BROWN UNIVERSITY
Department of Computer Science
Master's Project
CS-96-M1

"A Heuristic Search For Linear Programs with 0-1 Variables"

by

Hoog-Shen Wong

A Heuristic Search For Linear Programs with 0-1 Variables

Hoong-Shen Wong
Department of Computer Science
Brown University
Providence, RI 02912, USA

August 29, 1995

A Heuristic Search For Linear Programs with 0-1 Variables

Hoong-Shen Wong

Department of Computer Science
Brown University

Submitted in partial fulfillment of the requirements for the Degree of Master of Science in the Department of Computer Science at Brown University

August 1995

[illegible]

Professor Philip N. Klein

Advisor

Abstract

Solving a linear program with 0-1 constrained variables is an NP-complete problem. Such linear programs have many practical uses in the area of scheduling. This paper describes a heuristic-based method for finding feasible solutions to such linear programs. We will also provide the motivation for attempting to find feasible solutions for such problems by showing how several interesting scheduling problems can be formulated as linear programs with 0-1 constrained variables.

Contents

1 Introduction **1**
1.1 Problem definition 1
1.2 Exact Solutions 2
1.3 Other Approaches 2
1.4 Outline of Search Algorithm 3

2 Motivation **5**
2.1 Airplane Scheduling 5
2.2 Medical Residency Scheduling 8
2.3 Graph partitioning 9

3 Search Algorithm **11**
3.1 Overview 11
3.2 Constraints 13
3.3 Search Values 15
3.4 Data Structures 16
3.5 The Search 18
3.6 Non-Linear Objective Functions 22

4 Results **24**

5 Analysis **28**
5.1 Analysis of Results 28
5.2 Limitations 29
5.3 Conclusion 30

6 Future Developments **31**

A User Guide **33**

A.1 LSearch 33

A.2 Geninit and Search 35

A.3 File Formats 36

A.3.1 Format of Initial Conditions File 36

A.3.2 Format of Constraints File 36

A.3.3 Format of Objective File 37

A.3.4 Format of Output File 37

Chapter 1

Introduction

1.1 Problem definition

This paper serves to define a method for finding feasible solutions to linear programs with 0-1 variables. Such linear programs may be defined in the following manner:-

Find the assignments of values (1 or 0) to the variables in the vector X in order to minimize the objective function $F_0(X)$ subject to the constraints $F_i(X) \geq 0$ for $i \geq 1$ where each $F_i(X)$ is a linear function.

All linear programs can be easily reduced to the above form since $F_i(X) \leq 0$ is equivalent to $-F_i(X) \geq 0$. Moreover, maximization problems can also be determined by minimizing $-F_0(X)$ in cases where $F_0(X)$ is the objective function that needs to be maximized. Hence the above definition would encompass all linear programs as long as the variables have only values of either 1 or 0.

1.2 Exact Solutions

The first question to answer before attempting to develop a heuristic approach to solving this problem is whether there exists an efficient approach to determining the optimal solution.

Linear programs can be solved using variations of the *Simplex* method which traverse the vertices of the polyhedron describing the constraints, based on the property that the optimal solution lies on one of these vertices. In the general case, the vertices are not guaranteed to have integral values. However, (see [1] for a formal proof) a special case exists when a matrix A is totally unimodular , i.e. each square submatrix of A has determinant 0, +1, or -1. In such a case, for each integral vector b the polyhedron $\{x|x \geq 0, Ax \leq b\}$ has only integral vertices.

In the airplane scheduling problem I worked on, the constraint matrix is totally unimodular, thus the solution vectors obtained from the *Simplex* method are integral in value. It is possible to ensure that the solution vectors contain only 1's and 0's by adding additional constraints which constrain all the variables to be between 0 and 1. The addition of such constraints will not affect the total unimodularity of the constraints matrix.

However, not all linear programs have this property and thus in general the vertices of the polyhedron may not have integral values. This leads us to the question of whether a general form of this problem may be efficiently solved in polynomial time using some other approach? Unfortunately, as shown by Garey and Johnson in [2], solving a linear program with 0-1 constrained variables is an NP-complete problem, hence there does not exist any known polynomial time algorithm for finding the exact solution. In order to deal with general forms of this problem, we need a much more robust search algorithm.

1.3 Other Approaches

The easiest approach to solving this problem would be the use of brute force searching. After all, a problem as limited as a linear program with 0-1 variables should not have that many solutions. Unfortunately, even in cases where there are only about 100 variables, there are 1.27e+30 possible combinations to evaluate. Moreover, only a small fraction of these combinations may actually be feasible solutions. Most of these combinations may not fit into the given constraints. Such a search would thus be grossly inefficient.

We looked into a few alternative search methods such as genetic algorithms and simulated annealing. The current implementations of such search methods do not provide much ease of use with linear programs because problems have to be formulated in a manner which is understood by the program performing the search. Such a formulation is time-consuming and different heuristics need to be developed for different linear programs. Thus, even though such algorithms may be more effective in searching, their implementations do not make them easy to use.

The point of this paper is to introduce an implementation of a search algorithm based on a mix of heuristic techniques specifically for solving linear programs with binary variables. Such an algorithm contains built-in heuristics for dealing with slack in the constraints, the values of variables and the objective function. Moreover, there is an efficient feasibility evaluator which will keep track of the feasibility of generated solutions.

1.4 Outline of Search Algorithm

The approach which I will describe in this paper is based loosely on a method for graph partitioning as described by Lin & Kernighan in [3] and simulated annealing as described by S. Kirkpatrick, C. D. Gelatt, Jr., & M. Vecchi in [4]. In order to understand how the search is performed, it is necessary to take a different view of the assignment of values to the variables.

We can essentially look at the problem in the same way as a graph and variables as nodes of the graph (the two terms will be used interchangeably). In this manner, we can look at the problem as one of partitioning. We are essentially trying to partition the nodes into two sets, 1 and 0, representing the values assigned to the variables. The set of constraints specify which partitions yield feasible solutions and which do not.

The search algorithm basically involves swapping nodes from one set to the other set, moving the sets closer and closer to conforming to the constraints. In cases where we cannot find a feasible solution after several attempts, the algorithm would then perform a random swap in an attempt to get out of the local minima. After the sets are found to conform to the constraints, there will be further optimizations in the form of swaps which are used solely to enhance the value of the objective function.

Chapter 2

Motivation

In order to appreciate the usefulness of a technique in finding feasible solutions for linear programs with binary variables, it is necessary to look at several examples of practical applications of such linear programs or slightly more complicated programs with linear constraints and non-linear objective functions. We have dealt with a few scheduling problems which can be easily formulated in the above manner. Being able to find fairly good and feasible solutions in polynomial time will mean that we will be able to obtain reasonable schedules for practical use.

2.1 Airplane Scheduling

I developed an interest in solving linearly constrained problems due to previous work in attempting to schedule airplanes for cleaning. In this case, we were required to schedule individual airplanes for cleaning during a given time period. The planes are to be cleaned at least once every 3 days at airports where they are scheduled to visit. However, each airport has a limit on the number of planes that can be cleaned during any particular cleaning period and each airport has a different rate for cleaning airplanes. Thus, the objective of the scheduling would be to minimize the cost of cleaning the planes while ensuring that the capacity limit for each airport is observed and the planes are cleaned frequently.

In order to solve this as a linear program, we have to take a view of the cleaning opportunities which are available. Thus, if plane I can be cleaned at airport J during time K, the triple (I, J, K) represents a cleaning opportunity. Each variable $X_{i,j,k}$ that represents a cleaning opportunity can be assigned either a value of 0 or 1. If a value of 1 is assigned, then plane i would be cleaned at airport j during time k, otherwise it is not cleaned during that given opportunity.

As I had stated earlier, it is necessary to satisfy the capacity limits in the airports. I had made the assumption that for each airport, there is only one cleaning period and the plane must be there the whole period to be cleaned. As I shall explain later, this is not exactly what happens in reality, but it is a good starting point because it makes the capacity limit much easier to state in terms of constraints.

For each airport J and time period K, the following constraint has to be met:

$$\sum_i X_{i,J,K} \leq \mathrm{Cap}(J,K)$$

where $\mathrm{Cap}(J,K)$ represents the capacity of airport J at time period K, and i is the list of airplanes.

This works out nicely because if plane I is scheduled to be cleaned at airport J during time K, then the variable $X_{I,J,K}$ will have a value of 1, thus adding to the sum. Since each plane which will be cleaned at the particular airport and time period will contribute 1 to the sum, thus the total number of planes cleaned at J during time K will be given by

$$\sum_i X_{i,J,K}$$

.

We also have to ensure that each plane is cleaned during the time period, hence we add an additional constraint for each plane I:

$$\sum_{j,k} X_{I,j,k} \geq 1$$

This constraint ensures that for each airplane I, at least one of the cleaning opportunities will be assigned a value of 1, thus ensuring that the plane is cleaned at least once during the time period.

Now we describe the objective function. We associate a certain cost with each cleaning opportunity (i, j, k), which depends on the time from the last cleaning to this opportunity as well as the actual cost of performing the cleaning. The calculated cost is denoted by $C_{(i,j,k)}$, which may be manipulated as necessary in order to obtain a fair estimation of the cost. (The process of calculating the cost will not be described in detail here as it is not an integral part of the problem.) Finally, we will obtain an objective function to minimize as follows:

$$\sum_{i,j,k} C_{(i,j,k)} X_{(i,j,k)}$$

This is an extremely simple example of a scheduling problem because it can be neatly defined as a linear program. Moreover, the constraint matrix is totally unimodular, which meant that when this linear program is solved through the use of the *Simplex* method, the variables will be assigned discrete values, which in this case are either 1's or 0's.

Unfortunately, the real problem we had to deal with was not so simple. In reality, a cleaning opportunity exists for a plane as long as it stays at an airport for at least two hours during the airport's cleaning period. Furthermore, the capacity limit is not given by the total number of planes that can be cleaned during the cleaning period, but how many cleaning crews there are around, and thus how many planes can be cleaned at any point in time. Thus, it is necessary to sub-divide the cleaning time period into smaller intervals (15 minutes would be a fair estimate). Instead of using a triple (i, j, k), we need a quadruple (i, j, k, l) where l represents the time period when cleaning would start, i.e. $l = 0$ refers to the first 15 minutes, $l = 1$ refers to the next 15 minutes and so on. We also add additional constraints to ensure that each plane is cleaned at most once during each airport/time period.

$$\sum_{l} X_{(I,J,K,l)} \leq 1$$

In order to maintain the constraint imposed by the number of work crews, we have to slice up time and and ensure that at any point in time, the number of planes worked on is at most the number of work crews at that airport. Since there is a two hour period in which crews worked on planes, crews working on planes starting at period t would not be able to work on anything else until time $t+8$ where each unit of time represents 15 minutes. Hence, a variable which has a value of 1 for time period t will affect all constraints from t to $t+7$. For each time period t, it would be necessary to introduce a constraint which sums up all the variables for that airport/time-period from time $t-7$ to t to ensure that the work crews are not working on more than one plane at a time.

2.2 Medical Residency Scheduling

Even though scheduling for residents in the medical program is also a scheduling program, it is fairly different from the previous type of scheduling. While the previous type of scheduling seems fairly straightforward and well suited to formulation as a linear program, the scheduling required for medical residents is somewhat more complicated.

The scheduling was required for a group of 28 interns and another for 54-58 residents. Each resident does a different rotation each block - a block being either 4 weeks, 6 weeks or 8 weeks. Some rotations are much harder than others, so it is important that each resident gets a fair number of easy versus hard rotations, and has the hard ones spaced out to avoid as much burnout as possible. Some of the rotations are subspeciality consult services, and if a resident has done a particular one one year, it is better that they do not repeat it the next. Residents can make all sorts of preferences such as when they want their vacations (which can only come out of some rotations and not others), which consult services they want to do, particular times that they need easier months (eg. during interview season if they will be applying for a fellowship,) which hospitals they want to work at, etc., etc. Not all requests are granted, but the goal is to get as close as possible. There are constraints on who can do hard rotations in the beginning of the year as at first we only trust 3rd year residents, and by early fall, only the best of the 2nd years.

As in the first scheduling problem, we can just create variables for each triple (I, J, K) specifying that resident I is assigned rotation J during block K. The hard constraints and preferences can be specified as part of the problem similar to the plane scheduling problem as I will show later.

Many of the preferences and requirements of the problem can be formulated as part of a linear program. The objective function would take into account the unhappiness factor in assigning the different rotations, thus the residents would be able to specify the happiness or unhappiness associated with each assignment they obtained (for instance, each resident may have up to 1000 points to spend on specifying which assignments would make them happy and which assignments would make them unhappy). Thus, if a resident I specified 100 points to bolster his/her chance of getting assigned rotation J during block K, then $-100X_{(I,J,K)}$ will be added to the objective function, thereby reducing the objective function if such an assignment is made. The opposite (i.e. $100X_{(I,J,K)}$) would be added to the objective function in the case where the resident does not want to be assigned rotation J during block K. In the case of rotations which cannot be assigned to particular residents, those are just not put into the problem definition, i.e. variables are not created for such choices.

In this problem, we also had to spread the hard rotations across the semester for each resident so that each resident does not get two hard rotations in a row. Although this is a soft constraint, meaning that some resident may get two hard rotations in a row if there was no other choice, we can make it into a hard constraint in order to solve it as a linear program. We add constraints to limit the number of hard rotations for each resident in a certain time period (maybe 8 - 10 weeks) to be 1. If we are unable to find a feasible schedule with the chosen time period, we can reduce the time period, thus increasing the probability of some residents getting two hard rotations in a row and also the probability of obtaining a feasible schedule.

2.3 Graph partitioning

Graph partitioning is an example of a problem which cannot be easily formulated in the form of a linear program with 0-1 variables. The problem to be solved here is how to partition the set of nodes in a graph into two equally numerous sets such that the sum of the weights of the edges between nodes in different sets is minimal. In this case, we can formulate the problem as a set of linear constraints with a non-linear objective function.

In this problem, each node will be made a variable which is assigned 1 if it is in one set and 0 if it is in the other. There is only one equality constraint: that the sum of all the variables is equal to half the number of nodes (which is equivalent to two inequality constraints, the first stating that the sum of all the variables is greater than half the number of nodes and another stating that the sum is less than half the number of nodes.)

The entire problem of minimizing the weights of the edges would be encoded into the objective function. Each pair of nodes I,J which has an edge will be added to the objective function multiplied with the weight of the edge. Thus, the objective function will be:

$$\sum_{i,j} C(i,j)X_i(1-X_j) + C(i,j)(1-X_i)X_j$$

where i and j are the nodes and C(i,j) is the weight of the edge going from i to j.

In the case where both nodes i and j are in the same set, $X_i(1-X_j) = (1-X_i)X_j = 0$, so the weight of the edge will not be added to the objective function. However, should the two nodes be in different sets, then either $X_i(1-X_j)$ or $X_j(1-X_i)$ will be true but in no case will both be true. In that case, the value $C(i,j)$ will be added to the objective function. Hence the objective function gives the sum of the weights of the edges to be removed which is the value to minimize.

Chapter 3

Search Algorithm

3.1 Overview

In this section, I will give a description of how the search is performed. A random starting point in the search space is generated by randomly assigning either 1 or 0 to each variable. The search is then performed through a series of swaps transforming the starting point into a feasible solution. Each swap changes the value of a variable from either 1 to 0 or 0 to 1. It may be viewed in the following manner: variables are treated like nodes in a graph and this is similar to a partitioning problem. If a variable has a value of 0 it belongs to one block of the partition, if it has a value of 1 it belongs to the other block, thus a swap would basically be moving a variable from one block of the partition to another (Note: this is only to facilitate understanding, the implementation is slightly different).

There are two different concepts in use here: the value of the objective function, and the value of the search function. The value of the search function is basically a heuristic to determine how close a given solution is to the desired minima and is used during the search cycles to order swaps.

At the beginning of each cycle of the search, each swappable variable

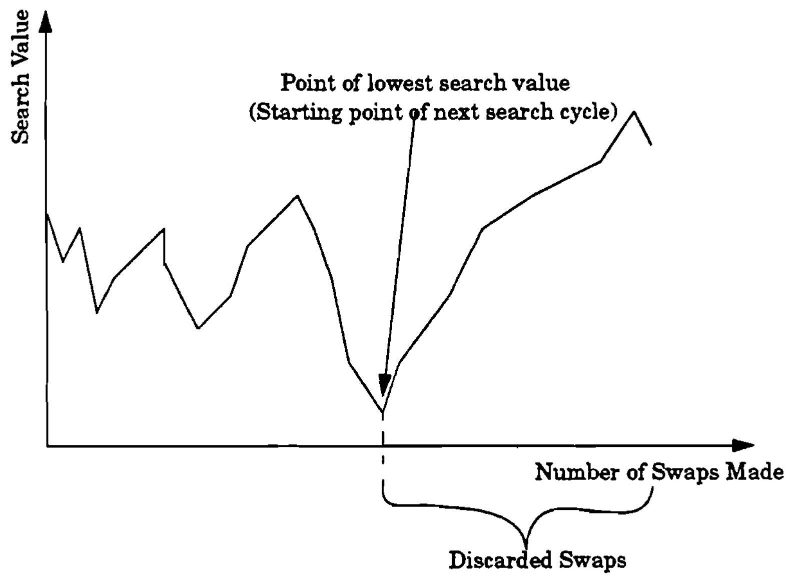
Search Value
Point of lowest search value
(Starting point of next search cycle)
Number of Swaps Made
Discarded Swaps

Figure 3.1: What happens during each Search Cycle

(how a variable is deemed swappable is discussed in Section 3.2) is placed onto a priority queue according to how much the swaps would benefit the search value. The best swaps would be made first, and as each swap is made, some variable may then be redesignated as unswappable, and thus removed from the priority queue, and other variables which become swappable will be placed onto the priority queue. After each variable is swapped in that cycle, it is frozen and no longer swappable for that cycle, hence each variable can only be swapped once in the cycle. The best search value is stored for that cycle and if there was a improvement in the search value, we will reverse the swapping to that point and unfreeze all the variables for swapping again (See Figure 3.1). The search cycles are repeated again using the solution which yielded the lowest search value as the starting point until there is no improvement in the search value for the current cycle over the previous one.

At the final stage, if we have found a feasible solution, then we will perform an optimizing cycle, whereby we will only consider the contribution to the objective function of each variable, instead of the search value. This will allow us to further improve the objective function, totally ignoring the effects on the constraints except to maintain the feasibility of the solution. If we have not found a feasible solution, the algorithm will take the constraint with the most negative slack and randomly select one variable to swap (this may or may not increase the number of constraints which are over the limit) and searching will continue from there. The current implementation of the program makes just one random swap before giving up.

Since the class of problems we are trying to solve in this case is NP-complete, there is no guarantee we will obtain the optimal answer. Generally, we cannot even determine whether the answer we get is anywhere close to optimal. Thus, it is usually useful to perform various searches starting at different random locations and keeping track of the best feasible solution obtained. In this case, we can manage a trade-off between time spent and optimality of the solution. We can make more searches from different starting locations and thus a better solution may be found if we have more time.

3.2 Constraints

One of the purposes of the swaps in the algorithm is to transform the solution to one which will comply with the set of constraints. Whether a solution fits the set of given constraints is determined by how much slack it has.

$$-100X_0 + 14X_1 - 13X_2 + X_3 - 2X_4 \quad \text{Minimize} \tag{3.1}$$
$$4 - 3X_0 - X_1 - X_2 - X_3 - X_4 \geq 0 \tag{3.2}$$
$$1 - X_1 - X_2 \geq 0 \tag{3.3}$$
$$-1 + 2X_1 - X_4 \geq 0 \tag{3.4}$$

Each constraint has a slack value associated with it. This slack value is equivalent to the evaluated sum of the left-hand side of the inequality. Table 3.1 shows the effect of a swap on the slack value of a constraint given the coefficient of the variable in the constraint. If a constraint has a slack value greater than or equal to 0, then the constraint has been met. To better illustrate this point, take a look at a simple linear program described by the objective function to be minimized, given in Equation 3.1 and the constraints given by Inequalities 3.2 to 3.4. Given the assignments shown in Figure 3.2, the slack values for the 3 constraints are 2,0,-2 respectively. Since $-2 < 0$,

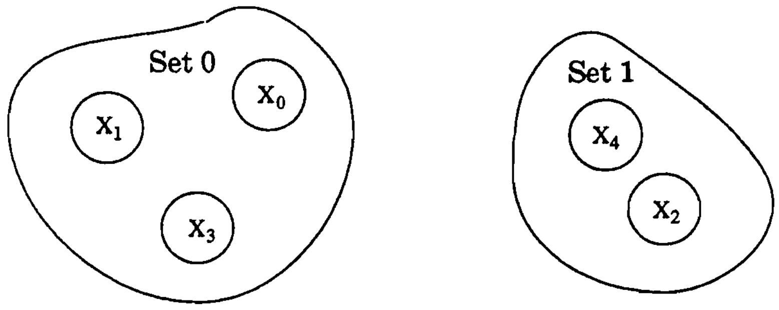

Figure 3.2: Initial partitioning of the Set

Slack	$1 \rightarrow 0$	$0 \rightarrow 1$
Coeff		
+ve	$\uparrow$	$\downarrow$
-ve	$\downarrow$	$\uparrow$

Table 3.1: Effect of Swap on Slack Value of Constraint

constraint 3.4 is not met. Hence the given assignment does not satisfy the set of constraints and so it does not represent a feasible solution.

The slack value plays a very important role in the search process as we want the search to move towards solutions which are more likely to be feasible. Thus, swaps are made if and only if they do not increase the number of constraints which are not met (except in the case of random swaps). For constraints which are already over the limit, the swap must not reduce the slack value any further. Table 3.2 and 3.3 show what types of swaps are allowed and what types are not, based on the slack value of the constraint and the coefficient of the variable in that constraint. In order for a variable to qualify as swappable, it is necessary to be swappable in all constraints in which its coefficient is not 0. For instance for the given linear program, if X_2 is in the 1 set and X_1 is in the 0 set, then X_1 would not be considered swappable because swapping it would make the slack value of constraint 2 negative.

Essentially what these set of rules do is to ensure that throughout the search process, the sum of the negative slack values of all the constraints will be always be decreasing, i.e. searches are always proceeding closer towards a feasible solution. This property of the search is particularly useful in the

Slack in Constraint	$< \lvert coeff \rvert$	$\geq \lvert coeff \rvert$
Coeff		
+ve	Y	Y
-ve	N	Y

Table 3.2: Can a variable be changed from $0 \rightarrow 1$?

case where the search has obtained a feasible solution. In such a case, future swaps will not destroy the feasibility of the solution.

3.3 Search Values

In performing searches, we have to develop an effective heuristic for searching. In this case, we use the concept of a search value, which tells us how well the search is progressing.

In the current implementation, the search value of each swap is given by the contribution of the variable to be swapped to the objective function if a swap is performed plus the effect on the constraint multiplied by the contribution to the objective function.

The effect of a swap on the constraint is determined by the minimum slack value obtainable in that constraint. Swaps which do not decrease the slack value have no constraint effect value. If the minimum slack value a constraint can have is positive or 0, it means that this constraint will always be satisfied, thus swapping variables will never affect this constraint, hence such constraints have no effect on the swappability of variables. These constraints can be safely ignored in the search process. In the event where the maximum slack of any constraint is less than 0, the set of constraints specified does not have any valid solution, since the constraint can never be satisfied and the search algorithm will terminate there. The only significant situation is when the minimum slack value of a constraint is negative. The absolute value of the slack value will be used to calculate the effect of a swap on the constraint.

Slack in Constraint	$< \|coeff\|$	$\geq \|coeff\|$
Coeff		
+ve	N	Y
-ve	Y	Y

Table 3.3: Can a variable be changed from $1 \rightarrow 0$?

3.4 Data Structures

A major design issue in the search algorithm is to ensure that the search progresses smoothly without being bogged down in calculating and recalculating the different values required for assessing the feasibility and value of the current solution. The Search Engine is designed to keep track of the variables and data structures as shown in Figure 3.3. There are two priority queues in the search engine used to maintain the order in which swaps are to be made and also to efficiently find the constraint with the most negative slack in the event that a random swap is to be made. A modified version of the priority queue is used in order to allow elements in it to have their key values changed and their positions in the queue updated. The modification is made through the use of an array to represent the queue. Nodes whose key values are changed will then perform upheap or downheap depending on which is necessary. The search engine will also need to keep track of the best objective function obtained , the best search value obtained and the number of constraints which are over the limit. This will tell the program whether the current solution satisfies the set of constraints.

Prior to beginning with the search, the algorithm will be given the starting assignments, the list of constraints and the objective function. The constraints are stored in the manner shown in Figure 3.4. Each constraint has a constant value associated with it, as well as the amount of slack it has. For each variable in the constraint, a pointer to it along with its coefficient is added into a linked list in that constraint.

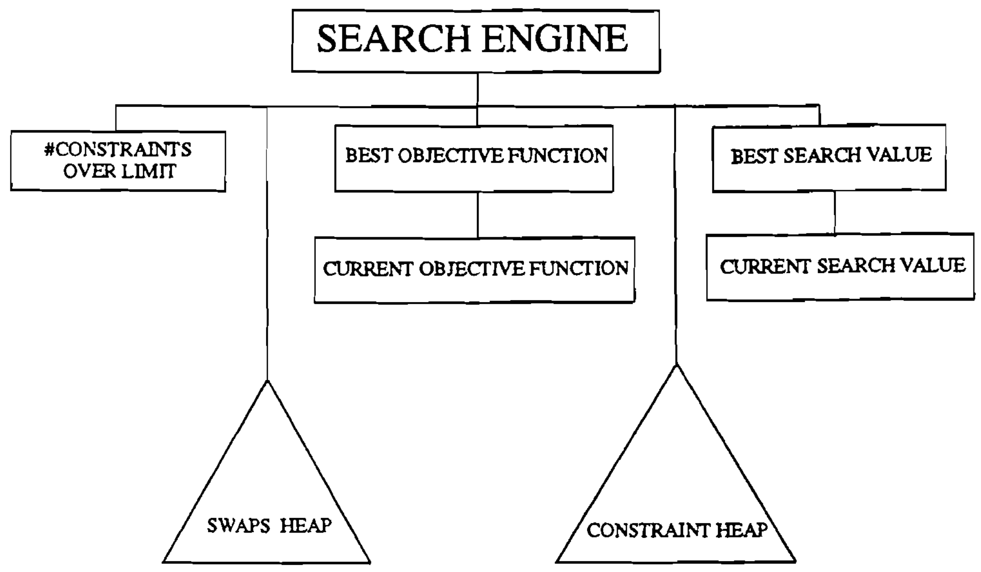

Figure 3.3: Data maintained by the Search Engine

Variables in the linear program are stored in the program as an array of n elements, where n is the number of variables in the linear program. Each element keeps track of its coefficient in the objective function. The value of the objective function is given by

$$\sum_i C_i X_i \tag{3.5}$$

where C_i is the coefficient of variable i in the objective function and X_i is the value of the ith variable.

Each variable has a list of pointers to constraints whose slacks would be affected if a swap were to be made with this variable. This is used for updating the slack value of all the constraints. Moreover, in order to keep track of whether it is swappable, each variable keeps a number which tells it how many constraints would not allow this variable to swap from 0 to 1, as well as how many constraints would not allow this variable to swap from 1 to 0. This is a very efficient manner of determining whether the variable is swappable, because as each constraint's slack value is changed, those two pieces of information in the variable can be updated as well. This minimizes the actual processing required to determine whether a variable is swappable.

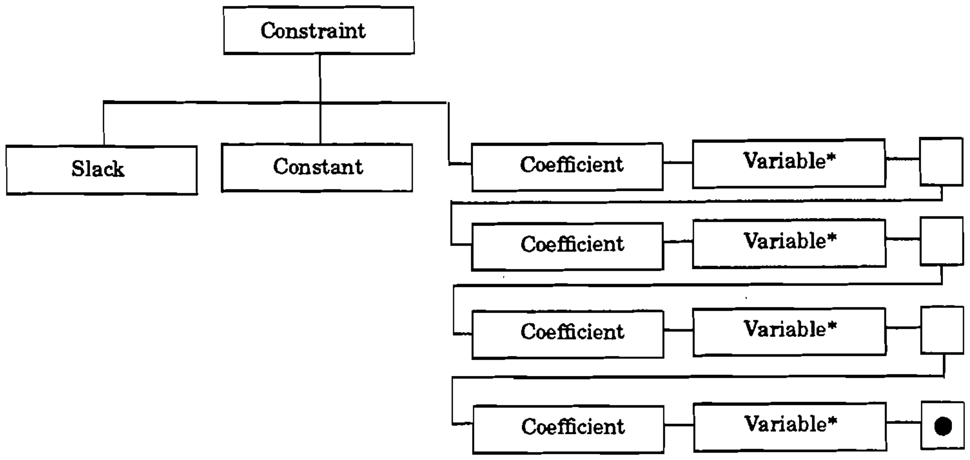

Figure 3.4: Data Structures for storing Constraints

3.5 The Search

After reading in all the initial data, the search routine will cycle through the list of constraints and determine how much slack each has. As the slack value of the constraints are set, each variable will also be updated with the number of constraints which would not allow swapping. The constraints are placed into a modified priority queue sorted by their slack values, with the constraint having the least slack (or biggest negative slack value) at the top of the queue.

After all the initialization steps are performed, the search process begins in earnest. At the beginning of each search cycle, all the variables are unfrozen allowing for swapping to be done. All the swappable variables are placed into a queue sorted in order of the contribution to the search value, thus the variables which reduces the search value by the greatest value will be swapped first.

The variables are then taken off the queue one at a time, and the variable is swapped. As the swap is made, the change to the objective function is calculated along with the change to the search value and the search engine is updated with these new values. There is no necessity to recalculate either the objective function or the search function. The change to the objective function due to variable i being swapped can be calculated as follows:

$$\delta_{obj} = \begin{cases} \text{Coeff}_{obj}(X_i) & \text{if swap is 0 to 1} \\ \text{-Coeff}_{obj}(X_i) & \text{if swap is 1 to 0} \end{cases}$$

The update of the search value is performed in the same manner as the objective function. The current search value is not recalculated, only the change is calculated. This is taken directly from the search value of the swap.

After calculating the changes to the objective function and search value, the search algorithm needs to update the slack values of all the constraints in which the swapped variable has a non-zero coefficient. Each constraint has its slack value changed by the following formula:

$$\delta_{slackvalue,C_j} = \begin{cases} \text{-Coeff}(C_j, X_i) & \text{if swap is 0 to 1} \\ \text{Coeff}(C_j, X_i) & \text{if swap is 1 to 0} \end{cases}$$

where C_j is the jth constraint, X_i is the variable that is swapped and Coeff(C_j,X_i) is the coefficient of the variable X_i in the constraint C_j.

As the slack values of the different constraints are adjusted, the different variables in the constraints have to be updated to ensure that they are keeping track of whether they are swappable. This is done through two pieces of information which tell it how many constraints would go over the limit if it were swapped from 1 to 0 and vice versa. The search algorithm uses Table 3.2 and Table 3.3 in order to determine the effect of a swap. If the table disallows swapping from 0 to 1, then the number of constraints which would go over the limit during a swap from 0 to 1 would be increased by 1. The same is done for swaps from 1 to 0 using the table. This is updated regardless of what state the variable itself is in, i.e. even if the variable had a value of 1, the information for both types of swaps would be updated. It is done in this manner in order to keep updating minimal. Thus, whether variable is swappable from 0 to 1 is easily determined by checking whether the number of constraint which will be broken during a swap from 0 to 1 is equal to 0 and vice versa for swaps in the other direction. A variable can be marked as unswappable after being placed in the queue. In such a case, the proposed swap will be ignored.

The change in slack values of the constraints are also propagated to the queue used to keep track of which constraint has the highest negative slack value. The slack value of the constraint being modified may change its position in the queue and this is updated accordingly, thus effectively performing dynamic sorting. In this case, we only need to know which constraint has the highest negative slack value, hence this is an efficient manner to keep track of that without resorting to deletions and insertions.

After all the accounting has be taken of, the search algorithm proceeds to freeze the variable to ensure that it is swapped only once during this cycle. Then the next best swap is taken off the top of the queue and the swap made and the updating is performed once again. This continues until there are no more swaps to be made. In a linear program with n variables, there are at most n swaps that can be made during each cycle, thus each cycle is guaranteed to terminate at some point. Upon termination, the program determines which is the best point in the search cycle to revert to by determining what is the best search value obtained. Then, the variables are unswapped to the solution which yielded the best search value. This is a lot more efficient than keeping track of the best solution because in the latter case, we need to keep track of the state of the data maintained by the search engine for the best solution, which would require much more memory and take more time to make a copy.

The search then starts from the best solution obtained in the previous cycle and starts swapping all over again. If the search has been unable to obtain a feasible solution and the current cycle has completed without any improvement to the search value, then a random swap will be made and the search will proceed from that point. If even after the random swap is made, there is no improvement in the best search value obtained in the current cycle, the search will terminate signifying that the solution is infeasible. In the event that a feasible solution is found, the search will continue until the current cycle is unable to obtain any improvement to the search value. In the last round of search, an optimizing pass is performed. In this round, the swaps are ranked according to only their contributions to the objective function instead of their search values. Thus, changes to the slack values of the constraints no longer has any effect, only the amount by which the objective function can be reduced. This is useful because it is often possible to obtain a better solution than the one obtained solely from the previous few cycles just through that one cycle. It effectively exchanges excess slack for a better objective function value.

The entire search process is shown as a flow of control diagram in Fig-

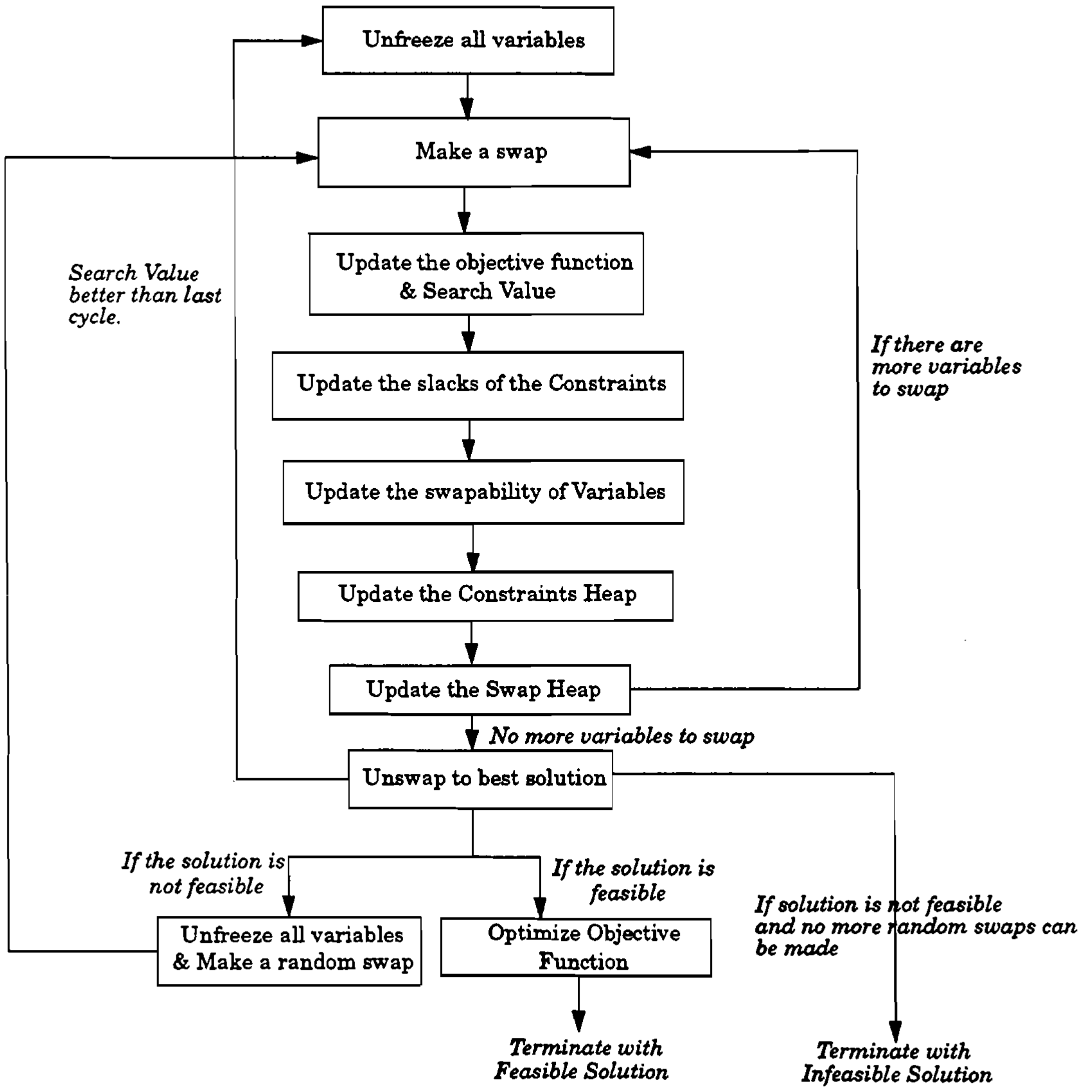
Unfreeze all variables
Make a swap
Update the objective function & Search Value
Update the slacks of the Constraints
Update the swapability of Variables
Update the Constraints Heap
Update the Swap Heap
Unswap to best solution
Unfreeze all variables & Make a random swap
Optimize Objective Function
Search Value better than last cycle.
If there are more variables to swap
No more variables to swap
If the solution is not feasible
If the solution is feasible
If solution is not feasible and no more random swaps can be made
Terminate with Feasible Solution
Terminate with Infeasible Solution

Figure 3.5: Search Sequence

ure 3.5. As shown earlier, each search cycle is guaranteed to terminate in at most n swaps where n is the number of variables. The entire search process is also guaranteed to terminate in one of two cases, either a feasible solution is obtained or an infeasible solution is obtained. It is obvious there is a limit on the improvement of the objective function (since variables can only be 1 or 0) and a limit on the slack value (since each constraint has a maximum and minimum slack value). As the search value can only be improved through an improvement in the objective function or the slack values of the constraints, there is also a limit to how much the search value can be improved on. Hence improvements in the search value will stop at a certain point, at which the search may either terminate or make a random swap. However, as long as there is a limit on the number of random swaps that can be made, the search is guaranteed to terminate.

3.6 Non-Linear Objective Functions

In our implementation of the search algorithm, we attempted to implement non-linear objective functions which will enable us to find solutions for problems which have linear constraints and non-linear objective functions. We deal with a very small sub-set of non-linear objective functions, namely objective functions with terms made up of multiples of different variables such as $mX_aX_bX_c$... (which we will refer to as conjunctive terms). We chose to deal with only such a small sub-set because, as we have shown in Section 2.2 and Section 2.3, some problems are easier to solve using non-linear objective functions in the above form.

Our current implementation does not deal entirely with the problem of non-linear objective functions with only conjunctive terms, but it provides some support for conjunctive terms.

The objective function is the sum given in Equation 3.5 plus the contribution of the conjunctive terms. A conjunctive term will only contribute its coefficient in the objective function to the objective function if every single variable in that term has a value of 1. If even one of the variables in the conjunctive term has a value of 0, then that conjunctive term will not contribute anything to the objective function.

Each conjunctive term keeps a linked list of pointers to variables in it and the coefficient of the term in the objective function. The update of the objective function is done by keeping track of how many variables in that term have a value of 0. If a swap changes the number of zero variables from 1 to 0, it means this conjunctive term starts to have an effect on the objective function, thus the value of the coefficient of that term is added to the objective function. If a swap changes the number of zero variables from 0 to 1, then that means the conjunctive term no longer contributes to the objective function, thus the change in the objective function is the negative of the coefficient of the term in the objective function.

Although it is possible to perform searching on these types of problems using our current implementation, the search does not use any heuristics based on the conjunctive terms. The implemented heuristic takes into account only the linear constraints and expects a linear objective function. Thus, the search is unlikely to be as efficient as for linear programs.

Chapter 4

Results

As a test of the ability of the algorithm to find fairly good answers, I ran it on a set of linear programs which fitted the conditions in Section 1.2 and thus optimal solutions could be found using the *Simplex* method.

The linear programs are basically derived from the airplane cleaning problem I had worked on. It is based on only the first portion of the problem described in Section 2.1. Thus there is a linear objective function and there are two types of constraints in the linear program. The two types of constraints specify that each plane has to be cleaned at least once in the given time period, and the capacity limits of the airports have to observed.

Figure 4.1 to Figure 4.6 show the average best objective function value graphed against the number of searches performed. A search in this case refers to multiple searches from different random starting points until a feasible solution is obtained. For each test set, we ran 1000 searches for feasible solutions. We obtained the average best objective function value for each value n on the X-Axis by taking the best value for each n-sequence of searches and finding the average of those best values for each n.

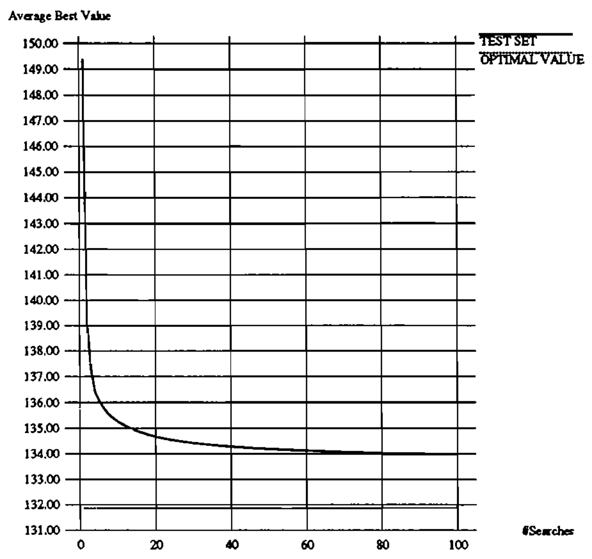

Figure 4.1: Test Set 1 with solutions averaging 1.5% above optimum for 100 runs

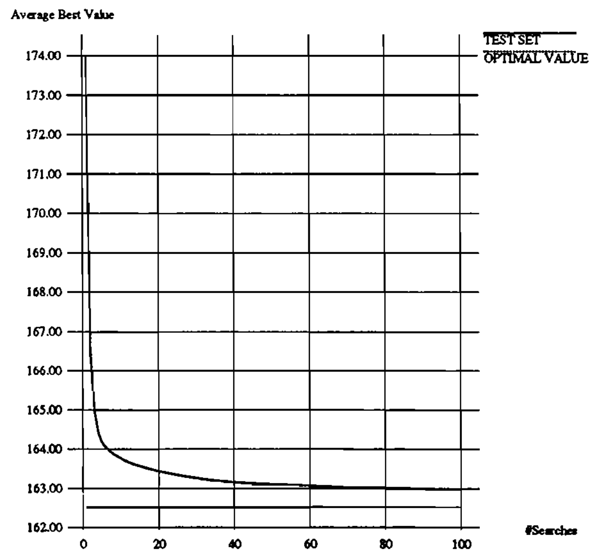

Figure 4.2: Test Set 2 with solutions averaging 0.3% above optimum for 100 runs

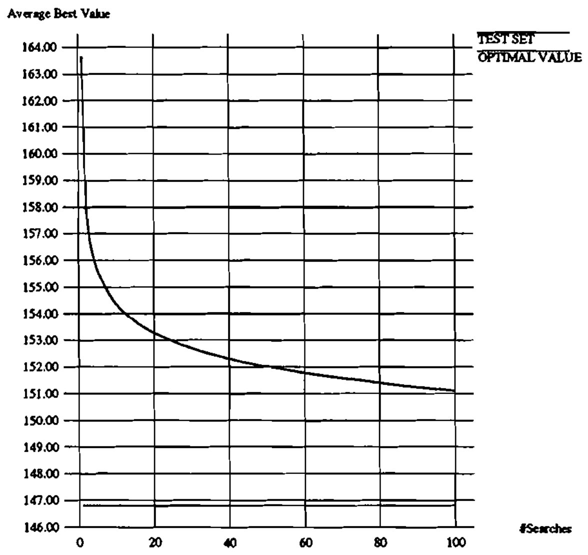

Figure 4.3: Test Set 3 with solutions averaging 2.9% above optimum for 100 runs

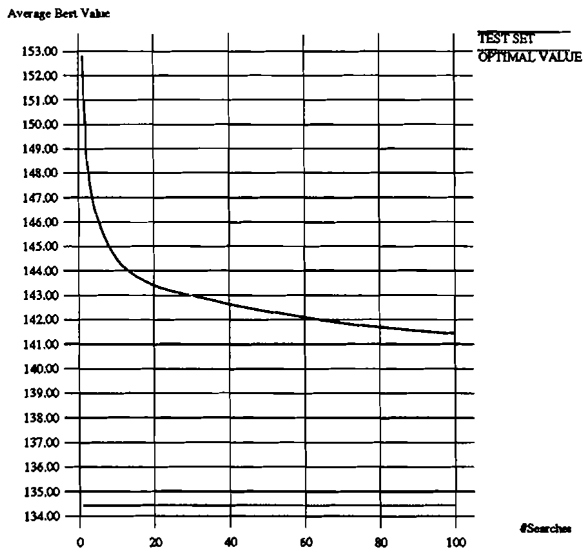

Figure 4.4: Test Set 4 with solutions averaging 5.2% above optimum for 100 runs

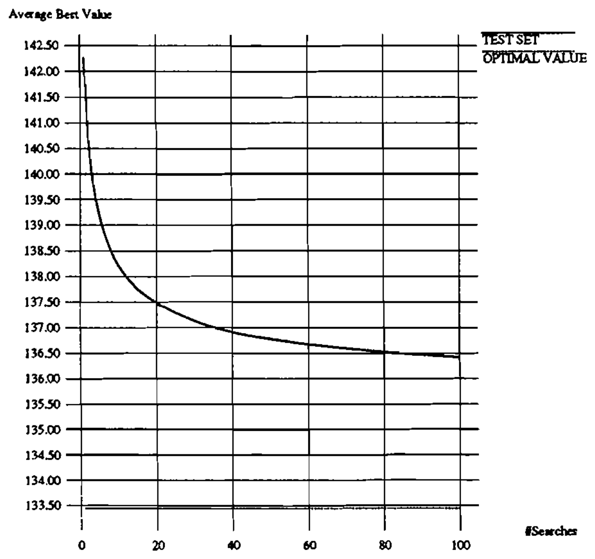

Figure 4.5: Test Set 5 with solutions averaging 2.2% above optimum for 100 runs

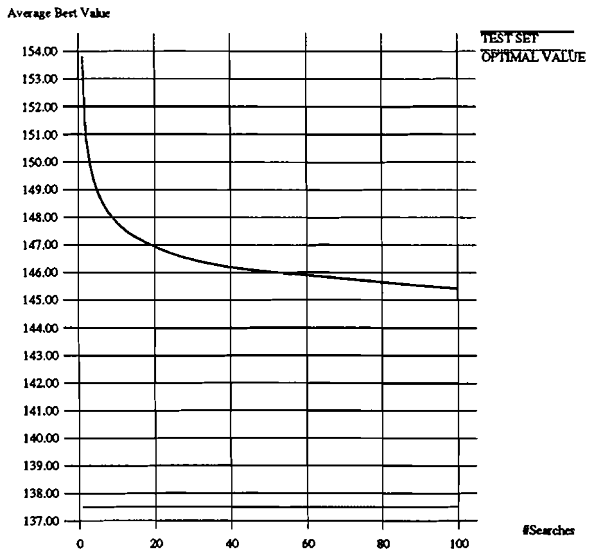

Figure 4.6: Test Set 6 with solutions averaging 5.8% above optimum for 100 runs

Chapter 5

Analysis

5.1 Analysis of Results

The results obtained in the previous chapter show that it is possible to obtain a trade-off between number of cycles spent on the search and the optimality of the solution obtained. As we spend perform more and more searches, we tend to obtain better and better answers which are close to the optimal solutions. As this is a heuristic approach, generally it may never reach the optimum, but in most cases, a fairly good approximation may be sufficient. However, it has to be noted from the tests we ran, that the search produces a solution that is reasonably close to the optimal very quickly (after obtaining about 20 feasible solutions) and further searches would only improve the solution marginally.

The time taken for the searches vary widely depending on the number of variables, the number of constraints, the number of variables which appear together in over-constrained constraints and the trade-off between the slack and objective function value in calculating the search value. On average, some linear programs will take much longer than others. Of the 10 test sets I have experimented on (on Sparc 10s with 32Mb of real memory), the longest took an average of 115 seconds per search which produces a feasible solution, while the shortest takes an average of 3 seconds for the same type of search. Thus, it is likely to take less than 5 minutes to generate each feasible solution and making hundreds of searches is a feasible option.

One way of coping with the huge number of searches is to perform searches on parallel machines. Different search processes can be started on different machines until a feasible solution is found. Thus the time taken to perform 10 searches on 10 different machines would be at most the worst case time for one search.

5.2 Limitations

There are certain limitations in this algorithm. The most obvious one being that we need to start at various random points in order to obtain even a feasible solution to the problem. Unfortunately, in many cases it is not possible to get to some solutions using this form of swapping. This is due to the way each search cycle is performed and the flaws inherent in searching only in the space of feasible solutions. As the best swap is made first, this meant that it would affect the constraints and thus future swaps are restricted by the earlier swaps in that cycle. Unfortunately, in most cases, certain swaps which may have led to a feasible solution would not have been made because by the time it was their turn to swap, they were no longer swappable.

Another fairly obvious limitation of such a search algorithm is the inability to deal adequately with linear systems with constraints that specify an equality. For instance, if a $F_i(X) \geq c$ and $F_i(X) \leq c$ are two valid constraints. However, if the starting point is one in which both constraints are met, then the search basically has no way of swapping out of there, since swapping out of one constraint would cause the other to be broken. Hence, it is difficult to deal with such a problem, which is the case with the graph partitioning problem. That is not to say the search algorithm will not be able to deal with these types of problems, just not as effectively as with other types of problems whereby swapping is a much more efficient approach. Since the search rarely starts with a point which fits all the constraints, it would still be able to performing some form of swapping to be able to get to a better solution than purely random assignments.

The most significant limitation of this approach is similar to all other heuristic approaches, i.e. there is no easy way of telling whether or not there exists a feasible solution. Although pre-processing in the form of determining the maximum slack value of each constraint can be used to weed out some problems without feasible solutions, there still exists many linearly constrained problems with no feasible solutions. Only through multiple search failures is it possible to conclude with a high degree of certainty that there is probably no feasible solution.

5.3 Conclusion

Even given its limitations, there are good signs from the results obtained due to the speed in which it is able to obtain fairly good results. Moreover, the optimality of the results can be time-dependent. If a better result is desired and there is a lot of time available, it may be possible to just perform search from different starting points and keeping track of the best result. Otherwise, if the application is pressed for time and does not particularly require absolute optimality, then fewer search runs may be made.

Chapter 6

Future Developments

There are several areas of development which would be nice to have in this search algorithm. The current implementation does not have the ability to deal with linearly constrained problems with non-linear objective functions efficiently. Even though the algorithm has several features which basically allow for dealing with conjunctive terms in the objective functions, those terms are not actually dealt with in terms of the developed heuristic. That is, to say, the heuristic does not take into account the conjunctive terms when performing swapping. As shown in the various sections on the medical scheduling as well as graph partitioning, certain classes of problems have to be specified in this form, thus it would be useful to be able to make use of the conjunctive terms in the search.

There are several ideas I had with using the conjunctive terms in the heuristics. This deals with both the search values as well as the objective function. In the case of the search values, it would be fairly easy to assign a certain value to each variable in a conjunctive term which depends on the coefficient of the term in the objective function, as well as the number of variables in that conjunctive term as shown in the following formula

$$value = k\frac{\text{coeff}}{\#\text{Variables}}$$

where k is a constant which may be varied in order to signify the importance of this component of the search value.

In the case of objective function values, it is a bit more complicated, because at any one point in time, we are considering only the effects of one swap. This means that each swap will cause a change in the objective function only if that variable is the last non-zero variable in that term that is being swapped from 1 to 0, or all the variables in the term are 0 and one is being swapped from 0 to 1.

This points out another another glaring limitation of the current approach to search, which is due to my desire to keep the implementation as efficient as possible. Each swap's search value depends solely on it being the next swap performed. Ideally, it would be nice to be able to predict what would happen for a series of swaps instead of just one swap. It would be possible to incorporate some sort of combination swaps, such as two-swaps, where changes in search values and objective function are dependent on both swaps being performed one after another. This would result in $O(n^2)$ swaps per search cycle if two-swaps were used, instead of at most n swaps. Of course, the more swaps considered together, the better the result, but unfortunately this would mean an exponential increase the number of swaps possible per search cycle. However, all things considered, it still might be useful to consider small number of swaps together in order to improve the performance of the search algorithm.

Appendix A

User Guide

A.1 LSearch

LSearch is an executable which performs the search procedures as explained in this paper, proceeding with a given starting point. It can be executed in the following manner:

```
sh> LSearch <initcond> <constraints> <objective> [output file]
```

The starting point of the search is specified to the search program in the file *initcond*. The constraints are specified in the file *constraints* and the objective function is specified in the file *objective*. The *output file* is an optional argument which specifies the file to output the final solution to. This file is written to when the program terminates, regardless of whether the solution obtained is feasible or not.

LSearch outputs the state of the solution after each swap is made. Figure A.1 shows the value of all the variables in the current solution. If a * is placed in front of the solution, it means that the solution is a feasible one. The number enclosed in the <> shows the number of constraints that are

```
*01100010101100101001000111001000011001101011011010110100100100
10010100100110100101000101001010101110100101110010101001100101 1
0100110010<0>->
```

Figure A.1: Output of Solution State

```
CYCLE 2
Number of swaps attempted:70
Best Change to objective function so far:0
Best Change to objective function in this cycle:0
Number of steps to back-track:70
Number of actual swaps made:0
Search Function value:0
Number of constraints over:0
```

Figure A.2: Report at the end of each cycle

over the limit. Since this solution is feasible, it means that the number of constraints over the limit has to be 0. In other cases, it is possible to tell how close we are to a feasible solution during the search.

At the end of each search cycle (shown in Figure A.2, a report is generated which tells us among other things how many swaps were attempted and the number of steps to backtrack. It also tells us whether a valid solution has been found from the number of constraints over the limit as well as the changes to the objective function obtained during this cycle.

At the end of the search, the program will output a final report. If the solution given is infeasible, then there will be a line specifying it as such, along with a line stating the number of constraints which are over the limit. Otherwise, only the total number of search cycles performed are displayed along with the best change to the objective function as well as the value of the objective function upon termination.

```
FINAL REPORT
INFEASIBLE SOLUTION!!
Number of constraints over:1
Total cycles:6
Best Change:0
Objective Function:
220.29
```

Figure A.3: Final Report on Termination of Search with infeasible solution

A.2 Geninit and Search

The starting points of a search may be generated using a perl script called *geninit*. It can be invoked as follows:

```
sh> geninit <num variables> <output file>
```

The program will randomly assign values to each of the $num_variables$ variables and output the assignments to the *output file*. The random assignment of values is based on an even probability of either 0 or 1 being assigned.

As *LSearch* can only perform searching from one random starting point, it is necessary to use a perl script for generating random starting points and calling LSearch on that starting point. The script *Search* performs that purpose. It can be invoked as follows:

```
sh> Search <#variables> <Constraints> <Objective> [output file]
```

The script will call on *geninit* in order to generate random starting points for the given number of variables. It will then call *LSearch* in order to perform search. It parses through the output generated by *LSearch* in order to determine whether a feasible solution is generated. If a feasible solution is generated, the script will then halt, displaying the best objective function obtained. Otherwise, it will find another random starting point and start searching again. If the *output file* is specified, the feasible solution will be written to that file.

A.3 File Formats

A.3.1 Format of Initial Conditions File

```
# This is a comment
# The next line gives the number of variables in the problem
5

# This is the initial values of the variables
# The format is
# variable_number:value

0:0
1:0
2:1
3:0
4:0
```

The variables can be specified to the program in any order, i.e. variable 10 can be assigned before variable 1. Variables which are specified more than once in the same initial conditions file will generate a warning and the last value will be used. Variables with values that are not 1 or 0 will terminate the program with an error message.

A.3.2 Format of Constraints File

```
# This is a comment
# The next two lines specify one constraint using \\ for line
# breaking
4 - 3X0 - X1 \\
- X2 - X3 - X4
# Constraint 2
1 - X1 - X2
# Constraint 3
-1 + 2X1 - X4
```

Constraints are specified in the form F(X) ≥ 0 . Constraints can be specified either in one line or with multiple lines joined using \\. Spaces are optional. Variables are specified in the form Xn. There is a bit of pre-processing done in order to determine whether a set of constraints yields any valid solutions. The minimum and maximum slack values of each constraint is calculated. If a constraint has a maximum slack value of less than 0, then no valid solution exists.

A.3.3 Format of Objective File

```
# This is a comment
-100X0
14X1
-13X2
X3
-2X4
20X1X4
```

Each term in the objective function is specified on a different line. All other terms except for the first on the line are ignored. Conjunctive terms can specified in the format $cXiXjXk$...

A.3.4 Format of Output File

The output file is in the same format as the initial conditions file without any comments in it.

Bibliography

[1] Alexander Schrijver. Totally unimodular matrices:fundamental properties and examples, *Theory of Linear and Integer Programming*, pages 266-281, 1986.

[2] Michael R. Garey & David S. Johnson. *Computers and Intractability:A Guide to the Theory of NP-Completeness*, page 245, 1979.

[3] B.W. Kernighan & S. Lin. An Efficient Heuristic Procedure for Partitioning Graphs *The Bell System Technical Journal*, pages 291-307, February 1970

[4] S. Kirkpatrick, C. D. Gelatt, Jr., & M. Vecchi. Optimization by simulated annealing. *Science 220*, pages 671-680, May 1983.