Branching Constraint Satisfaction Problems for Solutions Robust under Likely Changes

David W. Fowler and Kenneth N. Brown

University of Aberdeen, Aberdeen AB24 3UE, UK
{dfowler,kbrown}@csd.abdn.ac.uk

Abstract. Many applications of CSPs require partial solutions to be found before all the information about the problem is available. We examine the case where the future is partially known, and where it is important to make decisions in the present that will be robust in the light of future events. We introduce the branching CSP to model these situations, incorporating some elements of decision theory, and describe an algorithm for its solution that combines forward checking with branch and bound search. We also examine a simple thresholding method which can be used in conjunction with the forward checking algorithm, and we show the trade-off between time and solution quality.

1 Introduction

In this paper, we consider the problem of a solver that periodically receives additions to an existing problem, and must make a decision for each addition as it is received. We assume that there is a simple model of what additions are likely to occur. We believe that this knowledge can enable the solver to make decisions that will be robust under future events. The approach used here involves extending the framework of constraint satisfaction to include the model of future events to give a new form of CSP, called a *branching CSP* (BCSP). A detailed presentation can be found in [FB00].

As an illustration, consider a scheduling problem in which new tasks arrive during the process. The aim is to schedule as many tasks before their due date as possible. There are two identical resources that the tasks may use. The tasks for this problem are described in Fig. 1. We start with tasks A and B. We know that one of the other tasks will arrive at time 1. It will be C with probability 0.6, and D with probability 0.4. A final restriction (future work will look at how to overcome this) is that tasks A and B must be scheduled before it is known which of tasks C or D arrives next. How should we schedule tasks A and B?

There are three reasonable possibilities, shown in Fig. 2. Solution (a) allows C to be scheduled, but not D, whereas (b) allows D but not C. (c) allows both C and D to be scheduled, at the price of omitting B. Which is best depends on the probabilities and utilities. For this example, the expected utilities are (a) 3.8, (b) 5.2, and (c) 6.0. So it is best to schedule task A at time 0, omit B, and then schedule C at time 1 or D at time 2, depending on which arrives.

R. Dechter (Ed.): CP 2000, LNCS 1894, pp. 500–504, 2000.

Task	Duration	#Resources	Due	Utility
A	2	1	4	1
B	2	1	4	1
C	3	1	4	3
D	1	2	4	8

Fig. 1. Tasks for Example Problem

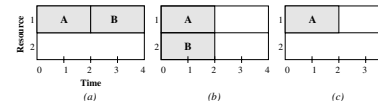

Fig. 2. Three Possible Solutions

2 Branching CSPs

The initial definition of a BCSP involves variables, constraints, and a state tree. The variables and constraints are as in standard CSPs, with the difference that each variable has an associated non-negative utility, which is gained if the variable has a value assigned to it. Variables can be left unassigned, in which case the utility gained from that variable is 0.

The state tree represents the possible development paths of the dynamic problem. Each edge in the tree is directed, and is labelled with a transition probability. Each node S_i has an associated variable X_{S_i}, with the restriction that a variable can appear at most once in any path from the root to a leaf node. There are transition probabilities p_{ij} labelling the edge (if it exists) between S_i and S_j. For any path through the tree from root to leaf node, a series of constraint satisfaction problems is produced, involving all variables that have been encountered at each node in the path so far. If a constraint involves variables that are all assigned values, then those values must satisfy the constraint.

A solution to a BCSP is a decision for each variable at each node in the state tree, so that on each path all relevant constraints (those that involve only variables that have been assigned values in the path) are satisfied. A solution is a plan for each possible sequence of variable additions, and we have assumed that the total utility of the solution can be found by summing the utilities of the assigned variables in the path that actually occurs. However, we must try to find a solution before we know which path will occur, and so we define the optimal solution to be the one with the highest expected total utility. For a solution, the expected utility from a node can be defined recursively as follows. The E.U. from a leaf node S_i is the utility of X_{S_i} if it is assigned a value, otherwise 0. For a nonleaf node S_i, the E.U. is the utility of X_{S_i} (or zero if it is unassigned) plus $\sum_j p_{ij} EU_j$, where the sum is over all the child nodes of S_i.

3 Solution Algorithms

Two complete algorithms have been implemented to solve BCSPs involving binary constraints on finite domains. These are: a straightforward branch and bound algorithm that examines each node in the state tree in a depth first order, and finds the value for the variable that maximises the expected utility from that node; and a forward checking algorithm that uses the constraints to prune the domains of variables lower down in the tree. As well as reducing the

number of values that need to be examined, the propagation is used to calculate an upper bound on the expected utility from the current node. If this value is less than that of the best solution found so far, the effects of propagation can be undone, and the next value tried for the current variable immediately. Experiments show that FC is much faster than the basic branch and bound algorithm, so we concentrate on FC for the rest of this paper.

To test FC, we generated random problems as follows. The number of variables, n, was fixed at 10. Each variable had a domain with $m = 10$ values, and a utility which was an integer selected at random uniformly from the range [1,50]. The state tree was produced by the following branching process. For each node the probability of no children was 0.05, for one child 0.5, for 2 children 0.25, and for 3 children 0.2. The transition probabilities for each child node were then selected so that they summed to 1.0. We varied the density of the constraint graph p_1 and the tightness of constraints p_2. p_1 was varied from 0.1 to 1.0 in steps of 0.1, and p_2 from 0 to 1 in steps of 0.02. 100 problems were generated for each combination of p_1 and p_2, and the median number of constraint checks recorded. The results are shown in Fig. 3.

It is interesting to compare the hardness of BCSPs with that of static CSPs, where all variables must be assigned values (if this is possible). In Fig. 3 we have shown the curve for $p_1 = 0.6$ (other values of p_1 show similar behaviour). The static CSP also has $n = 10$ and $m = 10$.

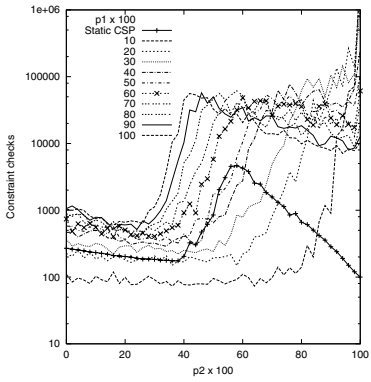

Fig. 3. FC Search

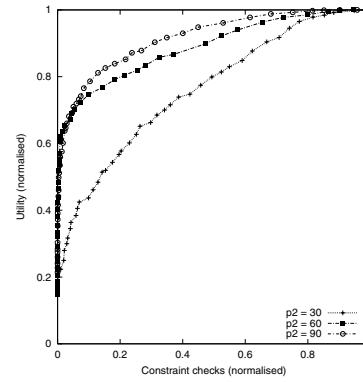

Fig. 4. Thresholding using FC

4 Thresholding

The algorithms presented above are complete, but may not have explored important branches by the time the first decision is required. In real world applications we usually prefer a slightly inferior result before a deadline has been reached, than an optimal result afterwards. It is also useful to be able to generate results that improve over time, instead of a single result at the end of the computation.

Thresholding is a simple method that can be used to implement either of the above. The idea is to ignore branches of the state tree that can not give a utility higher than a threshold value. For high thresholds, large sections of the tree will be pruned, giving a problem that can be solved much more quickly. For very low values the tree will be pruned only slightly, and for a threshold of 0 there will be no pruning at all.

The algorithm was tested on random problems generated as before. For each problem, the initial threshold was taken as an upper bound on the total expected utility, calculated by assuming that all variables could be assigned values. The threshold was reduced to zero in 50 evenly spaced steps, with the problem being solved with FC for each threshold. The last step (with the threshold equal to zero) corresponds to the full original problem. p_1 was fixed at 0.7, and three values of p_2 chosen to give underconstrained ($p_2 = 0.3$), hard ($p_2 = 0.6$), and overconstrained ($p_2 = 0.9$) problems. 100 problems were generated in each run. Fig. 4 shows how the expected utility increases with the number of constraint checks. For the hard problems, almost 80% of the achievable utility can be gained with 20% of the constraint checks needed to find the optimal solution. The overconstrained problems give similar results; for underconstrained problems the performance is poorer, but the time for solving these is not significant.

5 Related Work

Dynamic Constraint Satisfaction [DD88] models a changing environment as a series of CSPs. The emphasis in DCSPs has been on minimising the work needed to repair a solution when a change occurs. There is typically no model of the future, and thus no concept of solutions which are themselves robust to changes. Wallace and Freuder [WF97] do consider future events in their *recurrent* CSPs, and aim to find robust solutions, but they concentrate on changes which are temporary and frequently recurring. *Supermodels* [GPR98] are solutions which can be repaired in a limited number of moves, given a limited change to the original problem. This approach does not consider the likelihood of changes, nor does it take account of a sequence of changes. Fargier et al. [FLS96] propose *mixed* CSPs, in which possible future changes are modelled by uncontrollable variables. They search for conditional solutions, dependent on the eventual value of these variables, and thus the solutions are robust. However, they do not deal with sequences of events, but assume all changes occur at the same time. As a result, there is not necessarily any similarity between the different individual solutions derived from the conditional ones. Finally, it must be mentioned that the model of likely future events will occasionally be insufficient, and an unexpected event will occur. A practical solver will have to be able to fall back to existing DCSP methods in this case.

In addition to modelling changes, we also develop partial solutions to overconstrained problems. We choose to leave variables unassigned, and insist on all constraints being satisfied. Most work on partial CSPs, for example [FW92, BMR95], concentrates on finding solutions which violate the fewest constraints. Freuder

and Wallace's [FW92] general scheme for solving PCSPs searches for variations on the problem which would allow complete solutions. It is possible to recast the unassigned variable approach in that scheme by creating new problems which retract exactly those constraints which involve the variables we leave unassigned. Our algorithms could be considered to be doing exactly that; however, we believe explicitly designating variables as being unassigned is a more natural representation for many applications.

6 Future Work

For the example problem of section 1, it can be seen that it would be better to schedule A, and delay a decision on B until we see whether C or D arrives. The expected utility is then 7.0. We have implemented an algorithm which allows such delays, but have yet to produce experimental results. At present, we can only postpone a variable until the next variable arrives; future work may consider how to relax this restriction. We intend to continue developing our understanding of the current model by more experimentation with the existing algorithms, developing the algorithms to include more propagation during search, and finding better anytime algorithms. We aim to extend the model to include explicit times for events, and to allow constraint violations as well as unassigned variables. We have started to compare our algorithms with CSPs that use 0/1 variables to signify that variables are unassigned. Early results indicate that our methods allow for both easier formulation of problems and more efficient solution. Finally we will compare our algorithms with existing methods of scheduling under uncertainty - for example, MDPs and Just-In-Case scheduling [DBS94].

References

[BMR95] S. Bistarelli, U. Montanari, and F. Rossi. Constraint solving over semirings. In *Proceedings of IJCAI-95*, pages 624–630, 1995. 503

[DBS94] M. Drummond, J. Bresina, and K. Swanson. Just-in-case scheduling. In *Proceedings of AAAI-94*, Seattle, Washington, USA, 1994. 504

[DD88] R. Dechter and A. Dechter. Belief maintenance in dynamic constraint networks. In *Proceedings of AAAI-88*, pages 37–43, 1988. 503

[FB00] D. W. Fowler and K. N. Brown. Branching constraint satisfaction problems. Technical report, Dept. of Computing Science, Univ. of Aberdeen, 2000. 500

[FLS96] H. Fargier, J. Lang, and T. Schiex. Mixed constraint satisfaction: a framework for decision problems under incomplete knowledge. In *Proceedings of AAAI-96*, Portland, OR, 1996. 503

[FW92] E. C. Freuder and R. J. Wallace. Partial constraint satisfaction. *Artificial Intelligence*, 58:21–70, 1992. 503, 504

[GPR98] M. L. Ginsberg, A. J. Parkes, and A. Roy. Supermodels and robustness. In *AAAI-98*, pages 334–339, 1998. 503

[WF97] R. J. Wallace and E. C. Freuder. Stable solutions for dynamic constraint satisfaction problems. In *Workshop on The Theory and Practice of Dynamic Constraint Satisfaction*, Salzburg, Austria, November 1997. 503