Other approaches

Marco Kuhlmann & Guido Tack

Lecture 11

Plan for today

- CSP-based propagation
- Constraint-based local search
- Linear programming

CSP-based propagation

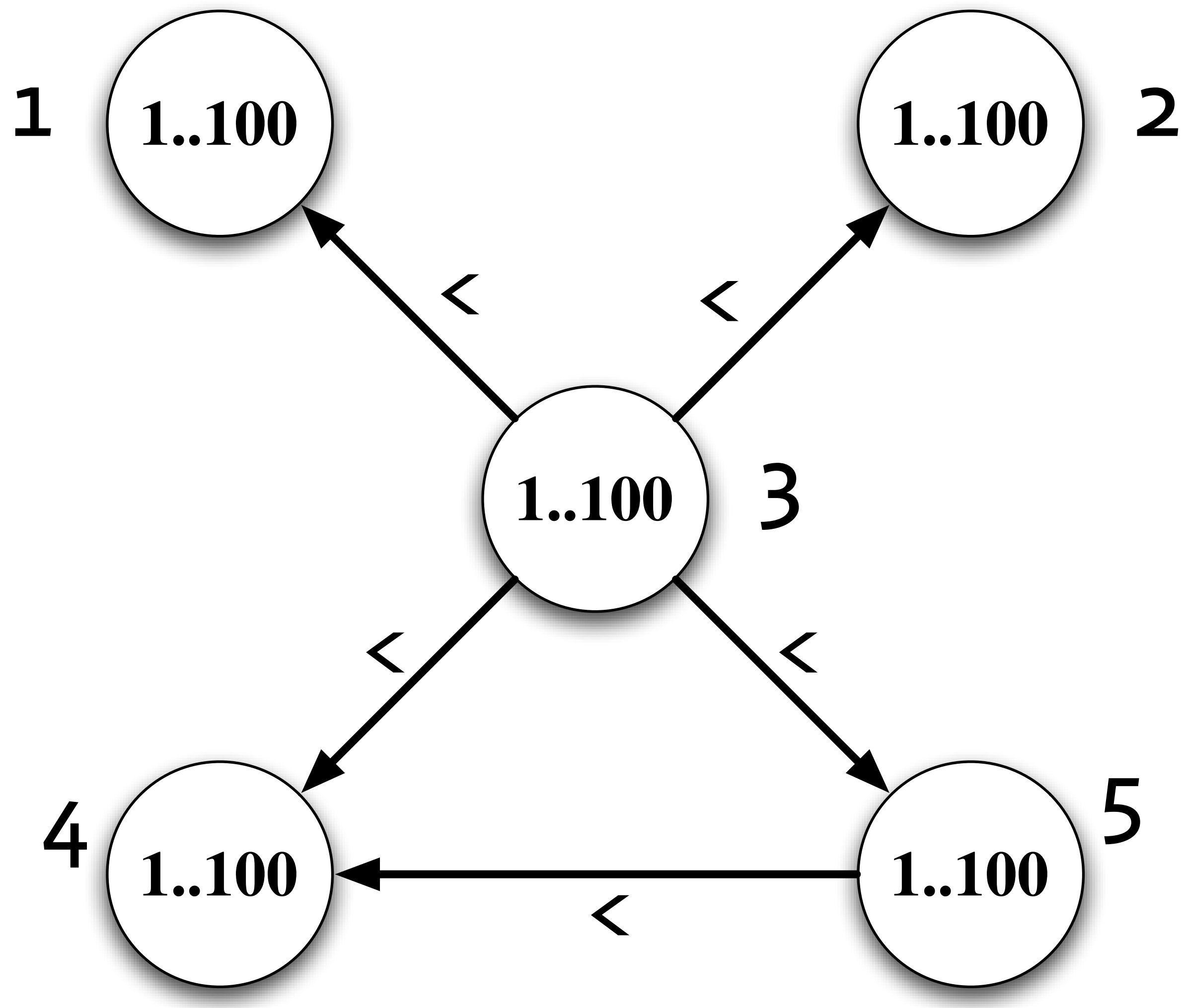
History of CSP-based propagation

- **AI research**
- **backtracking** (generate and test)
- **arc consistency**
  - a.k.a. domain consistency
- **path consistency**
- **k-consistency**

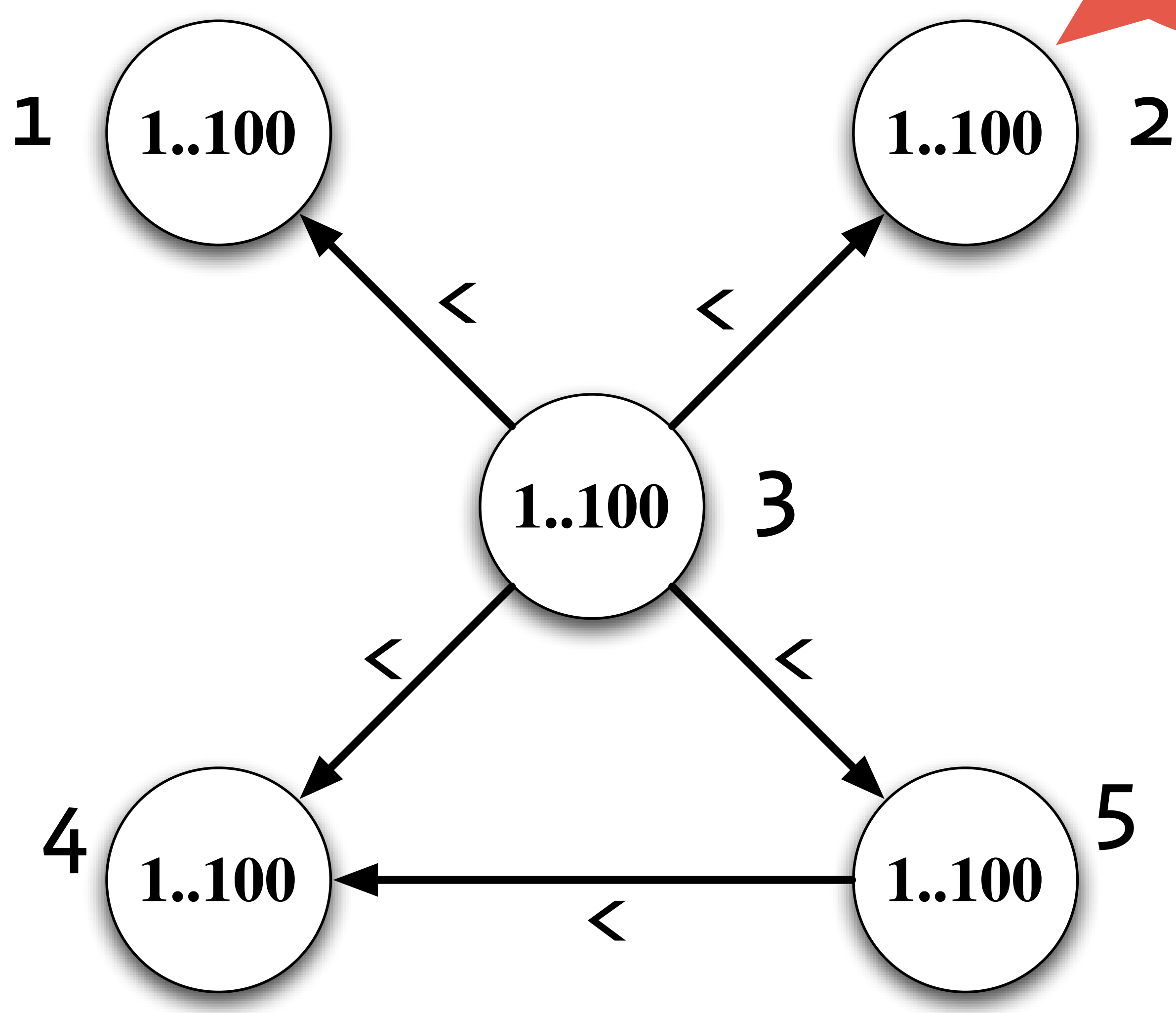
Reminder: CSP

- A **constraint satisfaction problem** contains
  - variables with associated domains
  - constraints (relations) between variables
- Here: only **binary constraints**
  - full relation on all but two variables

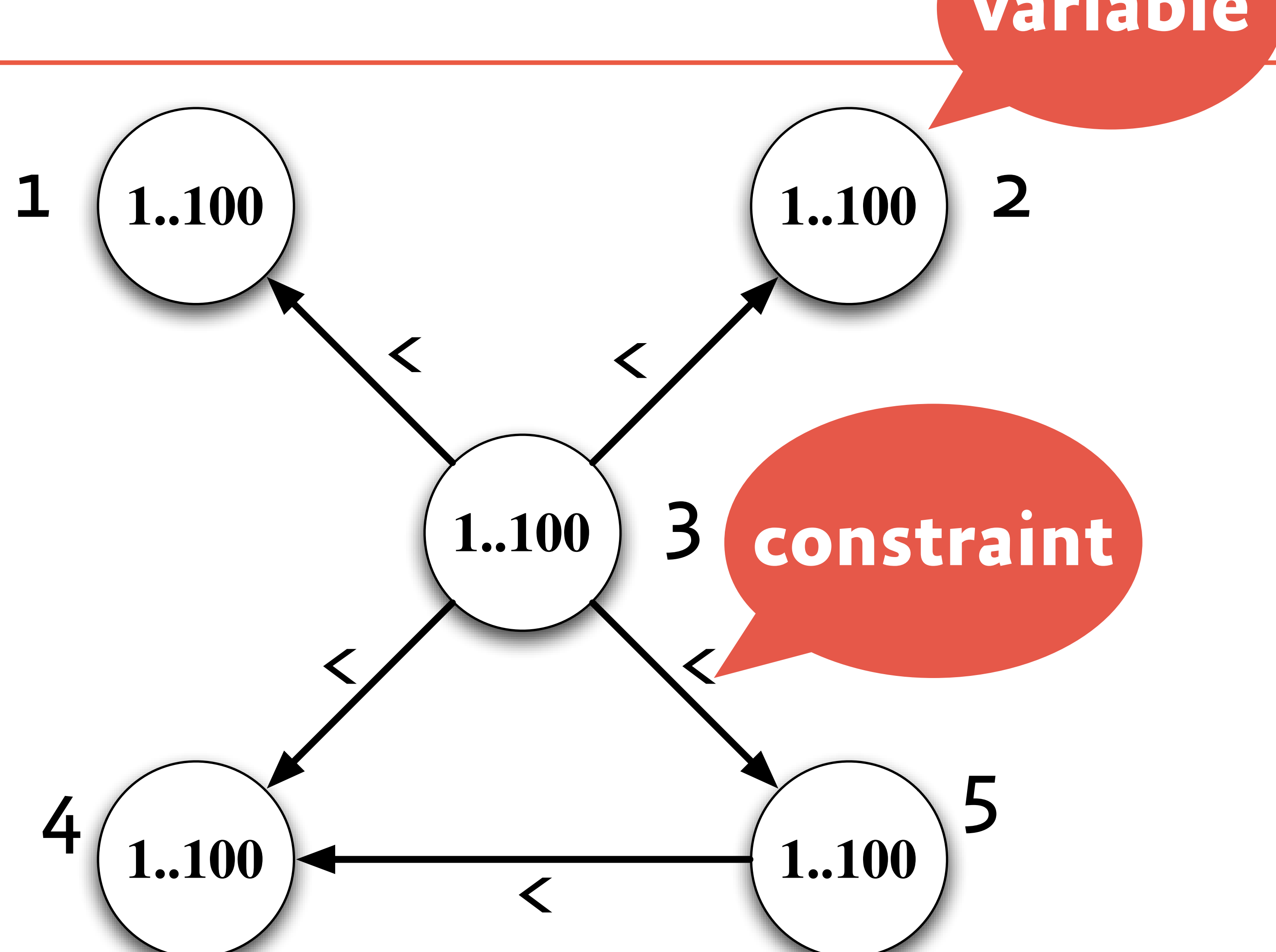
Constraint Network



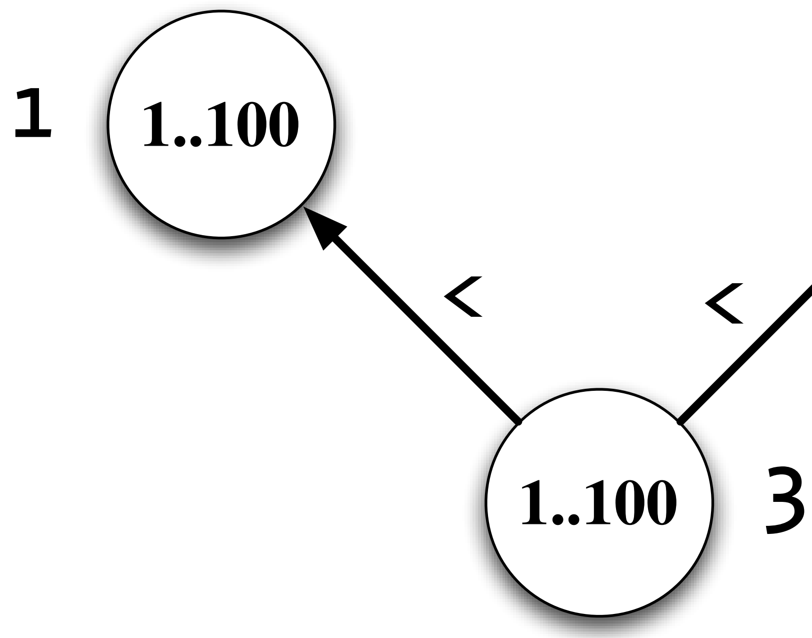
Constraint Network



Constraint Network

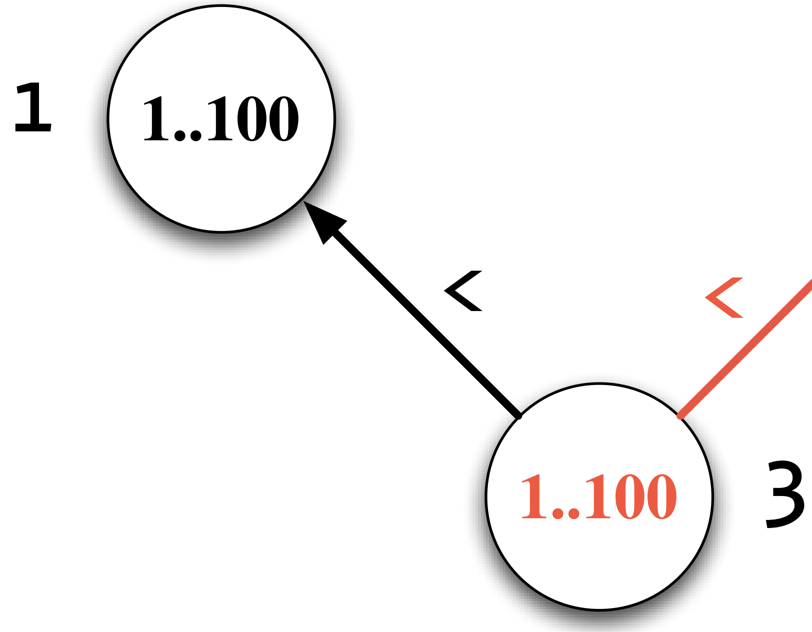


Arc inconsistency



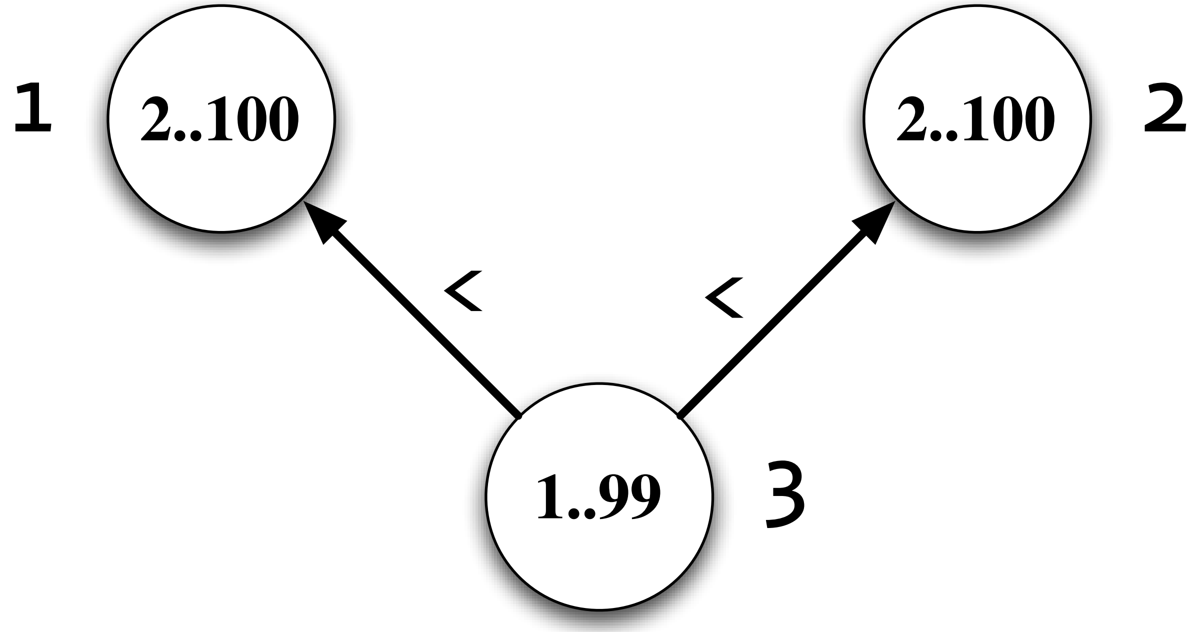
- No *x* in D_3 such that `x<1`
- The arc 3-2 is inconsistent

Arc inconsistency



- No x in D_3 such that `x<1`
- The arc 3-2 is inconsistent

Arc consistency



- All arcs are consistent
- The network is arc consistent

AC-1

```
revise(i,j)
  modified := false
  for x in Di do
    if ∄ y in Dj with (x,y) in ci,j then
      Di := Di - {x}
      modified := true
  return modified
```

```
ac1( )
  Q := {(i,j) for all ci,j}
  do
    change := false
    for (i,j) in Q do
      change |= revise(i,j)
  while (change)
  return
```

AC-1

```
revise(i,j)
  modified := false
  for x in Di do
    if ∄ y in Dj with (x,y) in ci,j then
      Di := Di - {x}
      modified := true
  return modified
```

```
ac1( )
  Q := {(i,j) for all ci,j}
  do
    change := false
    for (i,j) in Q do
      change |= revise(i,j)
  while (change)
  return
```

AC-1

```
revise(i,j)
  modified := false
  for x in Dᵢ do
    if ∄ y in Dⱼ with (x,y) in cᵢ,ⱼ then
      Dᵢ := Dᵢ - {x}
      modified := true
  return modified
```

```
ac1( )
  Q := {(i,j) for all cᵢ,ⱼ}
  do
    change := false
    for (i,j) in Q do
      change |= revise(i,j)
  while (change)
  return
```

Inefficiency: always revise all arcs!

AC-3

```
revise(i,j)
  modified := false
  for x in Dᵢ do
    if ∄ y in Dⱼ with (x,y) in cᵢ,ⱼ then
      Dᵢ := Dᵢ - {x}
      modified := true
  return modified
```

```
ac3( )
  Q := {(i,j) | for all cᵢ,ⱼ}
  while Q not empty do
    remove (k,m) from Q
    if revise(k,m) then
      N := {(i,k) for all cᵢ,ₖ,i≠m}
      Q := Q ∪ N
  return
```

Better: only revise arcs with modified variables

AC-3

```
revise(i,j)
  modified := false
  for x in Dᵢ do
    if ∄ y in Dⱼ with (x,y) in cᵢ,ⱼ then
      Dᵢ := Dᵢ - {x}
      modified := true
  return modified
```

```
ac3( )
  Q := {(i,j) | for all cᵢ,ⱼ}
  while Q not empty do
    remove (k,m) from Q
    if revise(k,m) then
      N := {(i,k) for all cᵢ,ₖ,i≠m}
      Q := Q ∪ N
  return
```

Better: only revise arcs with modified variables

AC-3

```
revise(i,j)
  modified := false
  for x in Dᵢ do
    if ∄ y in Dⱼ with (x,y) in cᵢ,ⱼ then
      Dᵢ := Dᵢ - {x}
      modified := true
  return modified
```

```
ac3( )
  Q := {(i,j) | for all cᵢ,ⱼ}
  while Q not empty do
    remove (k,m) from Q
    if revise(k,m) then
      N := {(i,k) for all cᵢ,ₖ,i≠m}
      Q := Q ∪ N
  return
```

Better: only revise arcs with modified variables

Still: possibly check many constraints several times

AC-2001

- **Idea:**
  - remember the smallest support for each value
  - when support is removed, try to find new support
  - only check bigger supports
- **Benefit:**
  - all possible supports checked at most once

AC-2001

```
revise(i,j)
  modified := false
  for x in Di, Last[i,x,j] not in Dj do
      y = min { n in Dj, n>Last[i,x,j],
                              (x,n) in ci,j }
      if y exists then
         Last[i,x,j] := y
      else
         Di := Di - {x}
         modified := true
  return modified
```

```
ac2001()
    for all i, x in Di, j do
       Last[i,x,j] := min(Dj)-1
    Q := {(i,j) | for all ci,j}
    while Q not empty do
       remove (k,m) from Q
       if revise(k,m) then
          N := {(i,k) for all
          ci,k,i≠m}
          Q := Q ∪ N
    return
```

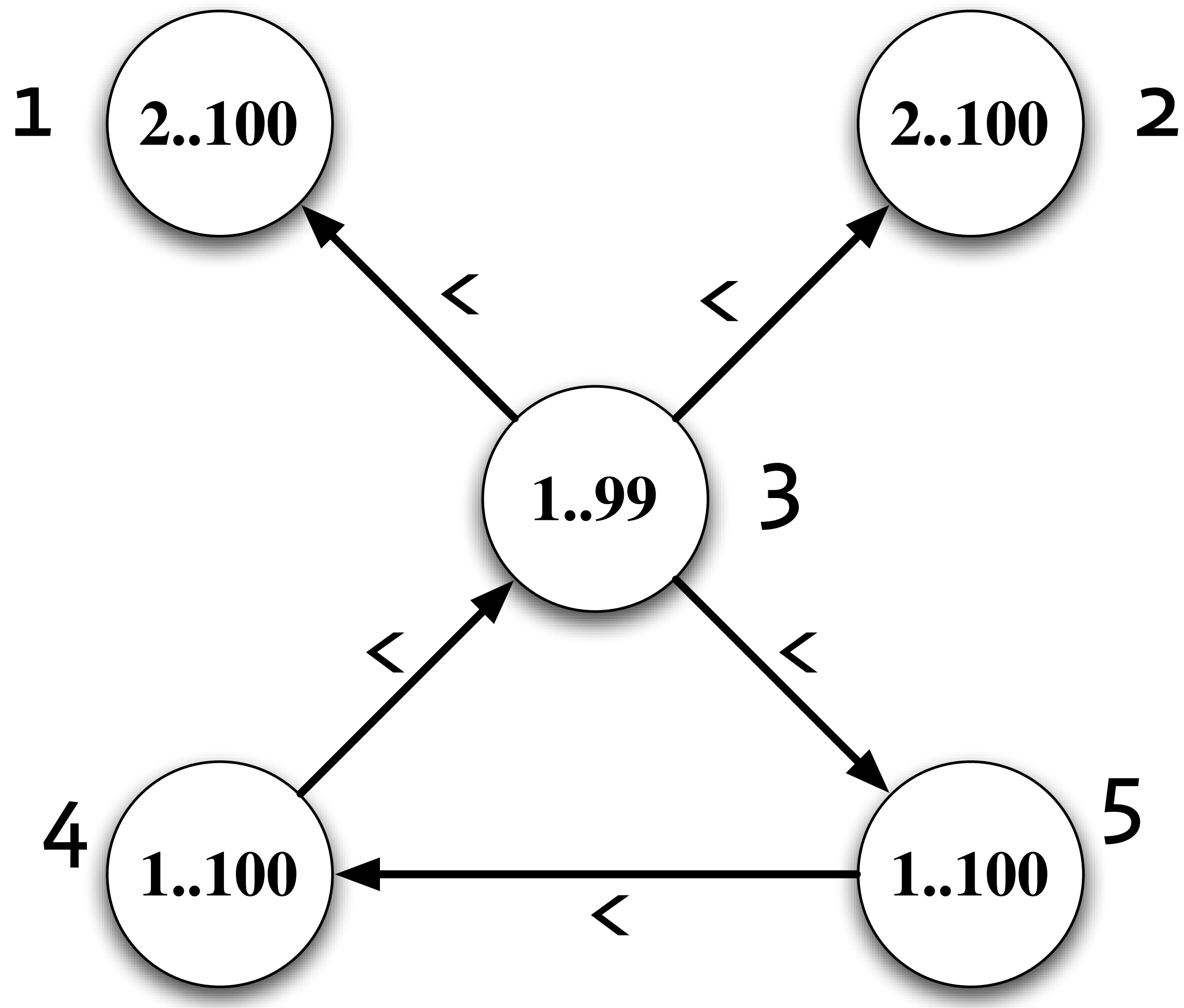
Propagators for arc consistency

- **Given:** constraint *c* on variables *x,y*
- $$\mathrm{ac}_{c,x}(s) = \begin{cases} s(z) & \text{for } x \neq z \\ \{n \in s(x) \mid \exists m \in s(y).\ (n,m) \in c\} & \text{else} \end{cases}$$
- $$\mathrm{ac}_{c,y}(s) = \begin{cases} s(z) & \text{for } y \neq z \\ \{m \in s(y) \mid \exists n \in s(x).\ (n,m) \in c\} & \text{else} \end{cases}$$
- **Propagation achieves arc consistency for *c***

Strong assumptions

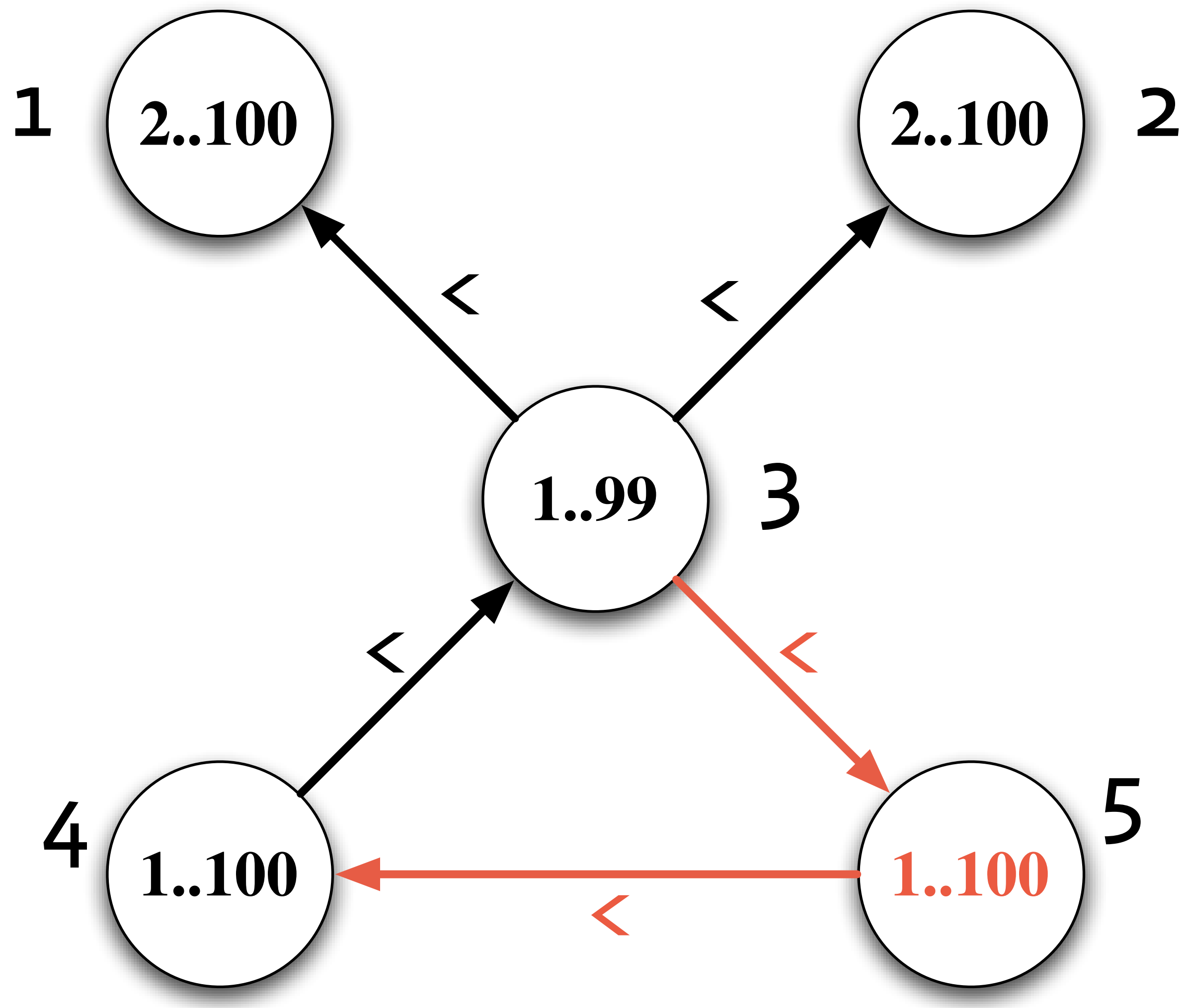
- Consider only **normalized networks**:
  - at most one constraint between each pair of variables
- **Consequence:**
  - (x<y and y<x) is detected as inconsistent by normalization

Path inconsistency



- **Unsatisfiable**
- But takes *O(|D|)* steps to find out!
- **Idea:** combine two constraints

Path inconsistency



- combining $x_3<x_5$ and $x_5<x_4$ yields $x_3<x_4$
- normalization detects that $x_3<x_4$ and $x_3>x_4$ are inconsistent

Path consistency

- Algorithm to make a network **path consistent:**
  - compute combinations, then use AC
- **Problem:**
  - arbitrary combinations of constraints expensive to compute
- Not used in actual CP systems

Literature

- Apt. *Principles of Constraint Programming*. Cambridge University Press, 2003.
- Mackworth. *Consistency in Networks of Relations.* Artificial Intelligence 8(1), 1977.

Constraint-based local search

Local search

- **Idea:**
  - incomplete!
  - start from some assignment
  - move to neighbouring assignments
  - improve overall "quality" of the assignment
- **Quality of an assignment**
  - Number of violated constraints
  - Objective function (for optimization)

Neighbourhoods and moves

- For an assignment *a*, *N(a)* are the **neighbouring assignments**
  - some neighbours may be **forbidden**
  - **legal** neighbours: identified by a function *L*
- Local search performs a **move** from *a* to one of the *N(a)*
  - select where to move using operation *S*

Local search blueprint

```
a  := GenerateInitialAssignment()
a' := a
for k in 1...MaxTrials do
  if f(a)<f(a') then
    a' := a
  a := S(L(N(a), a), a)

return a'
```

Local search blueprint

```
a  := GenerateInitial[illegible]()
a' := a
for k in 1...MaxTrials do
  if f(a)<f(a') then
    a' := a
  a := S(L(N(a), a), a)

return a'
```

cut-off

Local search blueprint

```
a  := GenerateInitialAssignment()
a' := a
for k in 1...MaxTrials do
  if f(a)<f(a') then
    a' := a
  a := S(L(N(a), a), a)

return a'
```

Local search blueprint

```
a  := GenerateInitialAssignment()
a' := a
for k in 1 … rials do
  if f(a)<f(a') then
    a' := a
  a := S(L(N(a), a), a)

return a'
```

quality

Local search blueprint

```
a  := GenerateInitialAssignment()
a' := a
for k in 1...MaxTrials do
  if f(a)<f(a') then
    a' := a
  a := S(L(N(a), a), a)

return a'
```

Local search blueprint

```
a  := GenerateInitialAssignment()
a' := a
for k in 1...MaxTrials do
  if f(a)<f(a') then
    a' := a
  a := S(L(N(a), a), a)
                              move

return a'
```

Local search blueprint

```
a  := GenerateInitialAssignment()
a' := a
for k in 1...MaxTrials do
  if f(a)<f(a') then
    a' := a
  a := S(L(N(a), a), a)

return a'
```

Reminder: GSAT

```
GSAT(α,MAX_FLIPS,MAX_TRIES)
    for i := 1 to MAX_TRIES
        T := random truth assignment
        for j := 1 to MAX_FLIPS
            if T satisfies α return T
            p :=  variable such that changing its truth value
                  gives largest increase in number of clauses of
                  α that are satisfied by T
            T := T with truth assignment of p reversed
        end for
    end for
    return "no satisfying assignment found"
```

Example: Graph partitioning

- find a balanced partition of vertices of a graph $G = (V, E)$

$$V = P_1 \uplus P_2, \quad |P_1| = |P_2|$$

- with minimal numer of cross-partition edges

$$f(\langle P_1, P_2 \rangle) = |\{\langle v_1, v_2 \rangle \in E \mid v_1 \in P_1, v_2 \in P_2\}|$$

Approach

- **Greedy local improvement**
  - N returns only balanced partitions
  - L licenses only better assignments

Example: Graph partitioning

- **Initial assignment:** arbitrary balanced partition
- **Move:** swap vertices between P_1 and P_2
  - assignments remains balanced partitions!
- **Neighbourhood:**

$$N(\langle P_1, P_2\rangle) = \{\langle (P_1 \setminus \{a\}) \cup \{b\}, (P_2 \setminus \{b\}) \cup \{a\}\rangle \mid a \in P_1, b \in P_2\}$$

Example: Graph partitioning

- **Legal moves:** improve objective value

$$L(N, a) = \{n \in N \mid f(n) < f(a)\}$$

- **Greedy selection:** choose one among best neighbours

$$S(M, a) = \min\{m \in M \mid f(n) = \min_{a \in M} f(a)\}$$

Heuristics and Metaheuristics

- **Heuristics**
    - typically choose next neighbour
- **Metaheuristics**
    - collect information on execution sequence
    - how to escape local minima
    - how to drive to global optimality

Example heuristics

- Locally improve f
- Choose best neighbour
  - randomize in case of ties
- Choose first better neighbour
- Multistage heuristics
- Random improvement
- ...

Example metaheuristics

- **Iterated local search**
  - start from different initial assignments
- **Simulated annealing**
  - based on "temperature"
  - initially: sample search space widely
  - later: towards random improvement
- **Tabu search**
- ...

Implementation aspects

```
a  := GenerateInitialAssignment()
a' := a
for k in 1...MaxTrials do
  if f(a)<f(a') then
    a' := a
  a := S(L(N(a), a), a)

return a'
```

Implementation aspects

```
a  := GenerateInitialAssignment()
a' := a
for k in 1...MaxTrials do
  if f(a)<f(a') then
    a' := a
  a := S(L(N(a), a), a)

return a'
```

incremental algorithms!

Constraint-based local search

- **Idea:**
  - **declarative model** based on constraints
  - give **structure** to neighbourhood computation
  - **efficient incremental algorithms**
  - separate model from **search specification**

COMET

- System for constraint-based local search
- Special, object-oriented programming language
- http://www.comet-online.org
- Developed at Brown University by P. Van Hentenryck and Laurent Michel

n-Queens in Comet

```
int n = 8;
range Size = 1..n;
LocalSolver m();
UniformDistribution distr(Size);

var{int} queens[i in Size](m,Size) := distr.get();
int neg[i in Size] = -i;
int pos[i in Size] = i;

ConstraintSystem S(m);
S.post(new AllDifferent(queen));
S.post(new AllDifferent(queen, neg));
S.post(new AllDifferent(queen, pos));
m.close();

while (S.violationDegree() > 0)
   selectMax(q in Size)(S.getViolations(queen[q]))
      selectMin(v in Size)(S.getAssignDelta(queen[q],v))
         queen[q] := v;
```

n-Queens in Comet

incremental variable

```
int n = 8;
range Size = 1
LocalSolver m(
UniformDistribu                     e);

var{int} queens[i in Size](m,Size) := distr.get();
int neg[i in Size] = -i;
int pos[i in Size] = i;

ConstraintSystem S(m);
S.post(new AllDifferent(queen));
S.post(new AllDifferent(queen, neg));
S.post(new AllDifferent(queen, pos));
m.close();

while (S.violationDegree() > 0)
   selectMax(q in Size)(S.getViolations(queen[q]))
      selectMin(v in Size)(S.getAssignDelta(queen[q],v))
         queen[q] := v;
```

n-Queens in Comet

```
int n = 8;
range Size = 1..n;
LocalSolver m();
UniformDistribution distr(Size);

var{int} queens[i in Size](m,Size) := distr.get();
int neg[i in Size] = -i;
int pos[i in Size] = i;

ConstraintSystem S(m);
S.post(new AllDifferent(queen));
S.post(new AllDifferent(queen, neg));
S.post(new AllDifferent(queen, pos));
m.close();

while (S.violationDegree() > 0)
   selectMax(q in Size)(S.getViolations(queen[q]))
      selectMin(v in Size)(S.getAssignDelta(queen[q],v))
         queen[q] := v;
```

n-Queens in Comet

random initial assignment

```
int n = 8;
range Size = 1..n;
LocalSolver m();
UniformDistribution distr(Size);

var{int} queens[i in Size](m,Size) := distr.get();
int neg[i in Size] = -i;
int pos[i in Size] = i;

ConstraintSystem S(m);
S.post(new AllDifferent(queen));
S.post(new AllDifferent(queen, neg));
S.post(new AllDifferent(queen, pos));
m.close();

while (S.violationDegree() > 0)
   selectMax(q in Size)(S.getViolations(queen[q]))
      selectMin(v in Size)(S.getAssignDelta(queen[q],v))
         queen[q] := v;
```

n-Queens in Comet

```
int n = 8;
range Size = 1..n;
LocalSolver m();
UniformDistribution distr(Size);

var{int} queens[i in Size](m,Size) := distr.get();
int neg[i in Size] = -i;
int pos[i in Size] = i;

ConstraintSystem S(m);
S.post(new AllDifferent(queen));
S.post(new AllDifferent(queen, neg));
S.post(new AllDifferent(queen, pos));
m.close();

while (S.violationDegree() > 0)
   selectMax(q in Size)(S.getViolations(queen[q]))
      selectMin(v in Size)(S.getAssignDelta(queen[q],v))
         queen[q] := v;
```

n-Queens in Comet

```
int n = 8;
range Size = 1..n;
LocalSolver m();
UniformDistribution distr(Size);

var{int} queens[i in Size](m,Size) := distr.get();
int neg[i in Size] = -i;
int pos[i in Size] = i;

ConstraintSystem S(m);
S.post(new AllDifferent(queen));
S.post(new AllDifferent(queen, neg));
S.post(new AllDifferent(queen, pos));
m.close();

while (S.violationDegree() > 0)
   selectMax(q in Size)(S.getViolations(queen[q]))
      selectMin(v in Size)(S.getAssignDelta(queen[q],v))
         queen[q] := v;
```

constraints

n-Queens in Comet

```
int n = 8;
range Size = 1..n;
LocalSolver m();
UniformDistribution distr(Size);

var{int} queens[i in Size](m,Size) := distr.get();
int neg[i in Size] = -i;
int pos[i in Size] = i;

ConstraintSystem S(m);
S.post(new AllDifferent(queen));
S.post(new AllDifferent(queen, neg));
S.post(new AllDifferent(queen, pos));
m.close();

while (S.violationDegree() > 0)
   selectMax(q in Size)(S.getViolations(queen[q]))
      selectMin(v in Size)(S.getAssignDelta(queen[q],v))
         queen[q] := v;
```

n-Queens in Comet

```
int n = 8;
range Size = 1..n;
LocalSolver m();
UniformDistribution distr(Size);

var{int} queens[i in Size](m,Size) := distr.get();
int neg[i in Size] = -i;
int pos[i in Size] = i;

ConstraintSystem S(m);
S.post(new AllDifferent(queen));
S.post(new AllDifferent(queen, neg));
S.post(new AllDifferent(queen, pos));
m.close();

while (S.violationDegree() > 0)
   selectMax(q in Size)(S.getViolations(queen[q]))
      selectMin(v in Size)(S.getAssignDelta(queen[q],v))
         queen[q] := v;
```

search

n-Queens in Comet

```
int n = 8;
range Size = 1..n;
LocalSolver m();
UniformDistribution distr(Size);

var{int} queens[i in Size](m,Size) := distr.get();
int neg[i in Size] = -i;
int pos[i in Size] = i;

ConstraintSystem S(m);
S.post(new AllDifferent(queen));
S.post(new AllDifferent(queen, neg));
S.post(new AllDifferent(queen, pos));
m.close();

while (S.violationDegree() > 0)
   selectMax(q in Size)(S.getViolations(queen[q]))
      selectMin(v in Size)(S.getAssignDelta(queen[q],v))
         queen[q] := v;
```

Meta-heuristic: tabu search

- **Idea:**

  Do not modify variables that you have modified recently

- Simple **tabu data structure:**
  - tabu[*v*] = earliest iteration when *v* may be modified
  - when modifying *v* in iteration *i*, set tabu[*v*] to *i*+*t*, where *t* is the **tabu tenure**

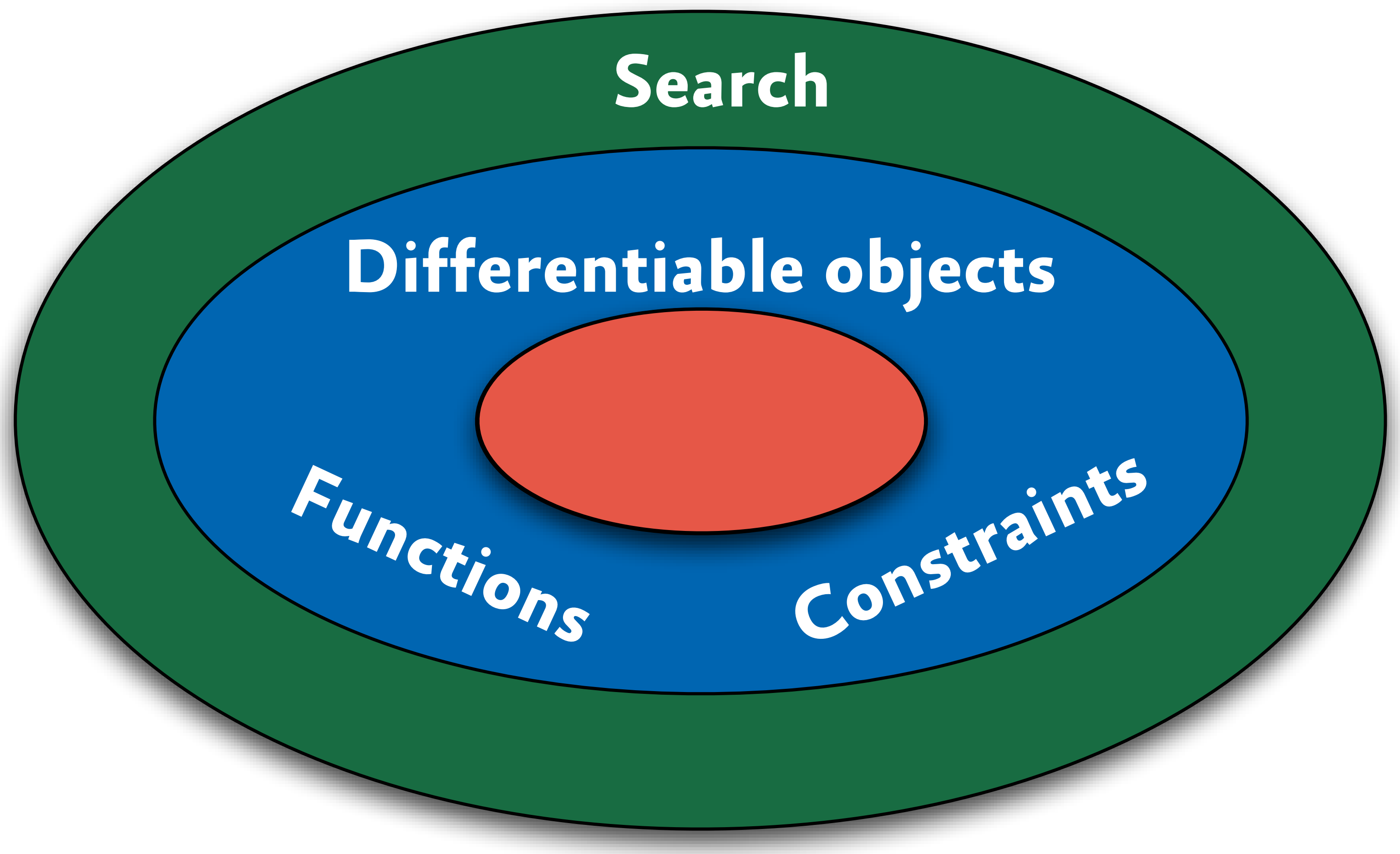
n-Queens (tabu search)

```
int n = 20;

...

int tabu[i in Size] = -1;
int iteration = 0;
int tenure = 4;

while (S.violationDegree()) {
  selectMax(q in Size : tabu[q] <= iteration)
     (S.getViolations(queen[q]))
     selectMin(v in Size)(S.getAssignDelta(queen[q],v)) {
        queen[q] := v;
        tabu[q] = iteration + tenure;
     }
   iteration++;
}
```

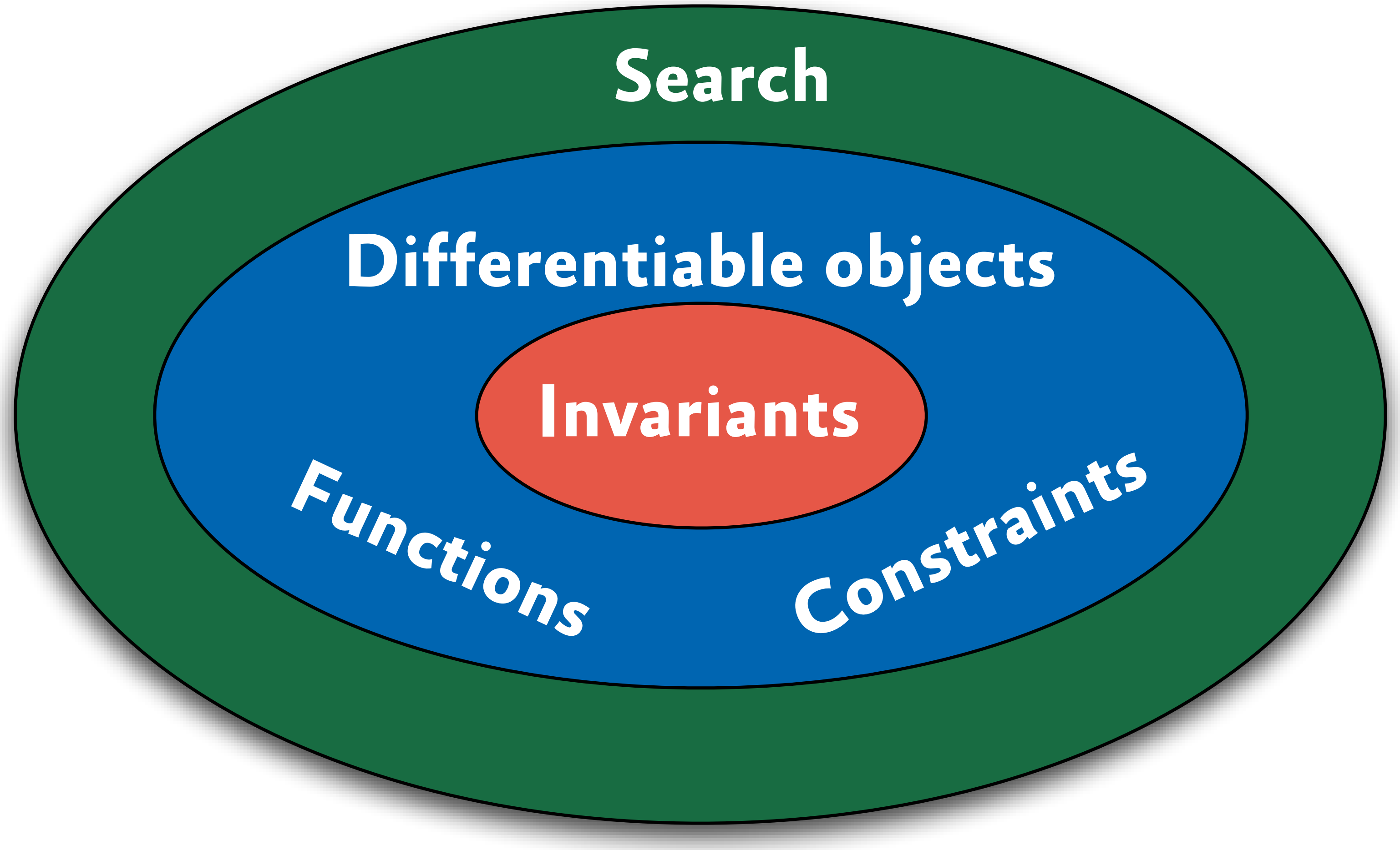
Constraints as differentiable objects

- **Maintain properties incrementally**
  - degree of violation
  - set of violated variables
- Allow **differentiation**, i.e. querying effect of modifications
  - how would violation **change** when moving to this assignment
- Possibly **difficult**, definitely **tedious** to implement!

CBLS - Architecture



CBLS - Architecture



Invariants

- **Example:**

```
inc{int} s(m) <- sum(i in 1..10) a[i]
```

- **Incrementally maintain** that s is the sum of the a[i]
- "one-way constraints"
- **Differentiable**

Constraints based on invariants

- **Implement constraints as objective functions:**
  - $V(e_1{=}e_2) \quad = \quad abs(e_1\text{-}e_2)$
  - $V(e_1{\leq}e_2) \quad = \quad max(e_1\text{-}e_2,\ 0)$
  - $V(e_1{\neq}e_1) \quad = \quad 1\ \text{-}\ min(1,\ abs(e_1\text{-}e_2))$
  - $V(r_1{\wedge}r_2) \quad = \quad V(r_1)\ +\ V(r_2)$
  - $V(r_1{\vee}r_2) \quad = \quad min(V(r_1),\ V(r_2))$
  - $V(\neg r) \quad = \quad 1\ \text{-}\ min(1,\ V(r))$

Local search vs. CP

- LS needs only **linear memory** in problem size
- LS deals well with **large** and **overconstrained** problems
- LS is good for **online algorithms** (dynamically changing constraints)

- CP can **guarantee that constraints hold**
- CP can **prove unsatisfiability**
- CP can **prove optimality**

Literature

- Van Hentenryck, Michel. *Constraint-Based Local Search*. MIT Press, 2005.
- Michel, Van Hentenryck. *A Constrained-Based Architecture for Local Search*. OOPSLA, 2002.
- Van Hentenryck, Michel. *Differentiable Invariants*. CP, 2006.

Linear programming

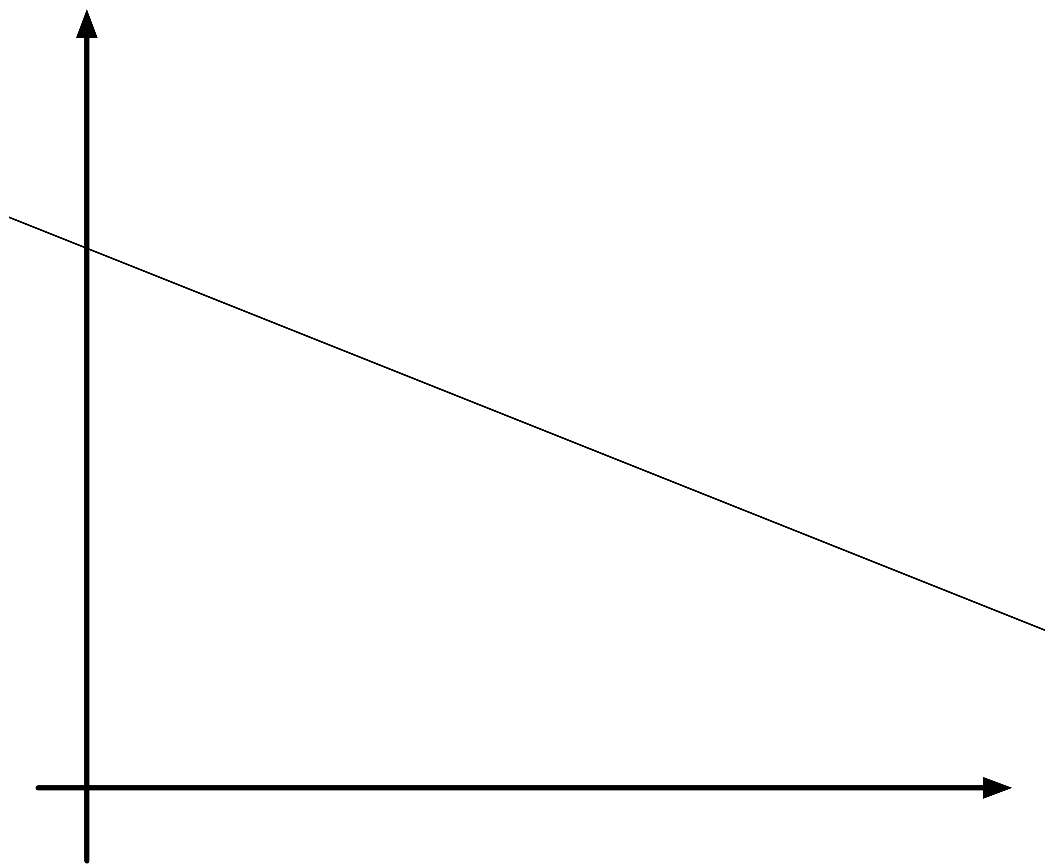
Linear Programming (LP)

- **Describe problem by**
  - linear equations
  - linear inequalities
  - linear objective function
- Find **optimal solution** over the reals
- **Variants:** some variables must have integer values
  - **ILP** Integer Linear Programming
  - **MILP** Mixed Integer Linear Programming

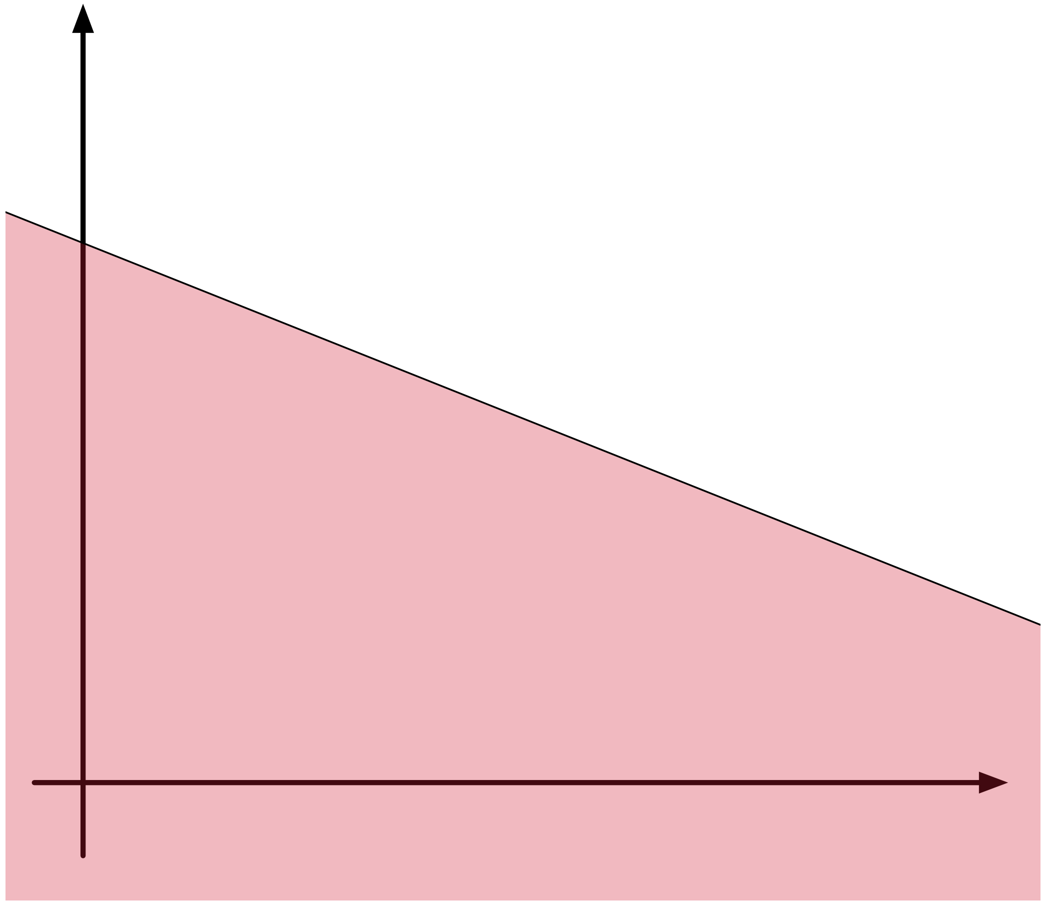
LP and CP Hybrids

- **Relaxation of global constraints**
    - tighten bounds on variables
    - can guide search
    - pruning in optimization
- **Decomposition schemes**
    - use CP/LP on well-suited problem parts
    - column generation
    - Benders decomposition

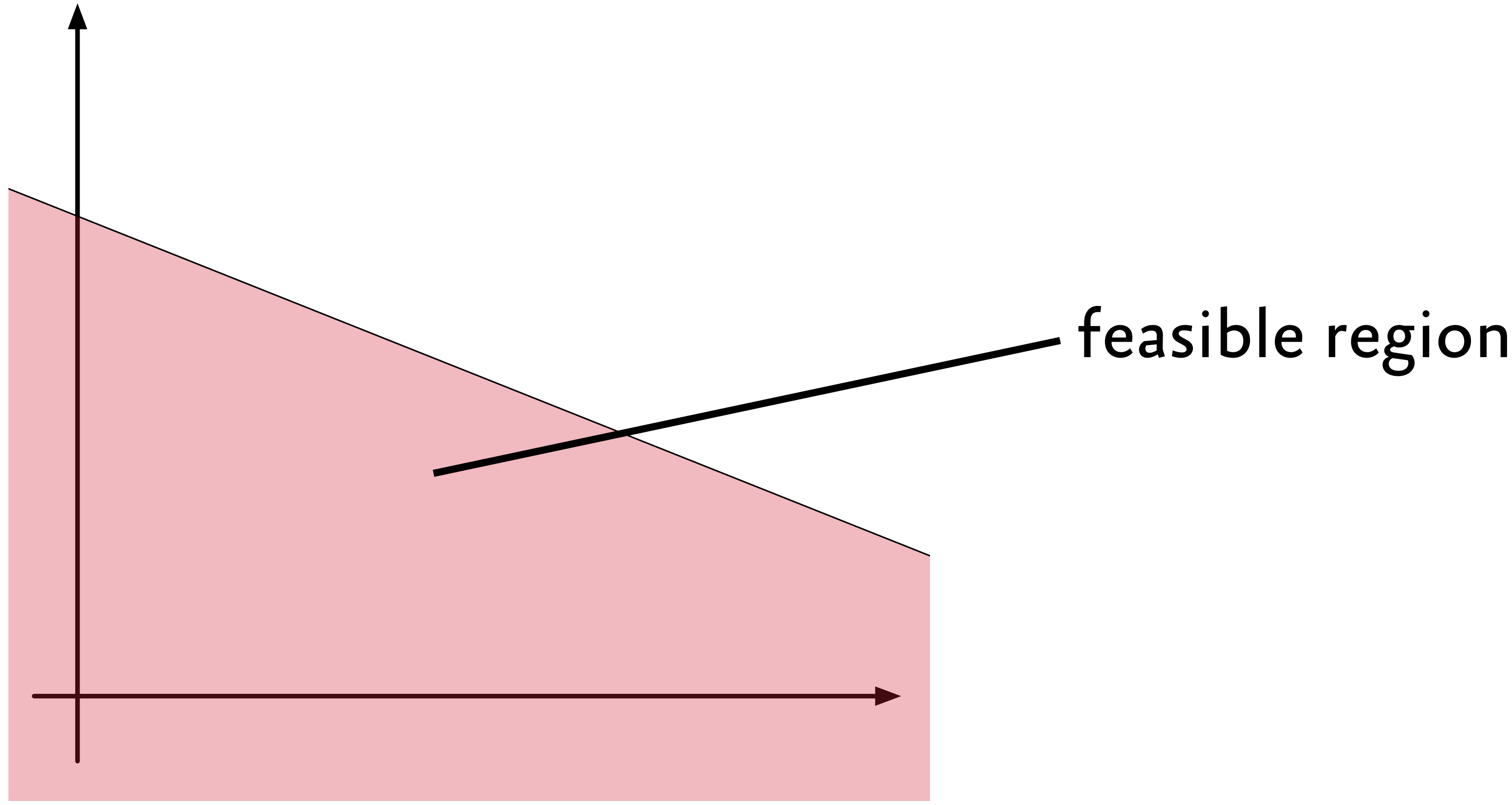
Simplex method



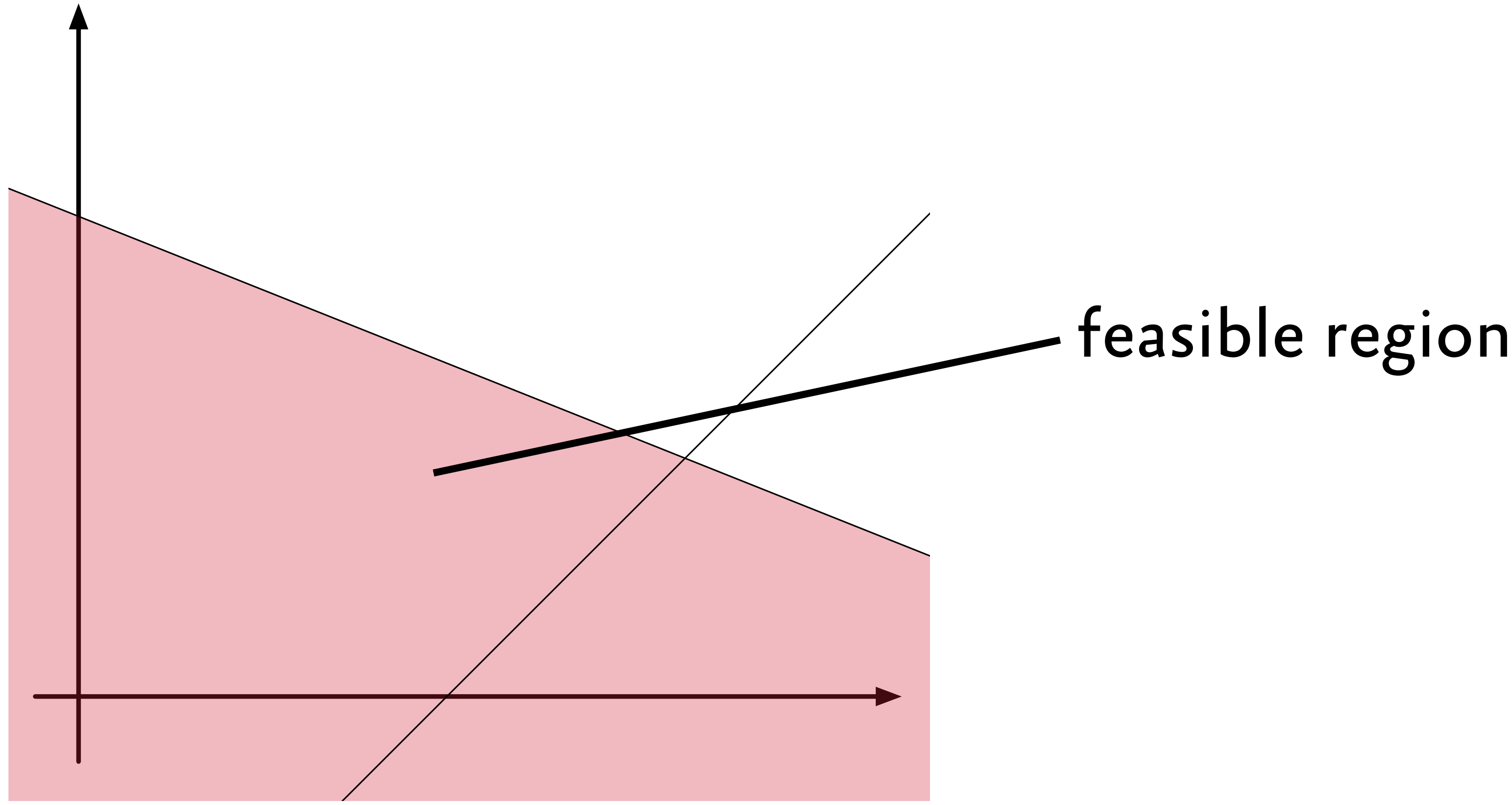
Simplex method



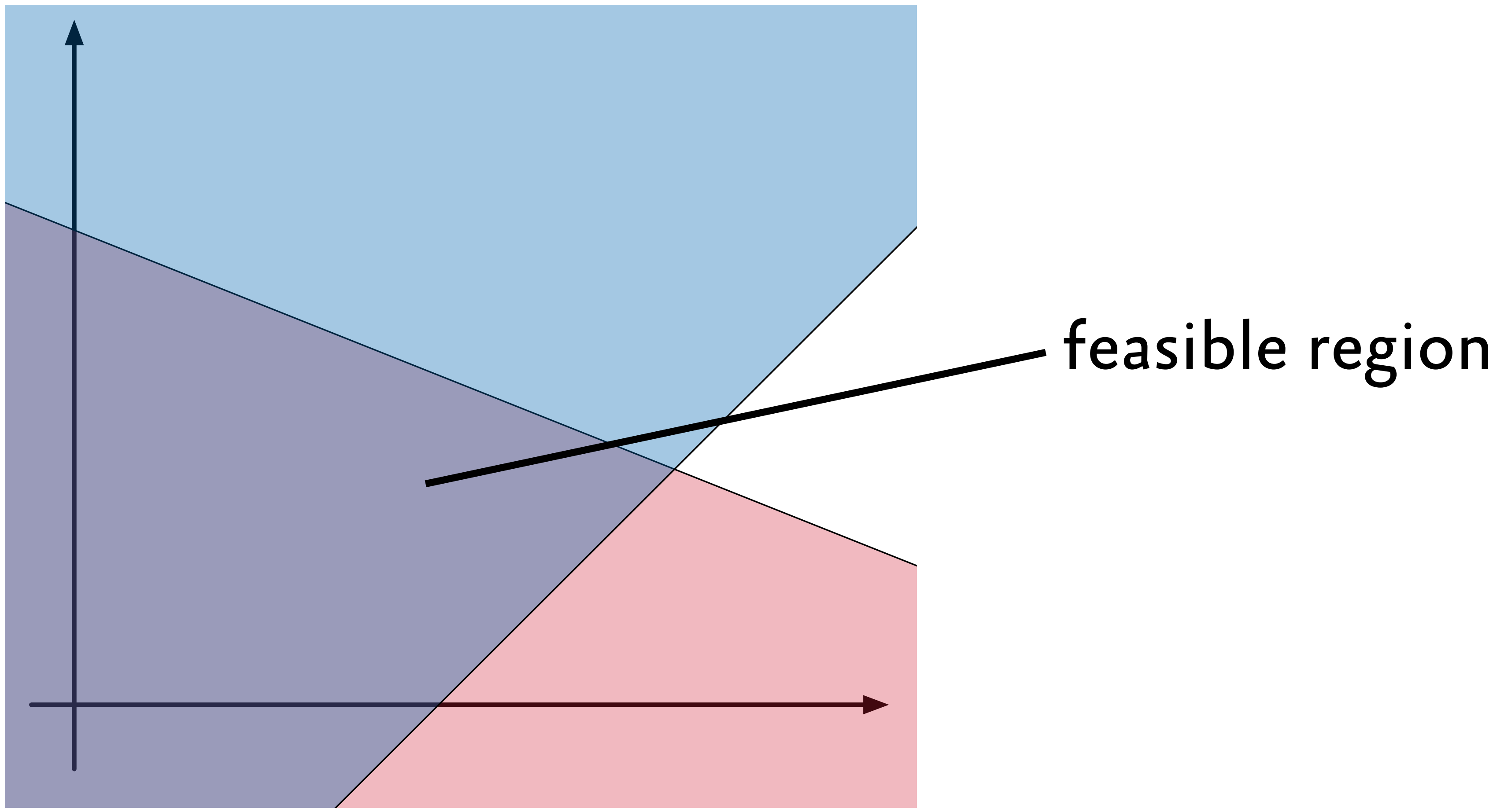
Simplex method



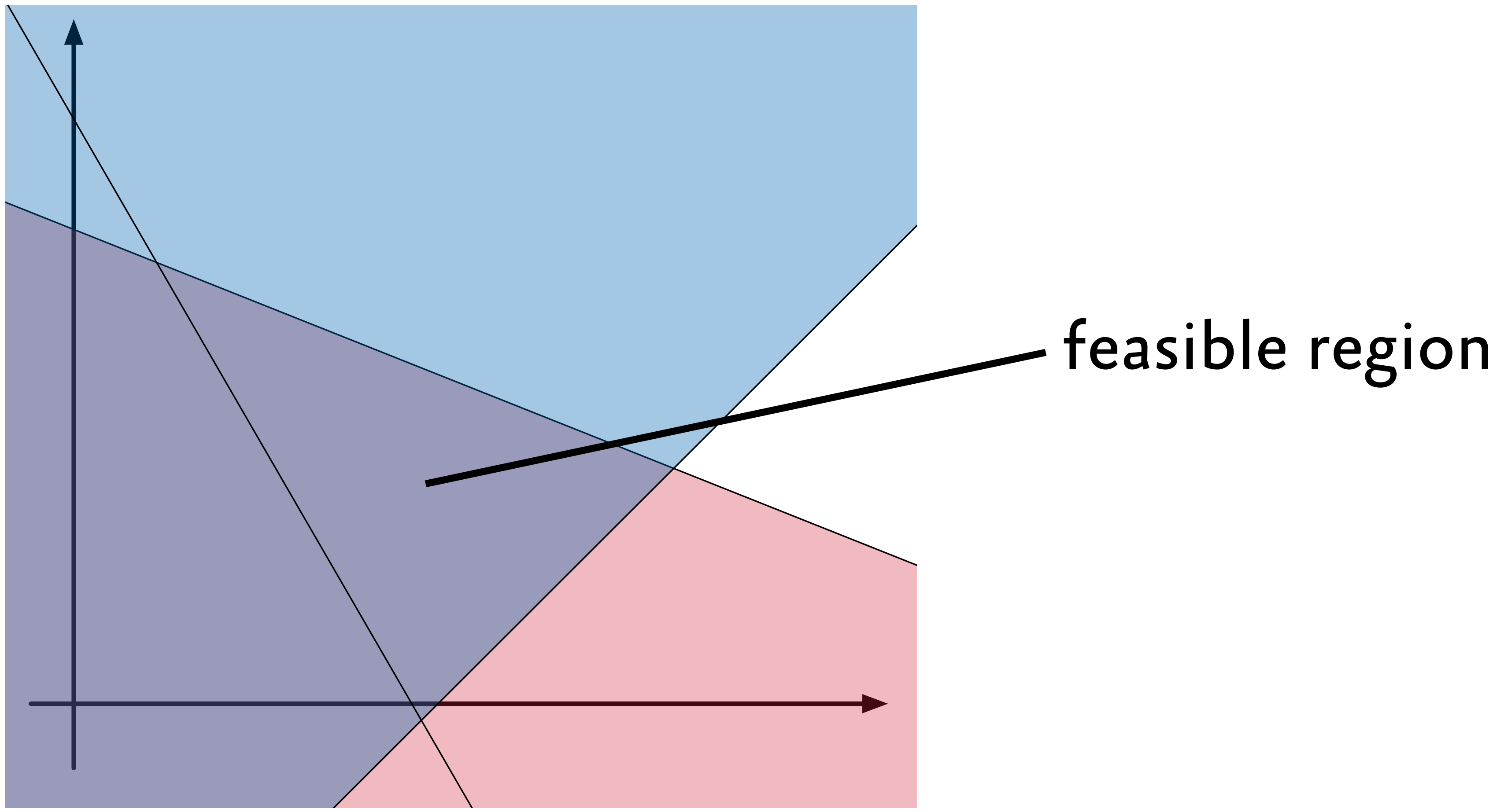
Simplex method



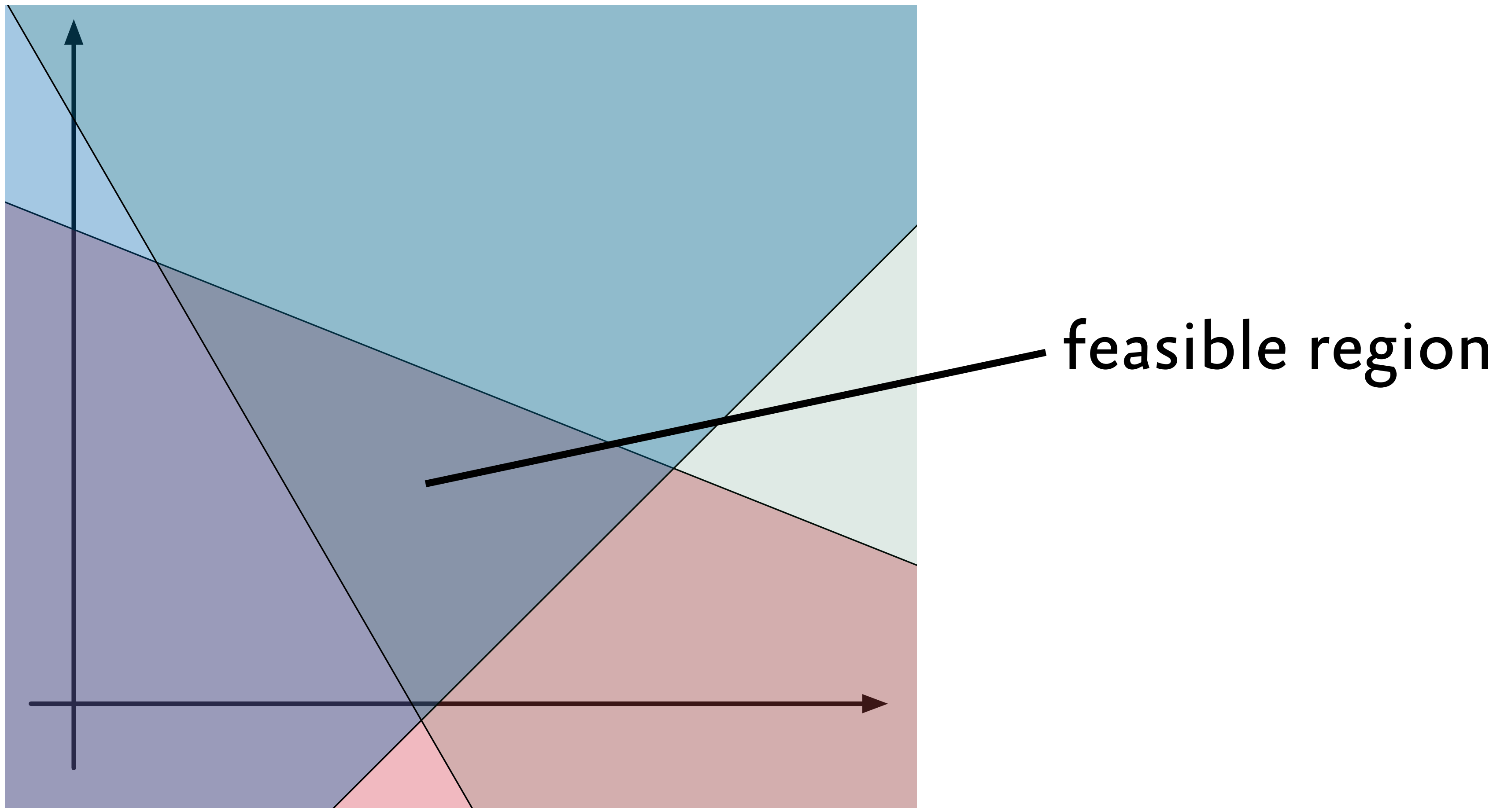
Simplex method



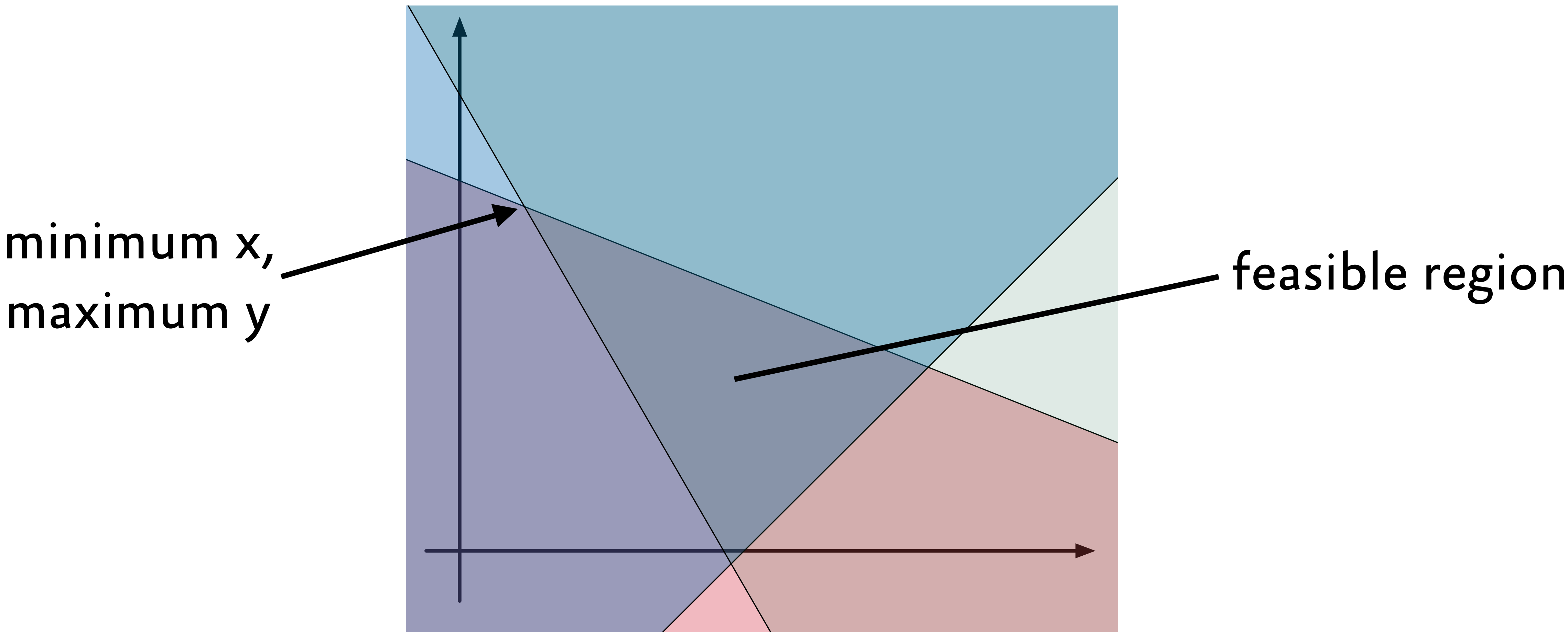
Simplex method



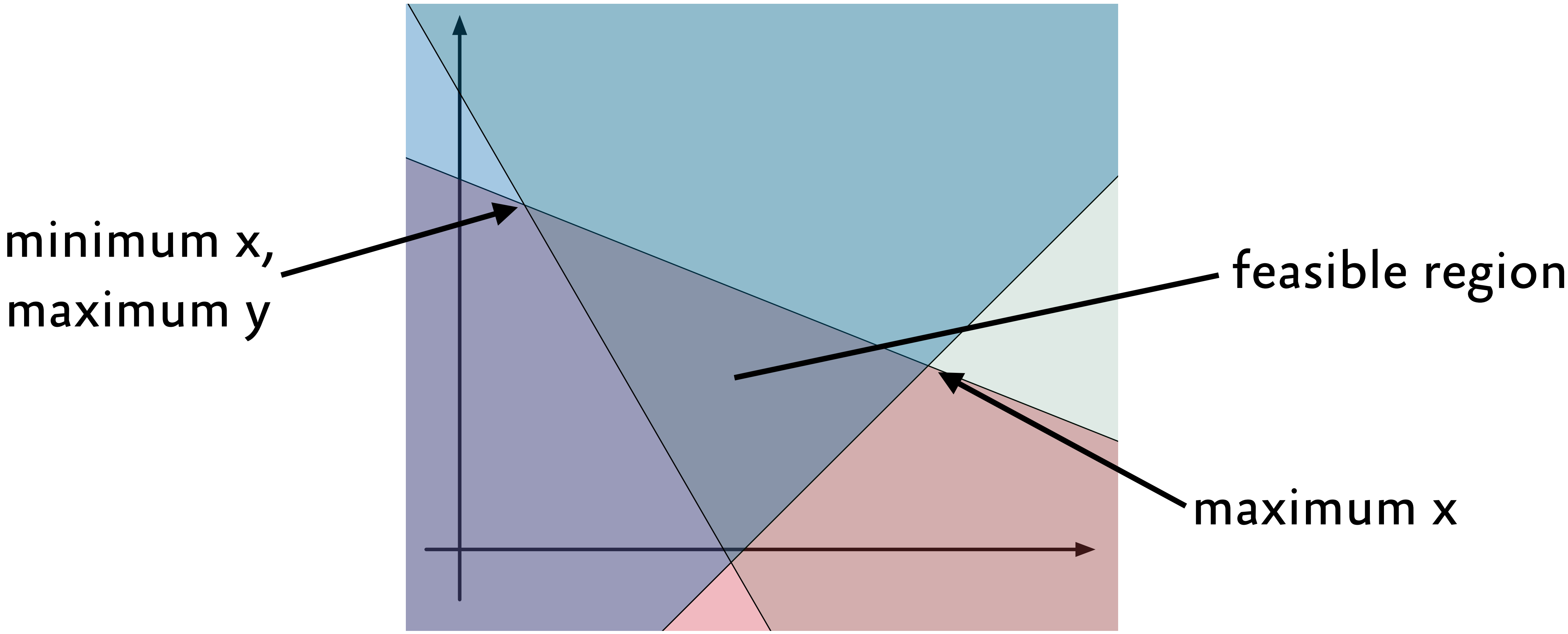
Simplex method



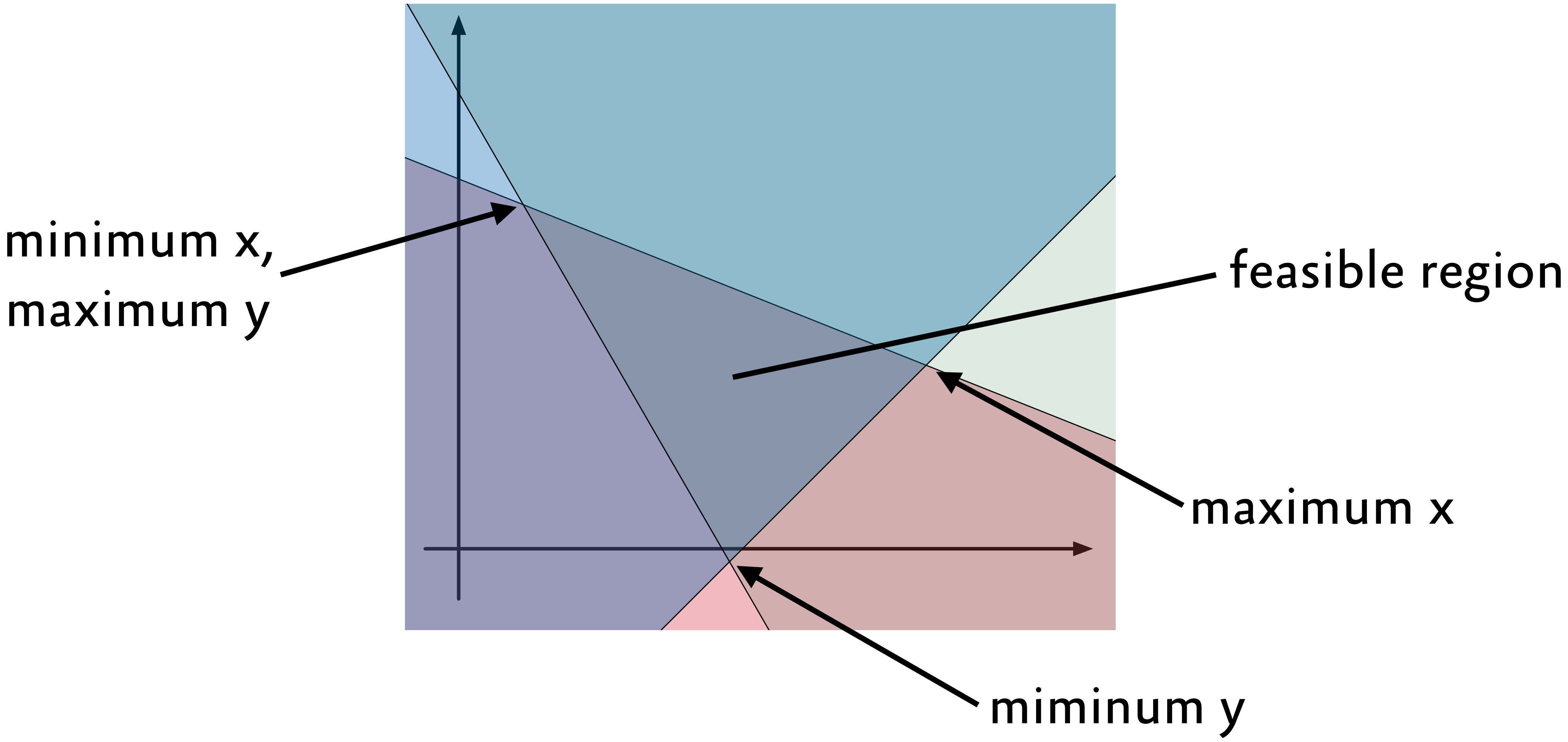
Simplex method



Simplex method



Simplex method



Linear Programming

- **Advantages**
  - very robust and scalable technique
  - very well understood
  - numerically highly stable
  - very good at optimization
- **Disadvantages**
  - common structures difficult to express (distinct, ...)

Literature

- Cormen, Leiserson, Rivest, Stein. *Introduction to Algorithms, Second Edition*. MIT Press and McGraw-Hill, 2001.

Summary

- **Propagation-based approaches**
  - exploit problem structure (global constraints)
  - good at finding feasible solutions
  - good at combining algorithmic techiques
- **Other approaches**
  - often scale better
  - sometimes better at optimization
  - may offer fewer guarantees