Backtracking search: look-back

Chapter 6
Look-back: backjumping

- Backjumping: Go back to the most recently culprit.
- Learning: constraint-recording, no-good recording.

Figure 6.1: A modified coloring problem.
Backjumping, conflict sets

- \((X1=r,x2=b,x3=b,x4=b,x5=g,x6=r,x7=\{r,b\})\)
- \((r,b,b,b,g,r)\) conflict set of \(x7\)
- \((r,-,b,b,g,-)\) c.s. of \(x7\)
- \((r,-,b,-,-,-,-)\) minimal c.s
- Leaf deadend: \((r,b,b,b,g,r)\)
Example 6.1.4 For the problem in Figure 6.1, the tuple $(<x_1, \text{red}>, <x_2, \text{blue}>, <x_3, \text{blue}>, <x_4, \text{blue}>, <x_5, \text{green}>, <x_6, \text{red}>)$ is a conflict set relative to $x_7$ because it cannot be consistently extended to any value of $x_7$. It is also a leaf dead-end. Notice that the assignment $(<x_1, \text{blue}>, <x_2, \text{green}>, <x_3, \text{red}>)$ is a no-good that is not a conflict set relative to any single variable.

\[\square\]
Conflict-set analysis

Definition 6.1.1 (conflict set) Let $\bar{a} = (a_{i_1}, ..., a_{i_k})$ be a consistent instantiation of an arbitrary subset of variables, and let $x$ be a variable not yet instantiated. If there is no value $b$ in the domain of $x$ such that $(\bar{a}, x = b)$ is consistent, we say that $\bar{a}$ is a conflict set of $x$, or that $\bar{a}$ conflicts with variable $x$. If, in addition, $\bar{a}$ does not contain a subtuple that is in conflict with $x$, $\bar{a}$ is called a minimal conflict set of $x$.

Definition 6.1.2 (leaf dead-end) Let $\bar{a}_i = (a_1, ..., a_i)$ be a consistent tuple. If $\bar{a}_i$ is in conflict with $x_{i+1}$, it is called a leaf dead-end.

Definition 6.1.3 (no-good) Given a network $\mathcal{R} = (X, D, C)$, any partial instantiation $\bar{a}$ that does not appear in any solution of $\mathcal{R}$ is called a no-good. Minimal no-goods have no no-good subtuples.

Definition 6.1.5 (safe jump) Let $\bar{a}_i = (a_1, ..., a_i)$ be a leaf dead-end state. We say that $x_j$, where $j \leq i$, is safe if the partial instantiation $\bar{a}_j = (a_1, ..., a_j)$ is a no-good, namely, it cannot be extended to a solution.
Gaschnig’s backjumping: Culprit variable

Definition 6.2.1 (culprit variable) Let $\vec{a}_i = (a_1, \ldots, a_i)$ be a leaf dead-end. The culprit index relative to $\vec{a}_i$ is defined by $b = \min\{j \leq i \mid \vec{a}_j \text{ conflicts with } x_{i+1}\}$. We define the culprit variable of $\vec{a}_i$ to be $x_b$.

- If $a_i$ is a leaf deadend and $x_b$ its culprit variable, then $a_b$ is a safe backjump destination and $a_j$, $j < b$ is not.
- The culprit of $x_7$ ($r,b,b,b,g,r$) is $(r,b,b) \rightarrow x_3$
Gaschnig’s backjumping [1979]

- Gaschnig uses a marking technique to compute the culprit.
- Each variable \( x_j \) maintains a pointer \( \text{latset}_j \) to the latest ancestor incompatible with any of its values.
- While forward generating \( \hat{a}_i \), keep array \( \text{latest}_i \), \( 1 \leq j \leq n \), of pointers to the last value conflicted with some value of \( x_j \).
- The algorithm jumps from a leaf-dead-end \( x_{i+1} \) back to \( \text{latest}_{i+1} \) which is its culprit.
Example 6.2.3 Consider the problem in Figure 6.1 and the order $d_1$. At the dead-end for $x_7$ that results from the partial instantiation (< $x_1$, red >, < $x_2$, blue >, < $x_3$, blue >, < $x_4$, blue >, < $x_5$, green >, < $x_6$, red >), latest$_7 = 3$, because $x_7 = \text{red}$ was ruled out by < $x_1$, red >, $x_7 = \text{blue}$ was ruled out by < $x_3$, blue >, and no later variable had to be examined. On returning to $x_3$, the algorithm finds no further values to try ($D_3 = \emptyset$). Since latest$_3 = 2$, the next variable examined will be $x_2$. Thus we see the algorithm’s ability to backjump at leaf dead-ends. On subsequent dead-ends, as in $x_3$, it goes back to its preceding variable only. An example of the algorithm’s practice of pruning the search space is given in Figure 6.2.
Properties

- Gaschnig’s backjumping implements only safe and maximal backjumps in leaf-deadends.
Gaschnig jumps only at leaf-dead-ends
Internal dead-ends: dead-ends that are non-leaf

Example 0.3.1: In Figure 0.4, all of the backjumps illustrated lead to internal dead-ends, except for the jump back to \(\langle x_1, \text{green} \rangle, \langle x_2, \text{blue} \rangle, \langle x_3, \text{red} \rangle, \langle x_4, \text{blue} \rangle\), because this is the only case where another value exists in the domain of the culprit variable.
Example of graph-based backjumping scenarios

- Scenario 1, deadend at $x_4$: $I_4(x_4) = \{x_1\}$
- Scenario 2: deadend at $x_5$: $I_4(x_4, x_5) = \{x_1\}$
- Scenario 3: deadend at $x_7$: $I_4(x_7, x_5, x_4) = \{x_1, x_3\}$
- Scenario 4: deadend at $x_6$: $I_4(x_6, x_5, x_4) = \{x_1, x_3\}$
Graph-based backjumping

- Uses only graph information to find culprit
- Jumps both at leaf and at internal dead-ends
- Whenever a deadend occurs at x, it jumps to the most recent variable y connected to x in the graph. If y is an internal deadend it jumps back further to the most recent variable connected to x or y.
- The analysis of conflict is approximated by the graph.
- Graph-based algorithm provide graph-theoretic bounds.
Ancestors and parents

- \( \text{anc}(x7) = \{x5, x3, x4, x1\} \)
- \( p(x7) = x5 \)
- \( p(r, b, b, b, g, r) = x5 \)

**Definition 6.3.2 (ancestors, parent)** Given a constraint graph and an ordering of the nodes \( d \), the ancestor set of variable \( x \), denoted \( \text{anc}(x) \), is the subset of the variables that precede and are connected to \( x \). The parent of \( x \), denoted \( p(x) \), is the most recent (or latest) variable in \( \text{anc}(x) \). If \( \vec{a}_i = (a_1, \ldots, a_i) \) is a leaf dead-end, we equate \( \text{anc}(\vec{a}_i) \) with \( \text{anc}(x_{i+1}) \), and \( p(\vec{a}_i) \) with \( p(x_{i+1}) \).
Definition 6.3.5 (session) We say that backtracking invisits $x_i$ if it processes $x_i$ coming from a variable earlier in the ordering. The session of $x_i$ starts upon the invisiting of $x_i$ and ends when retracting to a variable that precedes $x_i$. At a given state of the search where variable $x_i$ is already instantiated, the current session of $x_i$ is the set of variables processed by the algorithm since the most recent invisit to $x_i$. The current session of $x_i$ includes $x_i$ and therefore the session of a leaf dead-end variable has a single variable.

Definition 6.3.6 (relevant dead-ends) The relevant dead-ends of $x_i$’s session are defined recursively as follows. The relevant dead-ends of a leaf dead-end $x_i$, denoted $r(x_i)$, is $x_i$. If $x_i$ is variable to which the algorithm retracted from $x_j$, then the relevant-dead-ends of $x_i$ are the union of its current relevant dead-ends and the ones inherited from $x_j$, namely, $r(x_i) = r(x_i) \cup r(x_j)$.

Definition 6.3.7 (induced ancestors, induced parent) Let $x_i$ be a variable that is an internal or leaf dead-end. Let $Y$ be a subset of the variables consisting of all its relevant dead-ends in the current session of $x_i$. We denote $\text{anc}(Y) = \cup_{y \in Y} \text{anc}(y)$. The induced ancestor set of $x_i$ relative to $Y$, $I_i(Y)$, is the union of all $Y$’s ancestors, restricted to variables that precede $x_i$. Formally, $I_i(Y) = \text{anc}(Y) \cap \{x_1, \ldots, x_{i-1}\}$. The induced parent of $x_i$ relative to $Y$, $P_i(Y)$, is the latest variable in $I_i(Y)$. We call $P_i(Y)$ the graph-based culprit of $x_i$. 
Graph-based back-jumping algorithm, but we need to jump at internal dead-ends too

```
procedure GRAPH-BASED-BACKJUMPING
Input: A constraint network $\mathcal{R} = (X, D, C)$
Output: Either a solution, or a decision that the network is inconsistent.

compute $\text{anc}(x_i)$ for each $x_i$ (see Definition 6.3.2 in text)
i ← 1 (initialize variable counter)
$D_i' ← D_i$ (copy domain)
$I_i ← \text{anc}(x_i)$ (copy of anc() that can change)

while $1 ≤ i ≤ n$
    instantiate $x_i ← \text{SELECTVALUE}$
    if $x_i$ is null (no value was returned)
        $i_{\text{prev}} ← i$
        $i ← \text{latest index in } I_i$ (backjump)
        $I_i ← I_i \cup I_{i_{\text{prev}}} \setminus \{x_i\}$
    else
        $i ← i + 1$
        $D_i' ← D_i$
        $I_i ← \text{anc}(x_i)$
    end while

if $i = 0$
    return “inconsistent”
else
    return instantiated values of $\{x_1, \ldots, x_n\}$
end procedure

procedure SELECTVALUE (same as BACKTRACKING’s)
while $D_i'$ is not empty
    select an arbitrary element $a \in D_i'$, and remove $a$ from $D_i'$
    if CONSISTENT($\vec{a}_{i-1}, x_i = a$)
        return $a$
    end while
return null (no consistent value)
end procedure
```

Figure 6.5: The graph-based backjumping algorithm.
Properties of graph-based back-jumping

- Algorithm graph-based back-jumping jumps back at any dead-end variable as far as graph-based information allows.

- For each variable, the algorithm maintains the induced-ancestor set $I_i$ relative the relevant dead-ends in its current session.
Conflict-directed backjumping
(Prosser 1990)

- Extend Gaschnig’s backjump to internal dead-ends.
- Exploits information gathered during search.
- For each variable the algorithm maintains an induced jumpback set, and jumps to most recent one.
- **Use the following concepts:**
  - An ordering over variables induced a strict ordering between constraints: R1<R2<…Rt
  - Use *earliest minimal conflict-set* (emc(x_(i+1)) ) of a deadend.
  - Define the *jumpback set* of a deadend
Conflict-directed backjumping: Gaschnig’s style jumpback in all deadends:

Definition 6.4.1 (earlier constraint) Given an ordering of the variables in a constraint problem, we say that constraint $R$ is earlier than constraint $Q$ if the latest variable in $\text{scope}(R) - \text{scope}(Q)$ precedes the latest variable in $\text{scope}(Q) - \text{scope}(R)$.

Definition 6.4.2 (earliest minimal conflict set) For a network $R = (X, D, C)$ with an ordering of the variables $d$, let $\tilde{a}_i$ be a leaf dead-end tuple whose dead-end variable is $x_{i+1}$. The earliest minimal conflict set of $\tilde{a}_i$, denoted $\text{emc}(\tilde{a}_i)$, can be generated as follows. Consider the constraints in $C = \{R_1, \ldots, R_c\}$ with scopes $\{S_1, \ldots, S_c\}$, in order as defined in Definition 6.4.1. For $j = 1$ to $c$, if there exists $b \in D_{i+1}$ such that $R_j$ is violated by $(\tilde{a}_i, x_{i+1} = b)$, but no constraint earlier than $R_j$ is violated by $(\tilde{a}_i, x_{i+1} = b)$, then $\text{var-emc}(\tilde{a}_i) \leftarrow \text{var-emc}(\tilde{a}_i) \cup S_j$. $\text{emc}(\tilde{a}_i)$ is the subtuple of $\tilde{a}_i$ containing just the variable-value pairs of $\text{var-emc}(\tilde{a}_i)$. Namely, $\text{emc}(\tilde{a}_i) = \tilde{a}_i[\text{var} - \text{emc}(\tilde{a}_i)]$.

Definition 6.4.3 (jumpback set) The jumpback set of a leaf dead-end $J_{i+1}$ of $x_{i+1}$ is its $\text{var-emc}(\tilde{a}_i)$. The jump-back set of an internal state $\tilde{a}_i$ includes all the $\text{var-emc}(\tilde{a}_j)$ of all the relevant dead-ends $\tilde{a}_j$ $j \geq i$, that occurred in the current session of $x_i$. Formally, $J_i = \bigcup \{\text{var-emc}(\tilde{a}_j) \mid \tilde{a}_j \text{ is a relevant dead-end in } x_i \text{'s session}\}$.
Example 6.4.5 Consider the problem of Figure 6.1 using ordering $d_1 = (x_1, \ldots, x_7)$. Given the dead-end at $x_7$ and the assignment $\vec{a}_6 = (\text{blue}, \text{green}, \text{red}, \text{red}, \text{blue}, \text{red})$, the emc set is $\langle x_1, \text{blue} \rangle, \langle x_3, \text{red} \rangle$, since it accounts for eliminating all the values of $x_7$. Therefore, algorithm conflict-directed backjumping jumps to $x_3$. Since $x_3$ is an internal dead-end whose own var – emc set is $\{x_1\}$, the jumpback set of $x_3$ includes just $x_1$, and the algorithm jumps again, this time back to $x_1$. □
Properties

- Given a dead-end $\vec{a}_i$, the latest variable in its jumpback set $J_i$ is the earliest variable to which it is safe to jump.
- This is the culprit.
- Algorithm conflict-directed backtracking jumps back to the latest variable in the dead-ends’s jumpback set, and is therefore safe and maximal.
procedure CONFLICT-DIRECTED-BACKJUMPING
Input: A constraint network $R = \langle X, D, C \rangle$.
Output: Either a solution, or a decision that the network is inconsistent.

\[
i \leftarrow 1 \quad \text{(initialize variable counter)}
\]

\[
D'_i \leftarrow D_i \quad \text{(copy domain)}
\]

\[
J_i \leftarrow \emptyset \quad \text{(initialize conflict set)}
\]

while $1 \leq i \leq n$

\[
\text{instantiate } x_i \leftarrow \text{SELECT-VALUE-CBJ}
\]

if $x_i$ is null \quad \text{(no value was returned)}

\[
i_{\text{prev}} \leftarrow i
\]

\[
i \leftarrow \text{index of last variable in } J_i \quad \text{(backjump)}
\]

\[
J_i \leftarrow J_i \cup J_{i_{\text{prev}}} - \{x_i\} \quad \text{(merge conflict sets)}
\]

else

\[
i \leftarrow i + 1 \quad \text{(step forward)}
\]

\[
D'_i \leftarrow D_i \quad \text{(reset mutable domain)}
\]

\[
J_i \leftarrow \emptyset \quad \text{(reset conflict set)}
\]

end while

if $i = 0$

return “inconsistent”

else

return instantiated values of $\{x_1, \ldots, x_n\}$

end procedure

subprocedure SELECT-VALUE-CBJ

while $D'_i$ is not empty

\[\text{select an arbitrary element } a \in D'_i, \text{ and remove } a \text{ from } D'_i\]

\[
\text{consistent} \leftarrow \text{true}
\]

\[
k \leftarrow 1
\]

while $k < i$ and consistent

\[\text{consistent}(\sigma_h, x_i = a)\]

\[
k \leftarrow k + 1
\]

else

let $R_S$ be the earliest constraint causing the conflict

add the variables in $R_S$’s scope $S$, but not $x_i$, to $J_i$ \quad \text{consistent} \leftarrow \text{false}

end while

if consistent

return $a$

end while

return null \quad \text{(no consistent value)}

end procedure

Figure 6.7: The conflict-directed backjumping algorithm.
Graph-based backjumping on DFS orderings

Example 6.5.1 Consider, once again, the CSP in Figure 6.1. A DFS ordering \( d_2 = (x_1, x_7, x_4, x_5, x_6, x_2, x_3) \) and its corresponding DFS spanning tree are given in Figure 6.6c,d. If a dead-end occurs at node \( x_3 \), the algorithm retreats to its DFS parent, which is \( x_7 \). 

![Diagrams of ordered constraint graphs and DFS spanning tree]

Figure 6.6: Several ordered constraint graphs of the problem in Figure 6.1: (a) along ordering \( d_1 = (x_1, x_2, x_3, x_4, x_5, x_6, x_7) \), (b) the induced graph along \( d_1 \), (c) along ordering \( d_2 = (x_3, x_7, x_4, x_6, x_5, x_2, x_3) \), and (d) a DFS spanning tree along ordering \( d_2 \).
Graph-based backjumping on DFS ordering

- Example: \(d = x_1, x_2, x_3, x_4, x_5, x_6, x_7\)
- Constraints: \((6,7)(5,2)(2,3)(5,7)(2,7)(2,1)(2,3)(1,4)3,4)\)
- Rule: go back to parent. No need to maintain parent set

**Theorem 6.5.2** Given a DFS ordering of the constraint graph, if \(f(x)\) denotes the DFS parent of \(x\), then, upon a dead-end at \(x\), \(f(x)\) is \(x\)'s graph-based earliest safe variable for both leaf and internal dead-ends.
Complexity of graph-based backjumping on DFS ordering

- \( T_i \) = number of nodes in the And-Or search space rooted at \( x_i \) (level \( m-i \))
- Each assignment of a value to \( x_i \) generates subproblems:
  - \( T_i = k \cdot b \cdot T_{i-1} \)
  - \( T_0 = k \)
- Solution: \( T_m = b^m k^{m+1} \)

**Theorem 6.5.3** When graph-based backjumping is performed on a DFS ordering of the constraint graph, the number of nodes visited is bounded by \( O((b^m k^{m+1})) \), where \( b \) bounds the branching degree of the DFS tree associated with that ordering, \( m \) is its depth and \( k \) is the domain size. The time complexity (measured by the number of consistency checks) is \( O(ek(bk)^m) \), where \( e \) is the number of constraints.
Theorem 6.5.5 If $d$ is a DFS ordering of $(G^*, d_1)$ for some ordering $d_1$, having depth $m_d^*$, then the complexity of graph-based backjumping using ordering $d$ is $O(\exp(m_d^*))$. 
Graph parameters

- C - size of a cycle-cutset
- m - depth of a dfs in any induced graph
- m_s a simple depth of a dfs tree.

What is the relationship between these?
Learning, constraint recording

- Learning means recording conflict sets
- An opportunity to learn is when deadend is discovered.
- Goal of learning to not discover the same deadends.
- Try to identify small conflict sets
- Learning prunes the search space.
Look-back: constraint recording

- $(x_1=2, x_2=2, x_3=1, x_4=2)$ IS a dead-end
- Conflicts to record:
  - $(x_1=2, x_2=2, x_3=1, x_4=2)$ 4-ary
  - $(x_3=1, x_4=2)$ binary
  - $(x_4=2)$ unary
Learning algorithms

- Graph-based learning
- Deep vs shallow learning
- Jumpback learning
- Non-systematic randomized learning
- Complexity of backtracking with learning
- Look-back for SAT
Figure 6.9: The search space explicated by backtracking on the CSP from Figure 6.1, using the variable ordering \((x_6, x_3, x_4, x_2, x_7, x_1, x_5)\) and the value ordering \((\text{blue, red, green, teal})\). Part (a) shows the ordered constraint graph, part (b) illustrates the search space. The cut lines in (b) indicate branches not explored when graph-based learning is used.
Graph-based learning algorithm

**procedure** GRAPH-BASED-BACKJUMP-LEARNING

- instantiate $x_i \leftarrow \text{SELECTVALUE}$
  - if $x_i$ is null (no value was returned)
    - record a constraint prohibiting $\bar{a}_{i-1}[I_i]$.
    - $i_{prev} \leftarrow i$
    - (algorithm continues as in Fig. 6.5)

Figure 6.10: Graph-based backjumping learning, modifying CBJ
Deep learning

- Deep learning: recording all and only minimal conflict sets
- Example:
- Although most accurate, overhead is prohibitive: the number of conflict sets in the worst-case:
  \[ \binom{r}{r/2} = 2^r \]
Jumpback learning

- Record the jumpback assignment

Example 6.7.2 For the problem and ordering of Example 6.7.1 at the first dead-end, jumpback learning will record the no-good \((x_2 = \text{green}, x_3 = \text{blue}, x_7 = \text{red})\), since that tuple includes the variables in the jumpback set of \(x_1\).

```
procedure CONFLICT-DIRECTED-BACKJUMP-LEARNING

    instantiate \(x_i \leftarrow \text{SELECTVALUE-CBJ}\)
    if \(x_i\) is null                   (no value was returned)
        record a constraint prohibiting \(a_{i-1}[J_i]\) and corresponding values
        iprev \(\leftarrow i\)
        (algorithm continues as in Fig. 6.7)
```

Figure 6.11: Conflict-directed bakjump-learning, modifying CBJ
Bounded and relevance-based learning

Bounding the arity of constraints recorded.

- When bound is $i$: $i$-ordered graph-based, $i$-order jumpback or $i$-order deep learning.
- Overhead complexity of $i$-bounded learning is time and space exponential in $i$.

**Definition 6.7.3 (i-relevant)** A no-good is $i$-relevant if it differs from the current partial assignment by at most $i$ variable-value pairs.

**Definition 6.7.4 (i’th order relevance-bounded learning)** An $i$’th order relevance-bounded learning scheme maintains only those learned no-goods that are $i$-relevant.
Non-systematic randomized learning

- Do search in a random way with interrupts, restarts, unsafe backjumping, but record conflicts.
- Guaranteed completeness.
Complexity of backtrack-learning

Theorem 6.7.5 Let $d$ be an ordering of a constraint graph, and let $w^*(d)$ be its induced width. Any backtracking algorithm using ordering $d$ with graph-based learning has a space complexity of $O((nk)^{w^*(d)+1})$ and a time complexity of $O((2nk)^{w^*(d)+1})$, where $n$ is the number of variables and $k$ bounds the domain sizes.

The number of dead-ends is bounded by the number of possible no-goods of size $w^*$

$$
\sum_{i=1}^{w^*(d)} \binom{n}{i} k^i = O((nk)^{w^*(d)+1})
$$

Number of constraint tests per dead-end are

$$
O(2^{w^*(d)})
$$
Complexity of backtrack-learning (refined)

- **Theorem:** Any backtracking algorithm using graph-based learning along d has a space complexity $O(n k^{w^*(d)})$ and time complexity $O(n^2 (2k)^{(w^*(d)+1)}$ (book). Refined more: $O(n^2 k^{w^*(d)})$

- **Proof:** The number of deadends for each variable is $O(k^{w^*(d)})$, yielding $O(n k^{w^*(d)})$ deadends. There are at most $kn$ values between two successive deadends: $O(k n^2 k^{w^*(d)})$ number of nodes in the search space. Since at most $O(2^{w^*(d)})$ constraints are check we get $O(n^2 (2k)^{(w^*(d)+1)}$.

- Alternatively, if we have $O(n k^{w^*(d)})$ leaves, we have $k$ to $n$ times as many internal nodes, yielding between $O(n k^{(w^*(d)+1)})$
- And $O(n^2 k^{w^*(d)})$ nodes.
Analysis of backjumping and learning along DFS?

- Can we have a better bound than $O(n^2 k^m)$?
Look-back for SAT

- A partial assignment is a set of literals: \( \sigma \)
- A jumpback set if a J-clause:
- Upon a leaf deadend of \( x \) resolve two clauses, one enforcing \( x \) and one enforcing \( \neg x \) relative to the current assignment
- A clause forces \( x \) relative to assignment \( \sigma \) if all the literals in the clause are negated in \( \sigma \).
- Resolving the two clauses we get a nogood.
- If we identify the earliest two clauses we will find the earliest conflict.
- The argument can be extended to internal deadends.
Look-back for SAT

procedure SAT-CBJ-LEARN
Input: A CNF theory \( \varphi \), assigned variables \( \sigma \) over \( x_1, ..., x_{i-1} \), unassigned variables \( X \).
Output: Either a solution, or a decision that the network is inconsistent.
1. \( J_i \leftarrow \emptyset \)
2. While \( 1 \leq i \leq n \)
3. Select the next variable: \( x_i \in X, X \leftarrow X - \{x_i\} \)
4. instantiate \( x_i \leftarrow \text{SELECTVALUE-CBJ} \).
5. If \( x_i \) is null (no value returned), then
6. add \( J_{x_i} \) to \( \varphi \) (learning)
7. \( i_{prev} \leftarrow \text{index of last variable in } J_i \) (backjump)
8. \( J_i \leftarrow \text{resolve}(J_i, J_{i_{prev}}) \) (merge conflict sets)
9. else,
10. \( i \leftarrow i + 1 \) (go forward)
11. \( J_i \leftarrow \emptyset \) (reset conflict set)
12. Endwhile
13. if \( i = 0 \) Return "inconsistent"
14. else, return the set of literals \( \sigma \)
end procedure

subprocedure SELECTVALUE-CBJ
1. If CONSISTENT(\( \sigma \cup x_i \)) then return \( \sigma \leftarrow \sigma \cup \{x_i\} \)
2. If CONSISTENT(\( \sigma \cup \neg x_i \)) then return \( \sigma \leftarrow \sigma \cup \{\neg x_i\} \)
3. else,
4. determine \( \alpha \) and \( \beta \) the two earliest clauses forcing \( x_i \) and \( \neg x_i \),
5. \( J_i \leftarrow \text{resolve}(\alpha, \beta) \).
5. Return \( x_i \leftarrow \text{null} \) (no consistent value)
end procedure
Integration of algorithms

procedure FC-CBJ
Input: A constraint network \( \mathcal{R} = (X, D, C) \).
Output: Either a solution, or a decision that the network is inconsistent.

\[
i \leftarrow 1 \quad \text{(initialize variable counter)} \\
\text{call SELECTVARIABLE} \quad \text{(determine first variable)} \\
D_i' \leftarrow D_i \text{ for } 1 \leq i \leq n \quad \text{(copy all domains)} \\
J_i \leftarrow \emptyset \quad \text{(initialize conflict set)} \\
\text{while } 1 \leq i \leq n \\
\text{instantiate } x_i \leftarrow \text{SELECTVALUE-FC-CBJ} \\
\text{if } x_i \text{ is null} \quad \text{(no value was returned)} \\
\quad iprev \leftarrow i \\
\quad i \leftarrow \text{latest index in } J_i \quad \text{(backjump)} \\
\quad J_i \leftarrow J_i \cup \{iprev\} - \{x_i\} \\
\quad \text{reset each } D'_k, k > i, \text{ to its value before } x_i \text{ was last instantiated} \\
\text{else} \\
\quad i \leftarrow i + 1 \quad \text{(step forward)} \\
\quad \text{call SELECTVARIABLE} \quad \text{(determine next variable)} \\
\quad D_i' \leftarrow D_i \\
\quad J_i \leftarrow \emptyset \\
\text{end while} \\
\text{if } i = 0 \\
\quad \text{return } \text{“inconsistent”} \\
\text{else} \\
\quad \text{return instantiated values of } \{x_1, \ldots, x_n\} \\
\text{end procedure}
subprocedure SELECT\text{VALUE-FC-CBJ}

while $D'_i$ is not empty
    select an arbitrary element $a \in D'_i$, and remove $a$ from $D'_i$
    $\text{empty-domain} \leftarrow \text{false}$
    for all $k$, $i < k \leq n$
        for all values $b$ in $D'_k$
            if not CONSISTENT($\bar{a}_{i-1}, x_i = a, x_k = b$)
                let $R_S$ be the earliest constraint causing the conflict
                add the variables in $R_S$’s scope $S$, but not $x_k$, to $J_k$
                remove $b$ from $D'_k$
        endfor
    if $D'_k$ is empty $(x_i = a$ leads to a dead-end$)$
        $\text{empty-domain} \leftarrow \text{true}$
    endfor
if $\text{empty-domain}$ (don’t select $a$)
    reset each $D'_k$ and $j_k, i < k \leq n$, to status before $a$ was selected
else
    return $a$
end while
return null (no consistent value)
end subprocedure

Figure 6.14: The SelectValue subprocedure for FC-CBJ.
Relationships between various backtracking algorithms
Empirical comparison of algorithms

- Benchmark instances
- Random problems
- Application-based random problems
- Generating fixed length random k-sat \((n,m)\) uniformly at random
- Generating fixed length random CSPs
- \((N,K,T,C)\) also arity, \(r\)
The Phase transition (m/n)
Some empirical evaluation

- Sets 1-3 reports average over 2000 instances of random csps from 50% hardness. Set 1: 200 variables, set 2: 300, Set 3: 350. All had 3 values.
- Dimacs problems

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Set 1</th>
<th>Set 2</th>
<th>Set 3</th>
<th>ssa 038</th>
<th>ssa 158</th>
</tr>
</thead>
<tbody>
<tr>
<td>FC</td>
<td>207</td>
<td>68.5</td>
<td>-</td>
<td>46</td>
<td>52</td>
</tr>
<tr>
<td>FC+AC</td>
<td>40</td>
<td>55.4</td>
<td>1</td>
<td>1</td>
<td>3.5</td>
</tr>
<tr>
<td>FCr-CBJ</td>
<td>189</td>
<td>69.2</td>
<td>222</td>
<td>182</td>
<td>40</td>
</tr>
<tr>
<td>FC-CBJ+LVO</td>
<td>167</td>
<td>73.8</td>
<td>132</td>
<td>119</td>
<td>32</td>
</tr>
<tr>
<td>FC-CBJ+LRN</td>
<td>186</td>
<td>63.4</td>
<td>32</td>
<td>1</td>
<td>23</td>
</tr>
<tr>
<td>FC-CBJ+LRN+LVO</td>
<td>160</td>
<td>74.0</td>
<td>26</td>
<td>1</td>
<td>16</td>
</tr>
</tbody>
</table>

Figure 6.16: Empirical comparison of six selected CSP algorithms. See text for explanation. In each column of numbers, the first number indicates the number of nodes in the search tree, rounded to the nearest thousand, and final 000 omitted; the second number is CPU seconds.