Solving Certain Large Instances of the Quadratic Assignment Problem: Steinberg's Examples

Mika Nyström
Department of Computer Science
California Institute of Technology
Pasadena, California, U.S.A.

October 27, 1999

Caltech Computer Science Technical Report
caltechCSTR:2001.010

Abstract

This report discusses techniques for the solution of the quadratic assignment problem (QAP) and describes a prototype implementation used to solve two large ($n = 36$) instances of the problem.

1 Introduction

In 1961, Steinberg presented a specific problem encountered in the design of computer systems that he called "the backboard wiring problem"[5]. This was one of the first[1] references to practical applications of the quadratic assignment problem (QAP), which is to choose the permutation $\mathbf{p}$ to minimize the cost

$$C = \sum_{1}^{N} \sum_{1}^{N} a_{ij} b_{p_i p_j}, \tag{1}$$

where p_k represents a permutation index, viz., $\mathbf{p}$ is the vector of some permutation of the integers $1 \ldots N$. (In Steinberg's example, for instance, the a_{ij}s may denote the distances between slots where function blocks are to be placed, and the b_{kl}s would then denote the number of wires needed to connect two arbitrary blocks to each other.) In many concrete examples $\mathbf{A}$ and $\mathbf{B}$ are symmetric, and also C is often replaced by $C/2$. These modifications generally do not affect the

[1]According to Lawler[8], the first reference to the quadratic assignment problem in the literature was in reference to the loci of economic activities by T. C. Koopmans and M. J. Beckmann, *Econometrica*, **25**, 1957, pp. 52–76.

methods applied to the problem. N, the dimension of $\mathbf{A}$ and $\mathbf{B}$, is called the *size* of the problem. QAP is NP-hard.[2]

Historically, two different approaches have been taken to QAP. The first is the development of various heuristics to generate "good" solutions (sometimes together with a "goodness bound"), and Steinberg's application of the Hungarian algorithm for the assignment problem was the first of these. The second approach is the development of algorithms to prove the optimality of a given solution. The Gilmore-Lawler bound is an example of a bounding algorithm that can be used recursively to improve a lower bound on the solution (a standard branch-and-bound approach) until the bound coincides with a known solution, which is then proven to be optimal[8, 7]. Interestingly, the Gilmore-Lawler bound is closely related to Steinberg's heuristic, making it possible also to use it to find solutions as well as to bound them. We use the Gilmore-Lawler bound in our work.

Many modern parallel algorithms to solve various large problems have taken a "synchronous" approach in which the problem to be solved is partitioned into small *work units* that are then parceled out to a pool of computing devices. In certain situations, this may lead to large inefficiencies if it is not known how long it will take to complete a given work unit (as is the case if the completion time of the work units is variable and unknown). The inefficiences result from communications overhead (if the assigned work units are too small) or poor load balancing and consequent poor utilization of the available computing resources (if the assigned work units are too large). We develop a nondeterministic branch-and-bound algorithm to address these issues.

2 Computational Approaches

To solve Equation 1, there are in general no known methods that are guaranteed to be reasonably fast. While a great number of fast heuristic approaches have been developed that often produce provably good estimates of the best possible solution, branch-and-bound algorithms appear to be the most promising approach to generate provably optimal solutions in a reasonable number of compute cycles.

2.1 Steinberg's heuristic algorithm

Steinberg's heuristic algorithm represents the first attempt to find good solutions to large instances of QAP. The algorithm centers on the concept of a *zero flow submatrix.* Concretely, if it is possible to rearrange the rows and columns of $\mathbf{A}$ and $\mathbf{B}$ such that a reasonably large submatrix from a_{M+1M+1} to a_{NN} is zero, it is quite easy to solve for the minimum of C given already determined values for $p_1 \ldots p_M$. This reduces the size of the exponential search problem from N to M. In practice, we may split C into four components

[2]Lawler points out that the traveling salesman problem is a special case of QAP [8], p. 587.

$$C = \sum_1^M \sum_1^M a_{ij} b_{p_i p_j} + \sum_{i=1}^M \sum_{j=M+1}^N a_{ij} b_{p_i p_j} + \sum_{i=M+1}^N \sum_{j=1}^M a_{ij} b_{p_i p_j} + \sum_{M+1}^N \sum_{M+1}^N a_{ij} b_{p_i p_j}. \tag{2}$$

In order to simplify the algebra, we assume that **A** and **B** are symmetric and notice that the final term is zero since it involves only a linear combination of the elements a_{ij} of the zero submatrix. Thus,

$$C = \overbrace{\sum_1^M \sum_1^M a_{ij} b_{p_i p_j}}^{C_f} + 2 \sum_{i=1}^M \sum_{j=M+1}^N a_{ij} b_{p_i p_j} + \sum_{M+1}^N \sum_{M+1}^N a_{ij} b_{p_i p_j} \tag{3}$$

$$C = C_f + 2 \sum_{i=1}^M \sum_{j=M+1}^N a_{ij} b_{p_i p_j}, \tag{4}$$

where C_f is fixed. All that remains is to solve for the optimal $p_{M+1} \ldots p_N$ in

$$\sum_{i=1}^M \sum_{j=M+1}^N a_{ij} b_{p_i p_j}, \tag{5}$$

but the reader will recognize this as a linear (or "Hungarian") assignment problem (LAP), which can be solved in polynomial time by Kuhn's method.

Steinberg's procedure is based on the four-way decomposition of C. His approach was to find different maximal unconnected sets of the blocks of the Univac computer he was modeling and to use these unconnected sets to vary the placement order.[3] By starting with a random placement of blocks and modifying only the $p_{M+1} \ldots p_N$, this algorithm is guaranteed to generate a sequence of placements with monotonically non-increasing costs. However, it is obvious that this algorithm is never guaranteed to find the optimal **p**, nor is there any reference to a lower bound that might be used to judge the quality of the acheived feasible solutions.

2.2 The Gilmore-Lawler bound

The Gilmore-Lawler bound (GLB) represents a formalization and generalization of Steinberg's approach. The generalization extends Steinberg's approach to situations where the unplaced submatrix of **A** need not be exactly zero. (Although the quality of the bound improves as this submatrix becomes small.) Assuming the partial placement $p_1 \ldots p_M$, we again consider Equation 4 and rewrite it as

$$C(\mathbf{p}) = C_f + \sum_{j=M+1}^N \underbrace{\left(2 \sum_{i=1}^M a_{ij} b_{p_i p_j} + \sum_{i=M+1}^N a_{ij} b_{p_i p_j} \right)}_{\gamma_j}, \tag{6}$$

[3]The reader will recognize that finding the *maximum* unconnected set is isomorphic to the clique problem, which again is NP-hard.

where we can consider the term γ_j the cost of placing the single block j in the position p_j. The bounding method proceeds by bounding γ_j as follows. Consider a different "cost" $\tilde{c}_{jk}$ that we compute for all values of k in $M+1\dots N$

$$\tilde{c}_{jk} = 2\sum_{i=1}^{M} a_{ij} b_{p_i k} + \sum_{i=M+1}^{N} a_{ij} b_{q(i,j,k)k}, \tag{7}$$

where the function $q(i,j,k)$ is a permutation chosen to sort a_{ij} and $b_{q(i,j,k)k}$ in opposite directions as the sum is taken over $i = M+1\dots N$, minimizing the second sum. The total cost may be written as

$$\tilde{C}(\mathbf{p}) = C_f + \sum_{M+1}^{N} \tilde{c}_{ip_i} = C_f + \sum_{j=M+1}^{N} \left(2\sum_{i=1}^{M} a_{ij} b_{p_i p_j} + \sum_{i=M+1}^{N} a_{ij} b_{q(i,j,p_j)p_j} \right). \tag{8}$$

Finally, we solve the LAP for $\tilde{\mathbf{p}} = (\tilde{p}_{M+1}, \dots, \tilde{p}_N)$ to minimize

$$\tilde{C}(\mathbf{p}) = C_f + \sum_{M+1}^{N} \tilde{c}_{ip_i} \tag{9}$$

where clearly C_f drops from consideration since $\tilde{C}$ is minimized by the same permutation that minimizes $\sum \tilde{c}_{i\tilde{p}_i}$ on its own. By the same token, $p_1 \dots p_M$ also need no longer be considered. It is straightforward to prove that $\tilde{C}(\tilde{\mathbf{p}})$ bounds the cost of the QAP given the fixed permutation $p_1 \dots p_M$.

It is a simple matter of algebra to extend GLB to the cases of asymmetric $\mathbf{A}$ and $\mathbf{B}$. Also, it is clear that Steinberg's observation still holds true. Thus, if the submatrix of $\mathbf{A}$ pertaining to elements $M+1$ through N is zero, $\tilde{C}(\tilde{\mathbf{p}})$ will be the cost of the optimal solution to the QAP with the given constraint, and $\tilde{\mathbf{p}}$ will be one (not necessarily unique) permutation achieving this minimal cost.

3 Basic Algorithm

The algorithm used in our work is based on the GLB. Some additional information is required to turn GLB into a workable branch-and-bound algorithm. Most of this information is heuristic in nature, without a solid theoretical basis (but it is usually possible to see that these heuristics avoid common worst cases).

3.1 Placement order

One difference between Steinberg's approach and a branch-and-bound algorithm based on GLB concerns the placement order. In Steinberg's algorithm, the placement order is varied for each iteration of the improvement routine. When GLB is used as a lower bound in a branch-and-bound search, the placement order (i.e., the order of the rows and columns in $\mathbf{A}$ and $\mathbf{B}$) is fixed. Thus, the placement order becomes an important part of the algorithm, and we experimented with several choices for the placement order.

3.1.1 Clique approach to placement order

To take advantage of Steinberg's zero submatrix property, one might consider placing the blocks in such an order as to allow the problem to degenerate to an LAP problem in as few placements as possible. As pointed out in the footnote above, this entails solving a problem isomorphic to the clique problem. This clique problem is often considerably smaller than the original QAP, making it possible to solve it with a reasonable computational effort.

3.1.2 Greedy placement orders

A different placement order to consider is to pick the elements in such a way that each element is the one with the most connections to the already picked subset. A third order that we considered was the *ad hoc* approach of generating a greedy assignment according to the same LAP method used in Section 2.2 and placing the blocks in decreasing order of their cost contributions to this crude bound.

We implemented all the placement orders we have discussed. For reasons unknown to us, the cost contribution order from the zeroth level LAP approximation seemed to work well for Steinberg's example. Limited experimentation on other QAP instances from QAPLIB[9] shows that this is not universally true. Maximizing the connectivity of each element to the set of its ancestors worked better for some of these problems. This is clearly an area that merits further study.

3.2 Search order

An issue orthogonal to the placement order of Section 3.1 is, given the placement order, how to choose the order in which each possible permutation is tried. In many cases this order is unimportant, e.g., when one has a strong suspicion that the feasible solution one is working with is actually the optimal (as is the case when a good solution has been generated through some means external to the branch-and-bound search and it is desired to prove that it is the optimal solution). However, if the branch-and-bound algorithm is started without a known "good" solution, the search order is likely to affect the runtime greatly.[4] In our work, we use the bound as a heuristic indicator of the likelihood of finding a good solution down a given branch of the search tree. We search branches from a given node in the order of increasing lower bound (according to the LAP). This approach caused some difficulties associated with the distributed implementation and numerical accuracy, as described in Section 4.6.

3.3 Utilization of symmetries in Euclidean search grid

Since the search spaces we are mostly concerned with are grids in Euclidean space, we may make use of symmetries, as noticed by Mautor and Roucairol [6].

[4]It should be pointed out that some authors have concluded that using branch-and-bound searches to generate optimal solutions is inferior to using heuristic methods [6].

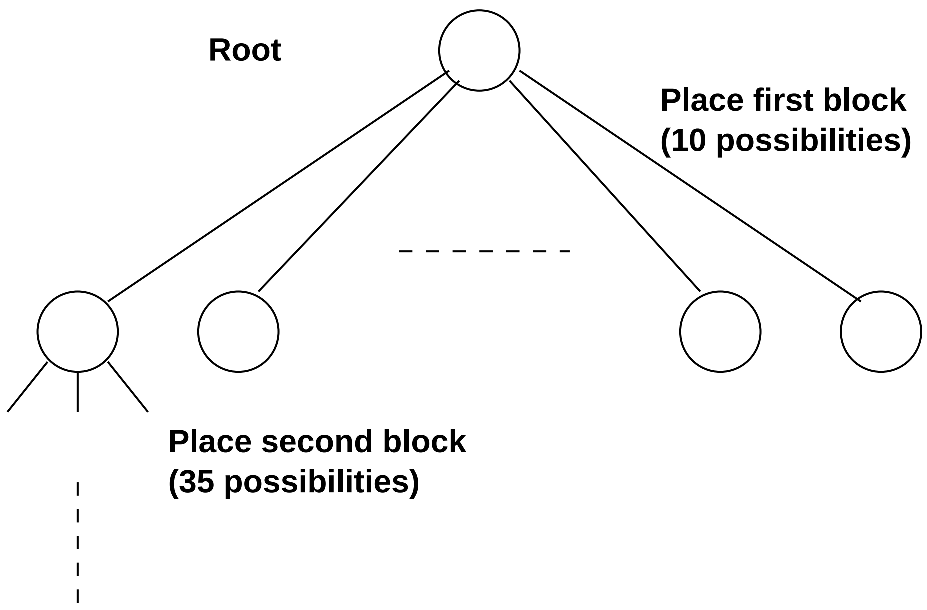

Figure 1: Schematic representation of search tree.

For instance, if we were to place the first block in the lower right corner, the search space induced by this partial placement is identical (with respect to a symmetry group under which Euclidean distances are invariant) to that induced by the partial placement of that block in any of the other corners. Our method was developed independently from [6], so we simply considered the symmetry to be broken as soon as the first block was placed. In the case of Steinberg's example, the grid is of size 4×9. With our method, we place the first block in all of the locations in the bottom left quadrant, an area of $2 \times 5 = 10$ possibilities. This makes for a 3.6-fold improvement in overall runtime, which is near the theoretical maximum[6]. After the first block has been placed, there are 35 possible locations for the second, 34 for the third, and so forth, so the total number of placements in the search tree is $10 \times 35! \approx 10^{41} \approx 2^{136}$. Part of the search tree is shown in Figure 1.

3.4 Sequential algorithm

The algorithm can be expressed in C-like syntax as:

```
void place_block( int cur_block, int depth )
{
  for( i = 0 ; i < positions ; ++i ) {

    /* skip already occupied positions */
    if ( position_holds_fixed_block( i ) ) continue;

    /* fix cur_block */
    fix_block_in_position( cur_block , i );

    /* use the Gilmore-Lawler algorithm for lower bound */

    lower_bound = greedy_placement();

     /* if the lower bound is less than
        the best feasible solution, recurse.

        Base case: done to unconnected subset;
        that means we have a new solution.
      */

    if ( lower_bound < best_feasible ) {
      if ( depth >= N - unconnected )
        best_feasible = lower_bound;
      else
         place_block( cur_block + 1 , depth + 1 );
    }
    unfix_block( cur_block );
  }
}
```

Here we have omitted all the details about placement order, symmetries, and search order. We have also assumed that **A** and **B** are set up properly and that the variables (e.g., `N` and `unconnected`) are initialized to reflect the arrangement of the matrices. This algorithm can be seen as visiting the nodes in the search tree in depth-first, left-to-right order.

4 Distributed Algorithm

Our distributed version of the GLB branch-and-bound search is implemented in two kinds of processes: a single *master* process and any number of *slave* processes. The master process is responsible for keeping track of the results of the slaves and for giving them work when they are idle.

In the sequential algorithm described in Section 3.4, the "current location" of the computation in the search tree is maintained implicitly as the loop counters `i` at every level of the recursion. (Literally, the location in the tree is represented as numbers on the call stack of the program.) In a distributed version of the algorithm, it becomes necessary to make explicit the representation of the search tree.

4.1 Dynamic representation of search tree

The search tree is maintained as a dynamic data structure in the memory of the master. The tree must be a dynamic data structure since its actual size may exceed the memory available for the program. Nodes of the tree are thus maintained in three states: *unchecked, inprogress,* and *done.* Transitions are normally (but not always) monotonically in the direction *unchecked* → *inprogress* → *done.* When the system is initialized, there exists a single *unchecked* node corresponding to the entire computation.[5]

Each node in the tree corresponds to the placement of a particular block (represented by the depth of recursion) in a particular location (represented by the left-to-right location of the node *vis-a-vis* its parent). When an *unchecked* node is expanded, it is given children corresponding to the $\texttt{N} - \texttt{depth}$ possible placements of the next block, given all the fixed placements represented by the current node and its parents. Nodes are recursively expanded until either a new, better feasible solution is reached or until the lower bound for that branch of the tree exceeds the currently known best feasible solution. (Of course, this corresponds to the recursive calls to `place_block` in the sequential program.) When the recursion peters out for a particular node (i.e., when `place_block` is *not* recursively called for one of the two reasons), that node is marked *done.* When all the children of a given node are *done,* that node is unexpanded and also marked *done.* The computation terminates when the root is marked *done.* In the sequential case, the *inprogress* state is unnecessary.

4.2 First concurrent implementation

While the previous paragraph may seem only to be a tedious restatement of parts of the sequential pseudocode of Section 3.4 in terms of trees, the crucial difference is that it becomes easy to work on the various parts of the tree concurrently.

This requires that the slaves know which tree nodes they are to process. Naïvely, all that is necessary is to modify the pseudocode of Section 3.4 to start each slave at the right point in the recursion and to make the assignment to `best_feasible` happen in all the slaves at the same time. In the master, the *inprogress* state is used as a marker to remember which nodes have been

[5]In the actual program used to solve Steinberg's examples, we used ten root nodes corresponding to the ten non-degenerate placements mentioned in Section 3.3.

assigned to slaves. No node that is a descendant of an *inprogress* node may be assigned to a slave. In our implementation of this scheme, this is handled by representing each node by its "serial number" from left to right on each level of the tree, in a list. Unexpanded levels are denoted with zeros or not printed at all. Thus the node (1) represents placing the first block in the first location. Issuing the command "**search**(1)" to a slave would instruct it to search all possible arrangements that have the first block in the first location. Similarly, $(7, 2, 9)$ would represent the node of the tree corresponding to the placement of the first block in slot 7, the second block in slot 2, and the third block in slot 9. The slot numbers go through one level of indirection due to the chosen search order, see Sections 3.2 and 4.6.

Even the simple first concurrent implementation is, generally speaking, nondeterministic.[6] This is because it is normally impossible (and always undesirable) to synchronize the slaves to the extent required to predict when the `best_feasible` variable in each slave is going to be updated.

4.2.1 Sketch of correctness proof

Progress is trivially satisified by the distributed algorithm as long as the master assigns nodes such that every leaf node is in the set of descendants of the assigned nodes. Safety is satisfied if the node corresponding to a minimum solution (note that the minimum solution is not necessarily unique, and we do not require finding *every* minimum solution) is searched. This is the case as long as every node is searched that has a lower bound of less than the minimum solution, and this is clearly the case since every node that has a lower bound less than `best_feasible` is searched and `best_feasible` is only set to the values of actual solutions. This depends on updates to `best_feasible` being atomic, which is true in practice. (I.e., if two slaves find new solutions simultaneously and try to update `best_feasible` simultaneously, the final value of `best_feasible` will correspond to one of the solutions.)

4.2.2 Improving the performance

To improve the performance[7] of the algorithm, we ensure that if a new solution has been found, *every* slave will eventually update its value of `best_feasible` to the new value (or less). By making the slaves update `best_feasible` only if it is less than the currently stored value, we handle the race condition that may occur if two slaves find new solutions simultaneously and try to update all the `best_feasible` variables simultaneously, which could lead to some being more than the others. The comparison is required in the master, to ensure that the actual minimum solution is stored.

[6]By nondeterministic we mean the "significant" nondeterminism that we cannot predict which nodes in the tree are going to be expanded.

[7]By performance we mean, generally speaking, the inverse of the number of nodes visited.

4.3 Second distributed implementation

The distributed implementation of Section 4.2 is correct, but it shows serious performance problems. Due to the nature of the branch-and-bound archetype, the amount of computational effort required to search fully a given node is unknown in advance. This property can result in large imbalances in the amount of work assigned to the slaves. (E.g., in Steinberg's examples, node (1) could take several months of processor time to search exhaustively, whereas node (10) takes perhaps only a few minutes.) The way this imbalance appears in practice is that some slaves are done while others still have work to do.

We approach the problem of imbalance by reassigning tasks. This requires more fine-grained knowledge about a slave's progress than only knowing if it is done with its task. In this scheme, each slave periodically notifies the master about which node it is working on. Since the sequence of nodes that is searched is deterministic *within each slave,* this procedure allows the master to mark a whole range of nodes as *done* in its representation of the search tree. The update defines a perimeter to which the search tree is expanded, defined as the minimum set of nodes required to cover the nodes reported as *done* by the slave. The tree is expanded to this perimeter, and all nodes that are *done* are marked accordingly. Remaining nodes are marked *unchecked,* but the node that was assigned as a task to the slave is left *inprogress.*

When a slave finishes its task, the master thus has relatively recent information about what nodes remain to be searched, even if the currently assigned tasks cover all the leaves of the tree (in the sense that all the leaves are descendants of nodes that are either *inprogress* or *done*). The task assignment algorithm is: First assign any unexpanded (perimeter) nodes in the tree that are *unchecked* and not descendants of nodes that are *inprogress;* if there are none left, pick the leftmost node that is *inprogress* and expand it if necessary to show its children. Re-assign the first *unchecked* child to the old slave, and assign the second *unchecked* child to the new slave. Both the children are now marked *inprogress,* and the parent node is marked *unchecked.* An approximate invariant is that each node that is marked *inprogress* is in one-to-one correspondence with a slave's task to search that node.

It must be realized that the invariant governing the *inprogress* nodes can be violated by race conditions—i.e., the master could reassign a slave to a node *after* it has completed work on it since the master's view of the progress of the slave is slightly out-of-date. Note, however, that for safety, all that matters is that no nodes are marked *done* that have not actually been searched. A simple way of dealing with the confusion that results when a slave is reassigned to an already completed task is for it simply to ignore the request and for the master subsequently to handle the situation as if the slave had died. As we shall see, we must satisfy the progress condition even in an unstable environment in which slaves can die at any time, so we postpone this issue to the next Section.

4.4 Fault tolerance

The need for fault tolerance in the algorithm used for this project was clear from the start.[8] There are at least three sources of faults in the system:

1. Stopping faults in the master, the slaves, or both.
2. Packet loss in the network. (Stopping faults in the network.)
3. Designed-in "faults" in response to otherwise difficult-to-handle situations.

With the very long runtimes (total CPU time) anticipated for the program, fault-tolerance is an absolute necessity.[9] The reason for handling stopping faults in the master or slaves needs no discussion. We handle stopping faults (packet loss) in the network explicitly so that we may use the User Datagram Protocol (UDP), which is an unreliable Internet Protocol (IP) service. Because of the way in which we handle faults explicitly within our algorithm, it is likely that this is a more efficient implementation overall than, e.g., using the reliable Transmission Control Protocol (TCP) and transferring the reliability problem to the operating system(s) of the involved computers. Given the mechanisms to handle stopping faults in the slaves, master, or network, it is easy to take advantage of them for other uses, as outlined in the previous Section.

Faults in the slaves are handled by a simple retransmission protocol. When a message is sent to a slave, it is expected to be acknowledged within one second. If this does not happen, it is retransmitted about ten times, first at one-second intervals, then with an exponential backoff. If the message has not been acknowledged at the end of this process, that slave is marked as "down" and its task (in the *inprogress* state) is marked *unchecked* once again.

Faults in the master are handled by checkpointing. If the master crashes, due to hardware or software failure, a special flag may be given when it is restarted, forcing it to read a text representation of the state of the computation from a file on disk. Since this file contains only the tree, a feasible solution (if any) and the current bound of the search, the master may be restarted on a different machine (even of a different architecture) in case of permanent hardware failure. It may also be restarted with a different set of slave systems. The case of the master crashing while it is writing the checkpoint file itself is handled by keeping several (nominally eight) of the latest versions of the checkpoint file.

In the case of network or slave faults, recovery may be automatic (if the slave or network returns to a normal state without operator intervention). This is done by the master repeatedly "pinging" all the slaves at regular intervals. When a slave reappears, the master will notice that the slave is available for computation and will duly assign a task to it. Recovery after a master fault is manual.

Again, we emphasize that we believe that the only part of the code that requires to be verified carefully is that concerned with updating the state of

[8] This point was driven home especially hard when a sequential, non-fault-tolerant early version of the implementation suffered a computer crash after five weeks' computing.

[9] The relatively short runtimes seen in practice were something of a surprise to us.

the search tree. While we have handled many race conditions in the code in various ways (as described in this paper), it is quite possible that others remain, and we claim to give no proof that this is not so. However, we do claim that safety cannot be violated (except as outlined in Section 4.6), which means that the program will either output the correct answer or deadlock(livelock).[10] Due to the use of timeouts and retries, it is easy to establish progress if sufficiently optimistic fault model is assumed, but we have not attempted to determine the exact properties required to achieve progress (really, absence of livelock).

4.5 Cap on search depth

The final addition we made to the algorithm was to limit the search depth to a fixed value. The reason for doing this is that when the search approaches the end, there are very few nodes left to expand, and these are quite deep in the tree. Since the LAP algorithm is $O(N^3)$ in time, where in this case N in the number of unconnected blocks, which decreases with depth, the finale of the computation quickly degenerates with a great deal of communications overhead when a large number of slaves compete for the very small remaining amount of work. This problem is easily avoided by ensuring that the tasks that are assigned be *moderately* large or larger. Limiting the search depth is an *ad hoc* way of accomplishing this.

4.6 Use of floating-point arithmetic

Since we originally studied the Euclidean norm version of Steinberg's example, our implementation uses floating-point arithmetic for all cost computations. For the sequential version of the code, this is not a problem, but for the distributed version, this introduces a subtle bug that reduced the utility of the implementation. Namely, the sorting procedure mentioned in Section 3.2 can sort the children of a search node in a different order on different hosts if the computation of the cost (which involves square roots) can lead to slightly different answers depending on the architecture of the host. Since we decided to save memory by always recomputing the order in which children should be searched rather than storing the order of the children of each node on a central server, sorting the children of a search node in different orders leads to certain branches of the search tree being searched multiple times, and others not at all. (In other words, it leads to nodes being named inconsistently on different computers, which can be disastrous since we use these names to identify tasks and in updates from the slaves to the master.) We sidestep this issue by running the code on a single architecture/operating system combination for the cases when this might matter. For most published instances of QAP, the costs are always integral. Also, as stated in Section 3.2, this matter is unimportant if the algorithm is only to be used to prove the optimality of an already known solution achieved through good heuristics.

[10]Please report to the author if it does the latter.

4.7 Comparison to large-scale distributed computational projects

Even though we may have a computationally difficult problem such as one of Steinberg's examples, there are only ten "root" tasks to assign the slaves. This contrasts greatly with most of the multitude of algorithms that have been run on large wide-area networks in recent years, such as the `rc5des` project to test the strength of the DES encryption system or the Seti@home project to search for signals from extraterrestrials in a large amount of radio telescope data[11, 12]. These systems all share the fact that they perform searches through or processing of data in a highly deterministic manner, thus requiring very little feedback from the slaves to the master. In our case, the high and unknown variability of computational effort required to expand the various nodes of the search tree leads to the use of much more frequent feedback messages to the master.

4.8 Low-level implementation issues

The branch-and-bound algorithm was implemented in ANSI C and in FORTRAN 77. All parts of the program (including parts used to manipulate actual VLSI layout from the Magic layout editor) consist of a total of approximately 12,500 lines of code. We used various kinds of systems, ranging from 120 MHz Intel Pentium PCs running NetBSD to a 500 MHz Digital Alpha system running OSF/1.

The heterogeneous nature of the computing systems we used led us to take special care in making network packet formats and data file formats interchangeable. We made them all ASCII text-based to avoid "endianness" and word-size problems.

For the solution of the LAP we used Munkres's version of Kuhn's algorithm, coded in C with FORTRAN array conventions[4, 3]. While there have been advances in LAP solvers since Munkres's method, the small average size of the LAPs solved seemed to make it a good choice for our implementation. Significant speedup (i.e., around threefold, from 600 days expected runtime to 200 days) was achieved through the use of low-level optimizations such as management of cache-coherence issues, lazy memory allocation (especially in Munkres's method, which was in the inner loop of the branch-and-bound algorithm), and recoding of the LAP solver using preprocessor macros instead of functions. FORTRAN 77 was further used for all linear algebra, and the compiled FORTRAN code was linked to the C program through the usual Unix conventions. We used the GNU C compiler (`gcc`) and the GNU FORTRAN 77 compiler (`g77`) on the Berkeley Unix systems. We used the DEC C and FORTRAN compilers as well as the Bell Laboratories `f2c` FORTRAN-to-C converter on the Digital Alpha systems.

Our work began with a sequential implementation of the algorithms described and progressed to a distributed implementation. In order to minimize the changes to the code as well as to minimize the runtime overhead of checking for messages to arrive, the message-passing was implemented as a separate thread of control using Unix signal handlers to switch context. Careful management of signals was necessary to ensure mutual exclusion on updates (as both the computational "core" of the program and the message-passing thread could update the same memory locations). We used explicit user-level semaphores (i.e., C variables declared `static volatile int`) for this purpose.

5 Results

These results were generated on a homogeneous cluster of eleven dual-processor Pentium Pro workstations running Berkeley Unix (i.e., 22 CPUs in all). This has shown that the solutions to Steinberg's example with the Euclidean and Euclidean-squared norms already published in QAPLIB are indeed optimal.

5.1 Optimality of solutions to Steinberg's example

For the version of Steinberg's example with the Euclidean norm, the run was started with a bound of infinity (no prior information). The program terminated after approximately 200 days of CPU time with the solution published in QAPLIB as optimal. (This same solution was generated after approximately 500 hours of CPU time on a 500 MHz Digital Alpha workstation in 1997 with the sequential version of the same program.) The metric of the solution was 4119.74 (8,239,110 in QAPLIB terminology because of scaling and rounding problems due to their use of only integer matrix coefficients). For the Euclidean-squared norm, the program was started with a bound of 7930 (15860 in QAPLIB terminology). After about 60 days of CPU time, the program terminated with a solution with metric 7926 (15854), the same as that published in QAPLIB.

5.2 Quality of the lower bound

The quality of the bound described in this paper may be investigated by asking how long it takes the implementation to generate a new lower bound. This may be done by setting the starting lower bound to a (low) value, and then running the program. If the program terminates without finding a solution, it is clear that the starting value was in fact a lower bound on the best possible solution. By this procedure, we quickly established a set of lower bounds for the version of Steinberg's example with the Euclidean cost function. We used the sequential version of the program for this and were able to improve on all previously published bounds in a few seconds of runtime on a 200 MHz Pentium Pro Unix workstation. Studying the runtime required to improve the lower bound reveals the exponential nature of the algorithm. We graph both the runtimes required to bound the Euclidean version and the Euclidean-squared version in Figures 2 and 3. Please remember that the data point for the optimal solution of the Euclidean-norm problem is shifted to the right since the program was started without preconceptions about the optimal solution. We have not yet re-run it

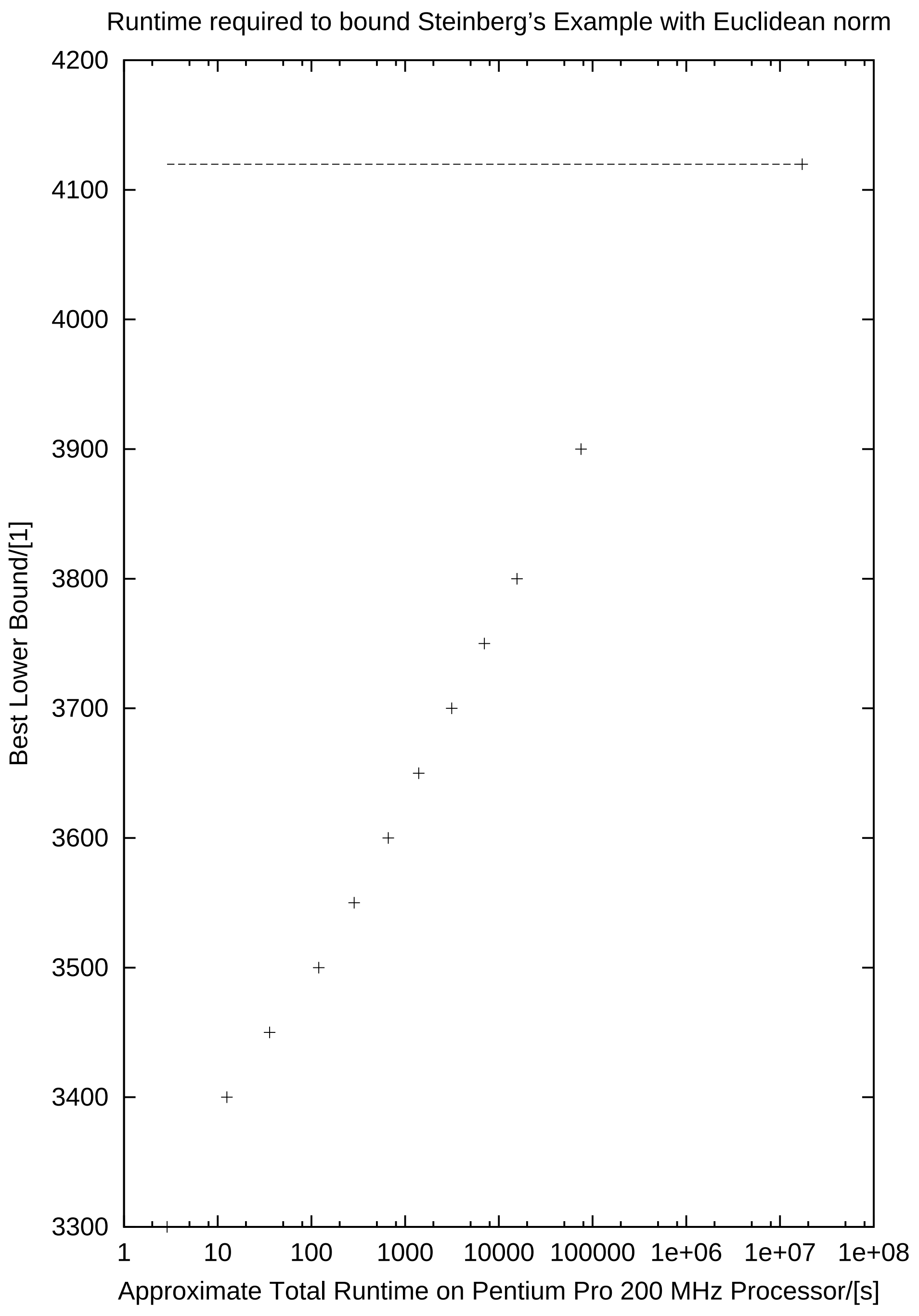

Figure 2: Runtime for Steinberg's example, Euclidean norm.

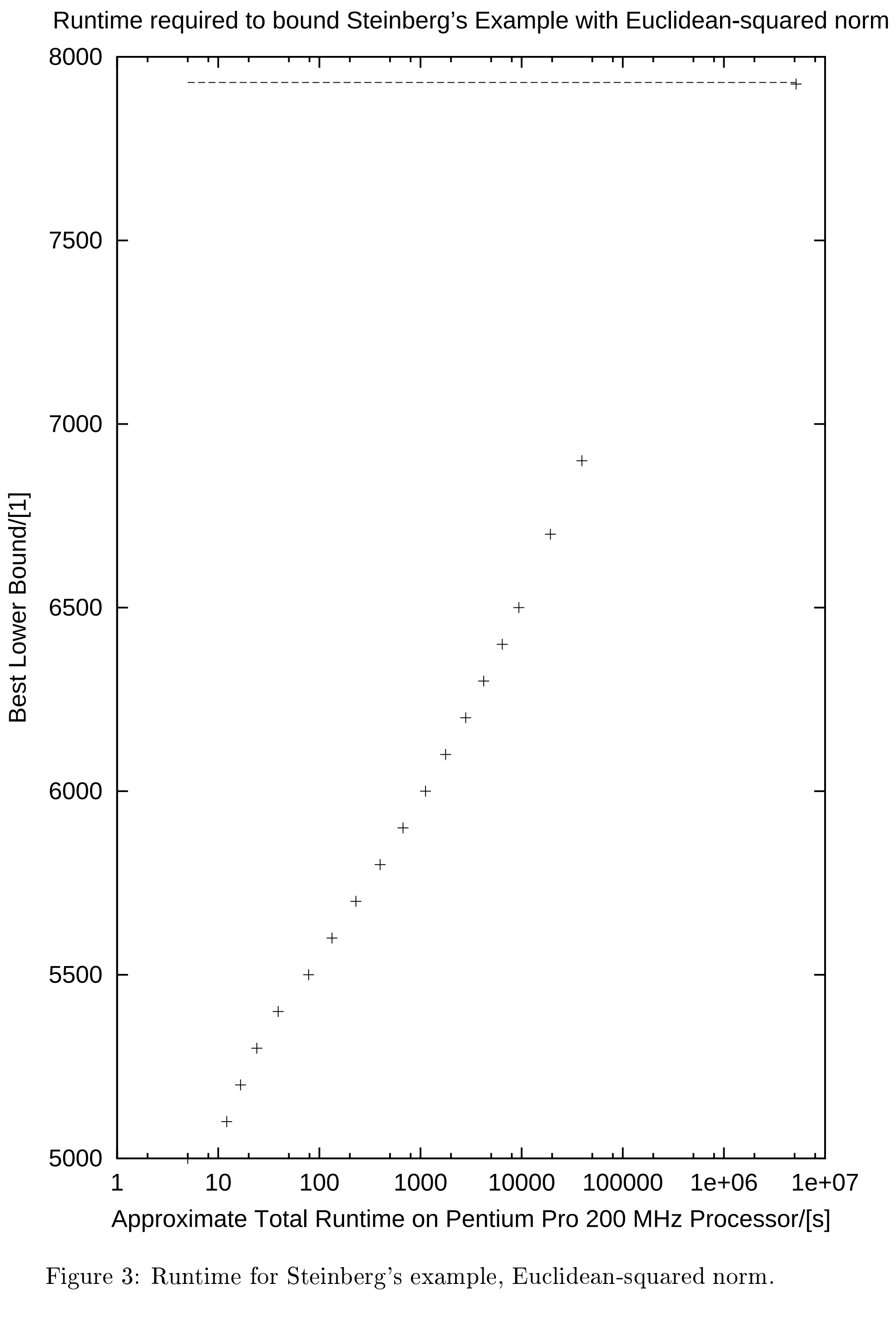

Figure 3: Runtime for Steinberg's example, Euclidean-squared norm.

35	2	17	16
3	8	18	9
5	4	10	1
6	13	7	15
11	12	20	28
27	14	19	29
26	23	32	30
25	21	34	31
24	22	33	36

Figure 4: Solution to Steinberg's example, Euclidean norm. Cost=4119.74.

with the optimum as the starting value. The optimal solutions to Steinberg's examples are shown in Figures 4 and 5.

5.3 Memory usage

The memory footprint of the program was small during the runs on Steinberg's example. Each slave took between 300 and 700 kilobytes of core memory (as indicated by Unix utilities), and the master consumed about the same amount of core memory. Also, each copy of the checkpoint file, which contained the entire relevant state of the computation, at no time took more than about 500 kilobytes of disk space. By lowering the execution priority of the processes, we were able unobtrusively to share active office workstations.

5.4 Summary

Steinberg's example with the Euclidean and Euclidean squared norms are the largest instances of QAP known to us to have proven optimal solutions[10]. We did not investigate the version of Steinberg's example with the Manhattan norm.

35	33	26	24
31	34	25	22
30	32	23	21
29	19	14	27
28	20	12	11
1	7	13	6
15	10	4	5
9	18	8	3
16	17	2	36

Figure 5: Solution to Steinberg's example, Euclidean^2 norm. Cost=7926.

6 Acknowlegdments

The work described in this paper was supported by the Defense Advanced Research Projects Agency (DARPA) and monitored by the Office of Army Research. It was also supported in part by an Okawa Foundation Research Grant, and computer systems used for the implementation were donated by Intel Corporation. At the time of writing, the author is supported by an IBM Research Fellowship.

References

[1] L. R. Ford, Jr. and D. R. Fulkerson. Solving the transportation problem. *Management Science,* **3**, 1956, pp. 24–32.

[2] D. König. Über Graphen und ihre Anwendung auf Determinantentheorie und Mengenlehre. *Mathematische Annalen,* **77**, 453–465, 1916.

[3] H. W. Kuhn. The Hungarian method for the assignment problem. *Naval Research Logistics Quarterly,* **2**, 1955, pp. 83–97.

[4] J. Munkres. Algorithms for the assignment and transportation problems. *Journal of the SIAM,* **5**, 1957, pp. 32–38.

[5] L. Steinberg. The backboard wiring problem: a placement algorithm. *SIAM Review,* 1(**3**), January, 1961.

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

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

[8] E. L. Lawler. The Quadratic Assignment Problem. *Management Science,* **9**, 1963, pp. 586–599.

[9] R. E. Burkard, S. E. Karisch, and F. Rendl. QAPLIB—A Quadratic Assignment Problem Library.
`http://www.imm.dtu.dk/~sk/qaplib/inst.html`

[10] S. E. Karisch. Private communication, 1998.

[11] `http://www.distributed.net/`

[12] `http://www.setiathome.ssl.berkeley.edu/`