Constraint-Based Intelligent Scheduling

Donald Smith*
Scott D. Goodwin

Technical Report CS-95-02

June 1995

Department of Computer Science
University of Regina
Regina, Saskatchewan
S4S 0A2

ISSN 0828-3494
ISBN 0-7731-0300-7

*Mitel, P.O. Box 13089, 360 Legget Drive, Kanata, Ontario, Ottawa, K2K 1X3

Part I
Introduction

1 Constraint Logic Programming

Constraint Logic Programming (CLP) is a new class of programming languages combining the declarativity of logic programming with the efficiency of constraint solving [FRU93].

One notable advantage to CLP beyond the fact that some domains are more amenable to a constraint based description is the concept of *"constrain and generate"*. As the CHIP [DVS88] project was the first to demonstrate, the performance problems of *"generate and test"* can be overcome by reversing the process and only generating what the constraints will allow.

1.1 Domains

To accommodate constraints in logic programming the unification algorithm is replaced with a decision procedure that can detect when a constraint or set of constraints is satisfiable. The size and type of the domain will greatly affect the behavior of the decision procedure. For example, if the domain is over the range 0...1 (the *boolean* domain), the decision procedure is able to make assumptions that a decision procedure over the real domain would not be able to make.

Many CLP languages will only work with linear constraints. A linear constraint takes the form:

$A_1X_1 + A_2X_2 + \ldots + A_nX_n \bowtie B_1Y_1 + B_2Y_2 + \ldots + B_mY_m$. The A_i's and B_i's are constants, the X_i's and Y_i's are variables and $\bowtie \in \{\leq, <, \geq, >, =, \neq\}$.

Some CLP languages have the ability to deal with non-linear constraints during execution, however, they must be reduced to linear constraints before they can be solved.

1.2 Search in CLP

Searching in a CLP framework requires the consideration of two points:

- On which aspect do we want to make an assumption? In other words, which constraint do we wish to "narrow" next?

- What assumption should we make? In other words, what values should we consider for the constraint?

One common strategy is known as the *first fail principle*. When using this approach we select the variable with the smallest domain to explore with the hope that it will find a solution or fail sooner.

Another strategy is known as the *binary chop*. When there are no relatively small domains to explore, we could take a variable and *assume* that a solution lies in one half of its current domain. If no acceptable solution could be found in that half, we examine the other half.

1.3 Relaxation of Constraints

A major obstacle in the creation of a constraint based scheduler in a CLP language like clp(FD) is the ability to *relax* constraints when it becomes necessary. According to Fox in [FOX87]:

> Any theory of constraint-directed reasoning must include a specification of how to relax constraints.

This fact holds especially true in scheduling since some constraints may not be attainable. For example, given various resource limitations it may be true that in order to satisfy the due date of one widget, we have to forsake the due date of another.

2 Scheduling

Scheduling is similar to the planning problem of AI. According to Fox in [FOX90c]:

> Planning selects and sequences activities such that they achieve one or more goals and satisfy a set of domain constraints.

In the same paper, Fox explains:

> Scheduling selects among alternative plans, and assigns resources and times for each activity so that they obey the temporal restrictions of activities and the capacity limitations of a set of shared resources.

A key difference between planning and scheduling is that planning aims to find a feasible solution to a problem; whereas scheduling is an optimization problem.

2.1 The classic "Machine Shop" problem

Throughout this project we will be referring to the classic "machine shop" example that is described in many scheduling papers. It has become a popular example much like the blocks world has become a standard example for planning papers.

The "machine shop" example is also commonly referred to as the "factory model" and the "job-shop" model.

A typical machine shop example will have a variety of machines M_1 through M_n, and a variety of items I_1 through I_m that must be manufactured using those machines. On an abstract level, we have resources $R_1 \ldots R_n$ and jobs $J_1 \ldots J_m$ each of which requires a series of activities $A_{j1} \ldots A_{jk}$ (and activities use resources).

The task of a machine shop scheduler is to schedule the production of the items given various constraints:

- No machine M_i can be working on both I_a and I_b where $a \neq b$ during the same discrete time interval.
- No I_j can be worked on by M_a and M_b where $a \neq b$ during the same discrete time interval.
- In some examples, it is possible for machine M_i to be replaced by $M_k (i \neq k)$ where M_k is of the same machine class as M_i. For example, *drill2* may be used in lieu of *drill1*, and so on.

The machine shop problem is a good example to use for many reasons.

- While appearing to be almost trivial, the machine shop example can exemplify many typical scheduling problems including resource bottlenecks, temporal restrictions and competition for resources.

- The machine shop example can exemplify many common scheduling goals such as lateness minimization, flow time minimization and cost minimization.

- There is a large demand for machine shop schedulers. There is real data to use in testing, and economic support for research that directly aids the machine shop scheduling problem. For example, in [FOX90], Fox acknowledges the US Air Force, Westinghouse Electronics, IBM, Schlumberger Co., Boeing and McDonnell Douglas Corporation. Clearly all of these organizations have great interest in machine shop scheduling.

2.2 Complexity of scheduling

Scheduling has already been defined as a form of planning. In fact, it is a more complicated form of planning since there are more aspects to consider. In [FOX90] Fox states that a machine shop producing 85 objects with eight operations each has over 10^{880} possible schedules. In fact, Fox in [FOX87] shows that the general complexity of scheduling is $(r * o!)^n$ where r is the average number of machine replacement possibilities, o is the number of orders to be processed and n is the average number of operations per order.

2.3 Evolution of Scheduling

In this section we will detail the major advances made in scheduling during the last twenty years. Figure 1 is a graphical representation of this evolution compiled from the various papers we have studied.

2.3.1 Scheduling as a search problem

Many early and simple schedulers treat scheduling as a search problem. However, heuristics must be applied during the search since scheduling is very complex (see Section 2.2). These heuristics are often too general to produce high quality schedules. To compensate for the quality, these kinds of schedulers intentionally overestimate activity times to allow for error. This results in under utilized machines and unnecessary extra inventory.

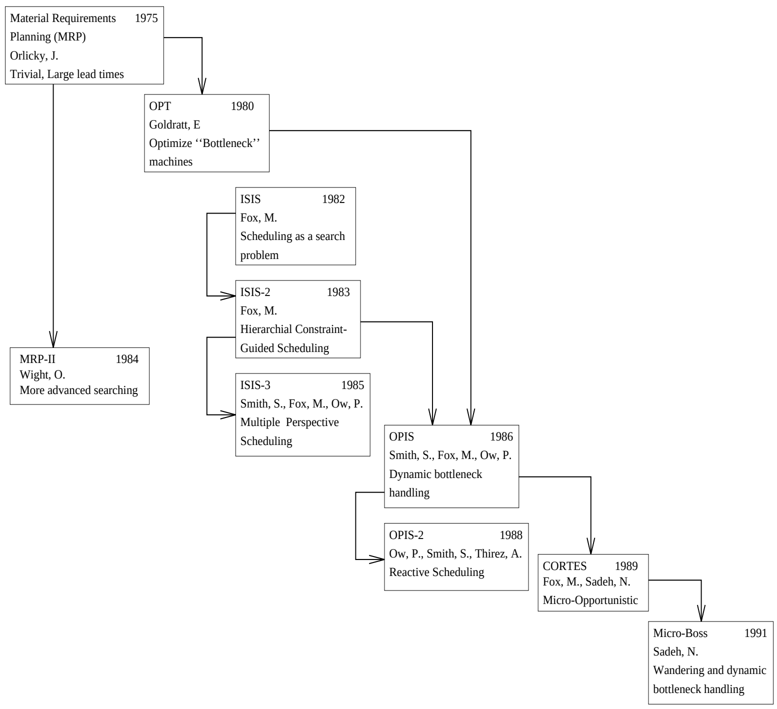

Figure 1: History of Scheduling Advances

Search based schedulers are good "engines" to use within more advanced algorithms. For example, as Figure 1 shows, MRP and ISIS were expanded upon to create MRP-II, ISIS-2 and ISIS-3.

2.3.2 Bottleneck sensitive schedulers

During the OPT project, Goldratt realized that in many machine shops, certain machines would act as a bottleneck in production [GOL80]. A bottleneck machine is one whose use is particularly high and whose limited capacity restricts the ability to produce a good schedule. By optimizing the schedule for these machines, better schedules will be produced. OPT was one of the first schedulers to take advantage of this.

2.3.3 Dynamic bottleneck sensitive schedulers

In 1986, Smith, Ow and Fox took the notion of bottleneck machines one step further [FOX90]. They realized that bottlenecks can occur dynamically in a system, and that by optimizing one bottleneck, another may be created. Their OPIS system was the first scheduler able to realize the existence of bottlenecks while in the process of creating the schedule, and deal with them dynamically.

2.3.4 Micro-opportunistic schedulers

The type of scheduler mentioned in Section 2.3.3 was later termed a macro-opportunistic scheduler [SAD91]. The difference between the newer micro-opportunistic schedulers and macro-opportunistic is the ability of micro-opportunistic schedulers to optimize various bottleneck machines as a group, examining the effects optimizing one bottleneck has on others; whereas, macro-opportunistic schedulers naively optimize bottleneck machines one at a time. CORTES [FOX90a] and Micro-Boss [SAD91] are two good examples of micro-opportunistic schedulers.

2.3.5 Present day schedulers

Since the late 80's, scheduling has taken many directions. One popular trend is uncertainty in scheduling [CHI90]. This kind of research attempts to produce schedules that are sensitive to probabilistic machine failures, employee errors and so on. These kinds of schedulers rely on concepts such as fuzzy logic to accomplish their tasks.

Another popular trend in scheduling is the concept of iterative repair [ZWE91]. Iterative repair schedulers allow users to report on progress dynamically so that the scheduler can make any necessary changes. For example, imagine a machine shop scheduler that has produced a daily schedule. A user could enter a progress report at regular time intervals (certain work is ahead of schedule, certain work is behind schedule, certain machines are under repair, etc.). The scheduler should be able to re-adjust the schedule according to the input.

Since scheduling ultimately deals with combinatorially explosive search, the application of heuristics is critical. [FOX90b] and [JOH92] are two good papers dealing with scheduling heuristics. Other search related research is applicable to scheduling as well including distributed and parallel computing, hill climbing strategies and space management.

3 ISIS-2: A Case Study Scheduler

IRIS-2 is a constraint based scheduler designed by Fox in the 1980's. It is described in [FOX87] which is a summary of his Phd thesis.

IRIS-2 is significant to us because Fox uncovers and describes most of the issues involved in constraint based scheduling. This includes the concept of constraint relaxation, temporal concerns, search issues and strategies, success criteria, and so on.

A major drawback for us is that IRIS-2 was implemented in a language called the "Schema Representational Language" (SRL). This language is based heavily on Minsky's formal definition of frames, and hence does not map well to our implementation language clp(FD).

4 Evans' Finite Domain Scheduler: A Case Study Scheduler

Evans describes in [EVA92] a finite domain scheduler designed in DECISION-POWER Prolog. Unlike SRL, DECISIONPOWER Prolog should be easily mapped to clp(FD).

Evans explains that scheduling should be done in conjunction with real time input from the user. In other words, the user should be able to inform the scheduler of machine outages, changes of priority, and so on, and have the scheduler re-schedule the affected regions. He does not, however, use any intelligent repair method. Instead all activities from the earliest affected temporal interval are re-scheduled. For example, if the user made a change to the schedule and the earliest temporal interval affect by that change was $T = 10$, then all the activities which occur during or after the time $T = 10$ will be re-scheduled.

According to Evans, constraints can be categorized as being conjunctive or disjunctive. Conjunctive constraints are constraints that must always be true. For example, the execution of T_i must come before the execution of T_{i+1}. Disjunctive constraints are "multiple-choice" constraints. For example, the execution of T_i must be performed on either M_x, M_y or M_z. Disjunctive constraints generally will cause backtrack points during execution whereas conjunctive constraints are hard and fast rules.

Evans uses lists as his main data structure. Task durations, machine class hierarchies, job profiles, and so on are all represented in list form. He explains that maintaining a list of machine reservation intervals was responsible for a significant (3x) slow down as compared to when he used finite domain constraints.

Evans lists some ways that he achieved better optimization:

- Rank the jobs by order of importance and schedule the most important jobs first. When creating the schedule, the resource contention is relatively low for the first few jobs and grows as more jobs are scheduled. By scheduling the *important* jobs first, better schedules should be created. Of course, it may not be easy to have suitable criteria for ranking jobs.

- Implement the *preference* concept as described in [FOX87]. For example, assume we had two machines that performed the same task. We may *prefer* to schedule jobs on one machine over the other because it may be newer and less likely to break down.

- Schedule tasks on the least contented resources. For example, if two machines perform the same task and a new task has to be scheduled using those machines, we may choose the machine with the fewest jobs scheduled on it. By doing this, we reduce the possibility of fragmentation and bottleneck anomalies.

- Use built in predicates to minimize or maximize certain domain variables. For example, if we want to minimize the flowtime of certain jobs, we can explicitly program this into the scheduler.

A final thing to note about the Evans scheduler is his use of graphics and user interfaces. When a schedule is built, a gantt chart is displayed on the screen. The user can drag and drop scheduled tasks to their liking and the scheduler will redesign the schedule around the changes. The user can also introduce machine outages and expected repair times at which point the scheduler will reschedule. As explained above, it does use an intelligent repair method.

5 van Hentenryck on Disjunctive Constraint Scheduling

In [VAN89], van Hentenryck describes the creation of a scheduler which schedules the creation of a bridge. There are resources to be considered such as an excavator, concrete mixer and crane. As well, there is a definite order in which the bridge must be assembled.

van Hentenryck describes what he calls a *critical task* as being a task that if delayed, by whatever extent, will extend the minimal duration of the schedule by the same amount. We can find *critical tasks* by first calculating t_i and T_i where t_i is the earliest possible start time for task i and T_i is the latest time that task i can start if the schedule is to achieve a specified minimum duration. T_i and t_i are calculated as follows:

$t_i = max(t_j + d_j)$ For all j that precede task i.

$T_i = min(T_j - d_i)$ For all j that task i must precede.

The difference between T_i and t_i is known as the *slack* time. Clearly if $T_i - t_i = 0$ then there is no slack time, and hence i is a critical task.

Another interesting point that van Hentenryck makes is that we can initially set the domain range for each time variable used in the scheduler as $0 \ldots lg$ where lg is the summation of all task durations. The reason for this is that the longest schedule that could be created would involve the sequential execution of each task.

van Hentenryck also clarifies one of the advantages gained when using a constraint logic programming language to solve the scheduling problem:

> What is interesting to stress is that no explicit programming is necessary. The user simply states the constraints and a generator of values for the variables.

6 ODO: A Case Study Scheduler

ODO is a constraint based architecture for representing and reasoning about scheduling problems [DAV93]. ODO was written using C++ at the University of Toronto by Eugene Davis and Mark Fox.

The ODO system uses a simple command language that allows the user to describe the problem, which textures to measure and which search strategy to use (the texture concept is explained in Section 6.2). The output from the ODO system is in the form of the Gantt chart.

6.1 ODO: Constraint Representation

ODO's constraint model consists of a collection of objects, variables and constraints. The objects contain various variables and constraints and occasionally are used to store measured texture information. Figure 2 is a graphical representation of an ODO constraint network.

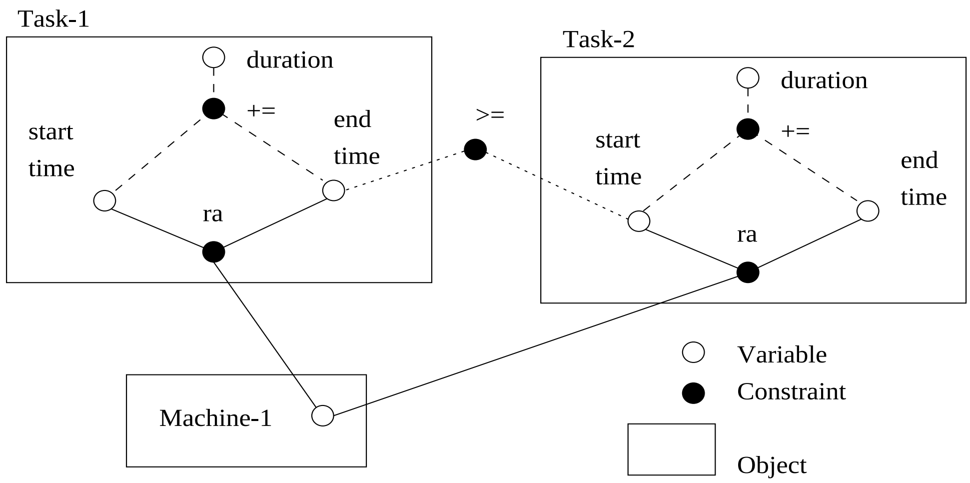

Figure 2: Constraint network

Figure 2 shows two *task* objects and one *machine* object networked via various constraints. We see that the *end time* variable in object *Task-1* is constrained to the *start time* variable in object *Task-2*. We also see that within each *Task* object that the *end time* is constrained to the *start time* by the duration of the task. The final constraint represented in Figure 2 is their association to *Machine-1* and that all temporal constraints within this machine must be obeyed.

6.2 Textures

The notion of a texture was first proposed in [FOX89]. A texture is a property of a constraint graph, and heuristics in such a system are a function of them.

Fox states:

> ...for search to be well focused, that is to decide where in the problem topology an operator must be applied, there must be features of the topology that differentiate one subgraph from another, and these features must be related to the goals of the problem. We have identified and are experimenting with such features that we call problem *textures*.

Here is the motive behind some textures given by Fox in his initial paper on them:

- To make the next modification on the state (which is an instance of a constraint graph) with the most variables involved in a constraint. Since the goal state is a state where all constraints are satisfied, the more constraints a state has satisfied, the closer to the goal it should be.

- To satisfy the most important constraint. Using this texture should create more optimal schedules. This texture can be used to give the scheduling of resource bottlenecks priority.

Here is the motive behind some textures used by ODO:

- To make the next schedule modification so as to reduce the probability of backtracking being invoked, and if backtracking appears to be inevitable, then "get it over with" as soon as possible so as to reduce thrashing.

- To choose the next modification that will reduce the number of violated constraints by the greatest amounts.

Textures can be difficult or expensive to calculate so usually they are estimated.

By allowing the user to decide which textures to use we are really allowing them to easily change the heuristic behavior.

6.3 Search

ODO allows the user to select from systematic or nonsystematic search. Systematic search is exhaustive, eventually all states will be visited. Nonsystematic search is not necessarily exhaustive (for example hill climbing search), and may cycle. However, nonsystematic search requires less maintenance and memory. For example, in the beam search, only the n best states are explored at each search level where "best" is defined by an evaluation function. With only n nodes per search level the memory usage and search time are linear in complexity. So while nonsystematic search may be faster and require less memory, it is not guaranteed to find a solution.

Figure 3 shows the problem solving flow chart used in ODO.

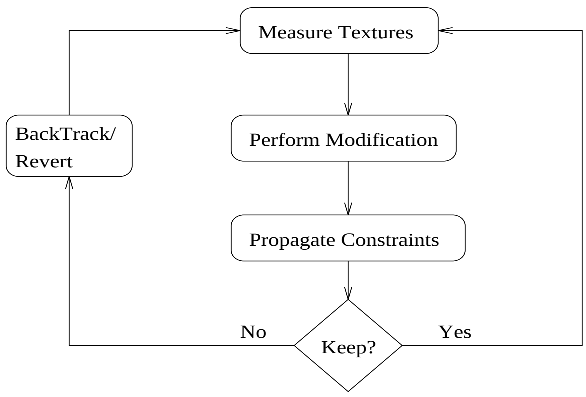

Figure 3: Search flow chart

6.4 Built-in Language

ODO uses a simple command based language to interface with ODO. The user can decide which textures to use, which search strategy to use and easily define the initial state, goal state and other such parameters.

There are two main sections of input ODO must receive before generating a plan.

6.4.1 Problem Declaration

Declaring Resources The resources of the domain world must be defined.

First, resource classes are defined, and then instances of those classes are defined. If a shop had two saws, for example, the class *saw* would be defined, and then instances *saw1* and *saw2*.

Declaring Tasks The tasks to be accomplished must be described. This is done by declaring the overall tasks to be accomplished and then adding successively adding detail. For example, the fact *Job1* must be done is declared, and then the fact that *Activity1* and *Activity2* must be done to accomplish *Job1* must be introduced.

Declaring Temporal Constraints All temporal related details are described next. This includes duration of tasks, order of tasks and start and end time restrictions.

Declaring Resource Constraints Which tasks use which resources must be defined. For example, if *Activity1* uses a *saw*, then this fact must be defined.

6.4.2 Policy Declaration

Variable and Value Selection Next the user must define the strategy for deciding which constraint to satisfy, and how to constrain it. This is accomplished by defining which strategy to use at each of four phases:

- The strategy for selecting a set of variables for consideration. Unassigned variables, random variables and all variables are some examples.
- Once a set of variables has been generated a set of possible values is generated for each variable. All possible values and random values (from a list of valid values) are examples.
- Now that a set of possible variable/value pairs exists, we must decide how to rate them. This is done by defining the *scoring function*. Cost lookahead is one example.
- Now that the variable/value pairs are ranked, we must decide which to use. Minimum score, maximum score, random and arbitrary are examples. Note that when randomly or arbitrarily pick a pair we can define the scoring function as *none* since the results are not used.

Constraint Propagation Propagation means to update related variables in a constraint network. For example, if we select a value for variable X we must update the new possible values for all variables constrained to X, and so on. We have the ability to decide which propagation techniques are to take place. For example, we can propagate all connected temporal constraints, the entire network, or nothing.

Accept Criteria Now that we have generated a new state, we must decide if we are going to accept it. There are only two functions presently implemented. Either you accept the state always, or only if it has not emptied any variables set of possible values.

Cost Function The cost function is an evaluation of a state. There are two cost functions in ODO. Either the number of unassigned variables is returned or the number of violated constraints.

Search Termination Criteria This expression is used to decide if the search should be terminated. For example, if we have defined the cost function to return the number of unassigned variables, we may want the termination criteria to be $(cost = 0)||exhaustedsearch$.

6.5 Summary of personal experiences

ODO is a good example of how far research has taken the scheduling problem in the past 20 years. The ability to easily select search strategies, termination criteria and other such things makes ODO very flexible and usable in many situations. C++ programmers have the ability to update existing functions, or add their own, making ODO even more flexible.

The output of ODO could be better. For example, the Gantt chart could be color coded to distinguish between activities. ODO was written with the intention of using it with some form of interface module. Without this module (it has not been created yet), ODO seems a bit cumbersome, most notably when trying to extract output. When a schedule is found, ODO does not automatically display the schedule, and since the Gantt chart is not color coded, we must print each task one at a time to see when it starts and ends.

In [SCH95] Schaeffer notes that ODO must be given a maximum time for each problem. If the maximum time is to short ODO will run indefinitely and if the maximum time is too big then ODO will generate poor schedules.

In other words, when entering the problem you must already have a good idea on how long the schedule should be.

Part II
Problem

The need for better schedulers is obvious. CLP is a promising programming domain for effective schedule creation.

In this project we:

- Research issues involved in building a constraint based scheduler.
- Implement a constraint based scheduler and apply it to a manufacturing process problem. clp(FD) (Constraint Logic Programming, Finite Domain) is the implementation language.

Here are some research areas for future consideration:

- Investigate some popular constraint logic programming packages, and note any advantages and disadvantages to each.
- Discover ways to use clp(FD) to simulate the relaxation of constraints when necessary.
- Discover areas where distributed computation could be advantageous in the schedule making process.

Part III
Data Tested

Our chosen main domain is the job shop scheduling problem. The example problems help demonstrate scheduling issues we have discovered including:

- Multiple machines of the same class,
- High contention for machine resources,

- Specific goals such as lateness minimization, flow time minimization or cost minimization.

Below we describe the job-shop scheduling problem we used when testing our scheduler.

```
 There is a machine shop with the following resources:
  - 2 saws   (saw1 and saw2),
  - 2 drills (drill1 and drill2),
  - 1 lathe  (lathe1).

 We must produce the following 5 widgets whose task
list and durations are given below:
  - widget1: saw   5, drill 5, lathe 5,
  - widget2: saw   5, drill 5, lathe 5,
  - widget3: drill 5, saw   5, lathe 2,
  - widget4: drill 5, saw   7, lathe 3,
  - widget5: lathe 4, drill 5, saw   5.

Durations are in discrete time units.  Note that widget1
and widget2 require the same operations, and hence we
are assuming they are in the same class.  In other words,
we are manufacturing two "widgetX's" called widget1 and
widget2.

We will run this data in our scheduler with three different
sets of due dates:

 data1: widget1 due: 15   data2: widget1 due: 50
        widget2 due: 20          widget2 due: 50
        widget3 due: 22          widget3 due: 50
        widget4 due: 25          widget4 due: 50
        widget5 due: 20          widget5 due: 50

              data3: widget1 due: 24
                     widget2 due: 22
                     widget3 due: 22
```

```
widget4 due: 30
widget5 due: 30
```

As well as the job-shop examples given above, we also tested our scheduler on two other scheduling examples.

One of the examples is van Hentenryck's bridge example given in [VAN89]. However, we do not implement his notion that some tasks must started within a certain time of some other tasks finish time. For example, A_i must be started within n time units of the completion of A_j. While this would not have been too difficult to implement, we simply did not have enough time.

The other example is known as the newspaper problem, and was discovered on the internet at an unknown site by a colleague, Wanlin Pang. The problem is described below.

```
Algy wakes up at 8:30am.  Bertie and Charlie wake up at 8:45am.
Digby wakes up at 9:30am.  All four people must read each of
four newspapers in the given order and duration listed below.
To save money, these four roomates only buy one copy of each
paper.  (Durations are in minutes).
         Algy           Bertie         Charlie        Digby
  1st  F. Times-60    Guardian-75    Express -5     Sun     -90
  2nd  Guardian-15    Express -3     Guardian-15    F. Times-1
  3rd  Express -2     F. Times-25    F. Times-10    Guardian-1
  4th  Sun     -5     Sun     -10    Sun     -30    Express -1

Given that these four students, to save gas, drive to school
together in Algy's car, create a schedule for reading the
newspapers that will get them to school nice and early.
```

Part IV
Approach

Scheduling can be viewed as either a satisfiability problem or an optimization problem. Scheduling as a satisfiability problem seeks to satisfy all, or as many as possible, constraints in a problem. Scheduling as an optimization problem, however, seeks to find the best solution to a problem. In the testing for this project, we treated scheduling as a satisfiability problem. Future work will view scheduling as an optimization problem.

The approach to the problem was to first read about, examine and try to understand other constraint based schedulers. We then implemented a "brute force" scheduler. By brute force we mean a scheduler which only uses only two "common sense" heuristics:

- Schedule the start time of a task greater than or equal to the end time of its prerequisite tasks.
- Obey resource capacities (in our example, only one activity per resource at any given time interval).

After that we implemented the critical task measure and the least constrained resource measure described in Part I.

We also implemented our own simple heuristic described as follows. Given two activities A_i and A_j which both use machine M_p, and given the sequencing, duration and due date constraints on A_i and A_j; when deciding whether to schedule A_i before A_j, or A_j before A_i, try to schedule the activity which has the earliest potential end time first. In the "brute force" implementation this decision is arbitrary.

Once we had the *brute force*, *critical task*, *least contended resource* and *disjunctive order* heuristics working, we combined the *least contended resource* heuristic with the *critical task* heuristic. Also, we combined the *least contended resource* heuristic with the *disjunctive order* heuristic. This gave us six different scheduling heuristics and heuristic combinations to test with our data.

Part V
Results

The results for each of the six data files using the six scheduling heuristics is given below. The table headings can be interpreted as follows.

Δ t The time, in milliseconds, the scheduler takes to find the first acceptable solution. This time is the mean of three executions on a DECStation running ULTRIX 4.3.

Δ schedule time The total running time of the schedule found, in discrete time units.

Mean flow time The mean flow time of the jobs in the data.

Mean early time The mean time a job is completed before it's due date.

bforce Brute force scheduling approach.

ct Critical task heuristic added to brute force scheduler.

do Disjunctive ordering heuristic added to brute force scheduler.

lcr Least contended resource heuristic added to brute force scheduler.

ctlcr Both the critical task and least contended resource heuristics added to the brute force scheduler.

dolcr Both the disjunctive ordering and least contended resource heuristics added to the brute force scheduler.

Table 1: Results from data1

	heuristic					
	bforce	*ct*	*do*	*lcr*	*ctlcr*	*dolcr*
Δt	655	855	720	205	270	220
Δ schedule time	25	25	25	25	25	25
Mean flow time	17.4	17.4	17.4	18.4	18.4	19.4
Mean early time	1.0	1.0	1.0	0.0	0.0	1.0

Table 2: Results from data2

	heuristic					
	bforce	*ct*	*do*	*lcr*	*ctlcr*	*dolcr*
Δt	175	207	192	200	235	210
Δ schedule time	49	48	33	39	48	25
Mean flow time	14.2	14.6	23.0	15.2	18.4	19.4
Mean early time	20.8	17.0	25.0	25.8	16.6	30.6

Table 3: Results from data3

	heuristic					
	bforce	*ct*	*do*	*lcr*	*ctlcr*	*dolcr*
Δt	435	570	510	215	255	240
Δ schedule time	27	27	30	25	25	25
Mean flow time	19.8	18.8	16.8	17.4	17.0	18.2
Mean early time	3.8	3.8	4.8	4.2	4.2	7.4

7 Interpretation of results

7.1 Job shop data

Tables 1–3 represent the job shop data described in Part III. The optimal schedule has a Δ schedule time of 25 time units.

In Table 1, since our due date constraints are set very close to the optimal solution, all of the schedulers find variations of the optimal schedule. It is interesting to note that the schedulers which use a *least contended resource* heuristic are able to find the solution between three and four times faster than those who do not. The reason for this is that in order for the due dates to be met we need a lot of parallel usage of the two saws and the two drills. The *brute force*, *critical task* and *disjunctive order* schedulers do not employ any special machine selection heuristics and therefore start off trying to schedule all the tasks on one of each machine. Only after much backtracking will the machine selection be widespread enough for a solution to be found.

Table 2 shows the results from an example where the due dates are relaxed. They are relaxed to the point that we can find a schedule using just one of each machine. As the table shows, since the *brute force*, *critical task*

Table 4: Results from bridge

	heuristic		
	bforce	*ct*	*do*
Δt	970	1115	1000
Δ schedule time	108	145	150

Table 5: Results from paper

	heuristic		
	bforce	*ct*	*do*
Δt	180	1210	190
Δ schedule time	195	196	180
Mean flow time	119.5	157.5	105.8

and *disjunctive order* schedulers do not have to do a lot of backtracking to find a schedule, they find a solution in roughly the same time as the *least contended resource* schedulers. However, the quality of the solutions is poor. Our schedulers will output the first schedule they find which meet all the constraints in the problem. If the constraints are slack, then the schedule will be slack. Notice, though, that when we combine some intelligent ordering with some intelligent machine selection (*dolcr* in Table 2), even though the due dates are slack, the solution found is quite good (optimal in this case).

Table 3 contains results from an example with due dates more restricted that the data used in Table 2 but more relaxed that the data used in Table 1. Notice the discrepancy in Δt between the *least contended resource* schedulers and the other schedulers. It falls midway between the discrepancy shown in Table 1 and Table 2, which is what we would expect given the relative tightness of the due dates.

In Table 2 and 3, notice how the *critical task* schedulers show a notable decrease in mean early time. This is opposite of what we would want a scheduler to do. However, the tables do not show that tasks which are more critical (as described in Section 5) are indeed scheduled earlier than with non *critical task* schedulers. The problem is that they are scheduled earlier at the expense of all other related jobs. We suspect that with larger and more complex examples, this would not be the case, and that the *critical* *task* heuristic would produce schedules with greater mean early times.

There do not appear to be any noticeable trends in the mean flow time statistic. We suspect that with larger and more complex example, we may begin to notice some more interesting features about the various scheduling heuristics.

7.2 Bridge data

Since there are no sub-activities for "jobs" in the bridge example, mean flow time is not a valid statistic to track. Also, since there is no hard and fast due date for this problem, mean early time is not valid either. Finally, since there is only one of each resource the *least contended resource* heuristic is useless. The critical task measure assumes a constant, large due date when doing calculations.

Table 4 contains the results from the bridge example. van Hentenryck claims that a schedule time of 104 time units is the optimal solution. Using a *brute force* approach provides us with a schedule very close to optimal. However, when we use ordering heuristics we create less favorable schedules. This can be explained in the fact that the brute force scheduler works from bottom to top, scheduling the earliest activities first and then scheduling on top of them. The ordering heuristics make the scheduler schedule in a non-linear order; scheduling some early tasks, then some late tasks, then some middle tasks and so on. The way the data is provided in this example, a straight forward linear scheduler appears to be the best approach.

7.3 Paper data

Since there are no real due dates in this problem there is no need to note the mean early time. Also, since there is only one of each newspaper, the *least contended resource* heuristic is useless. The critical task measure assumes a constant, large due date when doing calculations.

The optimal solution provided with the problem description is three hours (180 time units). Table 5 shows that our schedulers were able to come up with good schedules and the *disjunctive ordering* heuristic was able to point to an optimal solution.

It is unknown why Δt is so much larger with the *critical task* heuristic than with the other two schedulers. We suspect some sort of error in the data (not the scheduler), further investigation is required.

Part VI
Conclusions

- Good heuristics are key to providing good schedules, fast.
- *Disjunctive ordering* and *least contended resource* heuristics are simple yet powerful heuristics to employ.
- The *least contended resource* heuristic finds good solutions faster. The more restricting the due dates, the more effective this heuristic becomes.
- The *disjunctive ordering* heuristic proved to be a clever way to improve schedules. However, in examples where straight forward linear solutions are best, the heuristic causes a slight degradation in schedule quality.
- The *critical task* heuristic does not appear to work well with our data, however there is hope that it will work well with larger examples. At this level, the cost to benefit ratio is not acceptable. In fact, it quite often makes schedules worse than a simple *brute force* schedule.
- Our *brute force* scheduler provided as a good base for rapid prototyping the various heuristics we tried.
- Our job shop example data was suitable to show trends in results, and worked well with rapid prototyping, however, it was still not large enough to demonstrate the usefulness of the *critical task* heuristic. We suspect that an example with roughly 50 jobs and 7 machines is needed.
- We were quickly able to define examples other than the job shop problem, and were able to find good solutions quickly.

References

[BEN93] F. Benhamou and A. Colmerauer (eds.), (1993), Constraint Logic Programming, Selected Research, MIT Press.

[COY90] R. Coyne, M. Rosenman, A. Radford, M. Balachandran and J. Gero, (1990), Knowledge Based Design Systems, Addison-Wesley Publishing Company.

[CHI90] Whay-Yu Chiang and M. Fox, (1990), Protection against Uncertainty in a Deterministic Schedule, Fourth International Conference on Expert Systems in Production and Operations Management, Hilton Head, Sth Carolina.

[DAV93] E. Davis and M. Fox, (1993), ODO: A Constraint-based Architecture for Representing and Reasoning About Scheduling Problems, Proceedings of ISCAI '93 Workshop on Production Scheduling.

[DVS88] M. Dincbas, P. Van Hentenryck, H. Simonis, A. Aggoun, T. Graf and F. Berthier, (1988), The Constraint Logic Programming Language CHIP, In Proceedings on Fifth Generation Computer Systems FGCS-88, Tokyo, Japan IN [FRU93].

[EVA92] O. Evans, (1992), Factory Scheduling using Finite Domains, In Logic Programming in Action, LNCS 636, pages 45-53, Springer-Verlag.

[FOX87] M. Fox, (1987), Constraint-Directed Search: A Case Study of Job-Shop Scheduling, Morgan Kaufmann Publishers, Inc.

[FOX89] M. Fox, N. Sadeh and C. Baykan, (1989), Constrained Heuristic Search, Proceedings of Eleventh International Joint Conference on AI, Volume 1, August 1989.

[FOX90] M. Fox, (1990), Constraint-Guided Scheduling–A Short History of Research at CMU, Computers in Industry 14 (1990) 79-88.

[FOX90a] M. Fox and Katia P. Sycara, (1990), Overview of Cortes: A Constraint Based Approach to Production Planning, Scheduling and Control, Fourth International Conference on Expert Systems in Production and Operations Management, Hilton Head, Sth Carolina.

[FOX90b] N. Sadeh and M. Fox, (1990), Variable and Value Ordering Heuristics for Activity-Based Job Shop Scheduling, Fourth International Conference on Expert Systems in Production and Operations Management, Hilton Head, Sth Carolina.

[FOX90c] M. Fox and N. Sadeh, (1990), Why is Scheduling Difficult? A CSP Perspective, Proceedings of the 9th European Conference on AI, 1990.

[FOXG91] M. Fox and N. Sadeh, (1991), Why is Scheduling Difficult? A CSP Perspective, (paper was found in a collection of IRIS related papers, exact publisher and date unknown at present time).

[FOX94] M. Fox and Zweben, M. (eds), (1994), Intelligent Scheduling, Morgan Kaufmann Publishers.

[FRU93] T. Frühwirth, A. Herold, V. Küchenhoff, T. Le Provost, P. Lim, E. Monfroy and M. Wallace, (1993), Constraint Logic Programming–An Informal Introduction, ECRC Technical Report ECRC-93-5.

[GOL80] E. Goldratt, (1980), Beyond MRP: Something better is finally here, Speech to APICS National Conference, in [SAD91].

[HEN89] Van Hentenryck, (1989), Constraint Satisfaction in Logic Programming, MIT Press.

[JOH92] M. Johnston, S. Minton, (1992), Analyzing a Heuristic Strategy for Constraint-Satisfaction and Scheduling, in [FOX94].

[SCH95] S. Schaeffer, (1995), A Naive User's Comments on ODO and TRAPS, WWW document, http://web.cs.ualberta.ca/~steph/IC-6/private/comments.ps.Z

[SAD89] N. Sadeh and M. Fox, (1989), Preference Propagation in Temporal/Capacity Constraint Graphs, Computer Science Technical Report CMU-CS-88-193.

[SAD91] N. Sadeh, (1991), Micro-Opportunistic Scheduling: The Micro-Boss Factory Scheduler, in [FOX94].

[SYC91] K. Sycara, S. Roth, N. Sadeh and M. Fox, (1991), Distributed Constrained Heuristic Search, IEEE Transactions on Systems, Man and Cybernetics, Fall 1991.

[VAN89] van Hentenryck, (1989), Constraint Satisfaction in Logic Programming, MIT Press.

[WAR74] D. Warren, (1974), WARPLAN: A System for Generating Plans, Memo 76, Department of Computational Logic, University of Edinburgh.

[ZWE91] M. Zweben, B. Daun, M. Deale, Scheduling and Rescheduling with iterative repair, in [FOX94].