is full rank (being a permutation matrix) and thus we are unable to factor it to take advantage of computational efficiency as we did with the ELMVIS application. However, as mentioned above, the pre-multiplication

of S can be instead done by reordering indices. This can be done very efficiently and allows significant speedup compared to a QAP with full-rank, non-permutation matrix F.

As in Section 2.2.1, we calculate the block objective function and gradient. We are unable to directly use the results from that section because S is not factorable and $D = YZ^T$ is not positive semidefinite. Again, without loss of generality we divide our matrices into blocks as follows:

$$S = \begin{pmatrix} S_{11} & S_{12} \\ S_{21} & S_{22} \end{pmatrix} \qquad P = \begin{pmatrix} P_1 & \\ & P_2 \end{pmatrix}$$
$$Y = \begin{pmatrix} Y_1 \\ Y_2 \end{pmatrix} \qquad Z = \begin{pmatrix} Z_1 \\ Z_2 \end{pmatrix}$$

Note that S_{11} and S_{22} are matrices with ones on the first super diagonal while S_{21} and S_{12} are matrices with a single one in the bottom left corner. When premultiplying another matrix, S_{11} and S_{22} represent shifting all rows up by one (with the bottom row zeroed out) while S_{12} and S_{21} represent taking just the top row and placing it at the bottom (with every other row zeroed out). We procede with the calculations.

$$\phi(P) = \operatorname{tr}\left(SPDP^T\right) \tag{3.34}$$
$$= \operatorname{tr}\left(\begin{pmatrix} S_{11} & S_{12} \\ S_{21} & S_{22} \end{pmatrix}\begin{pmatrix} P_1 & \\ & P_2 \end{pmatrix}\begin{pmatrix} Y_1 \\ Y_2 \end{pmatrix}\begin{pmatrix} Z_1^T & Z_2^T \end{pmatrix}\begin{pmatrix} P_1^T & \\ & P_2^T \end{pmatrix}\right) \tag{3.35}$$
$$= \operatorname{tr}\left(\begin{pmatrix} S_{11} & S_{12} \\ S_{21} & S_{22} \end{pmatrix}\begin{pmatrix} P_1Y_1 \\ P_2Y_2 \end{pmatrix}\begin{pmatrix} Z_1^TP_1^T & Z_2^TP_2^T \end{pmatrix}\right) \tag{3.36}$$
$$= \operatorname{tr}\left(\begin{pmatrix} S_{11} & S_{12} \\ S_{21} & S_{22} \end{pmatrix}\begin{pmatrix} P_1Y_1Z_1^TP_1^T & P_1Y_1Z_2^TP_2^T \\ P_2Y_2Z_1^TP_1^T & P_2Y_2Z_2^TP_2^T \end{pmatrix}\right) \tag{3.37}$$

$$= \operatorname{tr}\left(\begin{array}{cc} S_{11}P_1Y_1Z_1^TP_1^T + S_{12}P_2Y_2Z_1^TP_1^T & \\ & S_{21}P_1Y_1Z_2^TP_2^T + S_{22}P_2Y_2Z_2^TP_2^T \end{array} \right) \tag{3.38}$$

$$= \operatorname{tr}\left(S_{11}P_1Y_1Z_1^TP_1^T + S_{12}P_2Y_2Z_1^TP_1^T + S_{21}P_1Y_1Z_2^TP_2^T + S_{22}P_2Y_2Z_2^TP_2^T\right) \tag{3.39}$$

This looks similar to the results for the ELMVIS objective function with the exceptions that S is broken into four parts and $Y \neq Z$. We calculate the gradient below.

$$\mathrm{d}\phi(P) = \operatorname{tr}\left(S_{11}\ \mathrm{d}P_1Y_1Z_1^TP_1^T + S_{11}P_1Y_1Z_1^T\ \mathrm{d}P_1^T + S_{12}P_2Y_2Z_1^T\ \mathrm{d}P_1^T + S_{21}\ \mathrm{d}P_1Y_1Z_2^TP_2^T\right) \tag{3.40}$$

$$= \operatorname{tr}\left((S_{11}P_1Y_1Z_1^T + S_{11}^TP_1Z_1Y_1^T + S_{12}P_2Y_2Z_1^T + S_{21}^TP_2Z_2Y_1^T)\ \mathrm{d}P_1^T\right) \tag{3.41}$$

$$\Downarrow$$

$$\nabla_{P_1}\phi(P) = S_{11}P_1Y_1Z_1^T + S_{11}^TP_1Z_1Y_1^T + S_{12}P_2Y_2Z_1^T + S_{21}^TP_2Z_2Y_1^T \tag{3.42}$$

With the gradient calculation finished, we implement block GS and compare numerical results in the next section. Again, it will be crucial to increasing the speed of the algorithm to represent S_{11}, S_{12}, and S_{21} as reordering of indices instead of matrix multiplication.

With the gradient computations finished, the remaining implementation of block GS is very similar to the ELMVIS implementation, so at this point we move on to a discussion of the perfomance of the algorithm.

3.2.3 TSP Results

As with Section 3.1.3, this section is devoted to discussing the results of block GS, applied to instances of the squared Euclidean distance version of the TSP. We will use the National TSP data from the University of Waterloo's website [1], specifically looking at the instances for Egypt, Finland, Burma, and China in order of increasing size. Each data set has an optimal or near-optimal tour given as well, though these optimal tours are for the standard Euclidean TSP. In general, the local optima found by GS from a random initialization are still not remotely as good as the given tour under the squared Euclidean metric and thus these will not be used for comparison purposes. While slightly disappointing, this is not surprising as no local optimization procedure can hope to approach the global optimal for such a notoriously difficult problem without extraordinary luck.

As above, we present the average value of gradient evaluations, iterations of the block GS method, and time in seconds over 10 trials. However, the objective function here is the percentage decrease in the objective value, $(\phi_0 - \phi_f)/\phi_0$. The numerical experiments were performed on the same machine as above. The parenthetical numbers next to the country name indicate the size of the instance. The stopping tolerance for these runs was $\epsilon = 10^{-4}$.

There are a number of trends in Table 3.4 based on subset size m and the size of the problem n. In general, as m increases the quality of the local optimization improves. However, with improving performance comes substantially longer computation times. One can see that computation time behaves superlinearly with m, though

	Objective Function Decrease (%)				Gradient Evaluations			
Subset size m	250	500	1000	2000	250	500	1000	2000
Egypt (7146)	90.5	92.7	95.2	96.6	6266	3379	2234	1395
Finland (10639)	86.6	89.8	92.5	94.3	5585	3456	2130	1350
Burma (33708)	85.7	88.9	91.7	93.0	11493	6636	4273	2212
China (71009)	83.1	86.8	89.2	92.0	21760	11592	5972	4128
	Time (seconds)				Iterations			
Egypt (7146)	19.8	36.3	122	349	92.5	53.1	39.0	27.4
Finland (10639)	17.3	36.3	117	335	70.3	50.6	32.6	23.8
Burma (33708)	41.6	76.5	248	575	154	99.5	69.4	35.5
China (71009)	89.8	144	347	1085	292	173	101	68.2

Table 3.4: National TSP Dataset Comparison of Subset Size

the asymptotics are difficult to determine from only four data points per instance.

The number of gradient evaluations is closely tied to the total number of iterations as well as m. For $m = 250$, the number of gradient evaluations per iteration varied between 65 and 80 while for $m = 2000$, the same statistic varied between 50 and 65. This implies that for larger m fewer gradient evaluations are required for a given collection of indices.

If GS were incorporated into a metaheuristic designed to optimize the squared Euclidean distance TSP, the user would have to choose the best m given the tradeoff between speed and quality of solution. Either $m = 250$ or $m = 500$ seem likely choices as many local optimizations may need to be done, thus the significantly increased speed trumps the marginal improvement in quality.

Once again, a comparison of the local optimal objective function value and the best known Euclidean route's objective value (under the squared Euclidean distance)

is deliberately not given. These best known solutions tend to be at least one if not multiple orders of magnitude better than the local optima. This makes sense as the TSP and QAP are known to have a tremendous number of local optima and with a random initialization the likelihood that a local optimum is close to the best known solution is miniscule.

This concludes our investigation of the squared Euclidean TSP as an application for block GS. The final section of this chapter will apply the GS algorithm to one final new application, the generation of data with a prescribed correlation.

3.3 Correlated Data Synthesis

The final application developed for GS will be the generation of random data with specified correlation. Note that in this section, when the term "correlation" is used, it implicitly means Pearson correlation. Pearson correlation ρ between two samples $x, y \in \mathbb{R}^n$ is defined to be

$$\rho = \frac{\sum_{i=1}^{n}(x_i - \overline{x})(y_i - \overline{y})}{\sqrt{\sum_{i=1}^{n}(x_i - \overline{x})^2}\sqrt{\sum_{i=1}^{n}(y_i - \overline{y})^2}} \tag{3.43}$$

where $\overline{x}$ and $\overline{y}$ are the means of sample x and y, respectively.

For many simulation purposes it is important that randomly generated data sample follow a desired correlation structure. This structure is often constructed through the use of post-multiplication by the Cholesky factor; theoretical justification for this approach will be given in the following section. However, this approach has two limitations. First, if a specific distribution was used to create the uncorrelated data, there is no guarantee the resulting correlated data will follow the same

distribution. Second, the Cholesky approach will not create data with correlations that perfectly matches the desired correlation matrix.

The idea to improve on the Cholesky factor approach is not new. Inspiration for this GS application comes from Voechovsk and Novk [127], though similar ideas can be seen in [30, 69] as well. However, formulation of this problem as a QAP has not been done before. We will see that this approach, while slower, creates data samples whose correlation structures more closely match the desired correlation than samples generated using the Cholesky factor. Additionally, since the samples will be rearranged and the values left unaltered, the distribution of each sample will remain unchanged.

3.3.1 Pearson Correlations

In this and all remaining sections, it is assumed that any data discussed is normalized, i.e. $\|\mathbf{x}\|_2 = \sqrt{\sum_{i=1}^{n} x_i^2} = 1$, and the mean of a given sample is zero. For a non-normalized, non-zero mean vector y, this can be accomplished through first zeroing the mean and then normalizing the vector, as seen in (3.44).

$$\mathbf{x} = \frac{\mathbf{y} - \overline{\mathbf{y}}}{\|\mathbf{y} - \overline{\mathbf{y}}\|} \tag{3.44}$$

With these assumptions, the terms covariance and correlation become synonymous for our purposes. If covariances are desired which do not fall in [-1,1], the extremes of correlation, this can be accomplished through scaling after the generation of the correlated variables.

Next, we give theoretic justification for the effectiveness of the Cholesky factor

in generating correlated data samples. The covariance between two data samples X and Y is

$$cov(X, Y) = E[(X - E[X])(Y - E[Y])^T] \tag{3.45}$$

where $E[X]$ is the expected value of X. Since we have zeroed out the mean of our samples, the expected value is zero as well. Let $\Sigma = CC^T$ be the correlation matrix and its Cholesky factorization, where C is a lower diagonal matrix. Note that the expectated value operator $E[X]$ is linear. Assume $X \in \mathbb{R}^{n \times k}$ is a collection of k randomly distributed, zero mean, unit standard deviation, n-dimensional data samples. With this in mind, we demonstrate the efficacy of the Cholesky factor:

$$cov(CX, CX) = E[(CX)(CX)^T] \tag{3.46}$$

$$= E[CXX^TC^T] \tag{3.47}$$

$$= CE[XX^T]C^T \tag{3.48}$$

$$= CC^T \tag{3.49}$$

$$= \Sigma \tag{3.50}$$

The equality between lines (3.48)-(3.49) is due to the fact that $E[XX^T] = cov(X, X) = I$ since X is an uncorrelated random sample. Therefore we see that the use of the Cholesky factor theoretically ensures the desired correlation structure of our random data. In practice, numerical problems cause the correlation of the generated data to not exactly match the desired correlation and the individual samples of CX do not follow the same distributions as the samples in X.

We wish to generate data X where $cov(X, X) = \Sigma$ such that $\|T - \Sigma\|_F^2$ is

minimized, where T is the target correlation matrix. Note that $\Sigma_{ij} = X_i \cdot X_j = (X^T X)_{ij}$ where a singly-subscripted matrix indicates the column of the matrix. The idea is to randomly generate an uncorrelated data sample X and iteratively permute each sample of X to improve the correlations with all other samples of X. Thus, for a single sample $X_i \in \mathbb{R}^{n\times 1}$ our objective function of $P_i \in S_n$ can be seen below in (3.51). The fact that $T = T^T$ is used throughout.

$$\phi(P_i) = \sum_{j=1}^{k} (P_i X_i \cdot X_j - T_{ij})^2 \tag{3.51}$$

$$= \|(P_i X_i)^T X - T_i\|_F^2 \tag{3.52}$$

$$= \operatorname{tr}\left(((P_i X_i)^T X - T_i)^T ((P_i X_i)^T X - T_i)\right) \tag{3.53}$$

$$= \operatorname{tr}\left((X^T P_i X_i - T_i^T)(X_i^T P_i^T X - T_i)\right) \tag{3.54}$$

$$= \operatorname{tr}\left(X^T P_i X_i X_i^T P_i^T X - 2X^T P_i X_i T_i + T_i^T T_i\right) \tag{3.55}$$

The transition from summation to Frobenius norm (3.51)-(3.52) is valid since $P_i X_i \cdot X_j$ is equal to the jth entry of $(P_i X_i)^T X$.

Clearly, (3.55) is a QAP. However, it is not the purely quadratic QAP as we have seen in other applications; this QAP has both a quadratic and linear term, as well as a constant term which we ignore. In the next subsection, we calculate the gradient of this function and fully outline the GS implementation.

3.3.2 QAP Correlation Computations

This implementation of GS will be slightly different than previous examples. First of all, we will not use the block approach, limiting our application to $n \leq 5000$.

If a larger sample were desired, block GS approach could be applied, but block GS' efficacy has already been demonstrated; here we focus on iteratively applying GS to different samples.

Recall $X \in \mathbb{R}^{n \times k}$ is a collection of k samples, each of size n. Our approach is to reorder each sample one by one, thus improving the correlations associated with that sample. The strategy is shown below in Algorithm 3.2.

Algorithm 3.2 Correlated Data Generation

Input: Distribution, Target Correlation T, size n, samples k

Output: Data X with correlation $\Sigma \approx T$

1: Generate k random n-samples with given distribution

2: **while** Convergence not satisfied **do**

3: Choose sample $i \in \{1 \dots k\}$

4: Minimize $\phi(P_i)$ via GS ($\phi(P_i)$ given in 3.51)

5: **end while**

The implementation details for the GS algorithm itself are exactly the same as in the previous chapter, once again with the convergence criterion of minimal relative change in the objective function. For the data generation algorithm, the convergence criteria could be a desired accuracy, a number of iterations, or a lack of improvement in the objective function, depending on the implementation and purpose.

The choice of sample i has a significant effect on the performance of the algo-

rithm. For example, little progress would be made if i alternated between the first and second sample out of more than 10 samples. We introduce three approaches for the selection of i: completely random from 1 to k, repeated random permutations of 1 to k, or repeating the natural sequence $1,\ldots,k$. These strategies are referred to as "Random," "RandPerm," and "Seq" in the discussion of the results.

We introduce another way to end GS for this application. Despite the backtracking introduced in the previous chapter, it is possible when very few swaps are done each iteration that the objective value increases slightly. Thus, we measure the mean of the last five relative changes in objective value (the calculation done for the original convergence criterion) and if that mean is greater than some tolerance (10^{-2} is used) the algorithm ends. Therefore we have two separate convergence criteria, the original relative improvement criterion and the new mean relative improvement criterion, each of which can stop the algorithm.

As with the other applications, a brief calculation of the gradient of $\phi(P_i)$ is in order.

$$\mathrm{d}\phi(P_i) = \operatorname{tr}\left(X^T\ \mathrm{d}P_i X_i X_i^T P_i^T X + X^T P_i X_i X_i^T\ \mathrm{d}P_i^T X - 2X^T\ \mathrm{d}P_i X_i T_i\right) \tag{3.56}$$

$$= \operatorname{tr}\left((XX^T P_i X_i X_i^T - XT_i X_i^T)\ \mathrm{d}P_i^T\right) \tag{3.57}$$

$$\Downarrow$$

$$\nabla\phi = XX^T P_i X_i X_i^T - XT_i X_i^T \tag{3.58}$$

Note that the constant term in the gradient will actually change during the GS algorithm since the rows of X_i are permuted. However, that means that constant

term's columns need to be permuted but not recalculated every iteration.

As with previous gradient calculations, the grouping of the multiplication makes a tremendous difference in the efficiency of the computation. Recall that GS evaluates the gradient at the identity ($P_i = I$) every iteration. Thus, the most efficient way to compute the gradient is $\nabla\phi(P_i) = (X(X^T X_i))X_i^T$ which takes $O(n^2)$ time since $X \in \mathbb{R}^{n\times k}$ and $X_i \in \mathbb{R}^{n\times 1}$.

This application requires a target correlation matrix, so we need a way to generate a correlation matrix. We initialize a random symmetric matrix with unit diagonal and entry-wise maximum absolute value 1, then find the nearest correlation matrix to that random symmetric matrix. Finding the nearest correlation matrix is a previously studied problem, with alternating projection [70, 46] and Newton algorithms [101, 22] being the most common approaches. As one may expect, the alternating projection approach converges more slowly than the Newton algorithms, though due to the relatively small size of our correlation matrices ($k \leq 50$) this is not a problem.

We find the nearest correlation matrix via an alternating projection approach suggested by Higham [70]: alternate between projections to the nearest positive semi-definite matrix and the nearest unit diagonal matrix. The entire process for generating a random correlation matrix is given in Algorithm 3.3.

In step 4, $P_{PSD}(T)$ is the projection to the nearest positive semi-definite matrix. To do this we take the eigenvalue decomposition $T = VDV^T$ and replacing the diagonal matrix D with the entry-wise maximum of D and zero, $\max(D, 0)$. Step 5

Algorithm 3.3 Random Correlation Matrix

Input: Number of samples k, tolerance ϵ

Output: Random Correlation Matrix T

1: $T \leftarrow$ matrix of uniformly random entries from $[-1, 1]$ with unit diagonal

2: $T \leftarrow \frac{1}{2}(T + T^T)$, the nearest symmetric matrix

3: **while** $\lambda_k(T) < -\epsilon$ where $\lambda_k(T)$ is the minimum eigenvalue of T **do**

4: $\quad T \leftarrow P_{PSD}(T)$

5: $\quad T \leftarrow P_I(T)$

6: **end while**

7: **if** $\lambda_k(T) < 0$ **then**

8: $\quad T \leftarrow T + (\epsilon - \lambda_k(T))I$

9: **end if**

is the projection to the nearest matrix of unit diagonal, done by setting the diagonal to one. Finally, step 8 is done at the very end to ensure the correlation matrix is positive semi-definite. The matrix may not have an exact unit diagonal after this step, but each diagonal entry will be off by no more than 2ϵ, thus for $\epsilon < 10^{-6}$) it is acceptable. This entire process takes less than a second for any k that we discuss in this thesis, thus justifying our choice to use the simpler alternating approach over the faster though more complicated Newton methods.

This concludes the theoretical calculations for this application. In the next subsection we will give experimental results comparing the quality and speed of generating random correlated data via the QAP formulation vs. the Cholesky method.

3.3.3 Results

In this section we give experimental results of applying GS to the correlated data generation QAP application. There are four parameters to investigate the effects on the performance of the algorithm: size of sample n, number of samples k, schedule of sample optimized, and the type of distribution used to generate the original, uncorrelated data. We choose to compare the effects of n and the sample schedule simultaneously while holding the other two static and vice versa. "% Relative Objective Value" in the tables is equal to 100*(GS objective value)/(Cholesky objective value); thus a % relative objective value of less than 100 implies GS was superior. We first compare the effects of varying n and the sample schedule in Table 3.5, which shows the mean of 10 trials.

	Random	RandPerm	Seq	Random	RandPerm	Seq
n	% Relative Objective Value			Gradient Evaluations		
500	39.8	77.6	62.5	192	170	177
1000	29.6	35.7	46.4	328	320	315
2000	20.7	24.2	19.5	561	541	598
3000	90.8	56.1	39.3	543	591	559
n	Time (seconds)			Function Evaluations		
500	1.54	1.38	1.41	1324	1148	1215
1000	12.5	12.1	12.1	2573	2534	2471
2000	87.0	84.7	92.7	4978	4771	5330
3000	203	215	206	4979	5456	5032
$\epsilon = 10^{-5}$, Triangle Distribution, $k = 10$						

% Relative Objective Value = 100*(GS objective value)/(Cholesky objective value)

Table 3.5: Correlation QAP Comparison of n and Schedule

There was an extreme outlier in the $n = 1000$ random schedule that was removed from analysis. The algorithm finished in a fraction of a second with a terrible result, thus indicating an improper convergence and justifying removal from the data set.

Additionally, a counterintuitive result is found in the $n = 3000$ objective function values; these values are not very good despite the fact the objective value trend seems to be improvement accompanying an increase in n. This is due to GS finishing from lack of sufficient relative improvement in the objective function, i.e. the traditional convergence criterion. For $n \in \{500, 1000, 2000\}$, the algorithm finished from an increase in the objective function instead. We suspect that if the convergence tolerance (set at $\epsilon = 10^{-5}$ here) was more restrictive, additional improvement would

have been made when $n = 3000$.

For $n \in \{500, 1000, 2000\}$, an increase in n corresponds to a more disparate result between GS and Cholesky, with GS improving relative to Cholesky. The number of function evaluations (one for every backtrack) and gradient evaluations (one for every iteration of GS) increase with n, as does the time required to perform the calculations. Time scales relatively poorly with n, appearing to increase at more than $O(n^2)$. Since the gradient and function evaluations scale with $O(n^2)$, this is likely due to the additional iterations required in GS.

One interesting result is that the sample schedule does not seem to make a significant difference in the performance of GS. In no category does sample schedule make a consistent difference, though we note that in the likely premature convergence for $n = 3000$, the sequential schedule was much better than the other two and the random schedule was barely an improvement on the Cholesky method.

Next, we investigate the effects of varying the type of distribution and number of samples k on correlated data generation. These results are summarized in Table 3.6. The four distributions compared are the triangular, normal, beta, and uniform distributions. For each distribution we chose to use the Matlab default as we scaled and shifted the data after generation to enforce zero mean and unit length. The default triangular distribution has minimum 0, mean 0.5, and maximum 1. The default normal distribution has mean 0 and standard deviation 1. The default beta distribution is a uniform distribution so instead we use the parameters $\alpha = \beta = 4$. The default uniform distribution has minimum 0 and maximum 1.

	Triangular	Normal	Beta	Uniform	Triangular	Normal	Beta	Uniform
k	% Relative Objective Value				Gradient Evaluations			
5	28.6	63.5	41.9	27.7	59	56	70	80
10	38.7	65.4	24.5	14.6	246	248	323	486
20	15.2	34.4	15.7	10.9	1345	842	1514	2123
50	21.6	27.3	28.1	16.6	2359	2411	1861	2389
k	Time (seconds)				Function Evaluations			
5	2.8	2.65	3.16	3.45	433	433	537	578
10	10.2	10.0	12.0	16.6	1924	1931	2521	3953
20	47.1	31.4	51.3	69.4	11344	6735	12901	18861
50	85.4	91.5	71.3	87.0	18083	18803	13791	18417
Relative Improvement tolerance $\epsilon = 10^{-6}$, $n = 1000$, Sequential Sample Schedule								

% Relative Objective Value = 100*(GS objective value)/(Cholesky objective value)

Table 3.6: Correlation QAP Comparison of k and Distribution

Table 3.6 is summarized by median over 10 trials. We chose median instead of mean because for larger k occasionally the GS algorithm would terminate much too early, creating significant outliers. Instead of removing them, we account for them by using an average less affected by outliers, i.e. the median. The early termination is evident due to both the rapid conclusion of the algorithm as well as the poor results, thus in practice the algorithm would simply be re-run.

One notable result in Table 3.6 is that there is no clear relationship between k and % relative objective value. Certainly for larger k GS takes more time, gradient evaluations, and function evaluations, but there is no clear effect on the relative quality of the results. However, overall GS resulted in an improvement of about 35-90%.

Note that all runs with $k \leq 10$ and most runs for $k = 20$ terminated due to an increasing objective function while for $k = 50$ the trials terminated from insufficient objective function improvement. As with the $n = 3000$ case in Table 3.5, this indicates that there may have been slightly more improvement possible in the $k = 50$ case if the GS algorithm were allowed to run for a longer duration.

The choice of distribution has a substantial effect on the quality of the results. For every value of k, a uniform distribution initialization resulted in the smallest % relative objective value while a normal distribution initialization resulted in the largest. The triangular and beta distributions fell between the normal and uniform distributions.

There is a direct correlation between the time, function evaluations, and gradient evaluations, with 7-9 function evaluations for every gradient evaluation on average. With an increase in k, there was a corresponding increase in gradient and function evaluations per second, i.e. the larger the number of samples, the more iterations per second. We hypothesize this is because each iteration converges more quickly with a higher k thus allowing more iterations to occur in total, though we acknowledge this result is rather counterintuitive. Note, as with other patterns observed, these results do not hold for $k = 50$, indicating there is some interaction between the type of convergence and the results.

This concludes our analysis of the correlated data generation application for GS. As this was the last novel large-scale QAP application, this also concludes our discussion of GS results. In the final chapter we will give an overview of our results

and propose future work.

CHAPTER 4
CONCLUSIONS AND FUTURE WORK

The quadratic assignment problem is one of the classic combinatorial optimization problems. It is extremely difficult to solve due to the large space of solutions inherent in assignment problems as well as the nonlinear objective function. As it is so difficult, the QAP has only been solved optimally for small ($n \leq 40$) instances of the problem with solutions to larger instances found using heuristic techniques. Even then, the largest QAP in the QAPLIB, a popular repository for instances of the QAP, is no larger than $n = 256$.

In this thesis we propose a local optimization technique for extremely large ($1000 \leq n \leq 70000$) instances of the QAP. This technique, GradSwaps, uses the first order Taylor approximation to optimize over the 2-swap neighborhood. The use of the first order approximation is accompanied by significant computational advantage and speedup. Additionally, instead of performing individual swaps at once, multiple swaps are done simultaneously to further reduce computational work. Finally, the applications developed use low-rank data matrices, allowing for efficient calculation of the gradient.

In Chapter 1 an extensive background of the QAP was given. Numerous different formulations and their respective advantages were discussed prior to a survey of past applications of the QAP. Many different solution methods were also discussed: exact methods (branch and bound, Bender's decomposition), local search, and heuristics. Among the most popular heuristics for the QAP are tabu search,

genetic algorithms, simulated annealing, and GRASP.

Chapter 2 gave a detailed theoretic description of the new local optimization technique, GradSwaps. GS uses the first order Taylor approximation to determine which moves in the 2-swap neighborhood are beneficial. As approximation is not perfect, the accuracy of the FOA was quantified by calculating the exact swap difference in the objective function and then bounding the difference between the exact calculation and the FOA. This difference proved to be largely insignificant for very large instances of the QAP, the main focus in this thesis. Important implementation details were given as well, such as the swap selection process, reordering indices instead of multiplication by permutation matrix, and backtracking to ensure improvement in the solution. Finally, we developed the block GS, a way to extend the GS to larger QAP while avoiding memory storage problems.

Ultimately, in Chapter 3 three large, low-rank QAP applications of GS were explored. The first, ELMVIS, used an extreme learning machine to visualize large amounts of data. We used this application to visualize the MNIST handwritten data set, demonstrating the ability to scale the GS from $n = 1000$ to $n = 60000$. Our second application was the well-known traveling salesperson problem, though we examined the less studied squared Euclidean distance version of the TSP. Here we applied GS to four different instances, using national geographic data for Burma (Myanmar), Egypt, Finland, and China. Finally, we showed that GS can be used to generate random data with desired distribution and higher accuracy correlation than the usual method involving the Cholesky factor.

There is substantial opportunity for future work involving GS. In addition to reduced computational work with lower rank data matrices, the relative improvement in the objective function also increases with lower rank. It is unclear why this is the case and an explanation to bolster the experimental evidence would be a significant boon to the theoretical foundation of GS. Additionally, GS is still only a local optimization technique; a future step could be the incorporation of GS into a heuristic such as a genetic algorithm.

GS could easily be applied to other difficult nonlinear combinatorial optimization problems. Most approaches to solving nonlinear combinatorial problems rely on linearization of the function which results in possibly intractably many additional constraints. The application of GS requires only the calculation of the gradient, a relatively simple task, and would allow efficient local optimization on a scale that has not been previously investigated.

REFERENCES

[1] TSP test data http://www.math.uwaterloo.ca/tsp/data/, Accessed November 2017.

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

[3] D. Adolphson and T. Hu. Optimal linear ordering. *SIAM Journal on Applied Mathematics*, 25(3):403–423, 1973.

[4] Ravindra K. Ahuja, Krishna Jha, James B. Orlin, and Dushyant Sharma. Very large-scale neighborhood search for the quadratic assignment problem. *INFORMS Journal on Computing*, 19(4):646–657, 2007.

[5] Ravindra K. Ahuja, James B. Orlin, and Ashish Tiwari. A greedy genetic algorithm for the quadratic assignment problem. *Computers & Operations Research*, 27(10):917–934, 2000.

[6] Anton Akusok, Stephen Baek, Yoan Miche, Kaj-Mikael Björk, Rui Nian, Paula Lauren, and Amaury Lendasse. ELMVIS+: Fast nonlinear visualization technique based on cosine distance and extreme learning machines. *Neurocomputing*, 205:247–263, 2016.

[7] Thomas Andreae and Hans-Jürgen Bandelt. Performance guarantees for approximation algorithms depending on parametrized triangle inequalities. *SIAM Journal on Discrete Mathematics*, 8(1):1–16, 1995.

[8] S. Anily and A. Federgruen. Simulated annealing methods with general acceptance probabilities. *Journal of Applied Probability*, 24(3):657–667, 1987.

[9] Kurt Anstreicher, Nathan Brixius, Jean-Pierre Goux, and Jeff Linderoth. Solving large quadratic assignment problems on computational grids. *Mathematical Programming*, 91(3):563–588, 2002.

[10] Kurt M. Anstreicher and Nathan W. Brixius. A new bound for the quadratic assignment problem based on convex quadratic programming. *Mathematical Programming*, 89(3):341–357, 2001.

[11] Kurt M. Anstreicher and Nathan W. Brixius. Solving quadratic assignment problems using convex quadratic programming relaxations. *Optimization Methods and Software*, 16(1):49–68, 2001.

[12] Esther M. Arkin, Refael Hassin, and Maxim Sviridenko. Approximating the maximum quadratic assignment problem. *Information Processing Letters*, 77(1):13–16, 2001.

[13] Nils Aall Barricelli. Symbiogenetic evolution processes realized by artificial methods. *Methodos*, 9(35):143–182, 1957.

[14] Nils Aall Barricelli and others. Esempi numerici di processi di evoluzione. *Methodos*, 6(21):45–68, 1954.

[15] Roberto Battiti and Giampietro Tecchiolli. The reactive tabu search. *ORSA Journal on Computing*, 6(2):126–140, 1994.

[16] Mokhtar S. Bazaraa and Hanif D. Sherali. Benders' partitioning scheme applied to a new formulation of the quadratic assignment problem. *Naval Research Logistics Quarterly*, 27(1):29–41, 1980.

[17] Mokhtar S. Bazaraa and Hanif D. Sherali. On the use of exact and heuristic cutting plane methods for the quadratic assignment problem. *Journal of the Operational Research Society*, 33(11):991–1003, 1982.

[18] Una Benlic and Jin-Kao Hao. Breakout local search for the quadratic assignment problem. *Applied Mathematics and Computation*, 219(9):4800–4815, 2013.

[19] Una Benlic and Jin-Kao Hao. Memetic search for the quadratic assignment problem. *Expert Systems with Applications*, 42(1):584–595, 2015.

[20] Béla Bollobás. *Extremal graph theory*. Courier Corporation, 2004.

[21] Immanuel M. Bomze, Marco Budinich, Panos M. Pardalos, and Marcello Pelillo. The maximum clique problem. In *Handbook of combinatorial optimization*, pages 1–74. Springer, 1999.

[22] Rüdiger Borsdorf and Nicholas J. Higham. A preconditioned newton algorithm for the nearest correlation matrix. *IMA Journal of Numerical Analysis*, 30(1):94–107, 2009.

[23] Aydn Buluç, Henning Meyerhenke, Ilya Safro, Peter Sanders, and Christian Schulz. Recent advances in graph partitioning. In *Algorithm Engineering*, Lecture Notes in Computer Science, pages 117–158. Springer, Cham, 2016. DOI: 10.1007/978-3-319-49487-6_4.

[24] R. E. Burkard and J. Offermann. Entwurf von schreibmaschinentastaturen mittels quadratischer zuordnungsprobleme. *Zeitschrift für Operations Research*, 21(4):B121–B132, 1977.

[25] R. E. Burkard and F. Rendl. A thermodynamically motivated simulation procedure for combinatorial optimization problems. *European Journal of Operational Research*, 17(2):169–174, 1984.

[26] Rainer E. Burkard. Quadratic assignment problems. In Panos M. Pardalos, Ding-Zhu Du, and Ronald L. Graham, editors, *Handbook of Combinatorial Optimization*, pages 2741–2814. Springer New York, 2013. DOI: 10.1007/978-1-4419-7997-1_22.

[27] R.E. Burkard, S.E. Karisch, and F. Rendl. QAPLIB-a quadratic assignment problem library. *Journal of Global Optimization*, 10:391–403, 1997.

[28] Vladimír Černỳ. Thermodynamical approach to the traveling salesman problem: An efficient simulation algorithm. *Journal of Optimization Theory and Applications*, 45(1):41–51, 1985.

[29] Jaishankar Chakrapani and JADRANKA Skorin-Kapov. A constructive method to improve lower bounds for the quadratic assignment problem. *Quadratic assignment and related problems*, 16:161–171, 1994.

[30] Dimos C. Charmpis and Panayiota L. Panteli. A heuristic approach for the generation of multivariate random samples with specified marginal distributions and correlation matrix. *Computational Statistics*, 19(2):283, 2004.

[31] Chun-houh Chen, Wolfgang Karl Härdle, and Antony Unwin. *Handbook of data visualization*. Springer Science & Business Media, 2007.

[32] Nicos Christofides and Enrique Benavent. An exact algorithm for the quadratic assignment problem on a tree. *Operations Research*, 37(5):760–768, 1989.

[33] Jack L. Crosby. *Computer simulation in genetics.* 1973.

[34] Michał Czapiński. An effective parallel multistart tabu search for quadratic assignment problem on CUDA platform. *Journal of Parallel and Distributed Computing*, 73(11):1461–1468, 2013.

[35] George Dantzig, Ray Fulkerson, and Selmer Johnson. Solution of a large-scale traveling-salesman problem. *Journal of the operations research society of America*, 2(4):393–410, 1954.

[36] Mark de Berg, Fred van Nijnatten, René Sitters, Gerhard J. Woeginger, and Alexander Wolff. The traveling salesman problem under squared euclidean distances. *arXiv:1001.0236 [cs]*, 2010.

[37] Howard B. Demuth, Mark H. Beale, Orlando De Jess, and Martin T. Hagan. *Neural Network Design*. Martin Hagan, 2nd edition, 2014.

[38] Tansel Dokeroglu. Hybrid teaching–learning-based optimization algorithms for the quadratic assignment problem. *Computers & Industrial Engineering*, 85:86–101, 2015.

[39] M. Dorigo and G. Di Caro. Ant colony optimization: a new meta-heuristic. In *Proceedings of the 1999 Congress on Evolutionary Computation-CEC99 (Cat. No. 99TH8406)*, volume 2, page 1477 Vol. 2, 1999.

[40] Marco Dorigo and Luca Maria Gambardella. Ant colony system: a cooperative learning approach to the traveling salesman problem. *IEEE Transactions on evolutionary computation*, 1(1):53–66, 1997.

[41] C. S. Edwards. A branch and bound algorithm for the koopmans-beckmann quadratic assignment problem. In *Combinatorial Optimization II*, Mathematical Programming Studies, pages 35–52. Springer, Berlin, Heidelberg, 1980. DOI: 10.1007/BFb0120905.

[42] C. S. Edwards. The derivation of a greedy approximator for the koopmans-beckmann quadratic assignment problem. In *Proceedings of the 77-th Combinatorial Programming Conference (CP77)*, pages 55–86, 1997.

[43] E.L. Lawler, J.K. Lenstra (Jan Karel), A.H.G. Rinnooy Kan, D.B. Shmoys, and Combinatorial Optimization and Algorithmics. *The Traveling Salesman Problem; A Guided Tour of Combinatorial Optimization*. Wiley, Chichester, 1985.

[44] Alwalid N. Elshafei. Hospital layout as a quadratic assignment problem. *Journal of the Operational Research Society*, 28(1):167–179, 1977.

[45] Güneş Erdoğan and Barbaros Tansel. A branch-and-cut algorithm for quadratic assignment problems based on linearizations. *Computers & Operations Research*, 34(4):1085–1106, 2007.

[46] René Escalante and Marcos Raydan. *Alternating projection methods*. SIAM, 2011.

[47] Thomas A. Feo and Mauricio GC Resende. A probabilistic heuristic for a computationally difficult set covering problem. *Operations research letters*, 8(2):67–71, 1989.

[48] Gerd Finke, Rainer E. Burkard, and Franz Rendl. Quadratic assignment problems. *Surveys in Combinatorial Optimization*, 132:61–82, 1987.

[49] Alex Fraser and Donald Burnell. Computer models in genetics. *Computer models in genetics.*, 1970.

[50] A. M. Frieze and J. Yadegar. On the quadratic assignment problem. *Discrete Applied Mathematics*, 5(1):89–98, 1983.

[51] A. M Frieze, J Yadegar, S El-Horbaty, and D Parkinson. Algorithms for assignment problems on an array processor. *Parallel Computing*, 11(2):151–162, 1989.

[52] Stefan Funke, Sören Laue, Zvi Lotker, and Rouven Naujoks. Power assignment problems in wireless communication: Covering points by disks, reaching few receivers quickly, and energy-efficient travelling salesman tours. *Ad Hoc Networks*, 9(6):1028–1035, 2011.

[53] Luca Maria Gambardella, É D. Taillard, and Marco Dorigo. Ant colonies for the quadratic assignment problem. *Journal of the operational research society*, 50(2):167–176, 1999.

[54] Xavier Gandibleux, Xavier Delorme, and Vincent T'Kindt. An ant colony optimisation algorithm for the set packing problem. *Ant Colony Optimization and Swarm Intelligence*, pages 478–481, 2004.

[55] P. Gilmore. Optimal and suboptimal algorithms for the quadratic assignment problem. *Journal of the Society for Industrial and Applied Mathematics*, 10(2):305–313, 1962.

[56] Andrej Gisbrecht and Barbara Hammer. Data visualization by nonlinear dimensionality reduction. *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery*, 5(2):51–73, 2015.

[57] Fred Glover. Tabu search—part i. *ORSA Journal on Computing*, 1(3):190–206, 1989.

[58] Fred Glover. Tabu search—part II. *ORSA Journal on computing*, 2(1):4–32, 1990.

[59] Fred Glover and Manuel Laguna. Tabu search. In *Handbook of Combinatorial Optimization*, pages 3261–3362. Springer, 2013.

[60] Robert L. Goldstone, Franco Pestilli, and Katy Börner. Self-portraits of the brain: cognitive science, data visualization, and communicating brain structure and function. *Trends in Cognitive Sciences*, 19(8):462–474, 2015.

[61] Alexander N. Gorban, Balázs Kégl, Donald C. Wunsch, and Andrei Y. Zinovyev. *Principal manifolds for data visualization and dimension reduction*, volume 58. Springer, 2008.

[62] Serigne Gueye, Sophie Michel, and Mahdi Moeini. Adjacency variables formulation for the minimum linear arrangement problem. In *Operations Research and Enterprise Systems*, Communications in Computer and Information Science, pages 95–107. Springer, Cham, 2014.

[63] 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.

[64] Scott William Hadley. *Continuous optimization approaches for the quadratic assignment problem*. Phd thesis, University of Waterloo, 1989.

[65] Richard W. Hamming. Error detecting and error correcting codes. *Bell Labs Technical Journal*, 29(2):147–160, 1950.

[66] Waqar Haque, Bonnie Urquhart, and Emery Berg. Visualization of services availability: Building healthy communities. In *New Perspectives in Information Systems and Technologies, Volume 2*, Advances in Intelligent Systems and Computing, pages 355–364. Springer, Cham, 2014. DOI: 10.1007/978-3-319-05948-8_34.

[67] Refael Hassin and Shlomi Rubinstein. An approximation algorithm for maximum triangle packing. *Discrete Applied Mathematics*, 154(6):971–979, 2006.

[68] Robert Hecht-Nielsen. Theory of the backpropagation neural network. *Neural Networks*, 1:445–448, 1988.

[69] Hannes Helgason, Vladas Pipiras, and Patrice Abry. Synthesis of multivariate stationary series with prescribed marginal distributions and covariance using circulant matrix embedding. *Signal Processing*, 91(8):1741–1758, 2011.

[70] Nicholas J. Higham. Computing the nearest correlation matrix—a problem from finance. *IMA journal of Numerical Analysis*, 22(3):329–343, 2002.

[71] Karla L. Hoffman, Manfred Padberg, and Giovanni Rinaldi. Traveling salesman problem. In *Encyclopedia of operations research and management science*, pages 1573–1578. Springer, 2013.

[72] Guang-Bin Huang, Qin-Yu Zhu, and Chee-Kheong Siew. Extreme learning machine: Theory and applications. *Neurocomputing*, 70(1):489–501, 2006.

[73] Tabitha James, Cesar Rego, and Fred Glover. A cooperative parallel tabu search algorithm for the quadratic assignment problem. *European Journal of Operational Research*, 195(3):810–826, 2009.

[74] Michael Jünger and Volker Kaibel. Box-inequalities for quadratic assignment polytopes. *Mathematical Programming*, 91(1):175–197, 2001.

[75] Stefan E. Karisch. *Nonlinear approaches for quadratic assignment and graph partition problems.* PhD thesis, Karl-Franzens-Universitaet Graz, 1997.

[76] Stefan E. Karisch and Franz Rendl. Lower bounds for the quadratic assignment problem via triangle decompositions. *Mathematical Programming*, 71(2):137–151, 1995.

[77] Richard M. Karp. Reducibility among combinatorial problems. In *Complexity of Computer Computations*, The IBM Research Symposia Series, pages 85–103. Springer, Boston, MA, 1972. DOI: 10.1007/978-1-4684-2001-2_9.

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

[79] Scott Kirkpatrick, C. Daniel Gelatt, Mario P. Vecchi, and others. Optimization by simulated annealing. *Science*, 220(4598):671–680, 1983.

[80] Cole Nussbaumer Knaflic. *Storytelling with Data: A Data Visualization Guide for Business Professionals*. John Wiley & Sons, 2015. Google-Books-ID: retR-CgAAQBAJ.

[81] Julia Kokina, Dessislava Pachamanova, and Andrew Corbett. The role of data visualization and analytics in performance management: Guiding entrepreneurial growth decisions. *Journal of Accounting Education*, 38:50–62, 2017.

[82] Tjalling C. Koopmans and Martin Beckmann. Assignment problems and the location of economic activities. *Econometrica*, 25(1):53–76, 1957.

[83] Jakob Krarup and Peter Mark Pruzan. Computer-aided layout design. In *Mathematical Programming in Use*, Mathematical Programming Studies, pages 75–94. Springer, Berlin, Heidelberg, 1978. DOI: 10.1007/BFb0120827.

[84] Eugene L. Lawler. The quadratic assignment problem. *Management Science*, 9(4):586–599, 1963.

[85] Yann LeCun, Corinna Cortes, and Chris Burges. MNIST handwritten digit database http://yann.lecun.com/exdb/mnist/, Accessed March 2017.

[86] Thomas Lengauer. *Combinatorial algorithms for integrated circuit layout*. Springer Science & Business Media, 2012.

[87] Yong Li, Panos M. Pardalos, K. G. Ramakrishnan, and Mauricio GC Resende. Lower bounds for the quadratic assignment problem. *Annals of Operations Research*, 50(1):387–410, 1994.

[88] Shen Lin. Computer solutions of the traveling salesman problem. *The Bell system technical journal*, 44(10):2245–2269, 1965.

[89] M. Lundy and A. Mees. Convergence of an annealing algorithm. *Mathematical Programming*, 34(1):111–124, 1986.

[90] Jan R. Magnus and H. Neudecker. Matrix differential calculus with applications to simple, hadamard, and kronecker products. *Journal of Mathematical Psychology*, 29(4):474–492, 1985.

[91] Nicholas Metropolis, Arianna W. Rosenbluth, Marshall N. Rosenbluth, Augusta H. Teller, and Edward Teller. Equation of state calculations by fast computing machines. *The journal of chemical physics*, 21(6):1087–1092, 1953.

[92] Alfonsas Misevičius. A modified simulated annealing algorithm for the quadratic assignment problem. *Informatica*, 14(4):497–514, 2003.

[93] Alfonsas Misevicius. An improved hybrid genetic algorithm: new results for the quadratic assignment problem. *Knowledge-Based Systems*, 17(2):65–73, 2004.

[94] Alfonsas Misevičius. An extension of hybrid genetic algorithm for the quadratic assignment problem. *Information Technology and Control*, 33(4), 2015.

[95] Manfred Padberg and Minendra P. Rijal. Quadratic assignment polytopes. In *Location, Scheduling, Design and Integer Programming*, International Series in Operations Research & Management Science, pages 151–166. Springer, Boston, MA, 1996. DOI: 10.1007/978-1-4613-1379-3_7.

[96] MANFRED W. Padberg and M. P. Rijal. Location, scheduling, design and integer programming. *Journal of The Operational Research Society*, 49(1):95–95, 1998.

[97] L. Pardalos and M. Resende. A greedy randomized adaptive search procedure for the quadratic assignment problem. *DIMACS Series on Discrete Mathematics and Theoretical Computer Science*, 16:237–261, 1994.

[98] P. Pardalos, K. Ramakrishnan, M. Resende, and Y. Li. Implementation of a variance reduction-based lower bound in a branch-and-bound algorithm for the quadratic assignment problem. *SIAM Journal on Optimization*, 7(1):280–294, 1997.

[99] P. M. Pardalos, L. S. Pitsoulis, and M. G. C. Resende. A parallel grasp implementation for the quadratic assignment problem. In *Parallel Algorithms for Irregular Problems: State of the Art*, pages 115–133. Springer, Boston, MA, 1995. DOI: 10.1007/978-1-4757-6130-6_6.

[100] Panos M. Pardalos and Jue Xue. The maximum clique problem. *Journal of Global Optimization*, 4(3):301–328, 1994.

[101] H. Qi and D. Sun. A quadratically convergent newton method for computing the nearest correlation matrix. *SIAM Journal on Matrix Analysis and Applications*, 28(2):360–385, 2006.

[102] Alain Quilliot and Djamal Rebaine. Linear arrangement problems and interval graphs. In *Combinatorial Optimization*, Lecture Notes in Computer Science, pages 359–370. Springer, Cham, 2014.

[103] Gerhard Reinelt. TSPLIB—a traveling salesman problem library. *ORSA journal on computing*, 3(4):376–384, 1991.

[104] Franz Rendl and Henry Wolkowicz. Applications of parametric programming and eigenvalue maximization to the quadratic assignment problem. *Mathematical Programming*, 53(1):63–78, 1992.

[105] Frank Rosenblatt. The perceptron: A probabilistic model for information storage and organization in the brain. *Psychological review*, 65(6):386, 1958.

[106] Sartaj Sahni and Teofilo Gonzalez. P-complete approximation problems. *Journal of the ACM (JACM)*, 23(3):555–565, 1976.

[107] Gamal Abd El-Nasser A. Said, Abeer M. Mahmoud, and El-Sayed M. El-Horbaty. A comparative study of meta-heuristic algorithms for solving quadratic assignment problem. *arXiv preprint arXiv:1407.4863*, 2014.

[108] Mohamed Saifullah Hussin and Thomas Stützle. Tabu search vs. simulated annealing as a function of the size of quadratic assignment problem instances. *Computers & Operations Research*, 43:286–291, 2014.

[109] Andrew I. Schein, Lawrence K. Saul, and Lyle H. Ungar. A generalized linear model for principal component analysis of binary data. In *AISTATS*, volume 3, page 10, 2003.

[110] Hanif D. Sherali and Warren P. Adams. A hierarchy of relaxations between the continuous and convex hull representations for zero-one programming problems. *SIAM Journal on Discrete Mathematics*, 3(3):411–430, 1990.

[111] Y. Shiloach. A minimum linear arrangement algorithm for undirected trees. *SIAM Journal on Computing*, 8(1):15–32, 1979.

[112] Ubonrat Siripatrawan and Bruce R. Harte. Data visualization of salmonella typhimurium contamination in packaged fresh alfalfa sprouts using a kohonen network. *Talanta*, 136:128–135, 2015.

[113] Jadranka Skorin-Kapov. Tabu search applied to the quadratic assignment problem. *ORSA Journal on computing*, 2(1):33–45, 1990.

[114] Jadranka Skorin-Kapov. Extensions of a tabu search adaptation to the quadratic assignment problem. *Computers & Operations Research*, 21(8):855–865, 1994.

[115] L. Steinberg. The backboard wiring problem: A placement algorithm. *SIAM Review*, 3(1):37–50, 1961.

[116] Thomas Stützle. Iterated local search for the quadratic assignment problem. *European Journal of Operational Research*, 174(3):1519–1539, 2006.

[117] Thomas Stützle and Rubén Ruiz. Iterated local search. *Handbook of Heuristics*, pages 1–27, 2017.

[118] Maxim Sviridenko and Justin Ward. Large neighborhood local search for the maximum set packing problem. In *Automata, Languages, and Programming*, Lecture Notes in Computer Science, pages 792–803. Springer, Berlin, Heidelberg, 2013.

[119] Éric Taillard. Robust taboo search for the quadratic assignment problem. *Parallel computing*, 17(4):443–455, 1991.

[120] E.-G. Talbi, Olivier Roux, Cyril Fonlupt, and Denis Robillard. Parallel ant colonies for the quadratic assignment problem. *Future Generation Computer Systems*, 17(4):441–449, 2001.

[121] David M. Tate and Alice E. Smith. A genetic approach to the quadratic assignment problem. *Computers & Operations Research*, 22(1):73–83, 1995.

[122] Alexandru C. Telea. *Data visualization: principles and practice*. CRC Press, 2014.

[123] Helga Thorvaldsdóttir, James T. Robinson, and Jill P. Mesirov. Integrative genomics viewer (IGV): high-performance genomics data visualization and exploration. *Briefings in Bioinformatics*, 14(2):178–192, 2013.

[124] Stefka Tyanova, Tikira Temu, Pavel Sinitcyn, Arthur Carlson, Marco Y. Hein, Tamar Geiger, Matthias Mann, and Jürgen Cox. The perseus computational platform for comprehensive analysis of (prote)omics data. *Nature Methods*, 13(9):731–740, 2016.

[125] Fred van Nijnatten. Range assignment with directional antennas. Master's thesis, Technische Universiteit Eindhoven, 2008.

[126] Juha Vesanto. SOM-based data visualization methods. *Intelligent Data Analysis*, 3(2):111–126, 1999.

[127] M. Vořechovský and D. Novák. Correlation control in small-sample monte carlo type simulations i: A simulated annealing approach. *Probabilistic Engineering Mechanics*, 24(3):452–462, 2009.

[128] Mickey R. Wilhelm and Thomas L. Ward. Solving quadratic assignment problems by 'simulated annealing': IIE transactions: Vol 19, no 1.

[129] H. P. Yap. Packing of graphs-a survey. *Annals of Discrete Mathematics*, 38:395–404, 1988.

[130] Nathan Yau. *Data points: Visualization that means something*. John Wiley & Sons, 2013.

[131] Qing Zhao, Stefan E. Karisch, Franz Rendl, and Henry Wolkowicz. Semidefinite programming relaxations for the quadratic assignment problem. *Journal of Combinatorial Optimization*, 2(1):71–109, 2017.