Search Algorithms

Combinatorial Problem Solving (CPS)

Enric Rodríguez-Carbonell (based on materials by Javier Larrosa)

March 15, 2019
Basic Backtracking

function $\text{BT}(\tau, X, D, C)$

$\quad // \ \tau$: current assignment
$\quad // \ X$: vars ; $D$: domains; $C$: constraints

$\quad x_i := \text{Select}(X)$

$\quad \text{if } x_i = \text{nil} \ \text{then return } \tau$

$\quad \text{for each } a \in d_i \ \text{do}$

$\quad \quad \text{if } \text{Consistent}(\tau, C, x_i, a) \ \text{then}$

$\quad \quad \quad \sigma := \text{BT}(\tau \circ (x_i \mapsto a), X, D[d_i \rightarrow \{a\}], C)$

$\quad \quad \text{if } \sigma \neq \text{nil} \ \text{then return } \sigma$

$\quad \text{return nil}$

function $\text{Consistent}(\tau, C, x_i, a)$:

$\quad \text{for each } c \in C \ \text{s.t. } \text{scope}(c) \nsubseteq \text{vars}(\tau) \land \text{scope}(c) \subseteq \text{vars}(\tau) \cup \{x_i\}$

$\quad \quad \text{if } \neg c(\tau \circ (x_i \mapsto a)) \ \text{then return } \text{false}$

$\quad \text{return true}$
Improvements on Backtracking

- We say a (partial) assignment is **good** if it can be extended to a solution, **nogood** otherwise

- We say BT **makes a mistake** when it moves from a good assignment to a nogood one

- We say BT **recovers from a mistake** when it backtracks from a nogood assignment to a good one

- Shortcomings of BT (which are related to each other):
  - **BT detects very late when a mistake has been made**
    (\(\iff\) Look-ahead)
Basic Backtracking
## Basic Backtracking

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Q</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>Q</td>
</tr>
<tr>
<td>Q</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>Q</td>
</tr>
<tr>
<td>X</td>
<td>Q</td>
<td>X</td>
<td>Q</td>
<td>X</td>
</tr>
<tr>
<td>X</td>
<td>X</td>
<td>X</td>
<td>Q</td>
<td>X</td>
</tr>
<tr>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

**Diagram:**

```
(0,0) (0,1) (0,2) (0,3)
(1,0) (1,1) (1,2) (1,3)
(2,0) (2,1) (2,2) (2,3)
(3,0) (3,1) (3,2) (3,3)
```

**Grid:**

- **Black Moves:** Q
- **Red Moves:** X

**Objective:**

Complete the grid from (0,0) to (3,3) with the black and red moves. The path should form a valid backtracking solution.
### Basic Backtracking

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th>Q</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Q</td>
<td>Q</td>
<td>Q</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>Q</td>
</tr>
<tr>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>Q</td>
</tr>
<tr>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>X</td>
<td>Q</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>X</td>
<td>Q</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>X</td>
<td>X</td>
<td>X</td>
<td>Q</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

<p>| | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Improvements on Backtracking

- We say a (partial) assignment is **good** if it can be extended to a solution, **nogood** otherwise.

- We say BT **makes a mistake** when it moves from a good assignment to a nogood one.

- We say BT **recovers from a mistake** when it backtracks from a nogood assignment to a good one.

- Shortcomings of BT (which are related to each other):
  - BT detects very late when a mistake has been made ($\iff$ Look-ahead)
  - BT **may make again and again the same mistakes** ($\iff$ Nogood recording)
Basic Backtracking

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Q</td>
<td></td>
<td></td>
<td></td>
<td>Q</td>
</tr>
<tr>
<td></td>
<td>Q</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Q</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>X</td>
<td>X</td>
<td>Q</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>X</td>
<td>X</td>
<td>X</td>
<td>Q</td>
<td>X</td>
</tr>
<tr>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>Q</td>
</tr>
<tr>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>


Basic Backtracking
Basic Backtracking
Improvements on Backtracking

- We say a (partial) assignment is **good** if it can be extended to a solution, **nogood** otherwise

- We say BT **makes a mistake** when it moves from a good assignment to a nogood one

- We say BT **recovers from a mistake** when it backtracks from a nogood assignment to a good one

- Shortcomings of BT (which are related to each other):
  
  - BT detects very late when a mistake has been made  
    \[\Rightarrow \text{Look-ahead}\]
  
  - BT may make again and again the same mistakes  
    \[\Rightarrow \text{Nogood recording}\]
  
  - **BT is very weak recovering from mistakes**  
    \[\Rightarrow \text{Backjumping}\]
### Basic Backtracking

```
    Q  X  X
Q  X  X  X  X  X
Q  X  X  X  X  X  X
X  X  X  X  X  X  X  X  Q  X  X
X  X  X  X  X  X  X  Q  X  X  X
X  X  X  X  X  X  X  X  X  Q  X  X  X
X  X  X  X  X  X  X  X  X  Q  X  X  X
X  X  X  X  X  X  X  X  X  Q  X  X  X
X  X  X  X  X  X  X  X  X  Q  X  X  X
X  X  X  X  X  X  X  X  X  Q  X  X  X
X  X  X  X  X  X  X  X  X  Q  X  X  X
X  X  X  X  X  X  X  X  X  Q  X  X  X
X  X  X  X  X  X  X  X  X  Q  X  X  X
X  X  X  X  X  X  X  X  X  Q  X  X  X
X  X  X  X  X  X  X  X  X  Q  X  X  X
X  X  X  X  X  X  X  X  X  Q  X  X  X
X  X  X  X  X  X  X  X  X  Q  X  X  X
X  X  X  X  X  X  X  X  X  Q  X  X  X
X  X  X  X  X  X  X  X  X  Q  X  X  X
X  X  X  X  X  X  X  X  X  Q  X  X  X
X  X  X  X  X  X  X  X  X  Q  X  X  X
X  X  X  X  X  X  X  X  X  Q  X  X  X
X  X  X  X  X  X  X  X  X  Q  X  X  X
X  X  X  X  X  X  X  X  X  Q  X  X  X
X  X  X  X  X  X  X  X  X  Q  X  X  X
X  X  X  X  X  X  X  X  X  Q  X  X  X
X  X  X  X  X  X  X  X  X  Q  X  X  X
X  X  X  X  X  X  X  X  X  Q  X  X  X
X  X  X  X  X  X  X  X  X  Q  X  X  X
X  X  X  X  X  X  X  X  X  Q  X  X  X
X  X  X  X  X  X  X  X  X  Q  X  X  X
X  X  X  X  X  X  X  X  X  Q  X  X  X
X  X  X  X  X  X  X  X  X  Q  X  X  X
X  X  X  X  X  X  X  X  X  Q  X  X  X
X  X  X  X  X  X  X  X  X  Q  X  X  X
X  X  X  X  X  X  X  X  X  Q  X  X  X
X  X  X  X  X  X  X  X  X  Q  X  X  X
X  X  X  X  X  X  X  X  X  Q  X  X  X
X  X  X  X  X  X  X  X  X  Q  X  X  X
X  X  X  X  X  X  X  X  X  Q  X  X  X
X  X  X  X  X  X  X  X  X  Q  X  X  X
X  X  X  X  X  X  X  X  X  Q  X  X  X
X  X  X  X  X  X  X  X  X  Q  X  X  X
X  X  X  X  X  X  X  X  X  Q  X  X  X
X  X  X  X  X  X  X  X  X  Q  X  X  X
X  X  X  X  X  X  X  X  X  Q  X  X  X
```
```
Improvements on Backtracking

- A (partial) assignment is good if it can be extended to a solution, nogood otherwise

- BT makes a mistake when it moves from a good assignment to a nogood one

- BT recovers from a mistake when it backtracks from a nogood assignment to a good one

- Shortcomings of BT (which are related to each other):
  - BT detects very late when a mistake has been made ($\implies$ Look-ahead)
  - BT may make again and again the same mistakes ($\implies$ Nogood recording)
  - BT is very weak recovering from mistakes ($\implies$ Backjumping)
Look Ahead

- At each step BT checks consistency wrt. past decisions.
- This is why BT is called a **look-back** algorithm.
- **Look-ahead** algorithms use domain filtering / propagation: they identify domain values of unassigned variables that are not compatible with the current assignment, and prune them.
- When some domain becomes empty we can backtrack (as current assignment is incompatible with any value).
- One of the most common look-ahead algorithms: **Forward Checking (FC)**.
- Forward checking guarantees that all the constraints between already assigned variables and one yet unassigned variable are arc consistent.
**Forward Checking**

function $\text{FC}(\tau, X, D, C)$

// $\tau$: current assignment
// $X$: vars; $D$: domains; $C$: constraints

$x_i := \text{Select}(X)$

if $x_i = \text{nil}$ then return $\tau$

for each $a \in d_i$ do

// $\tau \circ (x_i \mapsto a)$ consistent

$D' := \text{LookAhead}(\tau \circ (x_i \mapsto a), X, D[d_i \mapsto \{a\}], C')$

if $\forall d_i' \in D' \, d_i' \neq \emptyset$ then

$\sigma := \text{FC}(\tau \circ (x_i \mapsto a), X, D', C')$

if $\sigma \neq \text{nil}$ then return $\sigma$

return nil

function $\text{LookAhead}(\tau, X, D, C)$

for each $x_j \in X - \text{vars}(\tau)$ do

for each $c \in C$ s.t. scope($c$) $\not\subseteq$ vars($\tau$) $\land$ scope($c$) $\subseteq$ vars($\tau$) $\cup$ \{x_j\}

for each $b \in d_j$ do

if $\neg c(\tau \circ (x_j \mapsto b))$ then remove $b$ from $d_j$

return $D$
Other Look-Ahead Algorithms

In general:

```plaintext
function DFS+Propagation(X, D, C)
   // X: vars; D: domains; C: constraints
   x_i := Select(X, D, C)
   if x_i = nil then return solution
   for each a ∈ d_i do
      D' := Propagation(x_i, X, D[d_i → {a}], C)
      if ∀ d'_i ∈ D' d'_i ≠ Ø then
         σ := DFS+Propagation(X, D', C)
         if σ ≠ nil then return σ
   return nil
```
Other Look-Ahead Algorithms

Many options for function Propagation:

- **Full AC** (results in the algorithm Maintaining Arc Consistency, MAC)
- **Full Look-Ahead** (binary CSP’s):
  
  ```
  function FL(x_i, X, D, C)
  // ...; x_i: last assigned; x_{i+1},...: unassigned
  for each j = i + 1...n do  // Forward checking
    Revise(x_j, c_{ij})
  for each j = i + 1...n, k = i + 1...n, j ≠ k do
    Revise(x_j, c_{jk})
  ```

- **Partial Look-Ahead** (binary CSP’s):
  
  ```
  function PL(x_i, X, D, C)
  // ...; x_i: last assigned; x_{i+1},...: unassigned
  for each j = i + 1...n do  // Forward checking
    Revise(x_j, c_{ij})
  for each j = i + 1...n, k = j + 1...n do
    Revise(x_j, c_{jk})
  ```
Variable/Value Selection Heuristics

```
function DFS+Propagation(X, D, C)
  // X: vars; D: domains; C: constraints
  x_i := Select(X, D, C) // variable selection is done here
  if x_i = nil then return solution
  for each a ∈ d_i do // value selection is done here
    D' := Propagation(X, D[d_i → {a}], C)
    if ∀d'_i ∈ D' d'_i ≠ ∅ then
      σ := DFS+Propagation(X, D', C)
      if σ ≠ nil then return σ
  return nil
```

- **Variable Selection**: the next variable to branch on
- **Value Selection**: how the domain of the chosen variable is to be explored
- **Choices at the top of the search tree** have a huge impact on efficiency
Variable/Value Selection Heuristics

- **Goal:**
  - Minimize no. of nodes of the search space *visited* by the algorithm

- **The heuristics can be:**
  - Deterministic vs. randomized
  - Static vs. dynamic
  - Local vs. shared
  - General-purpose vs. application-dependent
Variable Selection Heuristics

- Observation: given a partial assignment \( \tau \)

  (1) If there is a solution extending \( \tau \),
  then any variable is OK

  (2) If there is no solution extending \( \tau \),
  we should choose a variable that discovers that asap

- The most common situation in the search is (2)

- First-fail principle:
  choose the variable that leads to a conflict the fastest
Variable Heuristics in Gecode

- Deterministic dynamic local heuristics
  - ...
  - INT_VAR_SIZE_MIN(): smallest domain size
  - INT_VAR_DEGREE_MAX(): largest degree

- degree of a variable = number of constraints where it appears
Variable Heuristics in Gecode

- Deterministic dynamic shared heuristics
  - ...
  - INT_VAR_AFC_MAX(afc, t): largest AFC

- Accumulated failure count (AFC) of a constraint counts how often domains of variables in its scope became empty while propagating the constraint

- AFC of a variable is the sum of AFCs of all constraints where the variable appears
Variable Heuristics in Gecode

More precisely:

- After constraint propagation, the AFCs of all constraints are updated:
  - If some domain becomes empty while propagating $p$, $afc(p)$ is incremented by 1
  - For all other constraints $q$, $afc(q)$ is updated by a decay-factor $d$ ($0 < d \leq 1$): $afc(q) := d \cdot afc(q)$

- The AFC $afc(x)$ of a variable $x$ is then defined as:
  $afc(x) = afc(p_1) + \cdots + afc(p_n)$,
  where the $p_i$ are the constraints that depend on $x$.

- The AFC $afc(p)$ of a constraint $p$ is initialized to 1.
  So the AFC of a variable $x$ is initialized to its degree.
Variable Heuristics in Gecode

■ Deterministic dynamic shared heuristics
  ◆ ...  
  ◆ INT_VAR_ACTION_MAX(a, t): highest action

■ The action of a variable captures how often its domain has been reduced during constraint propagation
Variable Heuristics in Gecode

More precisely:

- After constraint propagation, the actions of all variables are updated:
  
  - If some value has been removed from the domain of $x$, $\text{act}(x)$ is incremented by 1: $\text{act}(x) := \text{act}(x) + 1$
  
  - Otherwise, $\text{act}(x)$ is updated by a decay-factor $d$ ($0 < d \leq 1$): $\text{act}(x) := d \times \text{act}(x)$
  
  - The action of a variable $x$ is initially 1
Value Selection Heuristics

- Observation: given a partial assignment $\tau$ and a var $x$
  
  (1) If there is no solution extending $\tau$, we can choose any value for $x$
  
  (2) If there is a solution extending $\tau$, then value chosen for $x$ should belong to a solution

- First-success principle: choose the value that has the most chances of being part in a solution
Branching Strategies

Branching tells how to extend nodes in search tree. Let:

- $x$ be a var chosen by the variable selection heuristic
- $v$ be a value chosen by the value selection heuristic

A node can be extended according to different strategies:

- **Enumeration:** a branch $x = v$ for each value $v \in d_x$

- **Binary Choice Points:**
  two branches, one with $x = v$ and the other with $x \neq v$

- **Domain Splitting:**
  two branches, one with $x \leq v$ and the other with $x > v$
  (or one with $x < v$ and the other with $x \geq v$)

The constraints that label the new edges (e.g., $x = v$) are called branching constraints
Branching in Gecode

[enumeration]
- INT_VALUES_MIN(): all values starting from smallest
- INT_VALUES_MAX(): all values starting from largest

[domain splitting]
- INT_VAL_SPLIT_MIN(): values not greater than $\frac{min+max}{2}$
- INT_VAL_SPLIT_MAX(): values greater than $\frac{min+max}{2}$

- ...
Branching in Gecode

[binary choice points]

- INT_VAL_RND(r): random value
- INT_VAL_MIN(): smallest value
- INT_VAL_MED(): greatest value not greater than the median
- INT_VAL_MAX(): largest value
- ...
Improvements on Backtracking

- A (partial) assignment is good if it can be extended to a solution, nogood otherwise

- BT makes a mistake when it moves from a good assignment to a nogood one

- BT recovers from a mistake when it backtracks from a nogood assignment to a good one

- Shortcomings of BT (which are related to each other):
  - BT detects very late when a mistake has been made (⇒ Look-ahead)
  - BT may make again and again the same mistakes (⇒ Nogood recording)
  - BT is very weak recovering from mistakes (⇒ Backjumping)
Nogood Recording

- We can add redundant constraints recording past mistakes to avoid repeating them in the future.
- This can reduce the search tree significantly.
- A deadend in the search tree is a node that does not lead to a solution.
- A nogood is a set of branching constraints inconsistent with any solution.
- In backtracking search, each deadend gives a nogood.
- Adding a constraint forbidding this nogood is too late for this node, but may be useful for pruning in the future.
- Nogood recording is a form of caching/memoization: store computations & reuse them instead of recomputing.
Nogood Recording

\[
\begin{array}{ccccccccccc}
& & & & & & & & & & Q \\
& & & & & & & & & X & X \\
& & & & & & & & Q & & \\
& & & & & & Q & & X & X & X \\
& & & & & X & X & X & X & & X \\
& & & & & X & X & X & X & & X \\
& & & & X & X & X & X & & X & & X \\
& & & & X & X & X & X & & X & & X \\
\end{array}
\]

\[
c_1 = 11, \quad c_3 = 6, \quad c_4 = 3, \quad c_5 = 1, \quad c_6 = 10, \\
c_7 = 7, \quad c_8 = 9, \quad c_9 = 2, \quad c_{10} = 5, \quad c_{11} = 8,
\]

is a nogood
\[ c_3 = 6, \quad c_4 = 3, \quad c_5 = 1, \]
\[ c_6 = 10, \quad c_7 = 7, \quad c_8 = 9 \]

is a nogood too (it is the actual reason for the conflict!)

\[ \neg(c_3 = 6 \land c_4 = 3 \land c_5 = 1 \land c_6 = 10 \land c_7 = 7 \land c_8 = 9) \] can be added
Assume that constraint propagation records, for each $a$ removed from the domain of a var $x$ at node $p = \{b_1, \ldots, b_j\}$, an explanation $\text{exp}(x \neq a) \subseteq p$ s.t. $\text{exp}(x \neq a) \cup \{x = a\}$ is a nogood (i.e., $\text{exp}(x \neq a)$ implies $x \neq a$).

$\text{exp}(x \neq a)$ accounts for the removal of $a$ from the domain of $x$.

<table>
<thead>
<tr>
<th>$Q$</th>
<th>1</th>
<th>1</th>
<th>$Q$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>12</td>
<td>2</td>
</tr>
<tr>
<td>12</td>
<td>2</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

- $\text{exp}(c_3 \neq 1)$ is $\{c_1 = 1\}$
- $\text{exp}(c_3 \neq 4)$ is $\{c_2 = 3\}$
- $\text{exp}(c_3 \neq 3)$ can be $\{c_1 = 1\}$ or $\{c_2 = 3\}$
Discovering Nogoods

Let \( p = \{b_1, \ldots, b_j\} \) be a deadend node in the search tree. The jumpback nogood for \( p \), denoted \( J(p) \), is defined as:

- If \( p \) is a leaf node and \( x \) is a variable whose domain has become empty, let \( D \) be its original domain. Then

\[
J(p) := \bigcup_{a \in D} \exp(x \neq a)
\]
Let \( p = \{b_1, \ldots, b_j\} \) be a deadend node in the search tree. The jumpback nogood for \( p \), denoted \( J(p) \), is defined as:

- If \( p \) is not a leaf node, let:
  - \( x \) be the selected variable,
  - \( a_1, \ldots, a_k \) all the possible values of \( x \) attempted by the branching strategy, each of which has failed
  - \( a'_1, \ldots, a'_l \) the pruned values of \( x \) by propagation

(sorry the domain of \( x \) is \( \{a_1, \ldots, a_k, a'_1, \ldots, a'_l\} \)). Then

\[
J(p) := \bigcup_{i=1}^{k} (J(p \cup \{x = a_i\}) - \{x = a_i\}) \cup \bigcup_{j=1}^{l} \exp(x \neq a'_j)
\]

The constraint

\[
\neg \bigwedge_{c \in J(p)} c
\]

forbids the nogood
Nogood Database Management

- If the nogood database becomes too large and too expensive to query, the search reduction may not pay off.

- Idea: keep only nogoods that are most likely to be useful.

- E.g., clean up the nogood database after every $M$ decisions, discarding a nogood if it has not been active enough (for instance, measured with the accumulated failure count).
Improvements on Backtracking

- A (partial) assignment is good if it can be extended to a solution, nogood otherwise

- BT makes a mistake when it moves from a good assignment to a nogood one

- BT recovers from a mistake when it backtracks from a nogood assignment to a good one

- Shortcomings of BT (which are related to each other):
  - BT detects very late when a mistake has been made (⇒ Look-ahead)
  - BT may make again and again the same mistakes (⇒ Nogood recording)
  - BT is very weak recovering from mistakes (⇒ Backjumping)
Backjumping

■ BT very weak recovering from mistakes as it backtracks chronologically (back to previously instantiated variable)

■ However, the reason for the conflict may not be the last assigned variable, but earlier!

■ Backjumping: backtrack to last choice with responsibility in the conflict

■ Backjumping may jump more than one tree-level, without missing solutions
Backjumping

\[
\begin{array}{cccccc}
& & & Q & & \\
Q & X & X & X & & \\
Q & X & X & X & X & \\
X & X & X & X & X & X \\
X & X & X & X & X & X \\
X & Q & X & X & X & X \\
X & X & X & X & X & X \\
X & X & X & X & X & X \\
X & X & X & X & X & X \\
X & X & X & X & X & X \\
\end{array}
\]

\[c_1 = 6, \ c_2 = 3, \ c_3 = 1, \ c_4 = 10, \ c_5 = 7, \ c_6 = 9, \ c_7 = 2, \ c_8 = 5, \ c_9 = 8\]
is a nogood
### Backjumping

\[
\begin{array}{|c|c|c|c|c|c|c|c|c|c|}
\hline
 & & & & & & & & & \\
\hline
 & & & & & & & & & Q \\
\hline
Q & X & X & X & & & & & & \\
\hline
Q & X & X & X & X & & & & & \\
\hline
X & X & X & X & & & & & & \\
\hline
X & X & X & & & X & X & & & \\
\hline
X & X & X & & & & X & Q & & \\
\hline
X & & & & & & & X & Q & X \\
\hline
X & & & & & & X & X & X & Q \\
\hline
X & & & & & X & X & X & X & X \\
\hline
X & & & & & X & X & X & X & X \\
\hline
X & & & & & X & X & X & X & X \\
\hline
X & X & X & X & X & X & X & X & X & X \\
\hline
\end{array}
\]

\[c_1 = 6, c_2 = 3, c_3 = 1, c_4 = 10, c_5 = 7, c_6 = 9\] is the reason for the conflict!

Retract \(c_6 = 9, c_7 = 2, c_8 = 5, c_9 = 8\)
Conflict-Directed Backjumping

- Assume node $p = \{b_1, \ldots, b_j\}$ of search tree is a deadend
- We must backtrack: retract a branching constraint from $p$
- Chronological backtracking would choose $b_j$
- **Conflict-Directed Backjumping (CBJ)** chooses the largest $i$ ($1 \leq i \leq j$) such that $b_i \in J(p)$, where $J(p)$ is the jumpback nogood for $p$
- CBJ jumps back in search tree up to $b_i$: retracts $b_i$ and all branching constraints after $b_i$
Randomization and Restarts

- Backtracking algorithms can be very sensitive to variable/value heuristics
- Early mistakes in the search tree have dramatic effects
- Idea:
  - Add randomization to the backtracking algorithm
  - Each run of the algorithm terminates either when:
    - a solution has been found; or
    - current run is too long, so search must be restarted
  - After each restart, a new run is executed that hopefully behaves better
Randomizing Heuristics

- Variable/value selection heuristics can be randomized by
  - Taking a random variable/value for breaking ties
  - Ranking variables/values with the chosen heuristic and randomly taking one of those “close” to the best
  - Randomly picking among a set of existing selection heuristics
When to Restart

- A restart strategy \( S = \{t_1, t_2, \ldots \} \) is an infinite sequence where each \( t_i \) is either a positive integer or \( \infty \).

- Randomized backtracking algorithm is run for \( t_1 \) “steps”. If no solution is found so far, a restart is applied, and the algorithm is run again for \( t_2 \) steps, and so on.

- In a fixed cutoff strategy, all \( t_i \) are equal.

- What is a “step” of computation?
  Several possibilities:
  - Number of backtracks
  - Number of visited nodes

- What are good restart strategies?
Restart Strategies: Luby Sequence

- Luby showed that, given full knowledge of the runtime distribution, the optimal strategy is given by $S_{t^*} = (t^*, t^*, \ldots)$, for some fixed cutoff $t^*$

- For the (mostly common) case in which there is no knowledge of the runtime distribution, Luby shows that any universal strategy of the form $S_u = (l_0, l_1, l_2, \ldots)$ where

\[
  l_i = \begin{cases} 
  N \cdot 2^{k-1} & \text{if } \exists k \text{ with } i = 2^k - 1 \\
  l_{i-2^k+1} & \text{if } \exists k \text{ with } 2^{k-1} \leq i < 2^k - 1
  \end{cases}
\]

for a fixed constant $N > 0$ has a behaviour that is “close” to that of the optimal strategy $S_{t^*}$
Restart Strategies: Luby Sequence

- For $N = 1$ Luby sequence is:

\[(1, 1, 2, 1, 1, 2, 4, 1, 1, 2, 1, 1, 2, 4, 8, \ldots)\]

- For $N = 512$:

![Luby-based restart sequence with initial 512](chart-image)
Restart Strategies: Geometric Seq.

- Walsh proposes a universal strategy $S_g = (1, r, r^2, \ldots)$ where the restart values are geometrically increasing.

- Works well in practice ($1 < r < 2$), but comes with no formal guarantees of its worst-case performance.

- It can be shown that the expected runtime of the geometric strategy can be arbitrarily worse than that of the optimal strategy.
Optimization Problems

- Often CSP’s have, in addition to the constraints to be satisfied, an objective function $f$ that must be optimized (maximized/minimized).

A CSP with an objective function is called a constraint optimization problem (COP).

- Wlog, let us assume there is a constraint $c = f(X)$, where $c$ is a variable, and the goal is to minimize $f$.

- A COP is solved by solving a sequence of CSP’s:
  - Initially an algorithm for solving CSP’s is used to find a solution $S$ that satisfies the constraints.
  - A constraint of the form $c < f(S)$ is then added, which excludes solutions that are not better than solution $S$.
  - The process is repeated until the resulting CSP has no solution: the last solution that was found is optimal.
Optimization Problems

- Let us write this procedure in pseudo-code
- Assume that $\min(f) \in \text{dom}(c)$

\[
\begin{align*}
  u &= \max(\text{dom}(c)); & \text{ // } u \text{ is an upper bound on } \min(f) \\
  S &= \text{solve}(C \land c \leq u - 1); \\
  \text{while } (S \neq \bot) \{ & \text{ // } \bot \text{ means "no solution"} \\
    u &= f(S); \\
    S &= \text{solve}(C \land c \leq u - 1); & \text{ // equivalent to solve } (C \land c < f(S)) \\
  \} & \text{ // on exit } \min(f) \text{ is } u
\end{align*}
\]

It is a linear search for $\min(f)$ in the domain of $c$ from the largest value in $\text{dom}(c)$ to the smallest one (until a solution is no longer found):

- Another approach is to do a linear search from the smallest value in $\text{dom}(c)$ to the largest one (until a solution is found):

\[
\begin{align*}
  l &= \min(\text{dom}(c)); & \text{ // } l \text{ is a lower bound on } \min(f) \\
  S &= \text{solve}(C \land c \leq l); \\
  \text{while } (S == \bot) \{ \\
    l &= l + 1; \\
    S &= \text{solve}(C \land c \leq l); \\
  \} & \text{ // on exit } \min(f) \text{ is } l
\end{align*}
\]
Yet another approach is to do a **binary search**:

\[
\begin{align*}
l &= \min(\text{dom}(c)); & // \ l \ is \ a \ lower \ bound \ on \ \min(f) \\
u &= \max(\text{dom}(c)); & // \ u \ is \ an \ upper \ bound \ on \ \min(f) \\
\textbf{while} \ (l \neq u) \ { \\
& \quad m = (l + u)/2; \\
& \quad S = \text{solve}(C \land c \leq m); \\
& \quad \textbf{if} \ (S == \bot) \ l = m + 1; \\
& \quad \textbf{else} \ u = f(S); & // \ f(S) \leq m \\
}\}
\]

// on exit \ \min(f) \ is \ l

Which approach is the best?
Yet another approach is to do a binary search:

\[
l = \min(\text{dom}(c)); \quad // \ l \text{ is a lower bound on } \min(f)
\]
\[
u = \max(\text{dom}(c)); \quad // \ u \text{ is an upper bound on } \min(f)
\]

while \((l \neq u)\) {
    \[
m = (l + u)/2;
    \]
    \[
S = \text{solve}(C \land c \leq m);
    \]
    if \((S == \bot)\) \:
        \[
l = m + 1;
        \]
    else \:
        \[
u = f(S); \quad // \ f(S) \leq m
        \]
}

// on exit \(\min(f)\) is \(l\)

Which approach is the best?

It depends on the problem.

Binary search is likely to perform less calls to solve, but unfeasible CSP’s may be more difficult to solve.