Search Algorithms

Combinatorial Problem Solving (CPS)

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

March 20, 2020

Basic Backtracking

function $\texttt{BT}(\tau, X, D, C)$
// τ: current assignment
// X: vars ; D: domains; C: constraints
    $x_i := \texttt{Select}(X)$
    if $x_i =$ **nil then return** τ
    for each $a \in d_i$ **do**
        if $\texttt{Consistent}(\tau, C, x_i, a))$ **then**
            $\sigma := \texttt{BT}(\tau \circ (x_i \mapsto a), X, D[d_i \to \{a\}], C)$
            if $\sigma \neq$ **nil then return** σ
    return nil

function $\texttt{Consistent}(\tau, C, x_i, a)$:
    for each $c \in C$ **s.t.** $\mathrm{scope}(c) \not\subseteq \mathrm{vars}(\tau) \wedge \mathrm{scope}(c) \subseteq \mathrm{vars}(\tau) \cup \{x_i\}$
        if $\neg c(\tau \circ (x_i \mapsto a))$ **then return** false
    return true

Improvements on Backtracking

- We say a (partial) assignment is good if it can be extended to a solution, a deadend otherwise
- We say BT makes a mistake when it moves from a good assignment to a deadend
- We say BT recovers from a mistake when it backtracks from a deadend to a good assignment
- Shortcomings of BT (which are related to each other):
  - **BT detects very late when a mistake has been made ($\Longrightarrow$ Look-ahead)**

Basic Backtracking

					Q						
		Q									
Q											
									Q		
						Q					
								Q			

Basic Backtracking

					Q						
		Q		X	X	X					
Q	X	X	X		X		X				
X	X	X		X	X			X	Q		
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
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	X

Basic Backtracking

					Q						
		Q		X	X	X					
Q	X	X	X		X		X				
X	X	X		X	X			X	Q		
X	X	X			X	Q		X	X	X	
X		X	X		X	X	X	Q	X	X	X
X	Q	X		X	X	X	X	X	X		X
X		X	X	Q	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

Improvements on Backtracking

- We say a (partial) assignment is good if it can be extended to a solution, a deadend otherwise
- We say BT makes a mistake when it moves from a good assignment to a deadend
- We say BT recovers from a mistake when it backtracks from a deadend to a good assignment
- Shortcomings of BT (which are related to each other):
  - BT detects very late when a mistake has been made ($\Longrightarrow$ Look-ahead)
  - **BT may make again and again the same mistakes ($\Longrightarrow$ Nogood recording)**

Basic Backtracking

					Q						
		Q		X	X	X					
Q	X	X	X		X		X				
X	X	X		X	X			X	Q		
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
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	X

Basic Backtracking

										Q	
					Q						
		Q									
Q											
									Q		
						Q					
								Q			

Basic Backtracking

											Q
					Q						
		Q									
Q											
									Q		
						Q					
								Q			

Improvements on Backtracking

- We say a (partial) assignment is good if it can be extended to a solution, a deadend otherwise
- We say BT makes a mistake when it moves from a good assignment to a deadend
- We say BT recovers from a mistake when it backtracks from a deadend to a good assignment
- Shortcomings of BT (which are related to each other):
  - BT detects very late when a mistake has been made ($\Longrightarrow$ Look-ahead)
  - BT may make again and again the same mistakes ($\Longrightarrow$ Nogood recording)
  - **BT is very weak recovering from mistakes ($\Longrightarrow$ Backjumping)**

Basic Backtracking

											Q
										X	X
					Q				X		X
		Q		X	X	X		X			X
Q	X	X	X		X		X				X
X	X	X		X	X	X		X	Q		X
X	X	X			X	Q		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	•	X	X	X	X
X	X	X	X	X	X	X	X	X	X	X	X

Improvements on Backtracking

- We say a (partial) assignment is good if it can be extended to a solution, a deadend otherwise
- We say BT makes a mistake when it moves from a good assignment to a deadend
- We say BT recovers from a mistake when it backtracks from a deadend to a good assignment
- Shortcomings of BT (which are related to each other):
  - BT detects very late when a mistake has been made ($\Longrightarrow$ Look-ahead)
  - BT may make again and again the same mistakes ($\Longrightarrow$ Nogood recording)
  - BT is very weak recovering from mistakes ($\Longrightarrow$ 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 $\texttt{FC}(\tau, X, D, C)$
// τ: current assignment
// X: vars; D: domains; C: constraints
    $x_i := \texttt{Select}(X)$
    if $x_i =$ **nil then return** τ
    for each $a \in d_i$ **do**
        // $\tau \circ (x_i \mapsto a)$ consistent
        $D' := \texttt{LookAhead}(\tau \circ (x_i \mapsto a), X, D[d_i \to \{a\}], C)$
        if $\forall_{d'_i \in D'}\ d'_i \neq \emptyset$ **then**
            $\sigma := \texttt{FC}(\tau \circ (x_i \mapsto a), X, D', C)$
            if $\sigma \neq$ **nil then return** σ
    return nil

function $\texttt{LookAhead}(\tau, X, D, C)$
    for each $x_j \in X - \mathrm{vars}(\tau)$ **do**
        for each $c \in C$ **s.t.** $\mathrm{scope}(c) \not\subseteq \mathrm{vars}(\tau) \wedge \mathrm{scope}(c) \subseteq \mathrm{vars}(\tau) \cup \{x_j\}$
            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:

function $\texttt{DFS+Propagation}(X, D, C)$
// X: vars; D: domains; C: constraints
$x_i := \texttt{Select}(X, D, C)$
if $x_i =$ **nil then return** solution
for each $a \in d_i$ **do**
$D' := \texttt{Propagation}(x_i, X, D[d_i \to \{a\}], C)$
if $\forall_{d'_i \in D'}\ d'_i \neq \emptyset$ **then**
$\sigma := \texttt{DFS+Propagation}(X, D', C)$
if $\sigma \neq$ **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)
  // $\ldots, x_{i-1}$: already assigned; x_i: last assigned; $x_{i+1}, \ldots$: unassigned
      for each $j = i+1 \ldots n$ **do** // Forward checking
          Revise(x_j, c_{ij})
      for each $j = i+1 \ldots n$, $k = i+1 \ldots n$, $j \neq k$ **do**
          Revise(x_j, c_{jk})

- Partial Look-Ahead (binary CSP's):

  function PL(x_i, X, D, C)
  // $\ldots, x_{i-1}$: already assigned; x_i: last assigned; $x_{i+1}, \ldots$: unassigned
      for each $j = i+1 \ldots n$ **do** // Forward checking
          Revise(x_j, c_{ij})
      for each $j = i+1 \ldots n$, $k = j+1 \ldots n$ **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 $a \in d_i$ **do** // value selection is done here
    $D' :=$ `Propagation`$(X, D[d_i \rightarrow \{a\}], C)$
    if $\forall_{d'_i \in D'}\ d'_i \neq \emptyset$ **then**
      $\sigma :=$ `DFS+Propagation`(X, D', C)
      if $\sigma \neq$ **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 τ

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

  (2) If there is no solution extending τ,
  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, $\mathrm{afc}(p)$ is incremented by 1
  - For all other constraints q, $\mathrm{afc}(q)$ is updated by a decay-factor d $(0 < d \leq 1)$: $\mathrm{afc}(q) := d \cdot \mathrm{afc}(q)$
- The AFC $\mathrm{afc}(x)$ of a variable x is then defined as: $\mathrm{afc}(x) = \mathrm{afc}(p_1) + \cdots + \mathrm{afc}(p_n)$, where the p_i are the constraints that depend on x.
- The AFC $\mathrm{afc}(p)$ 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,
    $\mathrm{act}(x)$ is incremented by 1: $\mathrm{act}(x) := \mathrm{act}(x) + 1$
  - Otherwise,
    $\mathrm{act}(x)$ is updated by a decay-factor d $(0 < d \leq 1)$:
    $\mathrm{act}(x) := d\ \mathrm{act}(x)$
  - The action of a variable x is initially 1

Value Selection Heuristics

- Observation: given a partial assignment τ and a var x

  (1) If there is no solution extending τ,
  we can choose any value for x

  (2) If there is a solution extending τ,
  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

- We say a (partial) assignment is good if it can be extended to a solution, a deadend otherwise
- We say BT makes a mistake when it moves from a good assignment to a deadend
- We say BT recovers from a mistake when it backtracks from a deadend to a good assignment
- Shortcomings of BT (which are related to each other):
  - BT detects very late when a mistake has been made ($\Longrightarrow$ Look-ahead)
  - BT may make again and again the same mistakes ($\Longrightarrow$ Nogood recording)
  - BT is very weak recovering from mistakes ($\Longrightarrow$ 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 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

										Q	
									X	X	X
					Q			X		X	
		Q		X	X	X	X			X	
Q	X	X	X		X	X	X			X	
X	X	X		X	X			X	Q	X	
X	X	X		X	X	Q		X	X	X	
X		X	X		X	X	X	Q	X	X	X
X	Q	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	Q	X	X	X	X
X	X	X	X	X	X	X	X	X	X	X	X

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

is a nogood

Nogood Recording

					Q						
		Q		X	X	X					
Q	X	X	X		X		X				
X	X	X		X	X			X	Q		
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
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	X

$$c_3 = 6, \quad c_4 = 3, \quad c_5 = 1,$$
$$c_6 = 10, \quad c_7 = 7, \quad c_8 = 9$$

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

$\neg(c_3 = 6 \wedge c_4 = 3 \wedge c_5 = 1 \wedge c_6 = 10 \wedge c_7 = 7 \wedge c_8 = 9)$ can be added

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

- We say a (partial) assignment is good if it can be extended to a solution, a deadend otherwise
- We say BT makes a mistake when it moves from a good assignment to a deadend
- We say BT recovers from a mistake when it backtracks from a deadend to a good assignment
- Shortcomings of BT (which are related to each other):
  - BT detects very late when a mistake has been made ($\Longrightarrow$ Look-ahead)
  - BT may make again and again the same mistakes ($\Longrightarrow$ Nogood recording)
  - BT is very weak recovering from mistakes ($\Longrightarrow$ 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

					Q				
		Q		X	X	X			
Q	X	X	X		X		X		
X	X	X		X	X			X	Q
X	X	X			X	Q		X	X
X		X	X		X	X	X	Q	X
X	Q	X		X	X	X	X	X	X
X	X	X	X	Q	X	X		X	X
X	X	X	X	X	X	X	Q	X	X
X	X	X	X	X	X	X	X	X	X

$c_1 = 6, c_2 = 3, c_3 = 1, c_4 = 10, c_5 = 7, c_6 = 9, c_7 = 2, c_8 = 5, c_9 = 8$
is a nogood

Backjumping

					Q				
		Q		X	X	X			
Q	X	X	X		X		X		
X	X	X		X	X			X	Q
X	X	X			X	Q		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	X	X	X	X	X

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

Retract $c_6 = 9, c_7 = 2, c_8 = 5, c_9 = 8$

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 ∞

- 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

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

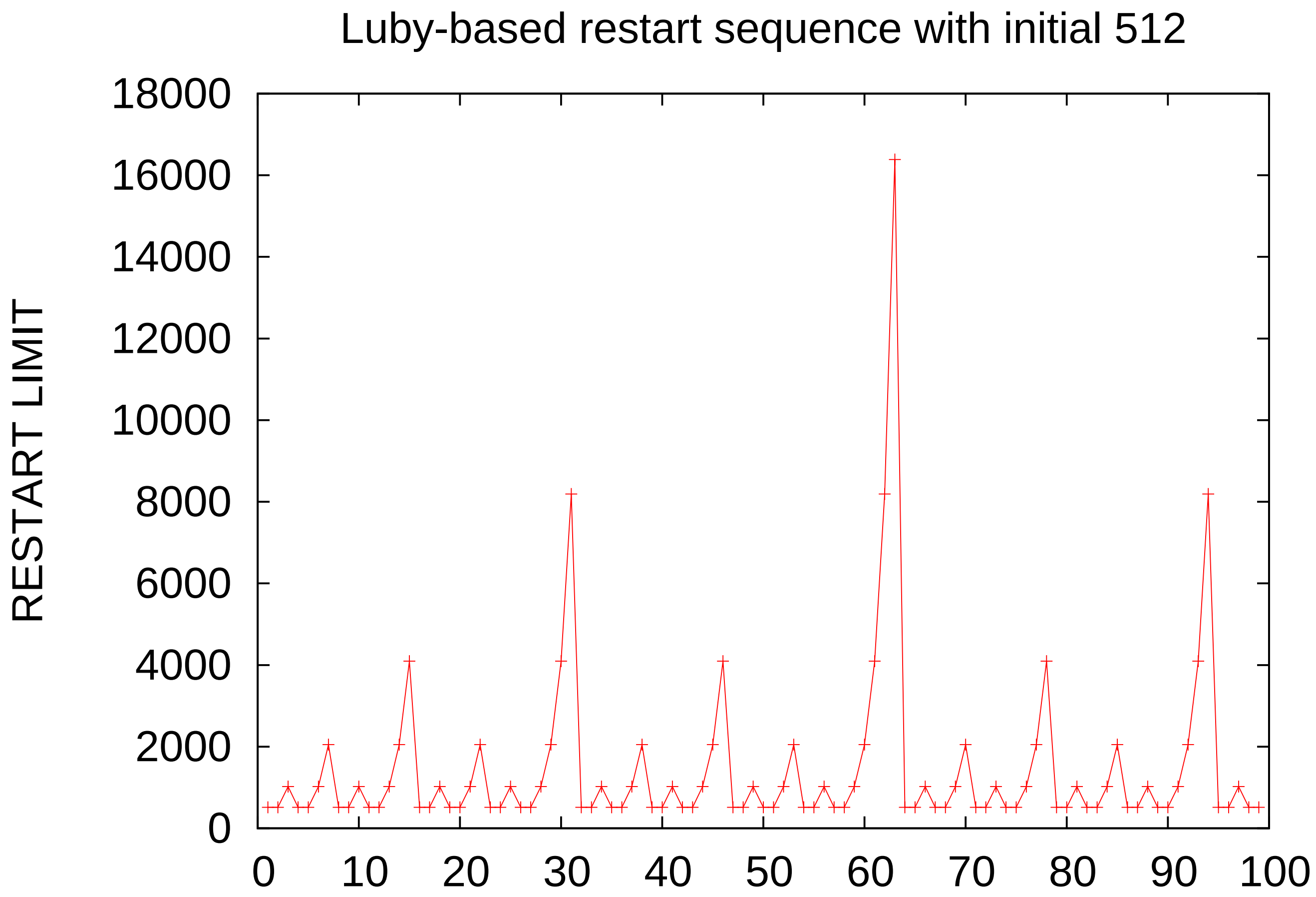
for a fixed constant $N > 0$ has a behaviour that is "close" to that of the optimal strategy S_{t^*}

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 $N = 512$:



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

- COP's can be 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 $\min(f) \in \mathsf{dom}(c)$

```
u = max(dom(c)); // u is an upper bound on min(f)
S = solve(C ∧ c ≤ u − 1);
while (S ≠ ⊥) {      // ⊥ means "no solution"
  u = f(S);
  S = solve(C ∧ c ≤ u − 1); // equivalent to solve(C ∧ c < f(S))
 } // on exit min(f) is u
```

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

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

```
l = min(dom(c)); // l is a lower bound on min(f)
S = solve(C ∧ c ≤ l);
while (S == ⊥) {
  l = l + 1;
  S = solve(C ∧ c ≤ l);
} // on exit min(f) is l
```

Optimization Problems

- Yet another approach is to do a binary search:

```
l = min(dom(c));   // l is a lower bound on min(f)
u = max(dom(c));   // u is an upper bound on min(f)
while (l ≠ u) {
  m = (l+u)/2;
  S = solve(C ∧ c ≤ m);
  if (S == ⊥) l = m+1;
  else u = f(S);  // f(S) ≤ m
}
// on exit min(f) is l
```

- Which approach is the best?

Optimization Problems

- Yet another approach is to do a binary search:

```
l = min(dom(c));   // l is a lower bound on min(f)
u = max(dom(c));   // u is an upper bound on min(f)
while (l ≠ u) {
  m = (l+u)/2;
  S = solve(C ∧ c ≤ m);
  if (S == ⊥) l = m+1;
  else u = f(S);  // f(S) ≤ 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.