School Timetables: A Case Study in Simulated Annealing

D. Abramson
H. Dang

Division of Information Technology
C.S.I.R.O.
723 Swanston St
Carlton, 3053
Australia.

R. V. V. Vidal (ed.), *Applied Simulated Annealing*

Abstract

This paper concerns the use of simulated annealing as an optimisation procedure for solving the school timetabling and related scheduling problems. It describes the basic algorithm and then gives details of the constraints required to. solve practical problems. Because annealing is a slow procedure, a special purpose computer architecture has been built which runs the algorithm two orders of magnitude faster than conventional workstations and super computers. The paper describes the details of this architecture. Some results are presented which compare the effectiveness of simulated annealing against a commercially available heuristic on very complex randomly generated data.

Keywords: Timetabling, Scheduling, simulated annealing, optimisation, computer architecture, high performance optimisation.

1. INTRODUCTION

The computation of school and university timetables has traditionally been an extremely complex procedure, involving many different constraints of varying importance. Over the years many different solutions have been proposed, ranging from techniques in operations research though artificial intelligence [5, 9, 10, 11, 13, 14, 15, 16, 17, 18, 20, 21 22]. In 1990 we proposed the use of simulated annealing as an optimisation technique for packing the various class, teacher and room tuples into the periods of the weekly timetable, and showed that annealing is capable of solving quite complex problems [3]. Since then work has continued in this area, and a system for solving timetables has been developed in which many real world constraints can be accommodated. A problem with the software solution proposed in [3] was that when used on large school problems the time to solution was so long that the scheme became unfeasible. Whilst parallel machines can offer substantial speedup, a specially designed computer architecture based around the simulated annealing algorithm has the ability to offer an even greater speedup, and makes the scheme feasible for real world problems.

This paper summarises the formulation of the timetabling problem as proposed in [3], and describes the computational structures necessary to allow rapid evaluation of the solution. It then introduces a new computer architecture capable of providing very rapid execution of the simulated annealing algorithm. Some benchmark results are presented which show the speedup attained by the new architecture. The performance of the new system is illustrated by showing results achieved on both randomly generated data as well as school data. The results compare the quality of solution obtained by using simulated annealing and a commercially available heuristic.

2. FORMULATION OF THE TIMETABLING PROBLEM

In its simplest formulation the timetabling problem can be written as a mapping problem. The input consists of a number of *tuples*, each of which contains a *class* identifier, a *teacher* identifier and a *room* identifier. The allocation task consists of mapping each tuple onto a *period* such that no class, teacher or room appears more than once per *period*. This procedure is illustrated in Figure 1. The formulation assumes that the binding of specific classes, teachers and rooms is known before the allocation is started.

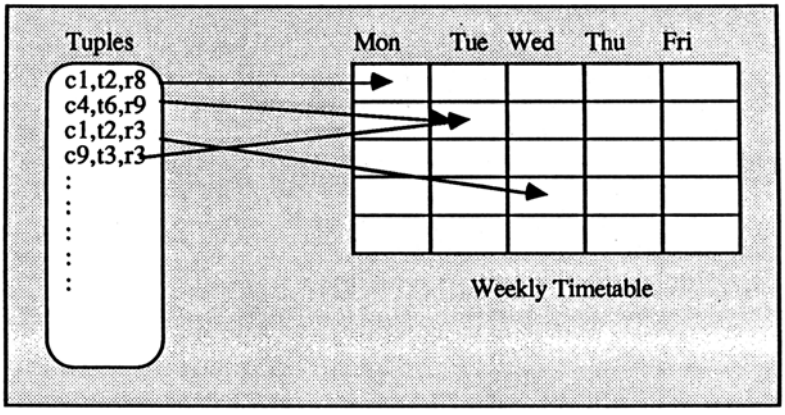

Figure 1 - Mapping tuples to periods

The quality of the timetable is measured by counting the number of clashes in the solution, thus a perfect solution has a zero cost. The simulated annealing algorithm [1, 2, 6, 7, 8, 12, 19,23, 24, 25] solves the problem by randomly moving tuples from one period to another, and measuring the cost of the transition. A decrease in cost is accepted, and an increase accepted depending on the value of the simulated temperature, as shown below in Figure 2.

```
Compute an initial temperature

while (cost <> 0) and timetable not frozen repeat

   repeat some constant number of times
         Choose a tuple Tn (class, teacher, room,  Period)
         Choose a new field value for Period called Period'
         Evaluate the cost of removing this tuple from Period
         Evaluate the cost of inserting this tuple into Period'
         Compute change in cost
         if      (change in cost ≤ 0) or
                 (change in cost is acceptable at this temperature)
                       then accept change and update cost

   compute new temperature
```

Figure 2 - The annealing algorithm

In order for the annealing algorithm to evaluate the effect of a change in cost rapidly, it is desirable that the change in cost should be computed incrementally. Thus, rather than recompute the cost of the timetable completely, the algorithm only need consider the effect of removing the class, teacher and room from a period and inserting them in another period. These operations can be made very efficient by maintaining counts of the number of occurrences of each class, teacher and room in each period. Given availability of such count values, it is possible to compute

whether a change will cause a cost increase or decrease based on some very simple state based rules:

```
If count(class,period) > 1 then saving := 1;
If count(teacher,period) > 1 then saving := saving + 1;
If count(room,period) > 1 then saving := saving + 1;
If count(class,period') > 0 then saving := saving -1;
If count(teacher,period') > 0 then saving := saving -1;
If count(room,period') > 0 then saving := saving -1;
```

This process makes it possible to model only very simple timetables. In the next section we address the issues involved in modelling real world constraints found in schools and universities.

3. REAL WORLD CONSTRAINTS

Real world timetabling problems are usually complicated by additional mapping requirements and constraints. Some examples of real world constraints are:

- Classes are bound either to specific rooms or groups of equivalent rooms during scheduling
- Classes are bound either to a specific teacher or groups of equivalent teachers during scheduling
- Classes may be designated as clashing with other classes because of common students
- The timetable may consist of an arbitrary number of days per week and periods per day
- Classes may request contiguous time slots
- Lunch and other break times may be specified
- If teachers are allocated automatically specific weekly limits may be enforced, as well as limits of how many times various types of classes are taught per week
- Teachers may limit the number of periods per day
- Teachers may request periods in which their classes are not taught
- Rooms may be declared unavailable during any period
- Class subject combinations may request a preference for certain periods
- If teachers are allocated automatically preferences may be declared for specific teachers
- Individual teacher and class preferences may be specified for specific rooms even if they are taken from a group of rooms
- Class subject combinations are spread throughout the week
- Classes may be grouped with other classes
- Classes may be fixed at certain times

In this section we examine each of these and indicate how they can be incorporated into the basic algorithm.

3.1 Room Assignment

Usually it is not necessary to assign exact room numbers to tuples. However, it may be necessary to request a room from a particular group of equivalent rooms. This can be accommodated by simply recording the number of rooms in each room group, and then not increasing the cost for that room until the count of occurrences in any period exceeds the group size. When a zero cost timetable is produced it is then possible to assign specific rooms to tuples, using an algorithm which attempts to maintain room allocations consistent across the week; that is once a room has been assigned to a class, subject combination it is desirable to maintain that allocation for the other instances of that combination. This system can be implemented easily using the count tables described in Section 2. In order to accommodate room groups, the count value is initialised to the negative value of the group size. Whenever a room group is assigned to a period, this count is incremented. When it reaches 1 it starts behaving like the other class and teacher counts, and the cost will rise accordingly.

This scheme only works if room groups are disjoint, as shown in Figure 3a. However, if a room appears in more than one room group, as shown in Figure 3b, it is not possible to use such a simple room allocation strategy because a room groups capacity is dependent on that of another room group. For example, if a room is allocated from group 1 in a particular period, it is not clear whether group 2 has a diminished capacity as a result. If hierarchical room groups are created, as shown in Figure 3c, it is possible to make a simple modification to the scheme described above. In this case, when a room is assigned from a group, the capacity of all room groups which are super sets of the group must be decreased by one. This means that when the count value for a room group is modified, all of the supersets must also be modified.

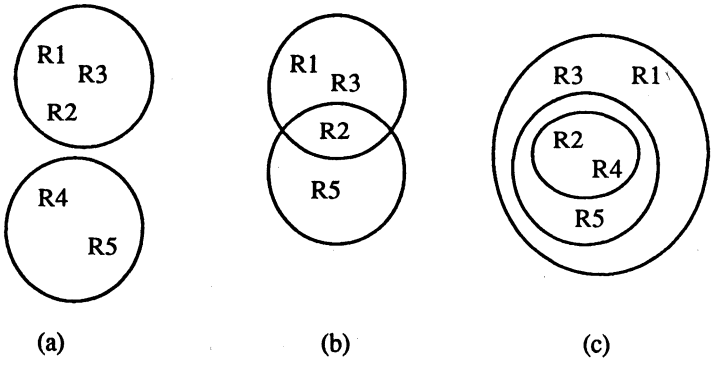

Figure 3 - Various Room Assignment techniques

3.2 Teacher Assignment

In the last section we only considered mapping tuples with fixed field values onto a period. In many schools there is flexibility in the assignment of teachers to classes. For example, if a school has 5 equivalent English classes, which can be taught by 5

different English teachers, then the assignment of teachers to classes may not be fixed before the period assignment is started. In fact, this flexible assignment may be required in order to construct a clash free timetable. Consequently, the basic annealing algorithm given in the previous section is modified to allow teachers to be mapped to tuples as well as period value. In the more general form, a tuple consists of a number of fixed field values, and a number of modifiable ones, which are then set by the annealing algorithm itself [4].

Removing a teacher from a tuple and inserting another requires basically the same cost computation to be performed as when a period is reassigned. A new teacher is chosen at random and the cost of changing the tuple is evaluated. Of course, in practice it is not possible to randomly choose any teacher from all available ones, instead a teacher is randomly chosen from one of a number of teacher groups. Teacher groups can contain any number of teachers, and unlike room groups, can interact with each other in arbitrary ways. A further constraint is that when a teacher is assigned to a class/subject combination, the same teacher must be used for all instances of the tuple throughout the week. This is enforced by examining all such tuples in one operation and either altering all of them or none. This will be discussed later under the section devoted to blocking tuples together.

The annealing algorithm is altered and appears in Figure 4.

```
Compute an initial temperature
while (cost <> 0) and timetable not frozen repeat
  repeat some constant number of times
        Choose a tuple Tn (class, teacher, room,  Period)
        randomly decide if doing teacher swapping or period swapping

        If period swapping then
              Choose a new field value for Period called Period'
              Evaluate the cost of removing this tuple from Period
              Evaluate the cost of inserting this tuple into Period'
           else
              Choose a new field value for Teacher called Teacher '
              Evaluate the cost of removing this Teacher  from tuple
              Evaluate the cost of inserting this Teacher' into tuple

        Compute change in cost
        if    (change in cost ≤ 0) or
              (change in cost is acceptable at this temperature)
                    then accept change and update cost
  compute new temperature
```

Figure 4 - The modified annealing algorithm

Similarly, the cost computation is altered in the case of teacher swapping to:

```
If count(teacher,period) > 1 then saving := 1;
If count(teacher',period) > 0 then saving := saving -1;
```

3.3 Clashing classes

The naming scheme used in the class assignment described above assumes that classes contain no students in common, and thus, classes with different identifiers can be scheduled concurrently. This scheme is adequate to capture the class structures present in the lower levels of secondary schools, but is not flexible enough to handle the various student preferences that occur in upper secondary schools and in universities.

A more powerful class costing constraint can be incorporated to reflect that some classes cannot be scheduled concurrently, even though they have different class identifiers, by maintaining a list of all the classes which *clash* with every class. Thus if two *clashing* classes are scheduled concurrently, then the cost should reflect the clash.

It is possible to devise a costing scheme in which changes can be computed incrementally when a class is moved from one period to another, however, it is necessary to read the count values for all classes which clash with the one being moved.

3.4 Contiguous Time slots

Most schools also have a requirement for multiple period classes. These are handled by allocating two or more tuples to *consecutive* periods, where the function *consecutive* can be defined by an arbitrary function of two period numbers. The value of this function is included in the cost of a solution so that when classes are configured as multiple periods, the cost is minimised.

This cost is incorporated into the annealing schedule by computing an incremental cost component when a tuple is moved based on the position of its partner tuple. Thus, the cost can be computed from the following formula:

```
for two partners:   tuple 1 = (class1,teacher1,room1,period1)
                    tuple 2 = (class2,teacher2,room2,period2)
evaluate change     tuple 1 = (class1,teacher1,room1,period3)
if contiguous(period1,period2) and not contiguous(period3,period2) then
                    saving = -1 else
if not contiguous(period1,period2) and contiguous(period3,period2) then
                    saving = 1 else
                    saving = 0
```

This scheme is very flexible and actually allows the function contiguous to enforce general relationships between any two tuples. For example, if contiguous is defined as *greater than* then the constraint can be used to force one tuple to appear before the other in the week. By linking more than two tuples together it is possible to enforce multi tuple orderings.

Another use for multiple periods is to allow teachers to request a free time period without fixing it before the timetable is computed. For example, a teacher may wish to request a day *off*, but may be prepared to accept any day of the week. This can be requested by creating a multiple period tuple set for the teacher and having the algorithm schedule it in the week as part of the timetable.

3.5 Teacher allocation limits

When teachers are allocated to tuples by the algorithm, it must be necessary to constrain the teaching load across the week. Thus, teacher limits can be specified to guarantee that no teacher exceeds the preset bounds. Further, it is necessary to restrict the number of times a teacher is allocated to a particular group of students. These constraints can be enforced by counting the number of occurrences of teachers in the week and against particular classes or teacher groups. Thus, the cost computation which is performed during teacher swapping can be augmented with the following terms:

```
If count(teacher) > 1 then saving := 1;
If count(teacher') > 0 then saving := saving -1;
If count(teacher,teachergroup) > 1 then saving := saving +1;
If count(teacher',teachergroup) > 0 then saving := saving -1;
```

3.6 Daily limits

Limits must be specified to restrict teachers from taking more than a specified number of classes per day. This constraint can be enforced by counting the number of times a teacher appears in a day, and re-evaluating the cost when the tuple is moved or when the teacher is reassigned. Thus, the following code computes the daily limit costs:

```
If count(teacher,dayof(period)) > 1 then saving := 1;
If count(teacher,dayof(period')) > 0 then saving := saving -1;
If count(teacher,dayof(period)) > 1 then saving := saving +1;
If count(teacher',dayof(period)) > 0 then saving := saving -1;
```

3.7 Unavailable times for classes, teacher and rooms

To account for times at which the classes, teachers or rooms are unavailable, the data structures used by the algorithm can be pre-initialised to *pretend* that they are already present in those periods. In this way, the same class, teacher or room will not be scheduled at the same time unless the final cost is non zero. This is achieved by simply initialising the counts so that they have values greater than 1. Similarly, the capacity of each of those resources can be increased by initialising them to less than zero, as is done in the case of room groups.

3.8 Preferences for periods

Certain tuples have preferences for particular time slots; for example it may be best if art if taught early in the morning. This requires that a preference can be attached to each tuple and that the preference is reflected in the cost function. The preference is computed by a set of predefined functions which define a preference value for each position of the week. Thus, the change in cost can be computed from:

saving := preference(period) - preference(period')

3.9 Spreading the work through the week

A good timetable has the multiple occurrences of the same class and subject distributed throughout the week, rather than being bunched together on one day. Thus, it is necessary to limit the number of times a class-subject combination is taught on one day. This can be accommodated by counting the number of class/subject combinations for each day. A unique key for the process can be computed by simply concatenating the class and subject identifiers. Thus the cost change can be computed from:

```
If count(classsubject,dayof(period)) > 1 then saving := 1;
If count(classsubject,dayof(period')) > 0 then saving := saving -1;
```

In practice it may not be serious if a class/subject appears twice per day, but it is highly undesirable for the combination to appear three or more times. To cater for this distinction a non-linear function is used to make occurrences of more than 2 times very costly, as follows:

```
If count(classsubject,dayof(period)) > 2 then saving := 3 else
If count(classsubject,dayof(period)) > 1 then saving := 1 ;
If count(classsubject,dayof(period')) > 1 then saving := saving -3
f count(classsubject,dayof(period')) > 0 then saving := saving -1;
```

3.10 Blocking classes together

In order to schedule a number of classes at the same time, it is possible to link tuples together. In this way, the cost of a move is the sum of the individual costs of each tuple, however, they are all moved at the same time. This linking is determined when the problem is formulated, and simply requires that the cost computation logic adds together the cost changes for all tuples that are linked. The system allows large blocks of optional subjects to be created and moved in one step.

The scheme is used when a teacher swap is performed to guarantee that all occurrences of a class subject combination are allocated the same teacher. In this case, these tuples are linked together.

3.11 Pre-allocating classes

Sometimes a scheduler will wish to pre-allocate tuples to specific periods. In this case it is simply necessary to alter the count values for the classes, teachers and rooms for the pre-allocated period values when the timetable is loaded. After this step, the tuple need not be processed by the annealing algorithm, and can be removed from the active set of tuples.

4. SPECIAL PURPOSE ARCHITECTURES

All of the cost constraints described in the previous sections have been implemented in software on conventional work stations. However, a number of experiments on *real* school timetables, which are generally highly constrained, showed that to solve large problems required many days of processing. In this section a special purpose architecture is described which allow all of the costs measures described in sections 2 and 3 to be enforced. The hardware has the advantage over a software implementation of being extremely fast. The architecture is based around the cost functions described in the previous section. A random number generator is used to choose a new period which is then applied to the cost evaluation section. The cost change can be computed in one time step of the machine. The change in cost is then applied to hardware which computes the negative exponential of the cost divided by the temperature, and if the change is acceptable at the temperature, it is accepted. Otherwise, it is rejected. The hardware acts as an attached processor to a host machine, and only interacts with the host at initial startup time for loading, and at each temperature shift.

To give some indication of the speedup possible the following chart in Figure 5 shows the performance of the machine against a number of software benchmarks on different commercial machines. The CRAY Y/MP column is the speed of a cut down version of the program written in C running on a uni-processor CRAY Y/MP. The SS1+ is the speed of the same code running on a SUN Sparc Station 1+. The NS32332 is the same code running on a uni-processor Encore Multimax. The Pascal columns show the speed of a more complete implementation of the algorithm written in Pascal, running on the Sparc Station and the Encore machines. The final column marked accelerator shows the performance of the full simulated annealing model running on the special purpose architecture.

Figure 6 shows a schematic diagram for the architecture of the hardware accelerator. The hardware is composed of three main sections, each of which is is outlined in the schematic. These are the timetable memories, the clash memories and the cost computation hardware.

4.1 Timetable Memory

The timetable information is held in a special timetable memory, which is accessed during the annealing algorithm. It contains the tuple information, and is updated whenever a period assignment or teacher assignment is altered. Requirements are processed cyclically rather than randomly, which allows fast inexpensive requirement selection.

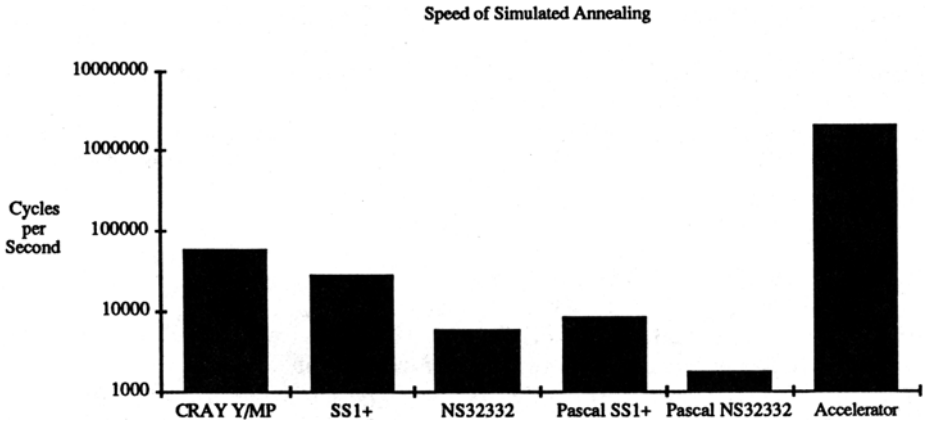

Figure 5 - Performance of annealing architecture

4.2 Clash Memories

The reason that the annealing algorithm runs slowly on conventional computer workstations is that it computes a moderately complex timetable evaluation function many times. The accelerator is able to decrease the time to solve a problem by using special purpose hardware to compute the evaluation function.

The evaluation function measures the quality of a particular timetable. At each stage of the annealing process a new timetable is generated, and is accepted based on its quality. The quality measure includes attributes such as, how may clashes are present, whether staff have their teaching desires met, etc. Many of these attributes are independent, and can be evaluated concurrently given the correct hardware. However, sequential computers forces sequential evaluation, which slows down the process. Further, the hardware can arrange for the information to be accessible with less effort than a computer program. For example, the timetable is stored in an array when implemented by software. In order to access the contents of the array, the software must perform time consuming index operations. These operations are not required when the scheme is implemented in hardware.

Each of the clash memories stores a count of the number of occurrences of that attribute in a given period or day. for example, the class clash memory holds the number of each class type in each period of the week. This allows the hardware to determine whether inserting a class into a period will cause the cost to rise or not, and similarly for removing a class, whether the cost will fall. In the case of the Class-Subject clash memories, the count pertains to the entire day, and not just a single period.

4.3 Cost computation hardware

After each cost computation the annealing algorithm must update the global cost. This operation is performed by a special purpose arithmetic unit, which can add and subtract 16 bit integers.

4.4 Basic costs

All of the requirements are stored in the special timetable memory, as shown in Figure 6. This memory is addressed by a sequential counter, thus requirements are chosen sequentially rather than randomly from the tuple space. Since requirements are composed of many tuples, they are stored in more than one word of timetable memory. Consecutive words are read from the requirement in order to process all of the tuples.

After each tuple is read from the timetable memory, the tuple attributes are presented to the clash memory. The clash memory stores counts of the number of occurrences of each attribute in each period and day of the timetable. In most cases, two copies of each count are stored to allow simultaneous computation of the cost of inserting the attribute in the new period and removing the attribute from the current period. The new and current period are routed via a cross-bar switch to each of the clash memories. The counts in each of the memories is processed by an incrementer/decrementer to produce a new count as well as a cost of insertion and deletion of that attribute. The costs are collated and sent to a cost table, which stores the change in cost for all possible combinations of cost change.

The cost changes are summed for all tuples of the requirement, to produce a net change in cost. If the net change is negative then the new period is written to the timetable memory, and the clash arrays are updated accordingly. If the change is positive then it is used as the key to an exponential table, which stores the probability of the cost being accepted given the current temperature. The probability is compared to a random number, which determines whether the change is accepted.

Each time a requirement is processed a new random period is chosen. Because the period must be chosen in a range which is not necessarily a power of two, a random number is processed by computing the modulus of the number with the number of periods per week. This computation yields a number in the correct range. Division is avoided by using a precomputed table of mod operations.

The same modulus memory is used for choosing a new teacher for those tuples which allow automatic teacher selection. Automatic teacher selection is performed in a similar manner to period selection, except that the tuples which require teacher selection are chained together by a special chain instead of being stored sequentially in the timetable memory. Also, the teacher is chosen from a number of teacher groups, each of which has different sets of teachers.

4.5 Class Clash Cost

Some classes are designated as clashing even though they have different class numbers. A cost is computed for these based on the number of occurrences of each class in the period. The change in cost is computed by taking the difference in the number of occurrences of each class in the list of clashing classes.

4.6 Preference Cost

It is possible to specify a preference that a requirement may have for one or more periods. These preferences are computed statically in a special lookup table, and can be recomputed using an adder when a requirement is moved.

4.7 Multiple Period Cost

Double periods are enforced by maintaining linkage between the two periods that make up the double. A special linkage memory stores the period number of the partner for every double period requirement. This allows the hardware to obtain the location of a requirement and its partner in one cycle. These two periods are presented to a period comparison table, which determines the cost of the configuration. The change in cost is computed by calculating the cost of removing the requirement from the current period and inserting it in the new period.

5. RESULTS

This section gives a number of experimental results. The data is a set of randomly generated data which is *totally constrained,* i.e. each period contains every class, teacher and room. Because the solution is generated, it is known there is a perfect solution to each problem, however, because there is no flexibility in the final solution, it is extremely difficult to find. This data type acts as an excellent test of an optimisation algorithm because most schools are heavily constrained. The results given compare the performance of the annealing hardware and a fixed commercially available heuristic. Whilst we have a number of real school timetables available, it is difficult to compare the performance of the simulated annealing version against the heuristic because the data sets make use of the different functionality available in the two systems. It is not possible to reveal the exact nature of the heuristic for commercial reasons, however, the code uses a backtracking type algorithm which uses a large amount of context sensitive information to determine the best position to place tuples.

The table in Figure 7 shows the behaviour of the simulated annealing algorithm using the hardware assist described in the previous section. It shows the best and average results obtained from six independent annealing scheduled at a number of cooling rates.

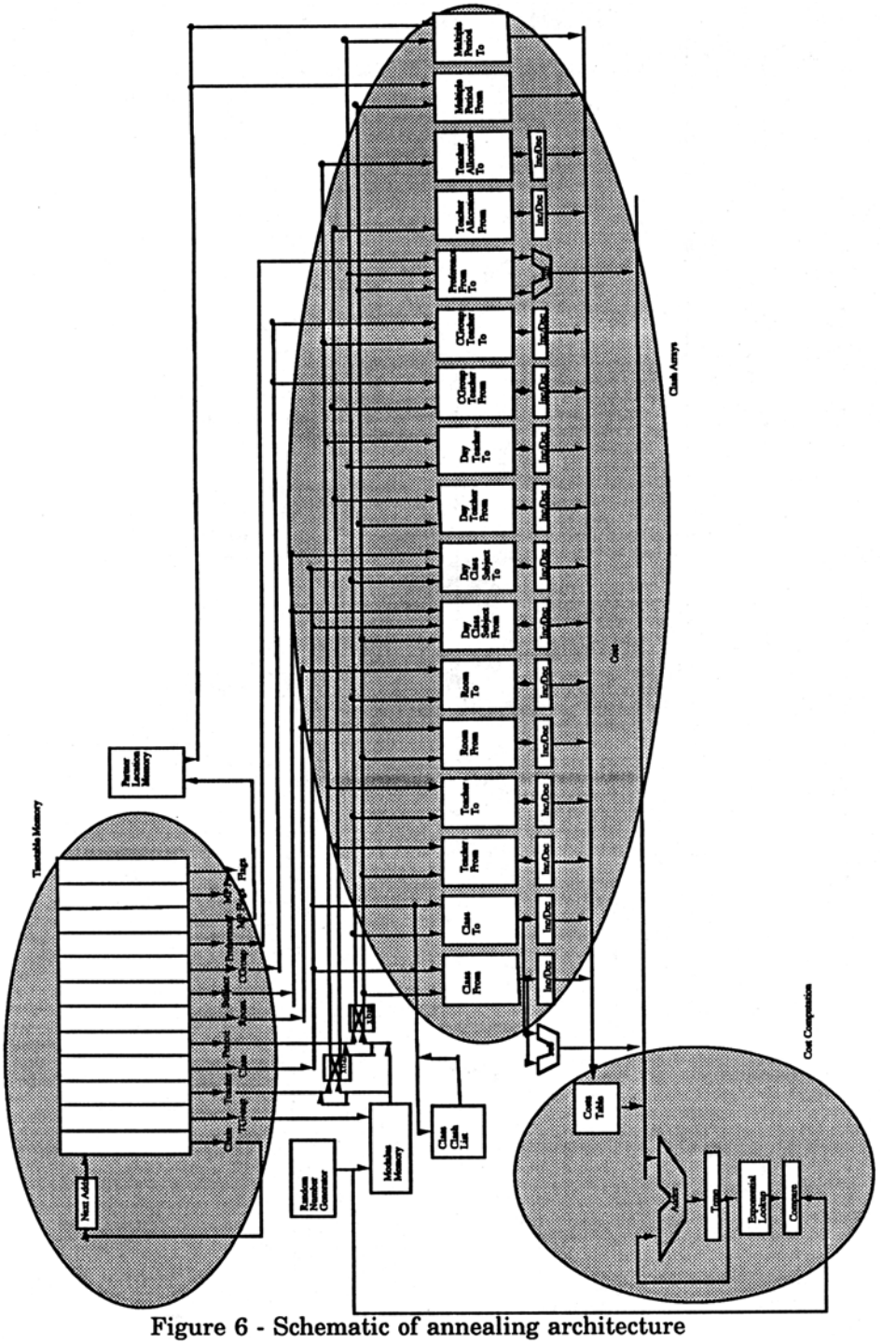

Figure 6 - Schematic of annealing architecture

Test	Number of Tuples	Number of Classes	Number of Teachers	Number of Rooms	Number of Periods	Cooling Rate	Average Cost	Best Cost	Average Time (Secs)
1	150	5	5	5	30	0.8	7.5	5	3.42
						0.9	6.83	5	3.83
						0.99	2.5	0	10.23
						0.999	0.67	0	72.45
					36	0.8	1.33	0	3.5
						0.9	1.67	0	3.83
						0.99	0.67	0	12.17
						0.999	0	0	94
2	180	6	6	6	30	0.8	11.33	6	3.68
						0.9	9	5	4.08
						0.99	5.5	3	8.67
						0.999	2.5	0	80.22
					36	0.8	2.33	0	3.83
						0.9	2.17	0	4.33
						0.99	0.67	0	14.5
						0.999	0	0	114.17
3	210	7	7	7	30	0.8	12.17	8	3.92
						0.9	11.83	7	4.08
						0.99	5.67	4	10.45
						0.999	3.83	2	107.05
						0.9999	2.5	2	6635
					36	0.8	4.83	3	4
						0.9	3.17	2	4.83
						0.99	3	2	17
						0.999	1	0	123
						0.9999	0	0	1330
4	240	8	8	8	30	0.8	14.5	10	4.23
						0.9	13.83	9	4.77
						0.99	9.33	7	16.85
						0.999	5.5	4	125.83
						0.9999	3.83	2	7859
					36	0.8	6.5	5	4.23
						0.9	5	3	4.77
						0.99	4	2	16.85
						0.999	3	2	125.83
						0.9999	0	0	1533
5	270	9	9	9	30	0.8	16.17	13	3.97
						0.9	16	14	5.02
						0.99	9.67	8	13.63
						0.999	6.17	5	122.45
						0.9999	5.17	4	8106
						0.99995	4.5	2	19167
					36	0.8	10.67	7	4.5
						0.9	7	5	5.83
						0.99	5.83	3	21
						0.999	3.17	2	191.83
						0.9999	1.17	0	1873
						0.99995	1.5	0	3278

Test	Number of Tuples	Number of Classes	Number of Teachers	Number of Rooms	Number of Periods	Cooling Rate	Average Cost	Best Cost	Time (Secs)
6	300	10	10	10	30	0.8	19.5	17	4.12
						0.9	20	17	5.28
						0.99	12.8	12	22.17
						0.999	8.17	6	154.42
					36	0.8	10.33	7	4.83
						0.9	10.33	7	6.17
						0.99	7.5	6	25.17
						0.999	6.83	5	206.17
						0.9999	3.67	2	2088
						0.99999	1.5	0	17918
7	330	11	11	11	30	0.8	23.67	18	4.22
						0.9	19.8	17	4.98
						0.99	15.33	13	19.08
						0.999	11.17	10	153.58
					36	0.8	12.33	12	4.33
						0.9	12.5	9	5.5
						0.99	10	8	19.17
						0.999	8.17	7	187.83
						0.9999	5.5	5	1643
						0.99999	3.33	3	21654
8	360	12	12	12	30	0.8	24.17	21	5.15
						0.9	24.67	21	6.3
						0.99	18.83	15	21.78
						0.999	10.67	9	181
					36	0.8	15.83	14	5.17
						0.9	14.5	10	6.83
						0.99	11.17	8	29.33
						0.999	9.67	9	251.17
						0.9999	6.33	4	2365
						0.99999	4.33	0	24005
9	390	13	13	13	30	0.8	25.83	21	4.78
						0.9	27.5	26	6.03
						0.99	21.5	20	24
						0.999	15.5	13	189.58
					36	0.8	18	17	5.33
						0.9	14.33	12	6
						0.99	14.83	13	23.83
						0.999	11.5	9	251.83
						0.9999	8.17	7	2338
						0.99999	5.17	4	25192

Figure 7 - Results

Even though the results are presented for two different size weeks, the resulting timetable is contained in a 30 period week. The experiments show that the system has difficulty solving problems when the number of available periods is exactly equal to the size of the week. This is because the system can get trapped in local minima in which it is not possible to swap two tuples. This can occur when the

required interchange cost is too high for the given temperature if the interchange is performed as two moves, as shown in 8 (a). However, if the interchange is performed as one swap then the cost of actually swapping them is sufficiently low, as illustrated in Figure 8 (b). This problem can be solved by allowing the system to use more than 30 periods, but by marking the extra 6 periods (one whole day) as less desirable than the first 30. In this way, a timetable which has zero cost will only use the first 30 periods. The scheme avoids the local minima problems cited above because it is possible to perform a swap by a three stage move as shown in Figure 8 (c). The results show that the annealing system is capable of solving some very hard problems when this modification is allowed.

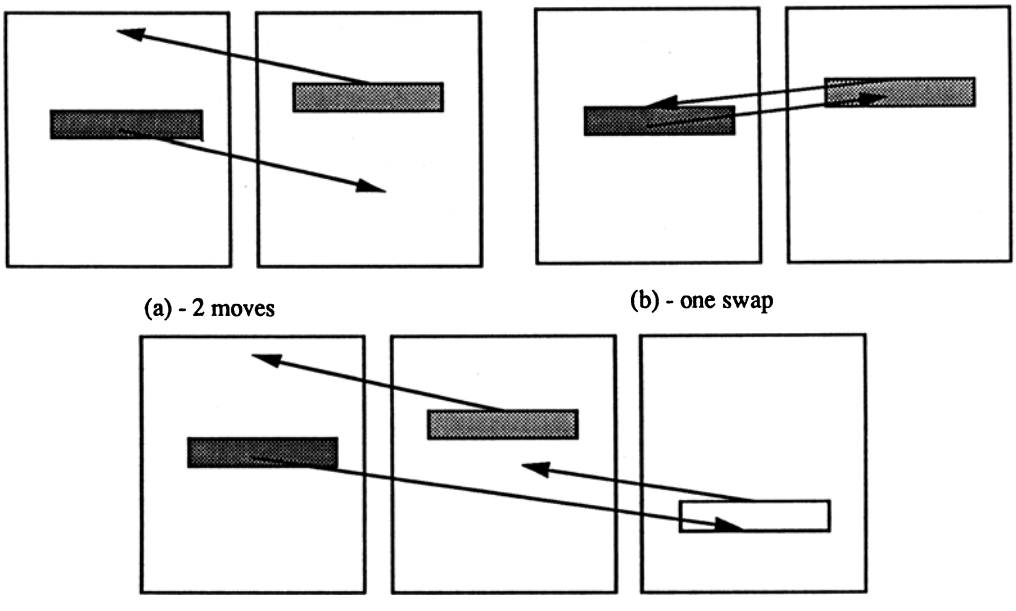

Figure 8 - swaps versus moves

The following graph in Figure 9 compares the performance of the annealing algorithm as executed on the accelerator hardware against the heuristic code executed on a PC486. It shows that when the heuristic algorithm can solve the problem then the annealing requires the accelerator to be competitive. However, the annealing is capable of solving much more complex problems (with many more constraint types) than the heuristic. A missing bar in the graph indicates that the system could not solve the problem. The time for annealing is the time required to achieve a zero cost solution using the lowest cooling rate which achieved the zero cost solution. This may have taken more than one cooling schedule to achieve the result. In the case of the heuristic, it is the time required to produce a solution.

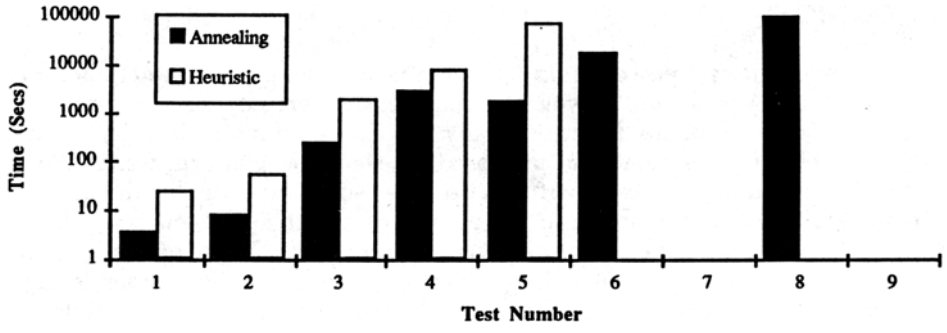

Figure 9 - Performance of Heuristic against annealing

The following graph in Figure 10 compares the performance of the annealing algorithm with the heuristic in the cases when the heuristic could not solve the problem. It shows the cost achieved for the same processing time (4 hours). It should be noted that the final cost value may not be the minimum achieved by the annealing because it was not possible to run the heuristic for the same amount of time. For example, tests 7 and 8 were actually solved by the annealing algorithm, but the time taken to produce a non-optimal solution is used for the comparison. The first 5 tests are omitted because they were solved by both systems. In spite of these problems, the graph gives an indication of the comparative performance. In all cases the annealing achieved a lower cost than the heuristic.

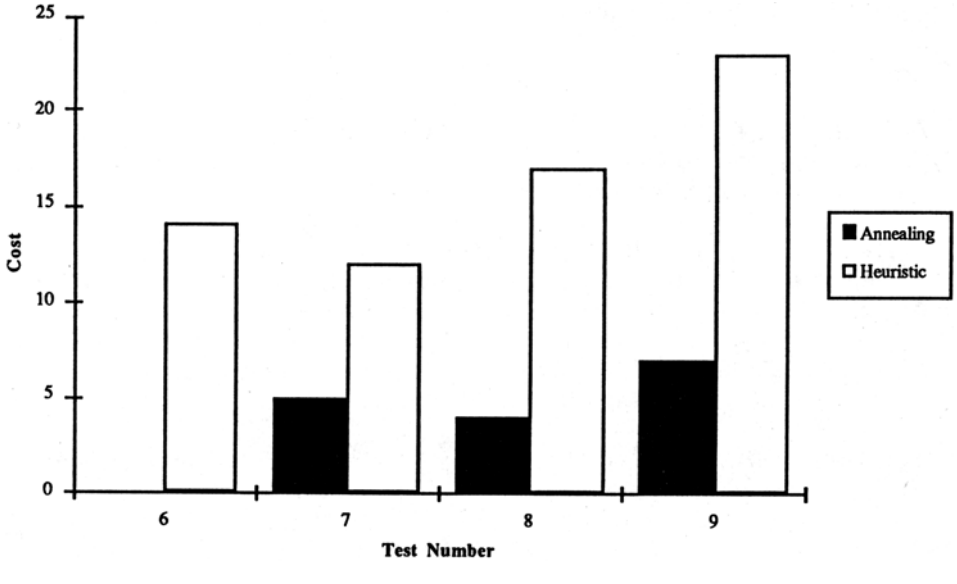

Figure 10 - Solution costs of annealing against heuristic

5. CONCLUSIONS

Simulated annealing is a powerful optimisation technique, and can be applied to the schools timetabling problem. However, it requires accelerated execution in order to be useful on large problems found in many schools. This paper has demonstrated the effectiveness of the scheme and proposed an execution platform which provides orders of magnitude increase in performance over software forms of the algorithm. The results indicate that annealing is capable of solving problems without any specialist knowledge of the solution techniques, and thus offers a generic optimisation technique. The annealing scheme described in this paper is able to handle many varied real world constraints, again without specialist knowledge of the implications of those constraints.

Some recent experiments have illustrated that better cooling schedules have the potential to improve the performance of annealing on the timetabling problem quite dramatically. However, these results will be presented in another paper.

Acknowledgments

The High Performance Computation Program is a joint program between the Commonwealth Scientific and Industrial Scientific Organisation (CSIRO) Division of Information Technology and the Royal Melbourne Institute of Technology (RMIT). H. Dang is supported by an Australian Post Graduate Research Award. The project is a joint activity between CSIRO and Computer Techniques P/L. Thanks go to Mr John Shaw of Computer Techniques for his assistance in this work. Thanks also go to Amal deSilva and Peter Fox for editing a draft of this paper.

References

[1] Aarts, E.H.L., F.M.J. de Bont, J.H.A Habers and P.J.M. van Laarhoven, "A Parallel Statistical Cooling Algorithm", Proceedings STACS 86, Springer Lecture Notes in Computer Science, 210(1986) pp 87-97.

[2] Aarts, E.H.L., F.M.J. de Bont, J.H.A Habers and P.J.M. van Laarhoven, "Parallel implementations of the Statistical Cooling Algorithm", Integration, 4(1986) pp 209-238.

[3] Abramson, D. "Constructing School Timetables using Simulated Annealing: Sequential and Parallel Algorithms", Management Science, Jan 1991.

[4] Abramson, D.A., "A Very High Speed Architecture to Support Simulated Annealing", IEEE Computer, May 1992.

[5] Akkoyunlu, E.A. , "A linear algorithm for computing the optimum university timetable", Computer J, 16(4), 1973, pp 347-350.

[6] Casotta, A., F. Romeo and A.L. Sangiovanni-Vincentelli, "A Parallel Simulated Annealing Algorithm for the Placement of Macro-Cells, "Proc IEEE Int. Conference on Computer Aided Design, Santa-Clara, November, 1986, pp 30-33.

[7] Cerny, V., "Multiprocessor Systems as a Statistical Ensemble: A Way Towards General Purpose Parallel Processing and MIMD Computers?", Comenius University, Bratislava, unpublished manuscript, 1983.

[8] Chyan, D.J., and M.A. Breuer, "A Placement Algorithm for Array processors", proceedings 20th Design Automation Conference, Miami Beach, June 1983, pp 182-188.

[9] Csima, J. and C.C. Gotleib, "Tests on a computer method for construction of school timetables", Comm. ACM, 7(3), 1961, pp 160-163.

[10] de Werra, D., "An Introduction to timetabling", European Journal of Operations research 19 (1985), pp 151-162.

[11] de Gans, O.B. "A computer timetabling system for secondary schools in the Netherlands", European Journal of Operations Research, Vol 7, 1981, pp 175-182.

[12] Greening, D., "A Taxonomy of Parallel Simulated Annealing Techniques", UCLA Computer Science department technical report CSD-890050, August 21, 1989, Los Angeles, California.

[13] Folkers, J.S. " A computer system of time-table conditions", Ph.D. Thesis, Delft University of Technology, 1967.

[14] Fujino, K.,"A preparation for the timetable using random number", Information processing in Japan 5, 1965, pp 8-15.

[15] Gotleib, C.C. "The construction of class-teacher time tables", in C.M.Popplewell, Ed., Proc IFIP Congress Munchen 1962 (North-Holland, Amsterdam, 1963).

[16] Greko, B. "School scheduling through capacitated network flow analysis", Swed. Off. Org. Man., Stockholm, 1965.

[17] Kang, Le and White, G. " A Logic approach to the resolutionof constraints in timetabling", European Journal of Operations research 56 (1991) EOR00732, North-Holland,

[18] Kang, Le, Belanger, P., White G., con Schoenberg G. "Computerized Scheduling", Proceedings of the 36th Annual College and University Users Conference, Louisville, Kentucky, April 28- May 1, 1991/

[19] KirkPatrik, S., C.D. Galatt Jr and M.P. Vecchi, "Optimization by Simulated Annealing", Science, 220(1983) pp 671-680.

[20] Lawrie, N.L. "An integer programming model of a school time-tabling problem", Comput. J. 12 (1969), pp 307-316.

[21] Nesa Wu and R. Coppins, "Linear Programming and Extensions", McGraw-Hill, Maidenhead.

[22] Macon N. and E.E. Walker, " A Monte Carlo algorithm for assigning students to classes", Comm. ACM 9(6), 1966, pp 339-340.

[23] Spira, P. and C. Hage, "Hardware Acceleration of Gate Array Layout", Proc 22nd Design Automation Conference, Las Vegas, June 1985, pp 359-366.

[24] Sechen, C. and A.L. Sangiovanni-Vincentelli, "The Timber Wolf Placement and Routing Package", IEEE Journal Solid State Circuits, SC-20 (1985), pp 510-522.

[25] van Laarhoven, P.J.M. and E.H.L. Aarts, "Simulated Annealing: Theory and Applications", D. Reidel Publishing Company, 1987, Kluwer Academic Publishers Group.