Annals of Operations Research 118, 85–100, 2003


# Branching Constraint Satisfaction Problems and Markov Decision Problems Compared

DAVID W. FOWLER and KENNETH N. BROWN
*Department of Computing Science, University of Aberdeen, Aberdeen, UK*

**Abstract.** Branching Constraint Satisfaction Problems (BCSPs) model a class of uncertain dynamic resource allocation problems. We describe the features of BCSPs, and show that the associated decision problem is NP-complete. Markov Decision Problems could be used in place of BCSPs, but we show analytically and empirically that, for the class of problems in question, the BCSP algorithms are more efficient than the related MDP algorithms.

**Keywords:** constraint satisfaction, uncertainty, Markov decision problems

## 1. Introduction

We consider a class of resource allocation problems in which a sequence of tasks is presented to a solver, which must find an assignment for each task when it arrives. There are constraints restricting the assignments that sets of tasks may be given. All possible tasks and their associated constraints are known in advance, but only a subset will arrive in the problem. The solver's goal is to specify an assignment for each task that arrives, such that no constraints are violated, while optimising an objective function. The solver may choose to reject some requests. The problem class can be used to model a number of different scenarios – for example: allocating incoming planes to gates, assigning deliveries to individual couriers, or deciding upon start times for jobs in a dynamic jobshop. We assume there is uncertain knowledge of the subset of the tasks and the order in which they arrive, given as a probabilistic branching tree of arrivals. Given this knowledge, the solver's goal is to pre-compute the optimal assignments, to enable an instant decision when each task arrives. We have modelled such problems as Branching Constraint Satisfaction Problems [8,9], in which we model each task as a variable, and we have presented solution algorithms based on extensions of constraint-based tree-search. Those algorithms have a worst-case time complexity exponential in the branching tree depth. Since our trees obey the Markov assumption, an obvious question is whether such problems can be modelled as Markov Decision Processes, and solved using the existing polynomial MDP solution methods. In this paper, we compare the two models. We show that the BCSP is NP-complete. However, the constraints complicate the translation to MDPs, and cause the MDPs to be, in the worst case, of size exponential in the branching tree depth. We then show empirically that the BCSP methods are more efficient. The two models are obviously related, and BCSPs could be considered to be an efficient method

of generating and solving MDPs for this class of problem. In the next section, we review the background to this research. We then discuss the general problem in more detail, and briefly review the BCSP solution methods. We then show how the model can be translated into an MDP representation, and compare that to the BCSP methods. Finally, we present some experiments comparing the two different solution methods.

## 2. Background

*Dynamic* Constraint Satisfaction [6] models a changing environment as a series of constraint satisfaction problems. The emphasis in DCSPs has been on minimising the work needed to repair a solution when a change occurs [19]. There is typically no model of the future, and no concept of solutions which are themselves robust to changes. Wallace and Freuder [20] consider future events in their *recurrent* CSPs, and aim to find robust solutions, but they concentrate on changes which are temporary and frequently recurring. Ginsberg et al. [10] propose *supermodels* in satisfiability problems, which are solutions which can be repaired in a limited number of moves, given a limited change to the original problem. Their approach considers neither the likelihood of changes, nor sequences of changes. Fargier et al. [7] propose *mixed* CSPs, in which possible future changes are modelled by uncontrollable variables. They search for conditional solutions, dependent on the eventual value of those uncontrolled 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, and so there is not necessarily any similarity between the different individual solutions derived from the conditional ones. Littman et al. [15] discuss *stochastic Boolean satisfiability*, in which multiple subsets of the variables are uncontrollable.

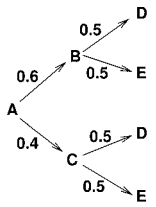
*Markov Decision Processes* [1,13] assume that the world can be described by a set of possible *states*. For each state, there is a set of non-deterministic *actions* an agent can take to move the world into a new state. There is a state-transition function which, for each state/action pair $(S_i, A_k)$, specifies the probability that the next state will be $S_j$. Each action in a state has an associated cost or reward. The *Markov property* requires that the probabilities depend only upon the current state and the selected action, and that the history of how a state was reached is irrelevant. The aim of a *Markov Decision Problem* is to determine a policy – an action specification for each state – that satisfies some criterion. Typically, the aim is to maximise the discounted expected utility, where actions that take place $n$ steps ahead have their costs or rewards reduced by a discount factor $f^n$. State transition diagrams may be either cyclic or acyclic, and the problem may have either a finite or infinite horizon (the length of the chain of actions to be considered). MDPs can be solved by Linear Programming, in time polynomial in the number of states, the number of actions, and the maximum space required to represent the probability and cost functions. Specific MDP algorithms include *value iteration* [1] and *policy iteration* [13]; for fixed discount factors, value and policy iteration are also polynomial-time algorithms. Puterman [18] gives a comprehensive survey of MDPs, Littman et al. [16] give a detailed analysis of the computational complexity of the solution methods for infinite horizon problems, and Boutilier et al. [2] survey their use in decision theoretic

planning. For acyclic MDPs, Boyan and Moore [4] state that the DAG-SP algorithm [5] is more efficient than either of the two standard algorithms. DAG-SP finds any topological sort of the states, so that each state precedes all the states reachable from it. It then traverses the resulting linear order in reverse, visiting each state once only, computing the value of each state, and hence the optimal policy for each state. Finally, Boutilier et al. [3] investigate methods of avoiding the enumeration of the full state space by reasoning about the values of atomic propositions that make up a particular state.

## 3. Branching constraint satisfaction problems

We assume that every variable that arrives will come from a known set of variables $X_i$. Each $X_i$ has an associated domain, $D_i$, of possible values. We have a set of constraints (relations over the variable domains) which restrict the possible values the variables may take, when those variables are active in the problem instance. The intention is that each variable represents a separate task, the values represent the possible allocations of resources to a task, and the constraints specify the legal combinations of allocations. We have a transition tree, which represents the possible sequences of events. Each node in the tree has an associated variable, representing the addition of that variable to the problem. The root of the tree represents the arrival of the first variable (or could be extended to be the initial empty problem). Each node will have a number of children, representing the possible subsequent additions. Each edge from parent $S_i$ to child $S_j$ has an associated probability, representing the probability that the child node is the next in the sequence. A complete path from the root node to a leaf represents one possible arrival sequence. A variable may be associated with multiple nodes, but can only appear once in any path. The tree obeys the Markov property – for any given node $S_i$, the probability that the next node is $S_j$ depends only on $S_i$, and not on the path from the root to $S_i$.

The goal for the solver will be to find a policy for the transition tree, determining which value to assign to the variables that arrive. Given a policy, we can make an instant decision when a variable arrives. The aim will be to make robust decisions for each variable that arrives, maximising our chances of finding good assignments for subsequent variables. Given this general framework, there are a number of variations that can be derived. The solver may fail as soon as it is unable to handle a request, or it might be allowed to reject some tasks, and continue to receive more requests. We model this by including a special *null* value in any domain corresponding to a task that can be rejected. The null value never violates a constraint. Secondly, the solver may gain a task-dependent reward for satisfying a request, or the reward may depend on the way the task is satisfied, or all tasks and assignments may be equally important. We can model all three by utility functions on the value assignments. In all cases, we forbid the violation of a constraint – if there is a constraint on a subset of the variables, they all appear in a path, and they are assigned non-null values, then the constraint must be satisfied. We consider three examples below.




| | | A | B | C | D | E |
|---|---|---|---|---|---|---|
| | X | ✓ | ✓ | ✓ | ✓ | ✓ |
| Staff Skills | Y | ✓ | × | × | × | ✓ |
| | Z | ✓ | × | ✓ | × | × |
| Utilities: | | 3 | 6 | 10 | 6 | 6 |

Figure 1. A simple problem that can be represented as a BCSP.

(i) A service company aims to maximise the number of requests it can handle before it is forced to reject a task. All requests and all value assignments are equally important. We associate a utility of 1 with each value assignment, and remove null from all domains.

(ii) A courier aims to maximise the utility of items he can deliver on a particular journey, and minimise the utility of the items left behind for later trips. Each item is represented by a variable, and its location in the hold is represented by the value assignment; items left behind are assigned null. The items arrive at time intervals, and must be loaded or rejected on arrival. We associate a utility with each variable, and gain the utility if we assign a non-null value to the variable.

(iii) Maintenance tasks bring rewards, but incur costs dependent on who is assigned to the task. Tasks can be rejected, receiving no reward and incurring no cost. The aim is to maximise the total reward. We associate a utility with each value in a domain, equal to the difference between the reward and cost. Null has a utility of 0.

We present a simple example of type (ii) in figure 1. A company receives service requests. There are three workers, x, y and z, and five possible tasks. Each worker is qualified to do some of the tasks, as shown in the table. No worker can do more than one task. We start with task A, and then either B or C will arrive, with the probabilities shown. Following that, D or E will arrive. A has utility 3, C has utility 10, and B, D and E have utility 6. The company may choose to reject some tasks, to ensure that a future task can be carried out. For this problem, an optimal solution assigns worker z to task A, x to B or C (depending on which arrives), and y to E. Task D is rejected if it arrives. This solution has an expected total utility of 13.6.

We now present the formal definition and establish the complexity class of type (ii) above. The symbol $\perp$ represents the null value.

**Definition 1.** A *Branching CSP* is a tuple $(X, D, U, C, S, \tau)$ where:

- $X = \{x_1, \ldots, x_n\}$, is a set of *variables*.
- $D : X \to \mathcal{P}(\{0, \ldots, m-1\})$ associates a *domain* to each variable.
- $U : X \to \mathbb{R}^+$ gives the *utility* of each variable.
- $C = \{c_1, \ldots, c_k\}$ is a set of *constraints*, where each $c_i$ is a pair $(X_i, R_i)$, $X_i$ being an ordered subset $\langle x_{i1}, \ldots, x_{ip} \rangle$ of $X$, and $R_i \subseteq D(x_{i1}) \times D(x_{i2}) \times \cdots \times D(x_{ip})$.

– $S = (V, E, P)$ is a probability transition tree of vertices $V$ and edges $E$; $v_1 \in V$ is the root, and if $v_i$ is the parent of $v_j$ then $i < j$; $P$ is a function $P : V \times V \to [0, 1]$ such that:
  - if $(v_i, v_j) \notin E$ then $P(v_i, v_j) = 0$;
  - $\forall v_i \in V$, if $\exists v_j \in V$ s.t. $(v_i, v_j) \in E$ then $\sum_{\forall i' : i' > i} P(v_i, v_{i'}) = 1$.
– $\tau$ is a function $\tau : V \to X$ assigning a variable to each node, such that for any path $(v_i, \ldots, v_j)$ in $(V, E)$, $\tau(v_i) = \tau(v_j)$ iff $i = j$.

**Definition 2.** An *assignment* to a BCSP $B$, is a function $\phi : V \to \mathbb{Z}^+ \cup \{\perp\}$, s.t. $\phi(v_i) \in D(\tau(v_i)) \cup \{\perp\}$.

**Definition 3.** A *solution* to a binary BCSP $B$, is an assignment $\psi$ such that if $(v_i, \ldots, v_j)$ is a path in $(V, E)$ and $(\{\tau(v_i), \tau(v_j)\}, R_{ij}) \in C$ then either $\psi(v_i) = \perp$ or $\psi(v_j) = \perp$ or $(\psi(v_i), \psi(v_j)) \in R_{ij}$.

**Definition 4.** The *expected total utility from a node* of a solution $\psi$ to a Branching CSP $B$, is $\omega_\psi : V \to \mathbb{R}^+$, where:

$$\omega_\psi(v_i) = \big(\psi(v_i) \neq \perp\big)U\big(\tau(v_i)\big) + \sum_{\substack{\forall j : (v_i, v_j) \in E \\ j > i}} P(v_i, v_j)\omega_\psi(v_j).$$

We define $(\psi(v_j) \neq \perp)$ to be 1 if $\psi(v_j) \neq \perp$, and 0 if $\psi(v_j) = \perp$. For leaf nodes the summation term above will be zero (the range of the index is empty) and the utility is just that of the leaf variable. For non-leaf nodes, the expected total utility is a weighted sum of the node's utility and the values for the child nodes.

**Definition 5.** The *expected total utility*, $\Omega_\psi$, *of a solution* $\psi$ to a Branching CSP $B$, is $\omega_\psi(v_1)$.

**Theorem 1** (Proof by induction – omitted).

$$\Omega_\psi = \sum_{\forall j : (v_1, v_j) \in Paths(E)} Pr(v_1, v_j)\big(\psi(v_j) \neq \perp\big)U\big(\tau(v_j)\big),$$

where $Paths(E)$ is the set of paths from root to leaf node in the tree, and $Pr$ gives the product of transition probabilities along the path.

The expected total utility of a solution is made up of a weighted sum of utilities from each node, where each utility is weighted by the probability of the path from the root to the node.

We now establish the complexity of BCSPs. First we express the problem as a decision problem:

**Branching Constraint Satisfaction (BCSP).**

*Instance*. Sets $X$, $D$, $U$, $C$, $S$, and $\theta$ (a threshold value).

*Question*. Is there an assignment of values to $S$, so that the expected total utility is at least $\theta$?

**Theorem 2.** BCSP $\in$ NP-complete.

*Proof.* To show BCSP $\in$ NP, the certificate is simply an assignment of values to $S$. This can be verified in time polynomial in the size of the input by checking the constraints, and then calculating the expected total utility from the bottom up, visiting each node once. The expected total utility can then be compared with $\theta$. We assume (following [12]) that each constraint check takes constant time, and so the time taken by each operation is polynomial in the size of the input representation. Assuming that there are $N$ nodes and $k$ constraints, and that probabilities can be represented by $t$ bits, we can find polynomial upper bounds on each of the individual steps, and therefore a polynomial bound on the whole verification.

Checking the constraints will require at most $Nk$ steps, as the variable in each node could appear in each of the $k$ constraints. Calculating the ETU will require at most $N-1$ multiplications, as each of the $N-1$ probabilistic edges is involved in one multiplication. The results of the $N-1$ multiplications must be added together, requiring $N-1$ steps. Comparison with $\theta$ will take 1 step. Each of these steps will depend on the number of bits required to represent the numbers involved, but the steps can be performed in time polynomial in the number of bits. For example, we assume that two $t$ bit numbers can be multiplied in time proportional to $t^2$ (which is good enough for our purposes, although there are theoretically more efficient multiplication algorithms). In conclusion, as each of the stages requires time polynomial in $N$, $t$ and $k$, the total time required will also be polynomial in $N$, $t$ and $k$.

To show that BCSP $\in$ NP-complete, we show how any 3-SAT instance can be reduced in polynomial time to a BCSP instance.

(i) Each 3-SAT variable is mapped to a BCSP variable, domain {0, 1}.

(ii) Each 3-SAT clause is mapped to a ternary constraint on the three corresponding BCSP variables. The tuples of the constraint are the subset of $\{0, 1\}^3$ which corresponds to the satisfying assignments of the clause.

(iii) A BCSP node is created for each BCSP variable. These nodes are connected in a linear tree with transition probabilities of 1.

(iv) The utility of each BCSP variable is set at 1.

If the original 3-SAT instance had $n$ variables and $k$ clauses, the corresponding BCSP instance has $n$ variables, $n$ nodes, and $k$ constraints. The question in the BCSP instance becomes: is there a solution with expected total utility $\geqslant n$? The BCSP instance has a solution if and only if all the nodes are assigned values that are not $\perp$. This will happen if and only if the 3-SAT instance has a solution.

Can this reduction be performed in time polynomial in the size of the representation of the 3-SAT instance? Any non-trivial 3-SAT instance will have $n \leqslant 3k$, as otherwise no two clauses will have a common variable. So, we can assume that the size of the 3-SAT representation is proportional to the number of clauses. To generate the BCSP instance, we examine each clause once, producing a new BCSP variable for each SAT variable we have not encountered so far (at most 3 per clause), and also generating a constraint. Generating each constraint involves finding which of the 8 members of $\{0, 1\}^3$ satisfy the clause. These operations clearly take time that has an upper bound that does not depend on the size of the 3-SAT instance, as do the operations involved in creating the BCSP tree. The number of operations is proportional to the size of the original instance, so the time taken by the whole process is polynomial. □

Our underlying solution method [8] is an all-solutions backtracking search in which we remember the best solution. We start by considering the first possible value for the variable in the root node of the transition tree, and then consider each probabilistic branch in turn. For a branch, consider the first possible value for the variable it leads to. If it violates a constraint with a value assigned higher up in a path, we backtrack, and consider another value; if it does not, we continue the search as before. Once we have considered all possible value assignments for a variable, we consider the next probabilistic branch, and repeat the process. Once we have considered all branches below a node where we have assigned a value, we can evaluate the objective function obtained by assigning that particular value. We continue until we have considered all choices, and we select the optimal policy. The pseudocode for this algorithm is given in figure 2, and we trace its execution on the sample problem in figure 3. The solid arrows represent alternative assignments for a variable, and the dashed lines represent probabilistic branches. Nodes where a constraint has been violated are shaded. The search is shown partially complete: the next node to be expanded is marked with a heavy arrow.

To estimate the running time of the algorithm, we assume that the average domain size is $m$, and that the branching tree has branching factor $b$ and a depth $d$, where $d$ is the number of variables in a path from the root to a leaf node. For each node, there are $m$ possible value assignments, and for each value assignment, there are $b$ successor nodes – i.e., at level $i$ of the search tree, there may be $(mb)^{i-1}$ nodes. Thus in total there may be $1 + mb + (mb)^2 + \cdots + (mb)^{d-1}$ nodes, which is $((mb)^d - 1)/(mb - 1)$ nodes, and thus the size of the search tree is exponential in the depth of the branching tree. At each node, for each value assignment we have to check the constraints on the assigned variables higher up in the current path. If we assume each variable is involved in $c$ constraints, then at level $i$ we have at most $mc$ constraint checks, where $c < i$. In addition, at each node, for each possible value, we have to multiply out the probabilities and utilities from the branches below, and sum them, and compare to the previous value assignment. That is $mb$ multiplications, $mb$ additions, and $m$ comparisons per node. Assuming constant work for each of these basic operations, we have a polynomial amount of work per node.

We then enhance this basic method by adding both branch-and-bound optimisation and forward checking. In branch-and-bound, the maximum achievable objective

```
BTAllSolns(node Si) returns <utility,solution>

1.  if dom(Xsi) = {} return <0,{}> //Xsi variable in Si
2.  bestEU := -inf
3.  for each v in dom(Xsi) do
4.    if (Xsi <- v) does not conflict with current path
5.      EU := utility(Xsi <- v)
6.      solution := {Xsi <- v}
7.      for each Sj in children(Si) do
8.        result := BTAllSolns(Sj)
9.        EU := EU + (P(Si -> Sj) * result.utility)
10.       solution := solution U result.solution
11.     if EU > bestEU then
12.       bestEU := EU
13.       bestSolution := solution;
14. return <bestEU,bestSolution>
```

Figure 2. The all-solutions backtracking algorithm.

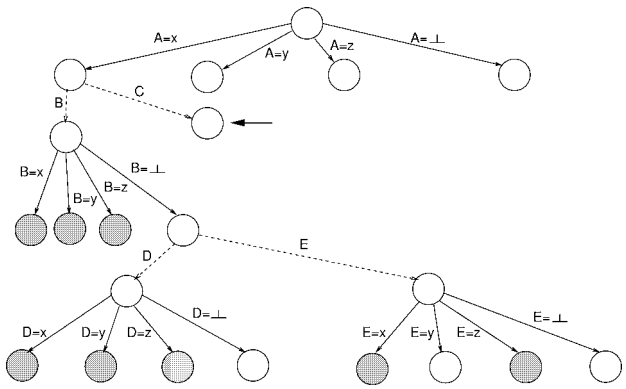



Figure 3. A trace of all-solutions backtracking on the example problem.

obtainable from any node in the tree is computed. We update this value each time we are about to consider a new probabilistic branch, using the information gained from the branches we have already explored. The search below a node is abandoned as soon as it is not possible to improve on the current best solution. In forward checking, each time a value is assigned to a variable, the assignment is propagated to all nodes below it in the transition tree. As in standard forward checking [11], any values in the domains which conflict with the current assignment are removed, and then search proceeds as before. Forward checking reduces the number of values considered at each node. It also induces early backtracking by informing the branch-and-bound heuristic if we have removed so many high-utility values from future domains so that we cannot improve on the current bound.

The pseudocode for the enhanced method is presented in figure 4. We maintain a single domain for each variable, regardless of how many nodes it appears in. In line 2, `LB` is the value that must be obtainable from this node to continue searching. In line 7, `propagate` removes all incompatible values from domains of variables in the set `future`($S_i$) (those that appear below $S_i$ in the transition tree), computes the maximum achievable utility at each node below $S_i$ after the domain reductions, builds an array (`UBs`) of the maximum achievable utilities for each of the children of $S_i$, and returns the total maximum achievable utility below $S_i$. The propagation maintains, for each variable, a set of variable/value pairs that record the removed value, and the variable that caused the removal. This is similar to the method described in [17].

Line 8 sets the value that must be obtained by summing over the probabilistic branches to improve on the current bound. This will be updated after searching below each branch. Line 10 computes the value that needs to be achieved below the currently selected probabilistic branch. In line 11, if this value is greater than or equal to the maximum achievable, we know we cannot improve on the bound for the current value assignment, so we stop considering branches, and move on to the next domain value. Line 12 is the recursive call to `FCB&B`. Line 13 integrates the returned value below the current branch into the target value. Line 19 retracts all the propagation caused by assigning the current value, using the sets of variable/value pairs created by `propagate`, to restore values to the domains of variables in `future`($S_i$). In line 20, we check against the known maximum achievable value below this node: if we have already achieved it, we do not consider any more values for this variable.

In the worst case, forward checking may require for each node a number of constraint checks exponential in the remaining depth of the tree; however, on average, the reduction in the size of the search space is much more significant. Results in [8] show that the enhancements can improve the search performance by an order of magnitude. In that paper, we go on to discuss a number of other issues, including a thresholding algorithm for incomplete search. It may be possible to include higher levels of achieving consistency during the search, but that appears to force us to maintain separate domains for each node in the tree, and duplicate many constraint checks. The use of null values in domains may also cause problems for higher consistency, as there are usually less deductions that can be made (if a variable has a null value, then it does not constrain any other variables). For this paper, we restrict our attention to forward checking.

## 4. Comparing BCSPs with MDPs

Our branching tree obeys the Markov property – the probability distribution over the successor node depends only on the current node, and not on the path taken to get to a node. We now show how to model the problem as an MDP.

The assignment of a value to a variable corresponds to an action. The aim is to find the optimal action for each node in the tree. In normal Markov decision problems, the transition probability depends on both the current state and the action; in this case, the transition probability is independent of the action. Simply adding in the actions to the

```
FCB&B(node Si, float LB) returns <utility,sol>

1.  if dom(Xsi = {} then return <0,{}>
2.  bestEU := LB
3.  stop := false
4.  for each v in dom(Xsi) while not(stop) do
5.    solution := {}
6.    EU := utility(Xsi <- v)
7.    if propagate(Xsi,v,future(Si),UBs)+EU > bestEU
8.      child_LB := bestEU - utility(Xsi) - sum(UBs)
9.      for each Sj in children(Si) do
10.       child_LB := child_LB + UBs[Sj]
11.       if child_LB >= UBs[Sj] break (to line 16)
12.       result := FCB&B(Sj, child_LB / P(Si -> Sj))
13.       child_LB := child_LB - result.utility * P(Si -> Sj)
14.       EU := EU + result.utility * P(Si -> Sj)
15.       solution := solution U result.solution
16.     if EU > bestEU
17.       bestEU := EU
18.       bestSolution := solution U {Xsi <- v}
19.   retract(Xsi,v,future(Si))
20.   if EU = maxUtility(Si) stop := true;
21. return <bestEU, bestSolution>
```

Figure 4. The forward-checking branch-and-bound algorithm.

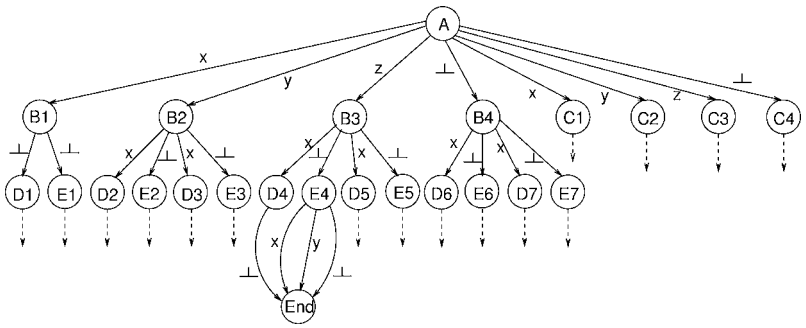



Figure 5. The MDP generated from a BCSP.

transition tree to replace the branches does not work, however. In our previous example, if we assign x to A in the root node, and we end up in the node containing B, then the action of assigning x to B is not available to us, because it would violate the constraint on A and B. The problem is that although the transition probabilities are independent of the history, the action set is not independent: the history determines the set of feasible actions at each node. Therefore, in order to apply the standard solution algorithms, we need to expand the tree to include the history in each node. Thus the node where B

arrives expands into multiple states in figure 5, corresponding to (B1) assigning x to A and B arriving next, (B2) assigning y to A, and B arriving next, (B3) assigning z to A, and B arriving next, and (B4) assigning null to A, and B arriving next. State (B1) has only one possible action, assigning null to B, because of the constraints. Note that the entire MDP has not been shown for reasons of space. States with a dashed arc leading from them do not have the full range of assignments shown. The probabilities and rewards have been omitted.

If we expand out the full tree (figure 5), we get a finite horizon, undiscounted Markov decision problem, to which we can apply any of the standard solution methods. But this MDP is exactly the tree explored by an all-solutions backtracking search applied to the original problem, before we compute the policy values. In the worst case, the size of this new tree is exponential in the depth of the old tree, by the same argument estimating the size of the backtracking search space, and thus although the MDP solution methods may be polynomial in the size of the MDP, generating the MDP could take exponential time. The BCSP methods are designed to cut down the work involved in generating this larger tree, and, in addition, compute the optimal policy at the same time, so we would expect the BCSP methods to be faster. We show below that generating the MDP by a backtracking search, and then solving using the DAG-SP algorithm is no faster than our underlying solution method. In addition, MDP generation/DAG-SP requires more space, as the states of the MDP have to be passed as input to the DAG-SP, while the BCSP methods never represent all states in the space at the same time. In the next section, we present some experiments which show that our forward checking branch-and-bound enhancement is, on average, faster.

As mentioned above, each MDP state corresponds to a node in the search tree, and each MDP action corresponds to a choice of value assignment. Each action has a probabilistic transition to a new state, corresponding to the probabilistic arrival of the next variable. Generating the MDP requires exactly the same constraint checks carried out by the all-solutions backtracking algorithm. Because of this correspondence, if we can show that the same arithmetical computations are performed at corresponding states/nodes in both the all-solutions backtracking algorithm and DAG-SP, we will have shown that the overall computations are equivalent.

At each search node, the BT algorithm does the following work. (i) For each assignment, the total expected utility is calculated by multiplying the expected utility of each of its children by the probability of the child, summing, and adding to the utility of the value assignment itself. This is performed once for each node. For leaf nodes, the expected utility is just the utility of the assignment. Note that these calculations are applied bottom-up. (ii) The optimal assignment is then selected by comparison.

At each state, the MDP/DAG-SP algorithm does the following work. (i) For each action, the value is calculated by multiplying the value of the successor states by the probability of reaching each state, and summing. This can be performed once for each state, using DAG-SP, starting at the terminal states. (ii) The optimal policy is then selected by comparison.

Thus the two algorithms perform equivalent computations. Therefore, generating

```
MDPgenFC(node Si) returns MDP <states, arcs>

1.  if dom(Xsi) = {} return <name(Xsi),{}>
2.  states := {name(Xsi)}
3.  arcset := {}
4.  for each v in dom(Xsi) do
5.    propagate(Xsi, v, future(Si))
6.    action := {}
7.    for each Sj in children(Si) do
8.      result := MDPgenFC(Sj)
9.      states := states U result.states
10.     action := action U {<name(Xsj), P(Si->Sj)>}
11.     arcset := arcset U result.arcs
12.   arcset := arcset U {<v,action>}
13. return <states,arcset>
```

Figure 6. The hybrid MDP/forward checking generation algorithm.

an MDP and solving it with DAG-SP is equivalent to all-solutions backtracking search without any pruning.

Finally, we could consider a hybrid approach, in which we apply our forward checking variant to the generation of the MDP, which will produce the same MDP, but (we would expect) in shorter time, and then apply DAG-SP. The pseudocode for this is shown in figure 6. We assume a naming function to create new state names, and we represent an MDP by states and directed arcs, where each arc has a probability value associated with a pair of states. We assume the propagate function is as before, but without any optimisation. Again, we would expect the BCSP methods to be faster since they incorporate the optimisation in the generation process, using the values to cut down on the search, and they end up with the same size of tree as the original problem. The only overhead incurred by BCSP is in the comparisons to the bounds. The experiments in the next section show that this overhead is negligible, and that the BCSP method is indeed more efficient.

## 5. Computational experiments

We compare three algorithms: (a) all-solutions backtracking search to generate the MDP from the transition tree; (b) the hybrid all-solutions with forward checking generation of the MDP; and (c) the full BCSP forward checking branch and bound. In these experiments, we consider only problems of type (ii) with binary constraints. We generate random problems in the style standard from constraints research. We start with the global set of variables, and randomly generate constraints: $p_1$ is the probability of there being a constraint between any given pair of variables; $p_2$ is the probability that a given tuple is disallowed in the constraint. We generate random trees according to the process described below. We fix the problems to contain 10 variables, with a domain size of 10.

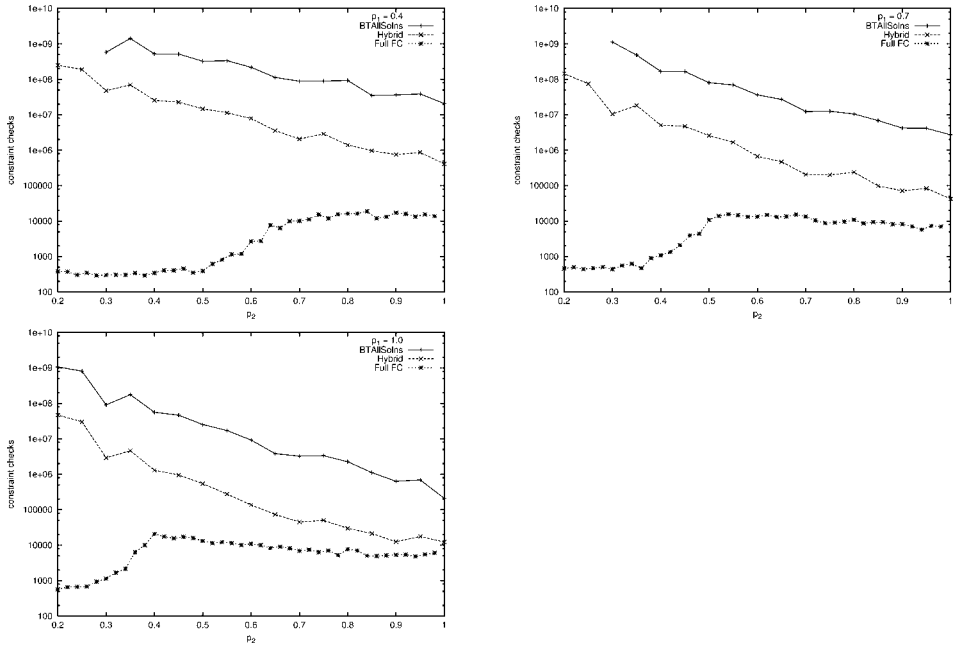



Figure 7. Constraint checks against tightness for random problems.

Table 1
Running times (in microseconds).

| | | | | |
|---|---|---|---|---|
| $p_1 = 0.4$ | $p_2$: | 0.2 | 0.8 | |
| | BCSP: | $1.00 \cdot 10^4$ | $1.94 \cdot 10^6$ | |
| | Hybrid | $1.00 \cdot 10^9$ | $2.36 \cdot 10^7$ | |
| $p_1 = 0.7$ | $p_2$: | 0.2 | 0.55 | 0.8 |
| | BCSP: | $1.00 \cdot 10^4$ | $1.24 \cdot 10^6$ | $1.02 \cdot 10^6$ |
| | Hybrid: | $1.03 \cdot 10^9$ | $2.46 \cdot 10^7$ | $3.50 \cdot 10^6$ |
| $p_1 = 1.0$ | $p_2$: | 0.2 | 0.4 | 0.8 |
| | BCSP: | $1.50 \cdot 10^4$ | $1.84 \cdot 10^6$ | $3.40 \cdot 10^5$ |
| | Hybrid: | $3.11 \cdot 10^8$ | $3.59 \cdot 10^7$ | $5.60 \cdot 10^5$ |

The trees are limited to a depth of 8. We generate utilities using a uniform distribution over [1, 50]. In figure 7 we show hardness curves for $p_1$ values of 0.4, 0.7 and 1.0, and we vary $p_2$ in steps of 0.05. The y-axis represents the median number of constraints checked (we have shown above that the number of search nodes examined and the number of mathematical calculations are never greater for BCSPs) over 20 runs at each data point. As we can see, the full BCSP method (c) requires significantly fewer constraint checks than the hybrid method (b), which in turn requires fewer checks than the simple method (a).

Note that the BCSP method (c) returns the optimal solution in those experiments,

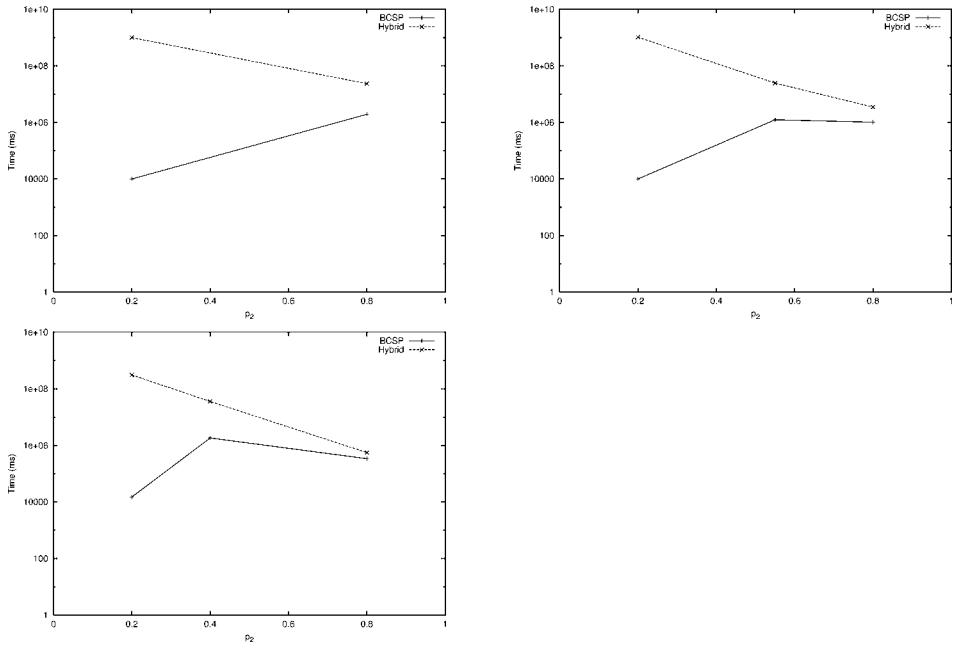



Figure 8. Graphical representation of table 1.

while in (a) and (b) we have yet to run DAG-SP. It is possible that the extra overhead of the optimisation could have increased the running time. However, in table 1 (with a graphical representation in figure 8) we show the median CPU times for (c) and (b) at different data points for each $p_1$ value, for 20 runs, with the CPU time measured in microseconds, which shows that the overhead is in fact negligible, and that the BCSP method always returns before the other method would have begun the optimisation.

## 6. Conclusion

We have considered a class of problems in which a sequence of tasks is presented to a solver, and the solver is required to make a decision on each task as it arrives. There are constraints on the decisions that can be made for the tasks. The sequence of tasks is not known for certain in advance, but it is assumed that the task arrivals can be modelled by a Markov process. The goal of the solver is to make decisions that are in some sense robust – each decision must maximise the chances of being able to find good assignments for the subsequent tasks. We have modelled such problems as branching constraint satisfaction problems, incorporating branch-and bound and forward checking on the constraints during the search, and we have established the complexity class. We then showed how the problems can also be modelled as Markov decision problems. We showed that the MDPs that are generated by the obvious translation from the original problems are equivalent to the trees searched by BCSP without the branch-and-bound or forward checking enhancements, and that this basic BCSP requires no more computation

than would be involved in solving the MDP. We also presented a hybrid MDP generation process incorporating forward checking. We showed experimentally that the full BCSP method, which includes optimisation, is significantly more efficient than both the simple and hybrid MDP generation methods, even before we apply any MDP solution methods. We showed also that the CPU time required by BCSP is significantly less than by the MDP generation methods. BCSPs also require less space, since the full set of states is never generated at the one time. For this class of problem, BCSPs could be considered to be an efficient method of generating and solving the implied MDPs, by integrating the optimisation with a constraint-based generation of the problem.

For future work, it would be interesting to use the theoretical approach of [14] to try to show that full BCSP method is more efficient than the MDP approach, rather than relying on experimental evidence using random problems. Currently, we are developing the BCSP approach, looking at incomplete methods, anytime algorithms, and achieving higher consistency during the search. We are also extending the method to deal with cyclic, infinite-horizon arrival processes. As part of this, we are extending the representation to include timed arrivals, enabling us to model more realistic scheduling problems. Finally, we are beginning to investigate how we can apply these techniques in decision-theoretic planning problems.

## Acknowledgments

We are grateful to Toby Walsh for the questions that prompted this paper, and to the anonymous referees for their helpful comments and suggestions.

## References

[1] R.E. Bellman, *Dynamic Programming* (Princeton University Press, Princeton, NJ, 1957).
[2] C. Boutilier, T. Dean and S. Hanks, Decision-theoretic planning: structural assumptions and computation leverage, Journal of Artificial Intelligence Research 1 (1999) 1–93.
[3] C. Boutillier, R. Dearden and M. Goldszmidt, Exploiting structure in policy construction, in: *Proceedings of IJCAI-95* (1995).
[4] J.A. Boyan and A.W. Moore, Learning evaluation functions for large acyclic domains, in: *Machine Learning: Proceedings of the Thirteenth International Conference*, ed. L. Saitta (1996) pp. 63–70.
[5] T.H. Cormen, C.E. Leiserson and R.L. Rivest, *Introduction to Algorithms* (MIT Press, Cambridge, MA, 1990).
[6] R. Dechter and A. Dechter, Belief maintenance in dynamic constraint networks. in: *Proceedings of AAAI-88* (1988) pp. 37–43.
[7] 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).
[8] D.W. Fowler and K.N. Brown, Branching constraint satisfaction problems for solutions robust under likely changes, in: *Proceedings of CP2000*, Singapore (2000) pp. 500–504.
[9] D.W. Fowler and K.N. Brown, Modelling and solving problems with probable changes, in: *ECAI 2000 Workshop on Modelling and Solving Problems with Constraints*, Berlin (2000) pp. E:1–9.
[10] M.L. Ginsberg, A.J. Parkes and A. Roy, Supermodels and robustness, in: *AAAI-98* (1998) pp. 334–339.

[11] M. Haralick and G.L. Elliott, Increasing tree-search efficiency for constraint satisfaction problems, Artificial Intelligence 14 (1980) 263–313.
[12] P.V. Hentenryck, Y. Deville and C. Teng, A generic arc-consistency algorithm and its specializations, Artificial Intelligence 57 (1992) 291–321.
[13] R.A. Howard, *Dynamic Programming and Markov Processes* (MIT Press/Wiley, 1960).
[14] G. Kondrak and P. van Beek, A theoretical evaluation of selected backtracking algorithms, Artificial Intelligence 89 (1997) 365–387.
[15] M.L. Littman, Initial experiments in stochastic satisfiability, in: *Proceedings of AAAI-99* (1999) pp. 667–672.
[16] M.L. Littman, T.L. Dean and L.P. Kaelbling, On the complexity of solving Markov decision problems, in: *11th Annual Conference on Uncertainty in Artificial Intelligence (UAI-1995)* (1995).
[17] P. Prosser, Hybrid algorithms for the constraint satisfaction problem, Computational Intelligence 9(3) (1993) 268–299.
[18] M.L. Puterman, *Markov Decision Processes* (Wiley, New York, 1994).
[19] G. Verfaillie and T. Schiex, Solution reuse in dynamic constraint satisfaction problems, in: *Proceedings of AAAI-94*, Seattle, WA, USA (1994) pp. 307–312.
[20] 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 (1997).