Constraint Programming

Marco Kuhlmann & Guido Tack

Lecture 5

Today:
Propagation algorithms

Brief recap: propagation in a loop

Propagators

- A **propagator** is a contracting, monotonic function
  p ∈ Store → Store
- A propagator *p* **implements** a constraint *C* if and only if

$$p(store(\alpha)) = store(\alpha) \cap C$$

(using slightly sloppy notation)

Naive propagation function

```
propagate ( (V,D,P,b), s)
  while p∈P and p(s)≠s do
    s := p(s);
   return s;
```

- **Termination:** (S, <) is well-founded
- **Result:** largest simultaneous fixpoint of all P

Improved propagation

```
propagate ( (V,D,P,b), s₀)
  s := s₀; DP = P;
  while DP ≠ ∅ do
    choose p∈DP;
    s' := p(s); DP = DP−{p};
    MV := { x∈V | s(x)≠s'(x) };
    N  := { q∈P | ∃x∈var(q): x∈MV };
    DP := DP ∪ N;
    s := s';
  return s;
```

Improved propagation

```
propagate ( (V,D,P,b), s0)
  s := s0; DP = P;
  while DP ≠ ∅ do
    choose p∈DP;
    s' := p(s); DP = DP−{p};
    MV := { x∈V | s(x)≠s'(x) };
    N  := { q∈P | ∃x∈var(q): x∈MV };
    DP := DP ∪ N;
    s := s';
  return s;
```

dirty propagators

Improved propagation

```
propagate ( (V,D,P,b), s₀)
  s := s₀; DP = P;
  while DP ≠ ∅ do
    choose p∈DP;
    s' := p(s); DP = DP−{p};
    MV := { x∈V | s(x)≠s'(x) };
    N  := { q∈P | ∃x∈var(q): x∈MV };
    DP := DP ∪ N;
    s := s';
  return s;
```

dirty propagators

dependency-directed

Propagation with fixpoint reasoning and events

```
propagate ( (V,D,P,b), s₀)
   s := s₀; DP = P;
   while DP ≠ ∅ do
      choose p∈DP;
      (stat,s') := p(s); DP = DP−{p};
      if stat=subsumed then P:=P−{p};

      N  := { q∈P | events(s,s') ∩ es(q) ≠ ∅ };
      if stat=fix then N:=N−{p};
      DP := DP ∪ N;
      s := s';
   return (P,s);
```

Propagation with fixpoint reasoning and events

```
propagate ( (V,D,P,b), s₀)
   s := s₀; DP = P;
   while DP ≠ ∅ do
      choose p∈DP;
      (stat,s') := p(s); DP = DP−{p};
      if stat=subsumed then P:=P−{p};

      N  := { q∈P | events(s,s') ∩ es(q) ≠ ∅ };
      if stat=fix then N:=N−{p};
      DP := DP ∪ N;
      s := s';
   return (P,s);
```

status messages

Propagation with fixpoint reasoning and events

```
propagate ( (V,D,P,b), s₀)
   s := s₀; DP = P;
   while DP ≠ ∅ do
      choose p∈DP;
      (stat,s') := p(s); DP = DP−{p};
      if stat=subsumed then P:=P−{p};

      N  := { q∈P | events(s,s') ∩ es(q) ≠ ∅ };
      if stat=fix then N:=N−{p};
      DP := DP ∪ N;
      s := s';
   return (P,s);
```

status messages

event sets

Propagation with fixpoint reasoning and events

```
propagate ( (V,D,P,b), s₀)
   s := s₀; DP = P;
   while DP ≠ ∅ do
      choose p∈DP;
      (stat,s') := p(s); DP = DP−{p};
      if stat=subsumed then P:=P−{p};

      N  := { q∈P | events(s,s') ∩ es(q) ≠ ∅ };
      if stat=fix then N:=N−{p};
      DP := DP ∪ N;
      s := s';
   return (P,s);
```

status messages

event sets

fixpoint knowledge

What is *p*?

- Remaining problem:

  How to compute *p* efficiently for different constraints

- We will look at
  - linear equations
  - all-distinct
  - element

Propagation strength

Addition

- Exercise: Define propagators p_x, p_y, p_z that implement x+y=z such that
  - they only propagate if input variables are assigned
  - they only use bounds of input variables
  - they use the full domains of input variables
- How much can they possibly propagate?

Strongest: domain consistency

- Using full domain information:

$$p_x = \{x \mapsto \{n \in s(x) \mid \exists m_y \in s(y), m_z \in s(z).n = m_y + m_z\} \ldots\}$$

- For fixpoint, we can show:

$$\forall n_x \in s(x).\ \exists n_y \in s(y), n_z \in s(z) : n_x = n_y + n_z$$

$$\forall n_y \in s(y).\ \exists n_x \in s(x), n_z \in s(z) : n_x = n_y + n_z$$

$$\forall n_z \in s(z).\ \exists n_x \in s(x), n_y \in s(y) : n_x = n_y + n_z$$

- For each value in the domain of some *x*, we find *support* in the other variables' domains

Alternative definition

- Propagator p **achieves domain consistency** for constraint C:

$$p(s) = store(s \cap C)$$

where

$$store(s \cap C) = \min\{s' \mid s \cap C \subseteq s'\}$$

(strongest store containing $s \cap C$)

- We also say p is **complete** for C
- Very similar: $p(store(\alpha)) = store(\alpha) \cap C$

Weaker: bounds consistency

- Using only bounds information:

$$p_x = \{x \mapsto \{n \in s(x) \quad | \quad n \leq \max(s(y)) + \max(s(z)) \wedge$$
$$n \geq \min(s(y)) + \min(s(z))\} \ldots \}$$

- For fixpoint, we can show:

$$\text{for } n_x \in \{\min(s(x), \max(s(x))\}.$$
$$\exists n_y \in [\min(s(y)) \ldots \max(s(y))],$$
$$n_z \in [\min(s(z)) \ldots \max(s(z))] : n_x = n_y + n_z$$

Example: consistency notions

- consider $x=y+z$ and domains

  $x \in \{1,2,4,8\}$, $y \in \{1,3,4\}$, $z \in \{1,2,3,4,6\}$

- a dom-propagator would produce

  $x \in \{2,4,8\}$, $y \in \{1,3,4\}$, $z \in \{1,3,4\}$

- a bnd-propagator only

  $x \in \{2,4,8\}$, $y \in \{1,3,4\}$, $z \in \{1,2,3,4\}$

Example: consistency notions

- consider distinct(w,x,y,z) and domains

  $w \in \{3,4\}$, $x \in \{1,3,5\}$, $y \in \{1,2,4\}$, $z \in \{3,4\}$

- a dom-propagator would produce

  $w \in \{3,4\}$, $x \in \{1,5\}$, $y \in \{1,2\}$, $z \in \{3,4\}$

- a bnd-propagator only

  $w \in \{3,4\}$, $x \in \{1,3,5\}$, $y \in \{1,2\}$, $z \in \{3,4\}$

Propagator Properties

- A domain consistent propagator is idempotent
- Is a bounds consistent propagator idempotent?
- A contracting function that always achieves domain consistency is a propagator
- Proof: Exercise

Propagation algorithms

Linear equations

Linear equations

- Propagator for

$$\sum_i a_i x_i = c$$

- How can bounds information be propagated efficiently?
- Example:

$$ax + by = c$$

Propagating bounds

- Rewrite:

ax + by = c   ax = c - by

x = (c-by)/a

- Propagate

$$x \leq \lfloor \max\{(c - bn)/a \mid n \in s(y)\} \rfloor$$

$$x \geq \lceil \min\{(c - bn)/a \mid n \in s(y)\} \rceil$$

Propagating bounds

- $m = \max\{(c - bn)/a) \mid n \in s(y)\}$
- a > 0:

  $$m = \max\{c - bn \mid n \in s(y)\}/a$$

- a < 0:

  $$m = \min\{c - bn \mid n \in s(y)\}/a$$

Propagating bounds

- For a>0:

$$\begin{aligned} m &= \max\{c - bn \mid n \in s(y)\}/a \\ &= (c - \min\{bn \mid n \in s(y)\})/a \end{aligned}$$

  - For b>0:

$$m = (c - b \cdot \min(s(y)))/a$$

  - For b<0:

$$m = (c - b \cdot \max(s(y)))/a$$

General Case

- Repeat until fixpoint, for each variable x_i
- Improvement: Compute

$$u = \max\left\{ d - \sum_{i=1}^{n} a_i n_i \mid n_i \in s(x_i) \right\}$$

$$l = \min\left\{ d - \sum_{i=1}^{n} a_i n_i \mid n_i \in s(x_i) \right\}$$

- Reuse by subtracting term for x_i in each iteration

Questions

- Is it necessary to iterate?

  Yes, otherwise not idempotent

- What level of consistency does the propagator achieve?

Consistency

- This propagator is not bounds consistent:

  $x = 3y + 5z$ with

  $x \in \{2,\ldots,7\}$, $y \in \{0,1,2\}$, $z \in \{-1,0,1,2\}$

- Propagator will compute

  $x \in \{2,\ldots,7\}$, $y \in \{0,1,2\}$, $z \in \{0,1\}$

  should be 6

Consistency

- Algorithm considers real-valued solutions:

  $x=7, y=2/3, z=1 \quad \Rightarrow \quad 7=3\cdot 2/3 + 5\cdot 1$

- New notion: bounds-R consistency

  (allow solutions over the reals)

- Even bounds consistency cannot be achieved efficiently for some propagators!

All-distinct

All-distinct

- Naive:
  - check that no two determined variables have the same value
  - remove values of determined variables from domains of undetermined variables
- Advantage: simple implementation, avoid $O(n^2)$ propagators
- Disadvantage: not very strong

All-distinct

- Is there an efficient bounds or domain consistent propagator?
- Puget: bounds consistent, $O(n \log n)$

  Régin: domain consistent, $O(n^{2.5})$

All-distinct

- Is there an efficient bounds or domain consistent propagator?
- Puget: bounds consistent, $O(n \log n)$

  Régin: domain consistent, $O(n^{2.5})$

Régin's algorithm

- **Construct a variable-value graph**

  bipartite, variable node → value node

- **Characterize solutions in the graph**

  maximal matchings

- **Use matching theory**

  one matching describes all matchings

- **Remove edges not taking part in any solution**

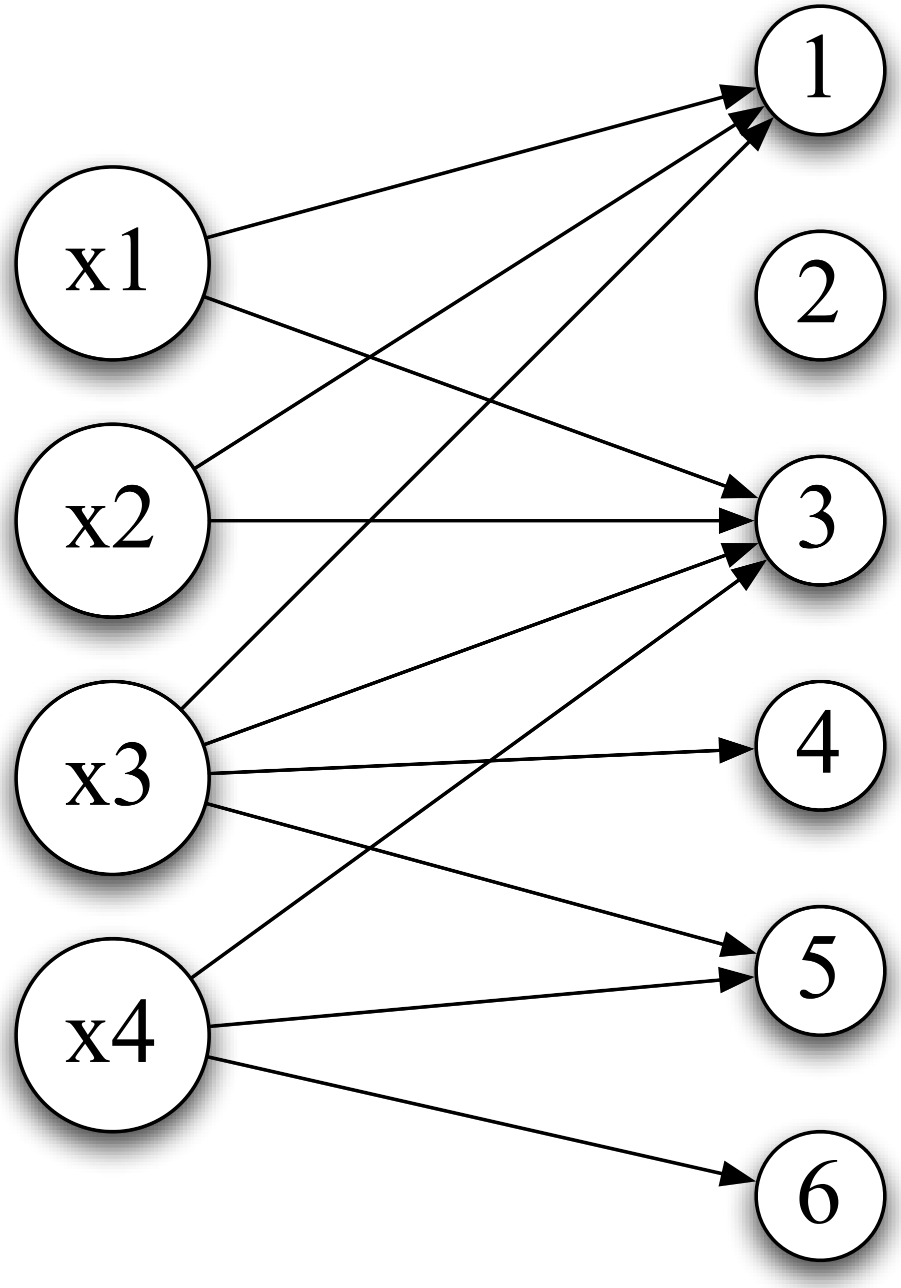
Variable-value Graph

$x1 \in \{1,3\}$

$x2 \in \{1,3\}$

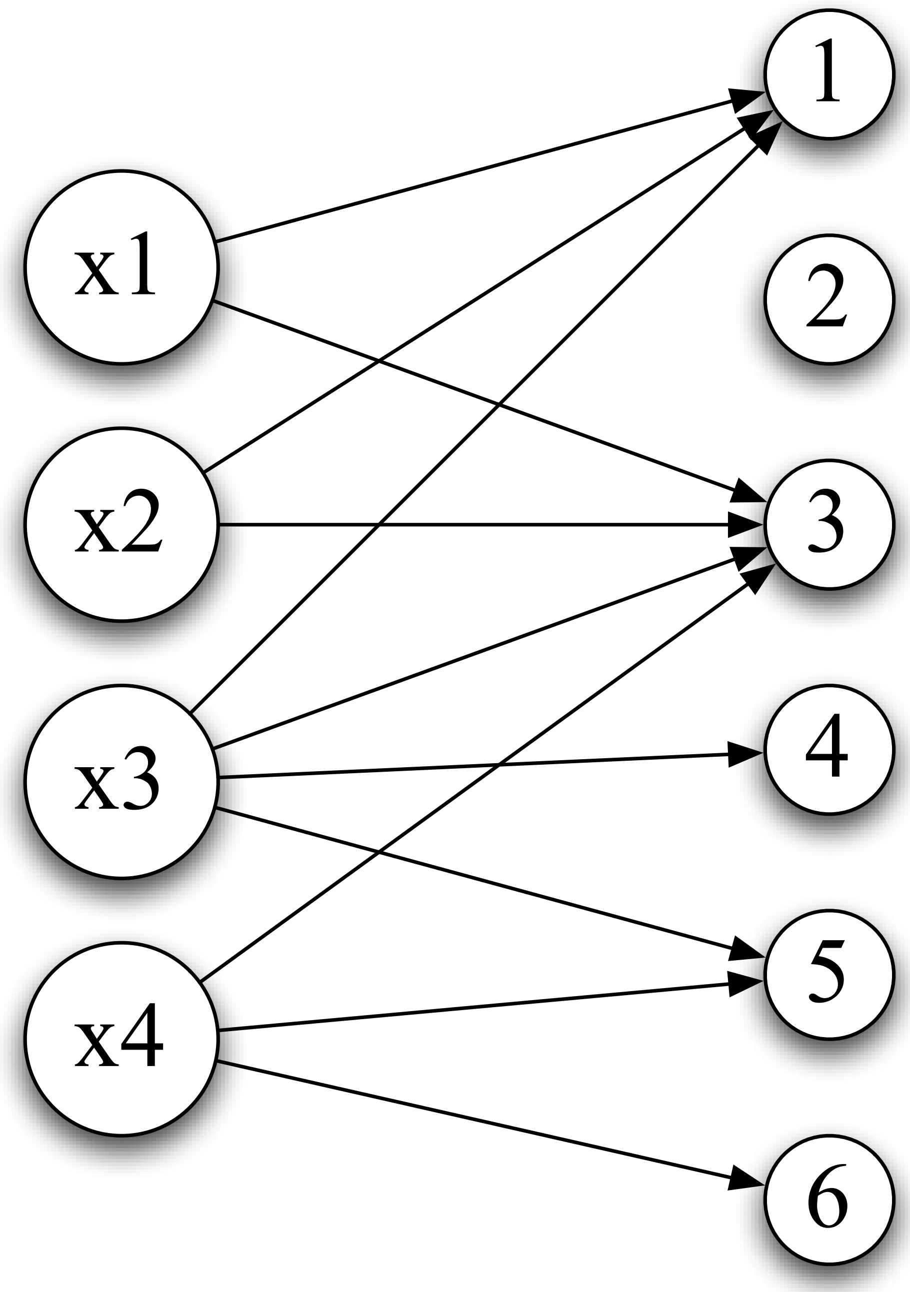
$x3 \in \{1,3,4,5\}$

$x4 \in \{3,5,6\}$



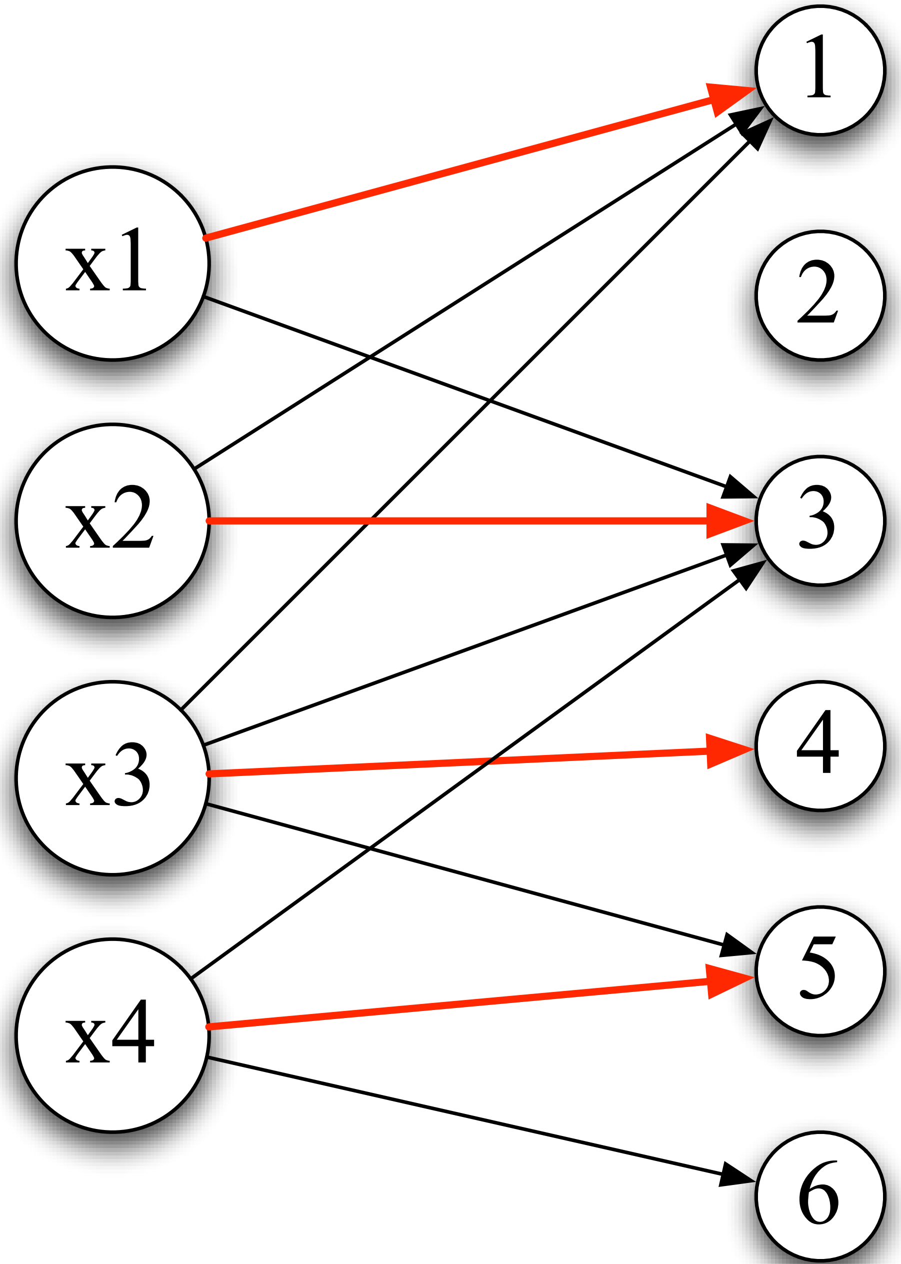
Matching

- subset of edges s.th. no two edges share a vertex
- maximal: maximum cardinality



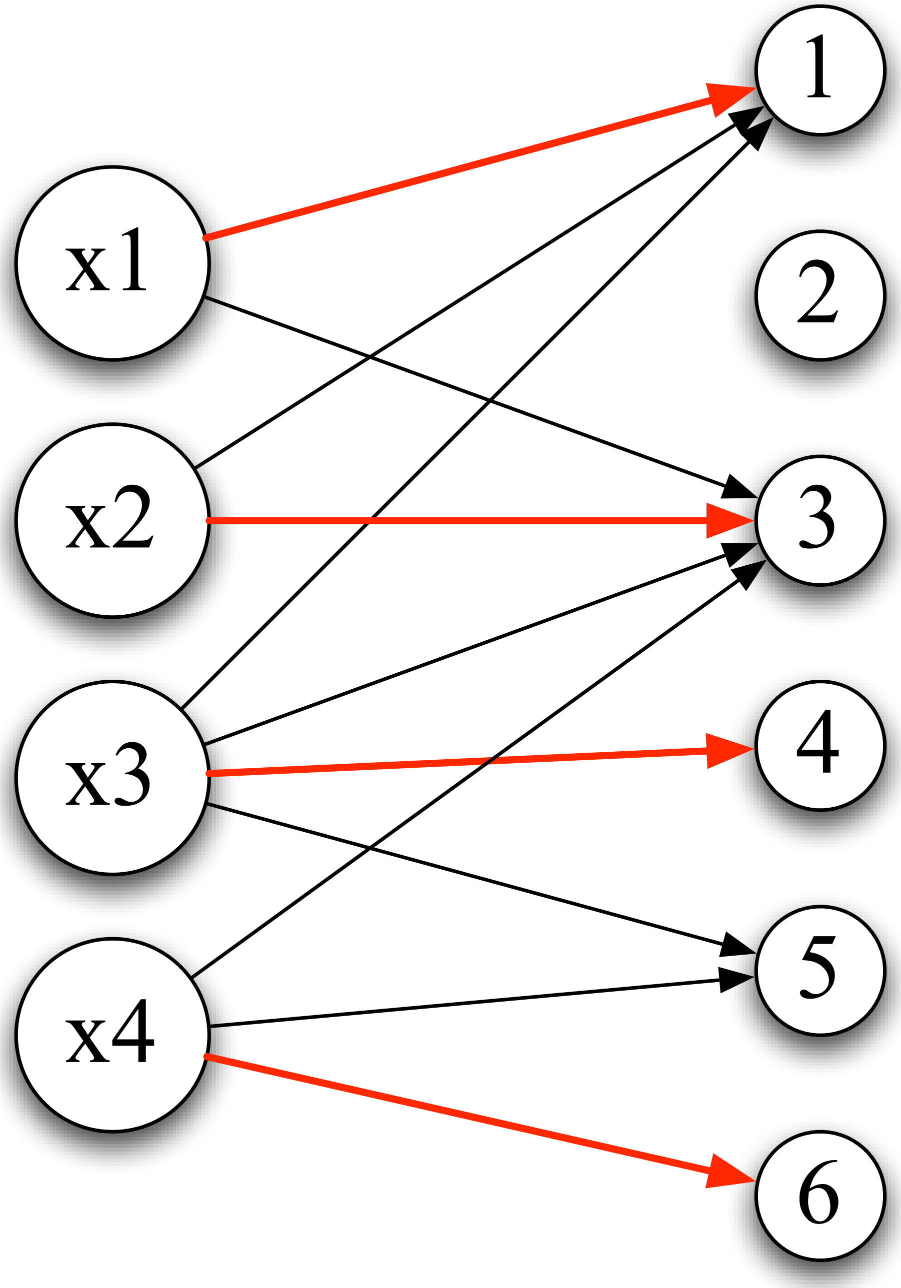
Matching

- subset of edges s.th. no two edges share a vertex
- maximal: maximum cardinality



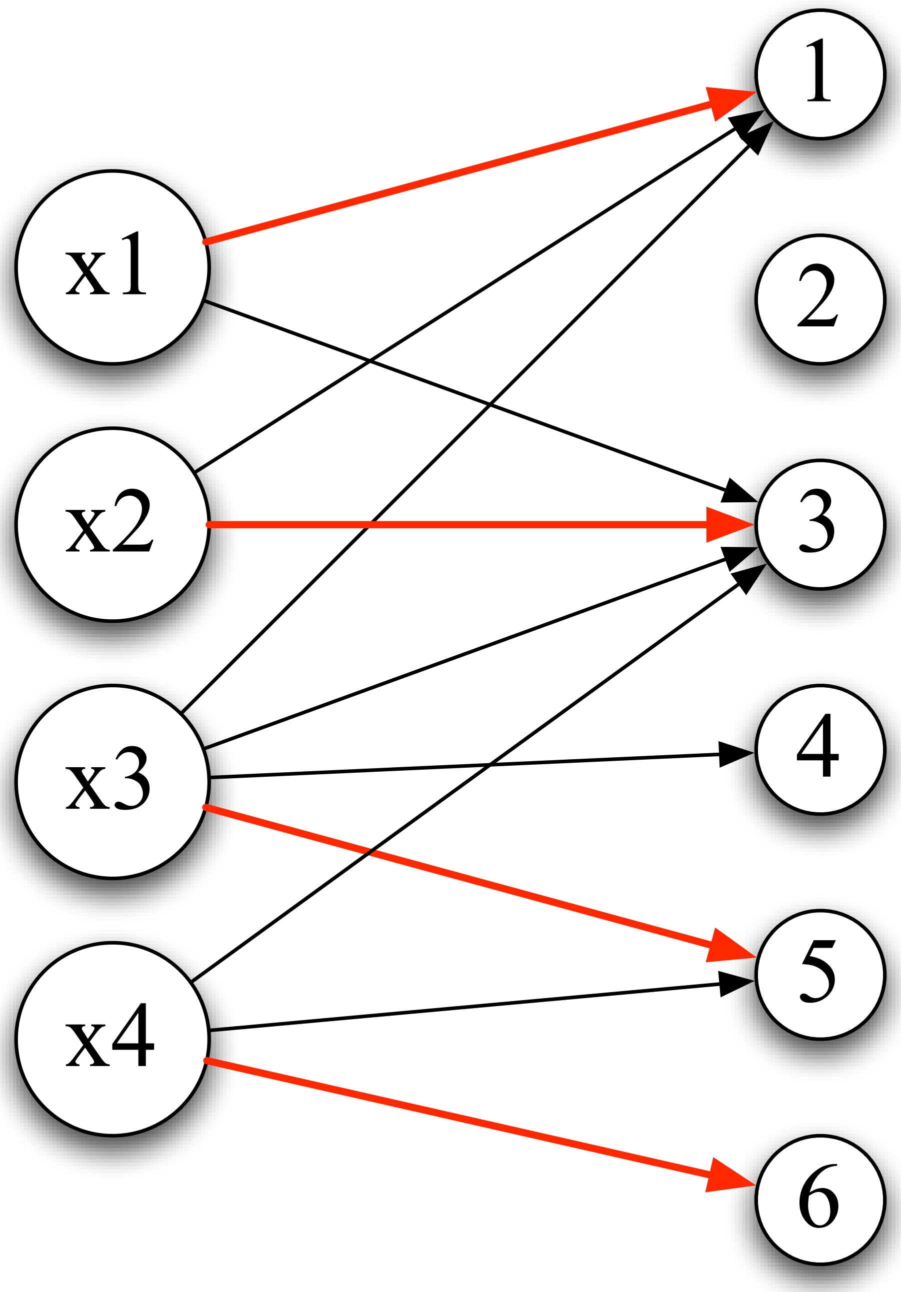
Matching

- subset of edges s.th. no two edges share a vertex
- maximal: maximum cardinality



Matching

- subset of edges s.th. no two edges share a vertex
- maximal: maximum cardinality



Notions

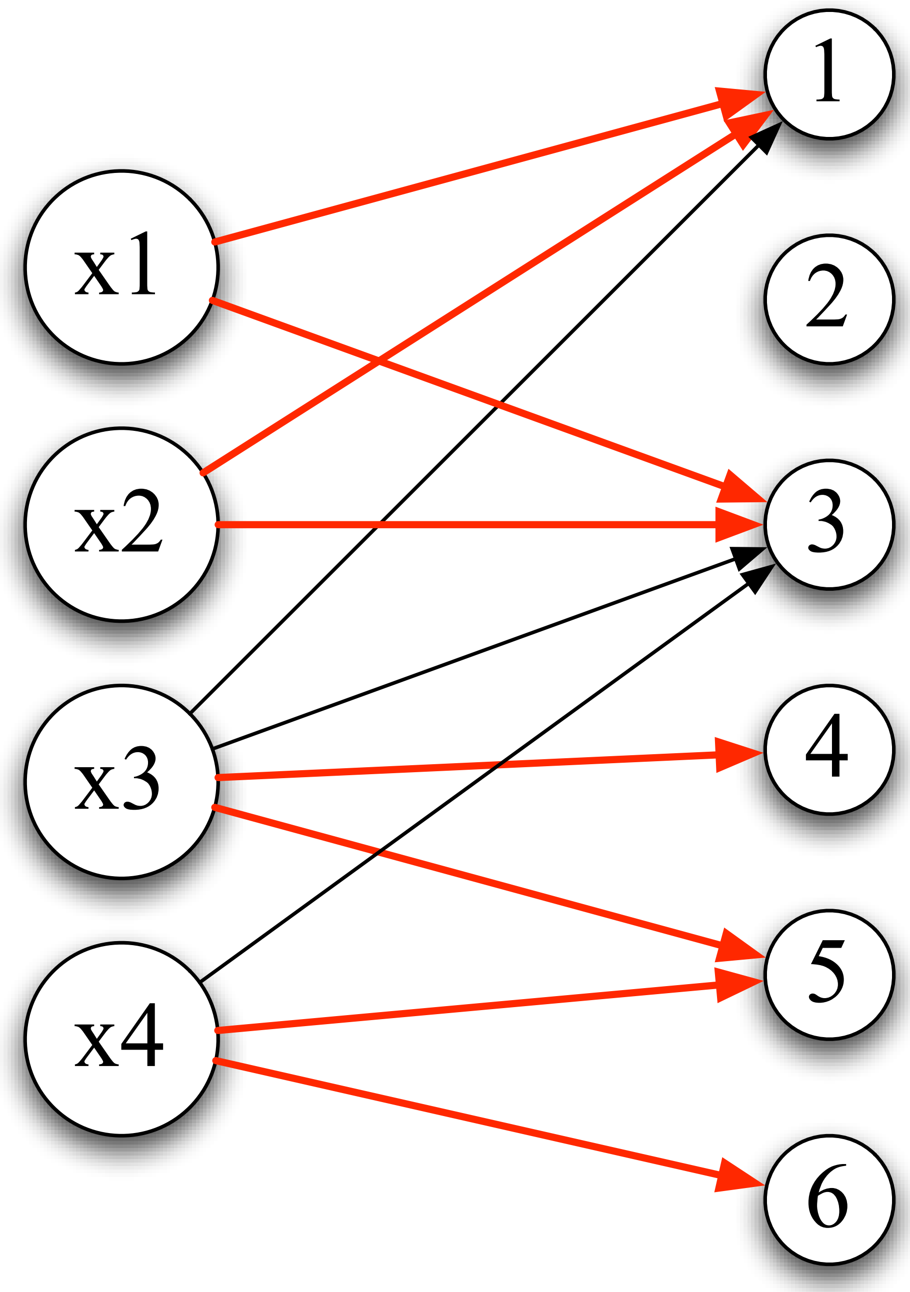
- For a given matching, we say that
  - an edge is matching if it belongs to the matching, otherwise it is free
  - a node is matched if incident to a matching edge, otherwise free

Maximal Matchings are solutions

- No value has two incoming edges
- All variable nodes are matched
- No matching: failure
- Edge **free** in all matchings: delete edge, remove value
- Edge **vital:** assign variable to single value

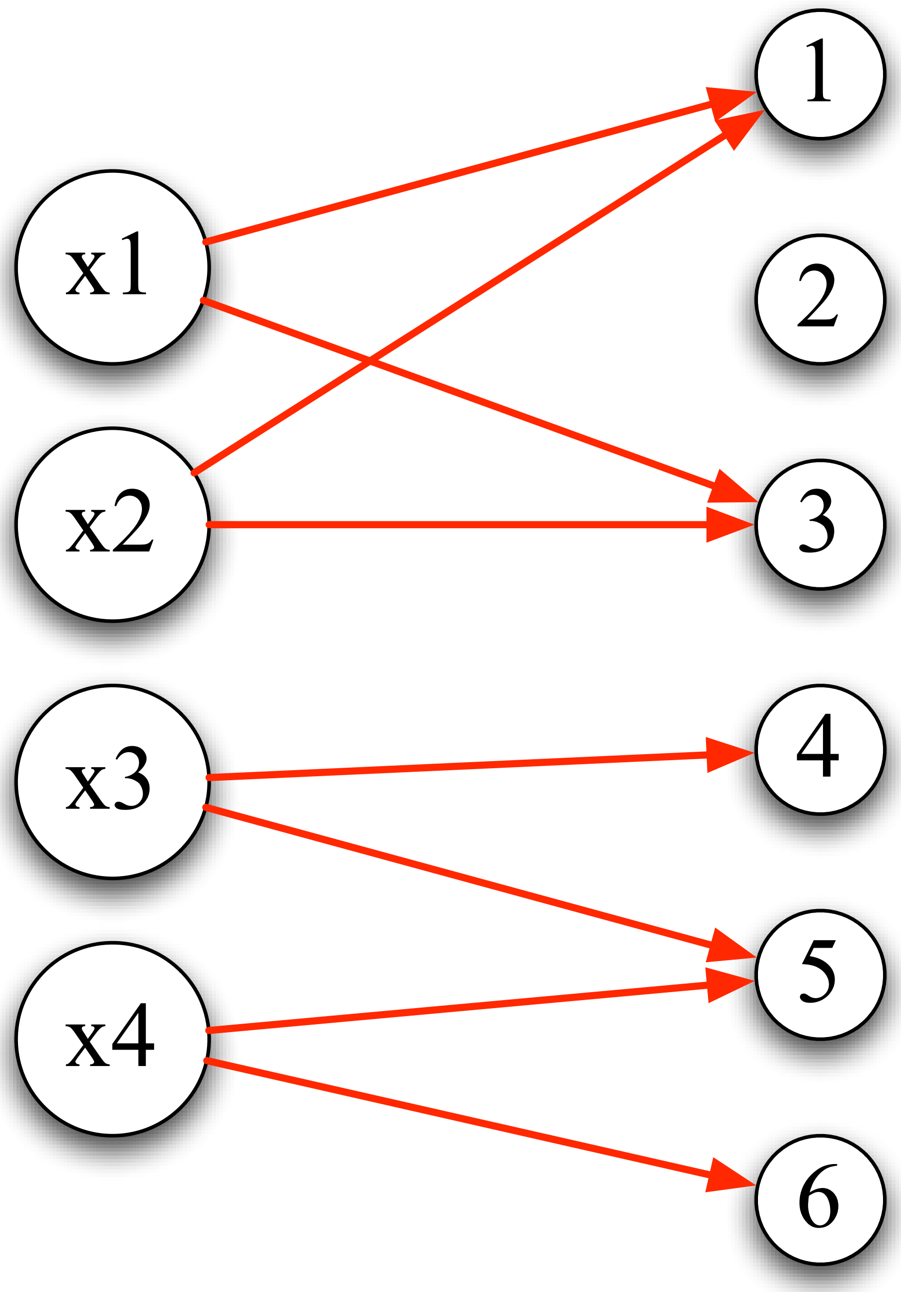
Matching

- Compute union of all maximal matchings



Matching

- Compute union of all maximal matchings
- Delete unmatched edges



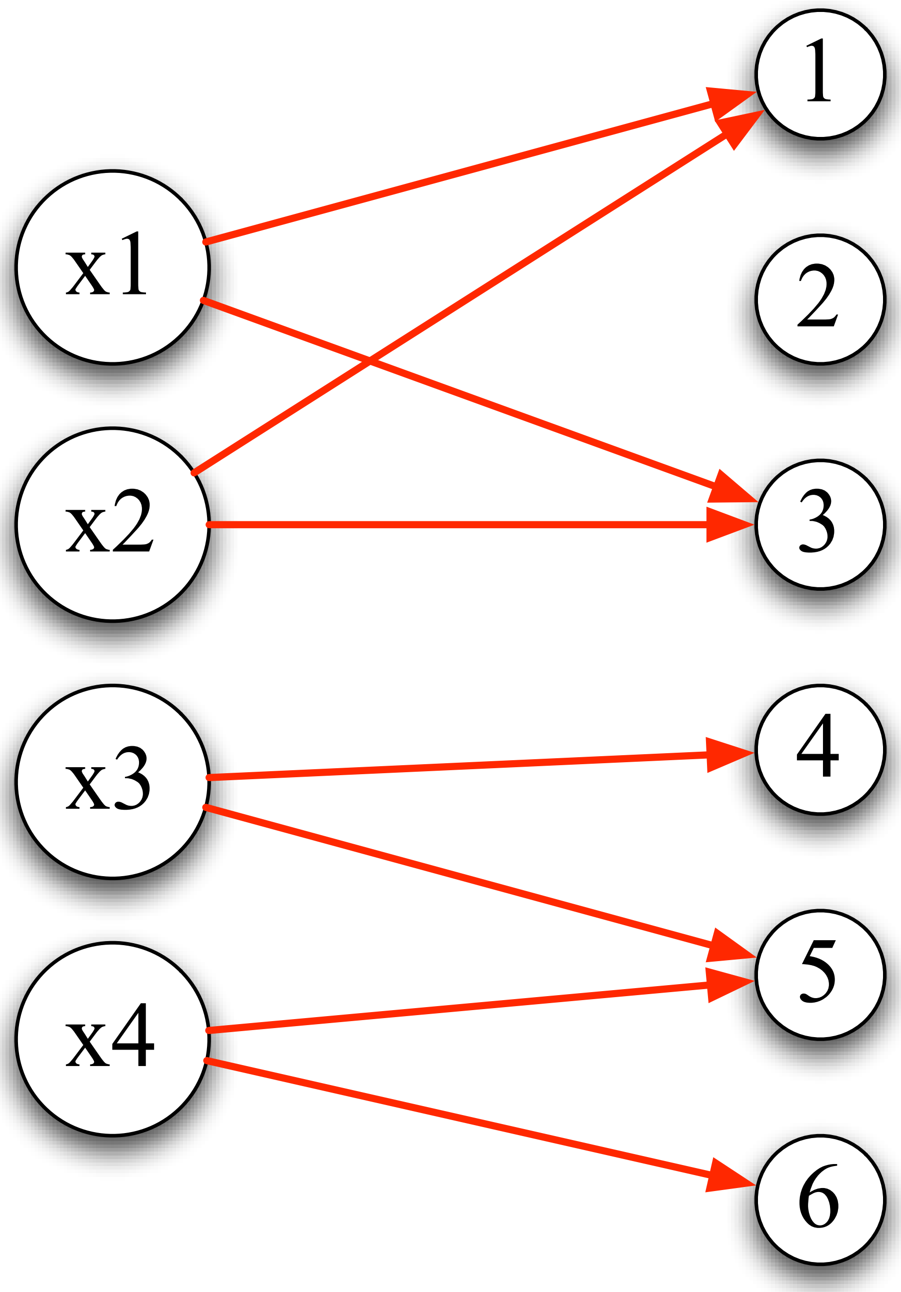
Compute new domains

$x1 \in \{1,3\}$

$x2 \in \{1,3\}$

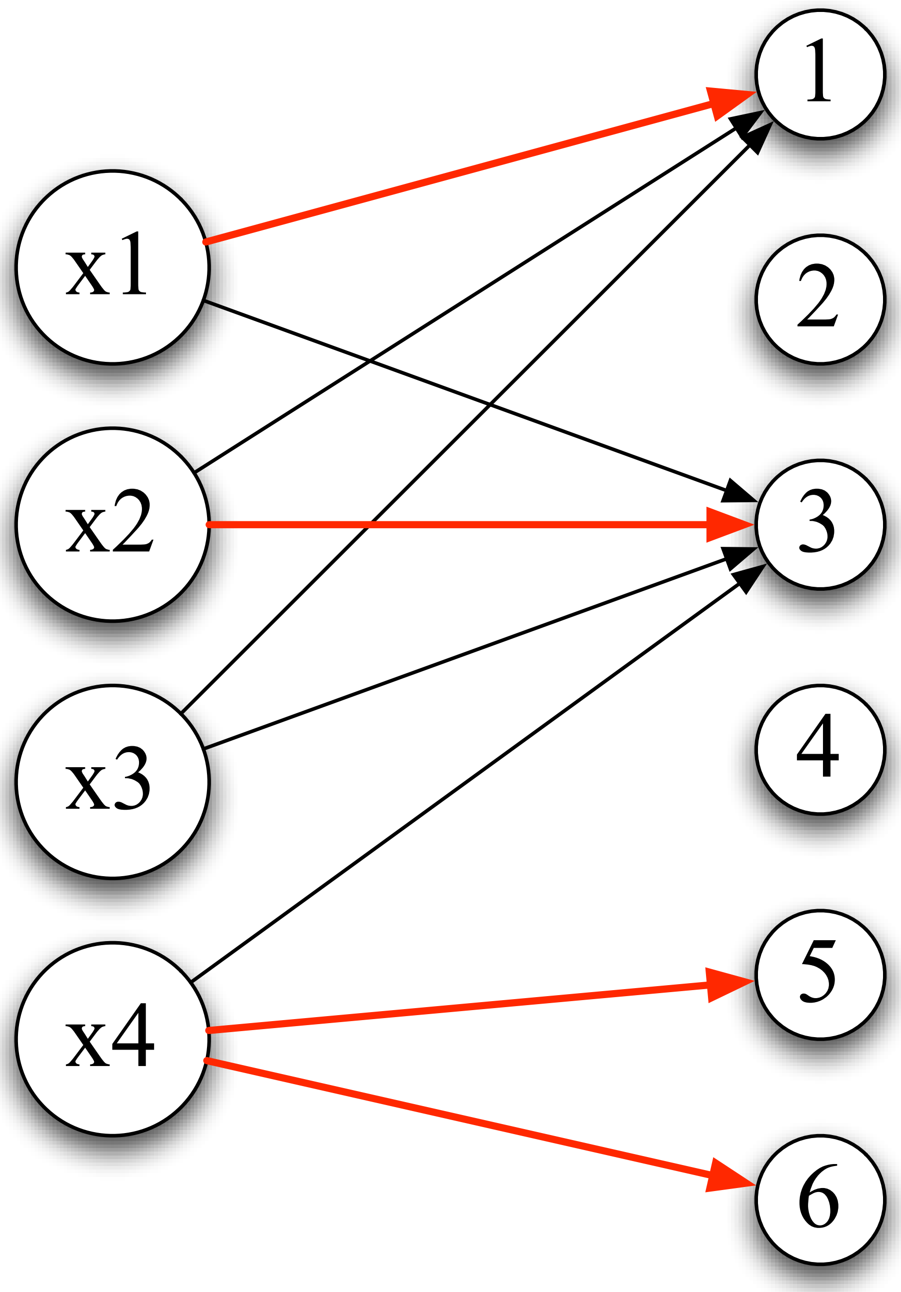
$x3 \in \{4,5\}$

$x4 \in \{5,6\}$



Failure

- If no maximal matching covering all variable nodes exists, we have detected failure



Maximal matching

- Can be computed in time $O(mn^{0.5})$, where m is the size of the union of the domains

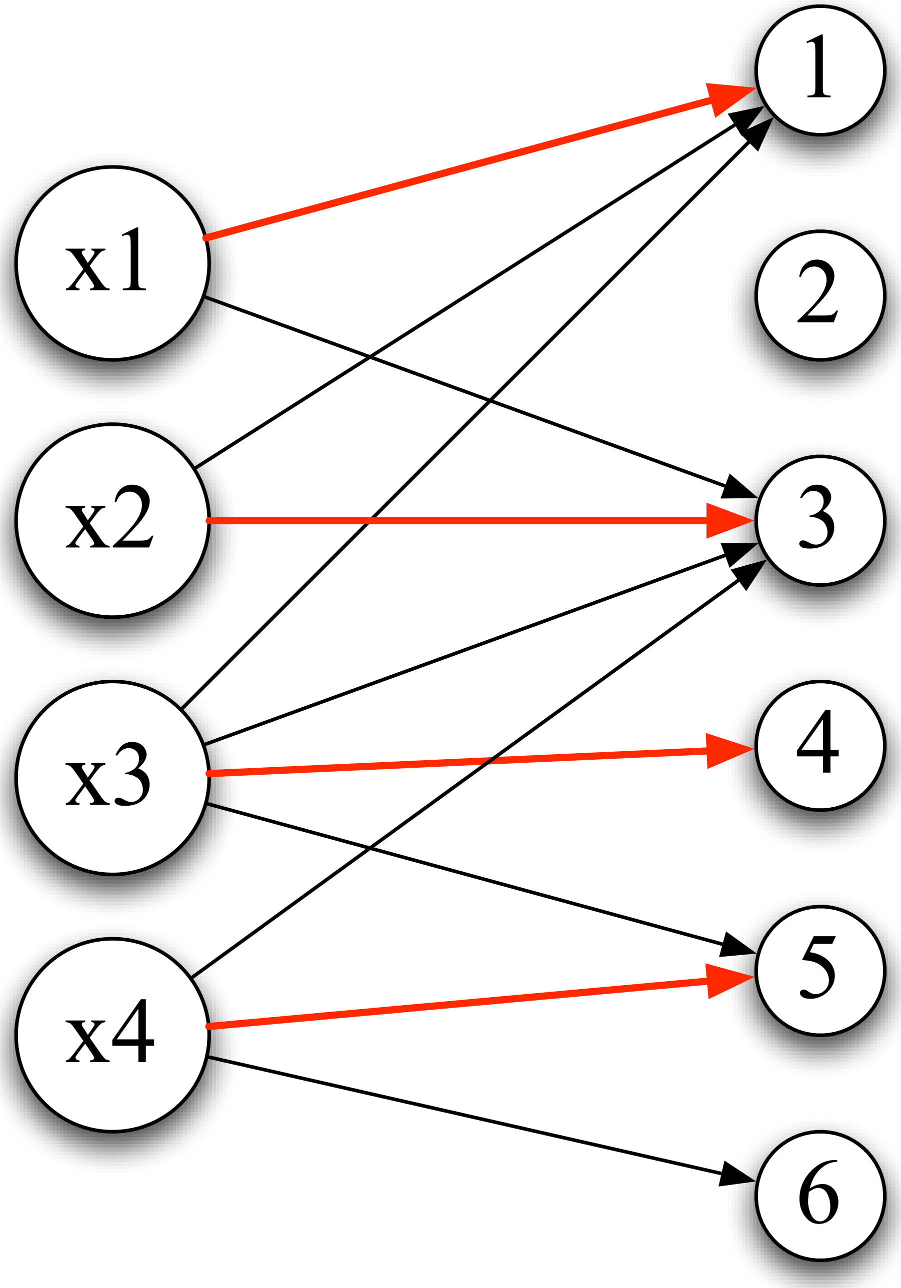
  (Hopcroft & Karp, 1973)

- Theorem:

  If M is some maximal matching in G, an edge belongs to any maximal matching in G iff it belongs to M, or to an M-alternating cycle, or to an even M-alternating path starting at an M-free node.

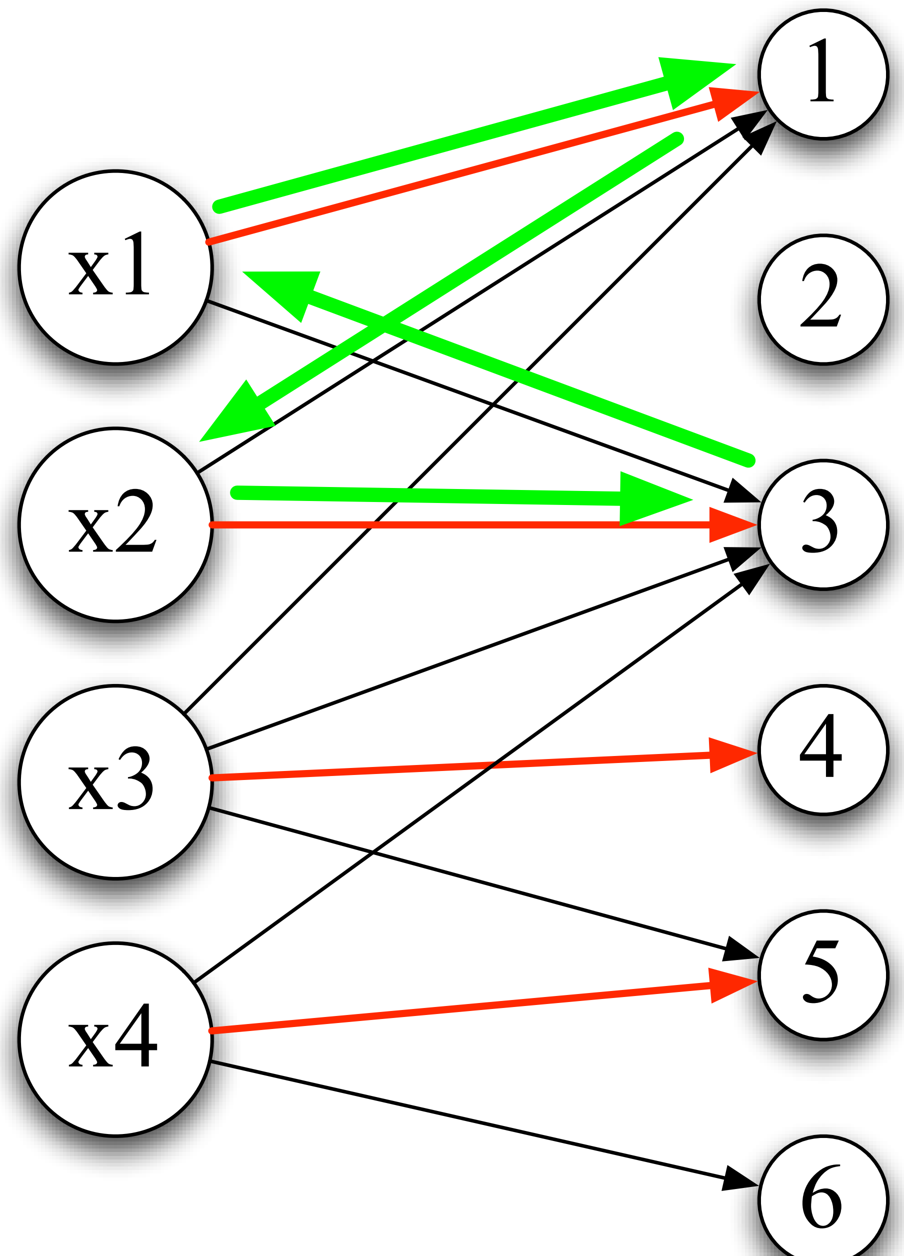
Maximal matching

- An M-alternating cycle



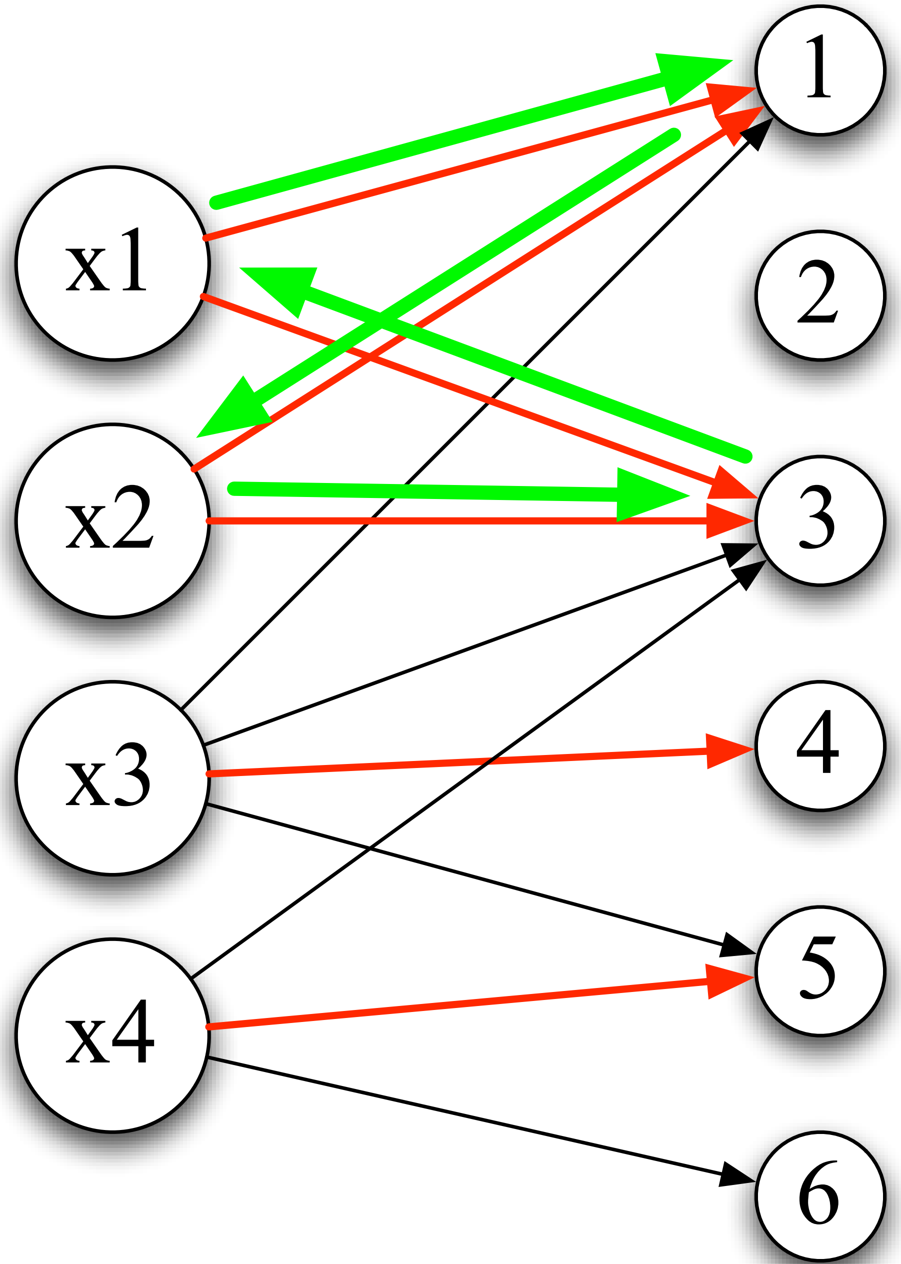
Maximal matching

- An M-alternating cycle



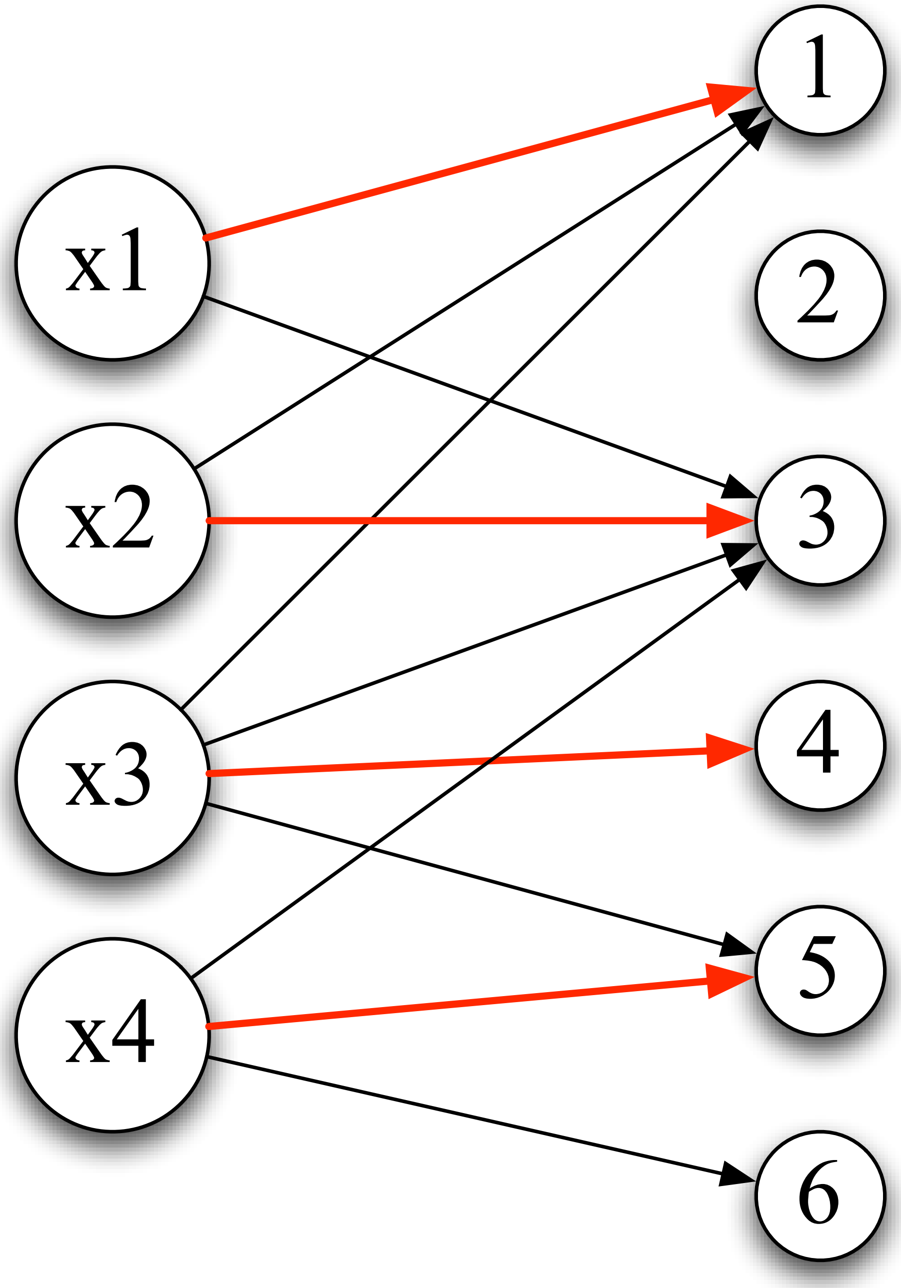
Maximal matching

- An M-alternating cycle



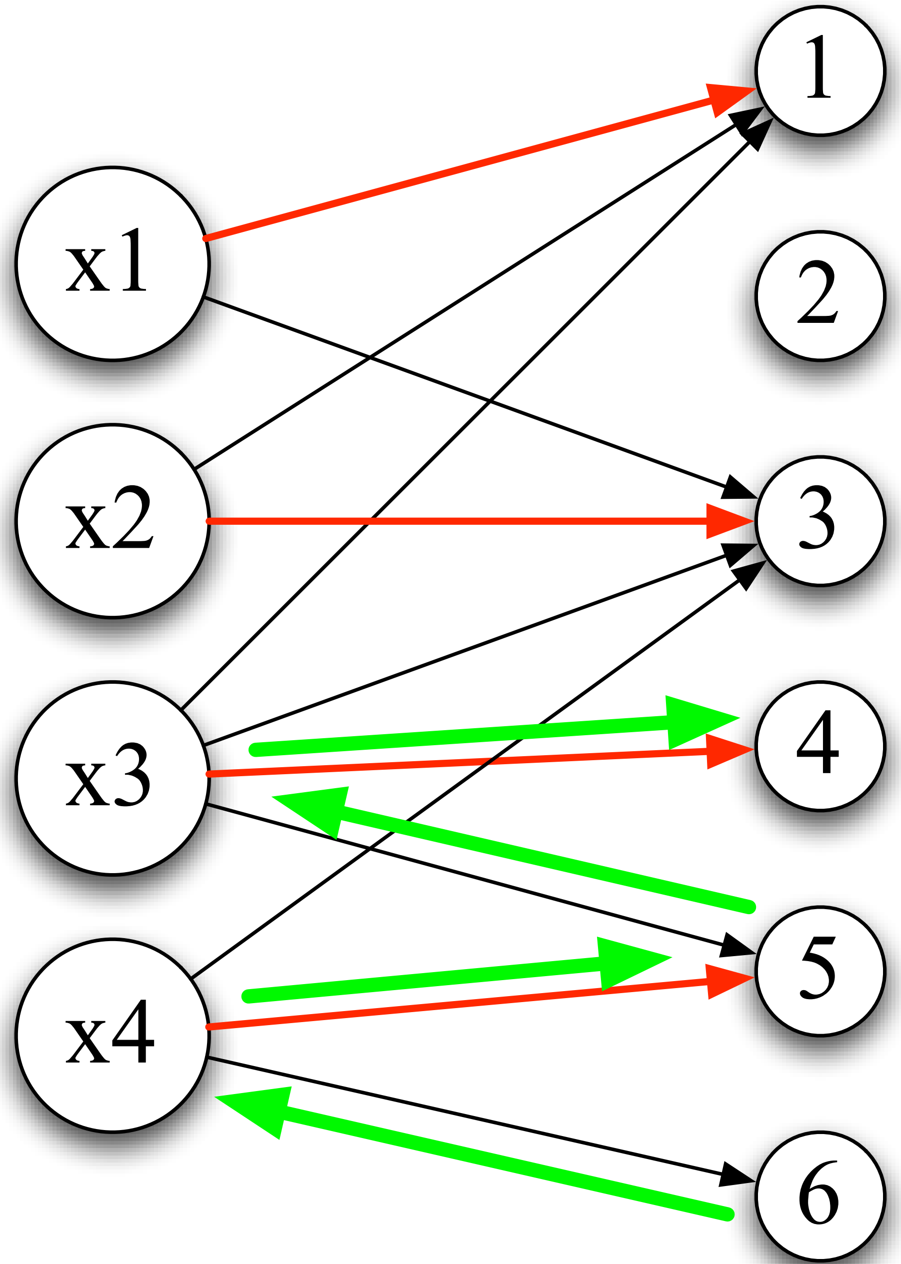
Maximal matching

- An even M-alternating path



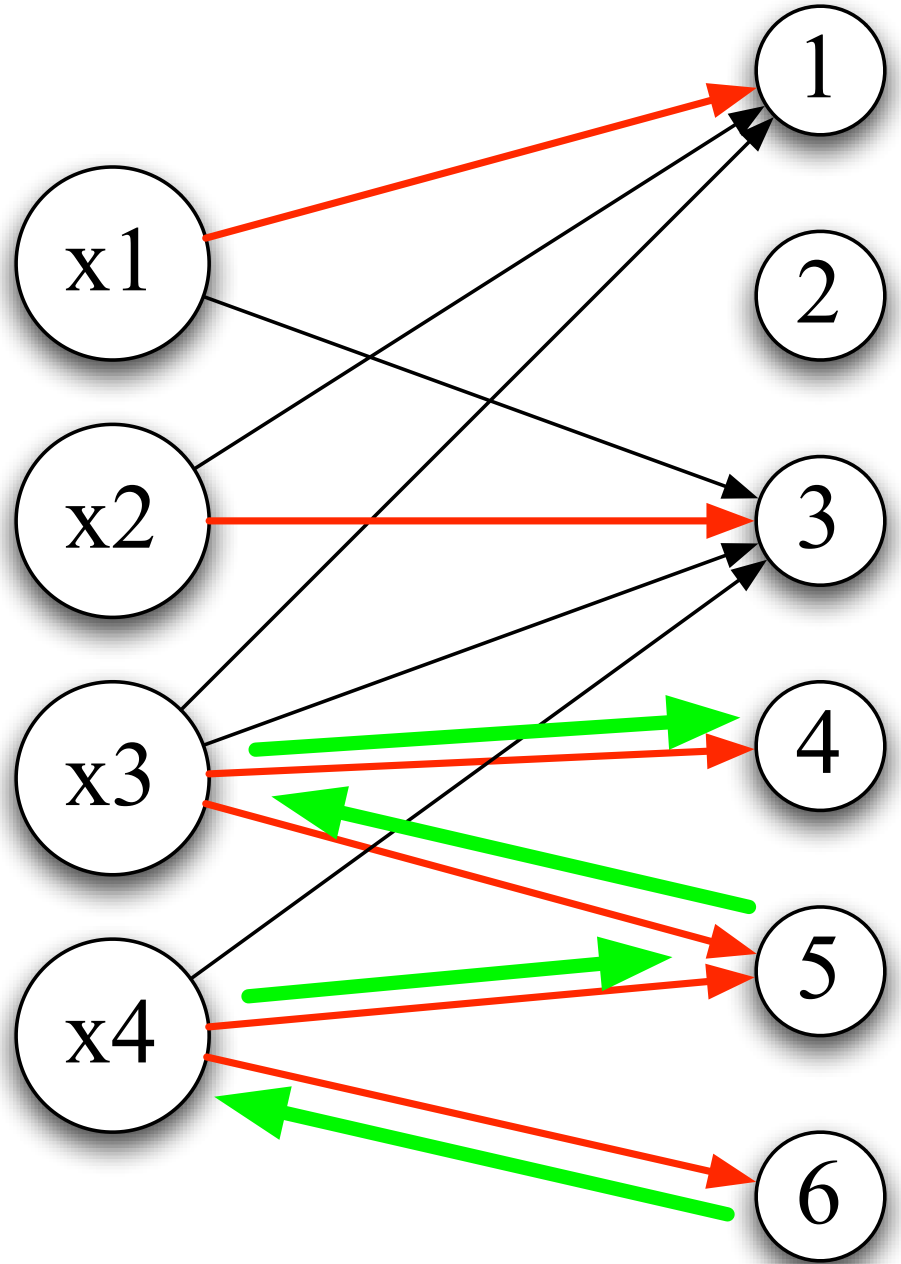
Maximal matching

- An even M-alternating path



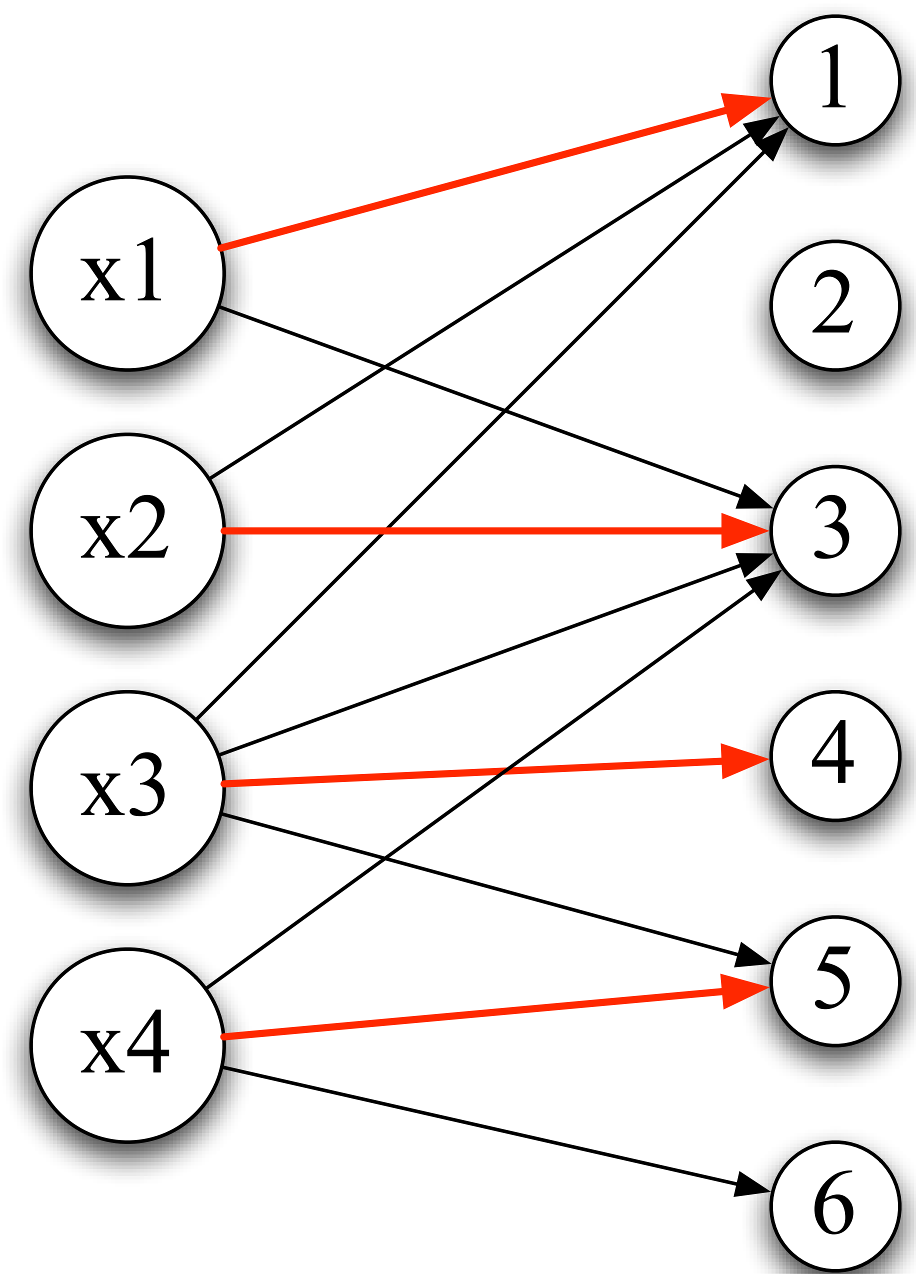
Maximal matching

- An even M-alternating path



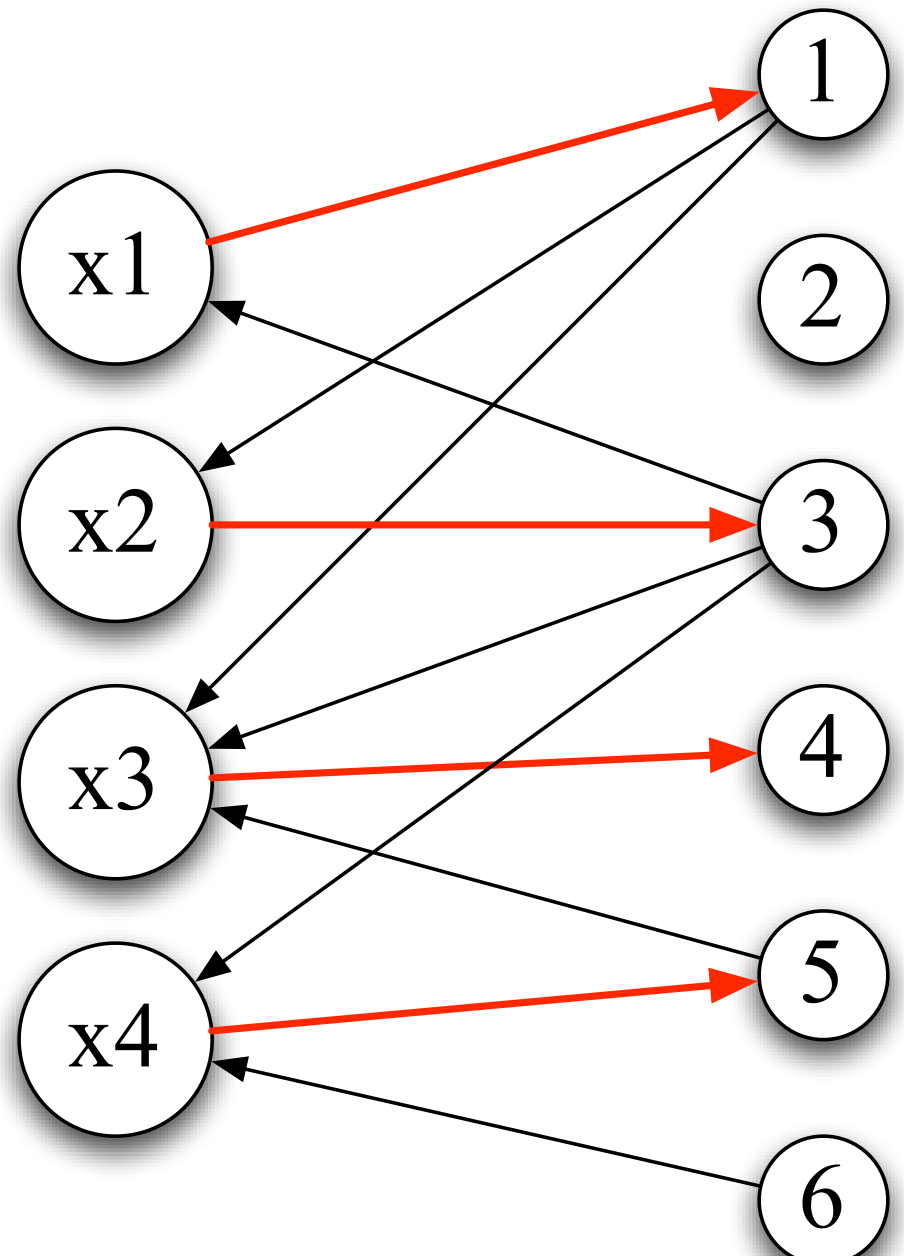
Maximal matching

- Reverse unmatched edges



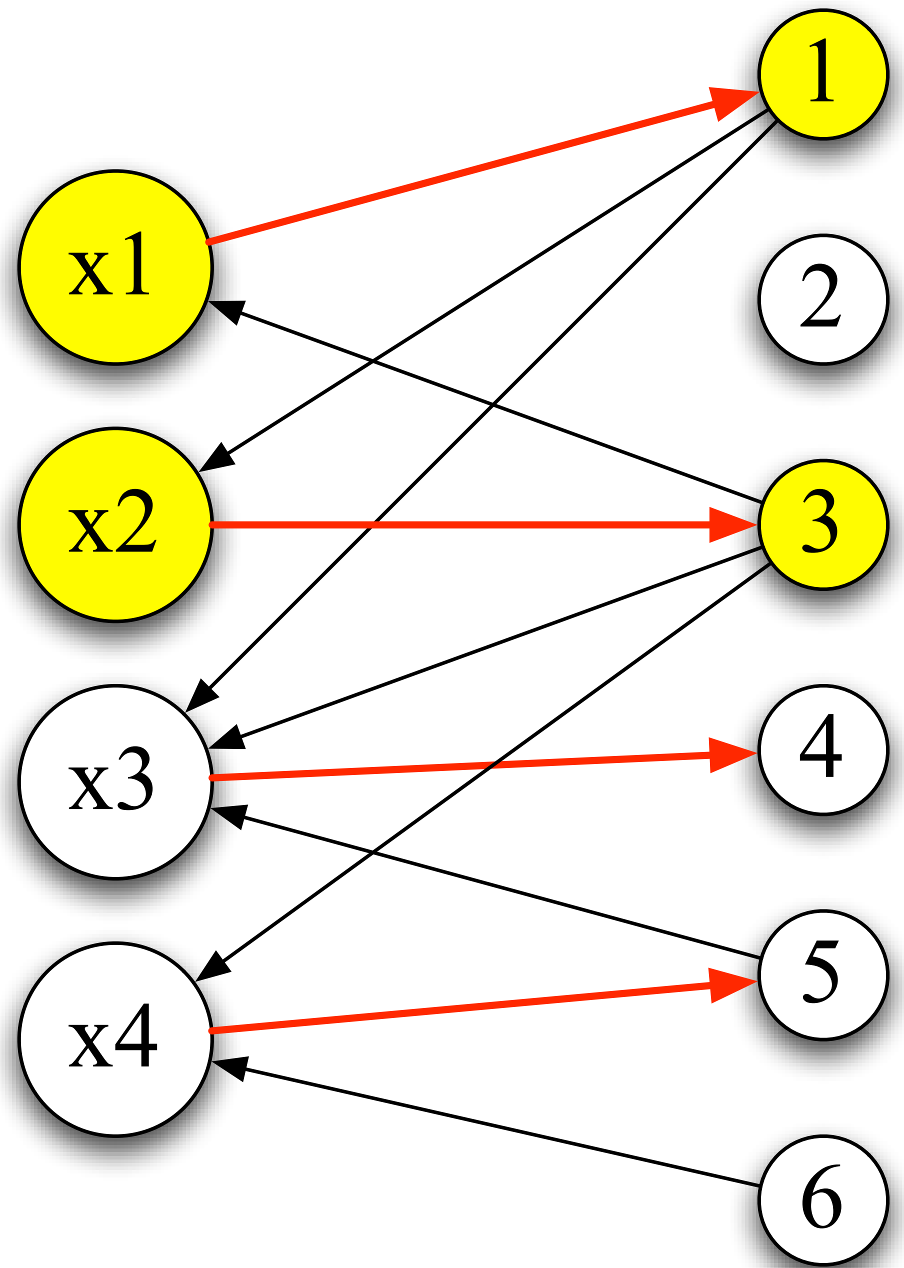
Maximal matching

- Reverse unmatched edges



Maximal matching

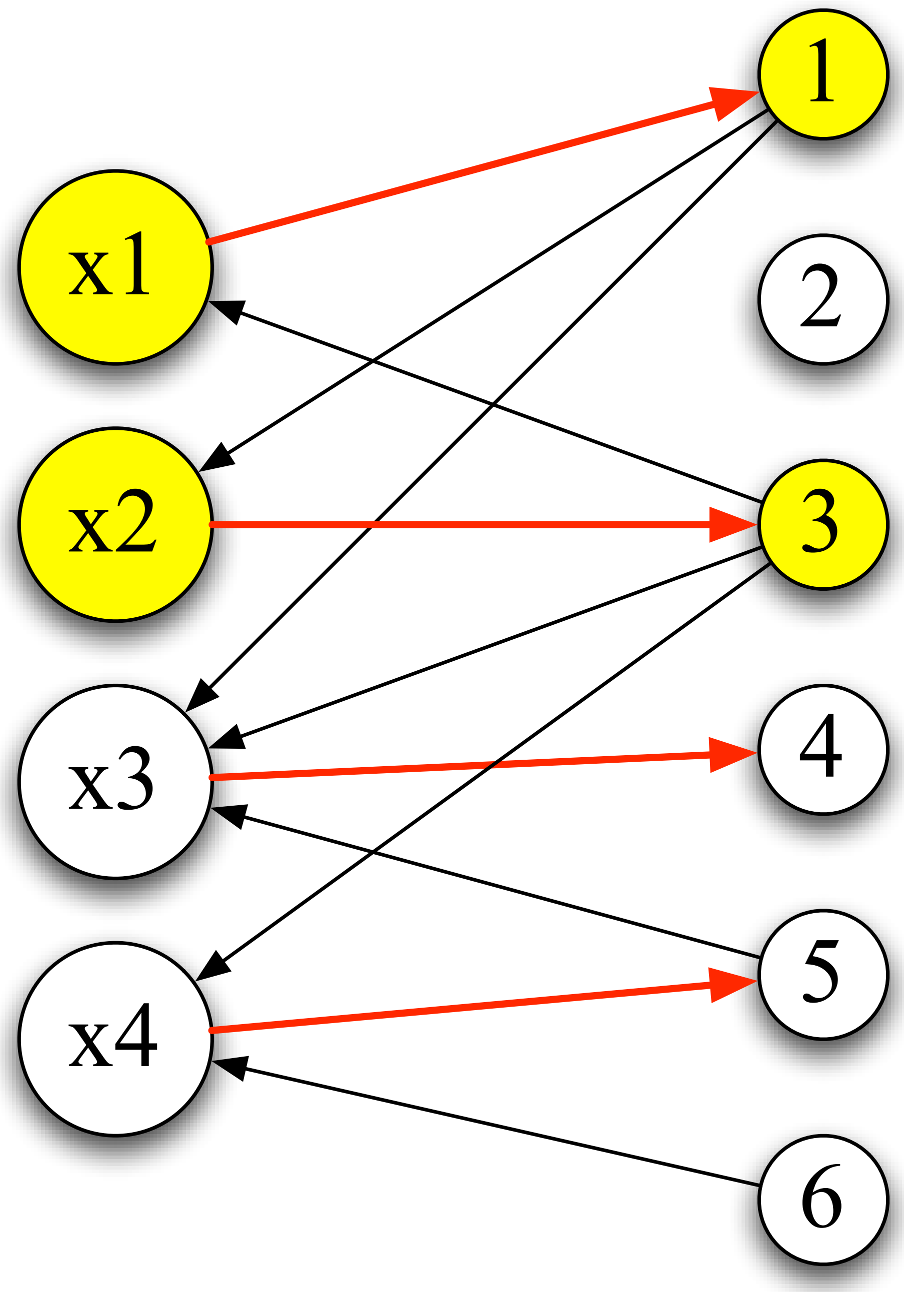
- Reverse unmatched edges
- Compute strongly connected components (SCCs)



Maximal matching

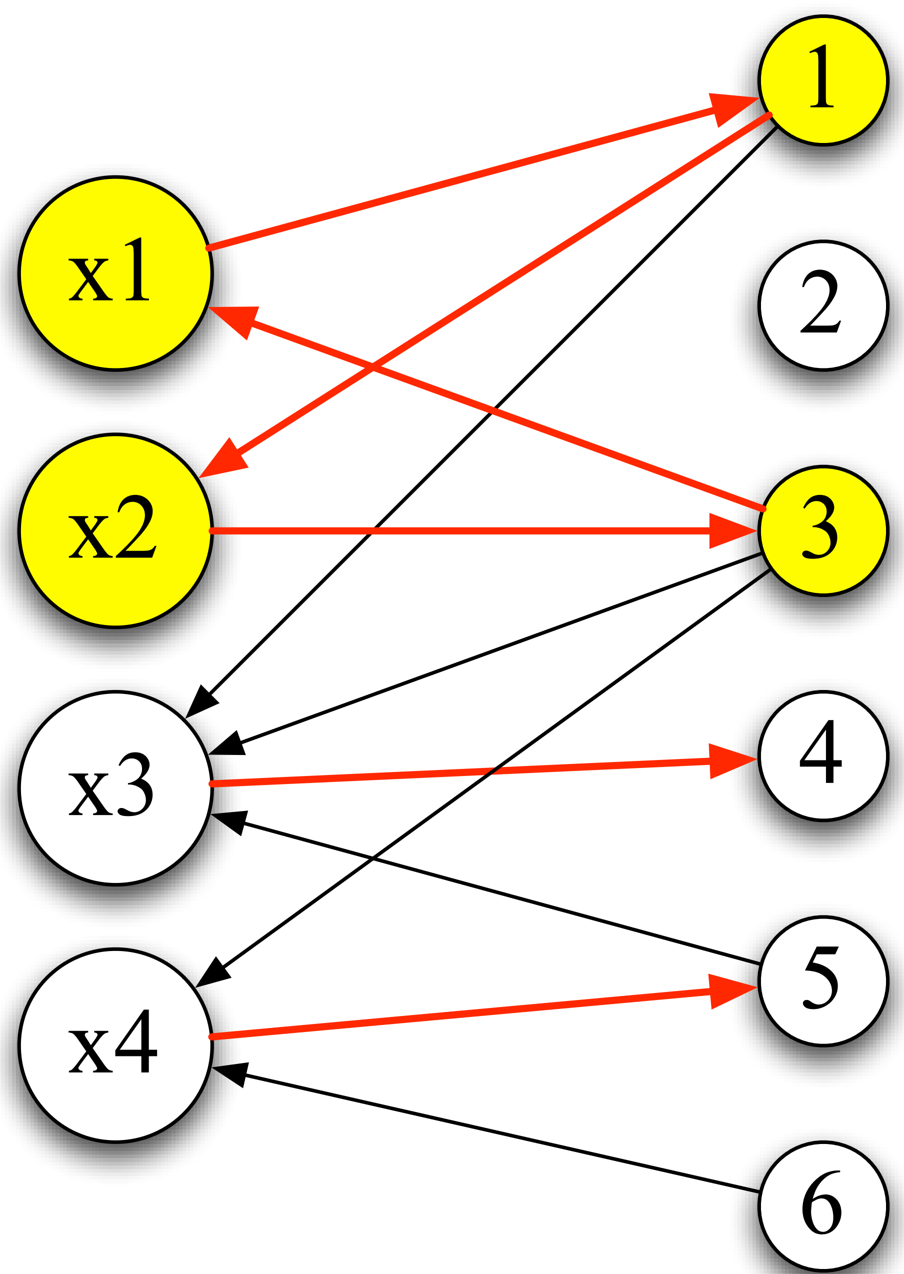
- Reverse unmatched edges
- Compute strongly connected components (SCCs)

maximal set of nodes where each node is reachable from any other node



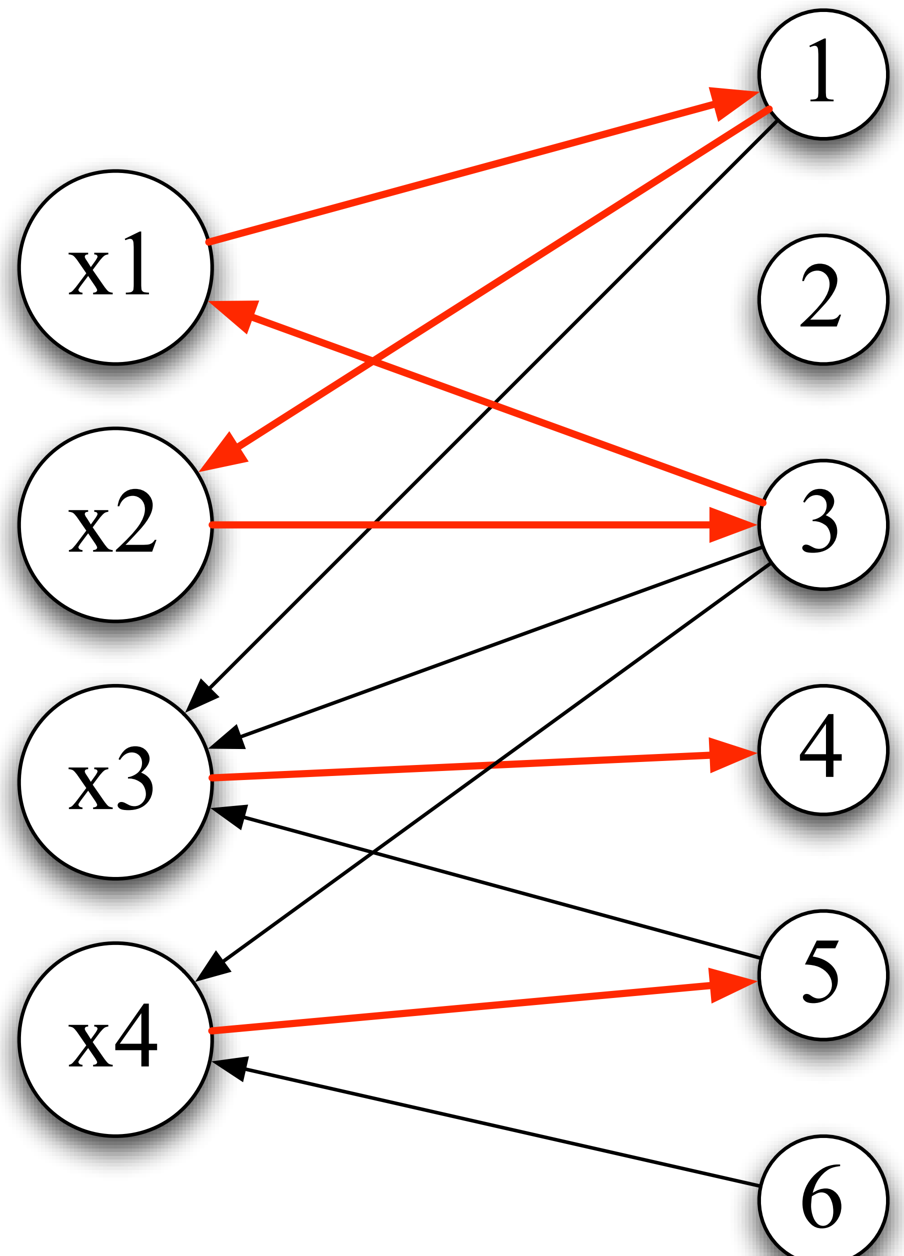
Maximal matching

- Reverse unmatched edges
- Compute strongly connected components
- Edges in one SCC are on an M-alt. circuit



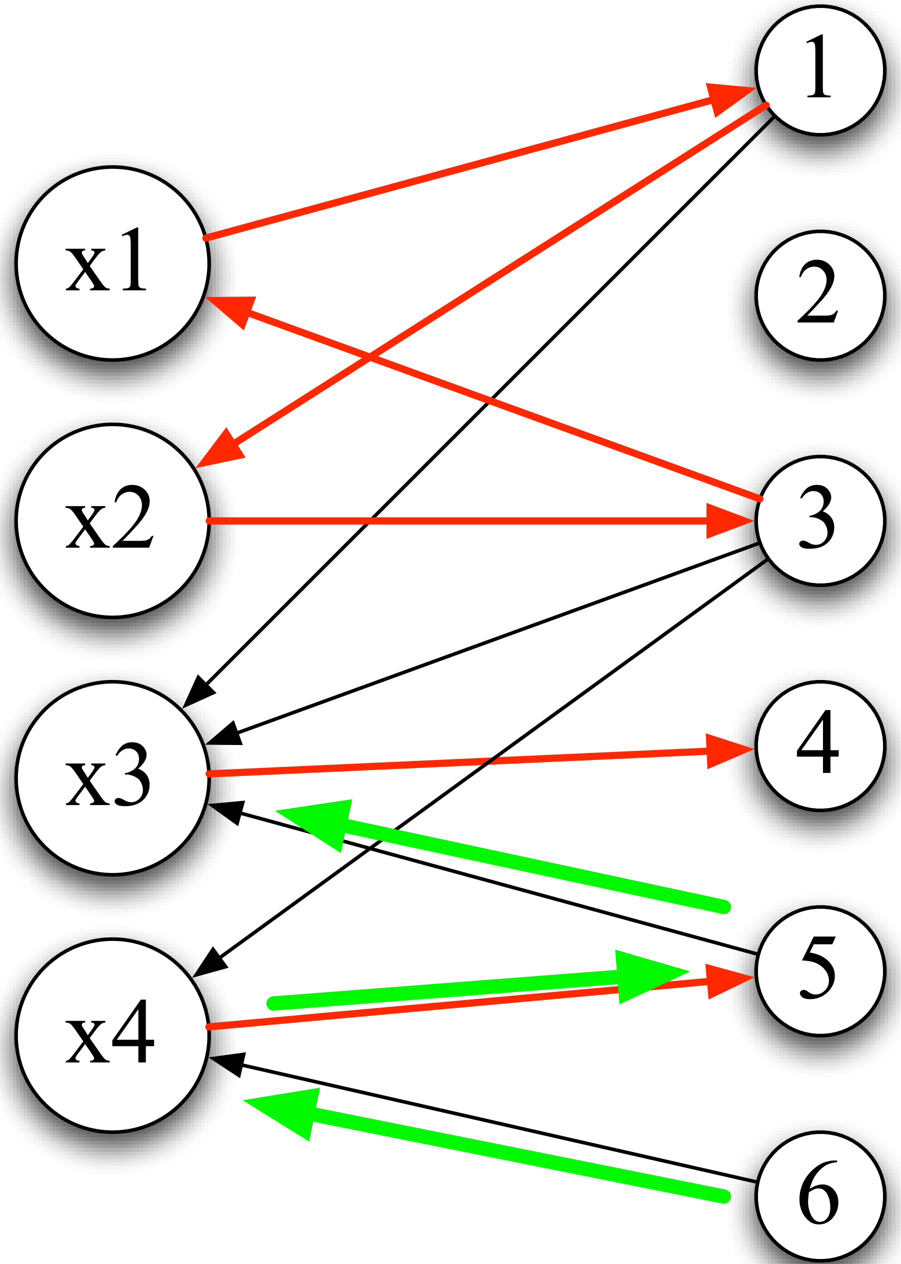
Maximal matching

- Edges on a directed path starting at a free vertex



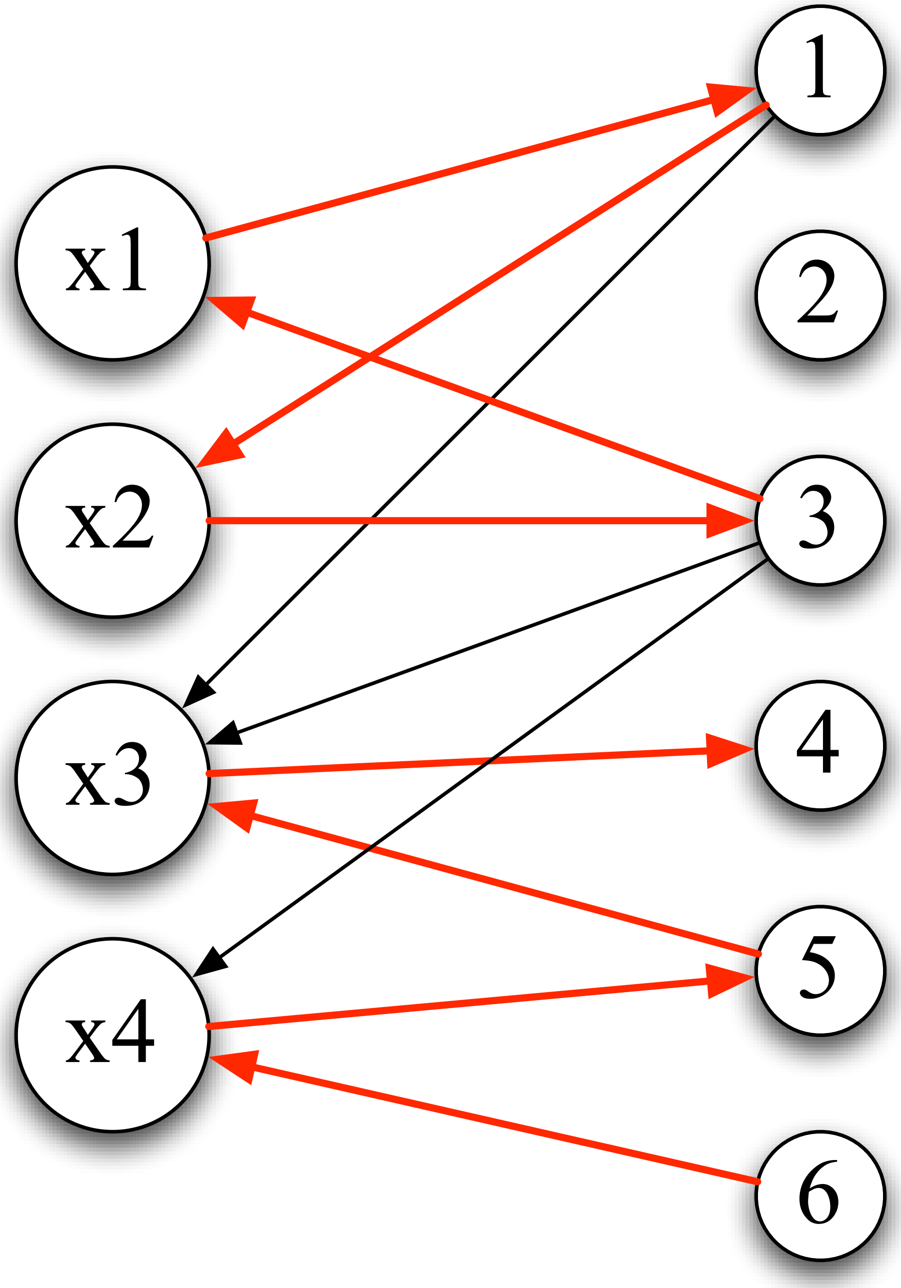
Maximal matching

- Edges on a directed path starting at a free vertex
- Breadth-first search



Maximal matching

- Edges on a directed path starting at a free vertex
- Breadth-first search



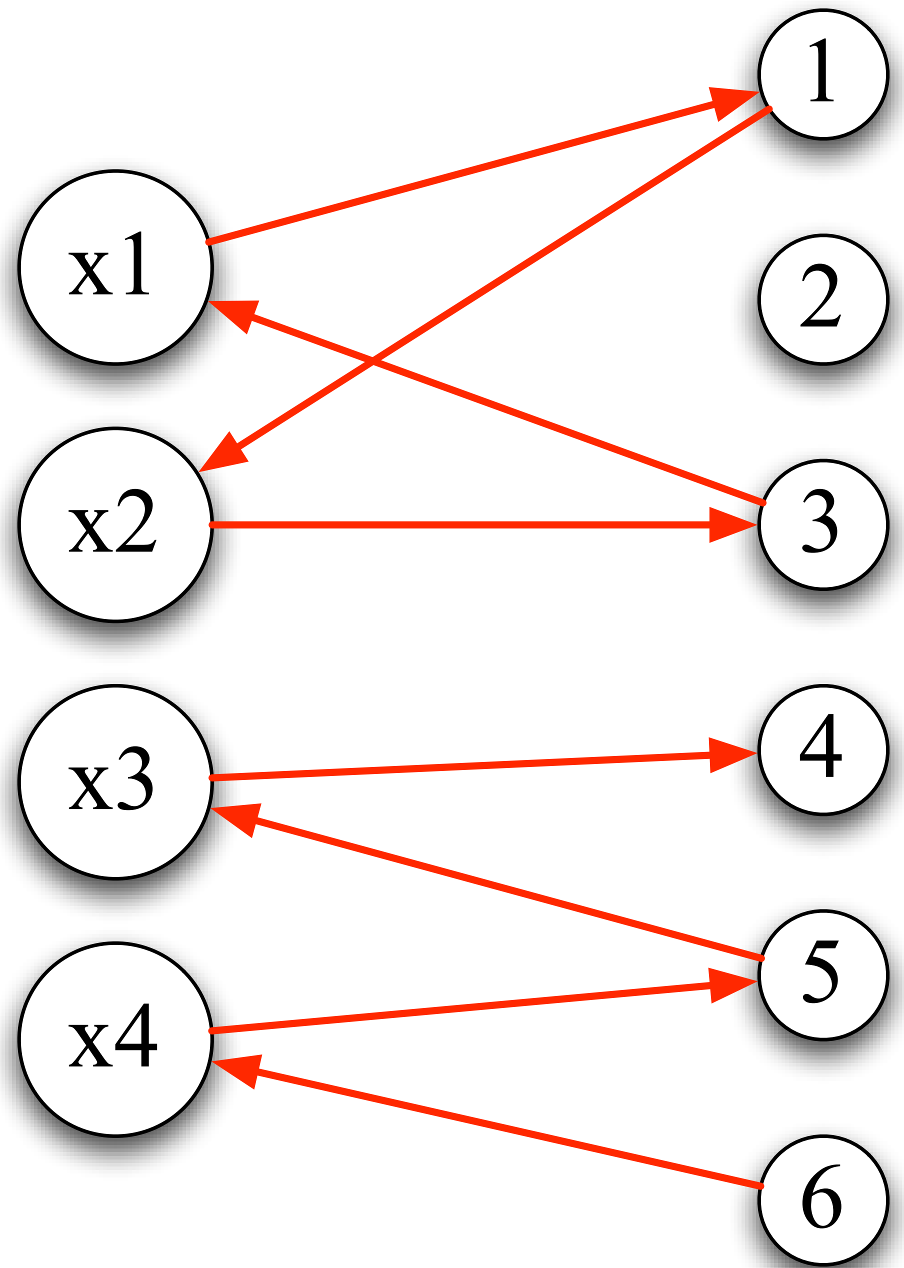
Compute new domains

$x1 \in \{1,3\}$

$x2 \in \{1,3\}$

$x3 \in \{4,5\}$

$x4 \in \{5,6\}$



Complete algorithm

- Construct the variable-value graph
- Compute maximal matching
- Orient the graph
- Find M-alternating cycles (SCCs)
- Find even M-alternating paths (graph search)
- Remove edges + narrow domains

Asymptotic Runtime Analysis

- Construction: $O(n+m)$
- Matching: $O(mn^{0.5})$
- SCC: $O(n+m)$ (Tarjan, 1972)
- Directed path: $O(m)$

- This gives overall complexity

  $O(mn^{0.5}) = O(n^{2.5})$

Optimizations

- Consider not only consistent and inconsistent edges, but also *vital* edges
- A vital edge is one that is contained in *all* matchings
- Vital edge between x and j means x must be assigned to j

Optimization: Incrementality

- Keep the variable-value graph between invocations
- When the propagator is run again, update the matching accordingly:
  - remove edges no longer present
  - if matched edge is removed, compute new maximal matching
  - otherwise, just find alternating paths & cycles

Alternative approach: Hall sets

- A set of variables $\{x_1,...,x_n\}$ is a **Hall set** of size *n* for a store *s* iff

  v=s(x_1) ∪ … ∪ s(x_n) has *n* elements

- Intuition:
  - no other variables can take values from *v*
  - Hall set of size 1: assigned variable

Alternative approach: Hall sets

- Propagation: if there is a Hall set, prune values for variables in hall set from all other variables
- Problem: how to find hall sets efficiently?
- Obviously: matching theory
- Other methods available (network flow algorithms)

Hall intervals

- A set of variables $\{x_1,\ldots,x_n\}$ is a **Hall interval** of size *n* for a store *s* iff

  v=range(s(x_1) ∪ … ∪ s(x_n)) has *n* elements

  where range(s) = { x→ [min(s(x))…max(s(x))] for all x }

- Intuition:
  - no other variables can take values from *v*
  - Hall set of size 1: assigned variable

Bounds consistency

- Efficient algorithms
  - based on Hall intervals O(n log n)

    (Puget, 1998) (Lopez-Ortiz & Quimper & al., 2003)
  - based on graphs & matchings O(n)

    (Mehlhorn & Thiel, 2000)

Bounds vs. domain consistency

- **Bounds:** only consider endpoints
- **Domain:** consider whole domains
- **Difference:** often a factor of O(m) if m is the size of the domains!
- **Sometimes:** bounds consistency polynomial, domain consistency NP hard!

Extension: Global Cardinality

- For each value, give lower and upper bound on how often it may be taken by the variables.
- distinct(x1,...,xn) = gcc(x1,...,xn,0,...,0,1,...,1)

  (all values at least 0 times and at most once)
- Algorithm by Régin (very similar to distinct)

Does it pay off?

- In most cases, domain consistent distinct leads to considerably smaller search trees than naive version
- In some cases, bounds consistent distinct is “just as strong”

  (Schulte, Stuckey, 2005)
- Try it out!

Element Constraint

Constraints defined by extension

Slides by Christian Schulte

Modeling Price

- Suppose variable modeling location in warehouse
  - values model good to be stored at location
  - different goods have different prices
- How to propagate the price while the variable is not yet assigned a good?
- Very common: map variable to variable according to given values

Example

- Assume goods represented by numbers 0, 1, 2, 3
- Prices
    - good 0: price 10
    - good 1: price 15
    - good 2: price 5
    - good 3: price 12

Model by Reification

```
BoolVar b0 = new BoolVar(this);
...
rel(this, g, IRT_EQ,  0, b0);
rel(this, p, IRT_EQ, 10, b0);
rel(this, g, IRT_EQ,  1, b1);
rel(this, p, IRT_EQ, 15, b1);
rel(this, g, IRT_EQ,  2, b2);
rel(this, p, IRT_EQ,  5, b2);
rel(this, g, IRT_EQ,  3, b3);
rel(this, p, IRT_EQ, 12, b3);
"b0 + b1 + b2 + b3 = 1";
```

- **Tedious:** several goods can have same price...
- **Inefficient:** too many propagators...
- **Propagation:** if propagators run to fixpoint, domain-consistency!

The Element Constraint

- Element constraint $a[x] = y$
  - array of integers a
  - variables x and y
  - value of y is value of a at x-th position
  - in particular: $0 \le x <$ elements in a
- In Gecode/J
  - element(this, a, x, y);
  - also for arrays of variables

Model with Element

```
int prices[] = {10,15,5,12};
element(this, prices, g, p);
```

- Just single propagator!
- Also works if same integer occurs multiply in array

Propagating Element

- We insist on domain-consistency
  - bounds-consistency too weak
- For $a[x] = y$ and store s propagate
  - if $j \in s(y)$ then keep all k from $s(x)$ with $j=a[k]$
  - if $k \in s(x)$ then keep all j from $s(y)$ with $j=a[k]$
  - remove all other values

Implementing Element...

- Fundamental requirement: new domains must be computed in order!
- Iterate over all elements $k \in s(x)$

  $\{ a[k] \mid k \in s(x) \} \cap s(y)$

- Iterate k from 0 to n-1:= width of a
  - construct new domain for x
  - if $a[k] \in s(y)$ then keep k
  - requires intersection and sorting

Problems...

- Array in element constraints can be very large
  - always iterate over entire array
  - always sort (or maintain sorted data structure)
  - always compute intersection
- We can do better!

Running Example

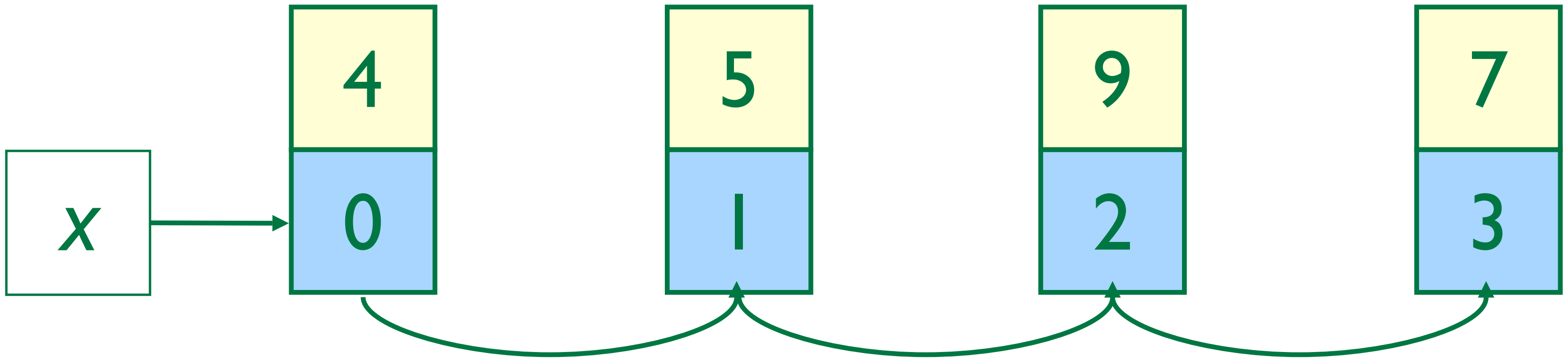
- Consider $a[x] = y$ with
  - $a = (4,5,9,7)$
  - $s(x) = \{1,2,3\}$
  - $s(y) = \{2...8\}$
- Propagation yields
  - $s(x) = \{1,3\}$
  - $s(y) = \{5,7\}$

Approach

- Construct data structure
  - contains pairs of (i,$a[i]$)
  - allows traversal for increasing i
  - allows traversal for increasing $a[i]$
  - allows removal of pairs (later)

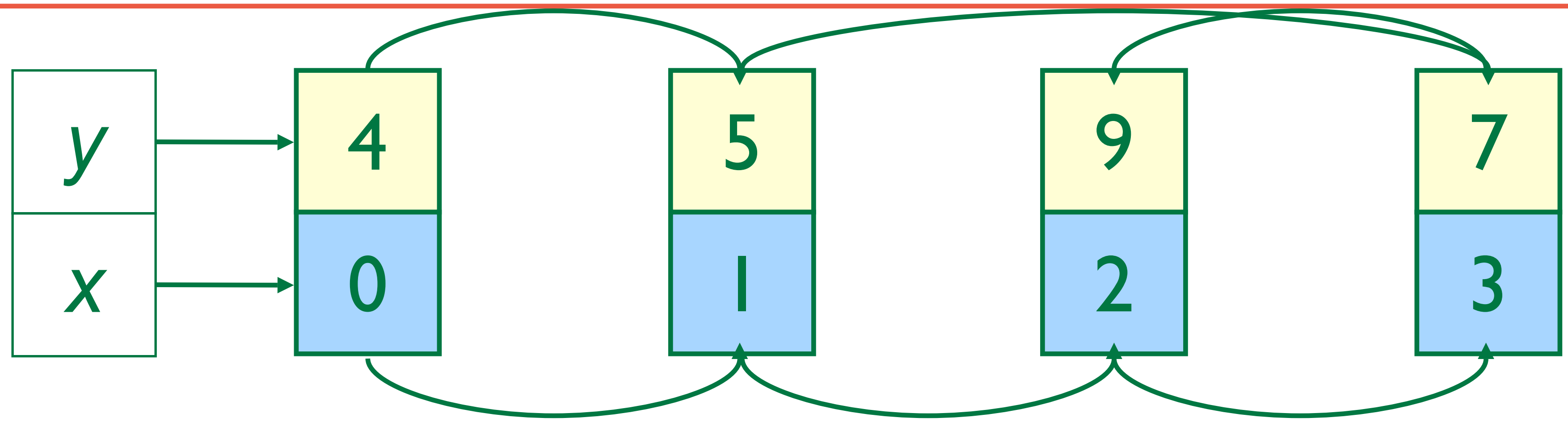
- Data structure constructed initially

Datastructure Construction



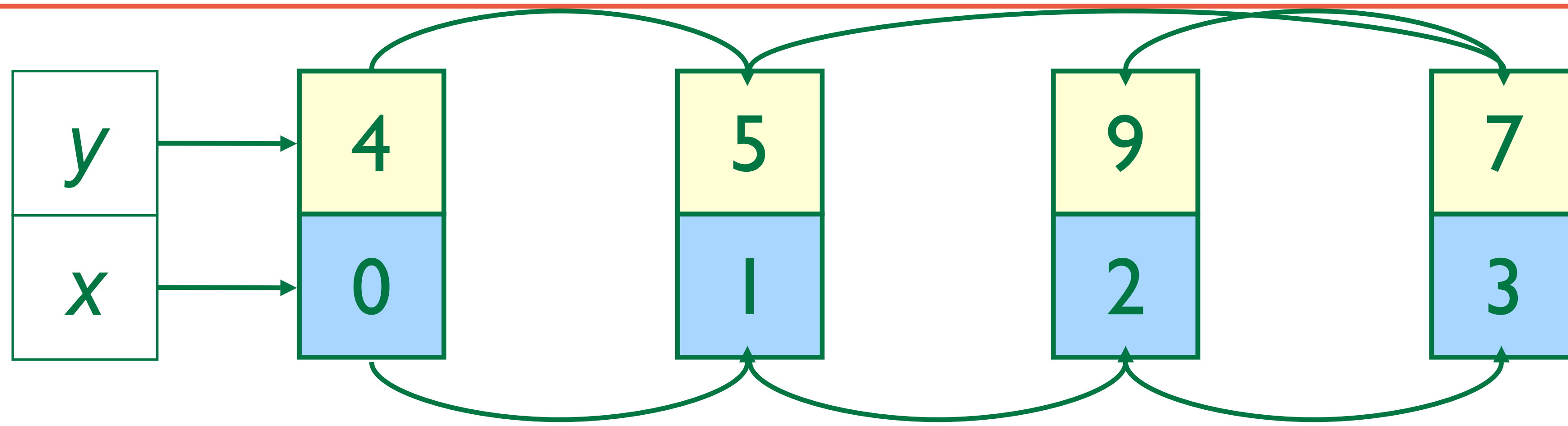
- Iterate over all *i* between 0 and 3
  - create node (*i*, *a*[*i*])
  - create links in order of creation (*x*-links)

Datastructure Construction



- Create links for $a[i]$ values in increasing order (y-links)
- sort and create links

Datastructure Invariant



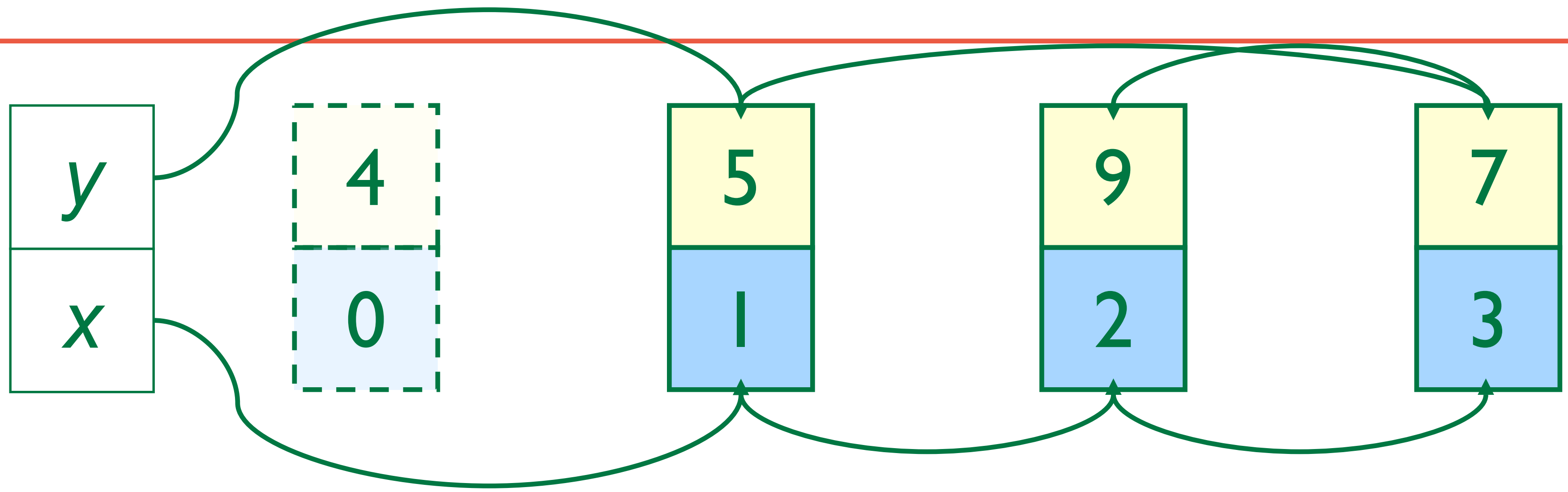
- Datastructure allows iteration
  - *i* values in order: follow *x*-links
  - *a*[*i*] values in order: follow *y*-links

Propagation

- **Follow x-links and iterate values in $s(x)$**
  - if value not in $s(x)$, remove node
- **Follow y-links and iterate values in $s(y)$**
  - if value not in $s(y)$, remove node

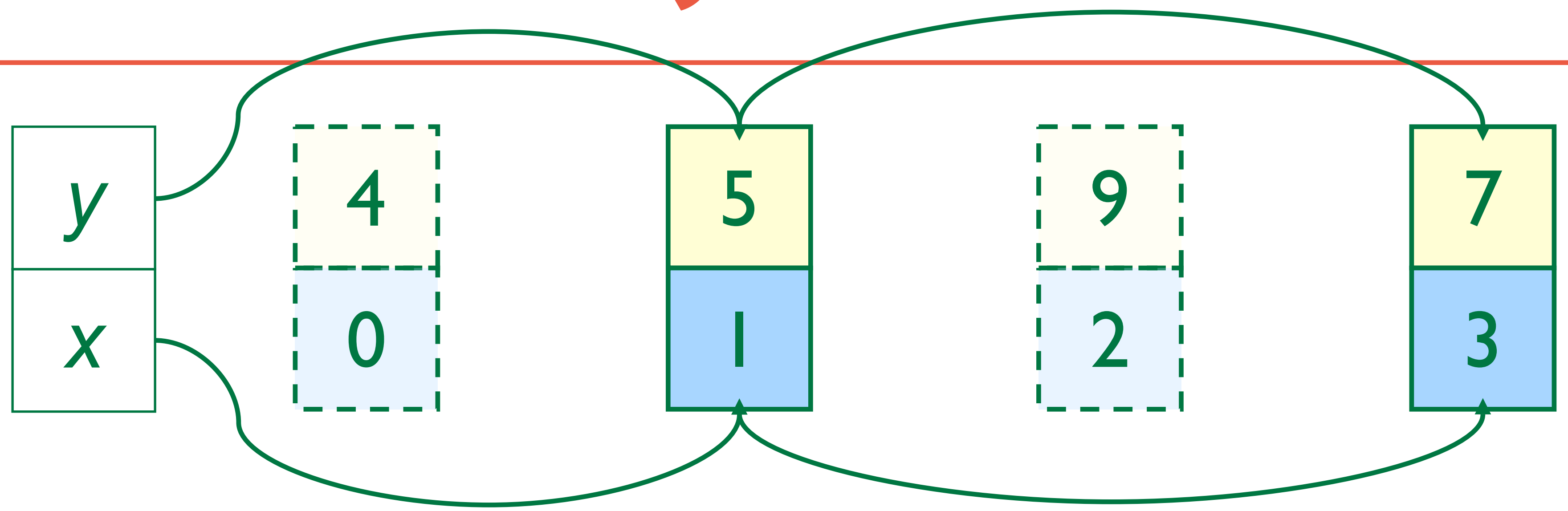
- **Result:** nodes with correct values remain for both x and y
  - in increasing order!

Follow x-links...



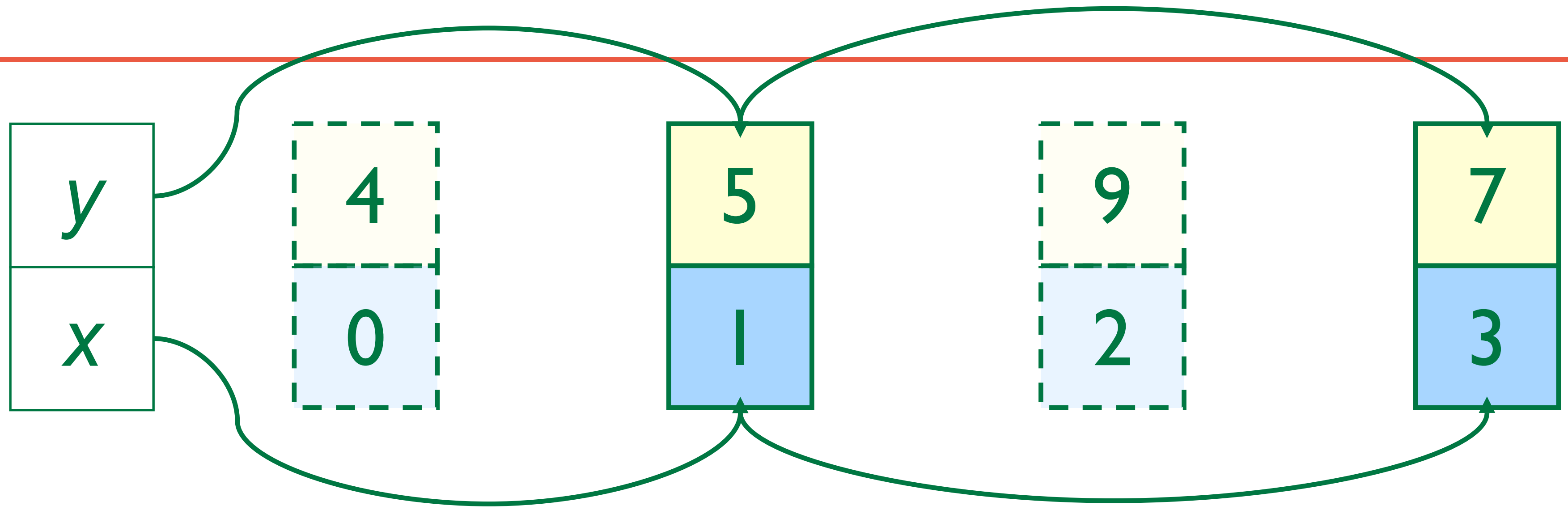
- Store $s(x) = \{1,2,3\}$
- Remove node for 0
  - by relinking
  - constant time: doubly-linked lists

Follow y-links...



- Store $s(y) = \{2,\ldots,8\}$
- Remove node for 9
  - by relinking
  - constant time: doubly-linked lists

Read-off Variable Domains



- New store: $s(x) = \{1,3\}$, $s(y) = \{5,7\}$
- By just following respective links
  - are sorted
  - are smaller than original domains

Incremental Propagation

- One option: destroy data structure
- Better: keep data structure for next propagator invocation
  - propagators with state
- Incremental propagation
  - construction only initially
  - sorting only initially
  - traversing never for full number of array elements

Summary: Element

- Important constraint for mapping variables to values
  - cost functions
  - arbitrary constraints defined extensionally

- Important for propagation
  - maintain clever data structure
  - make propagation incremental

Propagators in Gecode/J

Propagators in Gecode/J

- Not a function Store → Store
- Compute by **modifying** a space in place
- **Subscribe** to variables (event sets)
- Return **status message**
- Have to be **monotonic**
- Have to **check for failure**

Views

- **Wrapper class** for variables
- Provide **read/write** access to domain
- Provide **subscription/cancel** methods

Propagation conditions

- Gecode's way of specifying **event sets**
- For IntVars:
  - `PC_INT_VAL` when variable is assigned
  - `PC_INT_BND` when variable bounds change
  - `PC_INT_DOM` when variable comain changes
- subscribe to one PC per variable

Example: nq

```
class Nq extends BinaryPropagator<IntVarView> {
  public Nq(Space s, IntVarView x0, IntVarView y0) {
    super(s, x0, y0, PC_INT_VAL);
  }
  public Nq(Space s, Boolean share, Nq p) {
    super(s, share, p);
  }
  ...
}
```

Example: nq

```
class Nq extends BinaryPropagator<IntVarView> {
  public Nq(Space s, IntVarView x0, IntVarView y0) {
    super(s, x0, y0, PC_INT_VAL);
  }
  public Nq(Space s, Boolean share, Nq p) {
    super(s, share, p);
  }
  ...
}
```

Example: nq

```
class Nq extends BinaryPropagator<IntVarView> {
  public Nq(Space s, IntVarView x0, IntVarView y0) {
    super(s, x0, y0, PC_INT_VAL);
  }
  public Nq(Space s, Boolean share, Nq p) {
    super(s, share, p);
  }
  ...
}
```

Status messages

- ES_FAILED
- ES_FIX
- ES_NOFIX
- ES_SUBSUMED

Example: nq, propagation

```
class Nq extends BinaryPropagator<IntVarView> {
    public ExecStatus propagate(Space home) {
      if (x.assigned()) {
        if (y.nq(home, x.min()).failed()) return ES_FAILED;
      } else { // y is assigned
        if (x.nq(home, y.min()).failed()) return ES_FAILED;
      }
      return ES_FIX;
    }
}
```

Example: nq, propagation

```
class Nq extends BinaryPropagator<IntVarView> {
    public ExecStatus propagate(Space home) {
      if (x.assigned()) {
        if (y.nq(home, x.min()).failed()) return ES_FAILED;
      } else { // y is assigned
        if (x.nq(home, y.min()).failed()) return ES_FAILED;
      }
      return ES_FIX;
    }
}
```

Example: nq, propagation

```
class Nq extends BinaryPropagator<IntVarView> {
    public ExecStatus propagate(Space home) {
      if (x.assigned()) {
        if (y.nq(home, x.min()).failed()) return ES_FAILED;
      } else { // y is assigned
        if (x.nq(home, y.min()).failed()) return ES_FAILED;
      }
      return ES_FIX;
    }
}
```

Example: nq, propagation

```
class Nq extends BinaryPropagator<IntVarV
    public ExecStatus propagate(Space hom
      if (x.assigned()) {
        if (y.nq(home, x.min()).failed()) return ES_FAILED;
      } else { // y is assigned
        if (x.nq(home, y.min()).failed()) return ES_FAILED;
      }
      return ES_FIX;
    }
}
```

always check for failure!

Example: nq, propagation

```
class Nq extends BinaryPropagator<IntVarV
    public ExecStatus propagate(Space hom
        if (x.assigned()) {
            if (y.nq(home, x.min()).failed()) return ES_FAILED;
        } else { // y is assigned
            if (x.nq(home,              ailed()) return ES_FAILED;
        }
        return ES_FIX;
    }
}
```

always check for failure!

better idea?

Example: nq, propagation

```
class Nq extends BinaryPropagator<IntVarView> {
    public ExecStatus propagate(Space home) {
      if (x.assigned()) {
        if (y.nq(home, x.min()).failed()) return ES_FAILED;
      } else { // y is assigned
        if (x.nq(home, y.min()).failed()) return ES_FAILED;
      }
      return ES_SUBSUMED;
    }
}
```

Example: nq, propagation

```
class Nq extends BinaryPropagator<IntVarView> {
    public ExecStatus propagate(Space home) {
      if (x.assigned()) {
        if (y.nq(home, x.min()).failed()) return ES_FAILED;
      } else { // y is assigned
        if (x.nq(home, y.min()).failed()) return ES_FAILED;
      }
      return ES_SUBSUMED;
    }
}
```

Full example: `QueensJavaPropagator.java`

Summary

Propagation algorithms

- Trade-off between **propagation strength** and algorithmic **efficiency**
- Linear equations: consider real-valued solutions, achieve **bounds-R** consistency
- **Distinct:** efficient algorithms for bounds and domain consistency
- To get an efficient algorithm, **deep insight** into structure of the constraint is needed!

Pointers

- Distinct:

  W.-J. van Hoeve. **The Alldifferent Constraint: a Systematic Overview**

  http://www.cs.cornell.edu/~vanhoeve/papers/alldiff.pdf

- Linear constraints:

  Krysztof R. Apt. **Principles of Constraint Programming**

- Schulte, Stuckey. **When Do Bounds and Domain Propagation Lead to the Same Search Space**. TOPLAS, 2005.

Outlook

- We know how to propagate, so how does search work?
- spaces, search engines, recomputation, gist

Thank you.