Constraint Satisfaction Problems

Look-Back

Albert-Ludwigs-Universität Freiburg

UNI FREIBURG

Stefan Wölfl, **Christian Becker-Asano**, and **Bernhard Nebel**

November 26, 2014

Look-back techniques

- Look-ahead techniques reduce the size of the searched part of the state space by excluding partial assignments from consideration if they provably lead to inconsistencies.
- This is a form of forward analysis: We avoid assignments which must lead to dead ends in the future.
- Look-back techniques use a complementary approach: We avoid assignments which led to dead ends in the past.

Types of look-back techniques

We will consider two classes of look-back techniques:

- Backjumping: Upon encountering a dead end, do not always return to the parent in the search tree, but possibly to an earlier ancestor.
- Nogood learning: Upon encountering a dead end, record a new constraint to detect this type of dead end earlier in the future.

Nogood learning is commonly used when solving propositional logic satisfiability problems for CNF formulae. In this context, it is known as clause learning.

Conflict Sets

Conventions

- Throughout the chapter, we assume a fixed variable ordering $v_1, \dots, v_n$.
- Partial assignments $a = \{v_1 \mapsto d_1, \dots, v_i \mapsto d_i\}$ for $i \in \{0, \dots, n\}$ are abbreviated as tuples: $(d_1, \dots, d_i)$.

Dead ends

Recall:

Definition (dead end)

A dead end in an ordered search space is a state which is not a goal state and in which no operator is applicable to the next variable (in the fixed variable ordering).

In the context of look-back methods, such dead ends are called leaf dead ends:

Definition (leaf dead end)

A leaf dead end is a partial solution $(d_1, \ldots, d_i)$ such that $(d_1, \ldots, d_{i+1})$ is inconsistent for any possible value of v_{i+1}. Variable v_{i+1} is called the leaf dead end variable for the leaf dead end.

Conflict sets

Definition (conflict set)

Let a be a partial solution (on any set of variables), and let v_j be a variable for which a is not defined.

We say that a is a conflict set of v_j, (or: a is in conflict with v_j) if no extensions of a of the form $a \cup \{(v_j, d_j)\}$ is consistent.

If moreover a contains no subtuple which is in conflict with v_j, it is called a minimal conflict set of v_j.

$\rightsquigarrow$ A leaf dead end is a conflict set of the leaf dead end variable, but not every conflict set is a leaf dead end.

Nogoods and internal dead ends

Definition (nogood)

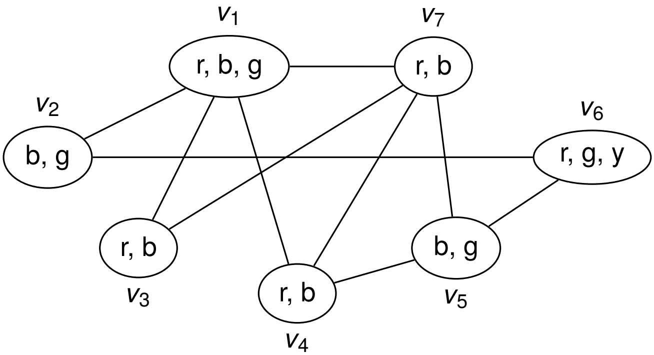
A partial solution that cannot be extended to a solution of the network is called a nogood.

A nogood is minimal if it contains no nogood subassignment.

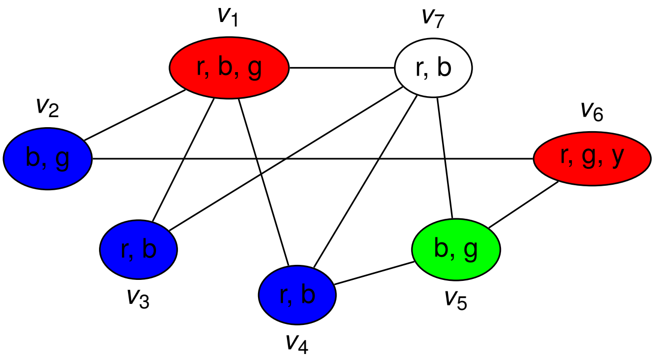
A nogood is called an internal dead end (in the fixed variable ordering) if it is defined on the first *i* variables, i.e., on $\{v_1, \ldots, v_i\}$ and it is not a leaf dead end. In that case, v_{i+1} is called the internal dead end variable.

Conflict sets are nogoods, but not all nogoods are conflict sets.

Example: Leaf dead ends, conflict sets, nogoods

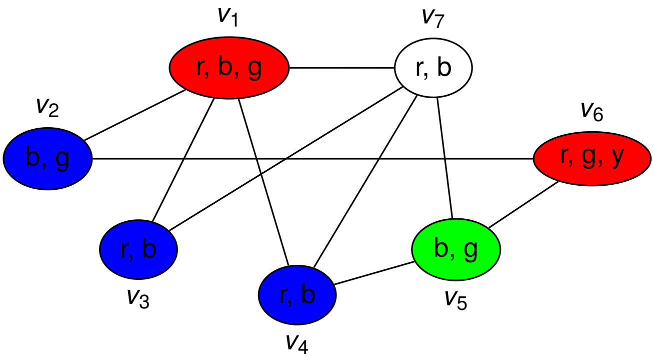


Example: Leaf dead end



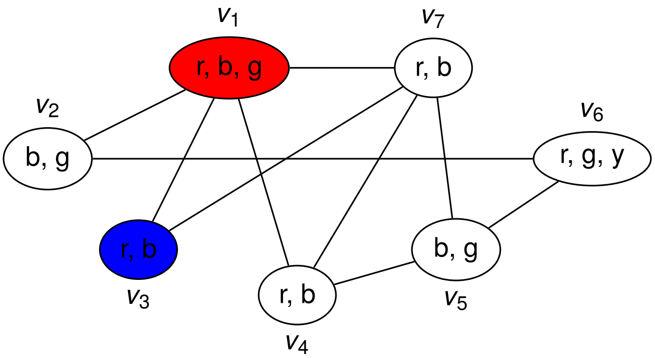
$\rightsquigarrow$ a leaf dead end with leaf dead-end variable v_7

Example: Conflict set



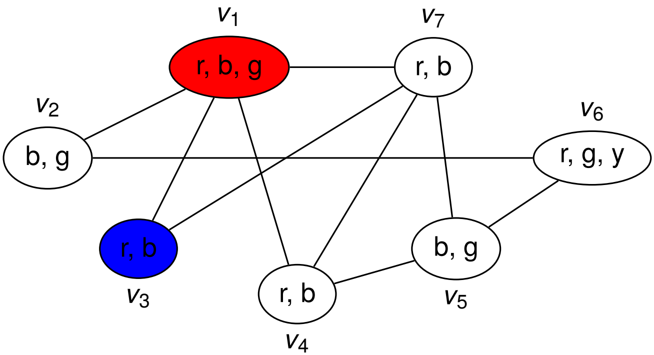
$\rightsquigarrow$ a conflict set of v_7, but not minimal

Example: Conflict set



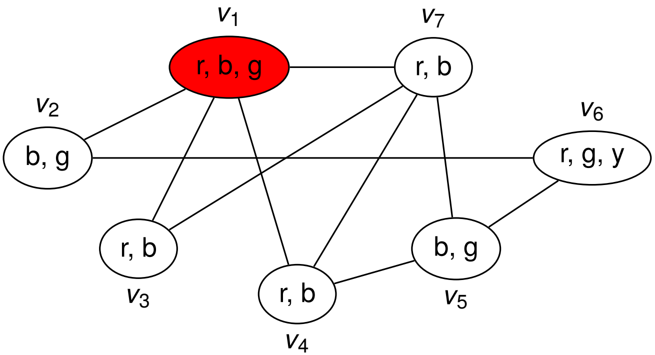
$\rightsquigarrow$ a minimal conflict set of v_7

Example: Nogood



⇝ a nogood, but not a minimal one

Example: Nogood



⇝ a minimal nogood (also an internal dead end)

Safe jumps

Definition (safe jump)

Let $a = (d_1, \ldots, d_i)$ be a (leaf or internal) dead end.
We say that v_j with $j \in \{1, \ldots, i\}$ is safe (or: a safe jump) relative to a if $(d_1, \ldots, d_j)$ is a nogood.

⇝ If v_j is safe (where $j < i$), we can backtrack several times and assign a new value to v_j next.

Backjumping

Backjumping

A backjumping algorithm is a modification of backtracking that may back up several layers in the search tree upon detecting an assignment that cannot be extended to a solution.

We study three variations:

- Gaschnig's backjumping
- Graph-based backjumping
- Conflict-directed backjumping

Gaschnig's backjumping

We first introduce Gaschnig's backjumping, which is one of the simplest backjumping algorithms.
It only backs up multiple layers at leaf dead ends.

Definition (culprit variable)

Let $a = (d_1, \ldots, d_i)$ be a leaf dead end.
The culprit index relative to a is

$$culp(a) := \min\{\, j \in \mathbb{N}_1 \mid (d_1, \ldots, d_j) \text{ conflicts with } v_{i+1} \,\}.$$

$v_{culp(a)}$ is called the culprit variable.

Gaschnig's backjumping

When detecting a leaf dead end a, jump back to $v_{culp(a)}$.

Gaschnig's backjumping

We first introduce Gaschnig's backjumping, which is one of the simplest backjumping algorithms.
It only backs up multiple layers at leaf dead ends.

Definition (culprit variable)

Let $a = (d_1, \dots, d_i)$ be a leaf dead end.
The culprit index relative to a is

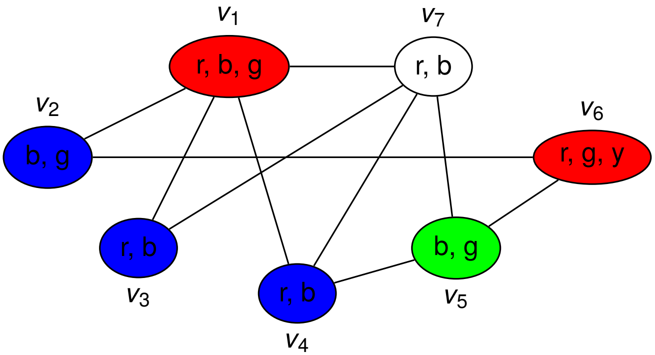
$$culp(a) := \min\{\, j \in \mathbb{N}_1 \mid (d_1, \dots, d_j) \text{ conflicts with } v_{i+1} \,\}.$$

$v_{culp(a)}$ is called the culprit variable.

Gaschnig's backjumping

When detecting a leaf dead end a, jump back to $v_{culp(a)}$.

Gaschnig's backjumping: Example



Remarks on Gaschnig's backjumping

- Gaschnig's backjumping was historically one of the first backjumping techniques.
- It clearly performs only safe jumps.
- It also performs maximal jumps in the sense that backing up further than Gaschnig's backjumping at leaf dead ends can lead to missing (potentially all) solutions.
- The algorithm is attractive because it is easy to implement efficiently (cf. implementation project).
- However, it is not very powerful: It expands strictly more states than look-ahead search with forward checking.
- One serious limitation is that it only jumps at leaf dead ends. The next backjumping technique will remedy this.

Remarks on Gaschnig's backjumping

- Gaschnig's backjumping was historically one of the first backjumping techniques.
- It clearly performs only safe jumps.
- It also performs maximal jumps in the sense that backing up further than Gaschnig's backjumping at leaf dead ends can lead to missing (potentially all) solutions.
- The algorithm is attractive because it is easy to implement efficiently (cf. implementation project).
- However, it is not very powerful: It expands strictly more states than look-ahead search with forward checking.
- One serious limitation is that it only jumps at leaf dead ends. The next backjumping technique will remedy this.

Remarks on Gaschnig's backjumping

- Gaschnig's backjumping was historically one of the first backjumping techniques.
- It clearly performs only safe jumps.
- It also performs maximal jumps in the sense that backing up further than Gaschnig's backjumping at leaf dead ends can lead to missing (potentially all) solutions.
- The algorithm is attractive because it is easy to implement efficiently (cf. implementation project).
- However, it is not very powerful: It expands strictly more states than look-ahead search with forward checking.
- One serious limitation is that it only jumps at leaf dead ends. The next backjumping technique will remedy this.

Remarks on Gaschnig's backjumping

- Gaschnig's backjumping was historically one of the first backjumping techniques.
- It clearly performs only safe jumps.
- It also performs maximal jumps in the sense that backing up further than Gaschnig's backjumping at leaf dead ends can lead to missing (potentially all) solutions.
- The algorithm is attractive because it is easy to implement efficiently (cf. implementation project).
- However, it is not very powerful: It expands strictly more states than look-ahead search with forward checking.
- One serious limitation is that it only jumps at leaf dead ends. The next backjumping technique will remedy this.

Remarks on Gaschnig's backjumping

- Gaschnig's backjumping was historically one of the first backjumping techniques.
- It clearly performs only safe jumps.
- It also performs maximal jumps in the sense that backing up further than Gaschnig's backjumping at leaf dead ends can lead to missing (potentially all) solutions.
- The algorithm is attractive because it is easy to implement efficiently (cf. implementation project).
- However, it is not very powerful: It expands strictly more states than look-ahead search with forward checking.
- One serious limitation is that it only jumps at leaf dead ends. The next backjumping technique will remedy this.

Remarks on Gaschnig's backjumping

- Gaschnig's backjumping was historically one of the first backjumping techniques.
- It clearly performs only safe jumps.
- It also performs maximal jumps in the sense that backing up further than Gaschnig's backjumping at leaf dead ends can lead to missing (potentially all) solutions.
- The algorithm is attractive because it is easy to implement efficiently (cf. implementation project).
- However, it is not very powerful: It expands strictly more states than look-ahead search with forward checking.
- One serious limitation is that it only jumps at leaf dead ends. The next backjumping technique will remedy this.

Graph-based backjumping

- Graph-based backjumping can also jump back at internal dead ends.
- Unlike Gaschnig's backjumping, it does not use information about the values assigned to the variables in the current state when backing up.
- Instead, it only uses information about the variables themselves, derived from the constraint graph.

Parents

Reminder:

Definition (parents)

The parents of v_i are those variables v_j with $j < i$ for which the edge $\{v_i, v_j\}$ occurs in the primal constraint graph.

Definition (parents)

Let v_i be a variable with at least one parent.
The latest parent of v_i, in symbols $par(v_i)$, is the parent v_j for which j is maximal.

Basic idea: Jump back to the latest parent.

Jumping back to the latest parent

Theorem

Let $a = (d_1, \ldots, d_{i-1})$ be a leaf dead end with dead-end variable v_i. Then $par(v_i)$ is a safe jump for a.

Proof.

Let j_0 be the index of $par(v_i)$. We need to show that $(d_1, \ldots, d_{j_0})$ is a nogood. We prove that the (locally consistent) assignment $(d_1, \ldots, d_{j_0})$ is in conflict with v_i.

Assume not. Then there exists $v_i \mapsto d$ such that

(a) $(d_1, \ldots, d_{j_0}, d)$ is locally consistent.

Moreover, we know:

(b) $(d_1, \ldots, d_{i-1})$ is locally consistent (it's a leaf dead end), but

(c) $(d_1, \ldots, d_{i-1}, d)$ is not (thus: $j_0 < i - 1$).

Jumping back to the latest parent

Proof (cont'd).

By (c) there exists a constraint C with scope $s \subseteq \{v_1, \ldots, v_i\}$ such that $(d_1, \ldots, d_{i-1}, d) \not\models C$.

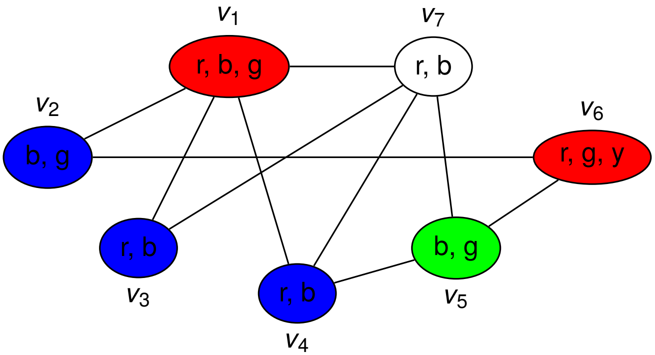
If we assume $v_i \notin s$, we would get $(d_1, \ldots, d_{i-1}) \not\models C$; a contradiction to (b). Thus $v_i \in s$.

If we assume $s \subseteq \{v_1, \ldots, v_{j_0}, v_i\}$, it would follow $(d_1, \ldots, d_{j_0}, d) \not\models C$; in contradiction to (a).

Hence, there must be a variable v_j with $j_0 < j < i$ such that $v_j \in s$. This is a contradiction to the fact that v_{j_0} is the latest parent.

□

Gaschnig vs graph-based



Comparison to Gaschnig's Backjumping

- Jumping back to the latest parent of a leaf dead end is strictly worse than Gaschnig's Backjumping: it never jumps further, and it sometimes jumps less far.
- However, the idea can be extended to jumping from internal dead ends.

First idea: When encountering an internal dead end, jump back to the latest parent of the internal dead-end variable.

Unfortunately, this is not safe.

Comparison to Gaschnig's Backjumping

- Jumping back to the latest parent of a leaf dead end is strictly worse than Gaschnig's Backjumping: it never jumps further, and it sometimes jumps less far.
- However, the idea can be extended to jumping from internal dead ends.

First idea: When encountering an internal dead end, jump back to the latest parent of the internal dead-end variable.

Unfortunately, this is not safe.

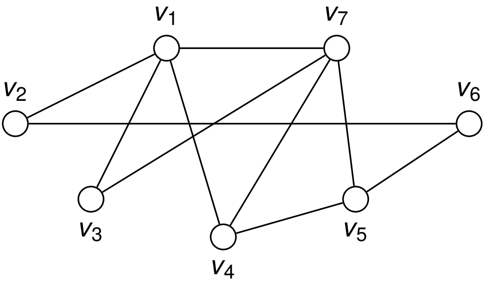
Comparison to Gaschnig's Backjumping

- Jumping back to the latest parent of a leaf dead end is strictly worse than Gaschnig's Backjumping: it never jumps further, and it sometimes jumps less far.
- However, the idea can be extended to jumping from internal dead ends.

First idea: When encountering an internal dead end, jump back to the latest parent of the internal dead-end variable.

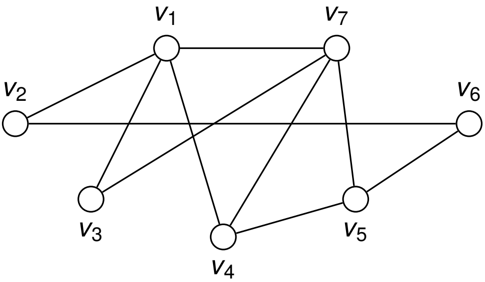
Unfortunately, this is not safe.

Backjumping at Internal Dead Ends: Example



- Scenario 1: Enter v_4 and encounter a leaf dead end with variable v_5. Jumping back to v_4, there are no further values for v_4. It is then safe to backtrack to v_1.
- Scenario 2: Now encounter a leaf dead end with variable v_7. Jump back to v_5 and then to v_4. Is it still safe to jump back to v_1 if there are no further values for v_4?

Backjumping at Internal Dead Ends: Example



- Scenario 1: Enter v_4 and encounter a leaf dead end with variable v_5. Jumping back to v_4, there are no further values for v_4. It is then safe to backtrack to v_1.
- Scenario 2: Now encounter a leaf dead end with variable v_7. Jump back to v_5 and then to v_4. Is it still safe to jump back to v_1 if there are no further values for v_4?

Sessions

Definition (invisit, session)

We say that the backtracking algorithm **invisits** variable v_i when it attempts to extend the assignment $a = (d_1, \ldots, d_{i-1})$ to v_i.

The current **session** of v_i starts when v_i is invisited and ends after all possible assignments to v_i have been tried, i.e., when the backtracking algorithm backs up to variable v_{i-1} or earlier.

Note: A session of v_i corresponds to a recursive invocation of the backtracking procedure where values are assigned to v_i.

Relevant variables

Definition (relevant variables)

The relevant variables of the current session of v_i, in symbols $rel(v_i)$, are computed as follows:

- When v_i is invisited, set $rel(v_i) := \{v_i\}$.
- When v_i is reached by backing up from a later variable v_j, set $rel(v_i) := rel(v_i) \cup rel(v_j)$.

Graph-based backjumping: Algorithm

Graph-based backjumping

When detecting the (leaf or internal) dead end *a* with dead-end variable v_i, jump back to the latest parent of any variable in $rel(v_i)$ which is earlier than v_i.

Theorem (Soundness)

Graph-based backjumping only performs safe jumps.

Proof:

⇝ exercises □

Graph-based backjumping: Algorithm

Graph-based backjumping

When detecting the (leaf or internal) dead end *a* with dead-end variable v_i, jump back to the latest parent of any variable in $rel(v_i)$ which is earlier than v_i.

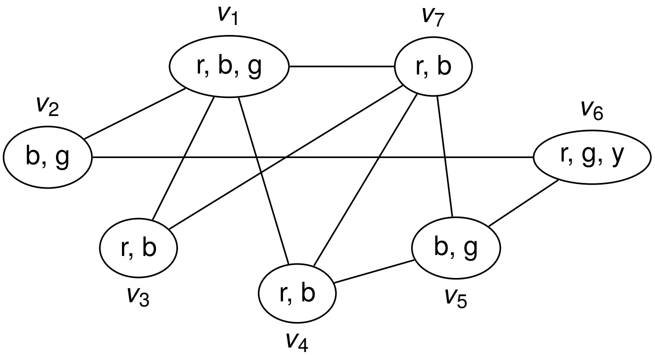
Theorem (Soundness)

Graph-based backjumping only performs safe jumps.

Proof:

⇝ exercises □

Graph-based backjumping: Example



Conflict-directed backjumping

- Gaschnig's backjumping exploits the information about a particular minimal prefix conflict set to jump further from leaf dead ends.
- Graph-based backjumping collects and integrates information from all dead ends in the current session to also jump back at internal dead ends.
- These two ideas can be combined to obtain the conflict-directed backjumping algorithm, which is better (avoids more states) than either of the two previous backjumping styles.

Constraint ordering

Definition (earlier constraint)

Let $v_1, \ldots, v_n$ be a variable ordering, and let Q and R be two constraints. We say that Q is earlier than R according to the ordering, in symbols $Q \prec R$ if

- $scope(Q) \subset scope(R)$, or
- $scope(Q) \not\subseteq scope(R)$ and $scope(R) \not\subseteq scope(Q)$ and the latest variable in $scope(Q) \setminus scope(R)$ precedes the latest variable in $scope(R) \setminus scope(Q)$.

If we assume that any two constraints have different scopes, this defines a total order on constraints.

Greedy conflict sets

UNI FREIBURG

Definition (greedy conflict set)

Let a be a (leaf or internal) dead end with dead-end variable v_i. For all $d \in D_i$, define V_d as follows:

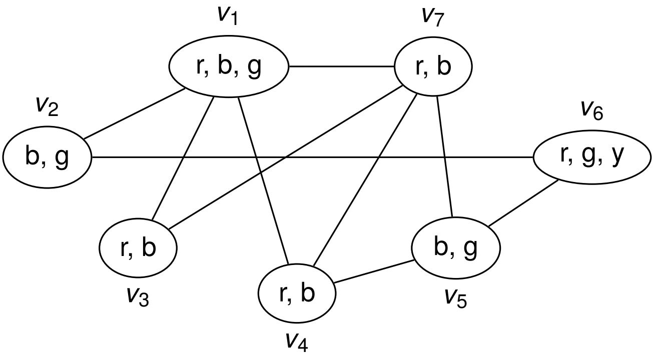
- If $a \cup \{(v_i, d)\}$ is inconsistent, let V_d be the scope of the earliest constraint which is not satisfied by $a \cup \{(v_i, d)\}$.
- Otherwise, $V_d := \emptyset$.

The greedy conflict variable set of a, in symbols $gcv(a)$, is defined as $gcv(a) := \bigcup_{d \in D_i} (V_d \setminus \{v_i\})$.

The greedy conflict set of a, in symbols $gc(a)$, is defined as $gc(a) := \{v \mapsto a(v) \mid v \in \text{gcv}(a)\}$.

In other words, $gc(a)$ is a restricted to the greedy conflict variable set.

Gaschnig vs graph-based



Greedy conflict sets are conflict sets

Theorem

Let a be a leaf dead end with dead-end variable v_i.
Then $gc(a)$ is a conflict set of v_i.

Proof.

Since a is a leaf dead end, it is a partial solution. Moreover, $gc(a)$ is a sub-assignment of a, and it is not defined for v_i.
We show that no assignment $gc(a) \cup \{(v_i, d)\}$ is consistent.
Consider an arbitrary value $d \in D_i$. In a leaf dead-end, there must be a constraint R_d with scope V_d which is not satisfied by $a \cup \{(v_i, d)\}$. Then $gcv(a)$ includes all variables in $V_d \setminus \{v_i\}$, and thus $gc(a)$ is defined and equal to a on these variables. As $a \cup \{(v_i, d)\}$ does not satisfy R_d, $gc(a) \cup \{(v_i, d)\}$ does not satisfy R_d either. Thus, $gc(a)$ cannot be consistently extended to v_i and hence is a conflict set for v_i. □

Greedy conflict sets are conflict sets

Theorem

Let a be a leaf dead end with dead-end variable v_i.
Then $gc(a)$ is a conflict set of v_i.

Proof.

Since a is a leaf dead end, it is a partial solution. Moreover, $gc(a)$ is a sub-assignment of a, and it is not defined for v_i.
We show that no assignment $gc(a) \cup \{(v_i, d)\}$ is consistent.
Consider an arbitrary value $d \in D_i$. In a leaf dead-end, there must be a constraint R_d with scope V_d which is not satisfied by $a \cup \{(v_i, d)\}$. Then $gcv(a)$ includes all variables in $V_d \setminus \{v_i\}$, and thus $gc(a)$ is defined and equal to a on these variables. As $a \cup \{(v_i, d)\}$ does not satisfy R_d, $gc(a) \cup \{(v_i, d)\}$ does not satisfy R_d either. Thus, $gc(a)$ cannot be consistently extended to v_i and hence is a conflict set for v_i. □

Greedy conflict sets are conflict sets

Theorem

Let a be a leaf dead end with dead-end variable v_i.
Then $gc(a)$ is a conflict set of v_i.

Proof.

Since a is a leaf dead end, it is a partial solution. Moreover, $gc(a)$ is a sub-assignment of a, and it is not defined for v_i.
We show that no assignment $gc(a) \cup \{(v_i, d)\}$ is consistent.
Consider an arbitrary value $d \in D_i$. In a leaf dead-end, there must be a constraint R_d with scope V_d which is not satisfied by $a \cup \{(v_i, d)\}$. Then $gcv(a)$ includes all variables in $V_d \setminus \{v_i\}$, and thus $gc(a)$ is defined and equal to a on these variables. As $a \cup \{(v_i, d)\}$ does not satisfy R_d, $gc(a) \cup \{(v_i, d)\}$ does not satisfy R_d either. Thus, $gc(a)$ cannot be consistently extended to v_i and hence is a conflict set for v_i. □

Minimality of greedy conflict sets

- Dechter calls *gc*(*a*) the earliest minimal conflict set of *a*.
- However, it is not always a minimal conflict set and not always the earliest conflict set that is a subassignment of *a*, so we avoid this terminology.

Note: The greedy conflict set is only a conflict set for leaf dead ends!

Greedy conflict sets vs. Gaschnig's backjumping

Reminder:

- Gaschnig's backjumping jumps back to $v_{culp(a)}$, where $culp(a) := \min\{ j \in \mathbb{N}_1 \mid (a_1, \ldots, a_j) \text{ conflicts with } v \}$

Observations:

- For the greedy conflict variable set, the latest variable in *gcv*(*a*) always equals *culp*(*a*).
- Thus, jumping from leaf dead ends to the latest variable in *gcv*(*a*) is the same as Gaschnig's backjumping.

Greedy conflict sets vs. graph-based backjumping

Observations:

- All variables in *gcv*(*a*) are parents of the leaf dead end variable of *a*.

Idea:

- Instead of considering all parents of relevant dead-end variables (as in graph-based backjumping), consider all greedy conflict sets of relevant dead ends.
- Using this scheme, jumping from internal dead ends jumps at least as far as graph-based backjumping.

Definition (jump-back set)

The jump-back set of a partial solution $a = (d_1, \ldots, d_{i-1})$, in symbols J_a, is computed as follows:

- When the current session of v_i starts, set $J_a := \emptyset$.
- When v_i is reached by backing up from a dead end $a' = (d_1, \ldots, d_j)$ with $j > i$, set $J_a := J_a \cup (J_{a'} \setminus \{v_i\})$.
- When the current session of v_i ends and a is a (leaf or internal) dead end, set $J_a := gcv(a) \cup J_a$.

Conflict-directed backjumping: Algorithm

Conflict-directed backjumping

When detecting the (leaf or internal) dead end a with dead-end variable v_i, jump back to the latest variable in J_a that is earlier than v_i.

Theorem (Soundness)

Conflict-directed backjumping only performs safe jumps.

Proof idea.

Combine the proofs for Gaschnig's backjumping and graph-based backjumping. □

Conflict-directed backjumping: Algorithm

Conflict-directed backjumping

When detecting the (leaf or internal) dead end a with dead-end variable v_i, jump back to the latest variable in J_a that is earlier than v_i.

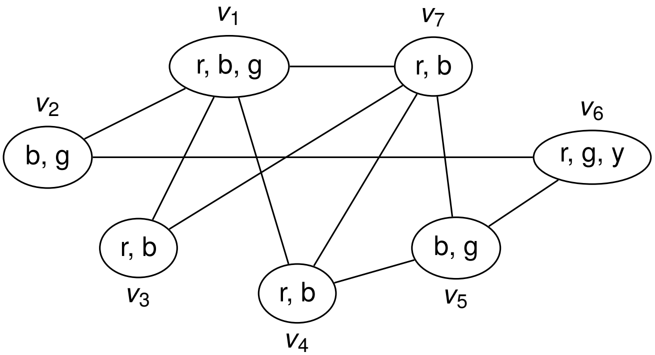
Theorem (Soundness)

Conflict-directed backjumping only performs safe jumps.

Proof idea.

Combine the proofs for Gaschnig's backjumping and graph-based backjumping. □

Conflict-directed backjumping: Example



No-Good Learning

Nogood learning

- Backjumping can significantly reduce the search effort by skipping over irrelevant choice points.
- However, thrashing is still possible: essentially the same nogood can be “rediscovered” over and over in different parts of the search tree.
- To alleviate this problem, we can make use of nogood learning or constraint recording techniques.

Adding nogood learning

Adding nogood learning to an existing (backtracking, look-ahead, backjumping, ...) algorithm is simple:

nogood learning

When the algorithm backtracks (or jumps back), determine a conflict set and add a constraint to the network that rules out this conflict set.

Variations of nogood learning

There are many variations:

- How to determine the nogood?
  - Determine one which is easy to generate, but not necessarily minimal $\rightsquigarrow$ shallow learning.
  - Determine one which is minimal, or even all minimal ones derivable from the current dead end $\rightsquigarrow$ deep learning
- Which nogoods to store?
  - Store all constraints.
  - Store only small nogoods (constraints with arity $\leq c$) $\rightsquigarrow$ bounded learning
- How long to store nogoods?
  - Store forever.
  - Discard once they differ from the current state in more than c variables $\rightsquigarrow$ relevance-bounded learning

Variations of nogood learning

There are many variations:

- How to determine the nogood?
  - Determine one which is easy to generate, but not necessarily minimal ⇝ shallow learning.
  - Determine one which is minimal, or even all minimal ones derivable from the current dead end ⇝ deep learning
- Which nogoods to store?
  - Store all constraints.
  - Store only small nogoods (constraints with arity $\leq c$) ⇝ bounded learning
- How long to store nogoods?
  - Store forever.
  - Discard once they differ from the current state in more than c variables ⇝ relevance-bounded learning

Variations of nogood learning

There are many variations:

- How to determine the nogood?
  - Determine one which is easy to generate, but not necessarily minimal $\rightsquigarrow$ shallow learning.
  - Determine one which is minimal, or even all minimal ones derivable from the current dead end $\rightsquigarrow$ deep learning
- Which nogoods to store?
  - Store all constraints.
  - Store only small nogoods (constraints with arity $\leq c$) $\rightsquigarrow$ bounded learning
- How long to store nogoods?
  - Store forever.
  - Discard once they differ from the current state in more than c variables $\rightsquigarrow$ relevance-bounded learning

Nogood learning: Issues

When performing nogood learning, there is a need to strike a good compromise between:

- pruning power:
  more constraints lead to fewer explored states
- constraint processing overhead:
  learning many constraints increases the satisfaction tests for every search node
- learning overhead:
  expensive computations of nogoods may outweigh pruning benefits
- space overhead:
  storing all nogoods eliminates the space efficiency of backtracking-style algorithms

Graph-based learning

Graph-based learning

Augment graph-based backjumping by applying the following learning rule when jumping back from an internal or leaf dead-end a with dead-end variable v_i:

- Let $V(a)$ be the set of parents of some variable in the relevant dead-end variable set $rel(v_i)$.
- Learn the nogood $\{(v, a(v)) \mid v \in V(a) \text{ and } v \prec v_i\}$.

Conflict-directed backjump learning

Conflict-directed backjump learning

Augment conflict-directed backjumping by applying the following learning rule when jumping back from an internal or leaf dead-end *a* with dead-end variable v_i:

- Learn the nogood $\{(v, a(v)) \mid v \in gcv(a) \text{ and } v \prec v_i\}$.

Nonsystematic randomized backtrack learning

- Learning algorithms are not limited to minor variations of the common systematic backtracking algorithms.
- One example of a very different algorithm is nonsystematic randomized backtrack learning:
  - Use backtracking with random variable and value orders.
  - At each dead end, learn a new conflict set.
  - After a certain number of dead ends, restart (remembering the newly learned constraints).
  - Terminate upon solution or when ∅ becomes a dead end.

Completeness:

- Each newly learned constraint reduces the number of states in the state space by at least 1.
- Thus, eventually either the empty assignment will be a dead end, or the search space will become backtrack-free.

Nonsystematic randomized backtrack learning

- Learning algorithms are not limited to minor variations of the common systematic backtracking algorithms.
- One example of a very different algorithm is nonsystematic randomized backtrack learning:
  - Use backtracking with random variable and value orders.
  - At each dead end, learn a new conflict set.
  - After a certain number of dead ends, restart (remembering the newly learned constraints).
  - Terminate upon solution or when $\emptyset$ becomes a dead end.

Completeness:

- Each newly learned constraint reduces the number of states in the state space by at least 1.
- Thus, eventually either the empty assignment will be a dead end, or the search space will become backtrack-free.

Nonsystematic randomized backtrack learning

- Learning algorithms are not limited to minor variations of the common systematic backtracking algorithms.
- One example of a very different algorithm is nonsystematic randomized backtrack learning:
  - Use backtracking with random variable and value orders.
  - At each dead end, learn a new conflict set.
  - After a certain number of dead ends, restart (remembering the newly learned constraints).
  - Terminate upon solution or when $\emptyset$ becomes a dead end.

Completeness:

- Each newly learned constraint reduces the number of states in the state space by at least 1.
- Thus, eventually either the empty assignment will be a dead end, or the search space will become backtrack-free.

Literature

Literature

Rina Dechter.
Constraint Processing,
Chapter 6, Morgan Kaufmann, 2003