LUND UNIVERSITY

LTH

FACULTY OF ENGINEERING

# Basics of Constraint Programming

KRZYSZTOF KUCHCINSKI
DEPT. OF COMPUTER SCIENCE, LTH

## Finite Domain Constraints

*“God made the integers, all else is the work of man.”*

Leopold Kronecker (1823-1891),
Jahresberichte der Deutschen Mathematiker Vereinigung.

## Outline

## Constraint Satisfaction Problem

## Constraint Satisfaction Problem

**Definition**

CSP is a 3-tuple $\mathcal{S} = (\mathcal{V}, \mathcal{D}, \mathcal{C})$ where

- $\mathcal{V} = \{x_1, x_2, \ldots, x_n\}$ is a finite set of variables, also called finite domain variables (FDVs),
- $\mathcal{D} = \{\mathcal{D}_1, \mathcal{D}_2, \ldots, \mathcal{D}_n\}$ is a finite set of domains, and
- $\mathcal{C}$ is a set of constraints restricting the values that the variables can simultaneously take.

## Constraint Satisfaction Problem (cont'd)

- *variable* $x_i$ has a finite set $\mathcal{D}_i \in \mathcal{P}(\mathbb{Z}) \setminus \emptyset$ of possible values, called a finite domain (FD).
  - example – $x :: \{1..10\}$,
  - example – $y :: \{23, 56\}$.
- A *constraint* $c(x_1, x_2, \ldots, x_n) \in \mathcal{C}$ between variables of $\mathcal{V}$ is a subset of the Cartesian product $\mathcal{D}_1 \times \mathcal{D}_2 \times \cdots \times \mathcal{D}_n$ that specifies which values of the variables are compatible with each other.

## Constraint Satisfaction Problem – Example

**Example**

- constraint $x < y$,
- finite domain varibales $x :: \{0..2\}$ and $y :: \{0..2\}$,
- the constraint is defined by the following 2-tuples of $\langle x, y\rangle$ values
  - $\{\langle 0, 1\rangle, \langle 0, 2\rangle, \langle 1, 2\rangle\}$

Comments

- *pruning* of the domain of $x$ and $y$
- pruning is achieved by executing *consistency* procedure.

## Constraint Satisfaction Problem (cont'd)

A *solution* $s$ to a CSP $\mathcal{S}$, denoted by $\mathcal{S} \models s$, is an assignment to <u>all variables</u> $\mathcal{V}$, such that it satisfies all the constraints.

- this assignment is often called *label* or *compound label*
- the process of finding a label is called *labeling*.
- *single solution*, *all solutions* or an *optimal solution*.
- an *optimal solution* $s$ to a CSP $\mathcal{S}$ is a solution $(\mathcal{S} \models s)$ which minimizes or maximizes a value $v$ assigned to a selected variable $x_i$ (cost function).

## Constraint Programming

- the enumeration of all possible combinations of values which are compatible with each other is difficult or impossible in practice,
- constraints are defined using equations, inequalities, combinatorial constraints, or programs defining compatible values,
- constraints define restrictions on the values which can be assigned to the constraint variables simultaneously,
- *unary*, *binary* and *n*-ary constraints,
- *binary constraint problems* – problems with unary and binary constraints only,
- all CSPs can be transformed to binary constraint problems.

## Transformation to Binary CSP

- assume a constraint on $k$ FDVs ($k > 2$) with variables $x_1, x_2, \ldots, x_n$.
- the transformation introduces a new *encapsulated variable* $v$ and replaces the constraint on $k$ variables by $k$ binary constraints on $v$ and $x_i$.
- the new variable $v$ domain captures a Cartesian product of the domains of individual variables.
- each of the newly created binary constraints connects $v$ and one of the $k$ FDVs.

## Transformation to Binary CSP (cont'd)

**Example**

- constraint $x + y = z$ with $x :: \{1..2\}, y :: \{1..2\}, z :: \{2..3\}$.

## Transformation to Binary CSP (cont'd)

**Example**

- constraint $x + y = z$ with $x :: \{1..2\}$, $y :: \{1..2\}$, $z :: \{2..3\}$.
- defined by 3-tuples of $\langle x, y, z \rangle$ values.
  - $\{\langle 1, 1, 2\rangle, \langle 1, 2, 3\rangle, \langle 2, 1, 3\rangle\}$

## Transformation to Binary CSP (cont'd)

**Example**

- constraint $x + y = z$ with $x :: \{1..2\}$, $y :: \{1..2\}$, $z :: \{2..3\}$.
- defined by 3-tuples of $\langle x, y, z\rangle$ values.
  - $\{\langle 1, 1, 2\rangle, \langle 1, 2, 3\rangle, \langle 2, 1, 3\rangle\}$
- new variable $v$ with domain $\{112, 123, 213\}$.

## Transformation to Binary CSP (cont'd)

**Example**

- constraint $x + y = z$ with $x :: \{1..2\}, y :: \{1..2\}, z :: \{2..3\}$.
- defined by 3-tuples of $\langle x, y, z \rangle$ values.
  - $\{\langle 1, 1, 2\rangle, \langle 1, 2, 3\rangle, \langle 2, 1, 3\rangle\}$
- new variable $v$ with domain $\{112, 123, 213\}$.
- new constraints
  - for $(v, x)$ $\{\langle 112, 1\rangle, \langle 123, 1\rangle, \langle 213, 2\rangle\}$
  - for $(v, y)$ $\{\langle 112, 1\rangle, \langle 123, 2\rangle, \langle 213, 1\rangle\}$
  - for $(v, z)$ $\{\langle 112, 2\rangle, \langle 123, 3\rangle, \langle 213, 3\rangle\}$

## Constraint Graph

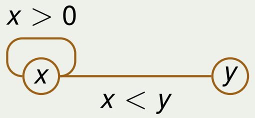
## Constraint Graph

- A binary constraint satisfaction problem can be depicted by a *constraint graph* (or constraint network).
- each node in this graph represents a finite domain variable,
- each arc represents a constraint between these variables,
- unary constraints have arcs originating and terminating at the same node (called self connecting arcs).

## Constraint Graph (cont'd)

**Example**

Consider two constraints $x < y$ and $x > 0$, i.e., one binary constraint and one unary constraint.



Constraint graph

- an extension of constraint graphs are *constraint hypergraphs*,
- a constraint hypergraph has nodes representing FDVs and hyperedges correspond to constraints of $n$ variables,
- constraint hypergrahs can represent general CSP problems.

## Constraint Entailment

## Constraint Entailment

The most basic question one can ask about a constraint is whether it is or it is not *entailed* or also called *satisfied*.

- each constraint can be in one of three states:
  - *satisfied*,
  - *not satisfied* ,
  - *'unknown'*.
- If the constraint is in the 'unknow' state a consistency algorithm for this constraint can be applied.

## Constraint Entailment (cont'd)

### Example

Consider constraint $x < y$ with different domains.

| | |
|---|---|
| $x :: \{1..5\}, y :: \{6..10\}$ | $x < y$ satisfied |
| $x :: \{6..10\}, y :: \{1..5\}$ | $x < y$ not satisfied |
| $x :: \{1..10\}, y :: \{1..10\}$ | $x < y$ unknow |
| | after application of propagation rules |
| | $x :: \{1..9\}, y :: \{2..10\}$ (still unknown) |

## Consistency Techniques

## Consistency Techniques

- used to remove these values from the domain of FDV that are incompatible with values of domains of other FDVs in a given constraint.
- consistency methods are implemented using *propagation rules* or *propagators*.
- propagators implement constraints.

## Properties of Propagators

- *correct*– no solution of a constraint is removed,

## Properties of Propagators

- *correct*– no solution of a constraint is removed,
- *assignment complete*– failure of a constraint is signaled at latest for the final assignments of values to FDVs,

## Properties of Propagators

- *correct*– no solution of a constraint is removed,
- *assignment complete*– failure of a constraint is signaled at latest for the final assignments of values to FDVs,
- *contracting*– domains of variables does not become larger after applying a given propagator (in practice we would like them to be narrowed), i.e., $p(D_x) \subseteq D_x$,

## Properties of Propagators

- *correct*– no solution of a constraint is removed,
- *assignment complete*– failure of a constraint is signaled at latest for the final assignments of values to FDVs,
- *contracting*– domains of variables does not become larger after applying a given propagator (in practice we would like them to be narrowed), i.e., $p(D_x) \subseteq D_x$,
- *monotonic*– propagator application to smaller domains will result in smaller domains than application to larger domains, i.e., $D_x \subseteq D_y \rightarrow p(D_x) \subseteq p(D_y)$.

## Properties of Propagators

- *correct*– no solution of a constraint is removed,
- *assignment complete*– failure of a constraint is signaled at latest for the final assignments of values to FDVs,
- *contracting*– domains of variables does not become larger after applying a given propagator (in practice we would like them to be narrowed), i.e., $p(D_x) \subseteq D_x$,
- *monotonic*– propagator application to smaller domains will result in smaller domains than application to larger domains, i.e., $D_x \subseteq D_y \rightarrow p(D_x) \subseteq p(D_y)$.

The propagator may also be

- *idempotent*– always computes the fixpoint, i.e., several applications of the same propagator produces always the same result, i.e., $p(p(D_x)) = p(D_x)$.

## Node Consistency

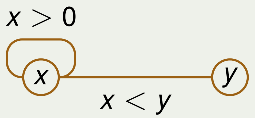
The *node consistency* is applied to unary constraints and removes values from the domain of FDV that are incompatible.

Node consistency algorithm

```
void NodeConsistency()
  for each $N \in nodes(G)$ do
    for each $v \in D_N$ do
      if the unary constraint on $N$ is inconsistent with $v$
        $D_N \leftarrow D_N \setminus \{v\}$
```

## Node Consistency Example

### Example



- $x :: \{0..10\}$ and $y :: \{0..10\}$,
- node consistency algorithm narrows $x$ to $\{1..10\}$,
- the self connecting arc can be removed from the constraint graph.

## Arc Consistency

- *arc consistency* is applied to the binary constraints of the constraint graph, i.e., constraints of two variables $x_i$ and $x_j$,
- the constraint graph is arc consistent if it is arc consistent for every arc $(x_i, x_j)$ in this graph,
- different algorithms exist, e.g., AC-1, AC-3.

## Arc Consistency

- *arc consistency* is applied to the binary constraints of the constraint graph, i.e., constraints of two variables $x_i$ and $x_j$,
- the constraint graph is arc consistent if it is arc consistent for every arc $(x_i, x_j)$ in this graph,
- different algorithms exist, e.g., AC-1, AC-3.

**Definition (Arc consistency)**

A variable $x_i$ is arc consistent relative to variable $x_j$ if and only if for every value $v_i \in D_i$ there exist a value $v_j \in D_j$ that is a consistent assignment $\langle v_i, v_j \rangle$, i.e., it is a solution for constraint $c(x_i, x_j)$.

## AC-1 algorithm

```
boolean revise($x, y$)
  reduced $\leftarrow$ false
  for each $u \in D_x$ do
    if there is no such $v \in D_y$ that $(u, v)$ is consistent
        $D_x \leftarrow D_x \setminus \{u\}$
        reduced $\leftarrow$ true
  return reduced

void AC-1()
  for each $\langle x, y\rangle \in arcs(G) \wedge x \neq y$
    $Q \leftarrow Q \cup \{\langle x, y\rangle, \langle y, x\rangle\}$
  do
    changed $\leftarrow$ false
    for each $(x, y) \in Q$ do
      changed $\leftarrow$ revise($x, y$) $\vee$ changed
  while changed
```

## AC-3 algorithm

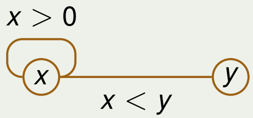
```
void AC-3()
  for each $\langle x, y\rangle \in arcs(G) \wedge x \neq y$
    $Q \leftarrow Q \cup \{\langle x, y\rangle, \langle y, x\rangle\}$
  while $Q \neq \emptyset$
    $Q \leftarrow Q \setminus (x, y)$
    if revise($x, y$)
      $Q \leftarrow Q \cup \{\langle z, x\rangle | (z, x) \in arcs(G) \wedge z \neq x \wedge z \neq y\}$
```

## Complexity of Arc Consistency

- AC-1 algorithm– worst case time complexity $\mathcal{O}(d^3 ne)$, where $d$ is domain size, $n$ the number of nodes in the constraint graph (FDVs) and $e$ is the number of edges with binary constraints,
- AC-3 has worst time complexity is $\mathcal{O}(d^3 e)$,
- more advanced arc consistency algorithms, such as, AC-4 has worst case time complexity $\mathcal{O}(d^2 e)$.
- AC-4 algorithm does not test many pairs $(u, v)$ which are already known from previous iterations to be consistent; it needs additional data structure to make it efficient.

## Arc Consistency Example

**Example**



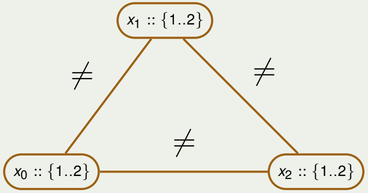
- $x :: \{1..10\}$ and $y :: \{0..10\}$.
- arc consistency algorithm pruning – $x :: \{1..9\}$ and $y :: \{2..10\}$.

# Arc Consistency Example

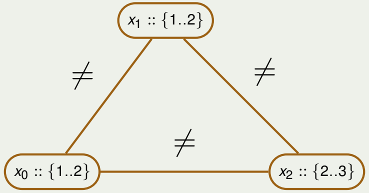
## Example

(a) arc consistency does not affect the domains of $x_0$, $x_1$ and $x_2$,

(b) FDVs cannot be pruned for the same reason.



(a) inconsistent constraint graph



(b) not pruned constraint graph

# Path Consistency

- *path consistency*, in addition to check arcs of the constraint graph between variables $x_i$ and $x_j$, further checks consistency of variables $x_i, x_k, x_j$ that form a path.

**Definition (Path Consistency)**

A two variable set $\{x_i, x_j\}$ is path consistent relative to variable $x_k$ if and only if for every consistent assignment $\langle v_i, v_j \rangle$, $v_i \in D_i$, $v_j \in D_j$ there is a value $v_k \in D_k$ such that the assignment $\langle v_i, v_k \rangle$ is consistent and $\langle v_k, v_j \rangle$ is consistent.
The constraint graph is path consistent if and only if for every binary constraint $c(x_i, x_j)$ and for every $k$ $(k \neq i, k \neq j)$, $c(x_i, x_j)$ is path consistent relative to $x_k$.

# PC-1 algorithm

```
boolean revise($(x, y), z$)
  for each $(u, v) \in D_x \times D_y$ do
    if there is no such $q \in D_z$ that $(u, q)$ and $(q, v)$ are
       consistent delete $(u, v)$ from constraint $c(x, y)$

void PC-1()
  do
    for k = 1..n
      for i = 1..n
        for j = 1..n
          revise($(x_i, x_j), x_k$)
  while there are changed constraints
```

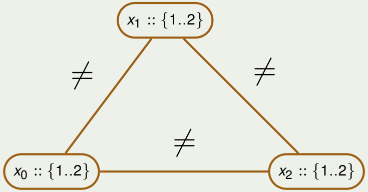
- worst case time complexity of this algorithm is $\mathcal{O}(d^5 n^5)$

# Path Consistency Example

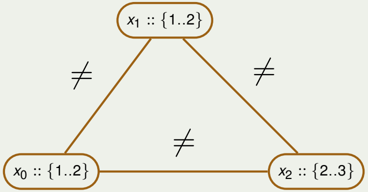
## Example

(a) constraints are inconsistent since there is no assignment for $x_2$ when $x_0 = 1, x_1 = 2$ or $x_0 = 2, x_1 = 1$ (the only allowed assignments of $x_0$ and $x_1$),

(b) it can prune domain of $x_2$ since the only allowed value for $x_2$ for two valid assignments to $x_0$ and $x_1$ is 3.



(a)



(b)

## Path Consistency Example

**Example**

- consider three constraints $x \leq y$, $y \leq z$ and $x \geq z$ with domains of $x$, $y$ and $z$ $\{0..2\}$,
- Arc consistency is not able to deduce that variables must be equal and have value 0, 1 or 2,
- Path consistency can find out that $y = z$ and $z = x$.

## *k*-consistency

- a constraint graph is *k*-consistent if and only if given any consistent instantiation of any $k - 1$ distinct variables, there exist an instantiation of any *k*th variable such that the *k* values taken together satisfy all of constraints among the *k* variables.
- arc consistency is equivalent to 2-consistency and path consistency to 3-consistency (if there exist non-binary constraints we need to extend path consistency to test ternary constraints as well),
- constraint graph is *strongly k-consistent* if and only if it is *j*-consistent for all $j \leq k$.

## Bounds Consistency

- arithmetical constraints, defined using equalities, inequalities and arithmetical operators, have well defined properties and can be implemented using efficient consistency methods,
- *bounds consistency* uses interval arithmetic to derive which values are consistent instead of considering all combinations of values which are allowed by a given constraint.
- in bounds consistency the FDV domain is approximated using a lower and upper bound,
- we use real number consistency of primitive constraints rather than integer consistency.
- the limitation of considering a lower and upper bounds is quite acceptable and it is often true in practice.

We denote a minimal and a maximal value in the domain of FDV $x$ as $\mathtt{min}(x)$ and $\mathtt{max}(x)$ respectively.

## Bounds Consistency

**Definition**

An arithmetic constraint $c$ is *bounds consistent* if for each variable $x$ of this constraint, there is:

- an assignment of *real* numbers, say $v_1, v_2, \ldots, v_k$ to remaining variables in $c$, say $x_1, x_2, \ldots, x_k$, such that $\min(x_j) \leq v_j \leq \max(x_j)$ for each $v_j$ and $x = \min(x), x_1 = v_1, \ldots, x_k = v_k$ is a solution of $c$, and
- an assignment of *real* numbers, say $v'_1, v'_2, \ldots, v'_k$ to $x_1, x_2, \ldots, x_k$, such that $\min(x_j) \leq v'_j \leq \max(x_j)$ for each $v'_j$ and $x = \max(x), x_1 = v'_1, \ldots, x_k = v'_k$ is a solution of $c$.

## Bounds Consistency

- bounds consistency are built using a functional rule of the form

$$x \ \mathtt{in} \ \{min \,..\, max\}$$

- it restricts the domain of variable $x$ to interval $\{min \,..\, max\}$.
- this rule can be defined using set intersection between min/max domain and the original domain of FDV

$$D_x \leftarrow \{min \,..\, max\} \cap D_x$$

- the rule detects creation of an empty domain for $D_x$ and notifies its propagator or a solver.

## Bounds Consistency (cont'd)

- the consistency for primitive arithmetic constraints can be defined using the defined functional rule and intervals computed using lower and upper bounds of FDVs' domains,
- a typical definition of a propagator for a constraint of three FDVs $x_1$, $x_2$ and $x_3$ is of the form

$$
\begin{array}{lll}
x_1 & \mathbf{in} & \{min_1 \,..\, max_1\} \\
x_2 & \mathbf{in} & \{min_2 \,..\, max_2\} \\
x_3 & \mathbf{in} & \{min_3 \,..\, max_3\}
\end{array}
$$

  where values $min_i$ and $max_i$ are new allowed minimal and maximal values for respective variables,
- the new values for all variables involved in bounds consistency computation are computed using *fixpoint iteration*, i.e., the computation is iterated until no further changes of FDVs domains occur.

## Bounds Consistency (cont'd)

**Example**

- the propagation rules for constraint $x < y$ are as follows.

$$x \ \mathtt{in} \ \{-\text{inf} \ .. \ \max(y) - 1\}$$
$$y \ \mathtt{in} \ \{\min(x) + 1 \ .. \ \text{inf}\}$$

- applied to $x :: \{0..10\}$ and $y :: \{0..10\}$ (assuming that the minimal and maximal integers allowed in our solver, denoted by -inf and inf, are -100000 and 100000 respectively).

$$x \ \mathtt{in} \ \{-100000 \ .. \ 9\}$$
$$y \ \mathtt{in} \ \{1 \ .. \ 100000\}$$

- result – $x :: \{0..9\}$ and $y :: \{1..10\}$.

## Bounds Consistency (cont'd)

### Example

- the propagation rules for constraint $x + y = z$

$$\begin{aligned} x\ &\mathbf{in}\ \{\min(z) - \max(y) \ ..\ \max(z) - \min(y)\} \\ y\ &\mathbf{in}\ \{\min(z) - \max(x) \ ..\ \max(z) - \min(x)\} \\ z\ &\mathbf{in}\ \{\min(x) + \min(y) \ ..\ \max(x) + \max(y)\} \end{aligned}$$

- When rules are applied to $x :: \{1..10\}$, $y :: \{1..10\}$ and $z :: \{1..10\}$ the result is obtained
$x :: \{1..9\}$, $y :: \{1..9\}$ and $z :: \{2..10\}$.

## Bounds Consistency Limitations

- bounds consistency has problems when numerical intervals contain “holes”, i.e., not all integers between minimal and maximal values are present in the domain.

**Example**

- constraint $x + 3 = y$ with domains $x :: \{1..3, 7..10\}$ and $y :: \{1..10\}$,
- propagation rules:

$$x \; \mathtt{in} \; \{\min(y) - 3 \;..\; \max(y) - 3\}$$
$$y \; \mathtt{in} \; \{\min(x) + 3 \;..\; \max(x) + 3\}$$

- produces $x :: \{1..3, 7\}$ and $y :: \{4..10\}$
- arc consistency produces $y :: \{4..6, 10\}$.

# Domain Consistency

- instead of using minimum and maximum values in the domain we can apply the required operation on the sub intervals.
- extention to functional rule to operate on domains and not only on intervals.

$$x \ \mathbf{in} \ \{D_y - Const\}$$
$$y \ \mathbf{in} \ \{D_x + Const\}$$

- operations "+" and "–" are defined to operate on an interval and a constant as explained in the algorithm

```
domain $out = \emptyset$;
for interval $i \in D_x$
  $out \leftarrow out \cup \{\min(i) + Const \ .. \ \max(i) + Const\}$
```

- similar solutions can be used for other primitive constraints, such as $x + y = z$.

## Generalized Consistency

- consistency methods can be implemented as specialized algorithms that are able to efficiently prune the domains of its FDVs.
- these algorithms can implement specific reasoning methods coming from different areas, such as operation research, geometry and combinatorics.
- solver can implement non-linear constraints (e.g., $x \cdot y = z$) as well as specialized combinatorial constraints (e.g., `cumulative` constraint).
- combinatorial constraints are specially interesting since we can build specific algorithms around them and obtain efficient pruning of the involved FDVs.

## Alldifferent Constraint

**Example**

`alldifferent` constraint is equivalent in pruning ability to imposing $\frac{n \cdot (n-1)}{2}$ constraints of the form $x_i \neq x_j, i \neq j$.

```
FDV[] V;
for (FDV $v \in V$)
  if $v$ has changed and has a single value
    for (FDV $w \in V \wedge v \neq w$)
      $w$ in $\{value(v)\}'$
```

where $\{value(v)\}'$ denotes a complement of a set containing a single value assigned to $v$.

## Solver Implementation

## Solver Implementation

- all FDVs and constraints are kept in a so called *constraint store*.
- the constraint store maintains all necessary data structures for the solver and provides access to all constraints consistency (propagators) and satisfiability procedures as well as access to FDVs and their actual domains,
- it also maintains solver state during search and makes it possible to organize an efficient backtracking.
- the solver implements a procedure that enforces consistency of all constraints registered in the constrained store – *propagation loop*.

## Propagation Loop

```
boolean solver.consistency()
  while ($Q \neq \emptyset$)
    c $\leftarrow$ fetch constraint from $Q$;
    try
      c.consistency();
      $modifiedFDVs \leftarrow \{x|$ modified by c.consistency()$\}$
      $constraintsToEvaluate \leftarrow \{c|x \in var(c), x \in modifiedFDVs\}$
      $Q \leftarrow Q \cup constraintsToEvaluate$
      if (c.satisfied())
        remove c from constraint store;
    catch (Fail exception)
      return false;
  return true;
```

## Propagation Loop (cont'd)

- queue of constraints is usually organized. as FIFO; the priority queue can be a complex implementation structure but offers a lot of flexibility for constraint scheduling for evaluation.
- constraints are added to queue *Q* when their domains changed; the solver can add all constraints with variables that were changed or a more selective strategy can be used.
- the solver assigns to each changed variable an event that indicates a reason for variable's change; typical events are:
  - *value*– the variable get a single value assigned,
  - *min*– the minimum value of a variable has been changed,
  - *max*– the maximum value of a variable has been changed,
  - *minmax*– the minimum and the maximum value of a variable has been changed, and
  - *all*– any change of a variable has occurred, e.g., removing a value from inside an interval.

## Search

- the propagation loop cannot, in general, deliver a solution for the constraint problem.
- the solver need to search for a solution or solutions.
- search is usually implemented using an algorithm based on the *constraint-and-generate* method– it assigns a new value to a selected FDV and calls propagation loop.
- if at some point inconsistency is detected (by obtaining an empty FD) the algorithm need to backtrack,
- backtracking annuls the results of the last decision by removing all values changed by this assignment and a new assignment is then performed.
- a simple depth-first-search algorithm is therefore well suited for this solving technique (in CLP community this method is called *labeling*).

## Backtracking

- backtracking search requires a method to restore the previous state of the constraint store, i.e., the state which we need to backtrack.
- three approaches to solve this problem:
  - *trailing*– the solver records only changes in the state and, if needed, undo this changes,
  - *copying*– the solver always copy the whole state (basically the constraint store) and, if needed, removes the old state and returns to the old one, and
  - *recomputing*– the solver always recomputes needed information from already made decisions.
- by far dominating approach is trailing.

## Organizing Backtracking

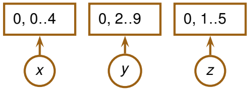
- search organizes the search space as a *search tree*,
- in every node of this tree a value is assigned to a variable and a decision whether the node will be extended or the search will be cut in this node is made,
- each node of this tree has assigned the *search level* – when the tree is extended the level is incremented and when the tree is cut the level is decremented.

## Organizing Backtracking (cont'd)

- each FDV points to the list of domains; the first element on this list is the current domain; the domain captures its actual domain, actual set of assigned constraints, and the *stamp* when the FDV has been assigned a new domain.
- when during the search a domain is pruned it is updated using a procedure that takes into account the stamp and the current level.
- if the level and the stamp are the same, the domain is updated directly, if the level is higher than the stamp a new domain with the new stamp is put at the top of the list.
- In case of backtracking, all domains with the with stamp equal to current level are removed.

## Search Example

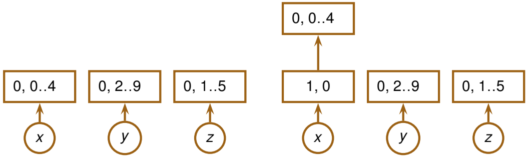
- three FDVs $x :: 0..4$, $y :: 2..9$, $z :: 1..5$ and constraints $x < z, y \geq z$,



(a) initial state

## Search Example

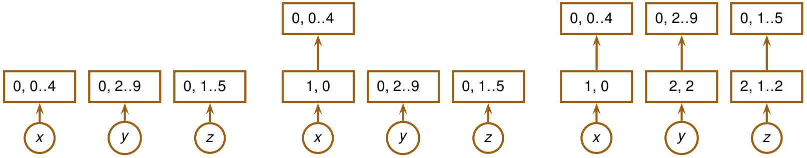
- three FDVs $x :: 0..4$, $y :: 2..9$, $z :: 1..5$ and constraints $x < z, y \geq z$,



(a) initial state (b) assignment $x = 0$

# Search Example

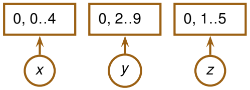
- three FDVs $x :: 0..4$, $y :: 2..9$, $z :: 1..5$ and constraints $x < z$, $y \geq z$,



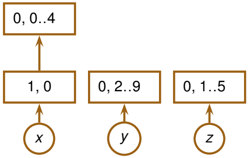
(a) initial state (b) assignment $x = 0$ (c) assignment $y = 2$

## Search Example

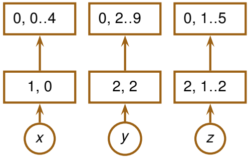
- three FDVs $x :: 0..4$, $y :: 2..9$, $z :: 1..5$ and constraints $x < z$, $y \geq z$,



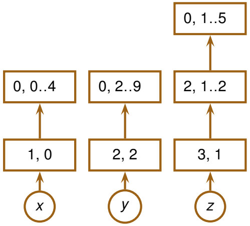
(a) initial state



(b) assignment $x = 0$



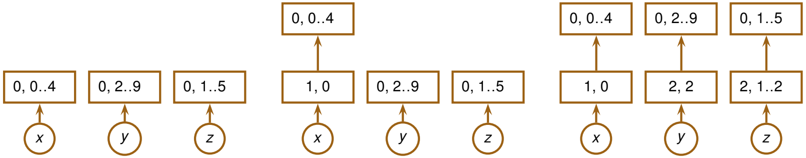
(c) assignment $y = 2$



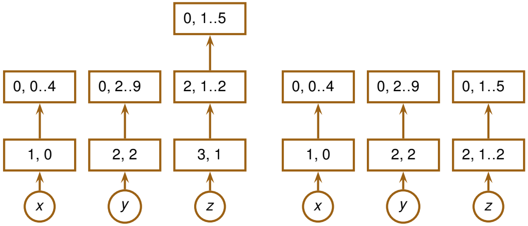
(d) assignment $z = 1$

## Search Example

- three FDVs $x :: 0..4$, $y :: 2..9$, $z :: 1..5$ and constraints $x < z$, $y \geq z$,



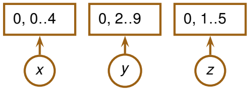
(a) initial state (b) assignment $x = 0$ (c) assignment $y = 2$



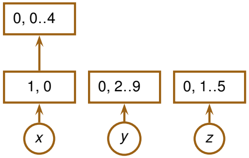
(d) assignment $z = 1$ (e) backtrack

# Search Example

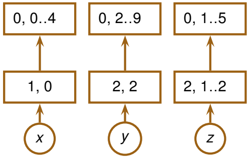
- three FDVs $x$ :: 0..4, $y$ :: 2..9, $z$ :: 1..5 and constraints $x < z$, $y \geq z$,



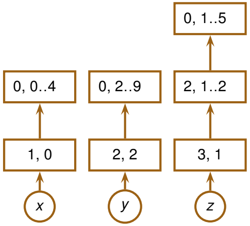
(a) initial state



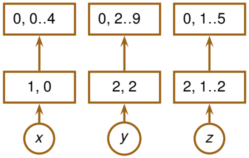
(b) assignment $x = 0$



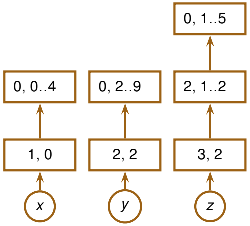
(c) assignment $y = 2$



(d) assignment $z = 1$



(e) backtrack



(f) assignment $z = 2$

## Simple Search

```
boolean solver.search(level, $\mathcal{V}$)
  if solver.consistency()
    if $\mathcal{V} \neq \emptyset$
      pick one variable $var$ from $\mathcal{V}$
      for each $v \in D_{var}$
        impose $var = v$
        if solver.search(level+1, $\mathcal{V} \setminus var$)
          return true
        else          // backtrack
          remove FDVs on trail with stamp=level
      return false    // no more values to assign for var
    else
      return true     // assignment for all FDVs found
  else
    return false      // inconsistent constraints
```

## Conclusions

## Conclusions

- there exist different methods to define constraints consistency,
- bounds consistency is usually used because it offers low complexity and good enough precision for most applications,
- solvers offer efficient methods for handling backtracking.