Backtracking search: look-back

ICS 275
Spring 2010
Look-back:
Backjumping / Learning

- **Backjumping:**
  - In deadends, go back to the most recent culprit.

- **Learning:**
  - constraint-recording, no-good recording.
  - good-recording

Fall 2010
Backjumping

- \((X_1=r,x_2=b,x_3=b,x_4=b,x_5=g,x_6=r,x_7={r,b})\)
- \((r,b,b,b,g,r)\) conflict set of \(x_7\)
- \((r,-,b,b,g,-)\) c.s. of \(x_7\)
- \((r,-,b,-,-,-,-)\) minimal conflict-set
- Leaf deadend: \((r,b,b,b,g,r)\)
- Every conflict-set is a no-good
Gaschnig jumps only at leaf-dead-ends

Internal dead-ends: dead-ends that are non-leaf

Example 6.3.1 In Figure 6.4, 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.

Fall 2010
Gaschnig jumps only at leaf-dead-ends

Internal dead-ends: dead-ends that are non-leaf

Example 6.3.1 In Figure 6.4, all of the backjumps illustrated lead to internal dead-ends, except for the jump back to \((x_1, \text{green}), (x_2, \text{blue}), (x_3, \text{red}), (x_4, \text{blue})\), because this is the only case where another value exists in the domain of the culprit variable.
Backjumping styles

- **Jump at leaf only** *(Gaschnig 1977)*
  - Context-based

- **Graph-based** *(Dechter, 1990)*
  - Jumps at leaf and internal dead-ends, graph information

- **Conflict-directed** *(Prosser 1993)*
  - Context-based, jumps at leaf and internal dead-ends
Gaschnig’s backjumping: Culprit variable

**Definition 6.2.1 (culprit variable)** Let $\vec{a}_i = (a_1, ..., a_i)$ be a leaf dead-end. The culprit index relative to $\vec{a}_i$ is defined by $b = \min\{j \leq i | \vec{a}_j$ 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 Implementation [1979]

- Gaschnig uses a marking technique to compute culprit.
- Each variable $x_j$ maintains a pointer (latest$_j$) to the latest ancestor incompatible with any of its values.
- While forward generating $\vec{a}_i$, keep array 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 latest$_{(i+1)}$ which is its culprit.
Gaschnig’s backjumping

```
procedure GASCHNIG’S-BACKJUMPING
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)}
\[ D_i' \leftarrow D_i \]  \quad \text{(copy domain)}
\[ \text{latest}_i \leftarrow 0 \]  \quad \text{(initialize pointer to culprit)}

\textbf{while} 1 \leq i \leq n

\textbf{instantiate } x_i \leftarrow \text{SELECTVALUE-GBJ}
\textbf{if} $x_i$ is null

\textbf{else}

\[ i \leftarrow i + 1 \]
\[ D'_i \leftarrow D_i \]
\[ \text{latest}_i \leftarrow 0 \]

\textbf{end while}

\textbf{if} $i = 0$

\textbf{return} “inconsistent”
\textbf{else}

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

procedure SELECTVALUE-GBJ

\textbf{while} $D'_i$ is not empty

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

\textbf{consistent} \leftarrow \text{true}
\[ k \leftarrow 1 \]

\textbf{while} $k < i$ and \textbf{consistent}

\textbf{if} $k > \text{latest}_i$

\textbf{else}

\[ k \leftarrow k + 1 \]

\textbf{end while}

\textbf{if} \textbf{consistent}

\textbf{return} $a$
\textbf{end while}

\textbf{return} null  \quad \text{(no consistent value)}
\textbf{end procedure}
```

Figure 6.3: Gaschnig’s backjumping algorithm.
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 $(<x_1, \text{red}>, <x_2, \text{blue}>, <x_3, \text{blue}>, <x_4, \text{blue}>, <x_5, \text{green}>, <x_6, \text{red}>)$, latest$_7 = 3$, because $x_7 = \text{red}$ was ruled out by $<x_1, \text{red}>$, $x_7 = \text{blue}$ was ruled out by $<x_3, \text{blue}>$, 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.
Example 6.3.1 In Figure 6.4, all of the backjumps illustrated lead to internal dead-ends, except for the jump back to \( (x_1, \text{green}), (x_2, \text{blue}), (x_3, \text{red}), (x_4, \text{blue}) \), because this is the only case where another value exists in the domain of the culprit variable.
Gaschnig jumps only at leaf-dead-ends

Internal dead-ends: dead-ends that are non-leaf

Example 6.3.1 In Figure 6.4, all of the backjumps illustrated lead to internal dead-ends, except for the jump back to $(<x_1, green>, <x_2, blue>, <x_3, red>, <x_4, blue>)$, because this is the only case where another value exists in the domain of the culprit variable.
Graph-based backjumping scenarios
Internal deadend at X4

- Scenario 1, deadend at x4:
- Scenario 2: deadend at x5:
- Scenario 3: deadend at x7:
- Scenario 4: deadend at x6:

Figure 6.1: A modified coloring problem.
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

- \( \text{anc}(x_7) = \{x_5, x_3, x_4, x_1\} \)
- \( \text{p}(x_7) = x_5 \)
- \( \text{p}(r, b, b, b, g, r) = x_5 \)

**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 \( \text{p}(x) \), 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 \( \text{p}(\vec{a}_i) \) with \( \text{p}(x_{i+1}) \).
Internal deadends analysis

Definition 6.3.5 (session) We say that backtracking invists $x_i$ 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 invist 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 $\text{anc}(Y) = \bigcup_{y \in Y} \text{anc}(y)$. 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 culpript of $x_i$. 

Fall 2010
Graph-based backjumping algorithm, but we need to jump at internal deadends 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)
\( i \leftarrow 1 \) (initialize variable counter)
\( D'_i \leftarrow D_i \) (copy domain)
\( I_i \leftarrow \text{anc}(x_i) \) (copy of \( \text{anc}(\cdot) \) that can change)
while \( 1 \leq i \leq n \)
    instantiate \( x_i \leftarrow \text{SELECTVALUE} \)
    if \( x_i \) is null (no value was returned)
        \( \text{spren} \leftarrow i \)
        \( i \leftarrow \text{latest index in } I_i \) (backjump)
        \( I_i \leftarrow I_i \cup \text{spren} - \{x_i\} \)
    else
        \( i \leftarrow i + 1 \)
        \( D'_i \leftarrow D_i \)
        \( I_i \leftarrow \text{anc}(x_i) \)
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(\( \bar{a}_{i-1}, x_i = a \))
        return \( a \)
end while
return null (no consistent value)
end procedure
```

When not all variables \( X \cdot i \) are relevant deadends?
In the session above
See example 6.6

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

- Algorithm graph-based backjumping jumps back at any deadend 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.
- The size of the induced ancestor set is at most $w^*(d)$.
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: \( R_1 < R_2 < \ldots R_t \)
- Use earliest minimal conflict-set (\( emc(x_{(i+1)}) \)) of a deadend.
- Define the jumpback set of a deadend
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 $\vec{a}_6 = (\text{blue, green, red, red, blue, red})$, the emc set is $(< x_1, \text{blue }>, < x_3, \text{red }>)$, 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 $\textit{var} – \textit{emc}$ set is $\{x_1\}$, the jumpback set of $x_3$ includes just $x_1$, and the algorithm jumps again, this time back to $x_1$. \qed
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.
Conflict-directed backjumping

**procedure** CONFLICT-DIRECTED-BACKJUMPING

**Input:** A constraint network $\mathcal{R} = (X, D, C)$.

**Output:** Either a solution, or a decision that the network is inconsistent.

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

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

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

while $1 \leq i \leq n$

\[
i \leftarrow \text{SELECTVALUE-CBJ} \quad \text{(instantiate)}
\]

if $x_i$ is null
\[
t_{prev} \leftarrow i
\]
\[
i \leftarrow \text{index of last variable in } J_i \quad \text{(backjump)}
\]
\[
J_i \leftarrow J_i \cup J_{t_{prev}} \setminus \{x_i\} \quad \text{(merge conflict sets)}
\]

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 $i = 0$
\[
\text{return "inconsistent"}
\]

else
\[
\text{return instantiated values of } \{x_1, \ldots, x_n\}
\]

end procedure

**subprocedure** SELECTVALUE-CBJ

while $D'_i$ is not empty

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

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

\[
k \leftarrow 1
\]

while $k < i$ and consistent

\[
\text{if CONFLICT}([a_k, x_i = a])
\]

\[
k \leftarrow k + 1
\]

else

\[
\text{let } R_S \text{ be the earliest constraint causing the conflict}
\]

\[
\text{add the variables in } R_S \text{'s scope } S, \text{ but not } x_i, \text{ to } J_i
\]

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

end while

if consistent
\[
\text{return } a
\]

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 $x_7$.

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 $d_2 = (x_1, x_7, x_4, x_5, x_6, x_2, x_3)$, and (d) a DFS spanning tree along ordering $d_2$. 

Fall 2010
Complexity of Graph-based Backjumping

- $T_i = \text{number of nodes in the AND/OR search space rooted at } x_i \text{ (level } m-i\text{)}$
- 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.
DFS of graph and induced graphs

Spanning-tree of a graph;
DFS spanning trees, BFS spanning trees.
Complexity of Backjumping uses pseudo-tree analysis

Simple: always jump back to parent in pseudo tree
Complexity for csp: $\exp(\text{tree-depth})$
Complexity for csp: $\exp(w \cdot \log n)$
Look-back: No-good Learning

Learning means recording conflict sets used as constraints to prune future search space.

- \((x_1=2, x_2=2, x_3=1, x_4=2)\) 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, 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.
Nogoods explain deadends

Learning means recording explanations to conflicts
They are implied constraints

- Conflicts to record are explanations
  - \((x_1=2,x_2=2,x_3=1,x_4=2)\) 4_ary
  - \((x_1=2,x_2=2,x_3=1,x_4=2)\) \(\rightarrow\) \((x \neq 1)\) and \(x_1=2\)
  - \((x_3=1,x_4=2)\) \(\rightarrow\) \((x \neq 1)\)
  - \((x_4=2)\) \(\rightarrow\) \((x \neq 1)\)
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.
Learning Issues

- Learning styles
  - Graph-based or context-based
  - i-bounded, scope-bounded
  - Relevance-based
- Non-systematic randomized learning
- Implies time and space overhead
- Applicable to SAT
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 $\tilde{a}_{i-1}[I_i]$.
      $iprev \leftarrow i$
   (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 \(\tilde{a}_{i-1}[J_i]\) and corresponding values
        \(iprev \leftarrow i\)
        (algorithm continues as in Fig. 6.7)
```

Figure 6.11: Conflict-directed backjump 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.
Complexity of backtrack-learning (improved)

- **Theorem:** Any backtracking algorithm using graph-based learning along d has a space complexity $O(nk^{w*(d)}$ 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 $O(nk^{w*(d)}$ deadends. There are at most $kn$ values between two successive deadends: $O(nk^{w*(d+1)}$ number of nodes in the search space. Since at most $O(2^{w*(d)}$ constraints-checks we get:
  - $O(n^2(2k)^{w*(d)+1}$

- Improved more: If we have $O(nk^{w*(d)}$ leaves, we have k to n times as many internal nodes, yielding between $O(nk^{w*(d+1)}$ and $O(n^2 k^{w*(d)}$ nodes.
Complexity of Backtrack-Learning for CSP

- The complexity of learning along d is time and space exponential in \( w^* (d) \):

  The number of dead-ends is bounded by \( O(n k^{w^* (d)}) \)

  Number of constraint tests per dead-end are \( O(e) \)

  \[ \text{Space complexity is } O(n k^{w^* (d)}) \]

  \[ \text{Time complexity is } O(n^2 e \cdot k^{w^* (d)}) \]

  \[ \text{Learning and backjumping: } O(n m e k^{w^* (d)}) \]

  m- depth of tree, e- number of constraints
Good caching:
Moving from one to all or counting
Summary:
time-space for constraint processing

- **Constraint-satisfaction**
  - Search with backjumping
    - Space: linear, Time: $O(\exp(\log n \ w^*))$
  - Search with learning no-goods
    - time and space: $O(\exp(w^*))$
  - Variable-elimination
    - time and space: $O(\exp(w^*))$

- **Counting, enumeration**
  - Search with backjumping
    - Space: linear, Time: $O(\exp(n))$
  - Search with no-goods caching only
    - space: $O(\exp(w^*))$ Time: $O(\exp(n))$
  - Search with goods and no-goods learning
    - Time and space: $O(\exp(\text{path-width}), O(\exp(\log n \ w^*)))$
  - Variable-elimination
    - Time and space: $O(\exp(w^*))$
Non-Systematic Randomized Learning

- Do search in a random way with interrupts, restarts, unsafe backjumping, **but record conflicts**.
- Guaranteed completeness.
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.
procedure SAT-CBJ-LEARN
Input: A CNF theory \( \varphi \), assigned variables \( \sigma \) over \( x_1, \ldots, x_{i-1} \), 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 \( x_i \leftarrow SELECTVALUE-CBJ \).
5. If \( x_i \) is null (no value returned), then
6. add \( J_{x_i} \) to \( \varphi \) (learning)
7. \( i_{prev} \leftarrow \) index of last variable in \( J_i \) (backjump)
8. \( J_i \leftarrow resolve(J_i, J_{prev}) \) (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 \( \text{CONSISTENT}(\sigma \cup x_i) \) then return \( \sigma \leftarrow \sigma \cup \{x_i\} \)
2. If \( \text{CONSISTENT}(\sigma \cup \neg x_i) \) 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. \( J_i \leftarrow resolve(\alpha, \beta) \).
5. Return \( x_i \) \leftarrow null (no consistent value)
end procedure

Figure 6.12: Algorithm SAT-CBJ-LEARN
Integration of algorithms

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

\begin{align*}
i & \leftarrow 1 \quad \text{(initialize variable counter)} \\
call \text{SELECT VARIABLE} \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 & \\
\quad \text{instantiate } x_i & \leftarrow \text{SELECT VALUE-FC-CBJ} & \\
\quad \text{if } x_i \text{ is null} & \quad \text{(no value was returned)} \\
\quad \quad \quad iprev & \leftarrow i \\
\quad \quad \quad i & \leftarrow \text{latest index in } J_i \quad \text{(backjump)} \\
\quad \quad \quad J_i & \leftarrow J_i \cup J_{iprev} - \{x_i\} \\
\quad \quad \quad \text{reset each } D'_k, k > i, \text{ to its value before } x_i \text{ was last instantiated} & \\
\quad \text{else} & \\
\quad \quad \quad i & \leftarrow i + 1 \quad \text{(step forward)} \\
\quad \quad \quad \text{call SELECT VARIABLE} \quad \text{(determine next variable)} \\
\quad \quad \quad D'_i & \leftarrow D_i \\
\quad \quad \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} & \\
\end{align*}

Figure 6.13: The main procedure of the FC-CBJ algorithm.
**subprocedure** SELECTVALUE-FC-CBJ

while $D'_i$ is not empty
  select an arbitrary element $a \in D'_i$, and remove $a$ from $D'_i$
  
  empty-domain $\leftarrow$ false
  
  for all $k$, $i < k \leq n$
    
    for all values $b$ in $D'_k$
      
      if not CONSISTENT($\tilde{a}_{i-1}, x_i = a, x_k = b$)
        
        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)
      
      empty-domain $\leftarrow$ true
    
  endfor
  
  if empty-domain $(\text{don't select } a)$
    reset each $D'_k$ and $j_k$, $i < k \leq n$, to status before $a$ was selected
  else
    return $a$
  
end while

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

end subprocedure
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>4</td>
<td>18</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.