Constraint Programming
Arc-consistency and MAC

Hadrien Cambazard

(ROSP)

Arc-consistency: AC1 – AC3

Removes values of $D(x_i)$ that are not consistent with c_{ij}

Algorithm 1 revise(a variable x_i, a binary constraint c_{ij})

```
1: change ← false
2: for v_i in D(x_i) do
3:     if there does not exist a support v_j in D(x_j) for v_i (i.e (v_i, v_j) ∉ R(c)) then
4:         remove v_i from D(x_i)
5:         change ← true
6: return change
```

Arc-consistency: AC1 – AC3

Removes values of $D(x_i)$ that are not consistent with c_{ij}

Algorithm 1 revise(a variable x_i, a binary constraint c_{ij})

```
1: change ← false
2: for v_i in D(x_i) do
3:     if there does not exist a support v_j in D(x_j) for v_i (i.e (v_i, v_j) ∉ R(c)) then
4:         remove v_i from D(x_i)
5:         change ← true
6: return change
```

Algorithm 2 AC1

```
1: change ← true
2: while change do
3:     change ← false
4:     for x_i ∈ X and c_ij ∈ C do
5:         if (revise(x_i, c_ij)) then change ← true
6:         if (D(x_i) = ∅) then return false;
7: return true
```

Arc-consistency: AC1 – AC3

Removes values of $D(x_i)$ that are not consistent with c_{ij}

Algorithm 1 revise(a variable x_i, a binary constraint c_{ij})

```
1: change ← false
2: for v_i in D(x_i) do
3:     if there does not exist a support v_j in D(x_j) for v_i (i.e (v_i, v_j) ∉ R(c)) then
4:         remove v_i from D(x_i)
5:         change ← true
6: return change
```

Algorithm 2 AC1

```
1: change ← true
2: while change do
3:     change ← false
4:     for x_i ∈ X and c_ij ∈ C do
5:         if (revise(x_i, c_ij)) then change ← true
6:         if (D(x_i) = ∅) then return false;
7: return true
```

Algorithm 3 AC3

```
1: Q ← {(x_i, c_ij) | c_ij ∈ C}
2: while Q ≠ ∅ do
3:     select and remove (x_i, c_ij) from Q
4:     if revise(x_i, c_ij) then
5:         if D(x_i) = ∅ then return false;
6:         else Q ← Q ∪ {(x_k, c_ki) | c_ki ∈ C ∧ k ≠ i}
7: return true
```

The friendship view of Arc-consistency

(borrowed and adapted from Romuald Debruyne)

- AC = **I** (a value) **stay** (in the domain) **if I have at least one friend** (a support) **in each neighboring** (link by a constraint) **building** (domain)

- AC1 = A person leaving sends a letter to **everybody**
  "I am leaving"
  - everyone check they still have a friend in each neighboring building.

- AC3 = A person leaving sends a letter to **the neighboring buildings**
  "I am leaving building n°i"
  - Only the neighbors of building i check if they still have a friend in building n°i

The friendship view of Arc-consistency

(borrowed and adapted from Romuald Debruyne)

- AC = **I** (a value) **stay** (in the domain) **if I have at least one friend** (a support) **in each neighboring** (link by a constraint) **building** (domain)

- AC1 = A person leaving sends a letter to **everybody**
  "I am leaving"
  - everyone check they still have a friend in each neighboring building.

- AC3 = A person leaving sends a letter to **the neighboring buildings**
  "I am leaving building n°i"
  - Only the neighbors of building i check if they still have a friend in building n°i

- AC4 = A person (let's say Alan) leaving sends a letter **to his friends**
  "I am leaving building n°i, take care, Alan".
  - Only the friends of Alan update their list of friends and check it is not empty for building n°i.

The friendship view of Arc-consistency

(borrowed and adapted from Romuald Debruyne)

- AC = **I** (a value) **stay** (in the domain) **if I have at least one friend** (a support) **in each neighboring** (link by a constraint) **building** (domain)

- AC1 = A person leaving sends a letter to **everybody**
  "I am leaving"
  - everyone check they still have a friend in each neighboring building.

- AC3 = A person leaving sends a letter to **the neighboring buildings**
  "I am leaving building n°i"
  - Only the neighbors of building i check if they still have a friend in building n°i

- AC4 = A person (let's say Alan) leaving sends a letter **to his friends**
  "I am leaving building n°i, take care, Alan".
  - Only the friends of Alan update their list of friends and check it is not empty for building n°i.

- AC6 = A person leaving sends a letter to **his friends who consider him as a "best" friend**
  "I am leaving building n°i, take care, Alan"
  - Only these people look for a new best-friend in building n°i.
  - A "best" friend is a friend leaving at the smallest possible number (so numbers i.e values are checked only once).

Event-based propagation mechanism

Propagation mechanism

Algorithm 6 propagate(Q)

```
1: while Q ≠ ∅ do
2:     evt(x) ← pop(Q) //an event on variable x
3:     for c ∈ C|x ∈ X(c) do
4:         filter(c, evt(x), Q) // for example filter(x ≤ y, x_i ≥ v_k, Q)
```

Typical events: instantiation $(x = a)$, value removal $(x \neq a)$, change of bounds $(x \geq a \text{ or } x \leq a)$

Event-based propagation mechanism

Propagation mechanism

Algorithm 6 propagate(Q)

```
1: while Q ≠ ∅ do
2:     evt(x) ← pop(Q) //an event on variable x
3:     for c ∈ C|x ∈ X(c) do
4:         filter(c, evt(x), Q) // for example filter(x ≤ y, x_i ≥ v_k, Q)
```

Typical events: instantiation $(x = a)$, value removal $(x \neq a)$, change of bounds $(x \geq a \text{ or } x \leq a)$

Filtering mechanisms

Algorithm 8 filter($x \leq y$, $x \geq v_k$, Q)

```
1: if y < v_k then
2:     y ← v_k
3:     Q ← Q ∪ {y ≥ v_k}
```

Note: this is the algorithm applied in reaction to an update of the lower bound of x

Event-based propagation mechanism

Propagation mechanism

Algorithm 6 propagate(Q)

```
1: while Q ≠ ∅ do
2:     evt(x) ← pop(Q) //an event on variable x
3:     for c ∈ C|x ∈ X(c) do
4:         filter(c, evt(x), Q) // for example filter(x ≤ y, x_i ≥ v_k, Q)
```

Typical events: instantiation $(x = a)$, value removal $(x \neq a)$, change of bounds $(x \geq a \text{ or } x \leq a)$

Filtering mechanisms

Algorithm 7 filterAC4(c_{ij}, $x_i \neq v_k$, Q)

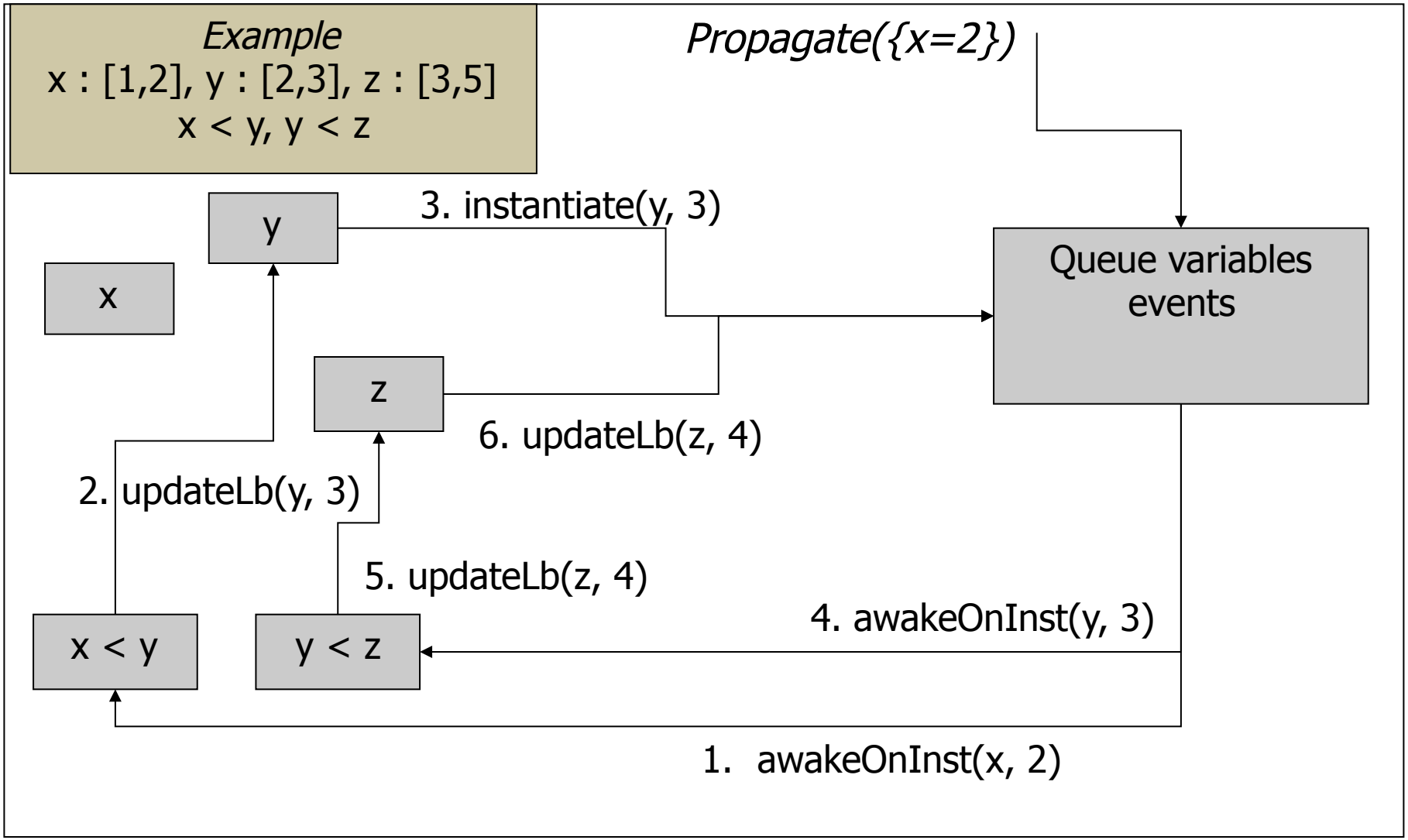
```
1: for v_l ∈ D(x_j) and such that (v_k, v_l) ∈ R(c_ij) do
2:     nbSupports[(x_j, v_l), x_i] ← nbSupports[(x_j, v_l), x_i] - 1
3:     if nbSupports[(x_j, v_l), x_i] = 0 then
4:         D(x_j) ← D(x_j) − {v_l};
5:         Q ← Q ∪ {x_j ≠ v_l};
```

Algorithm 8 filter($x \leq y$, $x \geq v_k$, Q)

```
1: if y < v_k then
2:     y ← v_k
3:     Q ← Q ∪ {y ≥ v_k}
```

Note: this is the algorithm applied in reaction to an update of the lower bound of x

Event-based propagation mechanism



Intentional constraints

- Usual properties expected of a filtering algorithm:
  - **Correction** :

    ”no solution is lost”
  - **Assignment completeness** :

    ”detect failure when all variables are instantiated”
  - **Contractance** :

    ”domains are smaller after applying the propagator”
  - **Monotonic** :

    ”applying the propagator on smaller domains give a smaller result”
  - **Idempotence (often)** :

    ”is able to reach its own fixpoint”

MAC algorithm

- Tree search:
  - Given a decision (a constraint), for example $(x = a)$, either:
    - There is a solution with $(x = a)$,
    - There is a solution with $(x \neq a)$ (the negation of the decision)
    - There is no solution
  - Typical decision (fix a value, split a domain, enforce a precedence, …)
  - Exploring both branches gives a complete algorithm
  - Local consistencies are enforced before each decision

- MAC : a tree search using AC as a local consistency

MAC: Maintaining Arc-consistency

(borrowed from Romuald Debruyne)