

# A Parallel Computational Framework for Solving Quadratic Assignment Problems Exactly

Michael C. Yurko

June 22, 2010

## Abstract

The Quadratic Assignment Problem (QAP) is a combinatorial optimization problem used to model a number of different engineering applications. Originally it was the problem of optimally placing electronic components to minimize wire length. However, essentially the same problem occurs in backboard and circuit wiring and testing, facility layout, urban planning, ergonomics, scheduling, and generally in location problems. Additionally, it is one of the the most computationally difficult combinatorial problems known. For example, a recent solution of a problem of size thirty using a state-of-the-art solver took the equivalent of 7 single-CPU years. The goal of this project was to create an open and easily extensible parallel framework for solving the QAP exactly. This framework has shown good scalability to many cores. It experimentally has over 95% efficiency when run on a system with 24 cores. This framework is designed to be modular to allow for the addition of different initial heuristics and lower bounds. The framework was tested with many heuristics including a new gradient projection heuristic and a simulated annealing procedure. The framework was also tested with different lower bounds including the Gilmore-Lawler bound (GLB). The GLB was computed using a custom implementation of the Kuhn-Munkres algorithm to solve the associated linear assignment problem (LAP). The core backtracking solver uses the unique approach of only considering partial solutions rather than recursively solving sub-problems. This allows for more efficient parallelization as inter-process communication is kept to a minimum.

## 1 Introduction

The Quadratic Assignment Problem (QAP) is a combinatorial optimization problem that is one of the most computationally difficult known problems. It is “strongly NP-hard” meaning that it is NP-hard to even find an approximate solution within any constant factor of the optimal solution[12]. This theoretical difficulty also translates into practical difficulty. Using a state-of-the-art solver, a problem of size 30 took over a week to solve on a computational grid of over 2,500 machines[11].

However, the QAP is not just a very difficult problem. It has many applications in many different fields. The first QAP was intended to model the minimization of total wiring length of a circuit on a backboard [13]. Since then, the QAP has also been used to model problems of urban planning and architectural design [5, 8]. The QAP also has other applications in graph theory[12], scheduling[2], facility layout[7], ergonomics [3], circuit testing [6], and biological testing[4].

Informally, the QAP can be thought of as follows: Assign  $n$  facilities to  $n$  locations such that the total cost is minimized. The cost is given by the sum of the products of the flows and distances between every pair of facilities. The distances between locations and flows between facilities are given by two matrices,  $d$  and  $f$ . Sometimes, there is also an additional matrix  $c$ , which contains the linear cost of assigning a given facility to a given location.

Formally, the QAP can be defined as the following Koopmans-Beckman[7] formulation:

$$\min \sum_{i=0}^{n-1} \sum_{j=0}^{n-1} f_{ij} d_{p_i p_j} + \sum_{i=0}^{n-1} c_{ip_i} \quad (1)$$

such that  $p$  is a permutation vector of length  $n$  such that  $p_i = j$  indicates that facility  $j$  is assigned to location  $i$  and  $f, d$ , and  $c$  are all  $n \times n$  matrices.

The purpose of this project was to create an open and easily extensible parallel framework for solving QAPs exactly. This framework is designed to allow for the easy addition of new lower bounds and heuristics. This is the first open-source and freely available framework for solving QAPs.

## 2 Core Solver

### 2.1 Overview

The core solver in this framework could be called either a backtracking or a depth-first branch and bound algorithm. It essentially works by traversing an implicit tree of partial solutions. The leaves of the full tree are all the possible permutations of solutions. As it searches, it checks the cost of any complete solutions and compares them to the previous best node. Because all of the permutations would be checked, then it would then return the optimal solution. However, this approach takes  $\Omega(n!)$  time. For even a small problem of size 12, over 1,302,063,773 nodes have to be explored.

### 2.2 Pruning

In order to improve the running time of the solver, we need to prune the search tree. Pruning means that we do not explore portions of the tree that we know cannot contain the optimal solution. While the QAP will always be strongly NP-Hard, a good branch and bound algorithm with good pruning can still provide good real-world performance.

In order to significantly prune the search tree, this framework does a few things. Before a node is expanded, a lower bound on the best possible solution of its children is computed. If the lower bound of the node is higher than the best solution yet found, then the node does not need to be expanded, and we can ignore its whole sub-tree. One way to improve the pruning that is used in this framework is to use an initial heuristic. Before the main solver is run, the best solution yet found is set to one found by a heuristic. This can often be very close to the optimal solution. While there are no guarantees, the simulated annealing heuristic in this framework often delivers the optimal solution for smaller instances ( $n < 15$ ) and is within a few percent of the optimal solution for most medium-sized instances ( $n < 50$ ). Because the algorithm finds a better value earlier, more branches can be pruned. It is important to prune the branches as early as possible to try to stem the exponential growth of the tree.

## 2.3 Pseudo-Code

The following is the pseudo-code for the core solver in this framework:

```

best_solution_yet = heuristic(problem)
best_cost_yet = cost(best_solution_yet)
push the empty solution on the stack
while the stack is not empty:

    pop the node p off the stack
    if p is a complete solution:
        c = cost(p)
        if c < best_cost_yet:
            best_cost_yet = c
            best_solution_yet = p
    else:
        lb = lower_bound(p)
        if lb < best_cost_yet:

            push all the children of p on the stack

```

## 3 Lower Bounds

### 3.1 Sub-Problem Generator

Most of the published lower bounds for the QAP do not work with partial solutions. Instead, they only give a lower bound on a complete QAP. However, we can use the lower bounds on sub-problems that we can generate from partial solutions and the complete problem. The lower bounds of these sub-problems corresponds to the lower bounds of the best solution in a given sub-tree.

In order to use the published lower bounds, this framework includes a sub-problem generator which takes the complete problem and partial solution and returns a sub-problem (which is really just another QAP). It works based upon the following logic.

If we fix a given facility  $q$  to a given location  $r$ , then we can generate a new QAP of size  $n-1$  by adding those fixed costs in the following manner.  $\forall i, j \in \{0, 1, \dots, n-1\}$ , add

$$c_{qr}/(n-1) + f_{iq}d_{jr} + f_{qi}d_{rj} \quad (2)$$

to  $c_{ij}$  and then delete row  $q$  and column  $r$  from  $f$ ,  $d$ , and  $c$ .  $c_{qr}/(n-1)$  is added to each element to keep the fixed cost from  $c$  that will be lost when the dimensions of the matrix are reduced.  $f_{iq}d_{jr} + f_{qi}d_{rj}$  represents the costs pairwise cost of assigning facilities  $q$  and  $i$  to locations  $j$  and  $r$ . This can be more formally derived by considering the trace formulation and substituting  $x_{qr} = 1$ .

### 3.2 Gilmore-Lawler Bound

The Gilmore-Lawler bound (GLB) is one of the oldest lower bounds for the QAP (references to Gilmore and Lawler papers). The main advantage of using the GLB in a branch and bound setting is that it is relatively easy to compute. The actual Linear Assignment Problem (LAP) can be created in  $\mathcal{O}(n^3)$  time and the associated LAP can be solved in  $\mathcal{O}(n^3)$  time using the Kuhn-Munkres or Hungarian algorithm.

The GLB is derived as follows. Let

$$A_{ijkl} = \begin{cases} f_{ii}d_{jj} + c_{ij} & \text{for } i = k, j = l \\ f_{ik}d_{jl} & \text{for } i \neq k, j \neq l \end{cases} \quad (3)$$

This essentially creates a Lawler QAP [9]. Partition  $A$  into  $n^2$  matrices of size  $n \times n$ ,  $A^{(i,j)} = (A_{ijkl})$  for each pair  $(i, j)$ ,  $i, j = 0, 1, \dots, n-1$ . Partition the solution matrix  $Y$  in the same way. Note that each sub-matrix  $C^{(i,j)}$  corresponds to the costs of the assignment  $p_i = j$ . Then for all pairs  $(i, j)$ ,  $i, j = 0, 1, \dots, n-1$  solve the following set of  $n^2$  LAP and store them in the matrix  $l_{ij}$ :

$$l_{ij} = \min \sum_{k=0}^{n-1} \sum_{l=0}^{n-1} A_{ijkl} Y_{ijkl} \quad (4)$$

$$s.t. \sum_{k=0}^{n-1} Y_{ijkl} = 1, \quad l = 0, 1, \dots, n-1 \quad (5)$$

$$\sum_{l=0}^{n-1} Y_{ijkl} = 1, \quad k = 0, 1, \dots, n-1 \quad (6)$$

$$Y_{ijij} = 1 \quad (7)$$

$$Y_{ijkl} \in \{0, 1\} \quad (8)$$

We can then denote

$$\text{GLB} := \min \sum_{i=0}^{n-1} \sum_{j=0}^{n-1} l_{ij} x_{ij} \quad (9)$$

$$\text{s.t.} \quad \sum_{i=0}^{n-1} x_{ij} = 1, \quad j = 0, 1, \dots, n-1 \quad (10)$$

$$\sum_{j=0}^{n-1} x_{ij} = 1, \quad i = 0, 1, \dots, n-1 \quad (11)$$

$$x_{ij} \in \{0, 1\}, \quad i, j = 0, 1, \dots, n-1 \quad (12)$$

However, this formulation takes  $\mathcal{O}(n^5)$  time since it doesn't exploit our knowledge of the structure of  $A$ . We can instead transform this into a single LAP instead of  $n^2$  LAPs due to the following result.

Given any two vectors,  $a$  and  $b$ , such that  $a$  is sorted non-descending and  $b$  is sorted non-ascending, the following holds for any permutation  $p$  (citation):

$$\sum_{i=0}^{n-1} a_i b_i \leq \sum_{i=0}^{n-1} a_i b_{p_i} \leq \sum_{i=0}^{n-1} a_i b_{n-i} \quad (13)$$

Let us denote the minimal vector product of  $a$  and  $b$  as  $\langle a, b \rangle^-$ . Using () we can obtain  $l$  as

$$l_{ij} = f_{ii} d_{jj} + c_{ij} + \langle f'_{(i,\dots)}, d'_{(j,\dots)} \rangle^-, \quad i, j = 0, 1, \dots, n-1 \quad (14)$$

where  $f'_{(i,\dots)}$  and  $d'_{(j,\dots)}$  are the  $i$ th and  $j$ th row of  $f$  and  $d$  respectively with the  $i$ th and  $j$ th element removed. We can then use the Kuhn-Munkres algorithm to solve this LAP in  $\mathcal{O}(n^3)$  time.

## 4 Heuristics

This framework provides an interface that allows for the easy addition of new heuristics. In addition, there are currently a few heuristics that are already implemented.

### 4.1 Simple Greedy

This was the first heuristic that was implemented. It is comparatively simple, but is also extremely fast. It is based around the idea that the pairs of facilities with the highest flow should correspond to the locations with the lowest lowest distances and vice-versa. The following is the pseudo code for the algorithm:

```
#get the flow and distance tuples into lists
for i=0 to n-1:
    for j=0 to n-1:
```

```

        flow_list.append( (f[i,j], i, j) )
        distance_list.append( (d[i,j], i, j) )

#sort the lists in opposite order by the flows and distances
flow_list.sort()
distance_list.sort(reverse=True)
#keep track of which facilities and locations are used
unused_locations = {0,1, ... ,n-1}
unused_facilities = {0,1, ... , n-1}
#make the assignments
while |unused_locations| > 1:

    hflow = highest unused flow tuple
    ldist = lowest unused flow tuple
    #assign the facilities and locations
    solution[ldist[1]] = hflow[1]
    solution[ldist[2]] = hflow[2]
    #remove the facilities and locations from set of unused
    unused_facilities -= hflow[1], hflow[2]
    unused_locations -= ldist[1], ldist[2]

#assign last facility and location if n is odd
if n%2 == 1:

    solution[last location] = last facility

return solution

```

## 4.2 Gradient Projection

The gradient projection heuristic is akin to many derivative-based function minimization algorithms. However, the gradient projection heuristic has to contend with the discrete nature of the QAP. The gradient projection heuristic attempts to overcome this by using the following well known relation:  $X = N \cup Q \cup E$ , where  $X$  is the set of permutation matrices,  $N$  is the set of non-negative matrices, and  $E$  is the set of doubly-stochastic matrices.

This heuristic also uses an alternative formulation of the QAP, the trace formulation:

$$\min \text{trace}(fxd^T x^T + cx) \quad (15)$$

$$\text{s.t.} \quad x \in X \quad (16)$$

where

$$\text{trace}(m) := \sum_{i=0}^{n-1} m_{ii} \quad (17)$$

The gradient projection algorithm takes the gradient of the problem ( $-f^T x d + f x d^T$ ) and does a line search until a lower objective function value is found.

However, the new matrix is almost guaranteed to not be a permutation matrix, so it is projected back into the feasible region by projecting it into  $N$ ,  $Q$ , and  $E$ . The projection onto  $N$  simply sets all negative elements to 0. The projection onto  $E$  uses a linear equation solver, and the projection onto  $Q$  uses a Q-R decomposition. The biggest issue with this algorithm is the projection onto  $Q$  since it is not a convex set. It is therefore, quite possible that the new value is actually farther away from the minimum than before. However, it generally does result in a better cost.

Because of the greedy nature of this algorithm, it will get stuck in local minima. However, this problem can be mitigated by running gradient projections multiple times with random starting points. For a more detailed coverage of gradient projection algorithms see (insert citation).

### 4.3 Simulated Annealing

Simulated annealing is a meta-heuristic that has been successfully applied to many different combinatorial optimization problems. It is a process that is analogous to the physical process of annealing. In a simulated annealing algorithm, there is a global variable called the temperature which is cooled according to a schedule which varies according to the implementation (although it usually decays exponentially). The algorithm does a certain number of random swaps at a certain temperature. If the swap reduces the cost, then it is accepted. If the swap increases the cost, then it is accepted with a certain probability which is usually given by  $e^{-\Delta/t}$  where  $\Delta$  is the change in objective function and  $t$  is the temperature. This probability is analogous to the total free energy of a system that is undergoing an annealing. After a certain number of iterations at a certain temperature have been done, the temperature is lowered according to the cooling schedule. This is continued until the system is considered frozen. When a system is declared frozen depends upon the specific implementation, but it is often indicated by a certain number of iterations that don't yield a successful swap.

Simulated annealing has been applied to the QAP before with promising results. However, this implementation has tried to borrow the good ideas from previous implementations. For example, (insert burkard reference) provided a method of directly computing the delta of a swap in  $\mathcal{O}(n)$  time as opposed to recomputing the whole cost and subtracting which takes  $\mathcal{O}(n^2)$  time. The authors observed the the change in the objective function when swapping two facilities,  $i$  and  $j$  can be found with the following formula:

$$\begin{aligned} \Delta = & (f_{jj} - f_{ii})(d_{p_i p_i} - d_{p_j p_j}) + (f_{ji} - f_{ij})(d_{p_i p_j} - d_{p_j p_i}) \\ & + \sum_{k \neq i, j} [(f_{jk} - f_{ik})(d_{p_i p_k} - d_{p_j p_k}) + (f_{kj} - f_{ki})(d_{p_k p_i} - d_{p_k p_j})] \quad (18) \end{aligned}$$

Additionally, (insert reference to Laursen) provided a sensible way of determining the starting temperature. Setting  $t_0 = n f_m d_m$  where  $f_m$  and  $d_m$  are the

means of the  $f$  and  $d$  matrices respectively. This temperature allows for most of the swaps to be accepted in the beginning.

The pseudo-code for this implementation is as follows:

```
t = n*mean(f)*mean(d)
alpha = exp(-log(t)/steps)
solution = random_permutation(n)
frozen = False
while not frozen:
    changes = 0
    for i = 1 to iterations:
        i1, i2 = random_int(n)           #0 to n exclusive
        d = delta(i1,i2)
        if d < 0:
            solution.swap(i1,i2)
            changes += 1
        else if rand() < exp(-d/t):
            solution.swap(i1,i2)
            changes += 1
    frozen = not changes
    t *= alpha
return solution
```

One final advantage of simulated annealing is how easily it is parallelized. Empirical results have shown that it is actually better in many cases to run many shorter annealings rather than one long annealing (i.e. if one is going to do  $10^8$  iterations, then it is usually better to do  $10^3$  annealing with  $10^5$  each than one annealing with  $10^8$  iterations). So, to parallelize the annealing, one can just run many annealings in parallel.

## 5 Parallelization

Even with all of the aforementioned methods of cutting down on the growth of the search tree, the QAP is still a very hard problem to solve. One additionally way to reduce the total computational wall time is to parallelize the algorithm. By parallelizing the algorithm, we are able to fully utilize the multiple cores that are in modern computers. Ever since Intel hit a frequency wall with its Pentium 4, the focus of the processor makers has been increasingly on adding more cores. An algorithm that is not parallel will not make use of this extra computational capacity.

The main concern when parallelizing an algorithm is how to handle the load balancing. Static load balancing schemes work well when it is easy to predict the size of a given workload. Problems like matrix multiplication are prime

candidates for static load balancing since the amount of time that it takes to do multiplications and additions is very constant (ignoring cache optimizations). However, static load balancing does very poorly for problems where the size of a given workload can't be easily determined. A branch and bound algorithm for the QAP is an example of this. Even two nodes at the same level of the tree are unlikely to take the same amount of time to explore their sub-trees. If one node contains a bad assignment it could be immediately pruned while another node at the same level in the tree could contain the optimal solution and need to be more fully explored.

Because of this, most branch and bound algorithms use a dynamic load balancing strategy. There are many different dynamic load balancing strategies (especially for tree based algorithms like branch and bound), but all of them in some way attempt to maintain a relatively even distribution of the workload. Most work in a way similar to this: Whenever a computational node is done with its work-load it requests new work from the load balancer. The load balancer will then take some work from a node that is not done. This can be done randomly, or the load-balancer can try to be more sophisticated about its selection. It then passes this work to the node that had finished its load. This is then continues until all the nodes are done with their work-load.

The advantage of this type of strategy is that the work is distributed evenly and no node sits idle for very long. However, it also imposes a significant overhead on the computation. Each node must check after a certain interval whether it needs to share its work. Also, if the load balancing isn't implemented carefully, then the efficiency of the algorithm can plummet at the end. It is possible that "thrashing" will occur where the nodes are fighting over the last bit of work and actually spend more time sharing the work than actually doing the computation.

This framework takes a hybrid approach. It tries to avoid the expensive overhead of more dynamic strategies while also utilizing as many cores as possible for the longest amount of time. First, the tree is expanded in a breadth-first manner until a certain minimum number of nodes is in a queue. This minimum is a heuristically determined value that is based upon the number of threads that are being used. Currently it is set to the number of threads cubed, but this could be refined. Each of the computational nodes then iterate over the nodes in the queue. The nodes represent roots of sub-trees that are explored by each thread. Whenever a thread finishes it is sent a new unexplored node from the queue. This continues until the queue is empty and each thread is done with its workload. Throughout the process there is only one variable that each thread must keep synced: the best cost yet. Whenever a node finds a better solution it sends this to the shared memory value. Each thread then updates its local value from this shared value a specified interval. This keeps communication overhead to a minimum.

## 6 Other Work

### 6.1 Importing Problems

In order for the framework to be useful, there has to be an easy way of importing or entering problems. For stored problems this framework includes a QAPLIB (cite) parser. QAPLIB is a library of benchmark problems that are useful for comparisons between different solvers. QAPLIB also defines a simple text based format for storing QAPs. One can easily import problems directly from QAPLIB, or can write ones own problem in the QAPLIB format. In addition, this framework includes two problem generators that will create both symmetric and asymmetric random QAPs.

### 6.2 Linear Assignment Problem Solver

At the core of the GLB is an LAP which needs to be solved efficiently. The LAP solver can be called millions of times in the course of solving a QAP, so it needs to have as little overhead as possible. Of the two other existing python LAP solvers, one had a severe memory leak, and the other was written in pure python. The implementation in this framework does not have a memory leak, and is over a thousand times faster than the one written in pure python.

This framework implements a version of the Kuhn-Munkres algorithm [10]. While the original Kuhn-Munkres algorithm took  $\mathcal{O}(n^4)$  time, the version implemented in this framework takes  $\mathcal{O}(n^3)$ . The Kuhn-Munkres is most easily explained in a high-level way as a series of steps :

- Step 1: Find the smallest element in each row and subtract it from its row. Go to Step 2.
- Step 2: Find all zeros in the matrix. Star the zero if there is no other starred zero in its row or column. Go to Step 3.
- Step 3: Mark each column that has a starred zero. If n columns are marked, then the algorithm is done. Otherwise, go to Step 4.
- Step 4: Prime all unmarked zeros. If there isn't a zero in the row of the primed zero, go to step 5. Else, mark its row and unmark its column. Keep the smallest number and go to Step 6.
- Step 5: Create a path alternating between primed and starred zeros. Then unstar each zero in the path, star each primed zero, remove all primes, and uncover every line. Go to Step 3.
- Step 6: Add the smallest number found in Step 4 to all covered rows and subtract it from all uncovered rows. Go to Step 4.

At this point the solution is found. If  $c_{ij}$  (where  $c$  is the cost matrix) is starred, then the pair  $(i, j)$  is assigned. The implementation in this algorithm keeps a separate Hungarian object which stores all intermediate memory needed for the

computation. This is done transparently, and prevents the need to allocate and reallocate the same memory at every node.

## 7 Notes on the Implementation

The framework was originally written in Python. Python is a dynamic, weakly-typed, interpreted language that is very easy to use and makes rapid application design possible. However, its biggest drawback is speed. This framework uses Numpy for its numerical data structures and fast algorithms. Using Numpy is significantly faster since it is mostly written in C and is statically compiled. However, even using Numpy isn't enough to make Python competitive with statically compiled languages like C.

To get good speed, much of the original code has been rewritten in Cython. Cython is a super-set of Python that is compiled to C which is then compiled down to native machine code. Cython can run almost any existing Python file. However, to get a real speed increase, Cython needs additional type information. In this framework many inner loops were converted to Cython for speedups ranging from 500x to 1000x. When all variables are typed, it is competitive and sometimes better than normal C code.

The combination of Python and Cython allows for this framework to be both easily extensible and fast. In addition, the use of Python means that it is easy to interface this framework with other Python applications as well as those written in C, Java, or .Net/Mono languages.

## 8 Empirical Results

In order to analyze the actual performance characteristics of this framework, many different experiments were performed. Each individual component of the framework was analyzed both as a whole and independently. All experiments were conducted on a Sun x4450 with 24 2.6 GHz Xenon cores and 128 GB of ram or a Sun T5240 with two T2 Niagra processors and 32 GB of RAM on Solaris 10. All of the graphs of this data can be seen on the board.

### 8.1 Heuristics

All of the heuristics were run on all of the QAPLIB instances of size 30 or less. For simulated annealing, the parallel version was run with  $10 * n$  annealings per QAP. For the gradient projection heuristics, 10 projections with random starting points were performed. Unsurprisingly, the simulated annealing heuristic performed the best. For many of the instances, it gave optimal solutions. For those problems which it did not give an optimal solutions, its solution was within a few percent of the optimum.

## 8.2 Lower Bound

The GLB was run against every problem in QAPLIB. Two different analysis were run. First, the gap between the GLB and the optimal value for all solved instances from QAPLIB of size less than 30 were compared. Surprisingly, there was no evidence in the data the GLB degrades on larger instances as it theoretically should. Secondly, the gap between the GLB and the optimal solutions was compared for the Li and Pardalos problems which are not from QAPLIB[?]. These problems are interesting in that they are generated in such a way that the optimal solution is known when they are created. However, they are also asymmetric problems which have proven empirically difficult to solve using normal methods. This allowed for the GLB to be analyzed on problems of up to size 200, a size where we do not normally know the optimal value. Again, the data did not indicate that the GLB was deteriorating. In fact, the regression lines of both plots indicate a negative slope for the gap. However, the correlation is poor enough to prevent any significant conclusion from being drawn.

## 8.3 Parallelization

A number of tests of the scalability of this framework were also conducted. The results indicated that the framework does indeed scale well. In fact, it scaled super-linearly for most of the tests. This means that it is performing better than an idealized perfectly efficient parallelization. These excellent results are due to additional pruning of the tree. Even without pruning, the parallelization is very efficient. In a test on the problem nug10 without pruning, the efficiency stayed at around 96% for up to 24 cores.

## 8.4 LAP solver

This framework includes a custom implementation of the Kuhn-Munkres algorithm. However, there are two other python implementations of the Kuhn-Munkre algorithm that the author is aware of[?, ?]. [?] is a pure python implementation while [?] is a wrapper around a C++ implementation. [?] which is the closest in speed to the custom implementation has a severe memory leak that precluded its use in a branch and bound solver. The implementation in this solver is faster than both of the other implementaions.

## 8.5 Overall

This framework with the GLB was tested on problems of up to size 25. Chr25a was solved using 16 threads on the x4450 in about 50 minutes. It is hard to compare this performance to other implementations since no other code for solvers has been released.

## 9 Conclusion and Future Improvements

The main goal of this project was to create a framework for solving the QAP that is easily extendable. The core branch and bound solver was written to be modular so that additional lower bounds and initial heuristics can be added easily. All these lower bounds and initial heuristics conform to a standard interface. This standard interface makes it easy to code new lower bounds directly in Python or Cython. However, Cython also provides an easy way to wrap existing bounds written in C or FORTRAN.

The main focus of this project was to create a framework. However, this framework also provides a fully functional QAP solver. This QAP solver can interface easily to many other languages and is competitive with the published results of other solvers based upon the GLB.

In order to truly advance the state of QAP solving techniques, the framework does need a few additions. Most importantly, it needs to include stronger lower bounds. The author plans to implement both the Anstreicher-Brixius bound[1] and the Hahn-Hightower bound[?]. Both of these bounds are much tighter than the GLB. However, this tightness does come at greatly increased computational costs. Finally, this framework could offer a specification for branching strategies. However, changing the branching strategy would have much less of an effect on running times than changing the lower bounds.

## References

- [1] Kurt M Anstreicher, Nathan W Brixius, and Semidefinite Programming. A new bound for the quadratic assignment problem based on convex quadratic programming. *MATHEMATICAL PROGRAMMING*, 89:2001, 1999.
- [2] R. E Burkard, E. Cela, P. M Pardalos, and L. S Pitsoulis. The quadratic assignment problem. *Handbook of combinatorial optimization*, 3(1):241, 1998.
- [3] R. E. Burkard and J. Offermann. Entwurf von schreibmaschinentastaturen mittels quadratischer zuordnungsprobleme. *Mathematical Methods of Operations Research*, 21(4):121–132, 1977.
- [4] S. A de Carvalho Jr, S. Rahmann, and G. Bioinformatik. Microarray layout as a quadratic assignment problem. In *Proceedings of the German Conference on Bioinformatics*, volume 83, pages 11–20.
- [5] Alwalid N. Elshafei. Hospital layout as a quadratic assignment problem. *Operational Research Quarterly (1970-1977)*, 28(1):167–179, 1977.
- [6] B. Eschermann and H. J Wunderlich. Optimized synthesis of self-testable finite state machines. In *Fault-Tolerant Computing, 1990. FTCS-20. Digest of Papers., 20th International Symposium*, pages 390–397, 1990.

- [7] Tjalling C. Koopmans and Martin Beckmann. Assignment problems and the location of economic activities. *Econometrica*, 25(1):53–76, 1957.
- [8] J. Krarup and P. M Pruzan. Computer aided layout design. *Mathematical programming in use*, 1978.
- [9] Eugene L. Lawler. The quadratic assignment problem. *Management Science*, 9(4):586–599, July 1963.
- [10] Bob Pilgrim. The Kuhn-Munkres algorithm. <http://www.public.iastate.edu/~ddoty/HungarianAlgorithm.html>.
- [11] Jason Riedy. Quadratic assignment problem. <http://www.cs.berkeley.edu/~ejr/GSI/cs267-s04/homework-0/results/sonesh/>.
- [12] Sartaj Sahni and Teofilo Gonzalez. P-Complete approximation problems. *J. ACM*, 23(3):555–565, 1976.
- [13] Leon Steinberg. The backboard wiring problem: A placement algorithm. *SIAM Review*, 3(1):37–50, 1961.