Cotiro:

A College

Time and Room

Scheduler

Using Constraint

Satisfaction

Gregory G. Leedberg
Cornell University
www.leedberg.com

Originally written 12/15/2003
for the CS 730 course at the University of New Hampshire

A college time and room schedule consists of a list of all courses offered by the college for a given semester, as well as the time the class takes place, the room in which it will meet, and the teacher who will be teaching the course. In a simple situation, generating this time and room schedule would be trivial. An example would be a small number of classes, a single building with rooms all of the same size, and teachers who are always available. However, in the real world, the process of generating an efficient schedule can be time-consuming. A real-world time and room schedule must take into account varying class and room sizes, teachers' availability, and teachers' building preferences, and generate one schedule consisting of potentially hundreds of classes taking place in dozens of buildings, being taught by hundreds of teachers, many of whom will be teaching several classes.

The problem of developing a program to automate this process is interesting for several reasons. First, this is a program which has a real, practical use, potentially for even more than just college time and room schedules. Making complicated schedules which must take many variables into account is a task performed in many industries – work schedules, airplane flight schedules, television network program scheduling, etc. A program which can efficiently generate college course schedules could be useful in these industries as well. Secondly, the problem is interesting simply because of the exceedingly complex nature of the problem itself. When one thinks about time and room schedules and all that they must take into account, it is amazing to think of how they are created. Developing an artificial intelligence-based system for doing such a task is surely an interesting problem, which even people outside the world of AI can appreciate.

The program described in this paper is designed to accomplish this goal. It takes a set of courses (with size and teacher attributes), a set of teachers (with time availability and building attributes), and a set of rooms (with building and size attributes), and generates a time and room schedule which meets several constraints inherent to the problem. Namely, a teacher cannot teach multiple classes at the same time, multiple classes cannot take place in the same room at the same time, a class must be assigned a room big enough to hold it, a teacher must teach in the building they prefer, and teachers should only be assigned class times during their available hours. The method for approaching this problem is constraint satisfaction. The program attempts to generate a satisfactory schedule as quickly and efficiently as possible, using various constraint satisfaction techniques and heuristics, which will be described later.

There is much literature available on general constraint satisfaction problems. However, there was not much to be found specifically about school scheduling problems. In general, references found to this particular class of problems merely referenced them as an example of what constraint satisfaction techniques could be used to solve. For example, Brelaz (1979) states that, "This type of [scheduling] problem arises in the so-called school scheduling problem where teachers are 'agents,' classes are 'machines,' and lectures are 'jobs.'" The remainder of the paper primarily relates constraint satisfaction to the problem of coloring the vertices of a graph. This was only one instance of finding a reference to job scheduling while investigating the constraint satisfaction literature, though. The problem of school scheduling seemed to lend itself quite well to the technique of constraint satisfaction, and even though no exact references were found to school scheduling, the numerous references to the more general problem of job scheduling only backs up this intuition. However, there are other approaches to job scheduling. One system was found, called "Decision Support System" (DSS) which is designed for complex job scheduling (Marinho, et al, 1999). The paper on DSS mentions that AI and constraint satisfaction are one method for job scheduling, but goes on to say that DSS does not make use of them. Instead, DSS is said to use heuristics for generating job schedules (although not much more information is given). Another job-scheduling system is Micro-Boss (Sadeh, 1995). Micro-Boss uses constraint satisfaction for creating schedules and is used for scheduling in environments such as the army. Micro-Boss uses constraint satisfaction combined with optimizing heuristics and constraints.

Even without much previous work done which directly relates to the problem at hand, there is much existing knowledge to work off of. As said, there are at least two significant systems in existence for job scheduling, which is what school scheduling is based upon. Additionally, the field of constraint satisfaction, as a whole, is very well developed and very able to be applied to various different problems. Constraint satisfaction was first considered in some form by the Indian mathematician Brahmagupta in the seventh century (Russell, Norvig 2003). In more modern times, the idea of backtracking search (which this system uses) dates back to the 1950's with the work of Lehmer (Bitner 1974).

In describing how this program works, we will start with a high-level description and then proceed to elaborate in more depth once the overall structure is understood. In any constraint satisfaction problem, the problem description must consist of a set of variables, V, a domain for each variable, D, and a set of constraints, C. A complete assignment is when all variables of V have been assigned some value from D, and a solution is said to be found when that assignment is consistent, i.e. is does not violate any of the constraints in C.

In this problem, the variables are the course sections which must be scheduled. For example, CS415 would be a course offered, and so would be a variable. Each variable has a unique name, generally the name of the course. Since each section of a course is a different instance which must be independently scheduled, we should simply name the first section of CS415 as CS415-1, so that it is unique from CS415-2, and so on. It is assumed at this point that the professor who is teaching a given course is already predetermined, and so that is a predefined attribute of the course. So, we could say that CS415.Teacher = Johnson. This scheduling system is not designed to assign teachers to courses, as it is assumed that it is decided by some other qualifications beyond the scope of constraint satisfaction. The other predefined attribute of a course is the estimated size of a course. This is important to know when scheduling, but there is no way for the program to deduce this from the supplied data, and so it is assumed that we can know approximately how many students can be expected to take the course.

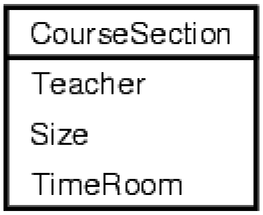

Figure 1. A CourseSection (variable) object

The domain for these variables is the set of available times and rooms. We do not consider just each room to be a separate value, but take into account each span of time available in that room. So, "Nesmith 326" would not be a value in the domain, but "Nesmith 326 8am – 9am" would be. For each value, the predefined attributes are building ("Nesmith"), room ("326"), size (the number of students that can fit in this room), and the particular time span for this value.

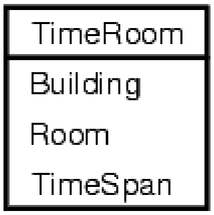

Figure 2. A TimeRoom (value) object

In fact, all attributes for the TimeRoom values are predefined, as a TimeRoom object is meant to exist as a value assigned to a course. A CourseSection object has predefined attributes as described, but also has one "open" attribute – timeRoom. This attribute will be assigned a TimeRoom object which represents the time and room this particular course has been scheduled for.

Lastly, there is the set of constraints, C. In this system, there are several different types of constraints which it is possible to define. These types are called InSpanConstraint, NoOverlapConstraint, BuildingConstraint, NotSameRoomConstraint, and SizeConstraint. Any number of these various types of constraints can be defined and then stored in a database of all constraints for the system. InSpanConstraint is a unary constraint which confines a specific course to a specified span of time (for instance, that CS415 must take place between 10am and 2pm). NoOverlapConstraint is a binary constraint which establishes that the time spans of two particular courses cannot overlap. BuildingConstraint is a unary constraint which constrains a particular course to a specified building. NotSameRoomConstraint is a binary constraint which prohibits two courses from being assigned the same TimeRoom. SizeConstraint is a unary constraint which says that the given course must be in a room which is able to seat the number of expected students for that course.

The system uses a "State" to embody a given assignment (either partial or complete). A state is comprised of two lists of courses. One list is a list of all assigned variables (this list is empty in the initial state), while the other is a list of all unassigned variables. This is shown graphically below. The system uses backtracking search (described later) to traverse a tree of partial assignments until it finds a complete, consistent assignment, which is then considered the solution. Each constraint described above can respond to a "satisfied" message, which, given a list of assigned variables, can return true or false, indicating whether that assignment is consistent with that constraint. The set of all constraints can also respond to a "satisfied" message, which also returns true or false, indicating whether a supplied state is consistent with all constraints in the system.

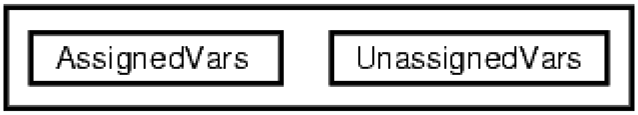

Figure 4. A State object

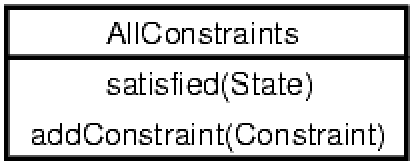

Figure 5. Interface to the set of constraints

What has been given is an overview of the system. We will now proceed with a more in-depth look at the various techniques used and how the system functions.

First, the course information (name, size, teacher) is read in from one file (default name is "courses"). A CourseSection object is created for each course. Next, room information (building name, room number, and room size) is read in from a file (default name is "rooms"). In order to generate all of the possible TimeRoom objects, we must break up the day into available time spans. The start and end times of the day for this college are defined within the system (default is 8am – 9pm). This time span is broken up into class-time slots (8:10am – 9am, 9:10am – 10am, etc), and then the set of rooms is crossed with the set of time slots to create a set of all possible TimeRooms.

All constraint satisfaction systems must use constraints to limit the possible instantiations of each variable. In this system, the constraints exist at runtime as actual, explicit constraints (as delineated above), but they are generated implicitly before the tree traversal from information supplied. A NotSameRoomConstraint is created for every possible pair of courses, because it is implicit that no two courses can occupy the same room at the same time. A SizeConstraint is also generated for every course, because it is necessary that every class be held in a room which can accommodate it. A NoOverlapConstraint is generated between courses which are taught by the same professor. This is prudent because it is impossible for one professor to teach two courses at the same time. This is accomplished simply by storing a list of each course taught by a professor and then creating a binary constraint between every possible pair.

The BuildingConstraints and InSpanConstraints require extra information in order to be created. In addition to reading in course information and room information from a file, the system reads in teacher information. If a teacher wishes, they can specify a preference for building and time span (i.e., they only want to teach in Nesmith and only can teach 12pm – 5pm). If this information is supplied, then corresponding constraints are created for each of the courses taught by that professor (using the course lists generated previously).

These constraints are all stored and make up the set of constraints which must be satisfied in order for a generated schedule to be considered a solution.

The program uses backtracking search to try and find a consistent assignment. Backtracking search involves assigning values to one variable at a time. Each time a variable is assigned a value, we must check to see if the resulting state satisfies all constraints. If it does, we may continue. If it does not, then no further assignments to other variables can possibly be consistent – this is called a "preclusion" by Bitner. So, we "backtrack" and try a different value for the current variable. If we exhaust all of those options, then we backtrack some more and try different values for the previously assigned variable, and so on. This process continues until we either find a consistent, complete solution, or have found every possible state to be inconsistent.

In doing research, there seemed to be at least two different ways of implementing backtracking search. One was "local search", in which we straightforwardly try assigning values to variables one at a time. Information is maintained which allows us to remember what assignments were made, so that undoing those assignments is possible. Another approach is depth-first search. According to Bitner, depth-first search is also equivalent to backtracking search. This is because each depth of the tree will be the assignment of one variable, and the children of each state are the various possible assignments for the next unassigned variable. So, if an inconsistent assignment is made, depth-first search will next assign the next values for the current variable, before moving up to the previous depth, i.e. trying new values for the previous variable. The following diagram displays a partial expansion of a search tree where the variables {A, B, C} have the domain {1, 2, 3}.

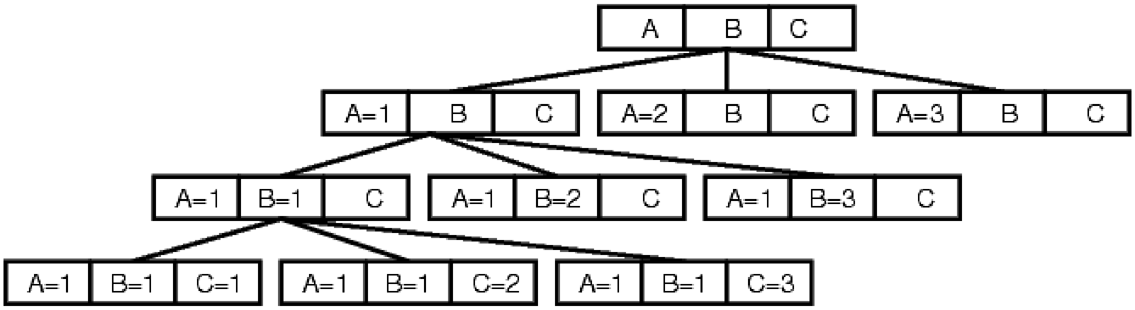

Figure 6. Partial depth-first expansion of a search tree

In many other problems in which we have used depth-first search, it has been found that depth-first search can run forever and not find a solution (such as in the 15-puzzle problem). At first, this was a fear for this program as well. However, the difference here is that these trees are finite, and the solution must be at the deepest level, so depth-first makes sense. Depth-first search allows us to get to the deepest level more quickly than breadth-first search, and will not run forever since the trees are of finite depth (unlike in the 15-puzzle).

Using just backtracking search, the program is always able to find a consistent solution. However, there are several types of optimizations discussed in the constraint satisfaction literature. Two of these have been implemented in the program to see how they affect the speed of schedule generation.

One idea in the constraint satisfaction literature is that of arc consistency (also called constraint propagation). Arc consistency involves removing values from a variable's domain which violate constraints. There are various levels of arc consistency, and the various algorithms for them are discussed in Bartak, but the one implemented in this program is called 1-consistency. 1-consistency involves removing from domains all values which violate unary constraints. To accomplish this, we go through each variable, instantiate it with each possible value sequentially, and check to see if this assignment violates the constraint set. If it does, it is removed from the domain. This constraint propagation is done up-front at the beginning of the program, because this is cheaper than trying to do it as we go (Moore 2002).

Implementing this idea involved some changes to the design of the program. Originally there was going to be a single shared domain for all variables. The strategy was that this would enforce the constraint that no two classes can share the same room at the same time, but without having to explicitly state that constraint. The program was redesigned to be more "correct" and give each variable its own domain. This required many new constraints to be added to the system, but allows us to use 1-consistency by removing values from domains which violate unary constraints – which would not have been possible with just one shared domain.

Arc consistency – including 1-consistency – is called look-ahead in Dechter (2001). This is because it looks ahead to possible assignments, and preemptively removes values from domains which could cause problems later on. This way, we need not ever consider these impossible assignments.

The other optimization in use is the idea of using heuristics to control variable ordering. There is much discussion in constraint satisfaction writings about the fact that the order in which you consider the variables can greatly affect how long it will take you to find a consistent assignment. Various heuristics are possible for ordering the variables. The heuristic in use in this program is called the "most-constrained variable" heuristic (Kumar 1992), which means that we consider how many constraints each variable is involved in.

To calculate the heuristic, prior to beginning the backtracking search, we iterate over all variables and count how many constraints they are part of. Each variable then stores its heuristic value, and the unassigned variables list is sorted in reverse order based on this value. In this way, the most-constrained variables will appear at the head of the list, while the least-constrained variables will appear at the tail. As we traverse the tree, the next variable to be assigned is taken from the head of the list, and so we attempt to assigned values to variables which are most constrained, which helps us find potential problems quickly, rather than deep within the tree.

The program outputs all of its generated constraints, as well as the generated schedule, so that we may verify that it is generating both correctly. It also outputs the number of states it considered (including both partial assignments and complete assignments) so that we can compare the various heuristics and arc consistency algorithms. We will use a small set of classes and rooms so that the reader may easily see that the constraints are all being satisfied, but the set used will be just complex enough to show that the program can handle real-world situations.

What follows are the input files (notes have been added to explain the format of these files):

```
courses
CS730                 Course name
Rubenstein            Professor
30                    Size
MATH531
Feldman
15
MATH532
Feldman
15
CS415
Johnson
60
```

```
rooms
Nesmith        Building
326                   Room number
30                    Number of seats
Horton
4
200
Kingsbury
319
15
```

```
teachers
Rubenstein            Teacher name
Nesmith       Preferred building
15                    Start of available time (hour)
00                    Start of available time (minute)
21                    End of available time (hour)
00                    End of available time (minute)
Feldman
Kingsbury
8
0
12
0
```

This set of inputs file comprises all the possible constraints. We have multiple classes (so have to make sure that classes do not take place in the same room at the same time), have multiple teachers (so cannot have the same teacher teaching two classes at the same time), multiple rooms with varying capacities (so a class must be matched with a correctly sized room), and teacher preferences for buildings and times.

If we run these variables and domains through the assignment algorithm without any constraints, we produce the following result (the start time and end time for the school day were set to 8:00am and 9:00pm, respectively):

```
[greg@fozzie Project]$ java Scheduler --noconstraints
Course: CS730
Instructor: Rubenstein
Size: 30
Room: Kingsbury 319: 19:10 - 20:00 (15 seats)
h-val: 0
-------------------------------------------
Course: MATH531
Instructor: Feldman
Size: 15
Room: Kingsbury 319: 19:10 - 20:00 (15 seats)
h-val: 0
-------------------------------------------
Course: MATH532
Instructor: Feldman
Size: 15
Room: Kingsbury 319: 19:10 - 20:00 (15 seats)
```

```
h-val: 0
-------------------------------------------
Course: CS415
Instructor: Johnson
Size: 60
Room: Kingsbury 319: 19:10 - 20:00 (15 seats)
h-val: 0
-------------------------------------------


**Traversed 5 states
```

This demonstrates that they all have the same domain, as they were each assigned the same time room value. It also shows that constraints are necessary for this algorithm, as this schedule is clearly not possible. This may seem like a useless exercise, however it does show that the depth-first search works, though. Only 5 states are traversed – one for each variable assignment, plus the empty initial state.

If we run the program with constraints enabled, but without 1's consistency and heuristics, we produce the following result:

```
[greg@fozzie Project]$ java Scheduler --noconsistency --noheuristics
Constraints:
Size constraint: CS730 must fit room
Size constraint: MATH531 must fit room
Size constraint: MATH532 must fit room
Size constraint: CS415 must fit room
Building constraint: MATH531 must be in Kingsbury
InSpan constraint: MATH531 must be in time span 8:00 - 12:00
NoOverlap constraint: MATH531 must not overlap course  MATH532
Building constraint: MATH532 must be in Kingsbury
InSpan constraint: MATH532 must be in time span 8:00 - 12:00
Building constraint: CS730 must be in Nesmith
InSpan constraint: CS730 must be in time span 15:00 - 21:00
NotSameRoom constraint: CS730 must not share a time/room with MATH531
NotSameRoom constraint: CS730 must not share a time/room with MATH532
NotSameRoom constraint: CS730 must not share a time/room with CS415
NotSameRoom constraint: MATH531 must not share a time/room with MATH532
NotSameRoom constraint: MATH531 must not share a time/room with CS415
NotSameRoom constraint: MATH532 must not share a time/room with CS415
```

```
-----------------------------------------
Course: CS730
Instructor: Rubenstein
Size: 30
Room: Nesmith 326: 19:10 - 20:00 (30 seats)
h-val: 0
-------------------------------------------
Course: MATH531
Instructor: Feldman
Size: 15
Room: Kingsbury 319: 11:10 - 12:00 (15 seats)
h-val: 0
-------------------------------------------
Course: MATH532
Instructor: Feldman
Size: 15
Room: Kingsbury 319: 10:10 - 11:00 (15 seats)
h-val: 0
-------------------------------------------
Course: CS415
Instructor: Johnson
Size: 60
Room: Horton 4: 19:10 - 20:00 (200 seats)
h-val: 0
-------------------------------------------


**Traversed 58 states
```

Upon inspection of the constraints which were generated, we can see that indeed every necessary constraint was created. The constraints here are represented in a natural-language representation for each, but within the program each one is actually a separate object which contains a relation between two other objects that must satisfied. If these constraints are followed like we would hope, then the generated schedule should be an acceptable one. In fact, when looking at the generated schedule, we see that all constraints are satisfied and so this is a good schedule. Most importantly, all classes are in rooms that fit them, no classes are assigned to the same time/room value, and classes taught by the same teacher do not overlap. However, this time it had to traverse 58 states in order to find a consistent solution.

We will now add in the feature of 1-consistency. This produces the same set of constraints, and the following schedule:

```
-----------------------------------------
Forcing 1 consistency...done
Course: CS730
Instructor: Rubenstein
Size: 30
Room: Nesmith 326: 19:10 - 20:00 (30 seats)
h-val: 0
-------------------------------------------
Course: MATH531
Instructor: Feldman
Size: 15
Room: Kingsbury 319: 11:10 - 12:00 (15 seats)
h-val: 0
-------------------------------------------
Course: MATH532
Instructor: Feldman
Size: 15
Room: Kingsbury 319: 10:10 - 11:00 (15 seats)
h-val: 0
-------------------------------------------
Course: CS415
Instructor: Johnson
Size: 60
Room: Horton 4: 19:10 - 20:00 (200 seats)
h-val: 0
-------------------------------------------


**Traversed 6 states
```

With 1-consistency, we removed all unary-constraint-violating values from the domains, and still produced the same schedule. As expected, 1-consistency substantially improved processing time – rather than analyzing 58 states, we only had to analyze 6. Surprisingly, this is only one more state than we traversed when we had no constraints at all.

We will now introduce the heuristic values for variable ordering, based on most-constrained variable calculations. We now get the following schedule (note the h-vals):

```
-----------------------------------------
Course: MATH531
Instructor: Feldman
Size: 15
Room: Kingsbury 319: 11:10 - 12:00 (15 seats)
h-val: 7
-------------------------------------------
Course: MATH532
Instructor: Feldman
Size: 15
Room: Kingsbury 319: 10:10 - 11:00 (15 seats)
h-val: 7
-------------------------------------------
Course: CS730
Instructor: Rubenstein
Size: 30
Room: Nesmith 326: 19:10 - 20:00 (30 seats)
h-val: 6
-------------------------------------------
Course: CS415
Instructor: Johnson
Size: 60
Room: Horton 4: 19:10 - 20:00 (200 seats)
h-val: 4
-------------------------------------------

**Traversed 6 states
```

The h-vals are correct – they should correspond to the number of constraints each course is involved in. MATH531 and MATH532 have the highest h-vals, as they should – they are the only courses taught by the same teacher, and by a teacher with building and time preferences, at that. We can tell by the order of the variables in the end state that they were indeed assigned values in order of their h-vals. Surprisingly, though, this does not seem to reduce the number of states traversed – we still go through 6 states as with just 1-consistency. Speculation of why this occurred will be saved for the conclusion of this paper.

Lastly, we should show what happens when an impossible assignment is found. Consider if *courses* was modified to be:

```
courses
```

```
CS730
Rubenstein
300
MATH531
Feldman
15
MATH532
Feldman
15
CS415
Johnson
60
```

Note that CS730 now expects 300 students. There is no room available which will hold this many students. Running the program produces the following output:

```
[greg@fozzie Project]$ java Scheduler
Constraints:
Size constraint: CS730 must fit room
Size constraint: MATH531 must fit room
Size constraint: MATH532 must fit room
Size constraint: CS415 must fit room
Building constraint: MATH531 must be in Kingsbury
InSpan constraint: MATH531 must be in time span 8:00 - 12:00
NoOverlap constraint: MATH531 must not overlap course  MATH532
Building constraint: MATH532 must be in Kingsbury
InSpan constraint: MATH532 must be in time span 8:00 - 12:00
Building constraint: CS730 must be in Nesmith
InSpan constraint: CS730 must be in time span 15:00 - 21:00
NotSameRoom constraint: CS730 must not share a time/room with MATH531
NotSameRoom constraint: CS730 must not share a time/room with MATH532
NotSameRoom constraint: CS730 must not share a time/room with CS415
NotSameRoom constraint: MATH531 must not share a time/room with MATH532
NotSameRoom constraint: MATH531 must not share a time/room with CS415
NotSameRoom constraint: MATH532 must not share a time/room with CS415

-----------------------------------------
Forcing 1 consistency...done
Calculating h-vals...done
-----------------------------------------
Impossible to make legal assignments!
```

```
**Traversed 21 states
```

The program correctly reports that no legal assignment is possible for this set of courses, rooms, and teachers.

The program, as presented, has several strengths. For one, this approach does indeed generate a good schedule in which no constraints are violated. This is the most important requirement, and it succeeds here. Additionally, it does this relatively quickly due to the arc consistency and heuristic techniques in place. The program is also sufficiently easy for a human to use. They merely need to enter in the data pertaining to courses, rooms, and teachers. Since all constraints are automatically generated, no knowledge of the generated constraints is necessary, and, in fact, no knowledge of the fact that the program even uses constraint satisfaction at all is necessary for using the program.

However, the program is not without its weaknesses. Foremost, the program cannot handle “preference constraints”. That is, it will only generate a schedule in which every constraint is satisfied. There is no way to identify a certain constraint as ideal, but not necessary. This would be most useful for the building constraints. Ideally, a teacher would like to be in the building that their office is in, but if that is not possible, the program should be able to schedule their classes for elsewhere. Preferences could be implemented in the future in one of two ways. First, they could be implemented by not considering preference constraints as actual constraints during schedule generation, and instead generate *every* possible schedule, then see which one satisfies the highest number of preference constraints. The drawback here is that this would be time-consuming. Or, they could be implemented with heuristics, where values (rather than variables) are ordered by how many preference constraints they satisfy.

Other areas for improvement are that the program could support variable-length classes, and scheduling over multiple days. For instance, if it was not possible for schedule a class for one hour on a Monday-Wednesday-Friday schedule, it could try for an hour and a half on Tuesday and Thursday or 3 hours on Wednesdays. This is how scheduling is actually done for colleges, and so the program would be more applicable to real-life scenarios if it did this.

Other weaknesses in the system are that it does not use local search. Local search would make the program less memory intensive, and it could potentially find a solution faster. Also, the heuristic in use obviously does not seem to improve the generation speed. Work could be done to find a better heuristic for this particular problem.

This program turned out to be just as interesting to develop as it had seemed that it would be. It was interesting to see how a rather simplistic algorithm (backtracking search) could be used to generate a schedule that takes so many different constraints into account. Despite the algorithm being simplistic, several challenges were met during the development of the system which kept the process interesting. Namely, how to represent constraints in memory such that they could be evaluated took quite a bit of brainstorming. Implementing the various optimizations, namely 1-consistency and heuristics, were also less than straightforward.

Which brings us to the final speculation on how heuristics affect the speed of generation. It was observed that introducing heuristics did not affect the number of states traversed at all (despite the fact that the heuristics allowed us to first consider courses which were involved in the most constraints). Why did this happen? It is my conjecture that the primary reason is that the calculated h-vals were all very close. Most of the constraints in this system are common to all courses – being in a room that fits the class, not overlapping with other classes in the same room, etc. Since they are all very close, using a heuristic based on the number of constraints a variable is involved in will not affect the speed, if at all. This leads to the hypothesis that the other common heuristic used in constraint satisfaction – number of remaining values – would also not affect speed much, as this value will also be very similar for all variables. The reason that 1-consistency helped so much is that many of the constraints are unary, and so removing unary constraint-violating values makes each domain significantly smaller – which directly affects the search tree.

Working on this project was very interesting and went a long way towards explaining many areas of constraint satisfaction (as well as tree searching in general). There is now a better sense for the strength of these techniques, and also for the weaknesses inherent in it. The possible improvements that can be made (and further comparisons of additional heuristics and arc consistency algorithms) are exciting, and work will continue on this program to further explore these enhancements in the future.

References / Reading

Bartak, R. (2001). Theory and practice of constraint propagation. In *Proceedings of the Third Workshop on Constraint Programming for Decision and Control (CPDC-01)*, pp 7-14, Giliwice, Poland.

Bistarelli, S., Montanari, U., and Rossi, F. (1997) Semiring-based constraint satisfaction and optimization. *Journal of the Association for Computing Machinery, 44(2),* 201-236

Bitner, J.R., and Reingold, E.M. (1975). Backtrack programming techniques. *Communications of the Associations for Computing Machinery, 18(11)*m 651-656

Brelaz, D. (1979). New methods to color the vertices of a graph. *Communications of the Associations for Computing Machinery, 22(4)*, 251-256.

Dechter, R., and Frost, D. (1999). Backtracking algorithms for constraint satisfaction problems. Tech. Rep., Department of Information and Computer Science, University of California, Irvine.

Kumar, V. (1992). Algorithms for constraint satisfaction problems: A survey. *AI Magazine, 13(1),*32-44.

Mairnho, et al. (1999). Decision Support System For Dynamic Production Scheduling. *http://www.dei.isep.ipp.pt/~alex/Artigos/isatp99/ISATP99.htm*

Moore, A. (2002). Constraint Satisfaction and Scheduling. *http://www.cs.cmu.edu/~15381/Lectures/constraint07.pdf*

Sadeh, N. (1995). Micro-Boss: Dual-Use ARPI Technology Helps Improve Manufacturing Performance. *IEEE Expert, February 1995.*