Caching Search States in Permutation Problems

Barbara M. Smith

Cork Constraint Computation Centre, University College Cork, Ireland
b.smith@4c.ucc.ie

Abstract. When the search for a solution to a constraint satisfaction problem backtracks, it is not usually worthwhile to remember the assignment that failed, because the same assignment will not occur again. However, we show that for some problems recording assignments is useful, because other assignments can lead to the same state of the search. We demonstrate this in two classes of permutation problem, a satisfaction problem and an optimization problem. Caching states visited has proved effective in reducing both search effort and run-time for difficult instances of each class, and the space requirements are manageable.

1 Introduction

The aim of this paper is to show that for some types of constraint problem it can be worthwhile to cache information about the assignments visited during the search for solutions; this information can be used to prune parts of the search visited later and avoid wasted effort.

When a constraint satisfaction problem (CSP) is solved by depth-first backtracking search, and the search backtracks, the failure of the current assignment is due to some inconsistency that is not explicitly stated in the constraints. The search has discovered that the assignment cannot be extended to a solution; it is a *nogood*. There is no point in recording the assignment itself, in order to avoid it in future, because the search will never revisit it anyway. However, in some problems, assignments can occur later in the search that are *equivalent* to the failed assignment, in the sense that they leave the remaining search in the same state, and hence whether or not the equivalent assignment will fail can be determined from the failed assignment.

In such a case, if assignments are recorded and an assignment occurs later in the search that is equivalent to one that has already failed, the search can immediately backtrack without rediscovering the same failure. Permutation problems are a promising type of problem where equivalent states might occur. We demonstrate the value of recording assignments in two classes of permutation problem, where both search effort and run-time can be considerably reduced.

Previous work on recording nogoods has depended on identifying a subset of the failed assignment that is responsible for the failure and adding this smaller nogood to the CSP as a new constraint; although the assignment will not occur again during the search, the subset may. For instance, Frost and Dechter [5] use a backjumping algorithm that identifies a conflict set causing the failing

P. van Beek (Ed.): CP 2005, LNCS 3709, pp. 637–651, 2005.

assignment. Katsirelos and Bacchus [10] adapt the nogood recording techniques successful in SAT, by keeping a record of the reasons for removing a value from the domain of a variable and using these reasons, with the constraints that each variable must have a value, to construct nogoods on failure. Bayardo and Mirankar [1] discuss the fact that the space requirements of these methods of learning nogoods during search can be prohibitive without some restriction on the nogoods learnt, or some way of deleting nogoods no longer considered useful.

In comparison to these methods, the ideas discussed here are specific to a particular type of problem. On the other hand, they have the advantage that it is not necessary to identify a reason for a failure. Furthermore, the process of matching an assignment with the cache is simplified by the fact that matching assignments can only occur at the same level of the search tree. The results will show that for the examples considered, the space requirements are manageable, even for the most difficult problems requiring extensive search.

Two classes of permutation problem are considered below: a satisfaction problem and an optimization problem. In both cases, caching is very beneficial in speeding up the solution of difficult problems. It is slightly more complex to incorporate caching into the search for optimal solutions, but essentially the same method is used. Other applications of the same idea are also discussed.

2 Permutation Problems and Caching

In this section, we discuss how equivalent assignments can arise in permutation problems. Two assignments can be considered equivalent if they leave the search in the same state, i.e. the subproblems consisting of the not-yet-assigned variables and their current domains (after any constraint propagation) are the same and the assignments to the future variables that are consistent with the assignments already made are the same in both cases. Hence, if an assignment cannot be extended to a complete solution, i.e. is a nogood, neither can any equivalent assignment. Two assignments can only be equivalent if they involve the same set of variables, because the set of variables that have not yet been assigned must be the same. Since equivalent assignments leave the search in the same state, we can think of a set of equivalent assignments as a (search) state; and they will be referred to as states below.

A *permutation problem* is a CSP with the same number of values as variables in which the constraints restrict every variable to have a different value [8]. If the search algorithm assigns the variables in lexicographic order, an assignment of size k consists of k of the values assigned to the first k variables. For some permutation problems, whether or not the assignment can be extended to a complete solution depends only on the *set* of values assigned, rather than on the order in which they are assigned to the variables, together with possibly a few other features of the assignment. In the 'Black Hole' problem discussed in the next section, for instance, two assignments to the first k variables are equivalent if the set of values assigned to the first $k-1$ variables is the same, and the kth variable is assigned the same value in both. Recording the set of values,

along with the other features, if any, can then allow the inconsistency of future assignments using the same set of values to be recognised without further search.

3 The Game of 'Black Hole'

The first problem class examined is a class of satisfaction problems, i.e. just one solution is required, or determination that there is no solution. It arises from a game of Patience or Solitaire, that can be straightforwardly modelled in constraint programming. 'Black Hole' was invented by David Parlett, who describes it thus:

> "*Layout* Put the Ace of spades in the middle of the board as the base or 'black hole'. Deal all the other cards face up in seventeen fans of three, orbiting the black hole.
> *Object* To build the whole pack into a single suite based on the black hole.
> *Play* The exposed card of each fan is available for building. Build in ascending or descending sequence regardless of suit, going up or down ad lib and changing direction as often as necessary. Ranking is continuous between Ace and King. For example, a start might be made as follows: A-K-Q-K-A-2-3-4-3- and so on."

The table below shows an instance of the game: the 17 columns represent the 17 'fans' of 3 cards each:

7♠	3♢	5♠	T♣	6♠	J♣	J♠	4♢	7♡	9♢	7♢	2♣	3♡	7♣	3♠	6♢	9♣
J♡	4♠	K♢	Q♢	T♠	T♢	A♣	9♠	9♡	Q♠	K♠	Q♡	5♡	K♣	8♡	J♢	2♢
2♡	5♣	T♡	3♣	8♣	A♡	2♠	8♠	5♢	K♡	Q♣	4♡	6♣	6♡	A♢	4♣	8♢

We can represent a solution as a sequence of the 52 cards in the pack, starting with the ace of spades, the sequence representing the order in which the cards will be played into the Black Hole. The top card in each column is available to add to the sequence of cards being built. A solution to this game is:

A♠-2♣-3♠-4♢-5♠-6♠-7♠-8♡-9♠-8♠-9♣-T♠-J♠-Q♡-J♡-T♣-J♣-Q♢-K♢-A♣-2♠-3♡-2♢-3♣-4♡-5♡-6♣-7♡-8♣-7♣-6♢-7♢-8♢-9♡-T♡-9♢-T♢-J♢-Q♠-K♠-A♡-K♡-Q♣-K♣-A♢-2♡-3♢-4♠-5♣-6♡-5♢-4♣

A constraint programming model for this problem is described in [6]. It is modelled as a permutation problem: the cards are numbered 0 (the ace of spades) to 51 and the sequence of cards is represented as a permutation of these numbers. There are two sets of dual variables: x_i represents position i in the sequence, and its value represents a card; y_j represents a card and its value is the position in the sequence where that card occurs. These are linked by the usual channelling constraints: $x_i = j$ iff $y_j = i$, $0 \leq i, j \leq 51$. The constraints that a card covering another card must be played before it are represented by $<$ constraints on the corresponding y_j variables. Constraints between x_i and x_{i+1}, $0 \leq i < 51$, ensure that each card must be followed by a card whose value is one higher or one lower.

We set $x_0 = 0$, i.e. the ace of spades is the first card in the sequence. The search strategy assigns values to the variables $x_i, i = 1, 2, ..., 51$ in order, i.e. the sequence of cards is built up as it would be played in the game. Some equivalent sequences that would result from exchanging two cards of the same rank but different suits are eliminated by *conditional symmetry breaking* constraints, as described in [7]; a conditional symmetry holds only within a subproblem of the CSP. Eliminating the conditional symmetry has a huge impact on the search, for any instance that requires a significant amount of backtracking to find a solution.

This model has been implemented in ILOG Solver 6.0 and applied to 2,500 randomly generated instances that were used to produce the results in [6]. The performance of the CP model is highly skewed: half of the instances take fewer than 100 backtracks to solve, or to prove unsatisfiable, whereas the most difficult instances take millions of backtracks. This is shown in Figure 1, where the instances are sorted by search effort. About 12% of the instances are unsatisfiable; for most of these, the proof is trivial (for instance, the game cannot be won if the top layer of cards contains neither a 2 nor a King). On the other hand, the instances that are most difficult for the CP model are also unsatisfiable.

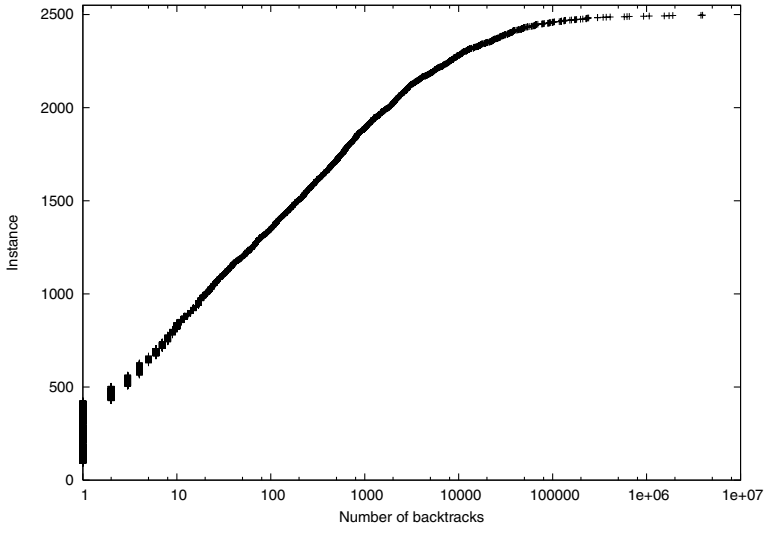

Fig. 1. Number of backtracks to solve 2500 random instances of 'Black Hole'

4 Caching States in 'Black Hole'

At any point during search where the current assignment is about to be extended, a valid sequence of cards has been built up, starting from the ace of spades. Whether or not the sequence can be completed depends only on the cards that have been played and the last card; apart from the last card, the order of the previously-played cards is immaterial.

For instance, suppose the following sequence of cards occurs during search (assuming that in some game the sequence is possible, given the initial layout of the cards):

A♠-2♣-3♠-4♢-5♠-4♣-3♣-2♠-A♣-K♢-A♢-2♢-3♢

If at some later point in the search, the following sequence occurs:

A♠-K♢-A♢-2♣-3♠-2♠-A♣-2♢-3♣-4♣-5♠-4♢-3♢

the second sequence will not lead to a solution. The set of cards in both sequences is the same, and they end with the same card. Hence, in both cases, the remaining cards and their layout are the same. Since the first sequence did not lead to a solution (otherwise the search would have terminated), the second will not either.

Based on this insight, the search algorithm in Solver has been modified to record and use the relevant information. The search seeks to extend the current sequence of cards at *choice points*. Suppose that the first unassigned variable is x_k and the values of the earlier variables are $x_0 = 0, x_1 = v_1, ..., x_{k-1} = v_{k-1}$. (Some of these values may have been assigned by constraint propagation rather than previous choices.) The search is about to extend this assignment by assigning the value v_k to x_k. A binary choice is created between $x_k = v_k$ and $x_k \neq v_k$, for some value v_k in the domain of x_k. The set of cards played so far, $\{v_1, v_2, ..., v_{k-1}\}$ and the card about to be played, v_k, are then compared against the states already cached. If the search has previously assigned $\{v_1, v_2, ..., v_{k-1}\}$ to the variables $x_1, x_2, ..., x_{k-1}$, in some order, and v_k to x_k, then the branch $x_k = v_k$ should fail. If no match is found, a new state is added to the cache, consisting of the set of cards already played and the card about to be played, and the search continues. In the example, when the 3♢is about to be added to the sequence, the set {2♠, 3♠, 5♠, A♢, 2♢, 4♢, K♢, A♣, 2♣, 3♣, 4♣}, and $x_{12} = 3♢$, would be compared with the states already visited.

(Note that constraint propagation may also have reduced the domains of some future variables to a single value, which will therefore have been assigned, but this can be considered as part of the state of the remaining search left by the sequence $x_0 = 0, x_1 = v_1, ..., x_{k-1} = v_{k-1}$.)

The implementation represents the set of cards in the current sequence, excluding the A♠, as a 51-bit integer, where bit $i = 1$ if card i is in the set, $1 \leq i \leq 51$. The current state can only match a state in the cache if both the number of cards played $(k - 1)$ and the current card (v_k) match. Hence, the cache is indexed by these items. It is stored as an array of extensible arrays, one for each possible combination of $k - 1$ and v_k: this is a somewhat crude storage system, but has proved adequate for this problem. Within the relevant extensible array, the integer representing $\{v_1, v_2, ..., v_{k-1}\}$ is compared with the corresponding stored integers, until either a match is found, or there is no match. In the former case, the search backtracks: the current state cannot lead to a solution. Otherwise, the integer representing $\{v_1, v_2, ..., v_{k-1}\}$ is added to the array, $x_k = v_k$ is added to the sequence being built and the search continues.

When an assignment of length k fails, it would in theory be possible to represent this information as a k-ary constraint. In 'Black Hole', the failure has revealed that assigning $\{v_1, v_2, ..., v_{k-1}\}$ to $\{x_1, x_2, ..., x_{k-1}\}$, in any order, and v_k to x_k is inconsistent, and a constraint expressing this could be added to the CSP. However, as will be seen, tens of thousands of states may be cached in solving an instance of 'Black Hole'; storing and processing so many constraints added during search would undoubtedly take more space and time than the caching proposed here.

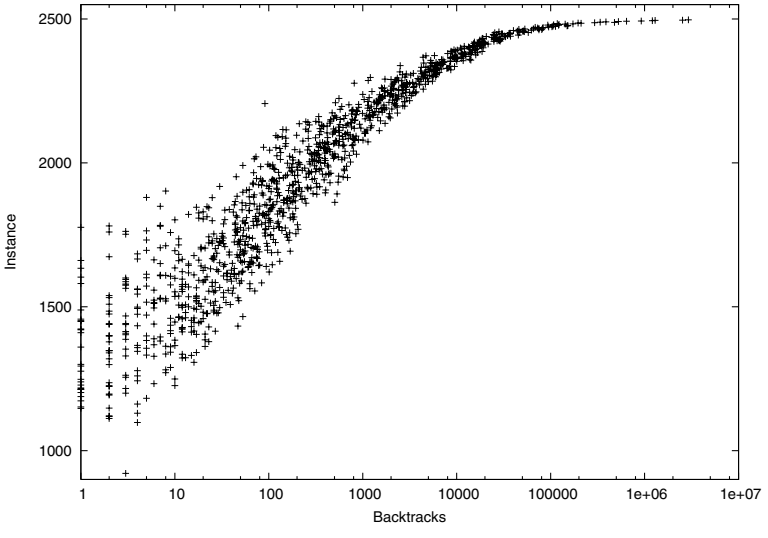

Fig. 2. Solving 2500 random instances of 'Black Hole': difference in number of backtracks between the original search and the search with cached states, instances in the same order as Figure 1

Figure 2 shows the reduction in the number of backtracks required to solve the 2,500 instances resulting from caching states. Only the instances which take fewer backtracks with caching than without are shown, but the instances are given the same numbering as in Figure 1 (so that the most difficult instance from Figure 1 is still shown as instance 2500). It is clear that the saving in search effort increases with the search effort originally expended.

For all but 15 of the 1,206 instances that take 50 or fewer backtracks to find a solution, caching states visited makes no difference to the search effort. However, since few states are cached in these cases, the run-time is hardly affected either. Solver occasionally reports a longer run-time with caching than without, by up to 0.01 sec., but only for instances that take little time to solve in either case.

At the other end of the scale, the instances that take more than 1 million backtracks with the original search are shown in Table 1; these instances have

Table 1. Number of backtracks and run-time in seconds (on a 1.7GHz Pentium M PC, running Windows 2000) to solve the most difficult of the 2,500 'Black Hole' instances, with and without caching states visited

No caching		Caching	
Backtracks	Time	Backtracks	Time
3,943,901	1,427.93	1,020,371	431.33
3,790,412	1,454.16	1,259,151	509.94
1,901,738	721.07	606,231	251.01
1,735,849	681.57	528,379	233.40
1,540,321	582.71	619,735	257.95
1,065,596	398.44	423,416	176.01

no solution. For these instances, caching states visited reduces the search effort by at least 60%; for the most difficult instance, the reduction is nearly 75%. In spite of the unsophisticated storage of the cache, the saving in run-time is nearly as great; more than 55% for all six instances, and 70% for the most difficult instance.

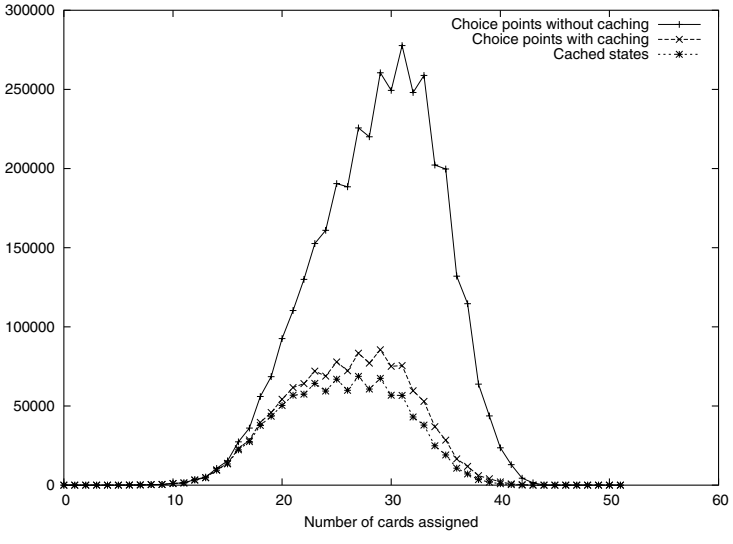

Fig. 3. Proving insolubility for the most difficult 'Black Hole' instance in the sample, with and without caching

To show more clearly how caching affects the search, Figure 3 shows the search profile for the most difficult instance of the 2,500 for the original search. The number of choice points is plotted against the number of variables assigned when the choice point is created, so showing the depth in the search where the choice point occurs. The number of cached states at each depth is also shown;

this is equal to the number of choice points where no matching state is found in the cache and the search is allowed to continue.

The total number of cached states for the instance shown in Figure 3 is about 1.25 million ($< 2^{21}$). In a permutation problem, the number of possible assignments is at most the number of subsets of the values, i.e. 2^n, where n is the length of the sequence, in this case effectively 51; hence, this is an upper bound on the number of states that need to be cached during the course of search. However, in this case, most of the subsets of the cards are not feasible states, since a valid sequence cannot be constructed in which the cards follow each other correctly in ascending or descending rank. Hence, the number of possible cached states is much less than 2^{51}, even for the difficult unsatisfiable instances.

5 An Optimization Problem: Talent Scheduling

In this section, the application of these ideas to an optimization problem is considered. The rehearsal problem is prob039 in CSPLib: constraint programming approaches to solving it have been discussed in [11]. The talent scheduling problem [2] is a generalization of the rehearsal problem, arising in film production. A film requires a certain number of days of filming, which can be shot in any order. Each day's filming (called a 'scene' below) requires some subset of the actors. Actors are paid from the first day that they are required to the last day, including any day when they are not working. Hence, the schedule should minimize the cost of paying actors while they are waiting for their next scene but not working; actors are paid different rates, so the cost is the waiting time, weighted by the pay rates of the actors. The rehearsal problem is similar, except that it is simply the total waiting time that is minimized: this is equivalent to the actors all being paid the same rate.

Again this is a permutation problem: a schedule is a permutation of the scenes. As before, we can define two sets of dual variables: s_i represents position i in the sequence, and its value represents a scene; d_j represents a scene and its value is the position in the sequence where that scene occurs. The channelling constraints are: $s_i = j$ iff $d_j = i$, $1 \leq i, j \leq n$, where n is the number of scenes to be shot and hence the length of the sequence. There are constraints in the model to allow the waiting time for each actor to be derived from the sequence of scenes: these are described in [11]. The model used for the experiments described in this paper differs slightly from the one described in [11]: in that case dominance rules were added as constraints to the CP model whenever there is a pair of scenes i and j such that every actor required for i is required for j, and j requires one additional actor, a. The rules specify that if scene i is before scene j in the sequence, actor a must not be required until after scene i. For the experiments described in this paper, similar rules are used to cover the case that scene j requires *two* additional actors.

An optimal solution is found using the default optimization strategy provided by ILOG Solver; the cost of the best sequence found so far becomes an upper bound on the cost of any sequence found in future. As the search proceeds,

this constraint on the cost becomes increasingly tight, so that eventually no further sequence can be found with cost less than the incumbent solution, which has therefore been proved optimal. Because any permutation of the scenes is a feasible schedule, the first solution is found immediately, simply sequencing the scenes according to the order that they appear in the data.

The search strategy used in [11] is to assign scenes from the ends to the middle of the sequence, i.e. the variables are assigned in the order $s_1, s_n, s_2, s_{n-1}, ...$ The advantage of this over building up the sequence consecutively from the start is that if an actor is assigned to a scene in the first part of the sequence and also to a scene in the second part of the sequence, that actor's total waiting time is known: it does not depend on the order of the remaining scenes. Hence, partial sequences that will be more expensive than the best solution found so far can be pruned early.

For this problem, with this search strategy, a state consists of the set of scenes already placed at the start of the sequence, and the set of scenes scheduled at the end of the sequence. Because it is an optimization problem, we also need to record the cost associated with the partial sequence: that is, the waiting cost during the scenes already sequenced that is incurred for actors that are on set but not working during those scenes.

In the 'Black Hole' problem, if a search state matches one of those in the cache, this branch of the search can be pruned. In this case, however, if there is a matching state in the cache, but the current state is cheaper, then it may lead to a better complete solution than the best found so far, and so the search should continue. In that case, the cost associated with the cached state is replaced by the cost of the current partial sequence.

The cache is indexed only by the total number of scenes sequenced, corresponding to the depth in the search tree where the state occurs. Given the search strategy, if this number is even, say $2m$, then m scenes have been sequenced in the first part of the sequence and m in the last part; if the number is odd, say $2m+1$, then $m+1$ scenes are in the first part and m in the second. For problems that require a lot of search to find an optimal solution, there can be many states cached at some levels of search. To speed up the search for a matching state, the cache at each level is divided between a fixed number of extensible arrays, and the states are distributed evenly over these arrays using a hash function. Applying the hash function to the current state gives the index of the array where any possible matching state will be stored. Again, this method could be made more sophisticated so that matching is faster, but as will be seen from the results, this method is good enough to speed up search significantly and so demonstrate that caching is worthwhile.

When a state is stored in the cache, the search continues and tries to complete the sequence at a cost lower than the best solution so far. If this can be done, then conceivably it would be worth storing the cost of completing the sequence in the cache. Then if any future partial sequence matches the cached state, with lower cost than the cost of the stored partial sequence, the cost of completing it can be immediately known, and so whether it can beat the incumbent solution.

This might allow some states whose cost is lower than the previous occurrence of the same state to be pruned, because the minimum cost of a complete sequence based on this partial sequence cannot beat the incumbent solution. This has not been done for several reasons; first, it would complicate the algorithm to return to the cache to store completion costs for all partial sequences leading to each new solution, and would mean that satisfaction and optimization problems would be treated very differently. Secondly, relatively few solutions of successively lower cost are found during the course of the search; most sequences are never completed, because at some point their cumulative cost is higher than the best known solution. Hence, storing the completion cost in the cache seems unlikely to give a great reduction in search, and would certainly complicate the algorithm and make it more difficult to generalize.

6 Talent Scheduling Results

In this section, the results of using caching are presented, both for the rehearsal problem described in CSPLib, and for a number of talent scheduling problems, which are much more difficult to solve. The 'Mob Story' problem is derived from the problem in [2] (based on a film of that name); the data for this problem is also given in CSPLib. It has 20 scenes to sequence. The 'Mob Story$x$' instances are derived from it by taking the first x scenes in the data.

Tables 2 and 3 show the effect of caching states for these relatively small problems together with the rehearsal problem. In Table 2, as in [11], the sequence is built from the ends to the middle, as described earlier. In Table 3, the sequence is built from the start to the end, i.e. the search variables are assigned in the order $s_1, s_2, ..., s_n$. This makes implementing caching simpler, since the state consists of only the set of scenes at the start of the sequence.

Table 2. Solving small instances of the talent scheduling problem, with and without caching, building the sequence from both ends to the middle

Problem	No caching		Caching		
	Backtracks	Time	Backtracks	Time	Cached states
rehearsal	286	0.04	276	0.04	204
Mob Story10	289	0.05	281	0.06	236
Mob Story12	2,859	0.27	2,579	0.33	1,670
Mob Story14	15,598	1.64	10,439	1.48	5,597
Mob Story15	41,796	4.71	23,565	3.51	10,833
Mob Story	1,026,328	132.93	405,888	64.71	136,765

As already claimed, building the sequence from the ends to the middle is much faster than building from start to end. However, caching states makes a much greater difference when the worse variable ordering is used; caching with the poor variable ordering is better than the 'ends-to-middle' variable ordering without caching. Caching is still very worthwhile for the larger instances

Table 3. Solving small instances of the talent scheduling problem, with and without caching, building the sequence from start to end

Problem	No caching		Caching		
	Backtracks	Time	Backtracks	Time	Cached states
rehearsal	1,734	0.12	967	0.07	301
Mob Story10	1,054	0.17	838	0.13	291
Mob Story12	11,613	1.46	6,765	0.64	1,387
Mob Story14	350,991	56.19	68,134	6.09	7,341
Mob Story15	758,270	143.84	109,381	10.98	11,781
Mob Story	13,614,469	1,917.92	658,784	83.93	72,382

Table 4. Solving larger instances of the talent scheduling problem, with and without caching, and with both variable orders

	Build sequence start to end, with caching			Build sequence ends to middle				
				No caching		Caching		
Problem	Backtracks	Time (sec.)	Cached states	Backtracks	Time (sec.)	Backtracks	Time (sec.)	Cached states
film105	536,299	51.18	61,100	459,071	48.10	118,361	16.07	40,511
film116	1,160,295	143.72	81,084	2,102,591	277.96	744,481	125.8	225,314
film119	1,505,228	97.49	127,459	1,493,988	171.47	526,392	70.80	144,226
film118	2,333,385	178.22	201,115	2,618,066	315.74	606,591	93.10	205,190
film114	2,569,252	217.21	162,027	4,909,250	472.79	1,032,902	127.00	267,526
film103	4,723,274	313.18	215,354	2,628,434	250.42	607,935	76.69	180,133
film117	6,303,052	396.04	193,163	4,078,225	384.52	651,781	76.86	174,100

with the 'ends-to-middle' variable ordering, and the combination gives the best performance overall. However, with this variable ordering, caching reduces the run-time only by half for the Mob Story problem, compared with an order of magnitude reduction with the poor variable ordering. Evidently, the better variable ordering already leads to less wasted search, and so gives less scope for further reductions from caching.

Finally, Table 4 gives results on randomly-generated instances based on the characteristics of the Mob Story problem. These instances were generated with originally 20 scenes, as in the Mob Story problem. However, two scenes requiring the same set of actors can clearly be treated as one scene taking two days to shoot: requiring these scenes to be sequenced consecutively will not affect the optimality of any solution. After merging scenes in this way, most of these instances have slightly fewer than 20 scenes. Even so, they proved to be more difficult than the original Mob Story problem, overall, and so have not been attempted with 'start-to-end' variable ordering and no caching.

As with the Mob Story problem, the poor variable ordering with caching states gives better performance on the whole than the better variable ordering without caching, and the 'ends-to-middle' ordering with caching gives better results still. The size of the cache does not present any difficulty for these instances. However, the cache size is much closer to 2^n than in the 'Black Hole' problems:

since every sequence represents a feasible schedule, a greater proportion of the possible subsets is likely to be met during search than in the 'Black Hole' problems, where many subsets cannot form feasible sequences. The 'ends-to-middle' ordering does not require more states to be cached, on average, than 'start-to-end' ordering. This is somewhat surprising, since the total number of possible states is larger, if the state consists of two subsets of the scenes (representing the first and last parts of the sequence) rather than one.

Figure 4 gives a similar search profile to Figure 3 for this problem class: it is based on the most difficult instance shown in Table 4. Recall that the search can extend an assignment even when it matches a state in the cache, provided that the cost of the assignment is lower than the stored cost. Hence, cached states can be 're-used' when their associated costs are updated, and so the number of cached states in relation to the number of choice points is much smaller than in the 'Black Hole' problems.

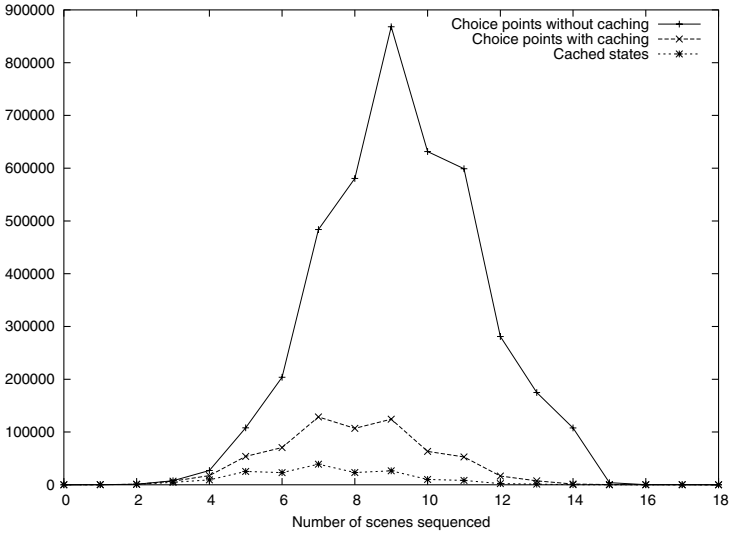

Fig. 4. Profile of number of choice points, with and without caching, and number of states cached, at each depth in search, for the random instance film117

7 Discussion

Caching would be a potentially valuable way of avoiding wasted search in other permutation problems, as well as those discussed here, though not all are suitable. For instance, Langford's problem (prob024 in CSPLib) is not a suitable candidate: if we consider a partial assignment to the first k variables representing the positions in the sequence, then whether or not the assignment can be extended to a solution depends on the order of the values assigned to the variables and not just the set of values. On the other hand, Fink and Voss [3]

discuss a number of sequencing problems that could potentially be modelled as permutation problems and are suitable for caching. The problems that they discuss are related to the talent scheduling problem, but have a variety of different objectives. For instance, 'minimization of the number of simultaneously open stacks' is equivalent to the talent scheduling problem but with the objective to minimize the maximum number of actors on set (either acting in the current scene or waiting) at any time. We could expect that caching could be useful for this problem, just as in the talent scheduling problem itself.

Focacci and Shaw [4] describe a similar approach to that in this paper, in the context of the Travelling Salesman Problem and the TSP with Time Windows (which are again permutation problems). They record nogoods during search and use local search to test whether the current assignment (a sequence of cities) can be rearranged to give a lower cost assignment that is an extension of a nogood; if so, the current assignment can be pruned.

Jefferson, Miguel, Miguel and Tarim [9] discuss the game of Peg Solitaire, modelled as a CSP. The game is played with pegs on a board studded with holes; a peg can jump over a neighbouring peg into a hole beyond, and the jumped-over peg is removed. The aim is to start from a state where all the holes but one are filled, to a state where only one peg is left. This is not a permutation problem, but it has some similarities and the authors noted that the same state of the board can be reached in multiple ways.

Jefferson *et al.* consider the existence of different paths to the same state as a form of symmetry, and sets of equivalent paths as symmetry equivalence classes. However, this does not seem a useful point of view, since the 'symmetries' are not identified, only equivalent paths. They attempted to deal with equivalent board states by preprocessing to find sets of equivalent paths and adding constraints to forbid all but one path in each set, as in conventional symmetry breaking. They found the sets by exhaustive search, which is only practicable for short paths. As a result, they could only eliminate equivalent board states occurring near the top of the search tree, which did not lead to great benefits.

On the other hand, Peg Solitaire seems a good candidate for caching states dynamically as they are encountered during the search, as described in this paper, even though this is not a permutation problem. Whether or not the game can be completed from a given board state does not depend on how that state was reached, so that this is similar to 'Black Hole'. Moreover, a given board state can only occur at a particular depth in the search (games require a fixed number of moves to complete). Hence, a cache of board states could be indexed by the depth in search, just as in the 'Black Hole' and talent scheduling examples. This example suggests that caching states during search could have wider application than permutation problems; the key feature is that different assignments should lead to the same state of the search.

The method described in this paper assumes that variables will be assigned in a static order. In the 'Black Hole' problem, for instance, an assignment to a subset of the variables that does not represent a consecutive sequence of cards would not leave the search in the same state as an assignment of the same values to

these variables in a different order. The requirement of a static variable ordering is not a restriction, however, since a static ordering that builds up the sequence consecutively is a good search strategy for problems requiring the construction of a sequence, such as 'Black Hole', talent scheduling, Peg Solitaire and so on.

The space requirements of the cache are not an issue for the problems investigated in this paper. For 'Black Hole', especially, many of the potentially 2^{51} states are not feasible, and so are never visited and never cached. Space is potentially more likely to present difficulties for the talent scheduling problem, where in theory every possible state could be visited, although since assignments that already exceed the current cost bound fail, this cannot happen in practice. The instances reported here are near the limit of what can be solved in a reasonable time with the current model and search strategy, so that solving larger instances, and thereby needing a larger cache, is not practicable. However, a better value ordering heuristic, for instance based on that described by Cheng *et al.* [2], should mean that the first solution found would be much closer to optimal than at present. This would allow larger instances to be solved; but at the same time the better cost bound would also limit the number of states cached. Future work will investigate the overall effect on the size of the cache. If the cache size ever becomes too large, it would be possible to limit its size, for instance by simply not saving more states when the cache reaches a preset size, thus trading a smaller reduction in run-time for space.

8 Conclusion

It has been shown that in some classes of problem, two assignments to the same set of variables can leave the search in the same state; hence if one assignment is a nogood, so is the other. By caching the problem-specific details that will allow equivalent states to be recognised, wasted search exploring equivalent assignments can be avoided. In satisfaction problems, such as the 'Black Hole' problem, if the current assignment matches one in the cache, an equivalent assignment has already failed; hence the current assignment will also fail and the search should backtrack. In optimization problems, such as the talent scheduling problem, the cost of the current assignment should also be compared with the cost associated with the cached state; the current assignment should fail if it is at least as expensive as the cached state. The two case studies considered in this paper have shown that caching can reduce the run-time for difficult instances by at least half, and sometimes by an order of magnitude, depending on the problem and the instance. It does not always give any benefit for instances that are already easy to solve, but in those cases does not increase run-time either. Although in previous work, recording nogoods has been problematic because of the very large number of nogoods generated, the number of states cached has not presented any difficulty for the cases investigated here, and looking for a matching state in the cache has not incurred a heavy overhead, in spite of somewhat crude storage methods. Permutation problems in general seem most likely to give rise to equivalent states; the example of Peg Solitaire, which is

not a permutation problem, shows that caching states could also have wider application.

Acknowledgments

I am grateful to Paul Shaw for providing Solver 6.0 code for the rehearsal problem and to Ian Miguel and Peter Stuckey for helpful discussions. This material is based on works supported by the Science Foundation Ireland under Grant No. 00/PI.1/C075.

References

1. R. J. Bayardo, Jr. and D. P. Miranker. A Complexity Analysis of Space-Bounded Learning Algorithms for the Constraint Satisfaction Problem. In *Proceedings AAAI-96*, pages 298–304, 1996.
2. T. Cheng, J. Diamond, and B. Lin. Optimal scheduling in film production to minimize talent hold cost. *J. Optimiz. Theory & Apps*, 79:197–206, 1993.
3. A. Fink and S. Voss. Applications of modern heuristic search methods to pattern sequencing problems. *Computers & Operations Research*, 26:17–34, 1999.
4. F. Focacci and P. Shaw. Pruning sub-optimal search branches using local search. In N. Jussien and F. Laburthe, editors, *Proceedings of the Fourth International Workshop on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimisation Problems (CP-AI-OR'02)*, pages 181–189, 2002.
5. D. Frost and R. Dechter. Dead-end driven learning. In *Proceedings AAAI'94*, pages 294–300, 1994.
6. I. Gent, C. Jefferson, I. Lynce, I. Miguel, P. Nightingale, B. Smith, and A. Tarim. Search in the Patience Game 'Black Hole'. Technical Report CPPod-10-2005, CPPod Research Group, 2005. Available from http://www.dcs.st-and.ac.uk/~cppod/publications/reports/.
7. I. P. Gent, T. Kelsey, S. A. Linton, I. McDonald, I. Miguel, and B. M. Smith. Conditional Symmetry Breaking. In P. van Beek, editor, *Principles and Practice of Constraint Programming - CP 2005*, LNCS. Springer, 2005.
8. B. Hnich, B. M. Smith, and T. Walsh. Dual Models of Permutation and Injection Problems. *JAIR*, 21:357–391, 2004.
9. C. Jefferson, A. Miguel, I. Miguel, and S. A. Tarim. Modelling and Solving English Peg Solitaire. *Computers & Operations Research*, 2005. In Press.
10. G. Katsirelos and F. Bacchus. Unrestricted Nogood Recording in CSP Search. In F. Rossi, editor, *Principles and Practice of Constraint Programming - CP 2003*, LNCS 2833, pages 873–877. Springer, 2003.
11. B. M. Smith. Constraint Programming in Practice: Scheduling a Rehearsal. Technical Report APES-67-2003, APES Research Group, September 2003. Available from http://www.dcs.st-and.ac.uk/~apes/apesreports.html.