The Construction of Timetables for Secondary Schools

Bevan Arps
Supervisor: Rod Harries

October 28, 1993
Abstract

The Ultimate Principle: By definition, when you are investigating the unknown, you do not know what you will find. - Anon [4]

The generation of timetables in secondary schools has never been an easy task, and with the modern trend towards flexible curricula, this task is becoming increasingly complex and difficult.

This paper investigates some of the characteristics of the problem which cause the difficulties. A computer based approach to the solution of problems in this area has been developed which finds optimal (or near optimal) solutions in a short enough time to permit iterative development of timetables in secondary schools.
Contents

1 Introduction 5

2 What is Timetabling? 6
  2.1 Secondary School Timetables 6

3 Theory 8
  3.1 Graph Colouring 8
  3.2 Application of Graph Theory 9
  3.3 Simulated Annealing 10
  3.4 Method of Application 12

4 How do Schools do it? 13

5 Implementation 15
  5.1 Input to System 15
  5.2 Operation of Program 16
    5.2.1 Blocking 17
    5.2.2 Scheduling 18
  5.3 Solutions Produced 18
    5.3.1 Papanui High School 18
    5.3.2 Pip Scheduling 18
List of Figures

3.1 A Typical Graph, Prior to Colouring .................. 9
3.2 A Typical Graph, After Colouring ...................... 9
3.3 Pseudo Code for simulated annealing algorithm ........ 11
5.1 A Track of the Annealing Process in Action .......... 20
Chapter 1

Introduction

*Machines should Work. People should Think.* - IBM Motto [4]

The purpose of this honours project was to investigate the various methods by which a near optimal timetable\(^1\) can be produced. Special attention has been paid to finding a method of solution which will be applicable to the types and sizes of timetabling problems found in typical secondary schools.

An approach to solving this problem from the realm of graph colouring has been investigated via the production of a Macintosh application to construct timetables. This program seems to produce reasonable results in a relatively short time, a finding which is in conflict with several of the previous papers in the field. The reasons for this behaviour have been investigated.

\(^1\)Because timetabling can be shown to be a member of the class of NP-Complete problems, attempting to obtain a fully optimal timetable is a futile task.
Chapter 2

What is Timetabling?

Timetabling can be considered as the task of organising people and resources to satisfy a set of constraints. These constraints include the timing of meetings, where meetings can take place, and the availability of the people involved.

When no two meetings take place simultaneously, and when there are no constraints on availability, timetabling is a trivial task. However, this situation almost never occurs in reality. Nearly all timetables constructed in the real world are constructed observing a large set of constraints.

2.1 Secondary School Timetables

The construction of timetables for conventional secondary schools has to be completed in the face of a large number of constraints, commonly making it very difficult (and occasionally, impossible) to construct a timetable satisfying all constraints. If an optimal timetable cannot be found, the people constructing the timetable often have to make decisions about which constraints should be ignored when trying to make the timetable as good as possible.

At this point, we shall list some of the major constraining factors which influence the construction of school timetables.

- Flexibility

Unlike the situation of years past, it is characteristic for modern secondary schools to support a great deal of flexibility in the course of study that each

---

1 A *optimal timetable* is one that satisfies all the constraints given.
2 For a glimpse of the ideas common to some timetables constructed in the 1930's and 1960's, see [6], an account of some of the timetabling experiences of the headmaster of a London Boys' School.
student follows. In contrast with the older, teacher centred, approach where the timetable was produced with respect to the wishes of the teachers, the approach now is student centred, with the timetable being designed to satisfy as many students as possible.

In particular, it is common for schools to allow almost any combination of subjects to be studied, and often these subjects can be studied at whatever level the student is capable of handling.

This flexibility brings a great burden upon whoever is given the task of timetable construction. Instead of having to meet the wishes of a handful of teachers, they now have to meet the wishes of a whole school of students.

- **Time**
  
The time available for classes to occur in is often itself limited. For example, most secondary schools operate on weekdays from around 9 in the morning until 3 in the afternoon. Once lunch-times, sports afternoon(s) and other fixed breaks are removed, the timetable often has to squeeze everything into 24 hours per week.

  To add to the difficulty, some teachers may only be available at certain times (part time teachers, or specialist teachers shared by more than one school), and some rooms may also have limited availability. Some classes may be required to occur at particular times, or may require double (or triple) periods.

- **Personnel**
  
Many constraints for a timetable come from the people who will be involved in following the timetable once it is constructed. A secondary school has a limited number of teachers available for teaching any one subject. This limits the number of classes in that subject that can occur at any one time.

Similarly, a student who wishes to follow a study program consisting of a particular set of classes needs to have those classes occurring at different times.

- **Locations**
  
Deciding *where* a class is to take place has an influence on deciding *when* that class is to take place. Teaching a Mathematics class in an English room is certainly possible (even if it is less than ideal), but taking an Experimental Biology class in that same room would clearly be impossible.
Chapter 3

Theory

If you don't know what your program is supposed to do, you'd better not start writing it - Dijkstra [4]

Before a solution to the timetabling problem can be attempted, in order to facilitate reasoned thought about the problem area a suitable basis for expressing and manipulating the problem field must be discovered.

*Graph Colouring* is a well formalised field whose structures closely match the needs of timetable construction. This leads to the examination of graph colouring algorithms as possible methods of solution.

### 3.1 Graph Colouring

A graph is a collection of objects, called nodes which are connected by edges. (See Figure 3.1, page 9.) Graph colouring is the process by which each node of the graph is “painted” with a colour, in such a way that no two nodes which are connected by an edge are the same colour. (See Figure 3.2, page 9.)

Any timetabling problem can be transformed into a graph colouring problem by applying the following transformation:

- For each class, create a node
- Create an edge between any two nodes (classes) with a common *Teacher*, *Room* or *Student Group*.
- Create an edge between any two nodes (classes) which should not occur at the same time.
When this has been completed, colour the resulting graph, with each colour representing a different time.

3.2 Application of Graph Theory

The interchangeability of graph colouring and timetabling implies that any algorithm that attempts to obtain an optimal solution for a timetable will take a prohibitively long time to run on moderate to large problems. This occurs because of the intractable\(^1\) nature of the best algorithms currently known for graph colouring.

However, there is a convenient way around this problem. When trying to solve a graph colouring problem, the intent is almost always to try to find a minimal coverage – a colouring which uses as few different colours as possible.

In a timetabling situation, this is no longer appropriate. While a timetable may be required to occupy no more than a certain time-span (most commonly a week,

---

\(^1\)Tractable algorithms are routines whose running time can be expressed via a polynomial relation to the problem size. Intractable algorithms have an exponential (or worse) relationship between running time and problem size.
Monday–Friday), there is no real concern to see if it can be made to occupy a shorter
time. Instead of trying to find a optimal solution, the aim is to locate a sufficiently
good solution. Also, if no solution is sufficiently good, finding a solution which is
as good as possible is beneficial.

3.3 Simulated Annealing

There comes a time when one must stop suggesting and evaluating
new solutions, and get on with the job of analysing and finally implement­
menting one pretty good solution.
Anon [4]

After research, and consultation with Professor Marek Kubale2, the graph colour­
ing technique of Simulated Annealing with Taboo Search was selected as a good
technique for application to the problem field. Not only does simulated annealing
run in a acceptable time, but it is capable of finding good solutions when optimal
ones are not possible.

The simulated annealing technique can be explained by an analogy to particle
physics (the field which inspired the technique). Each of the elements of the
optimisation problem is treated as a particle which will tend to settle into the state
with the lowest possible energy.

However, the entire system is subject to a temperature which imparts additional
energy to all the particles, enabling them to make transitions to higher energy levels.

As the system is slowly “cooled”, the elements of the system make transitions
between differing states, and the system gradually settles into a state closer to an
“optimal” configuration.

The idea behind the success of the technique is that, unlike deterministic approaches
such as greedy search, simulated annealing is less likely to get stuck in a local
minimum – a configuration which is better than any closely related configuration,
but which is not the best possible solution.

Detailed in Figure 3.3 is pseudo code for the simulated annealing process. This
code carefully avoids many implementation issues (such as how to choose a node
to colour, and how to evaluate the quality of a solution) in order to focus on the
steps of the algorithm itself. The “Quality” rating used in the algorithm is the
result of an evaluation function which gives some measure of the quality of various
decisions.

2Professor Kubale, who visited the department early in 1993, has done a great deal of research into
the areas of graph colouring and timetabling.
Define $N$ = Set of All Nodes
Define $T$ = Queue of Taboo Nodes
Define $C$ = Set of Possible Colours
Define $temperature$ = Current Temperature of Annealing process
Define $cool$ = Function to give a lower temperature
Define $bias$ = Function to generate a "Noise" value from $temperature$

Define $n_i$ = a single node $\in N$
Define $c_j$ = a single colour $\in C$
Define $q_{ij}$ = "Quality" of setting $n_i$ to colour $c_j$
Define $r_{ij}$ = $q_{ij} +$ random noise value based on temperature

foreach $n_i \in N$
begin
    select random colour $c_j \in C$
    set $n_i$ to colour $c_j$
end

set $temperature$ to initial value
while $temperature > 0$
begin
    select $n_i \in N$ such that $n_i \notin T$
    foreach $c_j \in C$
    begin
        $q_{ij}$ = "quality" of setting $n_i$ to colour $c_j$
        $r_{ij}$ = $q_{ij} +$ bias($temperature$)
    end
    set $n_i$ to colour $c_j$ such that $n_{ij} = \max_k(n_{ik})$
    remove head from queue $T$
    append $n_i$ to queue $T$
    $temperature = cool(temperature)$
end

Figure 3.3: Pseudo Code for simulated annealing algorithm
3.4 Method of Application

The use of Simulated Annealing as a solution technique allows the system to handle more diverse situations than would otherwise be possible. Conventional graph colouring assumes that each node requires exactly one colour – with simulated annealing, this corresponds to trying to optimise a set of states where each state has one degree of freedom.

Timetabling, however, is a problem with many, interrelated, degrees of freedom, where a decision for one variable can affect the possibilities for other variables at other nodes. Effectively, the construction of a timetable is an optimisation problem in many variables.

For example, a class requires a Teacher, a Room, and Students. If a class could have any one of a set of teachers and could take place in any one of a set of rooms, then the timetabling solution process should try to allocate both of these resources; since the selection of a teacher could affect the availability of a room, the two problems should be solved together.

This requirement has lead to the introduction of a Simulated Annealing algorithm which is capable of the optimisation of two variables simultaneously.
Chapter 4

How do Schools do it?

One important step that must be undertaken before any attempt to computerise a problem is begun, is an investigation into how that problem is solved by hand.

Research into the ways that timetables are currently constructed in secondary schools has revealed a consistent approach. The process proceeds something as follows:

- Based on the choices of students, work out what classes are going to be required at what levels in which subjects.
- Combine senior school classes into columns (or blocks) of classes which can occur at the same time.
- Teachers are assigned to senior school classes by Heads of Departments. Problems at this stage can bring about alterations to the columns developed in the previous step.
- Assign teachers to junior school classes
- Schedule the columns during the school week
- Schedule junior school classes “in the gaps” left by the senior school.

This approach reduces the complexity of the problems that need to be solved at each step, making it easier to produce a solution.

Some of the policies common at secondary schools also serve to reduce the complexity of the timetabling problem that needs to be solved. Typical policies include:

- Home Rooms
Certain teachers, particularly senior teachers, may have specific rooms assigned to them in which all of their teaching takes place. The choice of the room to assign is often influenced by the rank of the teacher in the school administrative hierarchy, and also by the subject ranges to be taught by that teacher.

Home Rooms simplify the timetabling process by reducing the number of allocation problems that have to be solved – an assignment of a teacher can automatically bring with it a room for the class to take place in.

• Class Unification

For purposes such as common tests or ease of administration, all of the classes of a particular subject (such as all sixth Form Mathematics Classes) may be constrained to take place at the same time.

This brings a simplification of the timetabling problem because all the classes being unified together can be treated as a single unit, instead of many diverse units.

When the application of simulated annealing to the timetabling problem was made by this author, the standard approach, detailed in this section, of breaking the problem into smaller pieces was partially retained.

The construction of the Columns of a timetable has been stressed as of great importance. A timetable constructed with columns as an intermediate step will retain greater flexibility when compared to one created without such an intermediate step. While each of the timetables may satisfy all of the constraints required of the system equally, a columned timetable will allow a greater variation of subject choices to be taken than its counterpart.

Flexibility, one important feature of today's timetables, is something which needs to be retained. Not only is it important to be able to create a timetable in a flexible manner (so that many different constraints can be accommodated), but it is equally (if not more) important that the timetable generated is itself flexible. The wider the range of subject choices that can be studied under a timetable, the better that timetable is for a secondary school.
Chapter 5

Implementation

As a demonstration of the feasibility of the technique of simulated annealing as a solution to the problem of constructing school timetables, a Macintosh application has been written. This application has produced extremely good results for moderately sized problems in a matter of minutes.

The application accepts a specification of the requirements of the timetable for a disc file (see Section 5.1 and Appendix C), and produces a set of columns satisfying the given constraints as well as possible. It then schedules those blocks during the specified time, and produces a second disc file containing the results of its operation.

5.1 Input to System

The input to the application comes from a text data file which is parsed by a bison grammar into various internal data structures. The bison source code, together with the source code for its lexical analyser, is given in Appendix C. This parser accepts information about the times available, the classes required, and the constraints to be applied to their placement.

The constraints supported by the system are divided into two groups, to reflect the two phase nature of the program. First, we have the constraints used by the Blocking Phase, where the classes are combined into sets as detailed in the previous chapter:

- **Grouping**
  A Grouping is a set of classes which must always occur in the same column of subject choices. This can be used to organise classes to happen concurrently.

- **Distinctions**
A Distinction is a set of classes which must not be placed in the same column. This can be used to require classes to occur at different times so that it is possible for a student to study a particular combination of classes.

- **Blocking**
  A Blocking is a set of classes which should form a column. The classes in the set should be in the same column, and no other classes should occur in that column.

- **Separation**
  A Separation is a set of classes which should be divided into columns independently from other classes.

Secondly, we have the constraints available during the scheduling phase. These constraints can all be applied to *Teachers, Rooms or Classes*, and are used to control when a class can occur.

- **At Times**
  A specification of times when it is preferable for a teacher to teach, a room to be used or a class to be taught.

- **Not At**
  A specification of times when a teacher or room is unavailable, or when a class may not be taught.

- **Only At**
  A shorthand method to specify At Times for a list of times, and Not At for all other times. This provides a way to require a teacher, a class or a room to be active at exactly specified times.

- **Concurrent**
  A specification of classes which should be scheduled at the same time. If these classes are assigned to disparate columns (perhaps because they were separated into different sets of columns), the scheduler will try to make those columns occur at the same times.

- **Not With**
  A specification of classes which should not be scheduled at the same time. These classes should not be combined into the same columns, and the columns thus created will not be scheduled to occur at the same time.

### 5.2 Operation of Program

When the program is started, various windows appear on screen. One window, labelled *Timetabler* is the window where "all the action occurs". This is the
window where the process of solving the timetabling problem is visible. At the bottom of the screen will be visible the top of another window, labelled Status Messages which is used by the program to provide information to the user. Finally, there is a small window labelled Controls which provides the user with some control over the progress of the system.

The first thing a user will typically do is to move the status window to a more visible location, as its default placement is somewhat less than ideal. Secondly, a data file should be loaded (via the File menu). If there is an error in the file, some error messages will be displayed in the status menu. These messages are terse, and may be cryptic, but usually serve to help isolate the problem.

Once information has been parsed from the source data file, internal data structures are initialised and the actual task of solving the timetable can begin. This task will be accompanied by various messages in the status window which provide some indication of the program's progress.

5.2.1 Blocking

Blocking is the first phase of the solution process, and consists of trying to find sets of classes that can occur at the same time. These blocks are equivalent to the columns discussed in Chapter 4, and are generated in accordance with the constraints described in Chapter 5.

The information from the data file is combined into a central data structure used for the first phase of the program's operation - a relationships matrix showing the relationship between individual classes.

The three sorts of relationship which can hold between two classes are Don't Care, Group, and Clash. All of the class relationship constraints described in Section 5.1 can be implemented with these three types of relationship.

Once the relationships array has been created, the annealer described in Appendix E is activated to carry out the first stage of the timetabling process – the combination of classes into blocks.

These blocks represent the fundamental element of the timetabling process used by this program. All of the classes in a block are (ideally) capable of occurring at the same time, and will always be scheduled at the same time during the week.

---

1 The Load option of the File menu is the only menu option that will actually do anything useful. The others represent uncompleted work.
5.2.2 Scheduling

Once all the blocks have been found, the next phase of the problem is to schedule those blocks during the available hours. As well as all of the constraints from the first phase of the problem, all of the remaining constraints from Section 5.1 (those relating to time) can be used to control the positioning of the blocks.

Each block has an associated vector of suitability, with one element for each possible time that classes can occur. Each of these elements indicates the suitability of that time for the associated block - Forbidden, Don't Care or Good. Blocks will not be assigned to times which are forbidden, and will by preference be scheduled during times which are good.

5.3 Solutions Produced

Trials with sample data seem to indicate that the system produces high quality results in just a few minutes

5.3.1 Papanui High School

An afternoon spent with Alana Davis, a teacher from Papanui High School, produced the information detailed in Appendix A. This information is representative of the requirements of a typical large secondary school, and was constructed largely from Alana's knowledge of the problem domain.

The application produced an optimal\(^2\) columns solution to the problem in seven minutes on a Mac II.

Unfortunately, due to time constraints while at Papanui High School, full details about the time restrictions on the classes, teachers, and rooms, could not be obtained. But experience with other data sets seems to indicate that the second phase would take less time than the first, resulting in a high quality timetable for a real secondary school being produced in around ten minutes.

5.3.2 Pip Scheduling

Just because the program was designed to handle secondary schools does not imply that it is incapable of handling other problems. Detailed in Appendix B (page 34) is a set of data relevant to the scheduling of postgraduate papers within the department.

\(^2\)Satisfying all given constraints
The information in this data set is, unfortunately, incomplete due to the loss of some data from the original Pip scheduling meeting. However, it is evident from trials with this data that the program handles such a problem well.

5.4 Examination of Results

The results produced by the system have been excellent. A partially completed version of the program was able to find optimal solutions to the Blocking section of the problem in a matter of minutes, raising the expectation that the second phase of the program would work as well. This expectation has not been disappointed.

This incomplete package was instrumented to keep track of the behaviour of the system as it operated. The results generated from this instrumentation were examined after each run of the program. The graph given in Figure 5.13, is typical of the graphs produced.

One interesting characteristic of this graph is the sudden drop in the rating around temperature 2500. This drop occurs with any set of input data (at differing temperatures) and seems to represents a crystallisation of part of a solution as the cooling of the simulated annealing algorithm progresses.

Careful observation of the application as crystallisation occurred showed that it corresponded to a small number of classes being fixed into a relationship which was not altered during the remainder of the optimisation process. These classes were those for which the problem of blocking could be shown to be most difficult – despite any lack of genuine intelligence in the program, it successfully found and solved the hardest part of the problem before any other part.

Examination of the result at the point of crystallisation has revealed that the classes involved in the freezing of positions are those with a high interrelation; that the configuration chosen is optimal; and that the configuration has been generated with some apparent “forethought” for the effect the configuration will have upon later decisions.

---

3The y axis of this graph represents arbitrary values that have no unit.
4The term crystallisation has been used because of the rapidity of the process as the temperature decreases. It has been likened to the sudden solidification that can occur when super-cooled pure water suddenly turns to ice.
Figure 5.1: A Track of the Annealing Process in Action
Chapter 6

Other Work

Applying the technique of simulated annealing to the construction of school timetables has been done in the past. However, many of these past applications have been less successful than the approach detailed in this paper. The question arises "Why have previous applications been less successful?"

As described by both Abramson [1] and Dang [2], simulated annealing can be applied to the problem of constructing timetables in the following fashion:

The information supplied to the system consists of a number of tuples, each containing a Student Group, a Teacher and a Room. Solving the timetabling problem requires the mapping of these tuples onto the available times in such a way that no Class, Teacher or Room occurs more than once in each time.

While this approach is not inconsistent with the approach taken by this author, there is a substantial difference with the methods of implementation used.

As detailed in Section 3.3, the method used here is to partition the problem into two distinct phases, one of generating Blocks of classes that can occur together, and one of mapping those blocks onto the available times.

This can be contrasted with the methods detailed in [1] and [2], where no such intermediate step is used.

Both of these papers come to the conclusion that simulated annealing is a worthwhile approach to the solution of school timetable construction, but with the reservation that the technique seems to become infeasible for large (real-world) sized problems. Dang [2] goes on from this conclusion to the construction of specialised hardware for the Simulated Annealing algorithm.

The view of this author, however, is the following: not only is simulated annealing suitable for the solution of timetable construction problems, but the partition of the
problem into two distinct phases reduces the size of the problem that needs to be solved, and increases the speed at which solutions can be obtained. This speed increase seems to more than counteract the necessity of solving two problems instead of just one.
Chapter 7

Conclusions

Solving the problem of the construction of high quality timetables for secondary school environments is not an easy task, taxing both the human mind and the computing power of today's computers.

Simulated Annealing has been shown to be a feasible approach for the generation of good school timetables, and the effectiveness of a two phase approach has illustrated the importance of good application of algorithms to the solution of a problem.

Much work remains to be done in this field, some areas of which are detailed in Chapter 8, however the results obtained thus far are encouraging.
Chapter 8

Further Work

During the completion of this research, a number of areas have been identified which deserve further investigation.

8.1 Improvements to Constraints

- Handling Multiple Period Classes
  Some secondary schools run 35 or 40 minute periods instead of 60 minute ones. While this could be seen as beneficial for some subjects, such as Mathematics or Music, other subjects can be put at a disadvantage. Most obviously, science classes would be very hard pressed to carry out experiments in a 35 minute time span.

  For schools in this situation, the ability to schedule double (or perhaps even triple) periods for classes becomes important. Experiments that would be impossible in a 35 minute period become possible when a 70 minute double period is available.

- Allocating Teachers to Classes
  Currently, the system requires that the allocation of teachers to groups of students has been completed before the start of timetabling. Extending the system to allow for teacher allocation (with well defined constraints) would increase the utility of the application, as well as further reduce the workload of the people charged with constructing the timetable.

  One important concept that would need to be included to allow teacher allocation to occur, would be the need to limit the number of classes that a teacher is assigned. While there may be 24 hours in each week available for teaching, teachers would not relish the thought of teaching for all of those. Also, some teachers (such as Senior Deans and Vice Principals) may have
other commitments that do not interfere with teaching, but which do restrict the number of classes that can be effectively taught.

- **Handling Multiple Teachers and Rooms**
  The system, as it stands, works on a basis of any one class having only one teacher, and only one room. This handles most situations, but there are times when a class requires more than one teacher and/or more than one room. An extension to handle these multiplicities may alter some areas of the system, but probably not many.

- **Handling Difficult Classes**
  Some classes may be physically or mentally harrowing to teach. The provision of a rest period after each period teaching such a class could be a feature very welcomed by the teaching staff. Also, periods of preparation before particular classes, such as 7th Form Chemistry, could be useful.

- **Recognition of Scheduling Constraints during Blocking**
  The constraints used during the scheduling phase of the process are currently ignored during the prior blocking phase. Adding additional constraints to the relationships array (See Section 5.2.1) would enable the blocking to occur with some foreknowledge of the impact of various decisions on the later process.

  This would most likely improve the quality of the blocking obtained, and reduce the number of problems which would remain in a final solution (perhaps removing the need for a third processing phase as described below).

### 8.2 Improvements to Algorithm

As completed, the application proves that simulated annealing can be used to solve timetabling problems for moderate to large systems. However, there are some improvements that could be made to the algorithm to improve the final results.

- **Allow for Swapping Elements**
  The current simulated annealing algorithm only looks at moving elements to new, unoccupied, positions. This can prevent it from looking at some configurations which may be significantly better than others that are more readily seen, especially when the number of free spaces is low.

  The algorithm could be improved to allow swapping of elements, removing the need to find free spaces to move into, and allowing the system to successfully handle timetabling problems with more highly constrained systems.

- **Resolving Conflicts in Final solution**
  The final timetable produced by the system may still have conflicts in it; constraints that may not be satisfied.
One possibility for the improvement of the final result is as follows: fix the locations of every class which is in a position satisfying all constraints, and re-anneal the positions of all the classes breaking constraints.

This approach may still not produce an optimal timetable, but it cannot make the existing timetable any worse, and freeing the classes from the restriction of fitting into columns would most probably improve the final result.

- The construction of the Evaluation Functions used in the program was a purely arbitrary exercise in what was intuitively correct.

However, computer science has many problems where the best results are only intuitive once the field has some familiarity. There is nothing to say that these evaluation functions are the only (or even the best) possibilities. An investigation needs to look at the possible evaluation functions for this problem and answer the following questions:

Is there a better evaluation function? Is a slower, but better evaluation function better than a faster, but less sophisticated one? (Does the extra annealing possible with a faster function outweigh the gain of using a slower one?) Should the function remain static as the solution process progresses, or should it alter in some way as progression is made (in the same way as chess programs work harder as the endgame approaches)?

- The length of the taboo list used in the algorithm has been arbitrarily set to a fraction of the number of elements in the list.

What length is the best length for the taboo list – is it a constant, or some function of the number of elements, or of the number of constraints? Should (can) the length of the taboo list be dynamically altered to improve the simulated annealing process, to aid in the finding of near optimal solutions?

### 8.3 Other Applications

Another area in which this work could be extended would be into the field of constructing university timetables. These timetables are much larger than school timetables, and have different properties. While they are almost certainly as tightly constrained as school timetables, these constraints may be structured in significantly different ways.

The scheduling of final examinations is another problem which could be subject to solution by the techniques detailed here. Each year, many University and Bursary students face examination clashes requiring elaborate administration and organisation to deal with. The construction of timetables that are free from such difficulties may not be an impossible task.

Another application could be to the scheduling of post graduate lectures. All of the people involved in such courses can have limited availability, and the accessibility of resources can be severely constrained. Even though the number of people
involved is small, the problem of scheduling postgraduate lectures is still one of moderate to large complexity. The application of the techniques of this project to lecture scheduling is an important area to consider.
Acknowledgements

I am indebted to Mr Rod Harries for his supervision of the project, for his proofing of this report, and for his insightful recollection of many of the problems faced in real life by the compilers of school timetables.

Many thanks to Professor Marek Kubale, whose experience with graph colouring and timetabling greatly assisted my research.

Thanks also to Henry Dang for his provision of a copy of his paper [2] at short notice; and to Maurice Cook and Alana Davis from Papanui High for their comments, information and time.

Finally, a heartfelt thank you to Gwenda Bensemann, a Classmate who proofread this report, and also helped me retain my sanity during an incredibly hard year of work.
Appendix A

Sample Timetabling Data from Papanui High

timetable mon900
NUMBLOCKS 6

; Accounting, Economics and Business Studies
; Subject Commerce 4
Class Accy7 Cn ; Accounting
Class Accy6 Cr
Class Accy51 Cr
Class Accy52 Cn

Class Eco7 Cn ; Economics
Class Eco6 Vx
Class Eco5 Cn

Class Bus6 At ; Business Studies,

; Make it possible to study Accy, Econ & Bus at the same time
Distinct Accy7 Eco7
Distinct Accy6 Eco6 Bus6
Distinct Accy51 Accy52 Eco5

; Art
; Subject Art 4
Class ArtH7 Gd ; Art History
Class Paint7 Gd ; Painting
Class Art6 Gd ; Art
Class Art5 Gd

; Make it possible to study Art History and Painting together
Distinct ArtH7 Paint7

; Biology
; Subject Biology 4
Class Bio7 Gz
Class Bio61 Pz
Class Bio62 Sh
Distinct Bio61 Bio62  ; Biology classes separate
;
; Computer Studies
;
Subject Computers 4
Class Comp61 Bt
Class Comp62 Bt
;
; English and Classics
;
Subject English 4
Class Eng71 Ln
Class Eng72 Rs
Class Eng73 Dv
Class Cla7 Mt
;
Class Eng61 Dv
Class Eng62 Dn
Class Eng63 Jn
Class Eng64 Ln
Class Eng65 Fr

Class Eng51 Dv  ; School Certificate
Class Eng52 Ln
Class Eng53 Rs
class Eng54 Me
Class Eng55 Dn
Class Eng56 Fr

Distinct Eng71 Eng72 Eng73 Cla77
Distinct Eng61 Eng62 Eng63 Eng64 Eng65 Eng66
;
; History and Geography
;
Subject HistGeo 4
Class Hist7 Mt
Class Geo6 Mm
Class Geo51 Ca
Class Geo52 Mm
Class Geo53 Ld
Class Hist51 Mt
Class Hist52 Me

Distinct Hist7 Geo7
Distinct Hist51 Hist52
Distinct Geo51 Geo52 Geo53
;
; Home Economics/Home Science
;
Subject HEc 4
Class HEc6Gr7 Rb
Class HEc51 Pa
Class HEc52 Rb

Distinct HEc51 HEc52
;
; Journalism
;
Subject Journalism 4
Class Jour6 Fr

;
; Languages
Subject Language 4
Class Jap1 Xx
Class Jap6 Xx
Class Fren5 Mv
Class Macri5 Wa
Class Jap5 Xx

Distinct Fren5 Macri5 Jap5

Subject Music 4
Class Mus7 Ws
Class Mus6 Ca
Class Mus5 Wa

Distinct Mus7 Mus6 Mus6

Subject Science 4
Class Sci51 Gr
Class Sci52 Hr
Class Sci53 Sr
Class Sci54 Fw
Class Sci55 Sh
Class Sci56 Py


Subject TdrawGraph 4
Class Des7 Sw
Class Des6 Gf
Class Graph7 Gf
Class Tech051 Da
Class Tech052 Gf

Distinct Des7 Graph7
Distinct Des6 Graph6
Distinct Tech05 Tech052

Subject Tourism 4
Class Tour61 Rs
Class Tour62 Rs

Subject Maths 4
Class Mac71 Bj
Class Mac72 Pt
Class Mac71 Tp
Class Mac72 Tp
Class Mac73 Bt

Class Math61 Po
Class Math62 Tp
Class Math63 Bt
Class Math64 Pt
Class Math51 Bt
Class Math52 Bt
Class Math53 Pt

; Japanese - Teacher unknown
; French
; Maori
; Japanese
; Music
; General Science

; Design
; Graphics
; Technical Drawing
; Tourism
; Maths with Calculus, Bursary
; Maths with Stats, Bursary

31
Class Math54 Po
Class Math55 Tp
Class Math56 Ck

Distinct Mac71 Mac72 ; Calculus classes separate
Distinct Mas71 Mas72 Mas73 ; Statistics classes separate

Group Math61 Math62 Math63 Math64 AMathe; Maths classes together

; Subject Photography 4
Class Pho7 Br
Class Pho6 Br

; Subject Physical and Outdoor Education 4
Class 0Ed7 Da
Class PEdc7 Sm
Class 0Ed6 Da
Class PEdc6 Mr

Class PEd251 Fk
Class PEd252 Sm
Class PEd253 Md
Class PEd254 Sm
Class PEd255 Fk
Class PEd256 Mr

Distinct 0Ed6 PEd36 ; Outdoor classes separate

; Subject Physics 4
Class Phy71 Sr ; Physics
Class Phy72 Py
Class Phys61 Py
Class Phys62 Sr

Subject Chemistry 4
Class Chem7 Hl ; Chemistry
Class Chem61 H
Class Chem62 Op

; Allow a student to do either Physics Class and Chemistry as well
Distinct Phy67 Chem7 Phy72

; Keep Physics and Chemistry classes separate

; Subject Typing 4
Class Typ7 Vk
Class Typ6 Vk
Class Typ51 At
Class Typ52 Vk

Distinct Typ51 Typ52

; Workshop

Subject Workshop 4
Class WSTech Ow ; Workshop Technology
; Fifth Form Six Subject Class
; Subject SixSubject5Form 4
Class Eng5s Mr ; English
Class Math5s La ; Mathematics
Class PHe5s Mr ; Physical Education
Class Sci5s Gr ; General Science
; Keep Six Subject classes separate
Distinct Math5s Eng5s Sci5s PHe5s

; Sixth Form Six Subject Class
; Subject SixSubject6Form 4
Class Eng6s Mr ; 6 Subject English
; Alternative Fifth Form Subjects
; Subject AltFifth 4
Class AEng5 Dn ; Alternative English
Class AMath5 Ck ; Alternative Mathematics
Class ASci5 Gr ; Alternative General Science
Class AHe6 Rb ; Alternative Home Economics
Class APhys Ed Fk ; Alternative Phys. Ed.
Class ATyp5 Vk ; Alternative Typing
Class AWorkshop Ow ; Alternative Workshop
Distinct AEng5 AMath5 ASci5

; Alternative Sixth Form Subjects
; Subject AltSixth 4
Class AHe6 Rb ; Alternative Home Ec
Class AMath6 Bt ; Alternative Mathematics
Distinct AHe6 AMath6

; Other Constraints on System
;
; Make it possible to do a full Science/Maths course in 7th Form
Distinct Phys7 Chem7 Bio7 Mac71 Mac72
; 5th Form Maths & English occur in two blocks
Group Eng51 Eng52 Eng53 Math54 Math55 Math56
Group Math51 Math52 Math53 Eng54 Eng55 Eng56
; Home oriented classes separate
Distinct AHe6 AHe6

; Notes
;
; Many of the Distinctions made between classes are made to increase
; the generality of the timetable produced by increasing the number
; of course combinations available.
Appendix B

Sample Timetabling Data for Pip Scheduling

Timetable

<table>
<thead>
<tr>
<th>Monday</th>
<th>Tuesday</th>
<th>Wednesday</th>
<th>Thursday</th>
<th>Friday</th>
</tr>
</thead>
<tbody>
<tr>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
<td>9</td>
</tr>
<tr>
<td>12</td>
<td>13</td>
<td>14</td>
<td>15</td>
<td>16</td>
</tr>
<tr>
<td>17</td>
<td>18</td>
<td>19</td>
<td>20</td>
<td>21</td>
</tr>
</tbody>
</table>

Gaps for: Cosec 303 lectures (tue9, fri11) and Seminar (tue10 tue11)

NumBlocks 6

Subject Pips 2
Class Cosec801 Bh
Class Cosec802 Jp
Class Cosec805 Fa
Class Cosec806 Wk
Class Cosec808 Bm
Class Cosec814 Ac

Distinct Cosec801 Cosec802 Cosec805 Cosec806 Cosec808 Cosec814

; 4th Year Students

; Mark Alexander
Schedule Cosec802
NotAt mon11 tue11 thu11 fri11
mon3 tue3 wed3 thu3 fri3

; Bevan Arps
Schedule Cosec801 Cosec805 Cosec808 Cosec814
NotAt tue2 tue3 wed12 thu2 thu3

; Gwenda Bensemann
Schedule Cosc805 Cosc802 Cosc814
NotAt thu9 thu10 thu11 thu12

; Craig Farrow
Schedule Cosc802 Cosc805 Cosc808
NotAt wed2 wed3

; Richard Jones
Schedule Cosc802 Cosc805 Cosc808 Cosc814
NotAt mon12

; Masters Students

; Mike Hardie
Schedule Cosc814
NotAt mon9 mon10 mon11 mon12 mon2 mon3 mon4
tue9 tue10 tue11 tue12 tue2 tue3 tue4

; Staff

; Paul Ashton
Schedule Cosc805
NotAt mon11 tue12 thu9 fri12 fri13

; Andy Cockburn
Schedule Cosc814
NotAt thu9 fri13

; Rod Harries
Schedule Cosc805
NotAt tue10 tue11 tue12 tue2
thu9 thu10 thu11 thu12
fri9 fri10 fri11 fri12

; Bruce McKenzie
Schedule Cosc808
NotAt mon12
tue10 tue11
wed2 wed3 wed4
fri2 fri3

; John Penny
Schedule Cosc802
AtTimes mon2 tue2 wed2 thu2 fri2
NotAt mon9 tue9 wed9 thu9 fri19
mon10 tue10 wed10 thu10 fri10
mon11 tue11 wed11 thu11 fri11
mon12 tue12 wed12 thu12 fri12
Appendix C

Source code for Parser and Lexer

/*
 * Lexical Analyser for Timetabler
 */

#include "PCommonTimetabler_common.h" /* Common */
#include "Common_Timetabler_common.h" /* Common */
#include "y.tab.h"
#include <stdlib.h>
#include "parserutils.h"
%

/* Comment Characters */

/* Character Groupings */

A [Aa]  
B [Bb]  
C [Cc]  
D [Dd]  
E [Ee]  
F [Ff]  
G [Gg]  
H [Hh]  
I [Ii]  
J [Jj]  
K [Kk]  
L [Ll]  
M [Mm]  
N [Nn]  
O [Oo]  
P [Pp]  
Q [Qq]  
R [Rr]  
S [Ss]  
T [Tt]  
36
union {
  Identifier text;
  int num;
  void *data;
} 

token CLASS_SYM
	token DISTINCT_SYM
	token GROUP_SYM
	token SEPARATE_SYM
	token SUBJECT_SYM
	token TIMETABLE_SYM
	token COLON_SYM
	token BLOCKS_SYM
	token BLOCK_SYM
	token CONCUR_SYM
	token AT_SYM NOTAT_SYM ONLYAT_SYM
	token SCHEDULE_SYM
	token <text> IDENTIFIER

type <data> class

type <data> IDList

type <data> ttable times entry timelimit

%
/*
 * Overall input language is one or more pieces of "Information"
 */

inputspecs : timetable inputinfo

inputinfo : inputinfo information

/*
 * Definition of Timetable times
 */
timetable : TIMETABLE_SYM ttable

| LTmes=$2;
| NurnPeriods=List_Size(LTmes);
|
| ttable : ttable IDENTIFIER

| List_append(copystring($2),$$);
| $$=01;
| IDENTIFIER

| $$=List_create();
| List_append(copystring($1),$$);
|}

/*
 * A Piece of "Information" is one of the following types:
 * + A timetabling restriction
 * + Specification of a Subject group of classes
 * + Some classes that must occur together
*/
Some Classes that must not occur together
Some classes to group independently from the others

information : schedule  
  subject  
  block  
  group  
  distinction  
  separation  
  blocks  
  Concurrent  

Scheduling Constraints

schedule : SCHEDULE_SYM IDList timelimit  
  element = make_constraint($2,$3);  
  List_add(element,LConstraint);  

  timelimit : timelimit timeentry  
    $2=merge_avail($1,$2);  
    free($1);  
    free($2);  
    timelimit  
      timeentry  
        $2=$1;  

  timeentry : AT_SYM times  
    $2=set_avail($2,GREAT,AVERAGE);  
    NOTAT_SYM times  
      $2=set_avail($2,BAD,AVERAGE);  
    ONLYAT_SYM times  
      $2=set_avail($2,GREAT,BAD);  

  times : IDList  

subject : SUBJECT_SYM IDENTIFIER NUMBER  
  classes  
    List_add(make_subject($2,$3,$4),  
    LSubjects);  

classes : classes class  
  $5=$1;  
  List_add($2,$5);  

A Block of Classes
* Like a group, but other classes CANNOT occur
* Shorthand for GROUPing and making DISTINCT from everyone else
/
block : BLOCK_SYM IDList
   { List_add($2, LBlocks); }
/
Concurrent Classes
* Classes which should occur at the same time -- no constraint on
* being in same group
/
Concurrent : CONCUR_SYM IDList
   { List_add($2, LConcurrent); }
/
A Group of classes which must occur together
* GROUP <classlist>
/
group : GROUP_SYM IDList
   { List_add($2, LGroups); }
/
A Group of classes which must not occur together
* DISTINCT <classlist>
/
distinction : DISTINCT_SYM IDList
   { List_add($2, LDistinct); }
/
A Group of classes to group separately
* SEPARATE <classlist>
/
separation : SEPARATE_SYM IDList
   { List_add($2, LSeparate); }
/
Number of blocks to combine classes into
* BLOCKS <number>
/
blocks : BLOCKS_SYM NUMBER
   { numgroups=$2; }

40
A Class from a Subject Grouping

* CLASS <name> <teacher> <room>

/*

A list of Identifiers
*/

IDList: IDList IDENTIFIER
    | IDENTIFIER
    | { List_add(copystring($2),$1);
    |   $$=$1;
    |   $$ = List_create();
    |   List_add(copystring($1),$$);
    |}
Appendix D

Source code for Timetable Processor

D.1 Operation

This module consists of the core of the timetabling process. The annealer (See Appendix E, page 61) is the core of the solution process, but this is the module which provides the annealer with what it needs to "get on with the job".

It works by stepping through a number of phases of activity, each of which accomplishes a different piece of the solution. Operation begins when a call to ProcessLaunch() starts the ball rolling. The steps of operation, and their purposes are as follows:

- **Blocker_Init**
  Initialise the data structures of the blocking phase. This includes translation of system constraints into internal forms.

- **Blocker_Place**
  The placement of all the classes known to the system onto the annealers' solution grid.

- **Blocker_Anneal**
  Finding a solution to the construction of columns. This simply involves letting the annealer get on with the job.

- **Schedule_Init**
  Take the results of the Blocker.Anneal step and use them to generate the constraints and internal data structures for the solution of the Scheduling problem.
• **Schedule.Place**
  Place the blocks generated by the system onto the annealers' solution grid for scheduling. Each block can occur multiple times, to allow for multiple meetings of the classes.

• **Schedule.Anneal**
  Finding a solution to the Scheduling of blocks.

• **Reporting**
  Saving a report detailing the results of program operation.

### D.2 Source Code
/*
 * Processor for Timetabling
 * This code oversees the actual process of creating the timetable
 * The process consists of a number of phases, each of which needs to be completed
 * before the next can begin.
 */

/*
 * Defines for Phase Names
 */
#define READY 0
#define BLOCK_INIT 1
#define BLOCK_PLACE 2
#define BLOCK_ANNEAL 3
#define SCHEDULE_INIT 4
#define SCHEDULE_PLACE 5
#define SCHEDULE_ANNEAL 6
#define REPORT 7
#define DONE 8

/*
 * Defines for Relationships between Classes
 */
#define NOREL 0
#define GROUP 1
#define CLASH 2

/*
 * Defines for Penalties for Breaking Relationships
 */
#define GROUP_PENALTY 100
#define CLASH_PENALTY 100

/*
 * Define Weightings for Popularity of Times
 */
#define AVERAGE_LOAD 10
#define GREAT_LOAD 50

/*
 * Function Prototypes
 */
void ProcessLaunch(void);
void Process(int EventID);
void Blocker_Init(void);
void Blocker_Place(void);
void Blocker_Aanneal(void);
void Schedule_Init(void);

44
void Schedule.Place(void);
void Schedule.Anneal(void);
void Timetable_Report(void);

void makeRelationship(List members, List allelements, int retype);
void makeSeparation(List members, List allelements);

long Blocker.eval(int x, int y, void *Class, int instance);
void Blocker.set(int x, int y, void *Class, int instance);

vector make_availvector(vector q);

long Schedule.eval(int x, int y, void *blockptr, int instance);
void Schedule.set(int x, int y, void *blockptr, int instance);
/*
  * Processor for Timetabling
  * This code oversees the actual process of creating the timetable
  * The process consists of a number of phases, each of which needs to be completed
  * before the next can begin.
  */

/*
  * Standard Includes
  */
#include <stdlib.h>
#include <stdio.h>

/*
  * Marksman Generated includes
  */
#include "Common_Timetabler_.h"
#include "PCommonTimetabler_.h"
#include "EventsTimetabler_.h"
#include "Controls.h"
#include "PUtils_Timetabler_.h"

/*
  * Our Own Includes
  */
#include "LList.h"
#include "Vectors.h"
#include "Processor.h"
#include "parserutils.h"
#include "Annealer.h"
#include "Utility.h"

/* Debugging Defines */
#define TRACE
#if defined TRACE
#define TRACK(X) fprintf(stderr,X)
#else
#define TRACK(X)
#endif

/*
  * Global Variables
  */
static int phase = READY; /* Which Phase of processing are we in? */
static int (*rels[][I]); /* Dynamically sized relationship array for relationships between
  * classes and blocks */
static int (*olds[][I]); /* Dynamically sized relationship array for relationships between
  * classes while generating blocks */
static int numclasses; /* Number of Classes - initialised & never altered */
static int numteachers; /* Number of Teachers - initialised & never altered */
static vector timeratings;  // Vector containing "popularity" weighting for each time slot */
/* (Weighting is higher for timeslots rated "GOOD" for more blocks) */

/*******************************************************************************/
/* Launcher */
/* * Start the processing of a timetable */

void ProcessLaunch()
{
    TRACK("Process Launched\n");
    phase = BLOCK_INIT;
}

/*******************************************************************************/
/* Despatcher */
/* * This routine triggers one step of the current process on demand from */
/* * the Event Handler. The static-global variable "phase" records which */
/* * phase of the process we are currently within. */

void Process(int EventID)
{
    switch (phase) {
    case READY:
        break;  // Do Nothing!

    case BLOCK_INIT:
        Blocker_InitO;
        break;  // Initialise Blocker Data Structures

    case BLOCK_PLACE:
        Blocker_PlaceO;
        break;  // Place classes onto grid first time

    case BLOCK_ANNEAL:
        Blocker_AnnealO;
        break;  // Work on finding a better solution

    case SCHEDULE_INIT:
        Schedule_InitO;
        break;  // Initialise Data structures for Scheduler

    case SCHEDULE_PLACE:
        Schedule_PlaceO;
        break;  // Place Blocks onto Grid

    case SCHEDULE_ANNEAL:
        Schedule_AnnealO;
        break;  // Work on finding better Timetable

    case REPORT:
        Timetable_ReportO;
        break;  // Generate report on results of operation

    case DONE:
        break;  // We're finished
    }
default:
    fprintf(stderr,"Internal Error: Unknown Phase number within Dispatcher.\n");
    break;
}
Add_UserEvent(CONTROL,STEPDONE, 0, nil); /* Say "Step Done" to Control window */

/*******************************************************************************/
/* Initialise Blocker Data Structures for Use */
/*******************************************************************************/

void Blocker_Init() {
    int i, j;
    List_Posn scanner;
    int a, b, c;
    int altered;
    static int step = 0;
    switch (step) {
        case 0:
            TRACK("Initialising Blocker\n");
            TRACK("Size Variables\n"); /* Initialise Size variables for later quick reference */
            numclasses = List_Size(LClasses);
            numteachers = List_Size(LTeachers);
            step = 1;
            break;
        case 1:
            List_Sort(LClasses);
            List_Sort(LTeachers);
            List_Sort(LRooms);
            step = 2;
            break;
        case 2:
            TRACK("Relationships array\n"); /* Initialise Relationships array */
            rel = makearray(numclasses, numclasses, NOREL);
            step = 3;
            break;
        case 3:
            TRACK("Separations\n"); /* Put SEPARATION information into Relationships Array */
            List_Marker(&scanner, LSeparate);
            List rewind(&scanner);
            while (listerror == LIST_NOERROR) {
                makeseparation(List_getdata(&scanner), LClasses);
                List next(&scanner);
            }
            step = 4;
            break;
        case 4:


TRACK(" Groups
"); /* Put Grouping Information into Relationships Array */
List_Marker(&scanner, LGroups);
List_rewind(&scanner);
while (listerror == LIST.NOERROR) {
    makerelationship(List_getdata(&scanner), LClasses, GROUP);
    List_next(&scanner);
} step = 5;
break;

case 5:
    TRACK(" Distinctions
"); /* Put Distinction Information into Relationships Array */
    List_Marker(&scanner, LDistinct);
    List_rewind(&scanner);
    while (listerror == LIST.NOERROR) {
        makerelationship(List_getdata(&scanner), LClasses, CLASH);
        List_next(&scanner);
    } step = 6;
    break;

case 6:
    TRACK(" Teacher Clashes
"); /* Add additional Distinctions for Classes taught by same teacher */
    List_Marker(&scanner, LTeachers);
    List_rewind(&scanner);
    while (listerror == LIST.NOERROR) {
        makerelationship(((Teacher) List_getdata(&scanner))->classes, LClasses, CLASH);
        List_next(&scanner);
    } step = 7;
    break;

case 7:
    TRACK(" Room Clashes
"); /* Add additional Distinctions for Classes in the Same room */
    List_Marker(&scanner, LRooms);
    List_rewind(&scanner);
    while (listerror == LIST.NOERROR) {
        makerelationship(((Room) List_getdata(&scanner))->classes, LClasses, CLASH);
        List_next(&scanner);
    } step = 8;
    break;

case 8: /* Generate Inferred Relationships */

/***************************************************************************/
* Transitivity by Groupings:
* If Class A is grouped with Class B,
* and Class A has relationship R to Class C,
* then Class B must also have relationship R to Class C.
* *
* Reflexivity:
* *
* If Class A has relationship R to Class B,
* then Class B must have relationship R to Class A */

49
```c
TRACK(" Propagation.\n");
altered = false;
for (a = 0; a < numclasses; a++) {
    /* Check all Classes A */
    for (b = 0; b < numclasses; b++) {
        /* Check relationships that A has to other Classes */
        if (((*(rel)[b])[a] != NOREL) {
            if (((*(rel)[a])[b] != *(rel)[b][a]) {
                /* Make reflexive relationship if not already present */
                (*(rel)[a])[b] = *(rel)[b][a];
                altered = true;
            }
        }
        if (((*(rel)[b])[a] == GROUP) {
            /* Classes A and B are Grouped - propagate other relationships */
            for (c = 0; c < numclasses; c++) {
                /* Relationship between A and C */
                if (((*(rel)[c])[a] != NOREL) {
                    if (((*(rel)[a])[c] != *(rel)[b][c]) {
                        /* Make relationship between B and C */
                        *(rel)[b][c] = *(rel)[a][c];
                        altered = true;
                    }
                }
            }
            /* If nothing changed, go to next step */
            if (altered)
            step = 9;
            /* If something changed, we get called again, to continue the propagation */
            break;
            case 9:
            step = 10;
            /* Unused - hook for debugging */
            break;
            case 10:
            TRACK(" Done!\n");
            phase = BLOCK_PLACE;
            break;
        }
    }
}

void Blocker_Place() {
    Identifier name;
    int i;
    List_Psm scanner;
    Class thisclass;
    TRACK("Placement\n");
    TRACK(" Block list\n");
    for (i = 0; i < numgroups; i++) {
        /* Create list of blocks we are trying to create */
        sprintf(name, "Block \%d", i);
        List_append(make_block(name), LFinalBlocks);
    }
    phase = BLOCK_FLACE;
    TRACK(" Finished!\n");
}```
TRACK(" Setting up Annealer\n");

setup_anealer(LFinalBlocks, LTeachers, Blocker.eval, Blocker_set);

TRACK(" Placing Classes\n");

I* Scan through all classes, placing them onto the grid */
List Marker(&scanner, LClasses);
List rewind(&scanner);
while (listerror == LIST.NOERROR) {
    thisclass = List_getdata(&scanner);
    put_element(thisclass, thisclass->name, makeboolvector(numgroups, true),
                makevector(thisclass->teachers, LTeachers), 0);
    List.next(&scanner);
}

TRACK(" Starting Annealer\n");

I* Start Annealer working on problem */

I* Use Annealer to find (near) optimal blocking */

void Blocker_Aanneal() {
    if (step_anealer() == true) {
        phase = SCHEDULE_INT;
        Running = false;
    }
}

I* Initialise Scheduler to produce final solution */

void Schedule_Init() {
    I* Blocking has been worked out as required - now need to do scheduling */
    List Posn scanner, scanner2;
    TRACK(" Setting Up Scheduler\n");

I* We fill in the Membership of Classes to Blocks as determined by just */
    completed Annealing process */

TRACK(" Block Memberships\n");
List Marker(&scanner, LClasses);
List rewind(&scanner);
while (listerror == LIST.NOERROR) {
    Block block;
    Class class;
    List Marker(&scanner, LClasses);
    List rewind(&scanner);
    while (listerror == LIST.NOERROR) {
        Block block;
        Class class;
class = List_getdata(&scanner); /* Get record for class */
block = List_getelement(class->groupnum, LFinalBlocks); /* Get Block record to add class to */
List_append(class, block->members); /* Add class to appropriate block */
List.next(&scanner);
}

TRACK(" Scanning Time Constraints\n"); /* Scan through time constraints to implement them */

List.Marker(&scanner, LConstraint);
List.rewind(&scanner);
while (listerror == LIST_NOERROR) {
    TimeConstraint tc;
    tc = List_getdata(&scanner); /* Get the next constraint */
    List.Marker(&scanner2, tc->members); /* Scan through members of constraint */
    List.rewind(&scanner2);
    while (listerror == LIST_NOERROR) {
        char *name;
        void *data;
        name = List_getdata(&scanner2); /* Get name of next member */
        /* Work out what sort of member it is, and include this constraint for that member */
        if (data = List_Find(name, LClasses)) != nil) {
            /* Class Identifier */
            combine.avail(((Class) data)->avail, tc->avail);
        } else if (data = List_Find(name, LTeachers)) != nil) {
            /* Teacher Identifier */
            combine.avail(((Teacher) data)->avail, tc->avail);
        } else if (data = List_Find(name, LRooms)) != nil) {
            /* Room Identifier */
            combine.avail(((Room) data)->avail, tc->avail);
        } else {
            /* Unrecognised Member of constraint */
            fprintf(stderr, "Unrecognised Identifier %s\n", name);
        }
        List.next(&scanner2);
    }
    List.next(&scanner);
}

TRACK(" Generating Class Time Constraints\n"); /* Now to generate Time constraints for Classes */

List.Marker(&scanner, LClasses);
List.rewind(&scanner); /* Scan through all Classes */
while (listerror == LIST_NOERROR) {
    Class thisclass;
    Teacher teacher;
    Room room;

    /* Add Teachers Time constraints to the class */
    thisclass = List_getdata(&scanner);
    teacher = List_getelement(thisclass->teachernum, LTeachers);
combine_avail(thisclass->avail, teacher->avail);

/* Need to add code to handle Room availability as well */
List_next(&scanner);

TRACK(" Generating Block Time Constraints
");

/* Time constraints for Classes have been generated – can now
* generate same for Blocks */

List rewind(&scanner); /* Scan through all classes */
while (listerror == LIST_NOERROR) {
    Class thisclass;
    Block thisblock;
    int i;

    thisclass = List_getdata(&scanner); /* Get class record */
    thisblock = List_getelement(thisclass->groupnum, LFinalBlocks);
    /* Get record for block this class belongs inside */

    combine_avail(thisblock->avail, thisclass->avail); /* Add classes availability to the blocks */
    List_next(&scanner);
}

TRACK(" Evaluating Block occurrence rates
");
/* Work out how often each block has to occur */

List rewind(&scanner); /* Scan through all classes, checking number of meetings needed */
while (listerror == LIST_NOERROR) {
    Class thisclass;
    Block thisblock;

    thisclass = List_getdata(&scanner);
    thisblock = List_getelement(thisclass->groupnum, LFinalBlocks);
    thisblock->repetitions = max(thisblock->repetitions, thisclass->meetings);

    List_next(&scanner);
}

/* Now to create "popularity" vector for times – to help evaluation
function. The idea is to make it "less good" to use a time which is
* available for a lot of classes – if there is a time which is only
* available for one class, then that class should occur at that time to
* reduce conflicts with other classes */

TRACK(" Creating Popularity Vector
");
timeratings = make_avail(O); /* Create appropriately sized array with zero totals */

List Marker(&scanner, LFinalBlocks); /* Scan through all defined blocks */
List rewind(&scanner);
while (listerror == LIST_NOERROR) {
    int i;
    Block thisblock;

    thisblock = List_getdata(&scanner); /* Get record for next block */
    for (i = 0; i < NumPeriods; i++) {

53
For each time slot, add penalty based on how good that time is

```c
/* for this block */

if ((*thisblock->avail)[i] == AVERAGE)
    (*timeratings)[i] += AVERAGE_LOAD;
else if ((*thisblock->avail)[i] == GREAT)
    (*timeratings)[i] += GREAT_LOAD;
```

```c
List_next(&scanner);
```

```c
/*
  * Normalise "popularity" vector to zero base. This (hopefully) makes it
  * possible to obtain a zero (perfect) solution for schedule
*/

{  
    int i;
    int rn;

    rn = (*timeratings)[0];
    /\ find time with lowest rating \/
    for (i = 1; i < NumPeriods; i++) {
        rn = min(rn, (*timeratings)[i]);
    }
    /\ And subtract that lowest value from all ratings \/
    for (i = 0; i < NumPeriods; i++) {
        (*timeratings)[i] -= rn;
    }
}
```

```
TRACK(" Creating Block relationship grid
");
/**
 * Create relationship grid for groups
*/

{  
    int i, j;
    Class thisclass;
    Class thatclass;

    oldrel = rel;
    /\ Keep a copy of class relationships grid \/
    rel = makearray(numgroups, numgroups, 0);
    /\ And create grid for group relationships \/
    for (i = 0; i < numclasses; i++) {
        thisclass = List_getelement(i, LClasses);
        /\ Scan through all classes i \/
        for (j = 0; j < numclasses; j++) {
            thatclass = List_getelement(j, LClasses);
            /\ And through all classes j \/
            if ((*(*oldrel)[i])j == CLASH) /* If classes i and j CLASHed */
                (*rel)[thisclass->groupnum][thatclass->groupnum] = CLASH;
            /* Then mark CLASH between their groups as well */
        }
    }
    phase = SCHEDULE.PLACE;
    /\ Activate next phase \/
}
```

```
/**
 * Place blocks onto Annealing grid and activate annealer
 */

void Schedule_Place0()
{
    /\ Place Blocks onto Initial Grid \/
```
LisLPosn scanner; /* Scanner for list traversals */

TRACK("Schedule - Placement\n"); /* Announce ourselves with debugging info */
TRACK(" Setup Annealer\n");

setup_annealer(LFinalBlocks, LTunes, Schedule..eval, Schedule.set); /* Setup Annealing Process */

TRACK(" Initialising Block Coordinate Vectors\n");

for (List..Marker(&scanner, LFinalBlocks);
    listerror == LIST\n          NOERROR;
    List..next(&scanner)) {
    Block thisblock;
    int i;

    thisblock = List..getdata(&scanner); /* Get pointer to structure */
    thisthblock->x = calloc(thisblock->repetitions, sizeof(int)); /* Allocate Memory */
    thisthblock->y = calloc(thisblock->repetitions, sizeof(int));

    for (i = 0; i < thisthblock->repetitions; i++) {
        (*thisthblock->x)[i] = -1;
        (*thisthblock->y)[i] = -1;
    }
}

TRACK(" Placing Blocks on Grid\n");

for (List..Marker(&scanner, LFinalBlocks);
    listerror == LIST\n          NOERROR) {
    Block thisblock;
    int i;

    thisblock = List..getdata(&scanner); /* Get block record */
    for (i = 0; i < thisblock->repetitions; i++) {
        put_element(thisblock, thisthblock->name, make_element_vector(thisthblock->name, LFinalBlocks), make..availvector(thisblock->avail), i);
        List..next(&scanner);
    }
}

TRACK(" Start Annealer\n"); /* Finish initialisation of Annealer */
start_annealer();

TRACK(" Ready to Anneal\n"); /* Trigger next phase */
phase = SCHEDULE\n        ANNEAL;

/*-----------------------------------------------
* Find solution for Scheduling of Blocks
*/

void Schedule_Anneal() {
    if (step..annealer()) /* Wait until Annealer says "Finished" */
        \n
phase = REPORT; /* And start reporting */

/*********************************************/
/* Generate Report on Solution */
/*********************************************/
void Timetable_Report()
{
    FILE *report; /* File handle for report generation */
    int i; /* General loop variable */
    List_Push scanner; /* Scanner for List Traversals */

    TRACK("Generating Report on Result\n");
    report = fopen("result", "w"); /* Open Results file */

    /* Display Block allocations of Classes */
    List_Marker(&scanner, LClasses);
    fprintf(report, "Allocations of Classes to Blocks\n");
    for (List.rewind(&scanner);
         listerror == LIST_NOERROR;
         List.next(&scanner)) {
        Class thisclass;
        /* Generate report on one class */
        thisclass = List_getdata(&scanner);
        fprintf(report, "%-*s%2d\n", MAXIDENTIFIER, thisclass->name, thisclass->groupnum);
    }

    /* Display Time Allocations of Blocks */
    List_Marker(&scanner, LFinalBlocks);
    fprintf(report, "Scheduling of Blocks\n");
    for (List.rewind(&scanner);
         listerror == LIST_NOERROR;
         List.next(&scanner)) {
        Block thisblock;
        /* Pointer to Block structure */
        int i; /* General loop variable */

        /* Generate report on one Block */
        thisblock = List_getdata(&scanner);
        for (i = 0; i < thisblock->repetitions; i++) {
            char *time;
            time = List_getelement((*(thisblock->y)[i]), LTunes);
            fprintf(report, "%-*s%-*s\n", MAXIDENTIFIER, thisblock->name, MAXIDENTIFIER, time);
        }
        fprintf(report, \n);
    }
    fclose(report);
    fprintf(stderr, "Done.\n");
    phase = DONE;
}

/*****************************************************/
/** Utility Routines */

/**
 * Given a list of elements, and a master list from which these elements are
 * drawn, fill in the relationships array with a CLASH between every element in
 * the list and every element not in the list.
 */
void makeseparation(List members, List allelements)
{
    List_Posn scanner;
    List_Posn scanner2;
    List_Marker(&scanner, members);
    List_Marker(&scanner2, allelements);
    List_rewind(&scanner);
    while (listerror == LIST_NOERROR) {
        int p1 = List_MemberNum(List_getdata(&scanner), allelements);
        if (p1 != -1) {
            List_rewind(&scanner2);
            while (listerror == LIST_NOERROR) {
                int p2;
                p2 = List_MemberNum(List_getdata(&scanner2), allelements);
                if (p2 != -1) {
                    if (List_Member(List_getdata(&scanner2), members) { /*(+(rel)(p1)(p2) = CLASH; */
                        *(rel)[p1][p2] = CLASH;
                    }
                    List_next(&scanner2);
                }
                List_next(&scanner);
            }
        }
    }
}

/**********************************************************/

/**
 * Given a list of elements (members), a master list of elements from which
 * they were drawn (allelements) and a relationship code (reltype) fill in the
 * relationships array (rel) with that relationship between every pair of
 * elements in that list
 */
void makerelationship(List members, List allelements, int reltype)
{
    List_Posn scanner;
    List_Posn scanner2;
    List_Marker(&scanner, members);
    List_Marker(&scanner2, members);
    List_rewind(&scanner);
    while (listerror == LIST_NOERROR) {
        int p1;
        ...
if ((pl = LisLMemberNum(LisLgetdata(&scanner), allelements)) \neq -1) {
    List.rewind(&scanner2);
    while (listerror == LIST.NOERROR) {
        int p2;
        if ((p2 = LisLMemberNum(LisLgetdata(&scanner2), allelements)) \neq -1) {
            (*(*rel)[pl][p2] = reltype;
        }
        List.next(&scanner2);
    }
    List.next(&scanner);
}

//*******************************************************************************
/*
   * Evaluation function for dividing classes into Blocks
*/

long Blocker.eval(int x, int y, void *classptr, int instance)
{
    int posn;
    int i;
    int tmp;
    List_posn scanner,
    long rating = 0;
    /* Optimum rating until we find otherwise */
    posn = List.MemberNum(classptr,LClasses);
    List.Marker(&scanner, LClasses);
    /* Scan through all other classes */
    listerror == LIST.NOERROR) {
        if (posn \neq i) {
            tmp = (*(*rel)[posn][i]);
            if (tmp == CLASH) {
                Class tmpclass;
                tmpclass = List.getdata(&scanner);
                if (tmpclass->groupnum == x)
                    rating += CLASH.PENALTY;
            } else if (tmp == GROUP) {
                if (((Class) List.getdata(&scanner))->groupnum == x)
                    rating += GROUP.PENALTY; /* Check for violated GROUPING */
            }
        }
        List.next(&scanner);
        i++;
    }
    return rating;
}

/*******************************************************************************/
/* Designation function for Blocker.
*/
/* Set location of a Class. X value == which Block; Y value == which teacher */

58
/**

void Blocker_set(int x, int y, void *classptr, int instance)
{
    ((Class) classptr)->groupnum = x;
    ((Class) classptr)->teachernum = y;
}

/* Make Availability Vector */

vector make_avail(vector q)
{
    vector newvector;
    int i;
    newvector = make_avail(true);
    for (i = 0; i < NumPeriods; i++) {
        if ((q[i]) == BAD)
            (*newvector)[i] = false;
    }
    return newvector;
}

/* Evaluation Function for Scheduler */

long Schedule_eval(int x, int y, void *blockptr, int instance)
{
    int i,
        p,
        q;
    List_Posn scanner;
    Block curblock;
    long rating = 0;
    curblock = blockptr;
    q = List_MemberNum(curblock, LFinalBlocks);
    if (q == -1)
        fprintf(stderr, "%s
" curblock--narnue);
    else
    {
        List_Marker(&scanner, LFinalBlocks);
        List_rewind(&scanner);
        p = 0;
        while (listerror == LIST_NOERROR)
        {
            Block thisblock;
            thisblock = List_getdata(&scanner);
            for (i = 0; i < thisblock--repetitions; i++)
            {
                if ((thisblock--y)[i] == y)
                {
                    if ((p != q) && ((*(rel)[p])[q] == CLASH))
                        rating += CLASH_PENALTY;
                }
                List_next(&scanner);
            }
        }
    }
    return rating;
}
p++;  
};
}

rating += (*timeratings)[y];  /* Add popularity rating so we avoid popular times */
return rating;
}

/***********************************************************************/
/* Designation function for Scheduler */
/***********************************************************************/

void Schedule.set(int x, int y, void *blockptr, int instance)
{  
    Block curblock;
    
    curblock = blockptr;
    (*curblock->x)[instance] = x;
    (*curblock->y)[instance] = y;
}  

Appendix E

Source code for Simulated Annealing Implementation

E.1 Usage

Because of the need to use Simulated Annealing to solve more than one optimisation problem within the timetabling program, the code to carry out the task has been separated into a separate module. The module is not reentrant – only one Simulated Annealing run can take place at a time, but serves to simplify the calling routines greatly.

As the program initialises, a single call to `init_annealer()` primes the module for action. When a Simulated Annealing problem arises, the solution process begins as follows:

- A call to `setup_annealer()` is made to define the problem domain, and provide two functions needed by the annealing algorithm.
  
The problem domain is defined by two Lists of identifiers, each of which represents one degree of freedom. The timetabling application, for example, uses Blocks and Teachers as the two domains for optimisation.

Two support functions are also required. The `evaluation` function, which returns an evaluation of the distance of a position from an optimal solution\(^1\). Similarly the `designation` function is used by the annealer designate which grid position has been assigned to an element. This function is used to communicate with the `evaluation` function so that it is aware of the current locations of elements when working out the rating of a position.

\(^1\)This is much like a quality rating, except that higher numbers are worse, not better.
• A series of calls to `put_element()` to supply the elements to be placed upon the solution grid.
  Each element is placed on the solution grid in the best location available when it is initially supplied. No more than one element can be present at any one grid location.

• One call to `start_annealer()` to signal the end of placement
  At this time, the annealer works out various internal parameters, such as the start temperature, and prepares itself for work.

• Repeated calls to `step_annealer()` to carry out the annealing process
  Each call to `step_annealer()` makes one change to the current solution, as required by the simulated annealing algorithm. When the temperature of the system reaches zero, or when the rating becomes zero (representing an optimal solution), the function returns `true` to say "All Done". Otherwise, it returns `false`.

After the annealer has finally returned `true` to a call to `step_annealer`, the calling application can use the positioning information previously supplied by calls to its designation function as required.

E.2 Operation

The algorithm given for simulated annealing in Figure 3.3 is not suitable for general implementation because it leaves several questions about the operation of the program unanswered. What follows is a list of the answers used in this particular implementation:

• The method used to choose the next node to re-colour
  This implementation works by selecting the highest rated node (the node in the worst position) which is not on the taboo list, and for which a better position exists.

  Behind this decision was the observation that it is often (although not always) better to optimise the elements in worse positions before those in better positions.

  To ensure that this approach doesn’t ignore nodes that are in bad positions, but which apparently have low ratings, a code segment at the start of `step_annealer()` cycles through all the nodes, updating their ratings.

  Without this additional code, the ratings stored for each element would always the rating at the time the element was placed in its current location, which would have been rendered inaccurate because of later placements. This code ensures that any ratings cannot be older than some well defined limit.
The taboo list works on time, not on nodes

The conventional way to implement the taboo list is as a fixed length queue of elements. For reasons of efficiency, this annealer maintains an internal clock which keeps track of the passage of time. Each element is stamped with the time of its last evaluation, and elements altered within tabootime are not examined.

One side effect of this is that any elements which are evaluated during the same clock tick will occupy the same position on the taboo list, temporarily increasing the size of the list. (See the evaluation loop within step_annealer(), and the clock setting within place_element().)

This can occur when an element is chosen for movement, and does not get moved because it is already in an "optimal" location. The implementation cycles at this point to find another element to move, and continues to do so until a successful movement is achieved. All of the elements considered during such a cycle end up with the same time-stamp, and therefore occur at the same point in the taboo queue.

E.3 Source Code
/*
 * Annealer
 * Abstract Data Type of the mechanism to implement Simulated Annealing
 * Optimization
 */

#define MAXELEMENTS 1000
#define MAXROWS 200
#define MAXTABOO 1000
#define COOLRATE 11

j

void init_annealer(void);
void setup_annealer(List columnnames,
   List rownames, long (*eval)(int x, int y, void *elem, int instance),
   void (*desig)(int x, int y, void *elem, int instance));

void put_element(void *element, char *name, vector goodcol, vector goodrow, int instance);

void start_annealer(void);
int step_annealer(void);

/* Macro Functions */

#define randint(n) (rand()*(n)/RAND_MAX)
/* Generate a random number from 0 ... n-1 */
/*
 * Annealer
 * Abstract Data Type of the mechanism to implement Simulated Annealing
 * Optimization
 */

#include <stdlib.h>
#include <stdio.h>
#include "LList.h"
#include "Vectors.h"
#include "Annealer.h"
#include "PCommonTimetabler...h"
#include "CommonTimetabler...h"
#include "PUtilsTimetabler...h"
#include "Utility.h"

/*
 * Specialised Types
 */

struct ElementSt {
  void *element;
  char *name;
  int x,
     y;
  int instance;
  vector goodcol;
  vector goodrow;
  long clock;
  long rating;
};

typedef struct ElementSt *Element;

/*
 * Global Variables
 */

static long (*evaluate)(int x, int y, void *elem, int instance);
  /* Pointer to evaluation Function to use */
static void (*designate)(int x, int y, void *elem, int instance);
  /* Pointer to designation Function to use */
static long rating;
static int tabootime;
static int numcol;
static int numrow;
static long clock;
static long temperature;
static Element elements[MAXELEMENTS];
static int numelements;

65
static int (*(*placements)[ ]) [ ];  // Placements grid
static int traceon = false;  // Flag to control debugging trace

/* Local Function Prototypes */
static int place_element(int which_element);  // Move an element to a better grid square
static long addheat(long rating);  // Add heat energy to a rating
static void cool(void);  // Decrease temperature one notch

/* Functions */

/* Initialise the Annealer prior to first use */
void init_annealer()
{
    placements = nil;
    evaluate = nil;
    designate = nil;
}

/* Begin setup of an Annealing Run */
void setup_annealer(List columnnames,
    List rownames,
    long (*eval)(int x, int y, void *elem, int instance),
    void (*desig)(int x, int y, void *elem, int instance))
{
    Rect rView,
    Bounds;
    Point CellSize;
    List_Posn scanner;
    int i,
    j;

    /* Initialise */
    numelements = 0;
    rating = 0;
    evaluate = eval;
    designate = desig;
    clock = 0;
    temperature = 0;
    numcol = List_Size(columnnames);
    numrow = List_Size(rownames);

    /* Destroy any existing List in Timetabler Window and create a new one to our requirements */
    if (ScreenList != nil) {
        /* Destroy old list (if any) */
    }
}
LDispose(ScreenList);
  ScreenList = nil;
}
Add_UserEvent(UserEvenLOpen_Window, Res_W_Tunetabler, 0, 0, nil);
/* Send Event so window is redrawn with new list */
SetPort(WPtr_Tunetabler);  /* Set Graphics Port TextPos(geneva); * Set Font and Size */
TextSize(10);
SetRect(&rBounds, 0, 0, numcol + 1, numrow + 1);  /* Set bounds for new list */
CellSize.h = 0;
CellSize.v = 0;
rView = WPtr_Tunetabler->portRect;  /* Autosize cells */
rView.right = 15;
rView.bottom = 15;
ScreenList = LNew(&rView, &rBounds, CellSize, 0, WPtr_Tunetabler, TRUE, TRUE, TRUE, TRUE);
/* Create the List */
(*ScreenList)->selFlags = OnlyOne;  /* Set constraints on List behaviour */
LActivate(FALSE, ScreenList);
LDoDraw(TRUE, ScreenList);
/* Now setup initial display of list with Column and Row labels */
fprintf(stderr, "Adding Column Labels\n");
ListMarker(&scanner, columnnames);
List.rewind(&scanner);
for (i = 1; i <= numcol; i++) {
    LMySetCell(i, 0, List_getdata(&scanner), ScreenList);
    LViewNext(&scanner);
}
fprintf(stderr, "Adding Row Labels\n");
ListMarker(&scanner, rownames);
for (i = 1; i <= numrow; i++) {
    LMySetCell(0, i, List_getdata(&scanner), ScreenList);
    LViewNext(&scanner);
}
fprintf(stderr, "Creating Placements Grid\n");
placements = calloc(numrow, sizeof(void *));
for (i = 0; i < numrow; i++) {
    (*placements)[i] = calloc(numcol, sizeof(int));
    for (j = 0; j < numcol; j++)
        (*placements)[i][j] = -1;
}
/* Put an element onto the Placement Grid */
void put_element(void *element,
    char *name,
    vector goodcol,
    vector goodrow,
    int instance)  /* Pointer to data for element we are placing */
    /* Name to display for element */
    /* Which columns are good */
    /* Which rows are good */
    /* Which instance of the element is this? */
Element newelement;
newelement = malloc(sizeof(struct Element.St)); /* Allocate space for information */
if (newelement == nil) {
    /* Did we get memory? */
    fprintf(stderr, "WARNING: Memory Allocation failed -- cannot proceed./n"); /* Display error message */
    return;
}
newelement->element = element; /* Remember location of element ... */
newelement->name = copystring(name); /* ... name ... */
newelement->goodcol = goodcol; /* ... good columns ... */
newelement->goodrow = goodrow; /* ... good rows ... */
newelement->clock = clock; /* ... when it was added. */
newelement->instance = instance; /* ... and its instance number */
newelement->rating = 0; /* Start with no location */
newelement->x = -1;
newelement->y = -1;

elements[numelements] = newelement; /* Store element in array */
place_element(numelements); /* Make initial placement of element */
numelements++; /* Increase count of elements on grid */
clock++; /* Increase count of events ocurred */

/***********************************************************/
/* Finish the Setup of the Annealer and trigger operation */

void start_annealer()
{
    int i;
    long r;
    Element elem;
    long maxrating = 0;

    /* Work out correct current rating (== total of individual element ratings) */
    fprintf(stderr, "Calculating total rating\n");
    for (i = 0; i < numelements; i++) {
        elem = elements[i];
        r = evaluate(elem->x, elem->y, elem->element, elem->instance);
        rating = rating - (elem->rating) + r; /* Update Global Rating */
        elem->rating = r; /* Store position rating */
        maxrating = max(r, maxrating); /* Store new maximum rating */
        elem->clock = randint(clock); /* Set random time for taboo list */
    }
    fprintf(stderr, "Done.\n");

    /* Work out initial start temperature - cf. Code in addheat() */
    temperature = maxrating * 10000; /* Hot enough to overcome max rating + 10 */
    tabootime = numelements / 2; /* Set initial Taboo length to something reasonable */

    /* Display some initial information to the User */
    fprintf(stderr, "Maximum Rating: %ld Initial Temperature:%ld -\nTabooTime:%d\n", maxrating, temperature, tabootime);
}
```c
int step_annealer()
{
    int whichelement,
        bestelement;
    int done;
    int timer,
        i;
    static int (*checked)[n] = nil;

    /* find element to move - want to avoid elements moved in previous tabootime
    * clock ticks
    */

    /* We sample a number of elements, looking for the highest rated element that
    * isn't taboo, but which can be moved */

    if (checked == nil) {
        checked = calloc(numelements, sizeof(int));
        if (checked == nil) {
            fprintf(stderr, "WARNING: Memory Allocation failed -- cannot
            Create Flags array at first call */
        }
    } 

    for (i = 0; i < numelements; i++) {
        /* Mark all elements on Taboo list as already checked */
        (*checked)[i] = ((elements[i] + clock + tabootime) > clock);
    }

    /* Keep Looping until we move an element to a new position, or timer expires */

    done = false;
    timer = numelements * 2;
    while (!done) {
        bestelement = -1;
        /* Find unchecked element with highest (worst) rating */
        for (i = 0; i < numelements; i++) {
            if ((checked[i]) {
                if ((bestelement == -1) || (elements[i] + rating > elements[bestelement] + rating)) {
                    bestelement = i;
                }
            }
        }

        if (bestelement == -1) {
            /* If needed, select an element at random to check */
            bestelement = randint(numelements);
        }

        (*checked)[bestelement] = true; /* Mark element as checked */
        done = place_element(bestelement); /* Try to move it - done==TRUE if class moved */
```
if (--timer == 0) done = true;
}
clock++;
cool();
traceon = false;

if (clock % 25 == 0) {
  fprintf(stderr, "\n")
  if (rating == 0) {
    fprintf(stderr, "\n")
      if (temperature == 0) {
        fprintf(stderr, "\n")
        printf();
        printf(stderr, "\n")
        return false;
      }
    else
      return true;
  }
  fprintf(stderr, "\n")
  if (temperature == 0) {
    fprintf(stderr, "\n")
    printf();
    printf(stderr, "\n")
    return true;
  }
}

static int place_element(int which_element)
{
  int i,
  int moved;
  long bestrating = -11;
  long br;
  int x,
  y;
  Element elem;
  long thisrating;
  long ir;

  static int curelem = 0;

  if (--timer == 0) done = true;
}

if (clock % 25 == 0) {
  if (clock % 25 == 0) {
    if (temperature == 0) {
      fprintf(stderr, "\n")
      printf();
      printf(stderr, "\n")
      return false;
    }
    else
      return true;
  }
  fprintf(stderr, "\n")
  if (temperature == 0) {
    fprintf(stderr, "\n")
    printf();
    printf(stderr, "\n")
    return true;
  }
}

/***********************************************************/
/* Look at placement of referenced element and possibly move it to a new location.
*/
/* Returns TRUE if element moved, FALSE if it did not */

/*****************************/

If (numelements > 0) {
    curelem = (curelem + 1) % numelements;
    elem = elements[curelem];
    thisrating = evaluate(elem->x, elem->y);
    // Recalculate rating for element as at this position
    elem->element, elem->instance);
    rating = elem->rating;
    elem->rating = thisrating;
    rating += elem->rating;
}

for (i = 0; i < numcol; i++) {
    if ((elem->goodcol == nil)
        || ((elem->goodcol)[i] == true)) {
        // Column i is legal to use
        for (j = 0; j < numrow; j++) {
            if ((elem->goodrow == nil)
                || ((elem->goodrow)[j] == true)) {
                // Row j is legal to use
                if (((*(*placements)[i][j] == whichelement)) { // This Cell position legal to use
                    thisrating = evaluate(i, j, elem->element, elem->instance);
                    // Check quality of moving here
                    tr = addheat(thisrating);
                    if ((tr < hr)
                        && (bestrating == -1)) {
                        x = i;
                        y = j;
                        bestrating = thisrating;
                        br = tr;
                    }
                }
            }
        }
    }
}

if (bestrating != -1) {
    rating = rating - (elem->rating) + bestrating;
    elem->rating = bestrating;
    elem->clock = clock;
}

if ((elem->x != x) || (elem->y != y)) {
    // New position - remember it
}
if ((elem->x ≠ -1) && (elem->y ≠ -1)) {
    (*(*placements)[elem->y][elem->x] = 1; /* Clear flag on placement grid */
    LMySetCell(elem->x + 1, elem->y + 1, "", ScreenList);
    /* Clear display at old position */
    elem->x = x;
    elem->y = y;
    (*(*placements)[elem->y][elem->x] = whichElement; /* Set flag on placement grid */
    LMySetCell(elem->x + 1, elem->y + 1, elem->name, ScreenList);
    /* Make display at new position */
    designate(elem->x, elem->y, elem->element, elem->instance);
    /* Call designation routine to store new position */
    moved = true; /* Yes - we did move it */
    } else {
        moved = false; /* No - we didn’t move it */ Best location is current position */
    } else {
        moved = false; /* We didn’t find anything to move */
    }
    return moved;
}

passport

Given a rating value, add “energy” to it based on the current temperature */

static long addheat(long rating)
{
    long dv;
    dv = randint(temperature / 100); /* Deviation is number from range [0, temperature/100) */
    return rating + dv;
}

passport

Cool temperature one notch */

static void coolO()
{
    if (temperature > 0) /* Only decrease temperature if > 0 */
        temperature -= (temperature * COOLRATE / 1000) + 1;
}

passport
Appendix F

Source code for Linked List
Abstract Data Type

F.1 Usage

The Annealer given in Appendix E makes extensive use of the following Linked List package, which is presented to make the Annealer source code more intelligible.

F.2 Source Code
/*
 * Linked List Creation/Manipulation/Destruction Routines - Header File
 * Written By Bevan Arps
 */

#pragma once

/*
 * List Element
 *
struct List.El {
    void *data;
    struct List.El *next;
    struct List.El *prev;
};

typedef struct List.El *List_Ptr;

/*
 * List Header
 *
struct List_St {
    List_Ptr first;
    List_Ptr last;
    int membercount;
    void *(*index)[];
};

typedef struct List_St *List;

/*
 * List Marker
 *
struct List_Ps {
    List list;
    List_Ptr member;
};

typedef struct List_Ps List_Pom;

/*
 * Function Prototypes
 *
List List_create(void);
int List_add(void *data, List list);
int List_insert(void *data, List list);
int List_append(void *data, List list);
int List_destroy(List list);
List List_join(List list1, List list2);
void *List_Find(void *data, List list);
Boolean List_Member(void *data, List list);
*/

74
List List_Sort(List list); /* Sort a list in place */
List List_Copy(List list); /* Make a copy of a list */
int List_Size(List list); /* How big is a List? */

void List_Marker(List_Psn *psn, List list); /* Create a Marker into a List */
int List_rewind(List_Psn *psn); /* Rewind a position marker to the start of a list */
int List_forward(List_Psn *psn); /* Fast Forward a position marker to the end of a list */
int List_next(List_Psn *psn); /* Step a position marker to the next item */
int List_prev(List_Psn *psn); /* Step a position marker to the previous item */
int List_setdata(List_Psn *psn, void *newdata); /* Set the data pointed to by a position Marker */
int List_Sortcompare(const void *ell, const void *e1); /* Comparison routine for List Elements */

void *List_getelement(int posn, List list); /* Get an element given its position in the List */
int List_MemberNum(void *data, List list); /* Find an element number within a list */

List List_storestring(char *data); /* Create a single item list with a copy of the passed string */

/* Error Codes */
#define LIST_NOERROR 0
#define LIST_OUTOFMEMORY 1
#define LIST_EMPTY 2
#define LIST_EOF 3
#define LIST_EOF 4

/* * Error Variable */
extern int listerror;

/* * Macro Functions */
#define List_getdata(posn) (*((posn)->member)->data) /* Find the data pointed to by a position marker */

/* List.getdata implemented as Macro to improve speed of usage */
/*
 * Linked List Creation/Manipulation/Destruction Routines - Implementation
 * Written By Bevan Arps
 *
 * Notes: This is an implementation of an extensive Linked List library. Lists can
 * be created/modified and destroyed. Markers within lists can be similarly
 * dealt with, and lists can be indexed for speed if necessary.
 *
 * All routines which Search or Sort a Linked List assume that the pointers stored
 * within elements can be treated as char * pointers. This does not limit use
 * of these facilities to char strings, but simply requires that the first
 * member of any structure added to a Linked list be a suitable char array.
 *
*/

#include <stdlib.h>
#include <string.h>
#include "LList.h"
#include <stdio.h>

/*
 * These routines have been debugged - ensure that the debugging
 * instrumentation is not active.
 */
#if defined DEBUG
#undef DEBUG
#endif

/*
 * Status variable - value indicates error condition of last call to a Linked
 * List Routine
 */

int listerror = LIST.NOERROR;    /* No error when we begin */

/*************************************************************/
/* Create a New List */
/* */
List List_create(void)
{
    List newlist;
    /* Temporary variable */

    newlist = (List) malloc(sizeof(struct List_St));    /* Try to get memory for new List */
    if (newlist == nil) {
        listerror = LIST.OUTOFMEMORY;    /* Failed to Allocate Memory */
    } else {
        /* No Error */
        listerror = LIST.NOERROR;    /* Succeeded in getting memory */
        newlist->first = nil;
        newlist->last = nil;
        newlist->membercount = 0;
        newlist->index = nil;
    }
    return newlist;
}
Add a new item to the start of a list

Returns error code. (Also available in listerror)

```c
int List_add(void *newdata, List list) {
    List_Ptr newelement;
    newelement = (List_Ptr) malloc(sizeof(struct List.El));
    if (newelement == nil) {
        listerror = LIST_OUTOFMEMORY;
    } else {
        newelement->prev = nil;
        newelement->next = list->first;
        newelement->data = newdata;
        if (list->first != nil) {
            list->first->prev = newelement;
            list->first = newelement;
        } else {
            list->last = newelement;
            list->membercount += 1;
            listerror = LIST_NOERROR;
        }
        if (list->index != nil) {
            free(list->index);
            list->index = nil;
        }
    }
    list->membercount ++;
    listerror = LIST_NOERROR;
    return listerror;
}
```

Add a new item into position in a Sorted List

Returns error code. (Also available in listerror)

```c
int List_insert(void *newdata, List list) {
    List_Ptr newelement;
    List_Ptr oldelement;
    newelement = (List_Ptr) malloc(sizeof(struct List.El));
    if (newelement == nil) {
        listerror = LIST_OUTOFMEMORY;
    } else {
        oldelement = list->first;
        while (oldelement != nil) {
            if (strcmp(newdata, oldelement->data) < 0) {
                newelement->prev = oldelement;
                oldelement->prev = newelement;
                newelement->next = oldelement;
                listerror = LIST_NOERROR;
                return listerror;
            }
            oldelement = oldelement->next;
        }
        listerror = LIST_OUTOFMEMORY;
        oldelement->next = newelement;
        newelement->prev = oldelement;
        listerror = LIST_NOERROR;
        return listerror;
    }
    return listerror;
}
```
if (strcmp(newdata, oldelement->data) < 0)
    break;
else
    oldelement = oldelement->next;
);

/* Initialise new element */
newelement->data = newdata;
newelement->next = oldelement;

/* Look at where insertion is to take place */
if (oldelement == nil) {
    /* Insert at end of List */
    newelement->prev = list->last;
    list->last->next = newelement;
    list->last = newelement;
    if (list->first == NULL) /* If list was empty, then new element */
        list->first = newelement;
} else {
    /* Insert between existing elements */
    newelement->prev = oldelement->prev;
    oldelement->prev->next = newelement;
}

if (list->index != nil) {
    /* Discard any existing index */
    free(list->index);
    list->index = nil;
} /* As it is now out of date */

list->membercount++;
/* Increase membercount */
listerror = LIST_NOERROR;
/* No problems */

return listerror;
}

/* Add a new item to the end of an existing list *
* Returns error code. (Also available in listerror) */

int List_append(void *newdata, List list)
{
    List_Ptr newelement;
    /* Temporary Variable */

    newelement = (List_Ptr) malloc(sizeof(struct List_El));
    /* Try to allocate Memory for the new element */
    if (newelement == nil) { /* Failed to get memory */
        listerror = LIST_OUTOFMEMORY;
    } else {
        /* Succeeded in Getting Memory */
        /* Initialise new Element */
        newelement->prev = list->last;
        newelement->next = nil;
        newelement->data = newdata;

        /* Set pointer of last element in list if list is not empty */
        if (list->last != nil) 
            list->last->next = newelement;

    } /* Succeeded in allocating Memory */

    /* Insert at end of List */
    newelement->prev = list->last;
    list->last->next = newelement;
    list->last = newelement;
    if (list->first == NULL) 
        list->first = newelement;

/* Increase membercount */
    list->membercount++;
    listerror = LIST_NOERROR;
/* No problems */

    return listerror;
}
I* Set end of List pointer */
list->last = newelement;

if (list->first == nil) { /* If list was empty ... */
  list->first = newelement; /* set pointer to start of list as well */
  list->membercount++; /* Increase member count */
  listerror = LIST_NOERROR; /* No Problems */
}

if (list->index != nil) { /* Discard any existing index */
  free(list->index); /* (As it is now out of date) */
  list->index = nil;
}

return listerror;
}

/***********************************************************************/
* Destroy a List
* * Returns error code. (Also available in listerror)
*/

int List_destroy(List list) {
  List_Ptr listposn;

  /* First destroy elements of list */
  listposn = list->first; /* Start at head of list */
  while (listposn != nil) {
    List_Ptr newposn;

    newposn = listposn->next; /* Remember next position ... */
    free(listposn); /* Free this position ... */
    listposn = newposn; /* ... and move to next position */
  }

  if (list->index != nil) { /* Discard Index (if any) */
    free(list->index);
  }
  free(list); /* Discard List head */
  listerror = LIST_NOERROR; /* No problems */

  return listerror;
}

/***********************************************************************/
* Join two lists together
* * Returns new list head
*/

List List_join(List list1, List list2) {
  List result;

  /* Handle trivial cases */

if (list1 == nil)
result = list2;
else if (list2 == nil)
result = list1;
else if ((list1->membercount) == 0) {
/* First list has no members - result is second list */
result = list2;
free(list1);
}
else if (list2->membercount == 0) {
/* Second list has no members - result is first list */
result = list1;
free(list2);
} else {
/* Nontrivial Case - actually need to join two lists */
list1->last->next = list2->first;
list2->first->prev = list1->last;
list1->last = list2->last;
list1->membercount += list2->membercount;
free(list2);
result = list1;
}
if (result->index != nil) {
free(result->index);
result->index = nil;
}
return result;

**************************************************************************
* Find an element from a list *
* Returns pointer to data for matching element, or NULL *
**************************************************************************

void *List_Find(void *data, List list)
{
/* Is there an index we can use? */
if (list->index == nil) {
/* No Index */
List_Posn scanner;
List_Marker(&scanner, list);
List_rewind(&scanner);
/* Use linear search to find a matching element */
while (listerror == LIST.NOERROR) {
if (strcmp(data, List_getdata(&scanner)) == 0)
break;
List_next(&scanner);
}
/* Find pointer to data as required */
if (listerror == LIST.NOERROR)
return List_getdata(&scanner);
else
return nil;
} else {
/* Index available */
void **result;

*i* Use Binary Search *
result = bsearch(data, list->index, list->membercount, sizeof(void *), &List_Sortcompare);
return result;

********************************************************************************
* Find out whether something is a member of the List or not *
********************************************************************************

Boolean ListMember(void *data, List list)
{
    return (ListFind(data, list) != nil); /* Try to find it - if pointer is non-null, element exists */
}

********************************************************************************
* How big is a List? *
********************************************************************************

int List_Size(List list)
{
    return list->membercount;
}

********************************************************************************
* Make a copy of a list *
********************************************************************************

List List_Copy(List list)
{
    List newlist;
    ListPosn scanner;

    /* Make new list */
    newlist = List_create();
    if (newlist == nil)
        return nil;

    /* Scan through old list */
    ListMarker(&scanner, list);
    List rew(ind(&scanner);
    while (listerror == LIST_NOERROR) {
        ListAppend(List_getdata(&scanner), newlist); /* Add next element from old list to new */
        if (listerror == LIST_NOERROR)
            List.next(&scanner); /* Step to next element in old list */
    }

    if (listerror == LIST_OUTOFMEMORY) { /* If something went wrong ... */
        List_destroy(newlist);
        newlist = nil;
    }
    return newlist;
}

********************************************************************************
* Work out what element number an element is

81
/*
 * Returns: Reference number or -1 if not present
 */

int List::MemberNum(void *data, List list)
{
    int posn = -1; /* If we don't find it, -1 is our answer */

    /* Is there an Index to use? */
    if (list->index == nil) {
        /* No Index */
        List::Posn scanner;
        int i = 0;

        /* Scan through list linearly until we run out of list, or we find it */
        List::Marker(&scanner, list);
        List::Rewind(&scanner);
        while (listerror == LIST::NOERROR) {
            /* Found it! */
            posn = i;
            break;
        }
        i++;
        List::Next(&scanner);
    } else {
        /* We have an Index */
        void **ptr;

        /* Use Binary Search */
        ptr = bsearch(data, list->index, list->membercount, sizeof(void *), &List::Sortcompare);

        /* Calculate answer to return */
        if (ptr == nil) {
            return -1;
        } else {
            return ptr - ((void**) (list->index)); /* Use C's pointer arithmetic tricks */
        }
    }
}

/********************************************************
 * Comparison routine for Searching & Sorting
 */

int List::Sortcompare(const void *el1, const void *el2)
{
    return strcmp(((char**) el1), ((char**) el2));
}

/********************************************************
 * Sort Linked List
 */

List List::Sort(List list)
{
    void *(*listpointers)[8];
    int i;

    82
lisLPosn listpos;

/* Allocate memory for index */
if ((listpointers = malloc(list->membercount * sizeof(void *))) == nil) {
    /* Failed to allocate memory */
    listerror = LIST.OUTOFMEMORY;
    return nil;
}
/* Scan through list filling in elements of index array */
List.Marker(&listpos, list);
List.rewind(&listpos);
for (i = 0; i < list->membercount; i++) {
    (*listpointers)[i] = LisLgetdata(&listpos);
    if (List.JJext(&listpos) == LIST.EOF)
        break;
}
/* Sort index array */
qsort(*listpointers, list->membercount, sizeof(void *), &List.Sortcompare);
/* Alter element pointers to reflect sorted order */
List.rewind(&listpos);
for (i = 0; i < list->membercount; i++) {
    List_setdata(&listpos, (*listpointers)[i]);
    if (List.JJext(&listpos) == LIST.EOF)
        break;
}
/* Discard any existing index */
if (list->index != nil) {
    free(list->index);
}
/* Keep track of index to speed later searches */
list->index = listpointers;
return list;

/**************************************************************/
/* Create a Marker to traverse a list */
/* Space for Marker is provided by caller; usually on Stack frame since */
/* Markers are usually shortlived, */
**************************************************************/
void List_Marker(List_Psn *newposition, List list) {
    newposition->list = list;
    newposition->member = list->first;
    listerror = LIST.NOERROR;
}
/**************************************************************/
/* Rewind a List Marker to the start of its list */
/* Returns error code. (Also available in listerror) */
```c
int List_rewind(List_Posn * posn)
{
    posn->member = posn->list->first;
    if (posn->member == nil)
        listerror = LIST_EOF; /* Return LIST_EOF if list is empty */
    else
        listerror = LIST_NOERROR;

    return listerror;
}

/*****************************************************************************/
/* Fast Forward a List Marker to the end of its list */
/* * Returns error code. (Also available in listerror) */
/*****************************************************************************/
int List_fforward(List_Posn * posn)
{
    posn->member = posn->list->last;
    if (posn->member == nil)
        listerror = LIST_BOF; /* Return LIST_BOF if list is empty */
    else
        listerror = LIST_NOERROR;

    return listerror;
}

/*****************************************************************************/
/* Step Forward a List Marker towards the end of its list */
/* * Returns error code. (Also available in listerror) */
/*****************************************************************************/
int List_next(List_Posn * posn)
{
    if (posn->member->next == nil)
        listerror = LIST_EOF; /* Return LIST_EOF if we run out of list */
    else
    {
        posn->member = posn->member->next;
        listerror = LIST_NOERROR;
    }

    return listerror;
}

/*****************************************************************************/
/* Step Backward a List Marker towards the start of its list */
/* * Returns error code. (Also available in listerror) */
/*****************************************************************************/
int List_prev(List_Posn * posn)
{
    if (posn->member->prev == nil)
        listerror = LIST_BOF; /* Return LIST_BOF if we run out of list */
    else
    {
        posn->member = posn->member->prev;
     */
```
listerror = LIST.NOERROR;
}

return listerror;
}

/**
 * Set the data pointed to by the Element currently pointed to by a Marker
 * Returns error code. (Also available in listerror)
 */

int List_setdata(List_Fosn *posn, void *newdata)
{
  posn->member->data = newdata;
  listerror = LIST.NOERROR;
  return 0;
}

/**
 * Create a single element list to store a character string
 * Returns pointer to new List header.
 */

List List_storestring(char *data)
{
  List newlist;
  char *newtext;

  /* Make List */
  newlist = List_create();
  if (listerror == LIST.NOERROR) {
    /* Get memory to store copy of string in */
    if ((newtext = (char *)malloc(strlen(data) + 1)) == nil) {
      free(newlist);
      newlist = nil;
      listerror = LIST.OUTOFMEMORY;
    } else {
      strcpy(newtext, data);
      (void)List_add(newtext, newlist);
    }
  }
  return newlist;
}

/**
 * Get an element from a given index position within a list
 */

void List_getelement(List_Posn *posn, List list)
{
  /* Do we have an index to use? */
  if (list->index == nil) {
    /* No Index */
    List_Posn scanner;
    int i;

    /* Scan linearly through list appropriate number of elements */

    }
List.Marker(&scanner, list);
List.rewind(&scanner);

for (i = 0; i < posn; ++i)
    List.next(&scanner);

return List.getdata(&scanner);
} else {
    /* Index available - can do job directly */
    return *(list->index)[posn];
}

86
Block  A Set of Classes which occur at the same time

Class  A Class consists of a Student Group, Teacher(s) and a Room.

Column  See Block

Constraint  A restriction on some feature of the timetable, preventing (or requiring) some characteristic to be missing (present).

Meeting  A collection of people getting together at a certain time, in a certain place, for a common goal.

Student Group  A Group of people who have some Classes in common.

Subject  A Topic area which will be taught to a Student Group by a Teacher.

Teacher  Someone who will supervise a Class while they are working on a given Subject.
Bibliography


