

Generalised Backjumping

Peter Clark and Rob Holte
Computer Science
University of Ottawa
K1N 6N5, CANADA
{pclark,holte}@csi.uottawa.ca

Abstract

Gaschnig's backjumping algorithm has become one of the standard methods for constraint satisfaction, performing dependency-directed backtracking ('backjumping') from nodes which cannot be further expanded during tree search. In this paper, we show how this algorithm can be extended to also permit backjumping from nodes which were expanded, but whose subtrees eventually failed. We experimentally compare Gaschnig's backjumping and our extended algorithm ('generalised backjumping') on two standard constraint satisfaction problems (8-queens and zebras). We show that generalised backjumping can substantially reduce the search performed by the original algorithm, thus demonstrating a significant development of this established search technique.

1 Introduction

In 1979 Gaschnig proposed an algorithm called backjumping, a depth-first tree search procedure for solving constraint satisfaction tasks [?]. The algorithm performs dependency-directed backtracking, such that when a node in the tree cannot be expanded further, search returns directly to the nearest parent node representing a choice which caused that failure. Its simplicity makes it an effective algorithm for assisting in constraint satisfaction, and it has been extensively studied and used for that purpose since then (eg. [?, ?]). Thus as a search algorithm it is important and effective, and hence development of this technique is valuable.

In this paper, we describe how the algorithm can be extended to additionally 'backjump' from nodes which *can* be expanded, but where subsequent search results in ultimate failure of all solutions in the subtree beneath that node. As a result, backjumps can potentially be made from any node in the tree (hence 'generalised backjumping'). The resulting additional backjumps can significantly reduce the search required for constraint satisfaction, pruning out redundant search performed by the original algorithm. The additional computational overhead of generalised backjumping compared with normal backjumping is minimal, the main computational difference being set union operations replace find-maximum operations.

In Section 3 we describe generalised backjumping, and illustrate how it can significantly reduce tree search. In Section ?? we compare its performance with Gaschnig's original algorithm on two standard constraint satisfaction problems (CSPs). We show that generalised backjumping can reduce the space searched by as much as a factor of 4 in these standard tests compared with normal backjumping, illustrating the value of generalised backjumping for constraint satisfaction and contributing to the general body of knowledge concerning search.

2 Constraint Satisfaction and Backjumping

2.1 Definitions and Introduction

In this section we motivate and present Gaschnig’s backjump algorithm. Then, in Section 3, we show how it can be extended to produce ‘generalised backjumping’, a new algorithm which achieves a large reduction in required search effort.

CSPs can be formulated as a search for an assignment of *values* to *variables* such that a set of *constraints* between variables is satisfied. A *partial hypothesis* is an assignment of values to some variables, a *hypothesis* is an assignment of values to all variables, and a *solution* is a hypothesis which satisfies the constraints. An *empty hypothesis* is a partial hypothesis with no variable assignments made. The *hypothesis space* is the space of all partial and complete hypotheses. We assume throughout that constraints are pairwise (ie. between pairs of variables), but note that all the backjumping methods described easily extend to the general case.

Algorithms for constraint satisfaction fall into two categories: tree search methods (eg. backtracking, backjumping) and arc consistency methods (eg. [?]). Tree search methods consider the hypothesis space as a tree of nodes, where the root node is the empty hypothesis, and each child node represents the hypothesis of its parent but with an additional variable assignment made. The tree is searched until a solution is found. Arc consistency methods can be thought of as simplification algorithms, which transform the original problem into a simpler one which has the same solutions. Arc consistency can be seen as an efficient way of eliminating incorrect hypotheses, as opposed to directly searching for a correct hypothesis. These two styles of problem-solving can (and often are) combined: for example, arc consistency can be used to simplify the problem, followed by tree search to locate a solution within the simplified problem formulation. It is important to note that these two styles are complementary, and thus that arc consistency methods have not rendered tree search obsolete.

Gaschnig’s backjumping algorithm is a simple but effective extension of depth-first search, in which dependency-directed backtracking is performed. When a node cannot be expanded during search, the algorithm ‘jumps’ back directly to the most recent parent node responsible for that search failure, skipping over intermediate nodes.

The algorithm has several advantageous properties. First, it is simple and fast. Second, its depth-first nature means only a small memory requirement is made. Third, it can be easily combined with additional techniques such as look-ahead (providing a more detailed test of whether a partial hypothesis can lead to a solution) or heuristic search (for deciding how best to organise the hypothesis space into a tree).

2.2 Gaschnig’s Backjump Algorithm

Gaschnig’s backjumping algorithm is presented in pseudocode in Figure 1. The algorithm takes as input a partial hypothesis, attempts to assign a value to a remaining uninstantiated variable X_i (line 2), and, if successful, attempts to assign values to all remaining uninstantiated variables (line 9). We refer to the uninstantiated variable as the *current* variable, those which already have assignments as *previous* or *earlier* variables, and those which are still uninstantiated as *later* variables. A variable is *exhausted* if all possible assignments have been tried and failed.

The algorithm maintains a pointer `LevelToBackjumpTo` to the highest level variable with which any values of the current variable X_i were found to be incompatible. If *all* the

```

PROCEDURE BJ(variables  $X_1, \dots, X_{i-1}$  with already assigned values  $x_1, \dots, x_{i-1}$ ,
             new variable  $X_i$  to assign value to)
    RETURNING LevelToBackjumpTo:

1  Let LevelToBackjumpTo := 0.
2  FOR all values  $v_j$  which  $X_i$  can take
3      FIND the first variable  $X_k$  in  $X_1, \dots, X_{i-1}$  whose value  $x_k$  is
4          inconsistent with  $X_i=v_j$ 
5      IF one is found, then note  $X_i=v_j$  fails at level  $L := k$ ,
6      IF one isn't found, then
7          IF  $X_i$  is the last variable needing a value in the problem, then
8              success! Output  $\{x_1, \dots, x_{i-1}, v_j\}$  and EXIT returning 0.
9          ELSE see if the other variables  $X_{i+1}, \dots, X_k$  can be
10             instantiated too, using  $BJ(x_1, \dots, x_{i-1}, v_j, X_{i+1})$ .
11             IF they can (BJ returns 0), THEN exit with success.
12             ELSE if they can't, then BJ will return the level  $L$  to backjump to.
13             IF  $L < i$  then EXIT returning  $L$ 
14                 /* backjumping:  $X_{i+1}, \dots, X_k$  fail independent of  $X_i$  */
15     Let LevelToBackjumpTo := max(LevelToBackJumpTo, L)
16 ENDFOR
17 Return LevelToBackjumpTo.

```

Figure 1: Gaschnig's Backjump Algorithm.

values of X_i are incompatible with earlier variables, then we can immediately 'backjump' to the variable pointed to by `LevelToBacktrackTo`, skipping over intermediate variables. This is effected as follows: first the procedure (at level i) exits, returning this pointer. The previous level of recursion $i - 1$ will catch this pointer (line 12). If it points to a level earlier than this previous level, no more values for the variable at this level are tried and the procedure immediately exits (line 13), thus backjumping. Otherwise, if the pointer points to the current (or a later) variable, indicating normal backtracking should start (or continue), alternative values for $X_{i - 1}$ are tried.

3 The Generalised Backjumping Algorithm

3.1 Algorithm

By maintaining a pointer to the earliest variable responsible for failure, Gaschnig's procedure will backjump from any node where *all* values of a variable are incompatible with earlier assignments. However, if one or more values of a variable X_i are *consistent* with all more recent variables, but all subsequent assignments to later variables fail, backjumping cannot recommence should search return to this node. This is because the procedure does not know which previous variables were responsible for X_i 's subtree (hence X_i itself) failing, and thus cannot identify variables irrelevant to the subtree's failure which could then be backjumped over. In fact, the variables responsible for failure of a value of X_i (where the value is consistent with the more recent variables), is actually the set of variables responsible for all failures in subsequent assignments to later variables. Our modification to Gaschnig's algorithm is to pass back this set, and use it to inspect whether backjumping can recommence at these nodes.

As a result, if we examine the part of the tree searched by backjumping, Gaschnig's algorithm will only backjump from the leaves of this part of the tree (ie. where all variable

```

PROCEDURE GBJ(variables  $X_1, \dots, X_{i-1}$  with already assigned values  $x_1, \dots, x_{i-1}$ ,
              new variable  $X_i$  to assign value to)
    RETURNING ParentsToBlame for failure:

1  Let ParentsToBlame := {}.
2  FOR all values  $v_j$  which  $X_i$  can take
3      FIND the first variable  $X_k$  in  $X_1, \dots, X_{i-1}$  whose value  $x_k$  is
4          inconsistent with  $X_i=v_j$ 
5      IF one is found, then note  $X_i=v_j$  fails because of variables  $V_s = \{ X_k \}$ 
6      IF one isn't found, then
7          IF  $X_i$  is the last variable needing a value in the problem, then
8              success!! output  $\{x_1, \dots, x_{i-1}, v_j\}$  and EXIT returning {}.
9          ELSE see if the other variables  $X_{i+1}, \dots, X_k$  can be
10             instantiated too, using GBJ( $x_1, \dots, x_{i-1}, v_j, X_{i+1}$ ).
11             IF they can (GBJ returns {}), THEN exit with success.
12             ELSE if they can't, then GBJ will return the variables  $V_s$  responsible
13                 for failure of  $X_i=v_j$ 
14                 IF  $X_i$  isn't in those variables, then immediately EXIT
15                     returning  $V_s$ 
16                 /* backjumping:  $X_{i+1}, \dots, X_k$  fail independent of  $X_i$  */
17         Let ParentsToBlame := ParentsToBlame union  $V_s$ 
18     ENDFOR
19     Return ParentsToBlame.

```

Figure 2: The Generalised Backjumping Algorithm.

assignments were inconsistent with earlier assignments). However, by passing back the set of variables to blame for an (initially consistent) assignment, our modified algorithm will permit backjumping from any node – hence its name ‘generalised backjumping’.

The generalised backjumping algorithm is shown in Figure 2, expressed in a similar way to Gaschnig’s for ease of comparison. Rather than returning a pointer, the algorithm returns the set of variables responsible for failure. A value may fail because it is inconsistent earlier assignments (line 5) or all possible subsequent assignments (line 12-13). The overall failure is the union of these reasons (line 17). On returning to earlier levels of recursion, the algorithm will backjump if the current variable is irrelevant to the later failure which occurred (line 14).

Generalised backjumping is similar to another variant of backjumping called ‘graph-based backjumping’ [?]. Graph-based backjumping performs a similar operation of passing back parent variables to blame for failure, however these variables to blame are determined by examination of the constraint graph rather than by executing constraint tests. As a result, graph-based backjumping returns the set of all possible variables which could be associated with failure rather than those which in practice were actually involved. Thus graph-based backjumping blames variables which *could* rather than *were* responsible for failure, and hence can return an overly large set `ParentsToBlame` (hence reducing backjumps). We include this algorithm in our experimental comparison in Section ?? also.

3.2 Example

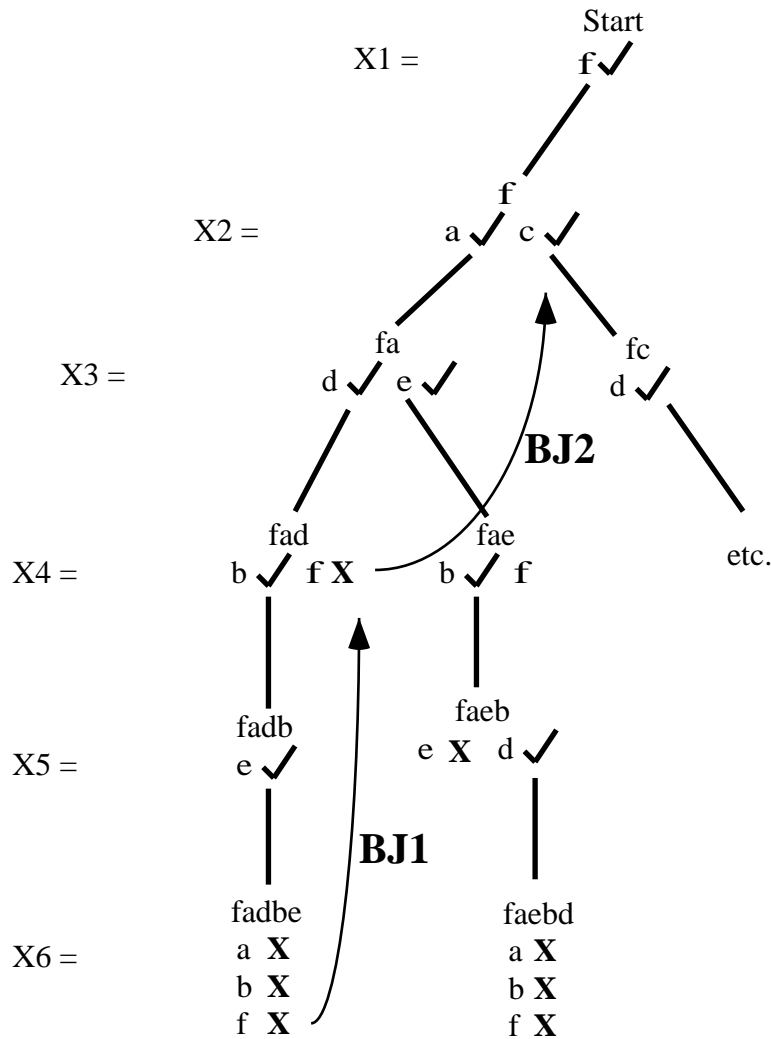
To illustrate Gaschnig’s and generalised backjumping, we provide a simple example as follows:

Variables: X1 takes values {f, a}
 X2 takes values {a, c}
 X3 takes values {d, e}
 X4 takes values {b, f}
 X5 takes values {e, d}
 X6 takes values {a, b, f}

Constraints: No two variables can have the same value.

The traces of backjumping and generalised backjumping, shown side by side, is given below. Each line show the (nodes representing the) partial hypotheses visited with brief comments (starting with a '%'). A partial hypothesis is shown as a list of values assigned to variables X1,...,Xn. (eg. 'fad' denotes X1 = f, X2 = a, X3 = d). We abbreviate the variable LevelToBackjumpTo (in Figure 1) to BJLevel, and ParentsToBlame (in Figure 2) to PTB. The exploration of the tree is also depicted in Figure ??.

BACKJUMPING:		GENERALISED BACKJUMPING:	
1	f % variable X1=f	f	% variable X1=f
2	fa % variable X2=a	fa	% etc...
3	fad % variable X3=d	fad	
4	fadb % etc...	fadb	
5	fadbe	fadbe	
6	fadbea % for X6, note BJLevel := 2	fadbea	% note Vs={X2} caused X6=a to fail
7	fadbeb % for X6, revise BJLevel := 4	fadbeb	% note Vs={X4} caused X6=b to fail
8	fadbef % for X6, BJLevel unchanged	fadbef	% note Vs={X1} caused X6=f to fail
9			
10	% X6 has exhausted its vals		% X6 has exhausted its vals
11	% backjump to level 4		% note X6 failed as PTB={X1,X2,X4}
12			% (= {X2} union {X4} union {X1})
13			% backjump to level 4
14	% for X4, note BJLevel := 4		% note Vs={X1,X2} -> X4=b to fail
15	fadf % for X4, BJLevel unchanged	fadf	% note Vs={X1} caused X4=f to fail
16			
17	% X4 has exhausted its vals		% X4 has exhausted its vals
18			% note X4 failed as PTB={X1,X2}
19			% (= {X1,X2} union {X1})
20	% backjump to level 3		% backjump to level 2
21	fae		
22	faeb		
23	faebe		
24	faebd		
25	faebda		
26	faebdb		
27	faebdf % backjump to level 4		
28	faef		
29			% note {X1} caused X2=a to fail
30	fc	fc	
31	fcd	fcd	
32	fcdb	fcdb	
33	fcdbbe	fcdbbe	
34	fcdbea % Final solution	fcdbea	% Final solution
TOTAL: 22 NODES VISITED		TOTAL: 14 NODES VISITED	



✓ = success, X = failure.

= search avoided by generalised backjumping. This line of search is bound to fail as no combination of X4's and X6's values are compatible with X1=f and X2=a.

BJ1: Backjump! Both backjump algorithms backjump here.

BJ2: Backjump! By recording why X4=b ultimately failed, generalised backjumping can perform a second backjump as shown, whereas Gaschnig's algorithm resorts to normal backtracking.

This figure depicts the search described in Section 3.2. Each node shows the earlier variable assignments made, with the different assignments tried for current variable below it. Where an assignment is consistent with earlier variables (shown by a tick), the search continues.

Figure 3: Search of the hypothesis space by backjumping algorithms.

	Naive	Backjumping method:		
	backtracking	graph-based	Gaschnig	generalised
8-queens				
Backtracks	105	105	66	66
Backjumps	0	0	15	15
Nodes searched	876	876	765	765
Zebras (reprn 1)				
Backtracks	466	338	246	49
Backjumps	0	9	37	20
Nodes searched	2405	1902	1844	629
Zebras (reprn 2)				
Backtracks	8669	8669	6302	568
Backjumps	0	0	625	531
Nodes searched	43420	43420	37571	8774

Table 1: Total nodes searched by different tree search algorithms

As shown in this trace, both algorithms backjump from variable **X6** to **X4** (line 8), when all assignments to **X6** fail. Backjumping can occur here as variable **X5** was irrelevant to this failure.

However, when the other value of **X4** also fails (line 15), Gaschnig’s algorithm will proceed with normal backtracking to find a new value for **X3** (line 21). However generalised backjumping, by passing back the set of variables responsible for $X4 = f$ and $X4 = b$, now has information that **X3** is irrelevant and can thus perform a second backjump to **X2**. This reduces the search performed, prunes out the redundant search performed by the former algorithm.

Note that the ability to perform second backjumps is highly valuable, as the size of search space pruned out by backjumping grows exponentially with the earliness of the variable backjumped over.

4 Evaluation

To compare these different backjumping methods, we evaluated them on two constraint satisfaction problems (CSPs), namely the standard 8-queens problem and two different representations of the ‘zebras’ problem (described in the Appendix). We also include naive (chronological) backtracking and graph-based backjumping in the comparison.

To evaluate each algorithm, we count the number of nodes visited during search. Also for interest we include the number of backjumps and normal backtracks performed by the different algorithms. The results are shown in Table ??.

The most significant feature of the results is the large decrease in search that generalised backjumping can achieve over other backjumping methods. This illustrates that Gaschnig’s backjumping algorithm can perform substantial unnecessary search, and that generalised backjumping can significantly reduce this search. In both representations of the ‘zebras’ problem, all other methods searched more than three times the number of nodes compared

with generalised backjumping.

We also note that the variation in computational overhead for all these algorithms (ie. that additional to constraint testing) is minimal. Generalised backjumping's additional computation is to replace Gaschnig's 'max' operation (line 15, Figure 1) with a set union operation (line 17, Figure 2). Graph-based backjumping does neither of these, instead simply looking up which variables to blame from the constraint graph, but this small speed gain is insignificant compared with the additional constraint testing resulting from the less constrained search.

We also comment on two other features of the results. First, the benefits to be gained from backjumping are significantly less in the 8-queens CSP than the zebras CSP. This is because the variables representing the positions of the 8 queens have many constraints between them (the constraint graph is highly connected). For this CSP, almost all variables are involved in any failure and thus the potential for finding irrelevant intervening variables, hence backjumping, is reduced. For the zebra problem, the constraint graph is sparser and hence more situations arise when irrelevant variable assignments stand between the current point of failure and the most recent earlier assignment relevant to the failure.

Second, generalised backjumping performs less backjumps than normal backjumping in the zebra problem. The reason the number of nodes searched is so small is that some of these backjumps are from earlier nodes where one or more assignment was consistent with its earlier variables, as described in Section 3.1. These backjumps prune out large parts of the space which normal backjumping would otherwise explore, thus reducing the number of nodes searched, backjumps and backtracks required for the remainder of the search.

5 Conclusion

We have described how Gaschnig's backjumping algorithm can be extended to permit backjumps from any node visited during search, rather than only those which cannot be further expanded. As a result, large parts of the search space explored by the original algorithm can be pruned out, significantly reducing search while incurring minimal extra computational overhead. We have demonstrated that large reductions in search can be achieved. This work is thus an important development of this standard search algorithm, which can thus enhance constraint satisfaction either when used on its own or in conjunction with other CSP methods such as look-ahead or arc consistency.

References

- [1] R. Dechter. Enhancement schemes for constraint processing: Backjumping, learning and cutset decomposition. *Artificial Intelligence*, 41:273–312, 1990.
- [2] J. Gaschnig. *Performance Measurement and Analysis of Certain Search Algorithms*. PhD thesis, Carnegie-Mellon Univ., PA, 1979.
- [3] A. K. Mackworth. Consistency in networks of relations. *Artificial Intelligence*, 8:99–118, 1977.
- [4] B. A. Nadel. Constraint satisfaction algorithms. *Computational Intelligence*, 5:188–224, 1989.

Appendix: The Zebra Problem

- There are five houses, each of a different colour and inhabited by men of different nationalities, with different pets, drinks and cigarettes.
- The Englishman lives in the red house.
- The Spaniard owns a dog.
- Coffee is drunk in the green house.
- The Ukrainian drinks tea.
- The green house is immediately to the right of the ivory house.
- The Gold smoker owns snails.
- Kools are smoked in the yellow house.
- Milk is drunk in the middle house.
- The Norwegian lives in the first house.
- The Chesterfield smoker lives next to the fox owner.
- Kools are smoked next to the house where the horse is kept.
- The Lucky-Strike smoker drinks orange juice.
- The Japanese smokes Parliament.
- The Norwegian lives next to the blue house.

The two alternative representations of the problem used are as follows:

1. In the first representation we use twenty five variables, all of which can take a value from 1,...,5. Each variable represents the number of the house which is red (variable 1), blue (variable 2),..., where the Chesterfields are smoked (variable 25).
2. In the second representation, we again use twenty five variables, the first five for the colours of houses 1,...,5, (ie. each takes one of the values red,blue, green, ivory, yellow), the second five for the nationalities, and the third, fourth and fifth five for the pets, cigarettes and drinks respectively.