Computational Logic

Constraint Programming: Finite Domains

Introduction

- Constraint domains in which the possible values that a variable can take are restricted to a finite set.
- Examples: Boolean constraints, or integer constraints in which each variable is constrained to lie in within a finite range of integers.
- Widely used in constraint programming.
- Many real problems can be easily represented using constraint domains, e.g.: scheduling, routing and timetabling.
- They involve choosing amongst a finite number of possibilities.
- Commercial importance to many businesses: e.g. deciding how air crews should be allocated to aircraft flights.
- Developed methods by different research communities:
  - Arc and node consistency techniques (artificial intelligence).
  - Bounds propagation techniques (constraint programming).
  - Integer programming (operations research).

Constraint Satisfaction Problems

- In the artificial intelligence community, satisfaction of constraint problems over finite domains has been studied under the name of “constraint satisfaction problems”.
- A constraint satisfaction problem (CSP) consists on:
  - a constraint C over variables $x_1, \ldots, x_n$, where C be of the form $c_1 \wedge \cdots \wedge c_n$ (each c_i is a primitive constraint), and
  - a domain D that maps each variable x_i to a finite set of values, written $D(x_i)$, which is allowed to take.
- The CSP is understood to represent the constraint $C \wedge x_1 \in D(x_1) \wedge x_2 \in D(x_2) \wedge \cdots \wedge x_n \in D(x_n)$.
- Examples of CSPs: map coloring and N-queens problem.
- Binary CSPs: the primitive constraints have at most two variables (e.g. map coloring).

A Simple Backtracking Solver

- It is always possible to determine the satisfiability of a CSP by “brute force search”: trying all combinations of different values (finite number).
- However, this can be prohibitively expensive.
- Simplest techniques for determining satisfiability of an arbitrary CSP is chronological backtracking:
  - choosing a variable, and then, for each value in its domain, determining satisfiability of the constraint which results by replacing the variable with that value.
  - This is done by calling the backtracking algorithm recursively.
  - It uses the parametric function *satisfiable(c)*, which takes the primitive constraint *c* which involve no variables and returns *true* or *false* indicating whether *c* is satisfiable or not.

Chronological Backtracking Solver

INPUT: a CSP with constraint C and domain D.
OUTPUT: Returns $true$ if C is satisfiable (has one or more solutions), otherwise $false$.
METHOD:

back_solve(C, D)
    if $vars(C) \equiv \emptyset$ then return partial_satisfiable(C)
    else choose $x \in vars(C)$
        for each values $d \in D(x)$ do
            let C_1 be obtained from C by replacing x by d
            if partial_satisfiable(C_1) then
              if back_solve(C_1, D) then return $true$ endif
            endif
        endfor
        return $false$
    endif

Chronological Backtracking Solver (Contd.)

partial_satisfiable(C)
    let C be of the form $c_1 \wedge \cdots \wedge c_n$ (each c_i is a primitive constraint)
    for i := 1 to n do
        if $vars(c_i) \equiv \emptyset$ then
          if satisfiable(c_i) $\equiv false$ then return $false$ endif
        endif
    endfor
    return $true$

Exercise: apply the algorithm to the CSP with constraint

$$X < Y \wedge Y < Z$$

and domain D, such that $D(X) = D(Y) = D(Z) = \{1, 2\}$.

Node and Arc Consistency

- There are solvers for CSPs which have polynomial worst case complexity, but are incomplete.
- These solvers are based in the observation that if the domain for any variable in the CSP is empty, then the CSP is unsatisfiable.
- Idea: transform the CSP into an “equivalent” CSP but one in which the domains of the variables are decreased.
- “Equivalent” means that the constraints represented by the CSPs have the same set of solutions.
- If any of the domains become empty, then this CSP, and also the original, are unsatisfiable.
- These solvers work by considering each primitive constraint in turn, and using information about the domain of each variable in the constraint to eliminate values from the domains of the other variables.

Node and Arc Consistency (Contd.)

- The solvers are said to be "consistency" based since they propagate information about allowable domain values from one variable to another until the domains are "consistent" with the constraint.
- Two special domains may result after application of a consistency based solver:
  - ◇ False domain: if some variable in the domain is mapped to the empty set.
  - ◇ Valuation domain: if every variable is mapped to a singleton set.
- The function $satisfiable(C, D)$ takes a constraint C and a valuation domain D and returns *true* or *false*, indicating whether C is satisfiable or not under this valuation (i.e. whether the constraint evaluates to *true* or *false*). Examples:
  - ◇ The domain D_1, in which $D_1(X) = D_1(Y) = \{1, 2\}$ and $D_1(Z) = \emptyset$ is a false domain.
  - ◇ The domain D_2, in which $D_2(X) = \{1\}$, $D_2(Y) = \{2\}$, and $D_2(Z) = \{1\}$ is a valuation domain.
  - ◇ $satisfiable(C, D_2)$, where C is $X < Y \wedge Y < Z$, is *false*.

Node and Arc Consistency: Defi nitions

- A primitive constraint c is node consistent with domain D if either $|vars(c)| \neq 1$ or, if $vars(c) = \{x\}$, then, for each $d \in D(x)$, $\{x \mapsto d\}$ is a solution of c.
- A CSP with constraint $c_1 \wedge \cdots \wedge c_n$ and domain D is node consistent if each primitive constraint c_i is node consistent with D for $1 \leq i \leq n$.
- A primitive constraint c is arc consistent with domain D if either $|vars(c)| \neq 2$ or, if $vars(c) = \{x, y\}$, then,
  - for each $d_x \in D(x)$, there is some $d_y \in D(y)$, such that $\{x \mapsto d_x, y \mapsto d_y\}$ is a solution of c, and
  - for each $d_y \in D(y)$, there is some $d_x \in D(x)$, such that $\{x \mapsto d_x, y \mapsto d_y\}$ is a solution of c.
- A CSP with constraint $c_1 \wedge \cdots \wedge c_n$ and domain D is arc consistent if each primitive constraint c_i is arc consistent with D for $1 \leq i \leq n$.

Node and Arc Consistency: Examples

- The CSP with
  - constraint $X \neq Y \wedge X \neq Z \wedge Y \neq Z$ and
  - domain D, such that $D(X) = D(Y) = D(Z) = \{1, 2, 3\}$

  is node and arc consistent.
- It is also satisfiable (a solution is $X = 1, Y = 2, Z = 3$).
- The CSP with the same constraint and domain $D(X) = D(Y) = D(Z) = \{1, 2\}$ is also node and arc consistent.
- However, it is not satisfiable! (it has no solutions).
- The CSP with constraint $X < Y \wedge Y < Z$ and domain D, such that $D(X) = D(Y) = D(Z) = \{1, 2\}$ is node consistent but is not arc consistent.
- The CSP with constraint $X < Y \wedge Y < Z \wedge Z \leq 2$ and domain D, such that $D(X) = D(Y) = D(Z) = \{1, 2, 3\}$ is neither node nor arc consistent.

Algorithm for Node Consistency

INPUT: a CSP with constraint C and domain D_1.
OUTPUT: a domain D_2 such that the CSP with constraint C and domain D_2 is node consistent and is equivalent to the input CSP.
METHOD: D_2 := node_consistent(C, D_1).
node_consistent(C, D) (C is a constraint and D a domain)

    let C be of the form $c_1 \wedge \cdots \wedge c_n$ (each c_i is a primitive constraint)
    for i := 1 to n do
        D := node_consistent_primitive(c_i, D)
    endfor
    return D

node_consistent_primitive(c, D)

    if $|vars(c)| = 1$ then
      let $\{x\} = vars(c)$ (x is a variable)
      $D(x)$:= $\{d \in D(x) \mid \{x \mapsto d\}$ is a solution of $c\ \}$ (d is a domain value).
    endif
    return D

Algorithm for Arc Consistency

INPUT: a CSP with constraint C and domain D_1.
OUTPUT: a domain D_2 such that the CSP with constraint C and domain D_2 is arc consistent and is equivalent to the input CSP.
METHOD: D_2 := arc_consistent(C, D_1).

arc_consistent(C, D) (C is a constraint and D a domain)
    let C be of the form $c_1 \wedge \cdots \wedge c_n$ (each c_i is a primitive constraint)
    repeat $W := D$
        for i := 1 to n do D := arc_consistent_primitive(c_i, D) endfor
    until $W \equiv D$
    return D
arc_consistent_primitive(c, D)
    if $|vars(c)| = 2$ then
      let $\{x, y\} = vars(c)$ (x and y are variables)
      $D(x) := \{d_x \in D(x) \mid$ for some $d_y \in D(y)$, $\{x \mapsto d_x, y \mapsto d_y\}$ is a solution of $c\}$
      $D(y) := \{d_y \in D(y) \mid$ for some $d_x \in D(x)$, $\{x \mapsto d_x, y \mapsto d_y\}$ is a solution of $c\}$
    endif
    return D

Incomplete Node and Arc Consistency Solver

INPUT: a CSP with constraint C and domain D.
OUTPUT: Returns *true*, *false* or *unknown*. *true* if the CSP is satisfiable (has one or more solutions); *false* if the CSP is unsatisfiable (has no solutions); and *unknown* if the algorithm is not able of determining the satisfaction of the CSP.

METHOD:

arc_solv(C, D)
    D := node_arc_consistent(C, D)
    if D is a *false domain* then return *false*
    elseif D is a *valuation domain* then return $satisfiable(C, D)$
    else return *unknown*
    endif

node_arc_consistent(C, D)
    D := node_consistent(C, D)
    D := arc_consistent(C, D)
    return D

Incomplete Node and Arc Consistency Solver: Example

- Consider the CSP with constraint $X < Y \wedge Y < Z \wedge Z \leq 2$ and domain D, such that $D(X) = D(Y) = D(Z) = \{1, 2, 3\}$.

Complete Node and Arc Consistency Solver

INPUT: a CSP with constraint C and domain D.
OUTPUT: Returns $true$ if the CSP is satisfiable (has one or more solutions), otherwise return $false$.
METHOD: if back_arc_solv(C, D) returns $false$, then return $false$, otherwise return $true$.

back_arc_solv(C, D)
    D := node_arc_consistent(C, D)
    if D is a *false domain* then return $false$
    elseif D is a *valuation domain* then
        if $satisfiable(C, D)$ then return D else return $false$ endif
    endif
    Choose a variable x such that $|D(x)| \geq 2$
    for each value $d \in D(x)$ do
        D_1 := back_arc_solv($C \wedge x = d,\ D$)
        if $D_1 \neq false$ then return D_1 endif
    endfor
    return $false$

Hyper-Arc Consistency

- Node and Arc work well for pruning the domains in binary CSPs.
- However, they do not work well if the problem contains primitive constraints that involve more than two variables, since they are ignored when performing consistency checks.
- A primitive constraint c is *hyper-arc consistent* with domain D if for each variable $x \in vars(c)$ and each value $d \in D(x)$, there are values $d_1, \ldots, d_k$ for the remaining variables in c, say $x_1, \ldots, x_k$, such that $d_j \in D(x_j)$ for $1 \leq j \leq k$ and $\{x \mapsto d, x_1 \mapsto d_1, \ldots, x_k \mapsto d_k\}$ is a solution of c.
- A CSP with constraint $c_1 \wedge \cdots \wedge c_n$ and domain D is *hyper-arc consistent* if each primitive constraint c_i is *hyper-arc consistent* with D for $1 \leq i \leq n$.
- hyper-arc consistency is a true generalization of arc and node consistency.
- hyper-arc consistency is equivalent to arc (node) consistency in the case when a primitive constraint has two (one) variables.
- Unfortunately, testing for hyper-arc consistency can be very expensive even for fairly simple constraints that involve more than two variables.

Bounds Consistency

- The restriction to integer and arithmetic constraints allows us to define a new type of consistency: bounds consistency.
- A CSP is arithmetic if each variable in the CSP ranges over a finite domain of integers and the primitive constraints are arithmetic constraints.
- The most important class of CSPs (most problems of commercial interest).
- Two ideas behind bounds consistency.
  - ◇ Approximate the domain of a variable using a lower and upper bound.
  - ◇ Use real number consistency of primitive constraints rather than integer consistency.
- A range $[l..u]$ represents the set of values $\{l, l+1, \ldots, u\}$ if $l \leq u$, otherwise it represents the empty set.
- If D is a domain over integers, $min_D(x)$ and $max_D(x)$ are the minimum and maximum elements in $D(x)$ respectively.

Bounds Consistency (Contd.)

- An arithmetic primitive constraint c is `bounds consistent` with domain D if for each variable $x \in vars(c)$ there is:
  - an assignment of real numbers, say $d_1, d_2, \ldots, d_k$, to the remaining variables in c, say $x_1, x_2, \ldots, x_k$, such that:
    $\{x \mapsto min_D(x), x_1 \mapsto d_1, x_2 \mapsto d_2, \ldots, x_k \mapsto d_k\}$
    is a solution of c, and $min_D(x_j) \leq d_j \leq max_D(x_j)$ for each d_j.
  - another assignment of real numbers, say $d'_1, d'_2, \ldots, d'_k$, to the remaining variables in c, say $x_1, x_2, \ldots, x_k$, such that:
    $\{x \mapsto max_D(x), x_1 \mapsto d'_1, x_2 \mapsto d'_2, \ldots, x_k \mapsto d'_k\}$
    is a solution of c, and $min_D(x_j) \leq d'_j \leq max_D(x_j)$ for each d'_j.
- An arithmetic CSP with constraint $c_1 \wedge c_2 \wedge \cdots \wedge c_n$ and domain D is bounds consistent if each primitive constraint c_i is bounds consistent with D for $1 \leq i \leq n$.
- Since bounds consistency only depends on the upper and lower bounds of the domains of the variables, when testing bounds consistency, we need only consider domains that assign ranges to each variable.

Bounds Consistency: Example

- Consider the constraint: $X = 3Y + 5Z$, with domain D, where:

  $D(X) = [2..7], D(Y) = [0..2], D(Z) = [-1..2]$

  the constraint is NOT bounds consistent with D.
- Reason: if we consider $5Z = X - 3Y$, the left hand side can take a maximum value of 10, however the left hand side can take a maximum value of 7.
- Hence the domain D can be changed (changing the range of Z), obtaining another domain D_1 such that the constraint is bounds consistent with D_1

  $D_1(X) = [2..7], D_1(Y) = [0..2], D_1(Z) = [0..1]$

Propagation Rules

- Propagation rules methods: are efficient methods so that given a current range for each of the variables in a primitive constraint, they calculate a new range for each variable in the constraint, which makes the constraint to be bounds consistent with the new domain.
- Example: consider the simple constraint: $X = Y + Z$.
- It can be written in three forms:

  $X = Y + Z, Y = X - Z$ and $Z = X - Y$
- We can see that:

  $X \geq min_D(Y) + min_D(Z)$, $\quad X \leq max_D(Y) + max_D(Z)$

  $Y \geq min_D(X) - max_D(Z)$, $\quad Y \leq max_D(X) - min_D(Z)$

  $Z \geq min_D(X) - max_D(Y)$, $\quad Z \leq max_D(X) - min_D(Y)$
- It is easy to implement an algorithm for the propagation rules for the constraint $X = Y + Z$.

Propagation Rules for the Constraint $X = Y + Z$

INPUT: a domain D.
OUTPUT: a domain which is bounds consistent with the constraint $X = Y + Z$.
METHOD:

bounds_consistency_addition(D)

X_{min} := $maximum(min_D(X), min_D(Y) + min_D(Z))$
X_{max} := $minimum(max_D(X), max_D(Y) + max_D(Z))$
$D(X)$:= $\{d_X \in D(X) \mid X_{min} \leq d_X \leq X_{max}\}$
Y_{min} := $maximum(min_D(Y), min_D(X) - max_D(Z))$
Y_{max} := $minimum(max_D(Y), max_D(X) - min_D(Z))$
$D(Y)$:= $\{d_Y \in D(Y) \mid Y_{min} \leq d_Y \leq Y_{max}\}$
Z_{min} := $maximum(min_D(Z), min_D(X) - max_D(Y))$
Z_{max} := $minimum(max_D(Z), max_D(X) - min_D(Y))$
$D(Z)$:= $\{d_Z \in D(Z) \mid Z_{min} \leq d_Z \leq Z_{max}\}$
return D

Example of Propagation Rules for the Constraint $X = Y + Z$

- Consider the domain:

  $D(X) = [4..8], D(Y) = [0..3], D(Z) = [2..2]$

- We can determine that:

  $2 \leq X \leq 5$, using $min_D(Y) + min_D(Z) \leq X \leq max_D(Y) + max_D(Z)$,

  $2 \leq Y \leq 6$, using $min_D(X) - max_D(Z) \leq Y \leq max_D(X) - min_D(Z)$,

  $1 \leq Z \leq 8$, using $min_D(X) - max_D(Y) \leq Z \leq max_D(X) - min_D(Y)$.

- Therefore we can update the domain to:

  $D(X) = [4..5], D(Y) = [2..3], D(Z) = [2..2]$

  without removing any solutions to the constraint.

- If we apply the propagation rules to the new domain we obtain:

  $4 \leq X \leq 5 \quad 2 \leq Y \leq 3 \quad 1 \leq Z \leq 3$

- Thus, $X = Y + Z$ is bounds consistent with the new domain.

- We need only apply these propagation rules once to any domain to obtain a domain which is bounds consistent with the constraint $X = Y + Z$.

Propagation Rules for More Complicated Linear Arithmetic Constraints

- Example: consider the constraint: $4W + 3P + 2C \leq 9$
- We can rewrite this into three forms:
  $W \leq \frac{9}{4} - \frac{3}{4}P - \frac{2}{4}C, \quad P \leq \frac{9}{3} - \frac{4}{3}W - \frac{2}{3}C, \quad C \leq \frac{9}{2} - 2W - \frac{3}{2}P$
- and obtain the inequalities:
  $W \leq \frac{9}{4} - \frac{3}{4}min_D(P) - \frac{2}{4}min_D(C),$
  $P \leq \frac{9}{3} - \frac{4}{3}min_D(W) - \frac{2}{3}min_D(C),$
  $C \leq \frac{9}{2} - 2min_D(W) - \frac{3}{2}min_D(P)$
- Given the initial domain: $D(W) = [0..9], \quad D(P) = [0..9], \quad D(C) = [0..9]$
  we can determine that $W \leq \frac{9}{4}, \quad P \leq \frac{9}{3}, \quad C \leq \frac{9}{2}$
- Note that instead of $W \leq \frac{9}{4}$ we can use $W \leq \lfloor \frac{9}{4} \rfloor$ (i.e. $W \leq 2$, since W takes integer values only).
- Using the propagation rules (to be explained later), we update the domain:
  $D(W) = [0..2], \quad D(P) = [0..3], \quad D(C) = [0..4]$

Propagation Rules for the Constraint $4W + 3P + 2C \leq 9$

INPUT: a domain D.

OUTPUT: a domain which is bounds consistent with the constraint $4W + 3P + 2C \leq 9$.

METHOD:

bounds_consistency_addition(D)

W_{max} := $minimum(max_D(W), \lfloor \frac{9}{4} - \frac{3}{4}P - \frac{2}{4}C \rfloor)$

$D(W)$:= $\{d_W \in D(W) \mid d_W \leq W_{max}\}$

P_{max} := $minimum(max_D(P), \lfloor P\frac{9}{3} - \frac{4}{3}W - \frac{2}{3}C \rfloor)$

$D(P)$:= $\{d_P \in D(P) \mid d_P \leq P_{max}\}$

C_{max} := $minimum(max_D(C), \lfloor \frac{9}{2} - 2W - \frac{3}{2}P \rfloor)$

$D(C)$:= $\{d_C \in D(C) \mid d_C \leq C_{max}\}$

return D

Propagation Rules for Nonlinear Constraints

- Example: $X = mimimum\{Y, Z\}$
- Propagation rules follow directly from:

  $Y \geq min_D(X)$

  $Z \geq min_D(X)$

  $X \geq mimimum\{min_D(Y), min_D(Z)\}$

  $X \leq mimimum\{max_D(Y), max_D(Z)\}$

Bounds Consistency Algorithm

- INPUT: an arithmetic CSP with constraint C and domain D.
- OUTPUT: a domain D_1 such that the CSP with constraint C and domain D_1 is bounds consistent and is equivalent to the input CSP.
- METHOD: D_1 := bounds_consistent(C, D).

Bounds Consistency Algorithm (Contd.)

bounds_consistent(C, D) (C is a constraint and D a domain)

let C be of the form $c_1 \wedge \cdots \wedge c_n$ (each c_i is a primitive constraint)

C_0 := $\{c_1, \ldots, c_n\}$

while $C_0 \not\equiv \emptyset$ do

choose $c \in C_0$

C_0 := $C_0 \backslash \{c\}$

D_1 := bounds_consistent_primitive(c, D)

if D_1 is a false domain then return D_1 endif

for i := 1 to n do

if there exists $x \in vars(c_i)$ such that $D_1(x) \neq D(x)$ then C_0 := $C_0 \cup \{c_i\}$

endif

endfor

D := D_1

endwhile

return D

Bounds Consistency Algorithm (Contd.)

bounds_consistent_primitive(c, D)

- Applies the propagation rules for primitive constraint c to the domain D and returns the new domain.
- We assume that the original CSP has been transformed into a CSP containing only legitimate primitive constraints.

Bounds Consistency Algorithm: Example of Execution

- Consider the execution of this solver with the constraint:

  $X = Y + Z \wedge Y \neq Z$ and the domain

  $D(X) = [4..8]$, $D(Y) = [0..3]$, $D(Z) = [2..2]$.

- Initially C_0 is set to the set $\{X = Y + Z, Y \neq Z\}$.
- A primitive constraint c, say $X = Y + Z$, is removed from C_0.
- The function bounds_consistent_primitive(c, D) is called and returns the updated domain (evaluating the propagation rules for $X = Y + Z$):

  $D(X) = [4..5]$, $D(Y) = [2..3]$, $D(Z) = [2..2]$.

- Since the range of variable X has changed, the constraint $X = Y + Z$ is added to C_0 (although this is unnecessary since its propagation rules ensure consistency with respect to itself).
- Another primitive constraint, say $Y \neq Z$, is removed from C_0.

Bounds Consistency Algorithm: Example of Execution (Contd.)

- The call to the function bounds_consistent_primitive(c, D) removes the value 2 from the range of Y giving the updated domain:

  $D(X) = [4..5]$, $D(Y) = [3..3]$, $D(Z) = [2..2]$.

- The constraint $Y \neq Z$ is added to C_0, since the range of variable Y has changed.
- Another primitive constraint, say $X = Y + Z$, is removed from C_0, and its propagation rules are applied yielding the domain:

  $D(X) = [5..5]$, $D(Y) = [3..3]$, $D(Z) = [2..2]$.

- Since the range of variable X has changed, the constraint $X = Y + Z$ is added to C_0.
- Further processing of the of the constraints, $X = Y + Z$ and $Y \neq Z$, in C_0 does not change the domain so the function terminates returning this domain which is bounds consistent with the constraint.

Incomplete Bounds Consistency Solver

INPUT: an arithmetic CSP with constraint C and domain D.
OUTPUT: Returns *true*, *false* or *unknown*. *true* if the CSP is satisfiable (has one or more solutions); *false* if the CSP is unsatisfiable (has no solutions); and *unknown* if the algorithm is not able of determining the satisfaction of the CSP.

METHOD:

bounds_solv(C, D)
    D := bounds_consistent(C, D)
    if D is a *false domain* then return *false*
    elseif D is a *valuation domain* then return $satisfiable(C, D)$
    else return *unknown*
    endif

Incomplete Bounds Consistency Solver: Example

- Consider the CSP with constraint $X < Y \wedge Y < Z$ and domain D, such that $D(X) = D(Y) = D(Z) = [1..4]$.
- The call to bounds_consistent(C, D) sets C_0 to $\{X < Y, Y < Z\}$.
- Considering the primitive constraint $X < Y$ we obtain:

  $D(X) = [1..3]$, $D(Y) = [2..4]$, $D(Z) = [1..4]$.
- The constraint $X < Y$ is added to C_0.
- Considering $Y < Z$ we obtain:

  $D(X) = [1..3]$, $D(Y) = [2..3]$, $D(Z) = [3..4]$.
- The constraint $Y < Z$ is added to C_0.

Incomplete Bounds Consistency Solver: Example (Contd.)

- Reconsidering the constraint $X < Y$ we obtain:
  $D(X) = [1..2]$, $D(Y) = [2..3]$, $D(Z) = [3..4]$.
- The constraint $X < Y$ is added to C_0.
- Further processing of the of the constraints, $X < Y$ and $Y < Z$, in C_0 does not change the domain so the function bounds_consistent returns:
  $D(X) = [1..2]$, $D(Y) = [2..3]$, $D(Z) = [3..4]$.
- Thus, bounds_solv returns *unknown*.

Complete Bounds Consistency Solver

INPUT: an arithmetic CSP with constraint C and domain D.
OUTPUT: Returns $true$ if the CSP is satisfiable (has one or more solutions), otherwise return $false$.
METHOD: if back_bounds_solv(C, D) returns $false$, then return $false$, otherwise return $true$.

back_bounds_solv(C, D)
    D := bounds_consistent(C, D)
    if D is a *false domain* then return $false$
    elseif D is a *valuation domain* then
        if $satisfiable(C, D)$ then return D else return $false$ endif
    endif
    Choose a variable x such that $|D(x)| \geq 2$
    for each value $d \in D(x)$ do
        D_1 := back_bounds_solv($C \wedge x = d,\ D$)
        if $D_1 \neq false$ then return D_1 endif
    endfor
    return $false$

Complete Bounds Consistency Solver: Example

- Consider the previous CSP (with constraint $X < Y \wedge Y < Z$ and domain D, such that $D(X) = D(Y) = D(Z) = [1..4]$).
- The call to bounds_consistent(C, D) returns:

  $D(X) = [1..2]$, $D(Y) = [2..3]$, $D(Z) = [3..4]$.
- The (complete) solver selects a variable, say Z, and tries different values in its domain.
- First, the (complete) solver calls itself recursively with the constraint:

  $X < Y \wedge Y < Z \wedge Z = 3$

  and the domain:

  $D(X) = [1..2]$, $D(Y) = [2..3]$, $D(Z) = [3..4]$.
- The function bounds_consistent is evaluated with this new constraint and domain.

Complete Bounds Consistency Solver: Example

- Examining $Z = 3$ gives $D(Z) = [3..3]$.
- Examining $Y < Z$ gives $D(Y) = [2..2]$.
- Examining $X < Y$ gives $D(X) = [1..1]$.
- No more propagation is performed, so the resulting domain is:
  $D(X) = [1..1]$, $D(Y) = [2..2]$, $D(Z) = [3..3]$.
- This is returned by bounds_consistent and, since this is a valuation domain which satisfies the constraint, the solver back_bounds_solv returns *true*.

Example Problem: The Smuggler's Knapsack

- Knapsack of limited capacity: 9 units.
- It can smuggle bottles of whiskey of size 4 units, bottles of perfume of size 3 units, and cartons of cigarettes of size 2 units.
- The profit for a bottle of whiskey, a bottle of perfume, and a cartons of cigarettes are 15 dollars, 10 dollars, and 7 dollars respectively.
- The smuggler will only take a trip if he makes a profit of 30 dollars or more.
- What can he take ?

Example Problem: The Smuggler's Knapsack

- Constraint: $4W + 3P + 2C \leq 9 \wedge 15W + 10P + 7C \geq 30$
- Domain: $D(W) = D(P) = D(C) = [0..9]$
- Using the complete bounds propagation solver, we begin by calling the bounds_consistent function with this constraint and domain. This gives the domain: $D(W) = [0..2], D(P) = [0..3], D(C) = [0..4]$
- Then, choosing a branch on W, we first try adding $W = 0$.
- Applying bounds_consistent returns: $D(W) = [0..0], D(P) = [1..3], D(C) = [0..3]$
- Now, choosing to branch on P, we add the constraint $P = 1$. Applying bounds_consistent we get:

  $D(W) = [0..0], D(P) = [1..1], D(C) = [3..3]$
- So, we have found a solution: $W = 0$, $P = 1$, $C = 3$.
- Note: this is not the optimal solution!

Generalized Consistency

- We have seen three consistency based approaches to solving CSPs: arc, node and bounds consistency.
- These can be combined with each other and also with specialized consistency methods for “complex” primitive constraints:
  - For constraints involving only two variables, we can use the stronger arc consistency tests to remove values from the domain.
  - For constraints involving more than two variables, we can use the weaker, but more efficiently computable, bounds consistency approach.
- One of the weaknesses of the consistency based approaches is that primitive constraints are examined in isolation from each other.
- Sometimes, knowledge about other primitive constraints can dramatically improve domain pruning.
- For this reason, it is common to provide “complex” primitive constraints which are understood as conjunction of simpler primitive constraints but which have specialized propagation rules.

Generalized Consistency (Contd.)

- For example, the specialized "primitive" constraint:

  alldifferent($[V_1, \ldots, V_n]$)

  holds whenever each of the variables $V_1, \ldots, V_n$ takes a different value from the others.

- Instead of this constraint, we could use a conjunction of primitive constraints.

- For example, the primitive constraint alldifferent($[X, Y, Z]$) can be replaced by: $X \neq Y \wedge X \neq Z \wedge Y \neq Z$, but this is a weaker approach.

- Example: the constraint

  $X \neq Y \wedge X \neq Z \wedge Y \neq Z$

  with domain

  $D(X) = [1..2], D(P) = [1..2], D(C) = [1..2]$

  has no solutions (there are two possible values for the three variables).

- But, arc consistency techniques cannot determine this!

A Consistency Method for the alldifferent Primitive Constraint

alldifferent_consistent_primitive(c, D) (c is a single alldifferent primitive constraint)

let c be of the form alldifferent(V)
while exists $v \in V$ with $D(v) = \{d\}$ for some d
V := $V - \{v\}$
for each $v' \in V$ do $D(v')$:= $D(v') - \{d\}$ endfor
endwhile
nv := $|V|$
r := $\emptyset$
for each $v \in V$ do r := $r \cup D(v)$ endfor
if $nv > |r|$ then return *false* endif
return D

Optimization for Arithmetic CSPs

- For many problems, the aim is not simply to find any solution, but rather, to find the optimal solution.
- The simplest approach to finding an optimal solution to an arithmetic CSP is to make use of a complete solver for these problems, and use it iteratively to find better and better solutions to the problems:
  - Use the solver to find any solution to the CSP.
  - Add a constraint to the problem which excludes solutions that are not better than this solution.
  - The new constraint is solved recursively, giving rise to a better solution.

Integer Optimizer Based on Retrying

INPUT: an arithmetic CSP with constraint C and domain D and an arithmetic expression f which is the objective function.
OUTPUT: an optimal solution θ or false if the CSP is unsatisfiable.
METHOD: The answer is the result of evaluating retry_int_opt($C, D, f, false$).

retry_int_opt(C, D, f, θ_{best}) (θ_{best} is either a solution or false)
    D_{val} := int_solv(C, D) (D_{val} is a valuation domain or false)
    if $D_{val} \equiv false$ then
        return θ_{best}
    else
        let θ be the solution corresponding to D_{val}
        return retry_int_opt($C \wedge f < \theta(f), D, f, \theta$)
    endif

Integer Optimizer Based on Backtracking

INPUT: an arithmetic CSP with constraint C and domain D and an arithmetic expression f which is the objective function.
OUTPUT: an optimal solution θ or false if the CSP is unsatisfiable.
METHOD: The answer is the result of evaluating back_int_opt($C, D, f, false$).

back_int_opt(C, D, f, θ_{best}) (θ_{best} is either a solution or false)

D := int_consistent(C, D)
if D is a false domain then return θ_{best}
elseif D is a valuation domain then return the solution corresponding to D
endif
choose a variable $x \in varc(C)$ for which $|D(x)| \geq 2$
W := $D(x)$
for each $d \in W$ do
    if $\theta_{best} \not\equiv false$ then c := $f < \theta_{best}(f)$ else c := true endif
    θ_{best} := back_int_opt($C \wedge c \wedge x = d, D, f, \theta_{best}$)
endfor
return θ_{best}