Constraint Satisfaction with Delayed Evaluation

Monte Zweben and Megan Eskey
NASA Ames Research Center
M.S. 244-17
Moffett Field, California 94035
zweben@pluto.arc.nasa.gov, eskey@pluto.arc.nasa.govr

Abstract

This paper describes the design and implementation of a constraint satisfaction system that uses delayed evaluation techniques to provide greater representational power and to avoid unnecessary computation. The architecture used is a uniform model of computation, where each constraint contributes its local information to provide a global solution. We demonstrate the utility of the system by formulating a real-world scheduling problem as a constraint satisfaction problem (CSP).

1 Introduction

A constraint satisfaction problem (CSP) is characterized by a set of variables. Each variable has a possibly infinite domain of values and a set of domain constraints. A solution to a CSP assigns a value consistent with the domain constraints to each variable. A constraint is a related condition, (or predicate), that must hold among a set of variables. The algorithms used in most CSPs are those that find locally consistent assignments, that is, those that satisfy all constraints on a particular variable, in constraint networks by local propagation. The locally consistent assignments can be used to find a globally consistent solution, i.e., one that satisfies all constraints on *all* variables. Many previous real-world problems have been characterized as CSPs: scheduling [Brown, 1987, Fox, 1983, Johnston, 1989, Rit, 1986, Zweben, 1987], understanding line drawings [Waltz, 1975], electrical circuit analysis [Steele, 1980], temporal reasoning [Dean, 1985] and aspects of planning problems [Lansky, 1988, Stefik, 1980, Wilkins, 1984].

2 Constraint Satisfaction Problems

In this section, we describe the general constraint satisfaction algorithm (without delayed evaluation) and show how to improve the basic algorithm by using constraint propagation.

2.1 Uninformed Systematic Search

The most common algorithm for solving CSPs is systematic backtracking search. In its simplest form, the system takes an arbitrary ordering of variables and provisionally assigns values to them in a generate-and-test manner, until the set of variables has a globally consistent assignment. The basic algorithm is shown in Figure 1.

```
1. Order the variables from 1 to n.
2. Starting with the first variable, pick a
   value from its set of possible values.
3. Test the constraints associated with the
   variable to determine whether the value
   is valid.
4. If it is not valid, continue choosing
   values until the set is exhausted or
   until a valid one is found.
5. If a legal value is generated, continue
   with the next variable at step two.
6. Otherwise, backtrack to the previous
   variable, reject its current value and
   continue with its next value at step two.
7. If the system exhausts the first variable's
   possibilities, then there are no consistent
   assignments.
```

Figure 1: Systematic Backtracking Search

Intelligent, or *dependency-directed,* backtracking schemes, based on knowledge of inter-variable dependencies, and constraint propagation heuristics can be used to improve the basic algorithm's search performance [Stallman and Sussman, 1977]. Suggested schemes include "look-ahead", such as variable ordering or value ordering, or "look-back", such as constraint recording [Dechter and Pearl, 1987] or going back to the source of failure.

2.2 Propagation

Our system uses constraint propagation, that is, it filters values from a variable that are inconsistent with its constraints. For example, consider a binary constraint, C, over variables x and y, each having initial domains ranging over the integers. If C requires that all values of y be equal to 2x 4- 1, then all even integers would be filtered from the domain of y.

The propagation of constraints, or the application of filters, can be done by a number of look-ahead schemes including full look-ahead, partial look-ahead, and forward checking [Haralick and Elliot, 1980]. These heuristics are also referred to as maintaining arc-consistency or path-consistency in constraint networks [Mackworth, 1977, Dechter and Pearl, 1987, Gusgen *et al*., 1988]. Our system uses the weakest of these heuristic procedures, forward checking, because it has been shown to yield good performance on many problems [Dechter and Pearl, 1987]. The forward-checking heuristic is an extension of the *test* phase in the algorithm shown in Figure 1 and is inserted as an elaboration of step 3.

```
3a. Apply the constraint filters associated
    with the variable and determine whether
    the value is valid.
3b. For each of the variables related by
    constraints to the variable under
    consideration, filter all values that
    are not consistent with the constraints.
3c. For each of these associated variables,
    determine whether there exists at least
    one consistent assignment.
3d. If not, then reject the value under
    consideration; otherwise, accept it.
```

Any system that propagates constraints runs the risk of introducing a large amount of unnecessary computation into the testing process. The application of a constraint to a variable serves to reduce the number possible of values, although, in general, there will still be more values remaining than will ever be needed. To avoid unnecessary enumeration of values, we have built a constraint satisfaction system that incorporates delayed evaluation techniques.

3 Delayed Evaluation

Delayed evaluation is a demand-driven evaluation scheme - parts of data structures are not generated until needed. There are many ways to implement delayed evaluation (see [Filman and Friedman, 1984]) but we have adapted *streams* [Abelson *et al*., 1985, Filman and Friedman, 1984].

Streams are an abstraction of lists. As a data abstraction, streams and lists are identical, but they differ in that streams delay the evaluation of their *cdr's* while lists evaluate all elements at once. The operations on streams *(head, tail, and cons-stream)* are similar to list operations *(car, cdr, and cons)* . *Cons-stream* is used to recursively define streams. It only partially constructs a a stream, however, and then passes the partial construction to the program that accesses the stream. If the part of the stream that is not yet constructed is accessed by the *tail* operation, the stream will automatically construct only that part of itself that the program requires. This permits representations of infinite data. In addition to the delay of the original construction of the stream, the filtering of a stream can be delayed. Both aspects of streams are crucial to our system in that they allow us to represent infinite data structures and they reduce unnecessary application of filters.

4 System Architecture

Our system is implemented in CommonLISP using the CommonLISP Object Standard (CLOS) prototyped by Xerox PARC [Bobrow *et al*., 1987]. The basic elements of our system are a collection of interrelated objects having a built-in hierarchy of attribute (slot) values. The constraint network consists of variable objects, generator objects (the domain of possible values on a variable), and constraint objects. We have defined our own constraint language, allowing the user to add new constraints of any arity (number of related variables).

Our system assigns values to variables and applies each variable's constraints as they are assigned. These constraints not only test whether a commitment is legal, but also propagate information from assigned (fixed) variables to unassigned (constrained) variables through filter application. The filters that are created during the application of a constraint can either *disallow* certain values on a variable's domain or *change* the variable's domain of values in some way.

When a commitment is made to a variable, forward checking is executed by posting filters on the related constrained variables and provisionally assigning a set of values to them. If the related variable has at least one acceptable value on its generator, then the next variable can be assigned. However, if the related variable's generator is exhausted, any added filters must be removed, and the generator must be returned to the state it was in before the attempted commitment. New values for the current variable are tried (with forward checking) until the variable runs out of possibilities (causing the system to backtrack), or until a legal value is found. Thus, in our system, constraint satisfaction is simply a process of assigning values, posting filters, removing filters, and reinstating generators until a solution is found, or until the first variable exhausts its possible assignments (indicating that there is no solution).

Because of the local/global consistency relationship in constraint networks, our system permits users to work on partially solved problems, to change previously made choices, and to add new information as it becomes available.

We have applied these ideas to scheduling. The section below describes the implementation of the scheduling system.

5 Scheduling

Our implementation views a resource allocation and scheduling problem as a set of task objects, each containing information (slots) about an event's start time, end time, duration, and resource requirements. Resource requirements of a task include information about what type (class) and how much of some resource is needed. In addition to task objects, the system defines resource objects that contain information about the availability of resources.

5.1 Temporal Variables

In our formulation, a task's start time, end time, and duration are represented as distinct variables. The domain of each variable is represented by a set of intervals; only those intervals contained in the set are legal possibilities. We also include special values :pos-inf and :neg-inf that allow us to represent open intervals when the temporal variables are unconstrained.

We represent intervals as dotted pairs whose first value is the start of the interval and whose second value is its end. For example,

```
((0 . 10) (100 . :pos-inf))
```

is interpreted as $\{t \mid 0 \leq t \leq 10 \vee 100 \leq t \leq \infty\}$.

5.2 Temporal Constraints

In general, temporal constraints create filters that change the values of the temporal intervals on a time variable's generator. The rules of a constraint are defined using interval algebra and James Allen's formalism of the relations between two intervals [Allen, 1984],

PLUS-INTERVAL is a 3-place constraint that adds two intervals. In the scheduling application, it maintains the following functional relationship among the start time, end time and duration variables of a task:

```
start time + duration = end time
```

The propagation rules for this constraint are:

```
start time <- end time - duration

end time <- start time + duration

duration <- end time - start time
```

A filter is applied to the variable on the left when it is constrained and the variables to the right of the arrow are fixed.

EQUAL is a binary constraint that relates two time variables and creates a filter that ensures that the generators of the two variables contain the same values.

AFTER is a binary constraint relating start time or end time variables, ensuring that some time interval follows another.

We have also defined a special instance of a constraint, NOT-DURING. NOT-DURING encodes information disallowing certain time intervals as possible generator values. For instance, if it is known that tasks can only be scheduled on weekdays, we can define a NOT-DURING-WEEKENDS constraint that filters any time intervals falling on weekends. Thus, if we assume that some particular start and end time variables have initial values, in days, of

```
(1 . :pos-inf)
```

representing an interval from relative day 1, (Monday) with no fixed end, the application of the NOT-DURING-WEEKENDS filter would create a new infinite possibility set of intervals

```
((1 . 5) (8 . 13) .....).
```

Constraints like these can significantly increase the number of possible values of a variable. It is in this particular case, and cases like it, that the power of the stream representation is realized. If the possible time intervals were represented as a list, the computation (application of filters to list elements) would continue endlessly. *

5.3 Resource Variables

A task's resource requirements and the available resources are represented by resource class and resource pool variables. A resource pool is an instance of a resource class. For example, if there exists a resource class, *technicians,* we can think of each resource pool as an equivalence class of individual *technicians.* We may want to distinguish between two pools of technicians for reasons such as different costs associated with each pool, or different locations in which the pools reside. For our purposes, if a resource pool contains more than one of some resource, each resource is interchangeable within a single pool, but not across multiple pools. Thus, different resource pools represent distinguishable resources.

Any variables whose values change over time are represented as histories. Histories are lists of changes to a variable's value indexed by time. The most important time-based variables in a scheduling system are those that track information about resource availability. In our system, resource availability histories are represented as lists (or streams) of time interval/quantity available pairs. For instance,

```
( ((0 . 10) . 75)
  ((11 . 50) . 25)
  ((51 . :pos-inf) . 75)
)
```

is interpreted as:

$$Availability(resource, t) = \begin{cases} 75 & 0 \leq t \leq 10 \\ 25 & 11 \leq t \leq 50 \\ 75 & 51 \leq t \leq \infty \end{cases}$$

5.4 Resource Constraints

Resource variables are used to evaluate the truth of the RESOURCE-AVAILABLE constraint and to propagate its effects. The RESOURCE-AVAILABLE constraint is a 4-place constraint relating a task's start time, end time, quantity-of-resource-needed, and resource pool. It is used to determine if a particular amount of some resource is available during the given interval of time. If the resource pool and required amount of the resource is known, the constraint restricts (filters) the values of the start and end times to intervals that begin when the required quantity of that pool is available. If the time and quantity variables are fixed, then those resource pools that do not meet the requirements are filtered from the resource pool variable.

The EQUAL and NOT-DURING constraints can be used in much the same way as they are used in temporal reasoning. Instead of filtering possible values of time intervals, the possible resource pools are filtered.

however, infinite streams must be handled carefully as they do not *avoid* infinite computation, but rather delay it. In practice, this means that filter application must be ordered so that more constraining filters are applied first.

5.5 Scheduling

The scheduling process begins by placing all tasks on a priority queue of unscheduled tasks. This priority queue generally prefers tasks that are closer to the anchors of the schedule (the tasks fixed in time due to external forces). As tasks are removed from the queue, a variable commitment strategy is chosen that guides the search for appropriate times and resource pools for the task.[2] An example of a commitment strategy is to schedule a task's variables in the following order: *end-time, start-time, resource-pooll*, *resource-pool2* ...[3]

When all of the variables (i.e., start time, end time, quantity-of-resource-needed, and resource pool) for a resource allocation are committed, the effects of this usage are recorded in the resource pool. When a task has been completely scheduled, it is placed on a list of scheduled tasks. We use the systematic backtracking procedure discussed above to instantiate the variables of a task (using the commitment strategy as the variable ordering). When a task can not be scheduled (i.e., there is no instantiation of its variables consistent with constraints) the procedure removes the previous task from the list of scheduled tasks and tries new values for its variables (in backwards chronological order).

5.6 Scheduling Example

To illustrate the use of streams for a scheduling application, we present a simple example. Consider two tasks, *receiving* and *installation,* whose start and end times are fixed to the same value (i.e., the tasks are performed in parallel) and each of which need one technician. The resource pool utilization variable of *receiving* is fixed to:

```
(Sid)
```

Installation's resource pool utilization is constrained to:

```
(Pete, Roger, Keith, Sid, Albert, Eric, Mick)
```

The system evaluates the constraints on the *installation's* usage that propagates information from the *receiving* usage. When lists arc used to represent the pools of technicians, this propagation manifests itself as a filter on the generator of *installation,* resulting in the following new list (reflecting that Roger and Albert were being used by some other task at this time):

```
(Pete, Keith, Eric, Mick)
```

In this case, the entire list was checked. Compare this to the stream representation. The new stream (after filtering) is:

```
(Pete . <delayed closure>)
```

Since Pete meets the constraints imposed upon the *installation* task, he remains on this list and the filtering of the rest of the list is delayed. If Pete were judged to be unacceptable, he would be rejected and the next available technician would be chosen.

[2]In general, the duration of tasks as well as the type and quantity of resources they require are known *a priori.* However, if these variables were not known, they would be added to the commitment strategy (and constrained) like the other variables.

[3]This is the commitment strategy chosen for the results reported in the next section.

```
(Keith . <delayed closure>)
```

Only at this point is Roger filtered and Keith chosen, avoiding the unneccessary processing employed by a traditional constraint propagator.

6 Results

We have formulated and solved two problems as CSPs using our system: the n-queens problem (i.e., place n queens on an n x n chess board so that no two queens share the same row, column, or diagonal) and a real world scheduling problem based on NASA's Space Shuttle payload processing domain (EMPRESS-A) [Hankins *et* a/., 1985] .

Payloads that fly on the shuttle rest upon modular containers called carriers. Kennedy Space Center personnel have generated a partially ordered hierarchy of tasks necessary to process the payloads and carriers before and after a shuttle flight. In addition, resource types and quantities required to accomplish these tasks have been determined. The anchor of a schedule in this domain is the launch date of the shuttle. The task queue is ordered so that post-launch tasks are scheduled first in chronological order (with respect to the partial order) and pre-launch tasks are scheduled backwards from launch in reverse chronological order.

6.1 Efficiency Gain

The most significant advantage of delayed evaluation is its efficiency. If one were to propagate constraints in a straightforward manner, without delayed evaluation, many more filter checks would be performed. We have demonstrated this efficiency gain by applying our system to the n-queens problem and to the EMPRESS-A scheduling problem with and without using delayed evaluation. To test without delayed evaluation, we simply redefined the head, tail and cons-stream operations so that they were car, cdr and cons, respectively. Then we compiled statistics on the total number of consistency checks made for finding the first acceptable solution. A consistency check occurs each time the program verifies that a variable's value is consistent with its constraints. In the n-queens problem, as the size of the problem increases, so does the efficiency gain (see Figure 2). The improvement was on the order of 20 percent. [4] Figures 3 and 4 describe the problem size and illustrate the benefit of using streams in the EMPRESS-A application. In general, representing possibilities as streams is most useful when the average number of candidate values for a variable is large. In EMPRESS-A, this is realized when resource pool variables have many possible pools and temporal variables have many possible interval values.

6.2 Representational Advantages

Our system is highly extensible, can be used to represent infinite data structures and can be used to find adequate or optimal solutions to a problem.

[4]The fluctuations in the results are due to the fact that some problems had initial streams that were serendipitously close to solutions.

# of Queens	Consistency Checks		% Improvement
	Streams	Lists	
8	1336	1687	20.81
9	805	996	19.18
10	1841	2254	18.32
11	11280	14423	21.80
12	4487	5873	23.60
13	75867	96863	21.68
14	35880	47357	24.24
15	101872	139483	26.96

Figure 2: N-Queens Efficiency Gain.

EMPRESS-A Missions	# of Tasks	# of Variables	# of Constraints
Flight-56	33	428	488
Flight-57	83	411	1240
Flight-58	64	598	1305
Flight-62	66	486	1301
Total	246	1923	4334

Figure 3: EMPRESS-A Problem Size

EMPRESS-A Missions	Consistency Checks		% Improvement
	Streams	Lists	
Flight-56	1844	2746	32.85
Flight-57	2696	4010	32.77
Flight-58	3962	6566	39.66
Flight-62	3568	5883	39.35
Total	12070	19205	37.15

Figure 4: EMPRESS-A Efficiency Gain

Problems are specified in our system by defining constraints and attaching instances of the constraints to variables. Because the filtering mechanisms coordinate constraint behavior during propagation, the definition of each individual constraint is made independently of others. When a vocabulary of constraints has been declared in a particular problem domain, users can extend and modify problems without programming, by using the constraint language to add and delete constraints. Furthermore, the system can dynamically adapt to changes in a problem domain without modifying its search algorithm by adding and retracting constraints. These characteristics contribute to the modularity and extensibility of the system.

We have discussed the advantages of having the ability to represent infinite lists as streams. The transparency between list and stream operations also extends the representational capabilities of our system by providing the ability to find optimal solutions to problems.

Although most problem spaces are so large that optimization is unrealistic, our system can find optimal solutions based on some metric. To optimize an assignment, all of a variable's values must be considered and compared. In this case, delayed evaluation offers no savings. Many domains, however, require some variables to be optimally assigned while others can be assigned any acceptable value. We allow both options by defining different kinds of generators, stream-generators and list-generators. The stream generators are used for satisfying and operate through the application of filters (using delayed evaluation). The list generators optimize and operate through the application of filters and comparators - predicates that determine if one value is more desirable than another. Without any significant change in the system's operations or representations, we are able to integrate the benefits of delayed evaluation into a CSP when satisficing.

7 Previous Work

Our implementation extends previous work in constraint satisfaction, planning and scheduling. The seminal work of Vere [Vere, 1983] represents actions by maintaining constraints between the start, finish, and duration of tasks. In our system, we can maintain arbitrary constraints between variables (e.g., resource constraints). We extend the constraint management mechanisms of non-linear planners like NonLin [Tate, 1977] and SIPE [Wilkins, 1984] by offering a uniform mechanism for constraint propagation over all variables. Like Dean's TMM [Dean, 1985], our system maintains information over time. However, the representation we use takes the resource perspective, that is, variables range over values that are not necessarily Boolean.

8 Conclusions and Future Work

We have implemented a system using a formulation of constraint satisfaction problems that is modular and takes advantage of delayed evaluation techniques.

The performance of the system improves significantly when it is applied to both EMPRESS-A scheduling and to n-queens with streams instead of lists representing the variables' domain of possible values.

Future goals of this project are to use explanation-based learning techniques to learn heuristics that recognize resource bottlenecks and adjust the search process accordingly, to build an efficient scheduling system, and to test the system against other NASA domains. This will provide a better analysis of the system since every scheduling problem has different constraints, different unknowns, different required optimality, and different levels of resource availability.

Acknowledgements

We thank Richard H. Brown for his invaluable ideas concerning constraint satisfaction that have made this work possible. We also thank Melissa P. Chase for introducing us to delayed evaluation, Will Taylor for designing the user interface, James Allen, Walter Hamscher, Paul Morris, and Amy Lansky for their comments and suggestions, and Mark Drummond, Phil Laird, and Peter E. Friedland for careful review of this paper.

References

[Abelson *et al.,* 1985] Abelson, H., Sussman, G.J., Sussman, J. *Structure and Interpretation of Computer Programs.* MIT Press, Cambridge, MA, 1985.

[Allen, 1984] Allen, James F. Toward a General Theory of Action and Time. *Artificial Intelligence,* 23, 1984.

[Bobrow *et al,* 1987] Bobrow, D.G., DeMichiel, L.G., Gabriel, R.P., Keene, S., Kiczales, G., Moon, D. *Common LISP Object System Specification.* Unpublished Draft, 1987.

[Brown, 1987] B rown, Richard H. Knowledge-based Scheduling and Resource Allocation in the CAMPS Architecture. In *Proceedings from the IEEE International Conference on Expert Systems and the Leading Edge in Planning and Control,* Benjamin/Cummings, 1987.

[Dean, 1985] Dean, Thomas. *Temporal Imagery: An Approach to Reasoning about Time for Planning and Problem Solving.* PhD thesis, Yale University, October 1985.

[Dechterand Pearl, 1987] Dechtcr, R., Pearl, J. Network based Heuristics for Constraint Satisfaction Problems. *Artificial Intelligence,* 34(1), December 1987.

[Filman and Friedman, 1984] Filman, R.E., Friedman, D.P. *Coordinated Computing: Tools and Techniques for Distributed Software.* McGraw-Hill Book Company, New York, NY, 1984.

[Fox, 1983] Fox, Mark S. *Constraint-Directed Search: A Case Study of Job Shop Scheduling.* PhD thesis, Carnegie-Mellon University, 1983.

[Gusgen *et a*/., 1988] Gusgen, Hans-Werner, Hertzberg, Joachim. Some Fundamental Properties of Local Constraint Propagation. *Artificial Intelligence,* 36, 1988.

[Hankins *et al.,* 1985] Hankins, G.B., Jordan, J.W., Katz, J.L., Mulvehill, A.M., Dumoulin, J.N., Ragusa, J. EMPRESS: Expert Mission Planning and Re-planning Scheduling System. In *Expert Systems in Government Symposium,* 1985.

[Haralick and Elliot, 1980] Haralick, R.M., Elliot, G.L. Increasing Tree Efficiency for Constraint Satisfaction Problems. *Artificial Intelligence,* 14, 1980.

[Johnston, 1989] Johnston, Mark D.. *Reasoning with Scheduling Constraints and Preferences.* Technical Report, Space Telescope Science Institute, 1989.

[Lansky, 1988] Lansky, Amy. *Localized Event-based Reasoning for Multiagent Domains.* Research Note, SRI International, 1988.

[Mack worth, 1977] Mack worth, Alan. Consistency in Network Relations. *Artificial Intelligence,* 8, 1977.

[Rit, 1986] Rit, Jean-Francois. Propagating temporal constraints for scheduling. In *AAAI-86 Proceedings,* 1986.

[Stallman and Sussman, 1977] Stallman, R.M., DUS8-man, G.J. Forward Reasoning and Dependency-Directed Backtracking in a System for Computer-Aided Circuit Analysis. *Artificial Intelligence,* 9, 1977.

[Steele, 1980] Steele, Guy. *The Definition and Implementation of a Computer Programming Language Based on Constraints.* PhD thesis, Massachusetts Institute of Technology, 1980.

[Stefik, 1980] Stefik, Mark. *Planning With Constraints.* PhD thesis, Stanford University, January 1980.

[Tate, 1977] Tate, Austin. Generating Project Networks. In *IJCA I-77 Proceedings,* 1977.

[Vere, 1983] Vere, S. A. Planning in Time: Windows and Durations for Activities and Goals. *IEEE Transactions,* PAMI-5(3), May 1983.

[Waltz, 1975] Waltz, David. Understanding Line Drawings of Scenes with Shadows. In P. Winston, editor, *The Psychology of Computer Vision,* McGraw-Hill, 1975.

[Wilkins, 1984] Wilkins, D.E. Domain Independent Planning: Representation and Plan Generation. *Artificial hitelligence,* 22, 1984.

[Zweben, 1987] Zweben, Monte. *CAMPS: A Dynamic Re-planning System.* Technical Report, MITRE Corporation, 1987.