A Combinatorial Search with Dancing Links

Author: Chris Morgan

Supervisor: Alexandre Tiskin

Year of Study: 1999–2000

Abstract

Knuth claims that backtracking algorithms may be improved by a technique known as *dancing links*, based entirely on linked list operations. We investigate his claim by considering some examples. Firstly, the n-queens problem requires n queens to be placed on an $n \times n$ *chess* board so that no queen is attacking another. Secondly, the tripod packing problem, posed by Stein and Szabó. A tripod consists of a corner cube, and three perpendicular arms. How many non-overlapping tripods can we fit into a cube of size n?

Keywords

Algorithm, Branch and Bound, Combinatorics, Dancing Links, Linked List, N-Queens, Optimisation, Tripod Packing.

Contents

0.1 Structure of Report 1

1 Project Introduction **2**

1.1 Motivation for Project 2

1.2 Author's Assessment of the Project 3

2 Dancing Links **4**

2.1 Introduction to Dancing Links 4

2.2 N-Queens Problem 6

3 Tripod Packing Problem **8**

3.1 Introduction 8

3.2 The Basics 9

3.3 Symmetries and Properties of Optimal Packings 14

3.3.1 Empty Plane 14

3.3.2 Corner Tripods 15

3.3.3 Reflection 16

3.3.4 Tripod Translation 16

3.4 Method of Upper Bounds (branch and bound) 17

3.4.1 Reduced Projection Method 19

3.4.2 Lines Method 23

3.4.3 Branch Comparison Method 24

3.4.4 Improving the Upper Bounds Method 25

3.5 Miscellaneous 28
3.6 The Greedy Approach 29
3.7 Practice Problem 30
3.8 Modelling Tripod Packings 31

4 Project Management **32**
4.1 Project Summary 32
4.1.1 Starting the project 32
4.1.2 Tripod Packing Problem 33

5 Conclusions **39**
5.1 Conclusions 39
5.2 Further Work 41

Acknowledgement **43**

A N-Queens Results **45**

B Selected Pseudocode **47**
B.1 Dancing Links Naive Algorithm 47
B.2 Adding to the Basic Algorithm 50

C Tripod Packing Results **53**

D Tripod Packing Code **55**

0.1 Structure of Report

The technical content of this project is very large, and consequently this applies to the report. I have tried to keep the technical and project management sides separate, but inevitably this has not always been possible. Here is a brief outline of the structure of the report, highlighting the sections I consider most important:

Chapter 1 concerns the motivation behind the project.

Chapter 2 introduces the idea of dancing links, and uses the n-queens problem as an example application, so can be omitted by someone familiar with Knuth's Dancing Links paper [4].

Chapter 3 is the most technical section of the report, and the most interesting part of the project. This section combined with the results in Appendix C is more in the style of a scientific report, and gives in depth details of how the algorithm works. To try to help the readability of this chapter, some pseudocode is given in Appendix B.

Chapter 4 deals with the management side of the project, and describes the project's progress. It explains why certain decisions were made, as well as some of my own views on the project.

Chapter 5 is the conclusion for the project, and gives ideas about further work.

The report is best read in order, although a full understanding of Chapter 3 is not essential for the final chapters.

Chapter 1

Project Introduction

1.1 Motivation for Project

The computer age has opened a whole new branch of mathematics, namely computational mathematics. A problem that would once have taken a lifetime to solve can now be solved by computer in seconds, and with the progress of computer technology, the boundary between what is feasible and unfeasible is ever changing. To match the ever-increasing computational power, new and more complicated problems are being considered. It is rarely enough to write the simplest algorithm to solve a particular problem; with a more *intelligent* algorithm, perhaps we can find an important result, or an elusive counterexample.

There are standard methods for solving problems: divide and conquer, dynamic programming and greedy algorithms are but a few of the more common techniques. However, these cannot solve all problems, and at times we must resort to a brute force method to check all possible combinations.

Occasionally, someone has an inspiration and a new idea emerges. A great idea may, in time, become indispensable, but who knows how many ideas fail to get the recognition they deserve?

For my project, I will be looking at the remarkably simple idea of dancing links. Whilst not exactly revolutionary, it is certainly a useful method. I will show that this method, in conjunction with other methods, can make a big impact on the study of certain problems.

1.2 Author's Assessment of the Project

There is a large technical content to this project, which is reflected by the nature of this report. The majority of this project is concerned with optimising the tripod packing problem, posed by Stein in [2, 7], in order to find results previously uncalculated. The original paper is mostly concerned with the theory behind the problem, rather than implementation techniques, so it has been necessary to develop my own ideas for the design of an algorithm.

This project makes use of a technique written about by Knuth in [4], known as dancing links. The technique is very simple, and deserves more recognition. The main focus of this report is to illustrate an additional example helped by the method, but we also show that this method alone is insufficient to write an efficient algorithm.

This project would be useful to someone studying the tripod packing problem, as it contains ideas relevant to both practical and theoretical study. I have included in the results the number of optimal packings (undocumented elsewhere) which could provide a way to check for errors in an implementation. It also gives another good example of how the dancing links technique can improve the efficiency of an algorithm. It is likely that similar ideas to those developed in this project could give equally dramatic savings for some other problems.

From a technical point of view, the work on the tripod packing problem has been the most successful part of the project, since I have been able to obtain results well beyond those previously calculated. From a personal perspective, the project has been successful in every area. Not only has the project been both an interesting and enjoyable course of study, but also I think I have successfully developed and managed the project within the obvious time and equipment constraints.

The only weakness in the project is the work relating to the *maximum set packing* problem, which I must emphasise is only a minor part of the project. Other than this, I do not feel the project has any significant weaknesses, although it would have been nice to find some *new* results, rather than simply confirming that known lower bounds are optimal.

Initially, I intended to concentrate on dancing links, studying more problems in less depth. Once I began working on the tripod packing problem, I realised there was far more work that could be done to improve upon known lower bounds, so I moved away from my deliberately loose specification to concentrate on this.

Chapter 2

Dancing Links

2.1 Introduction to Dancing Links

An essential concept for the project is the idea of *dancing links*, a term coined by Knuth in [4], although the idea was originally introduced in 1979 by Hitomtumatu and Noshita. The technique is simple, and based entirely on doubly linked lists. Knuth claims that dancing links can help improve the efficiency of backtracking algorithms, and gives several examples in his paper.

Linked lists are a very common data structure, and have many different applications. Adding and removing nodes in a linked list is achieved by adjusting pointers, and the deleted node is often destroyed to save memory. However, we can sometimes benefit from being less tidy, and by not automatically destroying a deleted node.

Consider a section of a linked list as in Figure 2.1, with *next* and *previous* pointers pointing to the next and previous nodes in the list. To delete say, a node q we simply set

$$\begin{aligned} previous(next(q)) &\leftarrow previous(q) \\ next(previous(q)) &\leftarrow next(q) \end{aligned} \tag{2.1}$$

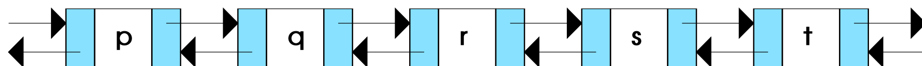

Figure 2.1: A section of linked list.

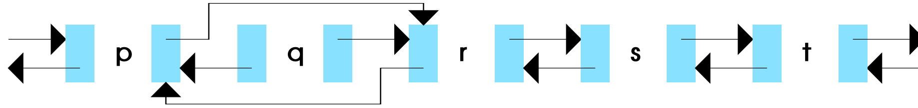

Figure 2.2: Node q deleted from the list.

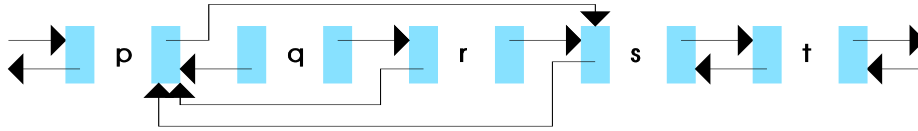

Figure 2.3: Node r deleted from the list.

so the list now appears as in Figure 2.2. We do not need to change the pointers of the deleted node q. To delete node r, we again have to adjust just two pointers, and the list appears as in Figure 2.3. The crucial part of the method is the ease with which we can replace the most recently deleted node. To replace node r we set

$$\begin{aligned} previous(next(r)) &\leftarrow r \\ next(previous(r)) &\leftarrow r \end{aligned} \tag{2.2}$$

and again we have a list as in Figure 2.3. Care must be taken however to replace the nodes in the reverse order to that in which they were deleted, as Figure 2.4 illustrates what happens should we attempt to replace node q without replacing node r.

It is easy to see that this property of linked lists could be useful in many applications, such as an *undo* and *redo* function, but it is less obvious how it could be used in a backtracking algorithm. To help understand why we should want such a technique, we will consider the n-queens problem (as used by Knuth in [4]), before embarking on the far more complicated tripod packing problem.

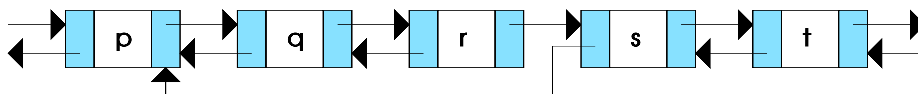

Figure 2.4: Attempt made to replace node q before node r.

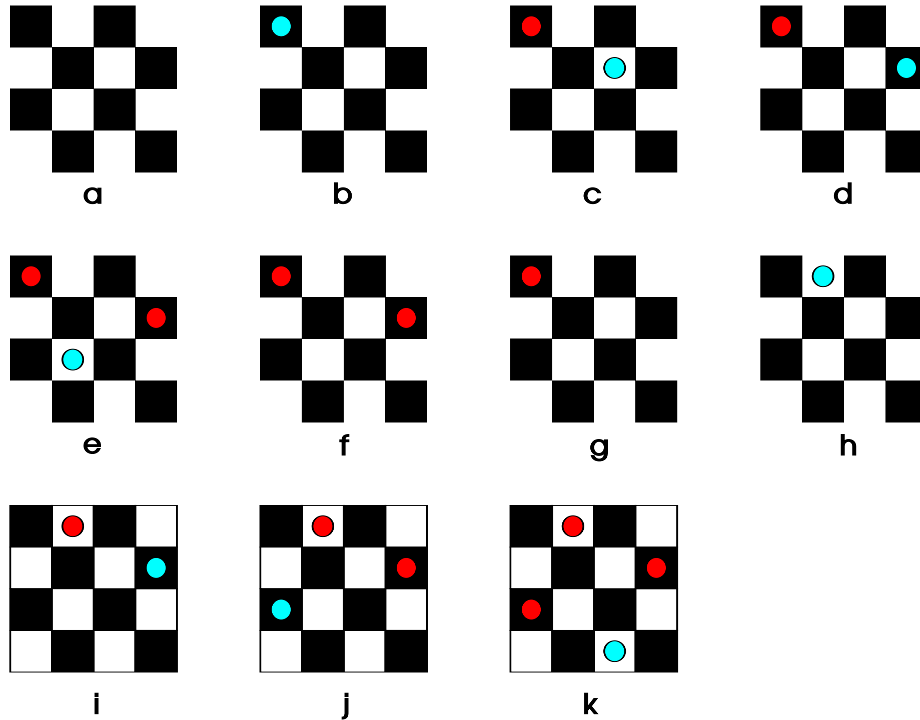

Figure 2.5: Solution for 4-queens problem.

2.2 N-Queens Problem

Problem 1 (The N-Queens Problem) *Using an $n \times n$ chess board, place n queens in such a way that no two queens are in the same rank, file or diagonal.*

We can use Dijkstra's algorithm to solve the n-queens problem. This does not store the board in the most obvious way, instead uses three global Boolean arrays to represent the files, diagonals and reverse diagonals. The algorithm places one queen in each rank. For each queen placed, the corresponding element is set in each of the three arrays, implying that we cannot place any queens in that particular line on the board. Solving the problem requires exactly one queen to be placed in each rank and in each file on the board. By systematically placing queens in each rank, we can avoid having to store the filled ranks. Figure 2.5 illustrates the algorithm up to finding a solution.

For each square of a rank, we must check against each of the three arrays to see whether a queen can be placed. This, however, is inefficient as it involves a lot of checking, and we can reduce this with the dancing links method.

We can replace the files array in Dijkstra's algorithm with a doubly linked list

(containing n nodes initially), each node corresponding to a file on the board. To place a queen, we must set the corresponding elements in both diagonal arrays, and delete the associated node from the list. Having placed a queen, if we do not have a solution we then move to the next rank on the board. The list consists of a reduced number of nodes, each corresponding to a file without a queen. By traversing the list from the head node, we cut out some of the nodes in the search tree where no queen can be placed. We may note however, that both algorithms are effectively the same, so they both place queens in exactly the same order.

Knuth claims that this method can almost halve the running time of the algorithm, and my own experiments (see Appendix A) show that it can even do slightly better than this. Knuth then concentrates on the exact cover problem, which requires us to find a set of rows in a matrix of 0s and 1s such that there is exactly one 1 in each column. The algorithm works by removing columns and rows from the matrix, which is easy to do with dancing links. By constructing an appropriate matrix, Knuth demonstrates how his exact cover algorithm can be used to solve the n-queens problem, which he claims is faster than the specific algorithm we have described for reasonable size n.

My experiments show the specific algorithm with dancing links to be fastest for the problem sizes I was able to try, although the exact cover algorithm produces a significantly smaller search tree. For this reason I have included the relevant running times based on my own Pentium 90 PC running Linux, although these times will vary on other systems. However, the claim relates to the running times rather than the tree sizes, and it is the running times that are greater problem for the algorithm.[1] By considering the size of trees produced by the algorithms, I would say it is likely that the exact cover algorithm is fastest for larger problems, but so far it has not been feasible to test this.

For completeness, it is worth noting that for $n > 3$ a solution to the n-queens problem can be found in linear time as mentioned by Sosic in [6]. However, no polynomial time method is known to find all solutions.

[1]This is based on my own implementation of the dancing links version of Dijkstra's algorithm, and both Knuth's and my own implementation of the exact cover algorithm.

Chapter 3

Tripod Packing Problem

3.1 Introduction

We are now ready to use dancing links with a deeper problem, and so this section is about finding an optimal solution to a packing problem discussed by Stein in [2, 7].

Definition 2 (k-tripod) *A k-*tripod *consists of a unit* corner *cube, with three arms each of length k stuck on non-opposing sides.*

Let us simplify our notation and assume the tripods are of suitable size, so we will now refer to tripods rather than k-tripods. By suitable size, we mean that if a tripod is placed at any position inside a cube, then each arm extends to a face of the cube.

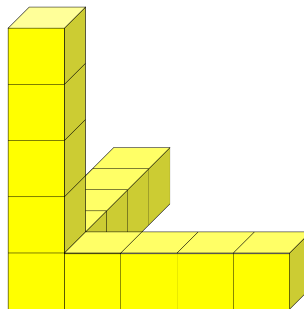

Figure 3.1: 4-tripod.

Problem 3 (Tripod packing problem) *What is the maximum number of tripods that can be placed in a cube of size n, so that all tripods face the same way, and no two tripods overlap.*

We will denote the number of tripods in an optimal packing for a cube of size n by $f(n)$.

3.2 The Basics

To solve the tripod packing problem we could choose a packing order, and naively implement a depth first search of the tree, with no consideration to the many symmetries within the problem. However, the complexity of such an algorithm is very high, and for any reasonable n this approach would be unfeasible. Any useful algorithm must cut down the number of possibilities at most stages, and we will consider some such optimisations shortly.

The general idea of the algorithm is to start at one corner of the cube we are packing, and for each position in the cube try placing a tripod (if possible), and then not placing a tripod at the same position.[1]. In both cases, the procedure is called recursively to complete the packing. For now we will assume we are packing a cube of size n.

Let us set up a coordinate system. We will number positions in the cube from 0 to $n-1$ along the axes x, y and z, as in Figure 3.2. We will say that a tripod is placed at position (x, y, z) if the corner cube of the tripod is placed at (x, y, z). For example the tripod in Figure 3.2 is in position $(0, 1, 1)$.

Definition 4 (Natural Packing Order) *The natural packing order is when we commence packing from one face of the cube and work towards the opposite face, placing the tripods in each plane (parallel to the initial plane) we pass through. To backtrack we remove the tripods in reverse order to that in which they were placed. Most importantly, we finish placing tripods in one plane before starting on the next.*

The natural packing order is the most intuitive order to pack the tripods, and from now on, we will assume this order unless otherwise stated. However, there are several possible natural orders, so we will choose to work from the

[1] Alternatively, we could proceed without placing a tripod before attempting to place a tripod. It makes no difference at this stage which order we choose, but later we will see that this is important.

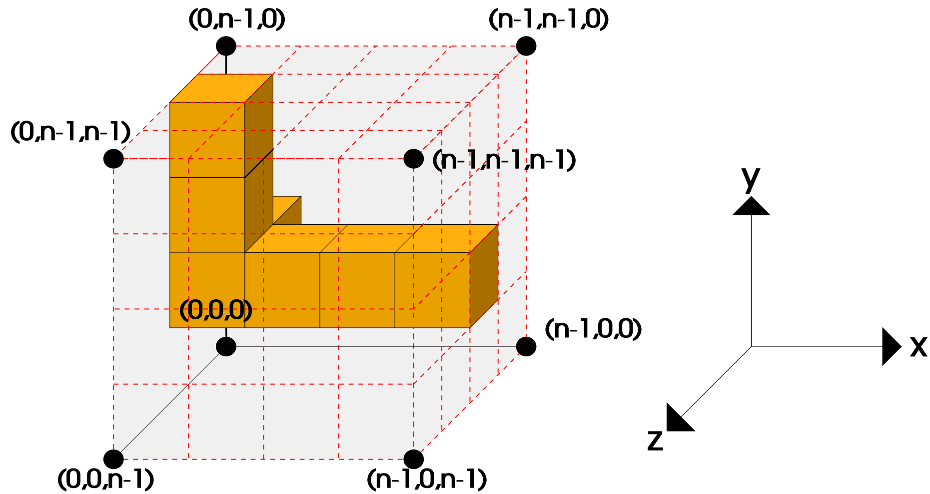

Figure 3.2: Order of packing.

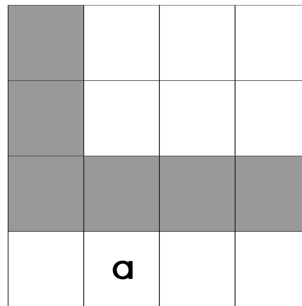

Figure 3.3: Projection of cube in Figure 3.2.

corner $(0, 0, 0)$ along the x axis first, followed by the y axis and finally the z axis. The arms of the tripods will point as shown in Figure 3.2 in the directions x, y and $-z$. It is important to remember that one of the tripods arms points through the previously filled planes, although there is an equivalent (but less intuitive) algorithm if this is not the case.

Now imagine that the tripods are made from an opaque material, and a light is shone from in front of the cube, so a shadow of the packing is projected onto a screen behind the cube. We lose the detail of the packing such as which planes any tripods are placed in, but this simplification can be very useful. Figure 3.3 shows the projection for the tripod in Figure 3.2.

We recall that one of the tripod arms points through the previously filled planes, and because of our chosen packing order, no tripod can be placed in

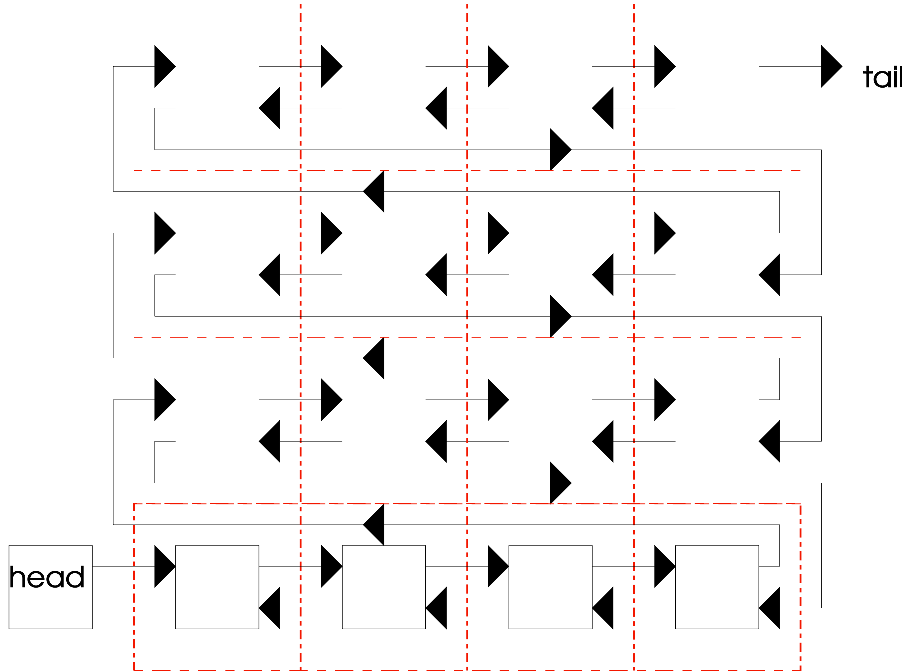

Figure 3.4: Linked list data structure for empty cube of size 4.

a position directly in front of one of the shaded cells. This enables us to use dancing links.

Since we have lost information about one dimension, we will use the most obvious two dimensional coordinate system to refer to positions in the projection. Therefore, a position (x, y, z) in the cube is mapped to (x, y) in the projection.

It is convenient to store the projection in an $n \times n$ dimensional array, but to enable us to use dancing links, the nodes of the array must also be stored in a doubly linked list. We can then access the data structure either as an array or as a linked list. We must create the list in the correct order, so that we traverse the linked list in the same order that we place tripods in each plane. Figure 3.4 represents the data structure for the case $n = 4$, before any tripods have been placed.

When placing a tripod, we must update the projection by deleting the corresponding nodes in the linked list. The data structure for the projection in Figure 3.3 is then represented in Figure 3.5 (to simplify the diagram, we have omitted the pointers of the deleted nodes since these depend on the order of deletion). To continue without placing a tripod we simple move to the next node in the linked list, and continue with the algorithm.

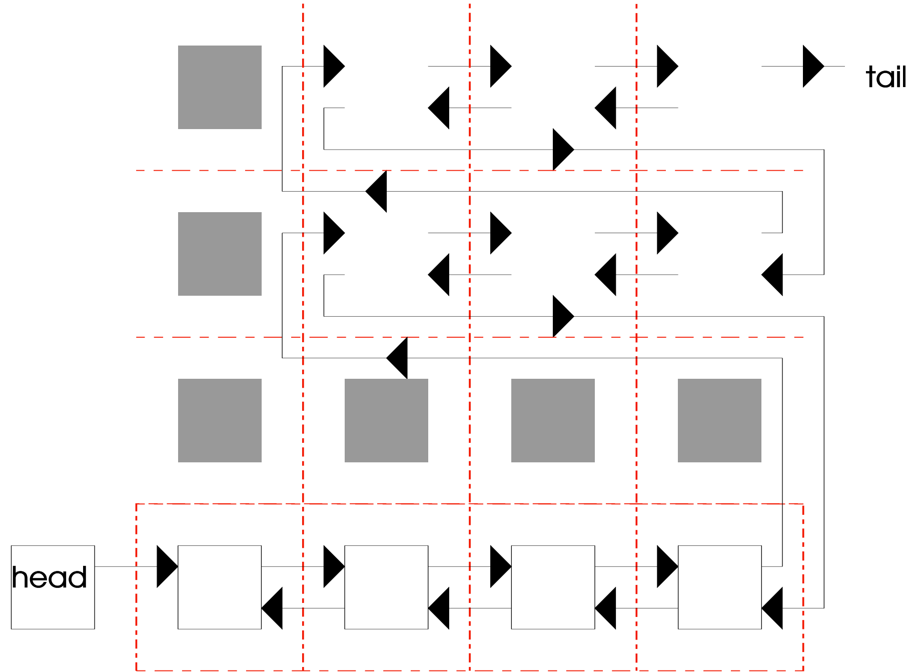

Figure 3.5: Linked list data structure for cube in Figure 3.2.

It is best to delete the node corresponding to the corner cube last, as we can continue to traverse the list in the usual way. The reason for this is that the corner cube corresponds to the current node in the linked list, and this node's *previous* and *next* pointers will point to the previous and next nodes in the linked list. This may not be the case if other nodes are deleted subsequently, as can be seen from Section 2.1, and in particular, if the current node is node q in Figures 2.1–2.3. It makes no difference in what order we delete the remaining nodes, provided we remember to replace them in the reverse order.

Once we reach the end of the list, we are ready to progress to the next plane of the cube, and we again traverse the list from the head node. Since many nodes corresponding to positions where a tripod cannot be placed have been removed, we dramatically cut down the search tree.

The most important point to remember is that a node deleted from the linked list corresponds to a position where a tripod cannot be placed.

Now suppose we move on to a new plane and try placing a tripod at position $(1, 0)$ (a) of Figure 3.3 to obtain Figure 3.6. The node corresponding to $(1, 1)$ (b) has already been deleted. Trying to delete a node twice could produce some very unpredictable results, so we must protect against this. When we are

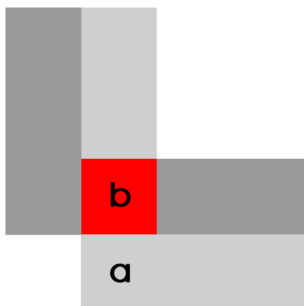

Figure 3.6: We must be careful not to delete node b twice.

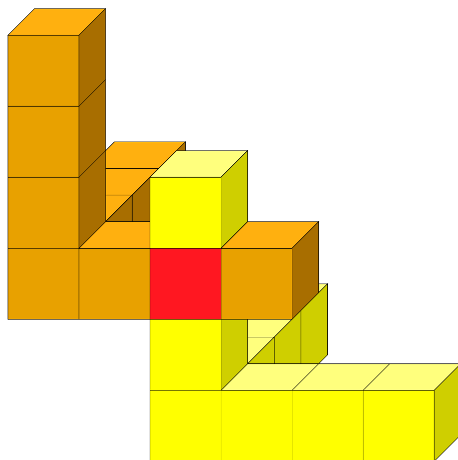

Figure 3.7: An arm of the left tripod clashes with an existing tripod in the same plane.

backtracking and need to remove this tripod we must add the node to the list if and only if we had to delete the node to place the tripod. This can be achieved in a number of ways, but I have chosen to store how many times the node would have been removed.

Unfortunately, this linked list structure does not store enough for our algorithm, and the existence of a node in the list does not guarantee that a tripod can be placed. The problem is due to clashes within the plane we are filling. One or both arms of the new tripod could clash with tripods already placed in the same plane. By choosing a good packing order, we can reduce this to at most one type of clash, that is between the arm in the x direction of the new tripod and the arm in the z direction of an existing tripod (Figure 3.7). We can therefore place a tripod at position (x, y, z) if and only if the corresponding

node is in our linked list and x is greater than the x coordinate of all the other tripods in the same plane. The first condition we have already discussed, and the second is far simpler. For each plane (we are filling) we will use a stack to store the x coordinates of the tripods. The second condition is then satisfied if the stack is empty or x is greater than the element at the top of the stack.

A more formal description of the basic algorithm is given in Appendix B in the form of pseudocode. The code does little to help follow the ideas in this chapter, so has been kept separate to aid the readability of the report.

We may also store the packings in matrix form as described by Stein in [2, 7]. The main use of this is to display the packings for output, although it can sometimes provide us with a more convenient way to check the current packing.

3.3 Symmetries and Properties of Optimal Packings

In this section, we consider some symmetries and properties of optimal solutions that will enable us to improve the efficiency of the algorithm. Some properties discussed in this section can cause us to *miss* some optimal solutions, but in each case, we can show that some optimal solution has a particular property. We must however be careful which properties we use together, since we cannot always be sure that an optimal solution has all these properties.

3.3.1 Empty Plane

We may note that an optimal packing for a cube of size n must contain at least n tripods as we can place n tripods along the diagonal of any plane. Now, suppose we have an optimal packing that has an empty plane, then some parallel plane must contain more than one tripod. We can *shift* a tripod into the empty plane from an adjacent parallel plane, and inductively we obtain an optimal solution with no empty planes.

The only examples I have found of optimal packings for cubes with empty planes are for the case when $n = 2$. I suspect, but have been unable to prove, that this is a unique case.

To use this heuristic in our algorithm, we must check that at least one tripod has been placed in the current plane when we are ready to move to the next.

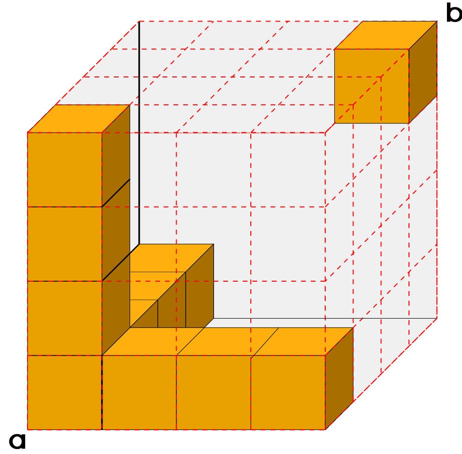

Figure 3.8: We can assume tripods are placed in the corners.

We backtrack if this condition is not satisfied.

3.3.2 Corner Tripods

Stein [2] tells us that we may assume that tripods are placed at two of the corners of the cube shown in Figure 3.8. Alexandre Tiskin justified this by noting that tripod (a) in position $(n-1, n-1, 0)$ of Figure 3.8 is a single cube, so any optimal packing must have this position occupied. If this position is occupied by part of a tripod arm, then that tripod can be shifted to the corner position to obtain an equivalent packing.

To see that the position (b) of Figure 3.8 can be occupied, Alexandre Tiskin observed that we can swap the direction of all three arms of each tripod, to obtain another valid packing. We can now assume that a tripod is placed at (b), position $(0, 0, n-1)$ by the same argument as above.

The problem with this method is that we cannot simply place the tripods in the corners at the start of the algorithm, since this would break our packing order. To get around this, we must wait until the algorithm reaches the correct stage in the packing, then ensure that a tripod is placed. However, this in turn leads to a problem if a tripod cannot be placed in the corner. We must therefore ensure that no tripod is placed in a position that would clash with a tripod in one of the necessary corners.

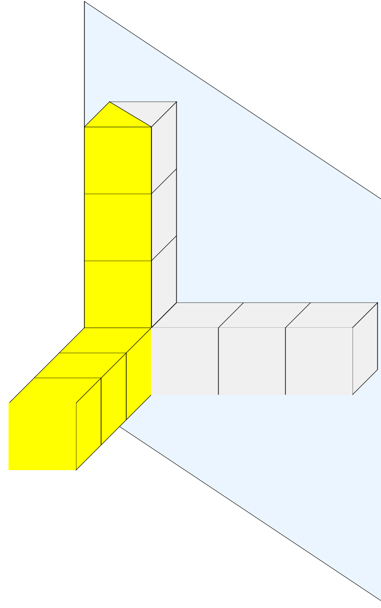

Figure 3.9: One of three planes of reflective symmetry.

3.3.3 Reflection

A tripod has three planes of reflectional symmetry, one of which is shown in Figure 3.9. The other two reflections can be seen by rotating the tripod about the corner cube.

We can make use of one of these reflections by ensuring that the first tripod in the first plane is always placed on the diagonal $(i, i, 0)$, or the same side of the diagonal. This may cause us to *miss* an optimal packing, but in this case, we will find the reflection of the packing at some stage of the algorithm. If we are interested in finding all optimal packings, we must check whether the reflection of a packing can be found by the algorithm each time we find a new solution. This is done by reflecting the packing in the diagonal $(i, i, 0)$, and checking the position of the tripod that would have had to be placed first to create such a packing.

3.3.4 Tripod Translation

Suppose we have found a position where we can place a tripod. Remember that one of the arms of the tripod points through the previously filled planes. Now, if a tripod could be placed at the same position in an earlier plane (Figure 3.10),

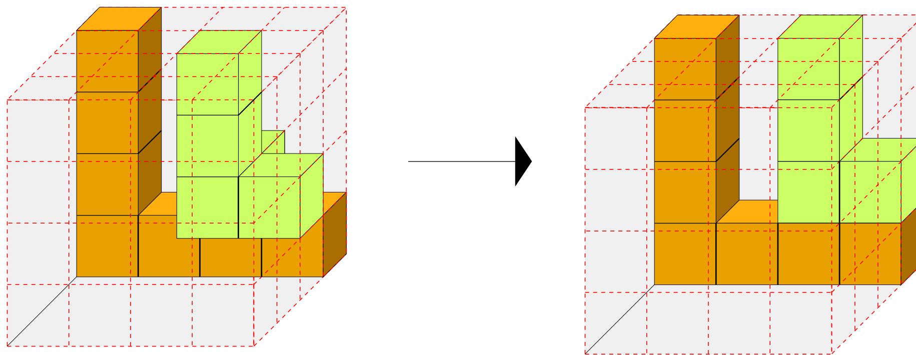

Figure 3.10: We can push the near tripod to the previous plane.

then we should not place a tripod. The reason for this is that in whichever plane we place the tripod, the resulting projection will be identical. However, placing the tripod in the current plane will restrict what other tripods can be placed in the same plane. Placing the tripod in the earlier plane will of course be considered at some point during the algorithm. If we choose to try placing a tripod at each position before proceeding without placing a tripod, then the packing with the tripod in the earlier plane will already have been considered, otherwise it will be considered at a later point in the algorithm.

If we require all solutions then we can only use the reflective property of the tripods, but requiring a single solution is far more flexible. Clearly, it is safe to use the reflective property and corner property together. Conflicts could occur should we try to use the other properties, although this could be rectified by checking for special cases.

3.4 Method of Upper Bounds (branch and bound)

This is really a continuation of the previous section, but is important enough to deserve a section (if not a whole chapter) to itself. We will see how this method can improve the algorithm to a far greater extent than all the methods of the previous section (including dancing links) put together. The method uses a branch and bound approach, such as discussed by Brassard and Bratley [1, pages 312–316].

Suppose that at some stage of the packing we can get an upper bound on the number of tripods to which the current packing be expanded, then we can often decide that a particular branch will not lead to a solution, so does not

need to be investigated. More formally, suppose we have an upper bound (U) for the number of tripods we can still add, a lower bound (L) for an optimal solution. Let us suppose that in the packing we have already placed T tripods. Then, if we want just one optimal solution, we only need to continue along a branch while

$$T + U > L \tag{3.1}$$

or, if we want to find all optimal solutions, while

$$T + U \geq L \tag{3.2}$$

This is fairly general to all optimisation problems. However, what is important is how to get the upper and lower bounds. The lower bound is very simple, since we can use the size of the best packing the algorithm has already found.

The method works much better if we can get close upper and lower bounds as quickly as possible, so basing the initial lower bound on known constructions can save a lot of time. Given a valid tripod packing, we can check to see if the packing is optimal by running the algorithm using (3.1), and choosing the initial lower bound to be the number of tripods in the given packing. Should the algorithm fail to find any larger sized packings, then the given packing is indeed optimal, and so we may show optimality without the algorithm finding any optimal packing. Choosing the initial lower bound in this way is more important when the algorithm is slow to find an optimal solution; unsurprisingly, this is usually the case for problems with few optimal solutions.

Until now, it has not mattered whether we evaluate placing a tripod or not placing a tripod first, since the same decedent nodes are always evaluated. However, now that we want to find a good lower bound for the optimal solution as quickly as possible, experimentation shows it is best to evaluate placing a tripod first. This should not be too surprising, as taking this approach we immediately place several tripods in the first plane, whereas we would otherwise evaluate many nodes before placing the first tripod.

The upper bound is the harder part of the inequality to find, and we will discuss some methods for finding one. None of the methods are perfect, but the first two can be used together, and the lower of the two bounds can be used as the overall upper bound..

3.4.1 Reduced Projection Method

Let us initially assume that we have finished packing one plane and are about to move to the next, and that there are p empty planes left on which to fit tripods. For a very loose bound we may note that we can fit at most n tripods per plane, and hence choose

$$U = pn.$$

With a bit of work we can greatly improve upon this bound, and so save a lot of processing time.

First let us generalise the problem to packing a cuboid of size $l \times m \times n$, the optimal size of which we denote as $f(l, m, n)$. We can now refer to our old notation as

$$f(n) \equiv f(n, n, n).$$

Let us assume from now on that when calculating $f(l, m, n)$, values l, m and n correspond to the sides as shown in Figure 3.11, and the packing order is the same as before.

We must be careful with the symmetries and optimisations discussed in Section 3.3. We can no longer assume that there are no empty planes, but we can still use the heuristic. The difference is that now we must assume all the empty planes are *shifted* to the final planes, so we only move to a new plane if we have placed at least one tripod in the current plane. For the reflective property, this can only be used if $l = m$. The changes for the corner tripod property should be obvious, while the use of the tripod translation property requires no changes.

As an initial observation, we may note that by symmetry

$$f(l, m, n) = f(l, n, m) = f(n, l, m) = f(n, m, l) = f(m, l, n) = f(m, n, l) \quad (3.3)$$

Now, when we have p planes remaining in which to place tripods, we can set

$$U = f(l, m, p)$$

as an upper bound. This gives a noticeable improvement over our earlier bound, but we can improve on this still further. Let us return to our earlier example (Figure 3.12).

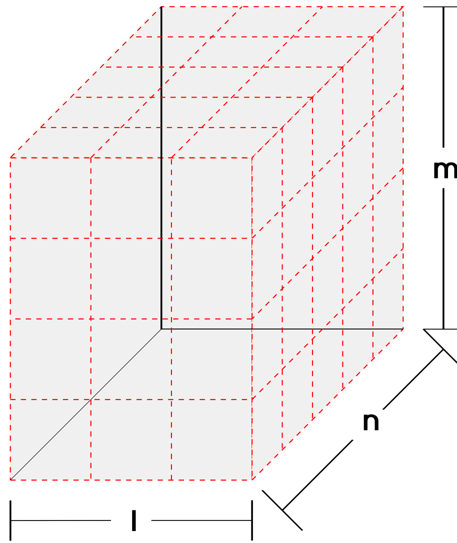

Figure 3.11: The size of the cuboid.

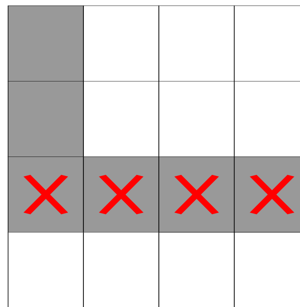

Figure 3.12: No tripods can be placed on one of the rows.

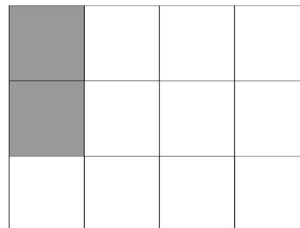

Figure 3.13: The *reduced projection* of Figure 3.12.

Clearly we cannot place any tripods in the marked row, so instead the problem is equivalent to trying to pack tripods in p planes with the projection in Figure 3.13. We will call this the *reduced projection.*

Hence, if we denote by c the number of columns with at least one empty (unfilled) cell, and by r the number of rows with at least one empty cell, then we can set

$$U = f(c, r, p).$$

where initially, $c = l$, $r = m$, $p = n$.

We can keep track of the values of c and r throughout the algorithm. For each row and each column in the projection we store the number of filled cells, and update these when placing or removing a tripod. When we place a tripod that causes a column in the projection to be filled we decrement c, and similarly increment c when we remove a tripod that causes a previously filled column to have an empty cell. The same method is used for rows in the projection.

To use this form of upper bounds we must calculate some values of $f(l, m, n)$ before we start to solve the problem. The obvious question is what values of $f(l, m, n)$ should we calculate? The answer is not obvious, and even with experimentation is hard to answer fully. We want to calculate

$$f(i, j, k) \qquad \begin{cases} 1 \leq i \leq l \\ 1 \leq j \leq m \\ 1 \leq k \leq K \quad \text{(for some } K\text{)} \end{cases}$$

For small k, the calculation is fast, and the result will generally give a good upper bound for the number of tripods that can still be placed. For large k, the calculation takes far longer, and the upper bound produced will be used early on in a packing when it is hard to tell whether the branch can lead to a new solution. So by making K too large, we save slightly more time in the main algorithm, but far more time is used to calculate the extra bounds. Choosing $K = \frac{2n}{3}$ seems to give a reasonable saving in most cases.

For this pre-processing, we only require the values of $f(p, q, r)$, not the actual solutions, so we should use (3.1) and as many optimisations described in the previous section as possible, regardless of whether we want all solutions for the actual problem or not. Using (3.3) we can save a substantial amount of time on these calculations, as well as getting for *free* values we had decided were not worth calculating.

An important point is that the equality $f(l, m, n) = f(m, n, l)$ does not mean that these two values require the same amount of processing to calculate. Although we could use the reflection property if two dimensions are the same size, it is in general best to choose the dimensions that produce the shortest possible linked list, and most planes. The order of the other two dimensions makes very little difference. Thus to find $f(2, 4, 3)$ we should calculate $f(2, 3, 4)$.

Recall that we store the linked list for the projection within an $l \times m$ dimensional array, but for the pre-processing we will need some smaller projections. One way to solve this is to create a large enough linked list for all the projections, then remove nodes of the linked list until we have a list of the required size. We could remove and replace these nodes one at a time using (2.1), but a slight change to the dancing links method will enable us to delete, with care, a whole section of the list. Let us assume that nodes p and q are in the list, such that q is not before node p in the list. Then to delete nodes from p to q

$$\begin{aligned} previous(next(q)) &\leftarrow previous(p) \\ next(previous(p)) &\leftarrow next(q) \end{aligned}$$

Clearly if p and q are the same node, then this reduces to the original method (2.1). Similarly to replace the section, we use

$$\begin{aligned} previous(next(q)) &\leftarrow q \\ next(previous(p)) &\leftarrow p \end{aligned}$$

which reduces to (2.2) as we would expect if p is the same node as q.

We do not use this within the main algorithm, even when deleting consecutive nodes. The reason for this is that it is often hard to tell exactly which nodes are to be deleted, and each node to be deleted still requires other processing.

In many situations the reduced projection method can give a good upper bound. However, let us consider a particularly bad projection for the above method shown in Figure 3.14.

In this case all rows and columns have at least one empty cell, and so using the above method with p planes to fill would lead to a very bad upper bound.

So what can be done about this worst case upper bound? One option is to consider a second method to obtain an upper bound, then use the smaller of the two bounds. We will now investigate this approach.

Figure 3.14: Worst case for first upper bound method.

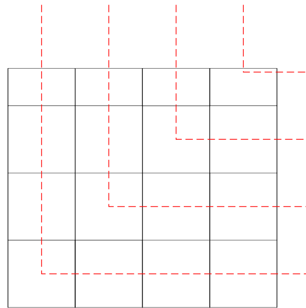

Figure 3.15: Only one tripod can be placed on a marked line in any plane.

3.4.2 Lines Method

We may note that in any single plane, there are always combinations of tripods that cannot be placed together. As an example, we cannot place two tripods in the same row or column of our projection. In Figure 3.15, we can place at most one tripod on any one of the lines indicated in any plane.

Let $q(i)$ be the line through cell (i, i) of the projection as in Figure 3.15. Let us define $k(q(i))$ to be the number of empty cells in the projection for the line $q(i)$. Then, in the final p planes we can fit at most p tripods and at most $k(q(i))$ tripods over line $q(i)$. Therefore, using all lines we can take our upper bound U to be

$$U = \sum_{i=0}^{\min(l,m)-1} \min(k(q(i)), p)$$

Let us see how this helps our worst case situation for the earlier upper bound. If we draw the lines as in Figure 3.15 onto the projection (Figure 3.16), we can calculate the upper bound U as

Figure 3.16: We can only fit one tripod per plane.

$$U = \sum_{i=0}^{\min(l,m)-1} \min(k(q(i)), p) = \min(k(q(0)), p) = \min(7, p)$$

This turns out to be the least upper bound for the example. The method is not always this good, but seems to work particularly well for the very bad cases in the reduced projection method. We can take the lower of the two upper bounds as our overall upper bound, and use either (3.1) or (3.2) to reduce many branches of the search tree.

For the lines method, there are several sets of lines we could use, but the set described seem the best, because

1. The profile of the lines matches the shape of the tripods.
2. Any set of lines must contain at least $\min(l, m)$ elements as we can fit this number of lines in any plane, so our set is the minimum size.
3. The lengths of lines vary, so we often remove a large number of cells from the projection at a cost of only a few tripods.

A bit of experimentation at this stage will show a huge difference between the running times for finding all solutions (3.2) and one solution (3.1). This shows that if we can calculate a better upper bound we could still save a lot of running time. We can also see that both methods of finding upper bounds are very bad in many cases, so there is still potential room for improvement. In fact, the whole problem can be considered as finding the least upper bound for any particular cuboid.

3.4.3 Branch Comparison Method

At a node in the search tree, having evaluated one child, can we say anything about the other? Using the naive algorithm, the largest packing obtainable by

placing a tripod is at most one greater than by not placing a tripod. This is clear since we can choose to place all subsequent tripods at the same positions.

To use this we must evaluate not placing a tripod first, and this can improve the naive algorithm, although nowhere near as much as the previous upper bound methods. However, this is a problem if we want to use the previous upper bound methods, since from the earlier discussion we decided that it is best to evaluate placing a tripod first.

The problems encountered using this method with the previous methods do not end here. Having evaluated a child node in the tree, we must pass the value of the largest obtainable packing up to the parent node. Since we do not always evaluate placing a tripod, the values passed back are often only upper bounds, so the method does not work as well as we might expect. However, things worsen when we try to use this method with the previous upper bound methods.

The previous upper bound methods often enable us to tell that a branch will not lead to an optimal solution, without having to evaluate it. We then only have an upper bound for the size of the largest packing that the branch leads to, which may be a substantial over estimate for the size. This upper bound must then be used to judge whether we need to try placing a tripod. The result is that we may have to investigate a branch by placing a tripod, whereas a more thorough investigation when not placing a tripod may show that this is not necessary.

The reduced projection and lines methods are far more effective than comparing branches, and the inclusion of the branch comparison method gives practically no improvement, even given a good initial lower bound. Without such a bound it is much slower, due to the order in which we evaluate the child nodes. For this reason, we shall omit the branch comparison method from our algorithm.

3.4.4 Improving the Upper Bounds Method

So far, we have only used the upper bound method when starting a new plane, but this is not the only time we can improve upon previous estimates of upper bounds. We may also be able to improve on our estimates after placing a tripod, since at least one extra cell of the projection has been filled.

The problem we encounter with obtaining upper bounds when within a plane is that the remaining space to be filled is less simple, and so less easy to find

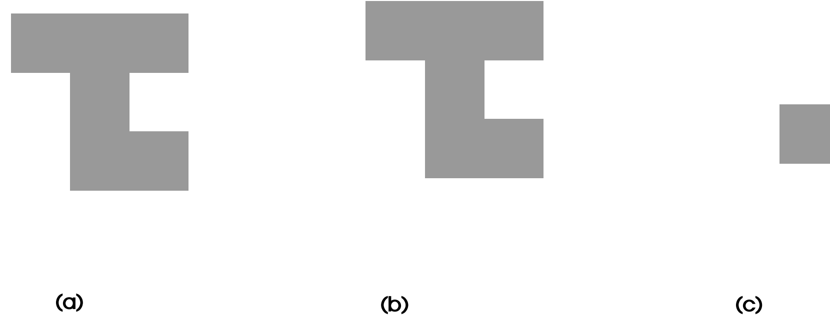

Figure 3.17: Combination of the two upper bound methods.

a good bound. To progress in this direction we have little choice but to bound the remaining space inside a cuboid. This is easy to do, and we can simply assume that the partially filled plane is in fact an empty plane, then proceed as before. Since we have placed a tripod, the projection may be reduced further than when starting the current plane, so we may improve on the upper bound.

We can take these two upper bound methods one stage further and combine them, rather than simply taking the lower of the two.

Let us first consider the example projection (a) in Figure 3.17, and let $p = 2$ be the number of planes remaining. Using the reduced projection method, we get

$$U = f(4, 4, 2) = 5$$

Using the lines method we get

$$U = 2 + 2 + 1 + 0 = 5$$

Suppose now we partition the projection as in (b) of Figure 3.17, and calculate upper bounds for each partition, the lines method on the outer section and reduced projection on the inner section. We can simplify the inner partition to obtain projections as in (c) of Figure 3.17. Hence, we get an upper bound

$$U = f(2, 2, 2) + \min(2, 7) = 2 + 2 = 4$$

We can keep applying this technique, repeatedly removing the outer edge of each of the inner sections in an attempt to get a better bound. But how do we efficiently partition the projection to find the dimensions of the reduced projection?

We can find $k(q(i))$ then remove all nodes on the line $k(i)$, to find the dimension of the reduced matrix and hence obtain an upper bound. By repeatedly removing the outer edge in this manner, we can obtain several upper bounds, before replacing the outer edges. We then choose the smallest of these bounds for use in the algorithm. Since this method incorporates the two basic bounding methods, we do not need to apply these methods separately.

We can improve this slightly by checking (3.1) or (3.2) every time we obtain a new bound, so we can stop early if we find a sufficiently small upper bound. In practice, we can reduce this processing further with little loss. If at any step of this routine the upper bound we obtain is higher than our lowest upper bound, then experimentation shows that a later step will rarely improve our upper bound, so we should stop searching. When the search is stopped early, the search tree is slightly larger, but each node takes less time, so the running time is significantly less.

The combined method yields a better upper bound than the two basic bounding methods, but takes significantly longer to check. Hence this should only be used when starting a new plane, and the two individual methods used within a plane after placing a tripod. The calculated bounds tend to be tighter when starting a new plane than within a plane, and for non trivial sized cuboids, we start a new plane far less frequently than we place a tripod. Using the combined method mid plane will only save a relatively small number of nodes, but at a high time cost.

Using the upper bounds methods and other optimisations, our algorithm is much cheaper than the naive algorithm in terms tree size and running time. Let us now take as closer look to see where about in the search tree the biggest savings are made. To do this we could consider the nodes at different depths in the search tree, but the naive algorithm produces a tree of height n^3 for a cube of size n, so this approach is impractical. Instead, we will look to see how long the algorithm spends filling each plane, and so we simply count the number of nodes used to evaluate each plane. Table 3.1 shows the number of nodes at different stages of the algorithm for a cube of size 5. The packing plane represents the z coordinate of the system, so in this example, the first plane is 0, and the final plane is 4.

This gives an indication of how the savings are made. In the naive algorithm, branches of the tree are not cropped, so every leaf node of the tree corresponds to the position $(l-1, m-1, n-1)$, and is lmn levels deep. Each level of the tree contains at least as many nodes as the previous level, so it follows that each

Packing plane	Naive	Naive & DL	All solutions	One solution
pre-processing	0	0	1,750	1,750
0	1,507	882	660	229
1	107,693	37,506	11,367	1,666
2	2,863,429	707,483	48,027	1,264
3	43,457,071	8,426,383	51,391	172
4	456,291,373	74,023,151	11,936	8
Total	502,721,073	83,195,405	125,131	5,089

Table 3.1: Number of nodes used to pack each plane for cube of size 5.

plane requires at least as many nodes as any previous plane.

In practice, we see that each plane of the cuboid requires significantly more nodes than the previous plane, so it is of little surprise that the naive algorithm spends the most time packing the final plane.

By introducing dancing links we can start backtracking when the projection is filled (i.e. the linked list is empty). By skipping many positions where no tripod can be placed, only a few leaves are as deep in the tree as those in the standard naive algorithm. The greatest savings are made in the later planes, when the projection is often fuller, and consequently the linked list is shorter.

When finding a single solution, many branches failing to lead to optimal packings are cropped at an early stage, so the algorithm rarely reaches later planes. Unsurprisingly, the upper bounds we generate work best at the later stages of a packing, so at an early stage in a packing the number of nodes increase. Consequently, most work is done around the middle planes.

The algorithm to find all solutions is a weaker form of that which concentrates on a single solution, so the size of the tree falls between the single solution algorithm and the naive method.

The results are similar for all but trivial sized cuboids, except the distribution of work becomes more extreme as the size of problem increases. Most work is done in final levels for the naive method, and middle levels for the fastest method.

3.5 Miscellaneous

All these improvements to the algorithm have dramatically reduced the running times for all but trivial sized problems. Despite this, the complexity of the

problem is still too large for us to study many new sizes of problem. One approach to reduce the running time is to make some assumptions about optimal packings, but this will only allow us to find lower bounds for the solutions. Here we will describe one such idea:

Definition 5 (Balanced) *We shall call a tripod packing* balanced*, if for any coordinate plane Q, the number of tripods in all planes parallel to Q is different by at most 1. If there are r tripods in the packing, and there are p planes parallel to plane Q, then the number of tripods k in every such plane will be*

$$\lfloor \frac{r}{p} \rfloor \leq k \leq \lceil \frac{r}{p} \rceil$$

Informally, a packing is balanced if the number of tripods in a plane is approximately the same as the number of tripods in each parallel plane.

Conjecture 6 *For every cuboid, there is a balanced optimal tripod packing.*

This approach can reduce the search tree size, and to do so we only need a weaker version of the conjecture. We require it to hold for the set of parallel planes we are filling. A good initial estimate for the size of optimal solution is required, and we can then calculate the minimum and maximum tripods in each plane. We will then disallow more than the maximum number of tripods in any plane, and backtrack if we have too few when we are ready to move to the next plane.

3.6 The Greedy Approach

I would also like to mention that I tried to write a greedy algorithm for packing tripods, in an effort to find some lower bounds. However, I was unable to find a good enough decision heuristic to make this approach worthwhile. All the attempts used the corner tripod property, and other heuristics I tried using include

- Choosing the position that takes up the least space in the cube.
- Choosing the position that least cuts down the number of other possible tripod positions.
- A combination of the first two heuristics.

Using a combination of the two heuristics gave the best solution, but it soon became apparent that the method was unsatisfactory. As an example, the greedy approach showed that $f(16) \geq 59$, but the construction in [2] shows that $f(16) \geq 64$.

3.7 Practice Problem

Stein proposed what he thought might be a simpler "practice problem" in [2], which is also unsolved. It is known that $\frac{f(j,j,k)}{j}$ converges as k increases. We will denote this limit by $c(k)$.

We can therefore write $c(k) = \sup \frac{f(j,j,k)}{j}$. It is known that $c(1) = 1$, $c(2) = \frac{4}{3}$, $c(3) = \frac{5}{3}$ and $c(4) = 2$. Stein suggests that following this pattern $c(5)$ could be $\frac{7}{3}$, but it is only known that it lies between $\frac{16}{7}$ and $\frac{5}{2}$.

The generalised cuboid version of the tripod packing problem from Section 3.4.1 is ideally suited for solving this problem, so we do not need to make any modifications to the algorithm.

In much the same way that the tripod packing problem can be generalised to a cuboid packing problem, so Stein's construction for tripod packings can be modified to work in the generalised case. It is easy to see that

$$\begin{aligned}
f(2l+1, 2m+1, 2n+1) &\geq 2f(l,m,n) + min(l,m) + min(l,n) + min(m,n) \\
f(2l+1, 2m+1, 2n) &\geq 2f(l,m,n) + min(l,m) \\
f(2l+1, 2m, 2n) &\geq 2f(l,m,n) \\
f(2l, 2m, 2n) &\geq 2f(l,m,n)
\end{aligned}$$

All other possibilities can be obtained by symmetry, using (3.3).

This generalised bound method is particularly bad for a cuboid of the form $(2l+1, 2m, 2n)$. For example

$$\begin{aligned}
f(8,8,7) &\geq 2f(4,4,3) \\
&= 12 \qquad\qquad f(8,8,7) = 21
\end{aligned}$$

Alternatively, specifically for the practice problem we can use

$$f(l,l,m) \quad \geq \quad \max\{f(i,i,m) + f(l-i, l-i, m) \quad | \quad 0 < i < l\}$$

Figure 3.18 illustrates the bounding method graphically. By placing tripods in a box of dimensions $i \times i \times m$ ($2 \times 2 \times 4$ in the figure) using a known construction, we have filled a shape that can be considered a tripod. We can then fill the

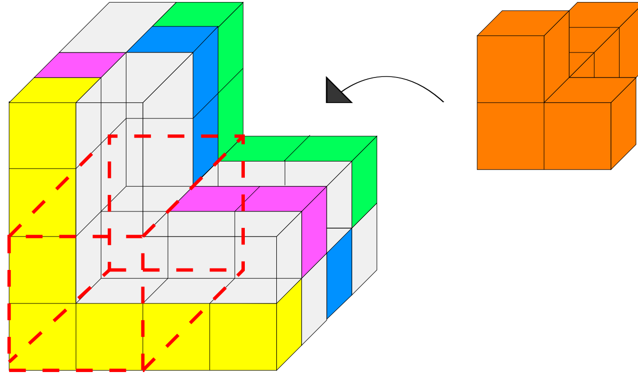

Figure 3.18: A simple construction for the practice problem.

remaining space, a box of dimension $(l-i) \times (l-i) \times m$ with a known tripod packing.

When we consider the previous example, we get

$$\begin{aligned} f(8,8,7) &\geq \max\{f(i,i,7) + f(8-i,8-i,7) \quad | \quad 0 < i < 8\} \\ &= f(3,3,7) + f(5,5,7) \\ &= 8 + 13 \\ &= 21 \end{aligned}$$

This method is far from perfect as can be seen if we consider the lower bound for the cube of size 7. Using this method we have $f(7) \geq 18$, whereas Stein's construction shows that $f(7) \geq 19$.

My results are listed in table C.3 of Appendix C.

3.8 Modelling Tripod Packings

It is very hard to visualise all but trivial tripod packings, so I have found it convenient to adapt my algorithm to produce VRML (Virtual Reality Modelling Language) code. These models can be viewed using a VRML viewer, and are very simple to create.

VRML has a number of simple predefined objects, one of which is a box. The simplest way to create a tripod is to use three appropriately defined boxes, placed in the appropriate positions. The tripods can be placed as necessary to build the packing, and by changing the colour of each tripod we obtain a very simple virtual model.

Chapter 4

Project Management

4.1 Project Summary

This chapter outlines the progress of the project, as well as reasons for some decisions made. A full understanding of the previous chapters is not required, but references are made to some parts of these chapters.

4.1.1 Starting the project

My first decision was to choose the language in which to implement Knuth's algorithms. I chose to use C++, despite the fact that I had never used the language before. My reasons for the decision are that it is fast compared to most languages, it is particularly good at using pointers – essential for dancing links, and that it was available both at university and at home. Having had experience with Java, I had no great difficulty in learning C++, and in hindsight it was definitely a good decision.

The original specification was deliberately loose due to the research nature of the project. Studying the n-queens problem helped me to understand more about dancing links, and why the method works, so this part of the project was useful rather than original. I chose to consider the time taken for the algorithm to run as well as tree sizes, since Knuth's claim was that his method was fastest. I used my own (unnetworked) PC to run the algorithm, since I was interested in the time taken, and the times could be affected by the network load if run on a university server.

Having finished studying the n-queens, I decided that there was little to be gained by spending time on the polyomino and tetrastick problems as in my original specification; these problems had been studied in depth by Knuth and others, so I decided to move away from my specification, and to look for another problem to study. I still believe that this was the right decision as I think I underestimated how long it would have taken to study those problems, and the time was far better spent looking at the tripod packing problem.

A task that I completed early in the project, was to write a project web page. This is currently available at http://www.dcs.warwick.ac.uk/~mavnp, and has been updated from time to time.

Another problem that I considered is that of the Soma cube, which requires all seven irregular shapes that can be formed by no more than four identical cubes to be packed into a larger cube. Gardner [3, page 53] claims that more than 230 essentially different solutions exist, although the exact number of solutions is unknown. This struck me as an ideal use for Knuth's exact cover algorithm, but ultimately a search on the Internet showed that the question had since been solved, and that 240 such solutions exist. Although there was scope for further investigation with the problem, I decided to search for an alternative problem to solve.

I considered briefly, but with no real success a number of NP-complete problems such as the Hamiltonian path problem. However, I was unable to find an efficient way of using dancing links to solve the problems. I also considered the knight tour problem, in which a knight must visit each square of a chessboard exactly once. Initially, I thought that it was a potential idea to study, since it was board based as the n-queens. However, I made no real progress on the problem (which is in fact an instance of the Hamiltonian path problem), so I turned my attention to what I thought could be a far more interesting problem.

4.1.2 Tripod Packing Problem

My supervisor, Alexandre Tiskin suggested the tripod packing problem as a possible area of study. Initially, I did not realise how much work this would involve, but was soon able to show that the dancing links version of the naive algorithm gave a reasonable improvement over the standard naive method.

I decided to continue to study the problem beyond dancing links, since it is not very conclusive to improve an algorithm using this method alone – there may be a far better algorithm that cannot use dancing links. Also, the problem

has not been studied by too many people, and I had already verified all known results of the main problem.

I first noticed the empty plane property (Section 3.3.1), which seemed to halve the running time for the algorithm. Alexandre suggested the corner tripods, as well as how to use the reflective property. I also discovered the tripod translation property soon after. Using some of these optimisations I was able to show $f(6) = 14$.

These properties improved the running time of my then best algorithm by a reasonable factor, although with later ideas these improvements seemed less significant. My project had reached this stage when I completed the progress report, at the end of the first term.

I briefly tried to write a greedy algorithm, but was not particularly surprised at the lack of success. I felt that such a simple approach would already have been discovered, although it was still necessary to check.

Maximum Set Packing Problem

Knuth develops a dancing links algorithm to solve the exact cover problem in [4]. He gives examples of problems that reduce from the exact cover problem, and uses his algorithm to solve these problems.

The tripod packing problem has some similarities with the problems solved by Knuth, but is subtly different in that it is an optimisation problem, rather than simply a packing problem. In Knuth's problems, we knew exactly how many objects could be fitted inside the object we were packing, and that a successful packing completely filled this object. The tripod packing problem does not have these two properties, so we cannot directly use the exact cover algorithm.

Closely related to the exact cover problem is the maximum set packing problem. Like the exact cover problem, we have a matrix of 0s and 1s. The problem is to find the largest set of rows containing at most one 1 in each column. For example, given the matrix

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

we see that the maximum set packing is the set consisting of the final two rows, but the matrix has no exact cover.

This leads to two questions. Firstly, can we adapt Knuth's algorithm to solve the maximum set packing problem, and secondly, can it be used to solve the tripod packing problem? Perhaps the fact that I am discussing this in this section rather than the previous chapter is a bit of a give away. However, I believed that the answer to both these questions was affirmative.

Knuth summarises his algorithm as

> If A is empty, the problem is solved; terminate successfully.
> Otherwise choose a column, c (deterministically).
> Choose a row, r such that $A[r, c] = 1$ (nondeterministically).
> Include r in the partial solution.
> For each j such that $A[r, j] = 1$,
>   delete column j from matrix A;
>   for each i such that $A[i, j] = 1$,
>   delete row i from matrix A.
> Repeat this algorithm recursively on the reduced matrix A.

I believed that the maximum set packing problem could be solved by changing the way a column is chosen deterministically in the second line. The exact cover can be solved whichever column is chosen, but Knuth shows it is most efficient to choose the shortest column list, that is the column of the matrix with least 1s. I thought that by choosing the column with most 1s should solve the maximum set packing problem, with of course the obvious changes required to check for a solution.

Simple problems seemed to be solved correctly with this alteration. The tripod packing data could be stored in an $n^3 \times n^3$ matrix, each row representing a tripod position, and each column a position in the packing. The results indicated that this did also solve the tripod packing problem correctly, faster than the naive method, but nowhere near as fast as the best algorithm. For this reason, I did not reconsider the problem until some time later.

When I did reconsider the problem, I realised that this approach may not work correctly if none of the rows in the maximum set had a 1 in the column with the most 1s. As an example, to show this situation can exist I constructed the matrix

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

and tested this with the adapted algorithm. Surely enough, the algorithm failed to find the maximum set, that is the first two rows of the matrix.

Tripod Optimisations

In an attempt to gain further results, I tried to surmise possible simplifications. In particular I had the idea of the balanced tripod packings, with the associated conjecture. This initially looked promising, and seemed to give a reasonable improvement in speed, but I was unable (and still am unable) to prove my conjecture.

It was over Christmas that I developed the most significant aspect of the project, the idea of using upper bounds, and the reduced projection method[1] to find them. For the first time I was able to show that $f(7) = 19$, and count the 10,068 optimal packings for a cube of size 6. This latter result being so large compared to the number of optimal packings for other sizes proved useful for checking the validity of my algorithm. When introducing a new heuristic that does not cause the algorithm to *miss* optimal packings, it gave a certain degree of confidence if all the solutions are correctly found.

As for the optimal amount of pre-processing, this has always been experimental, and has increased as improvements have been made to the algorithm.

Using my balanced packing conjecture, I ran the algorithm for a cube of size 8, requiring 3 tripods per plane in an attempt to find a packing of 24 tripods.[2] Because of the tight bound specified, the algorithm ran in about half an hour, showing that there were no balanced packings with 24 tripods, so implying 23 is optimal if the conjecture is true. It was some time later before I was finally able to check that 23 was indeed optimal.

If we are only interested in finding a single solution, then a good upper bound method should find a single solution. In the case $n = 6$ my algorithm found 9 solutions, the final 8 to be found were the packings of interest. By studying one of these, I was able to establish the worst case situation, and almost immediately develop the lines method for upper bounds. The algorithm now appeared to find a single solution given the correct heuristics. Using these heuristics, I did not find any example where more than one optimal solution was found, although of course this is not exactly conclusive evidence.

[1] I referred to this as the pre-processing method at the presentation

[2] Note that if there was a packing with more than 24 tripods, then there must exist (a non-optimal) packing with 24 tripods.

Now that I had developed the lines method for upper bounds, I was able to verify for the first time that using $n = 6$ gives rise to 10,068 optimal packings without the use of the reduced projection heuristic. For larger values of n ($n \geq 6$) the reduced projection method appears to be the better of the two methods, provided we choose to do the optimal amount of pre-processing. There is of course a lot of overlap between the two methods, where both upper bounds indicate that we should backtrack, but there is certainly enough scope to make both methods worthwhile.

The reduced projection method is the better of the two bounding methods, and immediately reduced the tree size to less than 5% of the previous best for $n = 5$.

With further study into worst case examples, I combined the two methods, which appeared to reduce the running time by about a half, provided we only do this when moving to a new plane.

With all these improvements to the algorithm, I finally checked that $f(8) = 23$. At first, I set the initial lower bound to be 23, so concentrating the search on packings with 24 or more tripods. Two hours later, the algorithm was completed, and no packings were found, which implied that 23 was optimal. Eventually, as suggested by Mike Paterson at the presentation, I ran the complete algorithm in 9 hours, which successfully found a packing of size 23.

On the validity of the algorithm, I have now run the algorithm several times with different heuristics on the $n = 6$ case, and verified that each one finds all the optimal solutions. This is undocumented anywhere, and indeed it was only previously unknown that $f(6) \geq 14$.

To get an estimate as to the difficulty of other problems sizes, I had hoped to use the Monte Carlo estimate procedure as described by Neapolitan and Naimipour [5]. This requires us to choose a random path from the root node of the tree until we reach a leaf node. Unfortunately, I was unable to use this method due to the problem of choosing a random path. Our search for the tripod packing problem uses a binary tree, where at each node we choose to place or not to place a tripod. However, the branches leading to optimal packings for all but trivial cube sizes, require that we choose not to place a tripod far more often than to place a tripod. However, the ratio of these does depend very much on the size of cube we are packing, so I was unable to find a good random path. Consequently, my attempts at an algorithm greatly underestimated the size of the search tree.

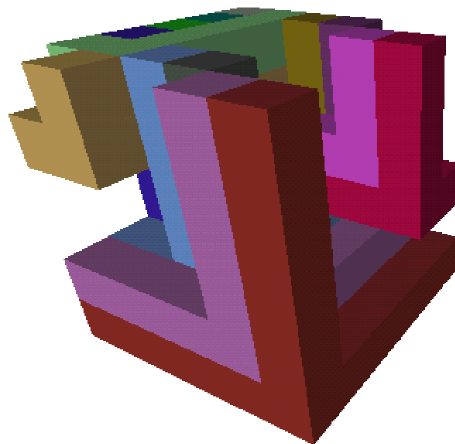

Figure 4.1: VRML model of an optimal packing for a cube of size 6. The tripods have been *trimmed* to fit inside the cube.

Visualising Tripod Packings

It is very hard to visualise the tripod packings, so at the suggestion of my supervisor I decided to try building some models. Not struck on the idea of using cardboard and scissors, I decided it would be appropriate to build some computer models. The idea of using VRML came from the *Automata and Formal Language* assignment in the second year, and the models worked out well. I added an option to my code to output the VRML code, either with the tripods as in the original definition, or *trimmed* to fit in the cuboid as in Figure 4.1.

With VRML and a good viewer, we can pick up and rotate the tripod packing, thus enabling us to view the packing from any angle. I did not build the VRML models until a late stage of the project, so its use for me was limited, but perhaps it could help to find new constructions or properties of packings.

Chapter 5

Conclusions

5.1 Conclusions

The project has proved to be very successful, both technically and in the project management side. I only failed to meet part of one of my original objectives; the study of the polyominoes and tetrasticks problems. I feel I made the right decision to leave these two problems since it gave me enough time to study in depth the tripod packing problem. In my original timetable, I had allocated myself two weeks for each problem, which in hindsight would probably not be enough. Since Knuth had studied these two problems in his paper, the amount of original work I could have done in the time available would be substantially less.

My results from the n-queens problem agrees with Knuth's claims that it can halve the time required for the algorithm, although the results from the exact cover version were less clear-cut. For the problem sizes I tried, my implementation of the dancing links version of Dijkstra's algorithm outperformed the exact cover method, but the complexity of the two meant that the exact cover should perform best for large enough problem size.

I successfully completed all the other objectives. Learning C++ was straightforward, and an easy transition from Java. I still believe it was the best language to use for implementation as it is powerful and reasonably easy to use. C++ is particularly strong with pointer manipulation, which is a necessary trait for such a linked list orientated algorithm.

As for the Monte Carlo method, this was not at all useful for the tripod packing problem, but may have been more useful had I chosen to study a dif-

ferent problem. I found the Monte Carlo method by accident whilst researching the n-queens problem, and long before attempting the tripod packing problem. For this reason, I do not consider being unable to apply the method as a failure, rather that a potential tool was ultimately ineffectual.

The most interesting and most successful part of the project was the work on the tripod packing problem. Optimal packings for the tripod packing problem were only known up to $n = 5$ by Stein, so considering the astronomical complexity of the problem I consider it an achievement to have found optimal packings for cubes size up to 8.

I have successfully developed an algorithm that is far more efficient than the naive approach. The dancing links method was able improve the naive algorithm, but the introduction of the upper bound methods made a far bigger impact on the efficiency of the algorithm. This highlights that we cannot rely solely on dancing links, but must look elsewhere to further improve an algorithm.

For the practice problem I assume that I have calculated more values than was previously known, but have not seen these values documented elsewhere. Since the same algorithm solves the practice problem as the main problem, so further improvements to the algorithm could affect both problems.

The results themselves are unfortunately not as significant as I had hoped, since I was unable to find new lower bounds, but have shown that known lower bounds are indeed optimal. The same can be said for the practice problem, where I was unable to improve upon any known bounds, although I almost certainly considered larger packings than those considered previously.

As for improving the dancing links method, my only idea has been to try deleting a section of the linked list rather than individual nodes. Although I did not use this for the backtracking part of any algorithm, it did simplify some of the preparation work required to set up the list for the tripod packing problem.

The work relating to the maximum set packing problem is the only area in which I feel disappointed. However, because it quickly became apparent the method did not work efficiently, I put the idea aside and did not consider that the algorithm could fail. The time I had spent on the idea was very short, since I had already implemented Knuth's exact cover algorithm; the adaptation for the (erroneous) maximum set packing algorithm was simple. I think, had the results of this algorithm been promising, I would have noticed much sooner that the algorithm could fail. The approach I should have taken was to check the algorithm worked properly before using it to solve the tripod packing problem. However, I think this was a very minor misjudgment, particularly in the context

of an otherwise very successful project.

My original specification was deliberately loose due to the research nature of the project, as it would be impossible to keep to a timetable. This has allowed me to spend most of the time on the tripod packing problem, which I think is justified by the successful results

Some of the techniques developed in this project could be applied to other problems. The definition of a tripod could be extended into higher dimensions, and a similar algorithm could be developed to pack such tripods. Alternatively, returning to lower dimensions, we could use a similar upper bound method to try packing other geometrical shapes.

The project has been a demanding, but rewarding course. To complete the project I have had to develop and learn new skills; technical; communication and project management. It was hard at times, particularly when I went a week or two without improving the algorithm. On the other hand, there were times when, in the course of a few days I could make huge savings in the running time of the algorithm.

5.2 Further Work

Throughout the project I have concentrated on the practical side of the problem, but there are still many unanswered theoretical questions posed by Stein. One question asks the value of $\sup \frac{\log f(n)}{\log n}$. The best lower bound found to date is for a cube of size 255, found by means of a construction rather than the implementation of an algorithm. Perhaps other constructions could improve upon this bound, or this construction could be adapted to work for other sized cubes. It is extremely unlikely that any algorithm will be able to compute optimal packings for such large cubes in the foreseeable future, so perhaps the next development will come from such an approach.

Returning to the algorithm, I am sure that there are still further improvements possible. There are a large number of optimisations and symmetries to the problem, and I continued to make progress in this area until I ran out of time, so I suspect there are still things to be found. An improved or new method of obtaining upper bounds could prove very profitable, and in particular a way of obtaining a bound early on in the search tree would be very worthwhile.

Another possibility would be to implement the algorithm on a parallel system. Different processors could evaluate different branches of the search tree,

so I think this algorithm would run well in this way. However, the complexity of the problem is such that even on a fair sized parallel system, without more improvements to the algorithm we would be limited as to how many results we could obtain. My best algorithm runs far faster on a single processor (for a reasonable problem size) than the naive algorithm could on a feasible parallel system.

An alternative way forward would be to make assumptions about the structure of optimal packings. Unless we could prove these assumptions, we would only be able to find lower bounds for different problem sizes, however this could still produce some interesting results.

The idea of tripod packing models could also be taken further, perhaps allowing the user to remove tripods from the display to gain a better understanding of the structure of the construction. This may require a more powerful modelling language than VRML, but should be possible in some other language.

Returning to the idea of dancing links, Knuth lists some other problems that can be helped with the use of dancing links, and there are probably many more. The performance increase may vary, so it could be interesting to see which type of problem is improved most by dancing links.

Acknowledgement

I would like to thank my project supervisor, Alexandre Tiskin without whose help this project would not have been possible.

Bibliography

[1] Gilles Brassard and Paul Bratley. *Fundamentals of Algorithmics*. Prentice Hall, 1996.

[2] David Gale. *Tracking the Automatic Ant, and Other Mathematical Explorations*. Springer, 1998.

[3] Martin Gardner. *More Mathematical Puzzles and Diversions*. Penguin Books, 1980.

[4] Donald Knuth. *Dancing Links*. Stanford University.

[5] Richard Neapolitan and Kumarss Naimipour. *Foundations of Algorithms, Using C++ Pseudocode* (second edition). Jones and Bartlett, 1998.

[6] Rok Sosic. *N-Queens, Constraint Satisfaction, Local Search.* http://www.cit.gu.edu.au/~sosic/nqueens.html

[7] Sherman Stein and Sándor Szabó. *Algebra and Tiling: Homomorphisms in the Service of Geometry*. The Mathematical Association of America, 1994.

Appendix A

N-Queens Results

The data relating to the n-queens problem is listed in table A. The array method corresponds to the classical version of Dijkstra's algorithm, the list method corresponds to the linked list version and the exact cover method is Knuth's exact cover algorithm from [4] using the pipe organ order. Knuth refers to *nodes* and *updates* whereas we refer to *placed* and *checks* respectively. Note that the times given are based on my Pentium 90 PC, so may not be representative of other systems.

n	Solutions	Array Method	List Method	Exact Cover Method
1	1	1 placed 1 check (0 secs)	1 placed 1 check (0 secs)	2 placed 3 checks (0 secs)
2	0	2 placed 6 checks (0 secs)	2 placed 4 checks (0 secs)	3 placed 19 checks (0 secs)
3	0	5 placed 18 checks (0 secs)	5 placed 11 checks (0 secs)	4 placed 56 checks (0 secs)
4	2	16 placed 60 checks (0 secs)	16 placed 32 checks (0 secs)	13 placed 183 checks (0 secs)
5	10	53 placed 220 checks (0 secs)	53 placed 101 checks (0 secs)	46 placed 572 checks (0 secs)
6	4	152 placed 894 checks (0 secs)	152 placed 356 checks (0 secs)	93 placed 1,497 checks (0 secs)
7	40	551 placed 3,584 checks (0 secs)	551 placed 1,345 checks (0 secs)	334 placed 5,066 checks (0 secs)
8	92	2,056 placed 15,720 checks (0 secs)	2,056 placed 5,508 checks (0 secs)	1,049 placed 16,680 checks (0 secs)
9	352	8,393 placed 18 checks (0 secs)	8,393 placed 24,011 checks (0 secs)	3,440 placed 54,818 checks (1 sec)
10	724	35,538 placed 348,150 checks (0 secs)	35,538 placed 110,004 checks (0 secs)	11,578 placed 198,264 checks (1 sec)
11	2,680	166,925 placed 1,806.705 checks (0 secs)	166,925 placed 546,397 squares (0 secs)	45,393 placed 783,140 checks (1 sec)
12	14,200	856,188 placed 10,103,868 checks (3 secs)	856,188 placed 2,915,740 checks (2 secs)	211,716 placed 3,594,752 checks (3 secs)
13	73,712	4,674,899 placed 59,815,314 checks (20 secs)	4,674,899 placed 16,469,569 checks (10 secs)	1,046,319 placed 17,463,157 checks (17 secs)
14	365,596	27,358,552 placed 377,901,398 checks (135 secs)	27,358,552 placed 99,280,504 checks (60 secs)	5,474,542 placed 91,497,926 checks (90 secs)
15	2,279,184	171,129,071 placed 2,532,748,320 checks (870 secs)	171,129,071 placed 636,264,861 checks (378 secs)	31,214,675 placed 513,013,152 checks (505 secs)
16	14,772,512	1,141,190,302 placed 842,815,472 checks (6,827 secs)	1,141,190,302 placed 35,907,964 checks (2,539 secs)	193,032,021 placed 3,134,588,055 checks (3,121 secs)

Table A.1: Optimal solutions for n-queens problem calculated with algorithm.

Appendix B

Selected Pseudocode

B.1 Dancing Links Naive Algorithm

This section contains some pseudocode for the dancing links naive algorithm.

We will use the following global variables:

head	Head of the linked list.
optimal	Size of optimal solution (initially 0).
projection[x][y]	Node corresponding to the cell (x, y) in projection.
projection_count[x][y]	Times projection$[x][y]$ should have been deleted.
xval[z]	Stack for the x coordinates of tripods.

The following functions/procedures are also used:

add(node)	Add a deleted node using (2.2).
delete(node)	Delete a node using (2.1).
next(node)	Return next node in linked list.
stack.push(value)	Push value onto stack.
pop(stack)	Pop element off stack.
top(stack)	Return top element of stack without removing.
x(node)	Return x coordinate in projection of node.
y(node)	Return y coordinate in projection of node.

Each stack should initially have the value -1 on top, so the stack never becomes empty and we save ourselves some checking.

To place a tripod, we must carefully remove the necessary elements from the linked list, and store the x coordinate on the stack.

```
▷ Place tripod at (x, y, z)
PLACE(x,y,z)

    ▷ Vertical tripod arm
    for a ← y+1 to m-1
        if (projection_count[x][a]=0)
            delete(projection[x][a])
        projection_count[x][a] ← projection_count[x][a]+1

    ▷ Horizontal tripod arm
    for a← l-1 to x
        if (projection_count[a][y]=0)
            delete(projection[a][y])
        projection_count[a][y] ← projection_count[a][y]+1

    ▷ Push element on stack
     xval[z].push(x)
```

We must remove the tripod projection in the reverse order to which it was placed. Otherwise the routine is similar to the place function.

```
▷ Remove tripod at (x, y, z)
REMOVE(x,y,z)

    ▷ Vertical tripod arm
    for a← x to l-1
        projection_count[a][y] ← projection_count[a][y]-1
        if (projection_count[a][y]=0)
            add(projection[a][y])

    ▷ Horizontal tripod arm
    for a← m-1 to y+1
        projection_count[x][a] ← projection_count[x][a]-1
        if (projection_count[x][a]=0)
            add(projection[x][a])

    ▷ Pop element off stack and discard
     pop(xval[z])
```

Checking a tripod can be placed is simple, since the node must be in the linked list, only the stack operation is required.

```
▷ Check if tripod can be placed at (x, y, z)
function CHECK(x,y,z)

    if (top(xval[z])<x)
        return false
    else
        return true
```

We need to check whether we can continue with the packing, or whether we should backtrack. We can add to this function later to introduce some other heuristics.

```
▷ Check if we can continue packing.
function CONTINUE(x,y,z)

    ▷ Continue if not at end of list
    if (projection[x][y] not end of list)
        return true

    ▷ Continue if list is nonempty, and not in final plane.
    if (list nonempty and z<n-1)
        return true

    return false
```

We need to obtain the next position in the cube to try placing a tripod, which may involve moving to a new plane.

```
▷ Obtain next coordinates in packing.
function NEXT(x,y,z)

    if (projection[x][y] not end of list)
        return (x(next(projection[x][y])),y(next(projection[x][y])),z)
    else
        return (x(head),y(head),z+1)
```

Finally, we have the main recursive function. To commence, we call TRIPODS(0,0,0,0). The size of optimal packing is stored in the global variable *optimal*.

```
▷ Main Recursive Routine
TRIPODS(x,y,z,count)

    ▷ Place a tripod at (x, y, z) if possible.
    if (CHECK(x,y,z))
        PLACE(x,y,z)

        ▷ Check if packing is optimal.
        if (count+1>optimum)
            optimum ← count+1

        ▷ Continue with packing having placed a tripod.
        if (CONTINUE(x,y,z))
            (x1,y1,z1) ← NEXT(x,y,z)
            TRIPODS(x1,y1,z1,count+1)

        REMOVE(x,y,z)

    ▷ Continue with packing without placing tripod.
    if (CONTINUE(x,y,z))
        (x1,y1,z1) ← NEXT(x,y,z)
        TRIPODS(x1,y1,z1,count)
```

B.2 Adding to the Basic Algorithm

We will now look to introduce the reduced projection and the lines method to obtain the upper bounds, to be used when moving to a new plane.

Let us begin by introducing some new global variables:

col_cells[m]	Number of cells in column m of projection (initially l).
cols	Number of columns removed from projection (initially m).
f[l][m][n]	Optimum values of tripod packing previously obtained.
line[k]	Number of empty cells for lines method (initially line[k] $= l + m - 2k + 1$).
row_cells[l]	Number of cells in row l of projection (initially m).
rows	Number of rows in projection (initially l).

Note that for values of f[l][m][n] not calculated, these should be given upper bounds (e.g. let $f[l][m][n] = lmn$), so as to avoid unnecessary checking.

We need to keep track of these variables throughout the algorithm, so we need to add to the place and remove tripod procedures to do this. The place procedure becomes:

```
▷ Place tripod at (x, y, z)
PLACE(x,y,z)

    ▷ Vertical tripod arm
    for a← y+1 to m-1
        if (projection_count[x][a]=0)
            delete(projection[x][a])
            row_cells[a] ← row_cells[a]-1
            if (row_cells[a] = 0)
                rows ← rows -1

            col_cells[x] ← col_cells[x]-1
            if (col_cells[x]=0)
                cols ← cols-1

            line[min(a,x)] ← line[min(a,x)]-1

        projection_count[x][a] ← projection_count[x][a]+1

    ▷ Horizontal tripod arm
    for a← l-1 to x
        if (projection_count[a][y]=0)
            delete(projection[a][y])
            row_cells[y] ← row_cells[y]-1
            if (row_cells[y] = 0)
                rows ← rows -1

            col_cells[a] ← col_cells[a]-1
            if (col_cells[a]=0)
                cols ← cols-1

            line[min(a,y)] ← line[min(a,y)]-1

        projection_count[a][y] ← projection_count[a][y]+1

    ▷ Push element on stack
     xval[z].push(x)
```

The reader should be able to see how the REMOVE procedure needs changing. We can now change the CONTINUE function to use the upper bounds. We will assume that we want a single solution, so will use (3.1).

```
▷ Check if we can continue packing.
function CONTINUE(x,y,z,count)

    ▷ Continue if not at end of list
    if (projection[x][y] not end of list)
        return true

    ▷ Calculate number of planes to be filled.
    p←n-z

    ▷ Reduced Projection upper bound.
    if (count + f[rows][cols][p]<optimum)
        return false

    ▷ Lines upper bound.
    total←0
    for a←0 to min(l,m)
        total←total+min(lines[a],p])
    if (count+total<optimum)
        return false

    return true
```

In particular, note that should we fail to produce an upper bound low enough to force us to backtrack, then we are guaranteed to be able to place more tripods.

Finally, since the CONTINUE function needs the extra parameter for the number of tripods currently placed, so we should add this extra parameter to any calls from the TRIPOD procedure.

There are many more optimisations that can be made to the algorithm, as discussed in Chapter 3. We can check the upper bounds after placing each tripod, use the combined upper bound method, reflections and many more properties. These are required to obtain the full efficient algorithm, with results in Appendix C, but the ideas shown here are the most important. The discussions from Chapter 3 should be sufficient to implement the full algorithm.

Appendix C

Tripod Packing Results

This section lists results obtained using the algorithm. Table C.1 lists the values I have been able to obtain for the function $f(n)$, along with the number of solutions. Note that the number of optimal solutions does not consider symmetries, so many solutions are geometrically the same.

Table C.2 illustrates the huge reductions in tree size made compared to the naive method, but ultimately the complexity is such that these improvements only enable us to find a few more values for $f(n)$. The time saving for these algorithms is not quite as impressive as the reductions in the tree size as more work is done at each stage, however the savings are still very significant.

Table C.3 displays values obtained for $f(x,x,y)$ calculated using the algorithm. This may help towards the "practice problem" proposed by Stein in [2]. Note that $c(y) = \sup \frac{f(x,x,y)}{x}$ using Stein's notation.

n	$f(n)$	Number of Solutions
1	1	1
2	2	10
3	5	2
4	8	14
5	11	56
6	14	10,068
7	19	8
8	23	?

Table C.1: Optimal solutions for initial problem calculated with algorithm.

n	Naive	Naive & DL	All Solutions	One solution
3	5,034	1,638	205	74
4	979,887	221,962	4,443	475
5	502,721,073	83,195,405	49,252	5,089
6			23,436,118	762,305
7			1,445,488,976	49,178,075
8				14,140,076,400

Table C.2: Size of search tree for each algorithm.

$x \backslash y$	1	2	3	4	5	6	7	8	9	10
1	**1**	1	1	1	1	1	1	1	1	1
2	2	**2**	3	4	4	4	4	4	4	4
3	3	4	**5**	5	6	7	8	8	9	9
4	4	5	6	**8**	9	9	10	10	11	12
5	5	6	8	10	**11**	12	13	14	15	15
6	6	8	10	12	13	**14**	16	17	18	19
7	7	9	11	14	16	17	**19**	20	21	22
8	8	10	13	16	18	20	21	**23**	(24)	(24)
9	9	12	15	18	20	22	(24)	(25)	**(28)**	(28)
10	10	13	16	20	22	(24)	(27)	(28)	(30)	**(32)**
$\sup \frac{f(x,x,y)}{x}$	1	$\frac{4}{3}$	$\frac{5}{3}$	2	$\geq \frac{16}{7}$	$\geq \frac{20}{8}$	$\geq \frac{19}{7}$	$\geq \frac{23}{8}$	$\geq \frac{28}{9}$	$\geq \frac{32}{10}$

Table C.3: Values calculated for $f(x, x, y)$. The values in brackets are lower bounds for the solution.

Appendix D

Tripod Packing Code

The following C++ code is my implementation of the algorithm outlined in Chapter 3, used to obtain the given results.

```
/*
/ Tripods Problem Solver
/ ----------------------
/
/ Work from one side of cube to the other, using dancing links method to
/ store a list of possible tripod positions.
/ Placing a tripod at a position in one plane prevents tripods being
/ placed at some positions in later planes, so we save some checking.
/
/ Tripod data will be stored in two ways:
/ (1) In matrix form. This means output is easy, and is also used for
/     backchecking to see if a tripod could be placed in an earlier plane.
/ (2) In an array of stacks. Each stack represents the 'highest' position
/     of a tripod in one of n parallel planes.Due to the order of placing
/     the tripods this is all the information we need stored.
/
/ The linked list is stored within an array to allow easy access to the
/ elements. The list starts initially with n^2 elements (plus head and tail
/ nodes). Nodes of the list corresponding to points that tripods cannot be
/ placed are deleted as a tripod is placed. However, we must be careful not
/ to delete a node twice as arms of one tripod can be in the same position
/ (but different plane) to those of another. To get around this we will use
/ an array of n^2 elements, indicating how many times we would have deleted
/ the node, and then replace the node as the last overlapping tripod is
```

```
/ removed. (The best way to picture this is to consider a 2d square
/ representing the cube).
/
/ The program will use (x, y, z) coordinates for the position of the
/ tripods, working along x, then y, then z, with the legs of the tripods
/ pointing towards (x, 1, z),(x, y, n) and (n, y, z).
/
/ Method of Upper bounds
/ ----------------------
/ There are two main methods to obtain an upper bound:
/ (1) Do preprocessing to get upper bound for tripods in remaining planes.
/     This is always done (unless we use -x) but we can specify (-a) only
/     to place tripods while we can beat our best solution rather than
/      draw with it.
/ (2) (Diagonal bounds) Using particular lines on a plane where at most
/     one tripod can be placed per plane (-d).
/
/ These two methods can be combined to some extent, but it is slow so
/ only combine when changing plane, not placing tripod.
/
/
/ Here is a list of other optimizations & symmetries that can be
/ exploited by the algorithm:
/ (i)   Each plane must have at least one tripod in. Since we have a
/       lower bound of n tripods for cube n, if we have an empty plane,
/       then at least one other plane has more than one tripod which
/       can be 'shifted' to the empty plane, thus will not lead to an
/       better solution. (-e)
/ (ii)  We must have tripods at (n, 1, n) and (1, n, 1) (by symmetry).
/       However both these tripods are placed out of the natural order
/       of placing, so these are special cases that need extra coding. (-c)
/ (iii) Do not place tripod if it could have been placed in an earlier
/       plane. We simply check the position in previous planes to see
/       if we could place a tripod. If we can, then we are not going to
/       get a better solution than if we had placed the tripod in the
/       earlier plane, so simply don't bother to place tripod. (-p)
/ (iv)  Only try to cover to place first tripod in one (triangular) 1/2
/       of initial plane (i.e. reflection in reverse diagonal)(-r)
/ (v)   Insist on a maximum & minimum number of tripods in each plane
/       - based on my conjecture, but this will give a big time saving.
*/
```

```
#include <time.h>
#include <iostream.h>
#include <stdio.h>
#include <stdlib.h>


 /*
 / Basic node for linked list
 /
 / prev & next point to previous & next nodes in list
 / px, py correspond to the x & y elements of array
 /
 / When using a linked list we must have a head node that points to the
 / first element in the list. This is a normal node, but not affected by
 / any list operations. Some of these operations are used a lot in the
 / tripods program, so error checking is not included here for reasons
 / of speed.
*/
class node
{
  public:
  node* prev;             // previous node pointer
  node* next;             // next node pointer
  int px;                 // equivalent x coordinate
  int py;                 // equivalent y coordinate

   /*
   / Constructor for node
  */
  node(int x, int y)
  {
    prev = 0;
    next = 0;
    px = x;
    py = y;
  }


   /*
   / Add element to head of a list
   /
   / Must check to see if list is empty. Note that the first element in
```

```
 / the list points back to the head node with its 'prev' pointer. This
 / reduces the need for error checking in other routines (particularly
 / add & delete nodes).
*/
void addstart(node* n)
{
  if (this->next==0)      //empty list
  {
    this->next = n;
    n->prev = this;
  }
  else
  {
    n->next = this->next;
    n->prev = this;
    (this->next)->prev = n;
    this->next = n;
  }
}


 /*
 / Add deleted node
*/
void add(node* n)
{
  (n->next)->prev = n;
  (n->prev)->next = n;
}


 /*
 / Add a whole section of deleted nodes
*/
void add_section(node* start, node* end)
{
  (end->next)->prev = end;
  (start->prev)->next = start;
}

 /*
 / Delete node (temporarily)
```

```
  */
  void del(node* n)
  {
    (n->next)->prev = n->prev;
    (n->prev)->next = n->next;
  }


   /*
   / Delete a whole section of nodes
  */
  void del_section(node* start, node* end)
  {
    (end->next)->prev = start->prev;
    (start->prev)->next = end->next;
  }
};


 /*
 / Integer Stack - Note this is a simple stack routine & will not trap
 / overflow/underflow errors since this is not needed in the context of
 / the algorithm (and would slow the program down)
*/
class stack
{
  private:
  int st[21];       // array to store stack
  int top;          // array pointer to top element of stack

  public:

   /*
   / Constructor for stack.
   /
   / We place the element 0 on to the top of the stack since this is
   / what we want returned from the get routine if the stack is empty,
   / so saves an 'if' statement.
  */
  stack()
  {
    top = 0;
```

```
    st[0] = 0;
  }

   /*
   / Push integer onto top of stack
  */
  void push(int i)
  {
    st[++top] = i;
  }

   /*
   / Remove & return element from the top of the stack
  */
  int pop()
  {
    return st[top--];
  }

   /*
   / Return element from top of stack without removing. This is a less
   / orthodox stack operation, however it is faster than popping &
   / pushing an element which we would otherwise need to do.
  */
  int get()
  {
    return st[top];
  }

   /*
   / Return n'th from top elt in stack without removing.(n=1 => top elt)
  */
  int get(int n)
  {
    return st[top-n+1];
  }
};


 /*
 / Return minimum of two integers
*/
```

```
int min(int a, int b)
{
  if (a>b)
    return b;
  else
    return a;
}


 /*
 / Return maximum of two integers
*/
int max(int a, int b)
{
  if (a<b)
    return b;
  else
    return a;
}


 /*
 / Return middle of three integers
*/
int mid(int a, int b, int c)
{
  if ((a<=b || a<=c) && (a>=b || a>=c))
    return a;
  if ((b<=a || b<=c) && (b>=a || b>=c))
    return b;
  return c;
}


// Declare global variables
// Note we use a lot of global variables as there are too many settings
// to pass as local variables

// Main general variables for algorithm
int l, m, n;              // Size of cube
node* base[21][21];       // Array to contain the linked list - remove
                          // elements where no tripods can be placed
```

```
int base_count[21][21];   // Keep a count of tripods for base, so do not
                          // try to remove node twice
int matrix[21][21];       // Store values for matrix output
int vrml_matrix[21][21];  // Store matrix copy for vrml output
int xycount[21];          // Keep count of tripods placed in each (x,y)
                          // plane -can backtrack earlier if plane empty
stack hix[21];            // Highest tripod in x coord of tripod in plane
node *list;               // Store head of linked list
node *endbuf;             // Buffer for end of linked list
int maxt = 0;             // Size of maximal solutions found
int output;               // Print output ?

// Upper bounds variables
int f[21][21][21];                 // Preprocess & store sizes of optimal solns
int hblockcount=0, vblockcount = 0;// Count # of tripods at bottom of cuboid
                                   // - reduces max # of tripods we can place.
int vcount[21], hcount[21];        // Count # of cells blocked in row/col in planes
int diagonal_count[21];            // # of tripods on diagonal

// Command line parameters
int complete = 0;           // Do not backtrack until cannot equal best soln (-a)
int display_sols = 0;       // Display all optimal solutions (-l)
int altnot = 0;             // Alternative notation for matrix output (-n)
int maxmin = 0;             // Max & min tripods in plane heuristic (-m)
int reflect = 0;            // Can we reflect in diagonal? (-r)
int push_forward = 1;       // push tripods forward? (-p)
int corners = 1;            // Tripods in corners (-c)
int diag_bound = 1;         // Diagonal method for upper bound
int display_report = 0;     // Display progress reports periodically?
int preprocessing = 1;      // Do preprocessing?
int lower_bound = 0;        // Lower bound for solution
int mintp = 0, maxtp;       // Minimum & maximum tripods in the plane
int vrml = 0;               // Send VRML file to standard output

// Stats info
long unsigned int node_count = 0;  // Number of nodes in tree
long unsigned int place_count = 0; // Number of tripods placed
int max_sol_count = 0;             // Count number of optimal solutions
int rep_count = 0;                 // Node counter for progress reports

 /*
 / Place a tripod
```

```
 /
 / When placing a tripod, we will put the data in in two ways.
 /
 / 1) In array of stacks storing highest x position
 /
 / 2) In matrix form in variable matrix[x][z] - only needed for
 /    displaying solns.
 /
 / Must also remove elements from linked list
*/
void place(int x, int y, int z)
{
  // Vertical leg
  for (int a = y+1; a<=m; a++)
    if (!base_count[x][a]++)
    {
      list->del(base[x][a]);

      if (++vcount[x]==m)
        vblockcount++;

      if (++hcount[a]==l)
        hblockcount++;

      diagonal_count[min(a,x)]--;
    }

  // Be careful to delete current node last - this way we can continue
  // to traverse list without jumping to a new node.
  for (int a = l; a>=x; a--)
    if (!base_count[a][y]++)
    {
      list->del(base[a][y]);

      if (++vcount[a]==m)
        vblockcount++;

      if (++hcount[y]==l)
        hblockcount++;

      diagonal_count[min(a,y)]--;
    }
```

```
  matrix[x][z] = y;
  xycount[z]++;
  hix[z].push(x);
}


 /*
 / Temporarily place a tripod, used to help get upper bound so save
 / time & update only necessary data
*/
void virtual_place(int x, int y)
{
  // Vertical leg
  for (int a = y+1; a<=m; a++)
    if (!base_count[x][a])
    {
      if (++vcount[x]==m)
        vblockcount++;

      if (++hcount[a]==l)
        hblockcount++;
    }

  for (int a = l; a>=x; a--)
    if (!base_count[a][y])
    {
      if (++vcount[a]==m)
        vblockcount++;

      if (++hcount[y]==l)
        hblockcount++;
    }
}


 /*
 / Remove a tripod
*/
void remove(int x, int y, int z)
{
  // Must add elements in reverse order of deletions in place procedure
```

```
  for (int a = x; a<=l; a++)
    if (!--base_count[a][y])
    {
      list->add(base[a][y]);

      if (vcount[a]--==m)
        vblockcount--;

      if (hcount[y]--==l)
        hblockcount--;

      diagonal_count[min(a,y)]++;
    }

  // Vertical leg
  for (int a = m; a>=y+1; a--)
    if (!--base_count[x][a])
    {
      list->add(base[x][a]);

      if (vcount[x]--==m)
        vblockcount--;

      if (hcount[a]--==l)
        hblockcount--;

      diagonal_count[min(a,x)]++;
  }

  matrix[x][z] = 0;
  xycount[z]--;
  hix[z].pop();
  place_count++;
}


 /*
 / Remove a 'virtually placed' tripod
*/
void virtual_remove(int x, int y)
{
```

```
  for (int a = x; a<=l; a++)
    if (!base_count[a][y])
    {
      if (vcount[a]--==m)
        vblockcount--;

      if (hcount[y]--==l)
        hblockcount--;
    }

  // Vertical leg
  for (int a = m; a>=y+1; a--)
    if (!base_count[x][a])
    {
      if (vcount[x]--==m)
        vblockcount--;

      if (hcount[a]--==l)
        hblockcount--;
    }
}


 /*
 / Check to see if a tripod can (or should) be placed in a cell.
 / Note that using some reflection & push forward heuristics return 0
 / when a tripod could be placed but is better not to.
 /
 / Returns 1 if tripod can be placed, otherwise 0
*/
int check(int x, int y, int z)
{
  // More thorough check for less obvious situations
  if (hix[z].get()> x)
    return 0;

  // Special case needed due to placing a tripod out of order
  if(corners && z==1 && (x==l || y==m) && !(x==l && y==m))
    return 0;

  // Reflect property - return 0 if first tripod attempted to be placed above diagonal
  if(xycount[1]==0 && reflect && z==1 && y>x)
```

```
    return 0;

  // backcheck - return 1 if cannot place tripod in earlier place since we
  //             know by now we can place one in this plane.
  // IMPORTANT NOTE - the conditions to be satisfied before using routine.
  // We must place tripod at (1,n,1), so do not do a backcheck.
  if(push_forward && z>1 && !(x<=2 && y<=2 && corners))
  {
    for(int c = z-1; c>=1; c--)
    {
      if(matrix[x][c]!=0)
        return 1;

      int flag=0;

      for (int b = 1; b<x && !flag; b++)
        if(y<=matrix[b][c])
          flag = 1;

      for (int b = x+1; b<=l && !flag; b++)
        if(y>=matrix[b][c])
          flag = 1;

      if (!flag)
        return 0;
    }
  }
  return 1;
}


 /*
 / Choose the next position to attempt placing a tripod
 /
 / First work along the basic list, once end reached then work along
 / again for next column - until list empty of end or final column
 /
 / Returns 1 if cannot continue otherwise 0
 /           If can continue listptr & z will point to new positions.
*/
int next(int &z, node* &listptr, int count)
{
```

```
if (listptr->next != endbuf)
{
  listptr = listptr->next;
  return 0;
}
else
{
  if (list->next != endbuf && z<n && xycount[z]>=mintp)
  {
    // Check we may still get good solution

    // VERY NEW UPPER BOUND !!!
    int planes = n-z;
    int best = f[l-vblockcount][m-hblockcount][planes]; //(i.e. lowest)

    if(diag_bound)
    {
      int flag = 0;
      int dist = 0;
      int totd = 0;

      for(int a=1; a<=min(l,m) && !((complete && best+count<maxt) ||
      (!complete && best+count <= maxt)); a++)
        if(!flag) // elt in list
        {
          totd = totd + min(diagonal_count[a],planes);
          virtual_place(a,a);
          dist = a;

          if(f[l-vblockcount][m-hblockcount][planes]+totd<=best)
            best = f[l-vblockcount][m-hblockcount][planes]+totd;
          else
            flag = 1;
        }
        else
        {
          flag = 1;
          totd = totd + min(diagonal_count[a],planes);
        }

      best = min(best,totd);
```

```
        for(int a = dist; a>=1; a--)
          virtual_remove(a,a);
      }

      if((complete && best+count<maxt)||(!complete && best+count<=maxt))
        return 1;

      // 2nd tripods...
      if(corners && l>=2 && m>=2 && n>=2)
      {
        // If about to leave 2nd plane, check a tripod in approp. positions
        if(z==2)
        {
          if(!matrix[l-1][2] && !matrix[l-1][1] && !matrix[l][2])
            return 1;

          if(matrix[hix[2].get()][1]<m-2 && matrix[hix[1].get(2)][1]<m-2)
            return 1;
        }

        // And don't continue if can't place tripod in post, near end
        if(z<=n-2 && base_count[1][2] && base_count[2][2] && base_count[2][1])
          return 1; // V. small saving
      }

      // Go to next plane
      z++;
      listptr = list->next;

      // Don't bother placing tripod at (1,i,1)(i<m) since we will
      // place one at (1,m,1) if option set
      if(listptr->px==1 && listptr->py==1 && z<n && corners)
        return next(z, listptr, count);

      return 0;
    }
    else // Time to backtrack
      return 1;
  }
}
```

```
/*
/ Main recursive routine to solve the problem
/
/ Note that a matrix is displayed (if the option is set) when the
/ solution found is as good or better than any known solution. - It
/ would take a lot more resources to display only optimal solutions.
*/
void tri(int z, int count, node* ptr)
{
  // Report how far through algorithm we are if option set
  if(display_report && output)
  {
    // Display report after this number of nodes
    if(rep_count>=500000000)
    {
      rep_count = 0;
      cout <<"PROGRESS REPORT - This is NOT a solution (best = "<<maxt<<")\n";

      // And display current matrix
      for(int b = l; b>=1; b--)
      {
        for(int a = n; a>=1; a--)
          cout << matrix[b][a] << " ";
        cout << "\n";
      }
      cout <<"\n";
    }
    rep_count++;
  }

  // count nodes (for statistical reasons only)
  node_count++;

  int bad = 0;

  // Move to next position in cube
  if (!next(z, ptr, count))
  {
    int x = ptr->px;
    int y = ptr->py;

    // Check if we can place a tripod here, if we can place a tripod
```

```
// & check if optimal solution
if (xycount[z]<maxtp && check(x,y,z))
{
  place(x, y, z);
  int upper_bound = l*m*n;

  // calculate upper bound using diagonal method
  if (diag_bound)
  {
    upper_bound = 0;

    for(int a = 1; a<=min(l,m); a++)
      upper_bound = upper_bound + min(diagonal_count[a],n-z+1);
  }

  // With red. proj. method - take the lower of the two bounds
  upper_bound = min(upper_bound,f[l-vblockcount][m-hblockcount][n-z+1]);

  // Decide whether we should continue along branch or backtrack
  if((complete && upper_bound+count+1 < maxt) ||
    (!complete && upper_bound+count+1 <= maxt))
      bad = 1;

  // Have we found a new solution?
  if (count+1>=maxt)
  {
    int reflected = 0;

    if(count == maxt)
    {
      maxt = count+1;
      max_sol_count = 1;

      if(output)
        cout << "Solution with " << maxt << " tripods found!\n";

      // Make a copy of the optimal matrix for VRML file
      if(vrml)
        for(int a=1;a<=l;a++)
          for(int b=1;b<=n;b++)
            vrml_matrix[a][b] = matrix[a][b];
    }
```

```
else
  max_sol_count++;

// Check if solution is a reflection of another we can find or
// not (but only if we want to calculate the number of solutions)
if(reflect && complete)
{
  int flag = 0;
  reflected = 1;

  for(int a = 1; a<=m && !flag; a++)
  {
    if(matrix[a][1])
    {
      flag = 1;
      if(matrix[a][1]>=a)
        reflected = 0;
      else
        max_sol_count++;
    }
  }
  if(flag==0)
    reflected = 0;
}

// Display matrix
if(output && (display_sols || (max_sol_count-reflected)==1))
{
  if(!altnot)
  {
    for(int b = l; b>=1; b--)
    {
      for(int a = n; a>=1; a--)
        cout << matrix[b][a] << " ";

      if(reflected && display_sols)
      {
      // Display reflection of solution alongside main solution
        cout << "     ";
        for(int a = n; a>=1; a--)
        {
          int disp = 0;
```

```
            for(int c=1; c<=m && !disp; c++)
              if(matrix[c][a]==b)
                disp = c;
              cout << disp << " ";
          }
        }
        cout << "\n";
      }
    }
    else
    {
      for(int b = 1; b<=l; b++)
      {
        for(int a = n; a>=1; a--)
          if(matrix[b][a]==0)
            cout << ".";
          else
            cout << matrix[b][a]-1;

        if(reflected && display_sols)
        {
        // Display reflection of solution alongside main solution
          cout << "     ";
          for(int a = n; a>=1; a--)
          {
            int disp = 0;
            for(int c = 1; c<=m && !disp; c++)
              if(matrix[c][a]==b)
                disp = c;
              if(!disp)
                cout << ".";
              else
                cout << disp-1;
          }
        }
        cout << "\n";
      }
    }
    cout << "\n";
  }
}
// Only continue along branch if we may still make our upper bound
```

```
      if(!bad)
        tri(z, count+1, ptr);

      // And now remove the tripod on backtracking
      remove(x, y, z);
    }

    // Must place tripods in corners if option set
    if(!corners || (!((z==1 && x==l && y==m) || (z==n && x==1 && y==1))))
      tri(z, count, ptr);  // without placing block
  }
}

void display_vrml(int trim)
{
  // Set up colours for tripods
  float cr=0, cg=0, cb=0;

  // Print file header
  cout << "#VRML V1.0 asciifile\n";

  // Print a comment
  cout << "\n# An optimal ";
  if(trim)
    cout << "(trimmed) ";
  cout << "tripod packing for ";
  if (l==m && m==n)
    cout << "cube of size " << n;
  else
    cout << "cuboid of size ("<< l << ","<< m << ","<< n << ")";
  cout << ".\n# This contains " << maxt << " tripods.\n";

  for(int x=1;x<=l;x++)
  {
    for(int z=1;z<=n;z++)
    {
      if(vrml_matrix[x][z])
      {
        int y = vrml_matrix[x][z];

        // Change tripod colours
        cr = cr + 0.89;
```

```
        cg = cg + 0.73;
        cb = cb + 0.39;
        if (cr>1)cr--;
        if (cg>1)cg--;
        if (cb>1)cb--;

        // Main output
        if(!trim)
        {
          cout << "\n# Tripod at (x,y,z) = ("<<x<<","<<y<<","<<z<<")\n";
          cout << "Material  { diffuseColor "<<cr<<" "<<cg<<" "<<cb<<" }\n";
          cout << "Transform { translation "<<l/2.0+x-2<<" "<<y-2<<" "<<z<<"}\n";
          cout << "Cube      { width "<< l+1 <<" height 1 depth 1}\n";
          cout << "Transform { translation "<<-l/2.0<<" "<<m/2.0<<" 0 }\n";
          cout << "Cube      { width 1 height "<<m+1<<" depth 1}\n";
          cout << "Transform { translation 0 "<<-m/2.0<<" "<<-n/2.0<<" }\n";
          cout << "Cube      { width 1 height 1 depth "<<n+1<<" }\n";
          cout << "Transform { translation "<<-x+2<<" "<<-y+2<<" "<<n/2.0-z<<" }\n";
        }
        else
        {
          float xx,yy,zz;
          // length of 3 legs
          xx = l-x+1; //line x to l
          yy = m-y+1; //line y to n
          zz = z;     //line z to 1
          cout << "\n# Tripod at (x,y,z) = ("<<x<<","<<y<<","<<z<<")\n";
          cout << "Material  { diffuseColor "<<cr<<" "<<cg<<" "<<cb<<" }\n";
          cout << "Transform { translation "<<(xx-1)/2+x-2<<" "<<y-2<<" "<< z<<" }\n";
          cout << "Cube      { width "<<xx<<" height 1 depth 1 }\n";
          cout << "Transform { translation "<<-(xx-1)/2<<" "<<(yy-1)/2<< " 0 }\n";
          cout << "Cube      { width 1 height "<<yy<<" depth 1 }\n";
          cout << "Transform { translation 0 "<<-(yy-1)/2<<" "<<-(zz-1)/2<<" }\n";
          cout << "Cube      { width 1 height 1 depth "<<zz<<" }\n";
          cout << "Transform { translation "<<-x+2<<" "<<-y+2<<" "<<(zz-1)/2-z<<" }\n";
        }
      }
    }
  }
}
```

```
/*
/  Main routine to run the algorithm. This section is intended to:
/ (1) Read in the command line inputs & return error if necessary
/ (2) Create the linked list, set up some global variables
/ (3) Deal with any preprocessing that is necessary
/ (4) Start the main algorithm
/ (5) Display the results
*/
int main(int argc, char* argv[])
{
  // Start the timer
  time_t t1 = time(NULL);

  // First 2 parameters read by default
  int par = 2;

  int trim = 0;

  // Read command line entry if possible
  if (argc<2)
  {  // Error/help message & quit
    cout <<"Usage: " << argv[0] << " d1 [d2] [d3] [options]\n";
    cout <<"   d1         Size of cube (between 1 & 20)\n";
    cout <<"   d1 d2      Dimensions of cuboid as in practice problem\n";
    cout <<"   d1 d2 d3   Dimensions of cuboid\n";
    cout <<"Options\n";
    cout <<"  -a          Start to backtrack when we can no longer beat our optimal";
    cout <<"              solution.\n";
    cout <<"              (rather than when we can no longer equal or beat it).\n";
    cout <<"  -b bound    Lower bound for solution (Does not attempt to find solutions";
    cout <<"              smaller than this bound). If you use 'g' and not -a parameter,";
    cout <<"              the algorithm will find an optimal solution, then re-run the";
    cout <<"              algorithm to get the other solutions.\n";
    cout <<"  -c          Place tripods in corners.\n";
    cout <<"  -d          Diagonal method of upper bounds.\n";
    cout <<"  -e          Do not allow empty planes.\n";
    cout <<"  -g          Display progress reports occasionally.\n";
    cout <<"  -l          List all possible solutions when found.\n";
    cout <<"  -m min max  Minimum & Maximum tripods per plane.\n";
    cout <<"  -n          Alternative Notation.\n";
    cout <<"  -p          Push tripods forward if possible.\n";
    cout <<"  -r          Reflect in diagonal.\n";
```

```
  cout <<"  -v          Output as VRML file (for an optimal packing).\n";
  cout <<"  -vt         Output as VRML file (for an optimal packing) with tripods\n";
  cout <<"              trimmed to fit in cuboid.\n";
  cout <<"  -x          No preprocessing.\n";
  cout <<"Common settings:\n";
  cout <<"  -C          Search complete subtree for all solutions (default).\n";
  cout <<"  -F          Fast method (gives correct answer but misses most solutions.\n";
  cout <<"  -N          Naive method.\n";

  cout <<"\nFor more details see:    http://www.dcs.warwick.ac.uk/~mavnp/\n";
  return 0;
}

// get cube size
n = atoi(argv[1]);
if(n<1 || n>20)
{
  cout << "Must choose dimension d1 between 1 and 20\n";
  return 0;
}
l = n;
m = n;

// Try to read in extra dimensions
if (argc>=3 && atoi(argv[2]))
{
  m = atoi(argv[2]);
  if(m<1 || m>20)
  {
    cout << "Must choose dimension d2 between 1 and 20\n";
    return 0;
  }
  par = 3;

  if (argc>=4 && atoi(argv[3]))
  {
    n = atoi(argv[2]);
    m = atoi(argv[3]);
    if(m<1 || m>20)
    {
      cout << "Must choose dimension d3 between 1 and 20\n";
      return 0;
```

```
    }
    par = 4;
  }
}

// set max tripods per plane
maxtp = min(l, m);

// Create the linked list
// Note we will create a list that is (probably) too big, but means we
// can cut down the pre-processing time by changing the orientations
// of the different cuboids.
list = new node(0, 0);
endbuf = new node(0, 0);
list->addstart(endbuf);

// And add the nodes to it, and set other stuff at same time
node* nd;
for (int a = max(l,max(m,n)); a>0; a--)
{
  for (int b = max(l,max(m,n)); b>0; b--)
  {
    node* nd = new node(b,a);
    base[b][a] = nd;
    list->addstart(nd);
    base_count[b][a] = 0;
  }
  vcount[a] = 0;
  hcount[a] = 0;
}

// Set command line args to default vals - store in temp variables
int temp_complete = 1;
int temp_corners = 0;
int temp_diag_bound = 0;
int temp_maxmin = 0;
int temp_push_forward = 0;
int temp_reflect = 0;
int temp_maxtp = maxtp, temp_mintp = mintp;

// Now we will go and read the other command line parameters
// Note that we will read some values directly to temp variables to
```

```
// save copying
for(int a = par; a<=argc-1; a++)
{
  if (argv[a][0]=='-')
  {
    switch(argv[a][1])
    {
      case ('a'): temp_complete = 0; break;
      case ('b'):
        if(a+1 > argc-1 || atoi(argv[a+1])<=0)
        {
          cout << "Error: Positive Integer for lower bound required\n";
          return 0;
        }
        a++;
        lower_bound = max(lower_bound , atoi(argv[a]));
      break;
      case ('c'): temp_corners = 1; break;
      case ('d'): temp_diag_bound = 1; break;
      case ('e'): temp_mintp = max(temp_mintp, 1); break;
      case ('g'): display_report = 1; break;
      case ('l'): display_sols = 1; break;
      case ('m'):
        if(a+2 > argc-1)
        {
          cout << "Error: Need minimum AND maximum values for -m parameter\n";
          return 0;
        }

        temp_mintp = atoi(argv[a+1]);
        temp_maxtp = atoi(argv[a+2]);
        a = a+2;
        temp_maxmin = 1;

        if(temp_mintp<0 || temp_mintp > temp_maxtp || temp_maxtp==0)
        {
          cout << "Error: Invalid min/max values for -m parameter.\n";
          return 0;
        }
      break;
      case ('n'): altnot = 1; break;
      case ('p'): temp_push_forward = 1; break;
```

```
      case ('r'): if (l == m)
                    temp_reflect = 1;
                  else
                    cout << "Warning: Cannot use -r heuristic on this shape - Option ignored.\n";
                  break;
      case ('v'): vrml = 1;
                  if(argv[a][2]=='t')
                    trim = 1;
                  break;
      case ('x'): preprocessing = 0; break;
      case ('C'): temp_complete = 1;
                  temp_corners = 0;
                  if (l == m)
                    temp_reflect = 1;
                  else
                    temp_reflect = 0;
                  temp_diag_bound = 1;
                  temp_push_forward = 0;
                  break;
      case ('F'): if (l == m)
                    temp_reflect = 1;
                  else
                    temp_reflect = 0;
                  temp_push_forward = 1;
                  temp_diag_bound = 1;
                  temp_complete = 0;
                  temp_corners = 1;
                  break;
      case ('N'): temp_complete = 1;
                  temp_corners = 0;
                  temp_diag_bound = 0;
                  temp_push_forward = 0;
                  temp_reflect = 0;
                  preprocessing = 0;
                  break;
      default :
        cout << "Invalid parameter " << argv[a] << "\n";
        return 0;
    }
  }
}
```

```
// ---------------------------------------------------------------------
// Note that we want to complete command line reading before
// preprocessing because:
// (1) We can immediately return an error message if necessary
// (2) We may skip time consuming stuff if certain options selected
//     However, this means we must store some global variables in
//     temp locations

  // Give very loose upper bounds, saves making special cases in main
  // routine
  int array_size = max(l,max(m,n));
  for(int a = preprocessing; a<=array_size; a++)
    for(int b = preprocessing; b<=array_size; b++)
      for(int c = preprocessing; c<=array_size; c++)
        f[a][b][c] = l*m*n;
  if(preprocessing)
  {
    // Copy some global variables to temporary variables
    int temp_m = m;
    int temp_l = l;
    int temp_n = n;

    // How much preprocessing should we do?
    int process_planes = n/2 + 2;

    // Don't output any messages or results
    output = 0;

    // Now get the upper bounds
    for(int mm = 1; mm<=temp_m; mm++)
    {
      for(int ll = 1; ll<=temp_l; ll++)
      {
        for(int nn = 1; nn<=temp_n; nn++)
        {
          if(f[ll][mm][nn] >= ll*mm*nn)
          {
            if(ll>1 && mm>1 && nn>1)
            {
              // Should we fully process plane, or give fast rough
              // upper bound?
```

```
if(nn>process_planes)
  f[ll][mm][nn] = ll*mm*nn; //Any upper bound will do!
else
{
  // Change problem to equivalent but faster to solve
  // problem if possible
  l = mid(ll,mm,nn);
  m = min(ll,min(mm,nn));
  n = max(ll,max(mm,nn));

  // Cut down the linked list to the right size
  if(m<array_size)
    list->del_section(base[1][m+1],base[array_size][array_size]);
  if(l<array_size)
    for (int a = 1; a<=m; a++)
      list->del_section(base[l+1][a], base[array_size][a]);

  // Set up diagonal bounds array
  for (int a = 1; a<=l; a++)
    diagonal_count[a] = l+m - 2*a + 1;

  // Decide whether we can use a reflection or not
  if (l == m)
    reflect = 1;
  else
    reflect = 0;

  // Set initial lower bound based on our previous results
  maxt = max(max(f[l-1][m][n], f[l][m-1][n]), f[l][m][n-1]);

  // Call the main routine, exactly as in the main algorithm
  tri(1, 0, list->next);

  // Store the result in an array
  f[ll][mm][nn] = maxt;

  // Replace the deleted nodes in the linked list
  if(l<array_size)
    for (int a = m; a>=1; a--)
      list->add_section(base[l+1][a], base[array_size][a]);
  if(m<array_size)
    list->add_section(base[1][m+1],base[array_size][array_size]);
```

```
          }
        }
        else // Very easy case, where one dimension is 1
          f[ll][mm][nn] = mid(ll,mm,nn);

        if(!(nn>process_planes))
        {
          // Copy the results as these are all equivalent
          f[ll][nn][mm] = f[ll][mm][nn];
          f[mm][ll][nn] = f[ll][mm][nn];
          f[mm][nn][ll] = f[ll][mm][nn];
          f[nn][mm][ll] = f[ll][mm][nn];
          f[nn][ll][mm] = f[ll][mm][nn];
        }
      }
    }
  }
}

// Reset variables to initial states
maxt = 0;
max_sol_count = 0;

// And recover global variables from temporary storage
m = temp_m;
l = temp_l;
n = temp_n;
}

// Reduce linked list to real size for main problem
if(m<array_size)
  list->del_section(base[1][m+1], base[array_size][array_size]);
if(l<array_size)
  for (int a = 1; a<=m; a++)
    list->del_section(base[l+1][a], base[array_size][a]);

// And copy temporary variables to 'real' variables
complete = temp_complete;
corners = temp_corners;
diag_bound = temp_diag_bound;
maxmin = temp_maxmin;
push_forward = temp_push_forward;
```

```
  reflect = temp_reflect;
  maxtp = temp_maxtp;
  mintp = temp_mintp;

  // Set up the array to deal with diagonal method of upper bounds
  if (diag_bound)
    for (int a=1; a<=l; a++)
      diagonal_count[a] = l+m - 2*a + 1;

  // ------------------------------------------------------------------

  // Want to display output for main function call
  if(!vrml)
  {
    output = 1;

    // Display message about problem
    cout << "Searching for maximal solution (for size ["<< l;
    if(l!=m || m!=n)
      cout << "," << m << "," << n;
    cout << "])...\n\n";
}

  // Take a note of stats about pre-processing
  int pre_node_count = node_count;
  node_count = 0;
  int preplace_count = place_count;
  place_count = 0;

  // Set the initial lower bound
  maxt = lower_bound;

  // Start the main search
  if(corners && l>1 && m>1 && n>1)
    tri(1, 0, list->next);
  else
    tri(1, 0, list);

  // Display results
  if(!vrml)
  {
    cout<<"maximum tripods: "<<maxt<<" ("<<max_sol_count<<" packings found)"<<"\n";
```

```
    cout<<node_count<<" (" <<pre_node_count+node_count<<") nodes\n";
    cout<<place_count<<" ("<<place_count+preplace_count<<") tripods placed\n";
    printf("Running time: %d seconds \n", time(NULL)-t1);
  }
  else
    display_vrml(trim);
}
```