Topic 17: Constraint-Based Local Search[1]
(Version of 26th November 2020)

Pierre Flener

Optimisation Group
Department of Information Technology
Uppsala University
Sweden

Course 1DL441:
Combinatorial Optimisation and Constraint Programming,
whose part 1 is Course 1DL451:
Modelling for Combinatorial Optimisation

[1] Based on an early version by Magnus Ågren (2008)

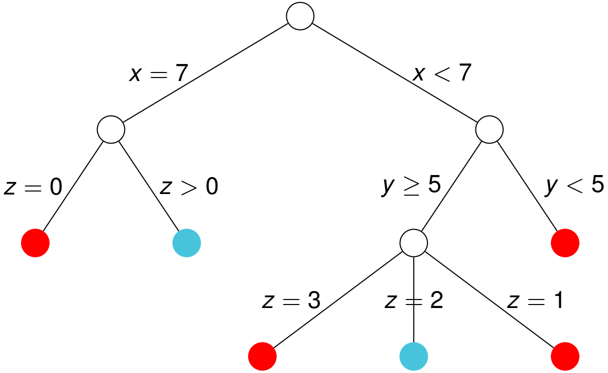
Outline

Outline

Outline

So Far: Inference + Systematic Search

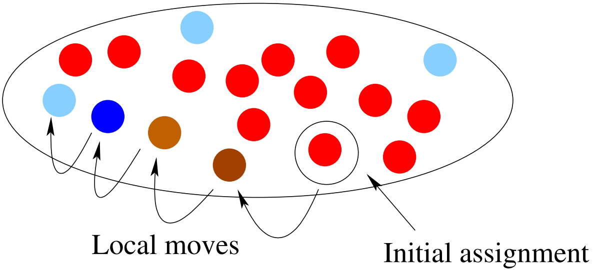
- The variables become fixed 1-by-1.
- Stop when solution or unsatisfiability proof is obtained.
- Search space from a systematic-search viewpoint:



UPPSALA UNIVERSITET

Now: Inference + Local Search

- All variables are always fixed, from initial assignment.
- Search proceeds by local moves: each move modifies the values of a few variables in the current assignment, and is selected upon probing the cost impacts of several candidate moves, called the neighbourhood.
- Stop when a good enough assignment has been found, or when an allocated resource has been exhausted, such as time spent or iterations made.



Example (BIBD: AED assignment after *i* moves)

	plot1	plot2	plot3	plot4	plot5	plot6	plot7
barley	✓	✓	✓	–	–	–	–
corn	✓	–	–	✓	–	✓	–
millet	✓	–	–	–	–	✓	✓
oats	–	✓	–	✓	✓	–	–
rye	–	✓	–	–	✓	–	✓
spelt	–	–	✓	✓	–	–	✓
wheat	–	–	✓	–	✓	✓	–

1. Equal growth load: Every plot grows 3 grains.
   Currently satisfied: zero violation.
2. Equal sample size: Every grain is grown in 3 plots.
   Satisfied by initial assignment and each move: implicit.
3. Balance: Every grain pair is grown in 1 common plot.
   But, e.g., oats & rye are grown in $2 \neq 1$ common plots.

Example (BIBD: AED assignment after *i* moves)

	plot1	plot2	plot3	plot4	plot5	plot6	plot7
barley	✓	✓	✓	–	–	–	–
corn	✓	–	–	✓	–	✓	–
millet	✓	–	–	–	–	✓	✓
oats	–	✓	–	✓	✓	–	–
rye	–	✓	–	–	✓	–	✓
spelt	–	–	✓	✓	–	–	✓
wheat	–	–	✓	–	✓	✓	–

1. Equal growth load: Every plot grows 3 grains.
   Currently satisfied: zero violation.
2. Equal sample size: Every grain is grown in 3 plots.
   Satisfied by initial assignment and each move: implicit.
3. Balance: Every grain pair is grown in 1 common plot.
   But, e.g., oats & rye are grown in $2 \neq 1$ common plots.

Selected move: let plot6 instead of plot5 grow oats.

Example (BIBD: AED assignment after *i* moves)

	plot1	plot2	plot3	plot4	plot5	plot6	plot7
barley	✓	✓	✓	–	–	–	–
corn	✓	–	–	✓	–	✓	–
millet	✓	–	–	–	–	✓	✓
oats	–	✓	–	✓	–	✓	–
rye	–	✓	–	–	✓	–	✓
spelt	–	–	✓	✓	–	–	✓
wheat	–	–	✓	–	✓	✓	–

1. Equal growth load: Every plot grows 3 grains.
   Currently satisfied: zero violation.
2. Equal sample size: Every grain is grown in 3 plots.
   Satisfied by initial assignment and each move: implicit.
3. Balance: Every grain pair is grown in 1 common plot.
   But, e.g., oats & rye are grown in $2 \neq 1$ common plots.

Selected move: let plot6 instead of plot5 grow oats.

Example (BIBD: AED assignment after $i + 1$ moves)

	plot1	plot2	plot3	plot4	plot5	plot6	plot7
barley	✓	✓	✓	–	–	–	–
corn	✓	–	–	✓	–	✓	–
millet	✓	–	–	–	–	✓	✓
oats	–	✓	–	✓	–	✓	–
rye	–	✓	–	–	✓	–	✓
spelt	–	–	✓	✓	–	–	✓
wheat	–	–	✓	–	✓	✓	–

1. Equal growth load: Every plot grows 3 grains.
   But plot5 grows $2 \neq 3$ grains; plot6 grows $4 \neq 3$ grains.
2. Equal sample size: Every grain is grown in 3 plots.
   Satisfied by initial assignment and each move: implicit.
3. Balance: Every grain pair is grown in 1 common plot.
   But, e.g., corn & oats are grown in $2 \neq 1$ common plots.

Example (BIBD: AED assignment after $i+1$ moves)

	plot1	plot2	plot3	plot4	plot5	plot6	plot7
barley	✓	✓	✓	–	–	–	–
corn	✓	–	–	✓	–	✓	–
millet	✓	–	–	–	–	✓	✓
oats	–	✓	–	✓	–	✓	–
rye	–	✓	–	–	✓	–	✓
spelt	–	–	✓	✓	–	–	✓
wheat	–	–	✓	–	✓	✓	–

1. Equal growth load: Every plot grows 3 grains.
   But plot5 grows $2 \neq 3$ grains; plot6 grows $4 \neq 3$ grains.
2. Equal sample size: Every grain is grown in 3 plots.
   Satisfied by initial assignment and each move: implicit.
3. Balance: Every grain pair is grown in 1 common plot.
   But, e.g., corn & oats are grown in $2 \neq 1$ common plots.

Selected move: let plot5 instead of plot6 grow corn.

Example (BIBD: AED assignment after $i + 1$ moves)

	plot1	plot2	plot3	plot4	plot5	plot6	plot7
barley	✓	✓	✓	–	–	–	–
corn	✓	–	–	✓	✓	–	–
millet	✓	–	–	–	–	✓	✓
oats	–	✓	–	✓	–	✓	–
rye	–	✓	–	–	✓	–	✓
spelt	–	–	✓	✓	–	–	✓
wheat	–	–	✓	–	✓	✓	–

1. Equal growth load: Every plot grows 3 grains.
   But plot5 grows $2 \neq 3$ grains; plot6 grows $4 \neq 3$ grains.
2. Equal sample size: Every grain is grown in 3 plots.
   Satisfied by initial assignment and each move: implicit.
3. Balance: Every grain pair is grown in 1 common plot.
   But, e.g., corn & oats are grown in $2 \neq 1$ common plots.

Selected move: let plot5 instead of plot6 grow corn.

Example (BIBD: AED assignment after $i+2$ moves)

	plot1	plot2	plot3	plot4	plot5	plot6	plot7
barley	✓	✓	✓	–	–	–	–
corn	✓	–	–	✓	✓	–	–
millet	✓	–	–	–	–	✓	✓
oats	–	✓	–	✓	–	✓	–
rye	–	✓	–	–	✓	–	✓
spelt	–	–	✓	✓	–	–	✓
wheat	–	–	✓	–	✓	✓	–

1. Equal growth load: Every plot grows 3 grains.
   Currently satisfied: zero violation.
2. Equal sample size: Every grain is grown in 3 plots.
   Satisfied by initial assignment and each move: implicit.
3. Balance: Every grain pair is grown in 1 common plot.
   Currently satisfied: zero violation.

Stop search: All constraints are satisfied (no optimisation).

Terminology and Choices

Definitions

Consider a problem $\langle V, U, C\,[, f\,]\rangle$ where $V = [v_1, \ldots, v_m]$ and f is to be minimised, without loss of generality.
An assignment $s\colon\ V \rightarrow U$ maps the variables to values, and is satisfying (or: feasible) if they satisfy all constraints in C.

Note how a store $s\colon\ V \rightarrow 2^U$ in Topics 13 to 16 differs.

Property: A satisfying assignment actually is a solution to a constraint satisfaction problem (CSP), but it might be sub-optimal for a constrained optimisation problem (COP).

Assume function COST gives the cost of an assignment s:

- CSP: $\text{COST}(s) = \sum_{c \in C} \text{VIOLATION}(c, s)$
- COP: $\text{COST}(s) = \alpha \cdot \sum_{c \in C} \text{VIOLATION}(c, s) + \beta \cdot f(s(v_1), \ldots, s(v_m))$

for problem-specific VIOLATION and parameters α and β.

Definition

A soft constraint c has a function $\text{VIOLATION}(c, s)$ that returns zero if c is satisfied under the assignment s, else a positive value proportional to its dissatisfaction.

Example: $\text{VIOLATION}(x \leq y, s) = \textbf{if } s(x) \leq s(y) \textbf{ then } 0 \textbf{ else } s(x) - s(y)$

Definition

A one-way constraint is kept satisfied during search, as one of its variables is defined by a total function on the others.

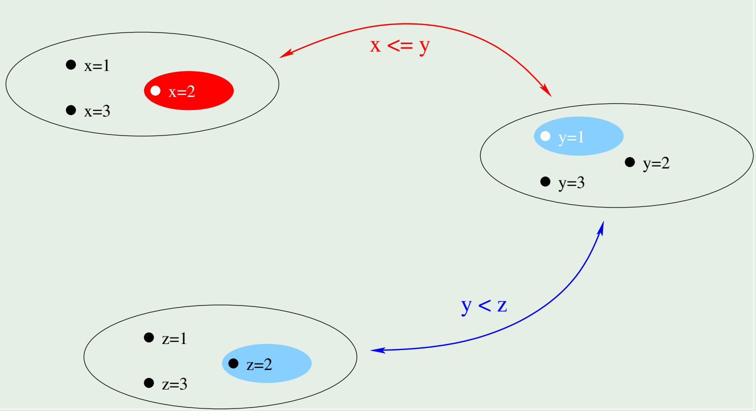
Example: For $p = x \cdot y$, if x or y or both are reassigned by a move to assignment s, then $s(p)$ is to be set to $s(x) \cdot s(y)$.

Definition

A violating variable in a constraint c unsatisfied, or violated, under assignment s can be reassigned, not necessarily within its domain, so that $\text{VIOLATION}(c, s)$ decreases.

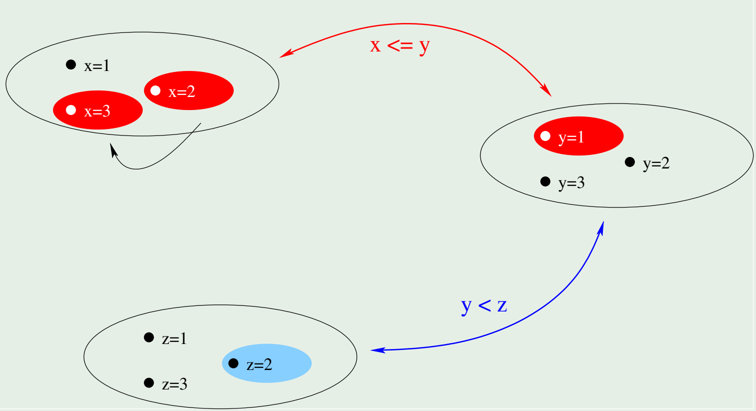
Example ($x, y, z \in \{1, 2, 3\} \wedge x \leq y \wedge y < z$)

Non-satisfying assignment (the constraint $x \leq y$ is violated; the decision variables x and y are violating w.r.t. $x \leq y$):



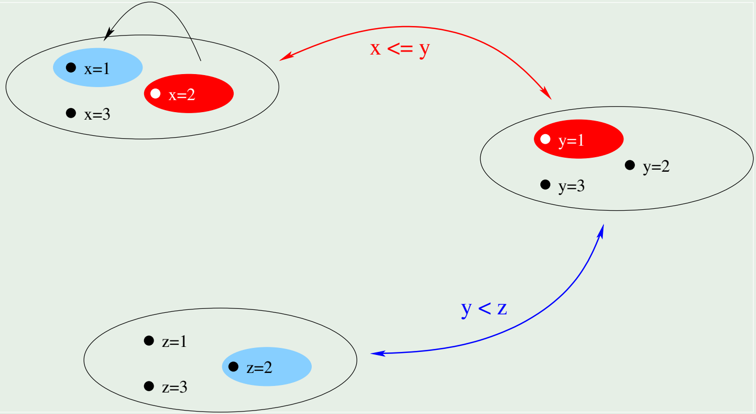
Example ($x, y, z \in \{1,2,3\} \land x \leq y \land y < z$)

Probed move $x := 3$, reaching another non-satisfying assignment (the constraint $x \leq y$ is still violated; the decision variables x and y are still violating w.r.t. $x \leq y$):



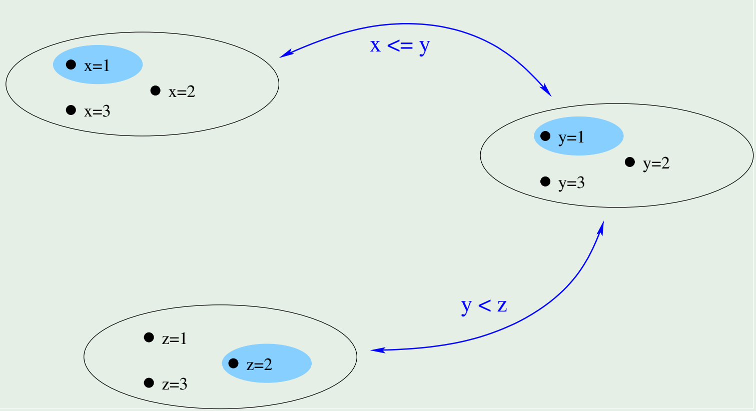
Example ($x, y, z \in \{1, 2, 3\} \wedge x \leq y \wedge y < z$)

Another probed move $x := 1$, reaching a satisfying assignment (there are no more violated constraints or violating variables):



Example ($x, y, z \in \{1, 2, 3\} \wedge x \leq y \wedge y < z$)

Another probed move $x := 1$, reaching a satisfying assignment (there are no more violated constraints or violating variables):



Systematic Search (as in SAT, SMT, MIP, CP):

+ Will find an (optimal) solution, if one exists.
+ Will give a proof of unsatisfiability, otherwise.
– May take a long time to complete.
– Sometimes does not scale well to large instances.
– May need a lot of tweaking: search strategies, . . .

Local Search: (Hoos and Stützle, 2004)

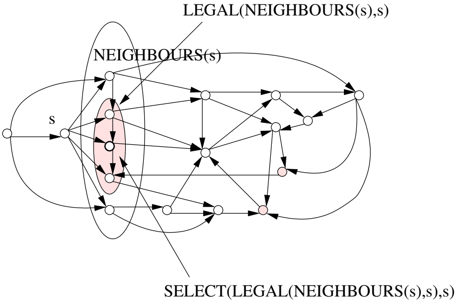
+ May find an (optimal) solution, if one exists.
– Can rarely give a proof of unsatisfiability, otherwise.
– Can rarely guarantee that a found solution is optimal.
+ Often scales much better to large instances.
– May need a lot of tweaking: heuristics, parameters, . . .

Local search trades completeness and quality for speed!

Outline

Local-Search Heuristics: Outline

- Start from the result of INITIALASSIGNMENT(V, U).
- Iteratively move to a neighbour assignment.
- Aim for a satisfying assignment minimising COST.
- Main operation: Move from the current assignment to a selected assignment among its legal neighbours:

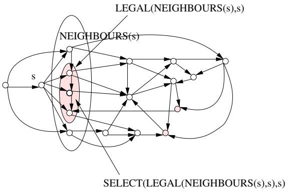


Local-Search Heuristics: Generic Algorithm

$s := \textsc{InitialAssignment}(V, U)$
$k := 0;\ s^* := s$ // s^* is the so far best assignment
while $\sum_{c \in C} \textsc{Violation}(c, s) > 0$ **and** $k < \mu$ **do**
  $k := k+1;\ s := \textsc{Select}(\textsc{Legal}(\textsc{Neighbours}(s), s), s)$
  if $\textsc{Cost}(s) < \textsc{Cost}(s^*)$ **then** $s^* := s$
return s^*

where (may need a meta-heuristic to escape local optima):

- $\textsc{Neighbours}(s)$ returns the neighbours of s.
- $\textsc{Legal}(N, s)$ returns the legal neighbours in N w.r.t. s.
- $\textsc{Select}(M, s)$ returns a selected element of M w.r.t. s.



Examples (LEGAL)

$$\text{Improving}(N, s) = \{n \in N \mid \textsc{Cost}(n) < \textsc{Cost}(s)\}$$

$$\text{NonWorsening}(N, s) = \{n \in N \mid \textsc{Cost}(n) \leq \textsc{Cost}(s)\}$$

$$\text{ViolatingVar}(N, s) = \{n \in N \mid n(x) \neq s(x) \text{ for a violating variable } x\}$$

$$\text{All}(N, s) = N$$

Examples (SELECT)

$$\text{First}(M, s) = \text{the first element in } M$$

$$\text{Best}(M, s) = \text{random}\left(\left\{n \in M \mid \textsc{Cost}(n) = \min_{t \in M} \textsc{Cost}(t)\right\}\right)$$

$$\text{RandomImproving}(M, s) = \textbf{let } n = \text{random}(M) \textbf{ in if } \textsc{Cost}(n) < \textsc{Cost}(s) \textbf{ then } n \textbf{ else } s$$

Local Search: Sample Heuristics

Examples (Heuristics for SELECT ∘ LEGAL)

Systematic (partial) exploration of the neighbourhood:

- First improving neighbour: First(Improving(N, s), s)
- Steepest / Gradient descent: Best(Improving(N, s), s)
- Min-conflict: Best(ViolatingVar(N, s), s)
- ...

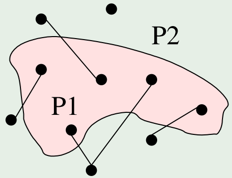
Random walk (pick a neighbour and decide on selecting it):

- Random improvement: RandomImproving(All(N, s), s)
- ...

Outline

Example (Graph Partitioning)

- **Problem:** Given a graph $G = (V, E)$, find a balanced partition $\langle P_1, P_2 \rangle$ of V that minimises the number of edges with end-points in both P_1 and P_2.
- **Definition:** A balanced partition $\langle P_1, P_2 \rangle$ of V satisfies $P_1 \cup P_2 = V$, $P_1 \cap P_2 = \varnothing$, and $-1 \leq |P_1| - |P_2| \leq 1$.
- **Example:**



We now design a greedy local-search heuristic.

Example (Graph Partitioning: Choices)

1. The **initial assignment** (InitialAssignment).
2. The **neighbourhood function** (Neighbours).
3. The **cost** of an assignment (Cost).
4. The **legal-neighbour filtering function** (Legal).
5. The **neighbour selection function** (Select).

Example (Graph Partitioning: Choices)

1. The **initial assignment** (INITIALASSIGNMENT).
   A random balanced partition $\langle P_1, P_2 \rangle$ of $G = (V, E)$.
2. The **neighbourhood function** (NEIGHBOURS).
3. The **cost** of an assignment (COST).
4. The **legal-neighbour filtering function** (LEGAL).
5. The **neighbour selection function** (SELECT).

Example (Graph Partitioning: Choices)

1. The **initial assignment** (InitialAssignment).
   A random balanced partition $\langle P_1, P_2 \rangle$ of $G = (V, E)$.
2. The **neighbourhood function** (Neighbours).
   Swapping two vertices:
3. The **cost** of an assignment (Cost).
4. The **legal-neighbour filtering function** (Legal).
5. The **neighbour selection function** (Select).

Example (Graph Partitioning: Choices)

1. The **initial assignment** (InitialAssignment).
   A random balanced partition $\langle P_1, P_2 \rangle$ of $G = (V, E)$.
2. The **neighbourhood function** (Neighbours).
   Swapping two vertices: $\textsc{Neighbours}(\langle P_1, P_2 \rangle) =$
   $\{\langle P_1 \setminus \{a\} \cup \{b\}, P_2 \setminus \{b\} \cup \{a\} \rangle \mid a \in P_1 \land b \in P_2\}$
3. The **cost** of an assignment (Cost).
4. The **legal-neighbour filtering function** (Legal).
5. The **neighbour selection function** (Select).

Example (Graph Partitioning: Choices)

1. The **initial assignment** (InitialAssignment).
   A random balanced partition $\langle P_1, P_2 \rangle$ of $G = (V, E)$.
2. The **neighbourhood function** (Neighbours).
   Swapping two vertices: $\text{Neighbours}(\langle P_1, P_2 \rangle) =$
   $\{\langle P_1 \setminus \{a\} \cup \{b\}, P_2 \setminus \{b\} \cup \{a\} \rangle \mid a \in P_1 \wedge b \in P_2\}$
3. The **cost** of an assignment (Cost).
   The number of edges with end-points in both P_1 and P_2, as the balance constraints cannot be violated:
4. The **legal-neighbour filtering function** (Legal).
5. The **neighbour selection function** (Select).

Example (Graph Partitioning: Choices)

1. The **initial assignment** (InitialAssignment).
   A random balanced partition $\langle P_1, P_2 \rangle$ of $G = (V, E)$.
2. The **neighbourhood function** (Neighbours).
   Swapping two vertices: $\text{Neighbours}(\langle P_1, P_2 \rangle) =$
   $\{\langle P_1 \setminus \{a\} \cup \{b\}, P_2 \setminus \{b\} \cup \{a\} \rangle \mid a \in P_1 \wedge b \in P_2\}$
3. The **cost** of an assignment (Cost).
   The number of edges with end-points in both P_1 and P_2, as the balance constraints cannot be violated:
   $\text{Cost}(\langle P_1, P_2 \rangle) = f(\langle P_1, P_2 \rangle) = |\{(a, b) \in E \mid a \in P_1 \wedge b \in P_2\}|$
4. The **legal-neighbour filtering function** (Legal).
5. The **neighbour selection function** (Select).

Example (Graph Partitioning: Choices)

1. The **initial assignment** (InitialAssignment).
   A random balanced partition $\langle P_1, P_2 \rangle$ of $G = (V, E)$.
2. The **neighbourhood function** (Neighbours).
   Swapping two vertices: $\textsc{Neighbours}(\langle P_1, P_2 \rangle) = \{\langle P_1 \setminus \{a\} \cup \{b\}, P_2 \setminus \{b\} \cup \{a\} \rangle \mid a \in P_1 \land b \in P_2\}$
3. The **cost** of an assignment (Cost).
   The number of edges with end-points in both P_1 and P_2, as the balance constraints cannot be violated:
   $\textsc{Cost}(\langle P_1, P_2 \rangle) = f(\langle P_1, P_2 \rangle) = |\{(a, b) \in E \mid a \in P_1 \land b \in P_2\}|$
4. The **legal-neighbour filtering function** (Legal).
   The improving neighbours:
5. The **neighbour selection function** (Select).

Example (Graph Partitioning: Choices)

1. The **initial assignment** (InitialAssignment).
   A random balanced partition $\langle P_1, P_2 \rangle$ of $G = (V, E)$.
2. The **neighbourhood function** (Neighbours).
   Swapping two vertices: $\textsc{Neighbours}(\langle P_1, P_2 \rangle) =$
   $\{\langle P_1 \setminus \{a\} \cup \{b\}, P_2 \setminus \{b\} \cup \{a\} \rangle \mid a \in P_1 \wedge b \in P_2\}$
3. The **cost** of an assignment (Cost).
   The number of edges with end-points in both P_1 and P_2, as the balance constraints cannot be violated:
   $\textsc{Cost}(\langle P_1, P_2 \rangle) = f(\langle P_1, P_2 \rangle) = |\{(a, b) \in E \mid a \in P_1 \wedge b \in P_2\}|$
4. The **legal-neighbour filtering function** (Legal).
   The improving neighbours:
   $\textsc{Legal}(N, \langle P_1, P_2 \rangle) = \text{Improving}(N, \langle P_1, P_2 \rangle)$
5. The **neighbour selection function** (Select).

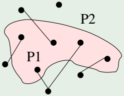
Example (Graph Partitioning: Choices)

1. The **initial assignment** (InitialAssignment).
   A random balanced partition $\langle P_1, P_2 \rangle$ of $G = (V, E)$.
2. The **neighbourhood function** (Neighbours).
   Swapping two vertices: $\textsc{Neighbours}(\langle P_1, P_2 \rangle) = \{\langle P_1 \setminus \{a\} \cup \{b\}, P_2 \setminus \{b\} \cup \{a\} \rangle \mid a \in P_1 \land b \in P_2\}$
3. The **cost** of an assignment (Cost).
   The number of edges with end-points in both P_1 and P_2, as the balance constraints cannot be violated:
   $\textsc{Cost}(\langle P_1, P_2 \rangle) = f(\langle P_1, P_2 \rangle) = |\{(a, b) \in E \mid a \in P_1 \land b \in P_2\}|$
4. The **legal-neighbour filtering function** (Legal).
   The improving neighbours:
   $\textsc{Legal}(N, \langle P_1, P_2 \rangle) = \text{Improving}(N, \langle P_1, P_2 \rangle)$
5. The **neighbour selection function** (Select).
   A random best legal neighbour:

Example (Graph Partitioning: Choices)

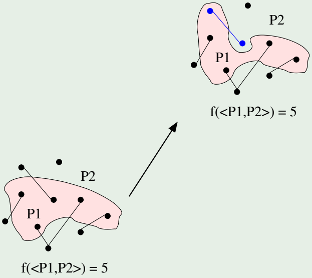
1. The **initial assignment** (InitialAssignment).
   A random balanced partition $\langle P_1, P_2 \rangle$ of $G = (V, E)$.
2. The **neighbourhood function** (Neighbours).
   Swapping two vertices: $\text{Neighbours}(\langle P_1, P_2 \rangle) =$
   $\{\langle P_1 \setminus \{a\} \cup \{b\}, P_2 \setminus \{b\} \cup \{a\}\rangle \mid a \in P_1 \land b \in P_2\}$
3. The **cost** of an assignment (Cost).
   The number of edges with end-points in both P_1 and P_2, as the balance constraints cannot be violated:
   $\text{Cost}(\langle P_1, P_2 \rangle) = f(\langle P_1, P_2 \rangle) = |\{(a, b) \in E \mid a \in P_1 \land b \in P_2\}|$
4. The **legal-neighbour filtering function** (Legal).
   The improving neighbours:
   $\text{Legal}(N, \langle P_1, P_2 \rangle) = \text{Improving}(N, \langle P_1, P_2 \rangle)$
5. The **neighbour selection function** (Select).
   A random best legal neighbour:
   $\text{Select}(M, \langle P_1, P_2 \rangle) = \text{Best}(M, \langle P_1, P_2 \rangle)$

Example (Graph Partitioning: Sample Run)

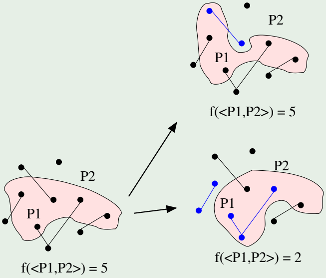


f(<P1,P2>) = 5

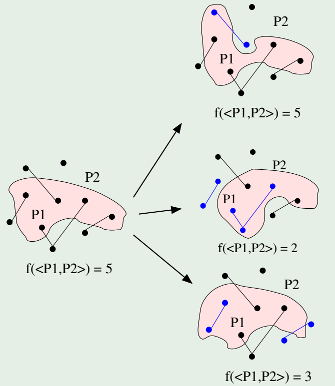
Example (Graph Partitioning: Sample Run)



Example (Graph Partitioning: Sample Run)

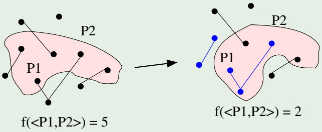


Example (Graph Partitioning: Sample Run)

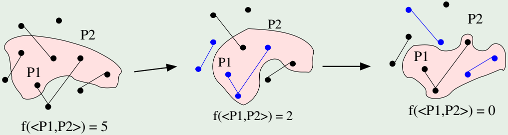


and 22 other probed neighbours $\langle P_1, P_2 \rangle$, but none of which with $f(\langle P_1, P_2 \rangle) < 2$

Example (Graph Partitioning: Sample Run)



Example (Graph Partitioning: Sample Run)



and 24 other probed neighbours $\langle P_1, P_2 \rangle$,
obviously none of which with $f(\langle P_1, P_2 \rangle) < 0$:
the trivial lower bound was reached, so search can stop,
with proven optimality (this is rare, in general)!

Example (Graph Partitioning)

Fundamental property of the chosen neighbourhood:
If a partition $\langle P_1, P_2 \rangle$ is balanced, then
each partition in $\textsc{Neighbours}(\langle P_1, P_2 \rangle)$ is also balanced.

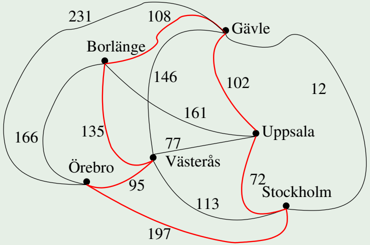
- Only satisfying assignments are considered, including the randomly generated initial assignment.
- The balance constraints are not checked explicitly.
- This is a common and often crucial technique: some constraints are explicit (either soft or one-way), while other constraints are implicit, in the sense that they are satisfied by the generated initial assignment and kept satisfied during search by the neighbourhood. Constraints are hard (either implicit or one-way) or soft.
- The size of the neighbourhood is $\left\lfloor \frac{|V|}{2} \right\rfloor \cdot \left\lceil \frac{|V|}{2} \right\rceil$.
- The search space is connected: any optimal solution can be reached from any assignment.

UPPSALA UNIVERSITET

Outline

Example (Travelling Salesperson)

- **Problem:** Given a set of cities with connecting roads, find a tour (a Hamiltonian circuit) that visits each city exactly once, with the minimum travel distance.
- **Representation:** We see the set of cities as vertices V and the set of roads as edges E in a (not necessarily complete) undirected graph $G = (V, E)$.



- **Example:** s:

We now design a greedy local-search heuristic.

Example (Travelling Salesperson: Choices)

1. The **initial assignment** (INITIALASSIGNMENT).

2. The **neighbourhood function** (NEIGHBOURS).

3. The **cost** of an assignment (COST).

4. The **legal-neighbour filtering function** (LEGAL).

5. The **neighbour selection function** (SELECT).

Example (Travelling Salesperson: Choices)

1. The **initial assignment** (InitialAssignment).
   A random edge set $T \subseteq E$ that forms a tour: NP-hard!
2. The **neighbourhood function** (Neighbours).
3. The **cost** of an assignment (Cost).
4. The **legal-neighbour filtering function** (Legal).
5. The **neighbour selection function** (Select).

Example (Travelling Salesperson: Choices)

1. The **initial assignment** (InitialAssignment).
   A random edge set $T \subseteq E$ that forms a tour: NP-hard!
   Complete E by adding infinite-distance edges:
   now any random permutation of V yields a tour.
2. The **neighbourhood function** (Neighbours).
3. The **cost** of an assignment (Cost).
4. The **legal-neighbour filtering function** (Legal).
5. The **neighbour selection function** (Select).

Example (Travelling Salesperson: Choices)

1. The **initial assignment** (InitialAssignment).
   A random edge set $T \subseteq E$ that forms a tour: NP-hard!
   Complete E by adding infinite-distance edges:
   now any random permutation of V yields a tour.
2. The **neighbourhood function** (Neighbours).
   Replace two edges by two other edges so that the edge set remains a tour:
3. The **cost** of an assignment (Cost).
4. The **legal-neighbour filtering function** (Legal).
5. The **neighbour selection function** (Select).

Example (Travelling Salesperson: Choices)

1. The **initial assignment** (InitialAssignment).
   A random edge set $T \subseteq E$ that forms a tour: NP-hard!
   Complete E by adding infinite-distance edges:
   now any random permutation of V yields a tour.
2. The **neighbourhood function** (Neighbours).
   Replace two edges by two other edges so that the edge set remains a tour: $\textsc{Neighbours}(T) =$
   $\{T \setminus \{(i,i'),(j,j')\} \cup \{(i,j),(i',j')\} \mid i,j \in V \text{ where } (i,j) \notin T\}$
3. The **cost** of an assignment (Cost).
4. The **legal-neighbour filtering function** (Legal).
5. The **neighbour selection function** (Select).

Example (Travelling Salesperson: Choices)

1. The **initial assignment** (INITIALASSIGNMENT).
   A random edge set $T \subseteq E$ that forms a tour: NP-hard!
   Complete E by adding infinite-distance edges:
   now any random permutation of V yields a tour.
2. The **neighbourhood function** (NEIGHBOURS).
   Replace two edges by two other edges so that the edge set remains a tour: $\text{NEIGHBOURS}(T) =$
   $\{T \setminus \{(i,i'),(j,j')\} \cup \{(i,j),(i',j')\} \mid i,j \in V \text{ where } (i,j) \notin T\}$
3. The **cost** of an assignment (COST).
   The sum of all distances, as the tour constraint cannot be violated:
4. The **legal-neighbour filtering function** (LEGAL).
5. The **neighbour selection function** (SELECT).

Example (Travelling Salesperson: Choices)

1. The **initial assignment** (InitialAssignment).
   A random edge set $T \subseteq E$ that forms a tour: NP-hard!
   Complete E by adding infinite-distance edges:
   now any random permutation of V yields a tour.
2. The **neighbourhood function** (Neighbours).
   Replace two edges by two other edges so that the edge set remains a tour: $\text{Neighbours}(T) = \{T \setminus \{(i,i'),(j,j')\} \cup \{(i,j),(i',j')\} \mid i,j \in V \text{ where } (i,j) \notin T\}$
3. The **cost** of an assignment (Cost).
   The sum of all distances, as the tour constraint cannot be violated: $\text{Cost}(T) = f(T) = \sum_{(a,b) \in T} \text{Distance}(a,b)$
4. The **legal-neighbour filtering function** (Legal).
5. The **neighbour selection function** (Select).

Example (Travelling Salesperson: Choices)

1. The **initial assignment** (InitialAssignment).
   A random edge set $T \subseteq E$ that forms a tour: NP-hard!
   Complete E by adding infinite-distance edges:
   now any random permutation of V yields a tour.
2. The **neighbourhood function** (Neighbours).
   Replace two edges by two other edges so that the edge set remains a tour: $\textsc{Neighbours}(T) = \{T \setminus \{(i,i'),(j,j')\} \cup \{(i,j),(i',j')\} \mid i,j \in V \text{ where } (i,j) \notin T\}$
3. The **cost** of an assignment (Cost).
   The sum of all distances, as the tour constraint cannot be violated: $\textsc{Cost}(T) = f(T) = \sum_{(a,b)\in T} \text{Distance}(a,b)$
4. The **legal-neighbour filtering function** (Legal).
   The improving neighbours:
5. The **neighbour selection function** (Select).

Example (Travelling Salesperson: Choices)

1. The **initial assignment** (InitialAssignment).
   A random edge set $T \subseteq E$ that forms a tour: NP-hard!
   Complete E by adding infinite-distance edges:
   now any random permutation of V yields a tour.
2. The **neighbourhood function** (Neighbours).
   Replace two edges by two other edges so that the edge set remains a tour: $\textsc{Neighbours}(T) =$
   $\{T \setminus \{(i,i'),(j,j')\} \cup \{(i,j),(i',j')\} \mid i,j \in V \text{ where } (i,j) \notin T\}$
3. The **cost** of an assignment (Cost).
   The sum of all distances, as the tour constraint cannot be violated: $\textsc{Cost}(T) = f(T) = \sum_{(a,b)\in T} \mathrm{Distance}(a,b)$
4. The **legal-neighbour filtering function** (Legal).
   The improving neighbours: $\textsc{Legal}(N,T) = \mathrm{Improving}(N,T)$
5. The **neighbour selection function** (Select).

Example (Travelling Salesperson: Choices)

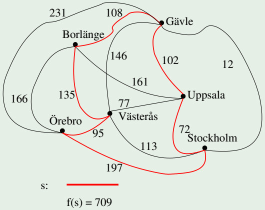
1. The **initial assignment** (INITIALASSIGNMENT).
   A random edge set $T \subseteq E$ that forms a tour: NP-hard!
   Complete E by adding infinite-distance edges:
   now any random permutation of V yields a tour.
2. The **neighbourhood function** (NEIGHBOURS).
   Replace two edges by two other edges so that the edge set remains a tour: $\textsc{Neighbours}(T) =$
   $\{T \setminus \{(i,i'),(j,j')\} \cup \{(i,j),(i',j')\} \mid i,j \in V \text{ where } (i,j) \notin T\}$
3. The **cost** of an assignment (COST).
   The sum of all distances, as the tour constraint cannot be violated: $\textsc{Cost}(T) = f(T) = \sum_{(a,b)\in T} \text{Distance}(a,b)$
4. The **legal-neighbour filtering function** (LEGAL).
   The improving neighbours: $\textsc{Legal}(N,T) = \text{Improving}(N,T)$
5. The **neighbour selection function** (SELECT).
   A random best legal neighbour:

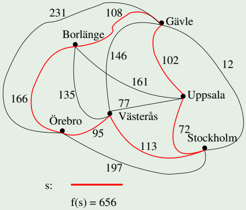
Example (Travelling Salesperson: Choices)

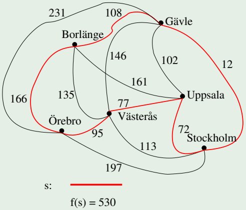
1. The **initial assignment** (InitialAssignment).
   A random edge set $T \subseteq E$ that forms a tour: NP-hard!
   Complete E by adding infinite-distance edges:
   now any random permutation of V yields a tour.
2. The **neighbourhood function** (Neighbours).
   Replace two edges by two other edges so that the edge set remains a tour: $\textsc{Neighbours}(T) =$
   $\{T \setminus \{(i,i'),(j,j')\} \cup \{(i,j),(i',j')\} \mid i,j \in V \text{ where } (i,j) \notin T\}$
3. The **cost** of an assignment (Cost).
   The sum of all distances, as the tour constraint cannot be violated: $\textsc{Cost}(T) = f(T) = \sum_{(a,b)\in T} \mathrm{Distance}(a,b)$
4. The **legal-neighbour filtering function** (Legal).
   The improving neighbours: $\textsc{Legal}(N,T) = \mathrm{Improving}(N,T)$
5. The **neighbour selection function** (Select).
   A random best legal neighbour:
   $\textsc{Select}(M,T) = \mathrm{Best}(M,T)$

Example (Travelling Salesperson: Sample Run)

Three consecutive improving satisfying assignments:







Example (Travelling Salesperson)

Fundamental property of the chosen neighbourhood:
If an edge set T is a tour,
then each edge set in $\textsc{Neighbours}(T)$ is also a tour.

- Only satisfying assignments are considered, including the randomly generated initial assignment, but sub-optimality surely occurs if some of the added infinite-distance edges are used.
- The tour constraint is not checked explicitly.
- Making all constraints implicit (by the search) is not always possible: moves to non-satisfying assignments must also be considered (as seen in the next section).
- This neighbourhood is called 2-opt: two edges on the current tour are replaced.
- The size of the neighbourhood is $|V| \cdot (|V| - 2)$, that is $6 \cdot 4 = 24$ neighbours for our instance.

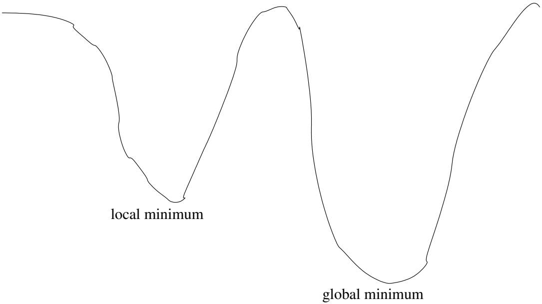
Outline

Heuristics drive the search to (good enough) solutions:

- Which decision variables are modified in a move?
- Which new values do they get in the move?

Meta-heuristics drive the search to global optima of COST:

- Avoid cycles of moves & escape local optima of COST.
- Explore many parts of the search space.
- Focus on promising parts of the search space.



Examples (Meta-heuristics)

- Tabu search (1986):
  forbid recent moves from being done again.
- Simulated annealing (1983):
  consider random moves and make worsening ones with a probability that decreases over time.
- Genetic algorithms (1975):
  use a pool of current assignments and cross them.

Tabu Search (Glover and Laguna, 1997)

- In order to escape local optima, we must be able to accept worse assignments, that is assignments that increase the value of COST.
- To avoid ending up in cycles, tabu search remembers the last λ assignments in a tabu list and makes them tabu (or taboo): moves in this list cannot be chosen, even if this implies increasing the value of COST.

Tabu Search

Compare with the generic algorithm of slide 16:

$s := \text{INITIALASSIGNMENT}(V, U)$
$k := 0; s^* := s$ // s^* is the so far best assignment
$\tau := [s]$ // initialise the tabu list
while $\sum_{c \in C} \text{VIOLATION}(c, s) > 0 \wedge k < \mu$ **do**
  $k := k + 1; s := \text{Best}(\text{NonTabu}(\text{NEIGHBOURS}(s), \tau), \tau)$
  $\tau := \tau :: s$ // but keep only the last λ assignments
  if $\text{COST}(s) < \text{COST}(s^*)$ **then**
    $s^* := s$
return s^*

function $\text{NonTabu}(N, \tau)$
return $\{n \in N \mid n \notin \tau\}$

Outline

Evaluation of Local Search

We have seen local-search algorithms for two problems:

- It is hard to reuse (parts of) a local-search algorithm of one problem for other problems.
- We want reusable software components!

In constraint-based local search (CBLS) (Van Hentenryck and Michel, 2005):

- A problem is modelled as a conjunction of constraints, whose predicates declaratively encapsulate inference algorithms that are specific to frequent combinatorial substructures and are thus reusable.
- A master search algorithm operates on the model, guided by user-indicated or designed (meta-)heuristics.

CBLS by itself makes no contributions to the state of the art of neighbourhoods, heuristics, and meta-heuristics, but it simplifies their formulation and improves their reusability.

CP Solving = Inference + Search

A CP solver conducts search interleaved with inference:

Each constraint has an inference algorithm.

Outline

Definition

Each constraint predicate has a violation function:
the violation of a constraint is zero if it is currently satisfied,
else a positive value proportional to its dissatisfaction.

Example

For `x <= y` and current assignment s, define the violation to be $s(\mathrm{x}) - s(\mathrm{y})$ if $s(\mathrm{x}) \not\leq s(\mathrm{y})$, and 0 otherwise.

Definitions

A constraint with violation is explicit in a CBLS model and soft: it can be violated during search but ought to be satisfied in a solution.

The constraint violations are queried during search.

Definitions

A one-way constraint is explicit in a CBLS model and hard: it is kept satisfied during search by the solver.

Example

For `p = x * y`, if `x` or `y` or both are reassigned by a move to assignment s, then $s(\mathtt{p})$ is to be automatically set by the solver to $s(\mathtt{x}) \cdot s(\mathtt{y})$.

CBLS solvers offer a syntax for one-way constraints, such as `p <== x * y` in OscaR.cbls, but CP solvers (such as Gecode) and technology-independent modelling languages (such as MiniZinc) do not make such a distinction.

Definitions

An *implicit constraint* is not in a CBLS model but hard: it is kept satisfied during search by choosing a satisfying initial assignment and only making satisfaction-preserving moves, by the use of a *constraint-specific neighbourhood*.

A constraint is implicit by search, or implied within a model.

Example

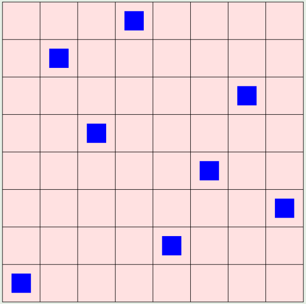
For `distinct`, when there are as many variables as values: the initial assignment gives distinct values to all the variables (by random permutation), and the neighbourhood only has moves that swap the values of two variables.

When building a CBLS model, a MiniZinc backend must:

- Aptly assort the otherwise all explicit & soft constraints.
- Add suitable neighbourhood, heuristic, meta-heuristic.

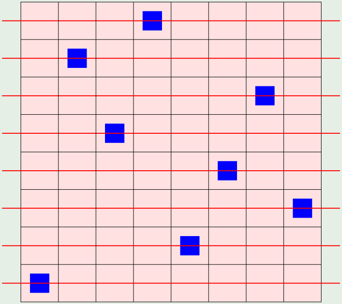
This is much more involved than just flattening and solving.

Example (8 Queens)



Place 8 queens on a chess board such that no two queens attack each other:

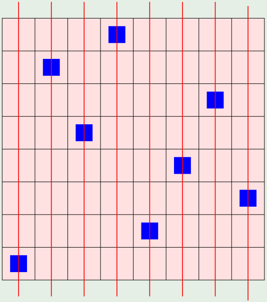
Example (8 Queens)



Place 8 queens on a chess board such that no two queens attack each other:

1. No two queens are on the same row.

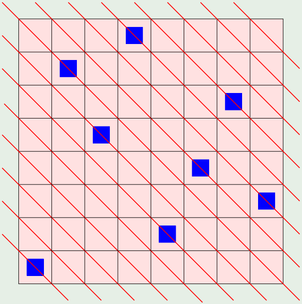
Example (8 Queens)



Place 8 queens on a chess board such that no two queens attack each other:

1. No two queens are on the same row.
2. No two queens are on the same column.

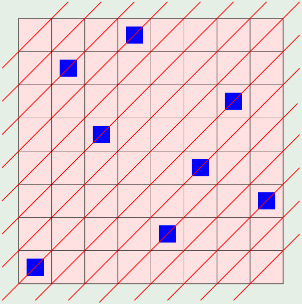
Example (8 Queens)



Place 8 queens on a chess board such that no two queens attack each other:

1. No two queens are on the same row.
2. No two queens are on the same column.
3. No two queens are on the same down-diagonal.

Example (8 Queens)



Place 8 queens on a chess board such that no two queens attack each other:

1. No two queens are on the same row.
2. No two queens are on the same column.
3. No two queens are on the same down-diagonal.
4. No two queens are on the same up-diagonal.

Example (8 Queens: CBLS Models)

Let variable $R[c]$ represent the row of the queen in col. c:

1. No two queens are on the same row:
2. No two queens are on the same column:
3. No two queens are on the same down-diagonal:
4. No two queens are on the same up-diagonal:

Example (8 Queens: CBLS Models)

Let variable $R[c]$ represent the row of the queen in col. c:

1. No two queens are on the same row:
   $\forall c, c' \in 1..8$ **where** $c < c' : R[c] \neq R[c']$,
   that is `distinct`$([R[1], \ldots, R[8]])$
2. No two queens are on the same column:
3. No two queens are on the same down-diagonal:
4. No two queens are on the same up-diagonal:

Example (8 Queens: CBLS Models)

Let variable $R[c]$ represent the row of the queen in col. c:

1. No two queens are on the same row:
   $\forall c, c' \in 1..8$ **where** $c < c' : R[c] \neq R[c'],$
   that is `distinct`$([R[1], \ldots, R[8]])$
2. No two queens are on the same column:
   Guaranteed by the choice of the decision variables.
3. No two queens are on the same down-diagonal:
4. No two queens are on the same up-diagonal:

Example (8 Queens: CBLS Models)

Let variable $R[c]$ represent the row of the queen in col. c:

1. No two queens are on the same row:
   $\forall c, c' \in 1..8$ **where** $c < c' : R[c] \neq R[c']$,
   that is $\texttt{distinct}([R[1], \ldots, R[8]])$
2. No two queens are on the same column:
   Guaranteed by the choice of the decision variables.
3. No two queens are on the same down-diagonal:
   $\forall c, c' \in 1..8$ **where** $c < c' : R[c] - c \neq R[c'] - c'$,
   that is $\texttt{distinct}([R[1] - 1, \ldots, R[8] - 8])$
4. No two queens are on the same up-diagonal:

Example (8 Queens: CBLS Models)

Let variable $R[c]$ represent the row of the queen in col. c:

1. No two queens are on the same row:
   $\forall c, c' \in 1..8$ **where** $c < c' : R[c] \neq R[c']$,
   that is $\texttt{distinct}([R[1], \ldots, R[8]])$
2. No two queens are on the same column:
   Guaranteed by the choice of the decision variables.
3. No two queens are on the same down-diagonal:
   $\forall c, c' \in 1..8$ **where** $c < c' : R[c] - c \neq R[c'] - c'$,
   that is $\texttt{distinct}([R[1] - 1, \ldots, R[8] - 8])$
4. No two queens are on the same up-diagonal:
   $\forall c, c' \in 1..8$ **where** $c < c' : R[c] + c \neq R[c'] + c'$,
   that is $\texttt{distinct}([R[1] + 1, \ldots, R[8] + 8])$

Example (8 Queens: CBLS Models)

Let variable $R[c]$ represent the row of the queen in col. c:

1. No two queens are on the same row:
   $\forall c, c' \in 1..8$ **where** $c < c' : R[c] \neq R[c']$,
   that is $\texttt{distinct}([R[1], \ldots, R[8]])$
2. No two queens are on the same column:
   Guaranteed by the choice of the decision variables.
3. No two queens are on the same down-diagonal:
   $\forall c, c' \in 1..8$ **where** $c < c' : R[c] - c \neq R[c'] - c'$,
   that is $\texttt{distinct}([R[1] - 1, \ldots, R[8] - 8])$
4. No two queens are on the same up-diagonal:
   $\forall c, c' \in 1..8$ **where** $c < c' : R[c] + c \neq R[c'] + c'$,
   that is $\texttt{distinct}([R[1] + 1, \ldots, R[8] + 8])$

Better model: Make the row constraint implicit, by using a random permutation of 1..8 as initial assignment and using a neighbourhood that keeps the row constraint satisfied.

Outline

Constraint Predicates in Local Search

The predicate of a soft constraint c is equipped with:

- A constraint violation function $\text{VIOLATION}(c, s)$, which estimates how much c is violated under the current assignment s: $\text{VIOLATION}(c, s) = 0$ if and only if c is satisfied, and $\text{VIOLATION}(c, s) > 0$ otherwise.
- A variable violation function $\text{VIOLATION}(c, s, x)$, which estimates how much a suitable change of the value of the decision variable x can decrease $\text{VIOLATION}(c, s)$.
- . . . (to be continued)

At the constraint-system level, one can query:

- The system constraint violation under s of a constraint system $C' \subseteq C$ is $\sum_{c \in C'} \text{VIOLATION}(c, s)$.
- The system variable violation under s of a variable x in a system $C' \subseteq C$ is $\sum_{c \in C'} \text{VIOLATION}(c, s, x)$.

Example ($x \neq y$)

When $x = 4$ and $y = 5$:

- The constraint violation is 0: the constraint is satisfied.
- The variable violations of x and y are both 0.

When $x = 4$ and $y = 4$:

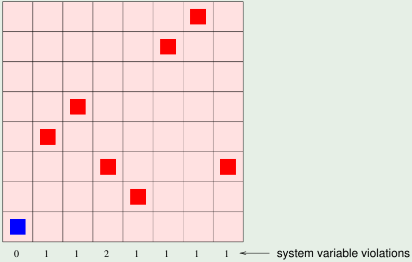
- The constraint violation is 1: the constraint is violated.
- The variable violations of x and y are both 1.

Example ($\texttt{distinct}([a, b, c, d])$)

When $a = 5$, $b = 5$, $c = 5$, $d = 6$, all with domain D:

- The constraint violation is 2, since at least two variables must be changed to reach a satisfying assignment: $\sum_{v \in D} \max(\text{occ}[v] - 1,\ 0)$, where $\text{occ}[v]$ stores the current number of occurrences of value v.
- The variable violations of a, b, c are 1, and 0 for d.

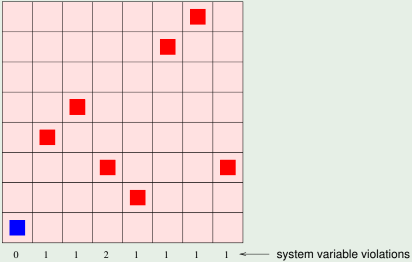
Example (8 Queens: Violations)



Let the upper-left corner have the coordinates $(1, 1)$:

- $\texttt{distinct}([R[1], \ldots, R[8]])$
- $\texttt{distinct}([R[1] - 1, \ldots, R[8] - 8])$
- $\texttt{distinct}([R[1] + 1, \ldots, R[8] + 8])$

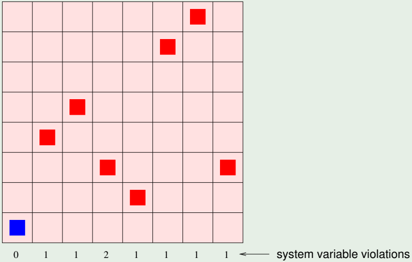
Example (8 Queens: Violations)



Let the upper-left corner have the coordinates $(1, 1)$:

- $\texttt{distinct}([R[1], \ldots, R[8]])$
  The violation of $\texttt{distinct}([8, 5, 4, 6, 7, 2, 1, 6])$ is 1.
- $\texttt{distinct}([R[1] - 1, \ldots, R[8] - 8])$
- $\texttt{distinct}([R[1] + 1, \ldots, R[8] + 8])$

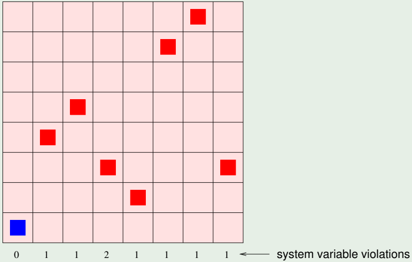
Example (8 Queens: Violations)



Let the upper-left corner have the coordinates $(1, 1)$:

- $\texttt{distinct}([R[1], \ldots, R[8]])$
  The violation of $\texttt{distinct}([8, 5, 4, 6, 7, 2, 1, 6])$ is 1.
- $\texttt{distinct}([R[1] - 1, \ldots, R[8] - 8])$
  The violation of $\texttt{distinct}([7, 3, 1, 2, 2, -4, -6, -2])$ is 1.
- $\texttt{distinct}([R[1] + 1, \ldots, R[8] + 8])$

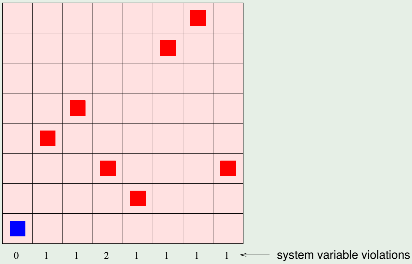
Example (8 Queens: Violations)



Let the upper-left corner have the coordinates $(1, 1)$:

- $\texttt{distinct}([R[1], \ldots, R[8]])$
  The violation of $\texttt{distinct}([8, 5, 4, 6, 7, 2, 1, 6])$ is 1.
- $\texttt{distinct}([R[1] - 1, \ldots, R[8] - 8])$
  The violation of $\texttt{distinct}([7, 3, 1, 2, 2, -4, -6, -2])$ is 1.
- $\texttt{distinct}([R[1] + 1, \ldots, R[8] + 8])$
  The violation of $\texttt{distinct}([9, 7, 7, 10, 12, 8, 8, 14])$ is 2.

Example (8 Queens: Violations)



Let the upper-left corner have the coordinates $(1, 1)$:

- $\texttt{distinct}([R[1], \ldots, R[8]])$
  The violation of $\texttt{distinct}([8, 5, 4, 6, 7, 2, 1, 6])$ is 1.
- $\texttt{distinct}([R[1] - 1, \ldots, R[8] - 8])$
  The violation of $\texttt{distinct}([7, 3, 1, 2, 2, -4, -6, -2])$ is 1.
- $\texttt{distinct}([R[1] + 1, \ldots, R[8] + 8])$
  The violation of $\texttt{distinct}([9, 7, 7, 10, 12, 8, 8, 14])$ is 2.

The system constraint violation is $1 + 1 + 2 = 4$.

Outline

Constr. Predicates in Local Search (cont'd)

The predicate of a soft constraint c is also equipped with:

- An assignment delta function $\textsc{Delta}(c, s, x := v)$, which estimates the increase of $\textsc{Violation}(c, s)$ upon a probed $x := v$ assignment move for variable x and its domain value v.
- A swap delta function $\textsc{Delta}(c, s, x :=: y)$, which estimates the increase of $\textsc{Violation}(c, s)$ upon a probed $x :=: y$ swap move for two variables x and y.

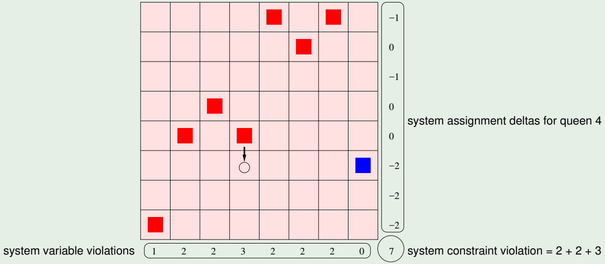
The more negative a delta the better the probed move!

At the constraint-system level, one can query:

- The system assignment delta under s of $x := v$ in a system $C' \subseteq C$ is $\sum_{c \in C'} \textsc{Delta}(c, s, x := v)$.
- The system swap delta under s of $x :=: y$ in a system $C' \subseteq C$ is $\sum_{c \in C'} \textsc{Delta}(c, s, x :=: y)$.

Other kinds of moves can be added.

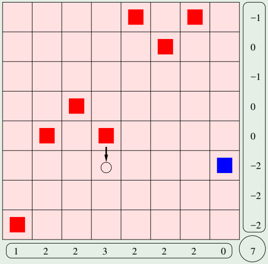
Example (8 Queens: Computing Deltas in $\mathcal{O}(1)$ Time)



- $\texttt{distinct}([R[1], \dots, R[4], \dots, R[8]])$
- $\texttt{distinct}([R[1]-1, \dots, R[4]-4, \dots, R[8]-8])$
- $\texttt{distinct}([R[1]+1, \dots, R[4]+4, \dots, R[8]+8])$

The violation increases by $[\mathrm{occ}[v] \geq 1] - [\mathrm{occ}[s(x)] \geq 2]$ upon $x := v$.

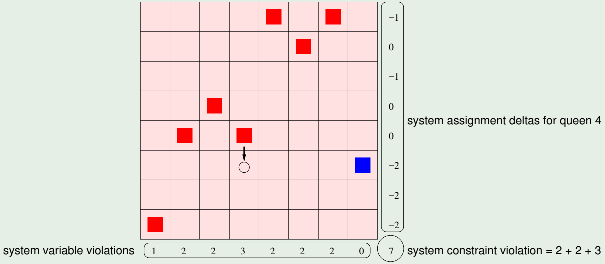
Example (8 Queens: Computing Deltas in $\mathcal{O}(1)$ Time)



- $\texttt{distinct}([R[1],\dots,{\color{red}R[4]},\dots,R[8]])$
  Delta of $R[4] := 6$ in $\texttt{distinct}([8,5,4,{\color{red}5},1,2,1,6])$ is ± 0.
- $\texttt{distinct}([R[1]-1,\dots,{\color{red}R[4]-4},\dots,R[8]-8])$
- $\texttt{distinct}([R[1]+1,\dots,{\color{red}R[4]+4},\dots,R[8]+8])$

The violation increases by $[\mathrm{occ}[v] \geq 1] - [\mathrm{occ}[s(x)] \geq 2]$ upon $x := v$.

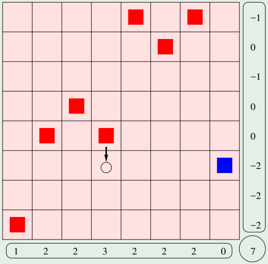
Example (8 Queens: Computing Deltas in $\mathcal{O}(1)$ Time)



- $\texttt{distinct}([R[1], \ldots, R[4], \ldots, R[8]])$
  Delta of $R[4] := 6$ in $\texttt{distinct}([8, 5, 4, 5, 1, 2, 1, 6])$ is ± 0.
- $\texttt{distinct}([R[1] - 1, \ldots, R[4] - 4, \ldots, R[8] - 8])$
  Delta of $R[4] := 6$ in $\texttt{distinct}([7, 3, 1, 1, -4, -4, -6, -2])$ is -1.
- $\texttt{distinct}([R[1] + 1, \ldots, R[4] + 4, \ldots, R[8] + 8])$

The violation increases by $[\text{occ}[v] \geq 1] - [\text{occ}[s(x)] \geq 2]$ upon $x := v$.

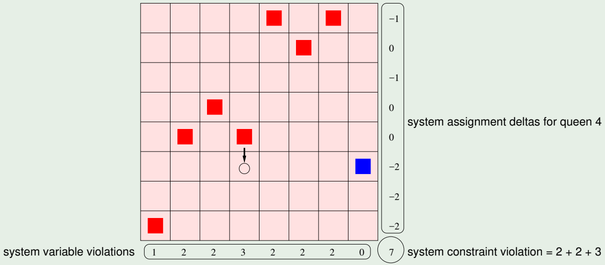
Example (8 Queens: Computing Deltas in $\mathcal{O}(1)$ Time)



- $\texttt{distinct}([R[1], \ldots, {\color{red}R[4]}, \ldots, R[8]])$
  Delta of $R[4] := 6$ in $\texttt{distinct}([8, 5, 4, {\color{red}5}, 1, 2, 1, 6])$ is ± 0.
- $\texttt{distinct}([R[1] - 1, \ldots, {\color{red}R[4] - 4}, \ldots, R[8] - 8])$
  Delta of $R[4] := 6$ in $\texttt{distinct}([7, 3, 1, {\color{red}1}, -4, -4, -6, -2])$ is -1.
- $\texttt{distinct}([R[1] + 1, \ldots, {\color{red}R[4] + 4}, \ldots, R[8] + 8])$
  Delta of $R[4] := 6$ in $\texttt{distinct}([9, 7, 7, {\color{red}9}, 6, 8, 8, 14])$ is -1.

The violation increases by ${\color{red}[}\mathrm{occ}[v] \geq 1{\color{red}]} - {\color{red}[}\mathrm{occ}[s(x)] \geq 2{\color{red}]}$ upon $x := v$.

Example (8 Queens: Computing Deltas in $\mathcal{O}(1)$ Time)



- $\texttt{distinct}([R[1], \ldots, {\color{red}R[4]}, \ldots, R[8]])$
  Delta of $R[4] := 6$ in $\texttt{distinct}([8, 5, 4, {\color{red}5}, 1, 2, 1, 6])$ is ± 0.
- $\texttt{distinct}([R[1] - 1, \ldots, {\color{red}R[4] - 4}, \ldots, R[8] - 8])$
  Delta of $R[4] := 6$ in $\texttt{distinct}([7, 3, 1, {\color{red}1}, -4, -4, -6, -2])$ is -1.
- $\texttt{distinct}([R[1] + 1, \ldots, {\color{red}R[4] + 4}, \ldots, R[8] + 8])$
  Delta of $R[4] := 6$ in $\texttt{distinct}([9, 7, 7, {\color{red}9}, 6, 8, 8, 14])$ is -1.

The system assignment delta of $R[4] := 6$ is $0 + (-1) + (-1) = -2$.

Outline

Constraint Predicates in Local Search (end)

- The functions equipping a constraint predicate can be queried in order to guide the local search:
  - The constraint violation functions can be queried to find promising constraint(s) in order to select promising decision variable(s) to reassign in a move.
  - The variable violation functions can be queried to select promising decision variable(s) to reassign in a move.
  - The probing functions can be queried to select a move in a good direction for a variable or constraint (system).
- The violation functions are the counterpart of the subsumption checking of systematic CP-style solving.
- The probing functions are the counterpart of the propagators of systematic CP-style solving.
- These functions must be implemented for highest time and space efficiency, as they may be queried in the probing of the neighbourhood at each search iteration.

Symmetry Handling in Local Search

When solving combinatorial problems by local search, the idea is often to exploit the presence of symmetries by doing nothing, rather than by making the search space smaller, as with CP / MIP / SAT / SMT-style systematic search.

Outline

The Comet Toolchain

Comet was a language and toolchain for the modelling and solving of constraint problems, inspired by Localizer (2000).

Comet had a CBLS back-end (Van Hentenryck and Michel, 2005), as well as CP (systematic search with propagation) and MIP (mixed integer linear programming) back-ends:

- High-level software components (constraint predicates) for formulating constraint models of problems.
- High-level constructs for specifying search algorithms.
- An open architecture allowing user-defined extensions.

Comet was free for academic purposes. It inspired, among others, the CBLS back-end of OscaR, which is open-source at `https://bitbucket.org/oscarlib/oscar/wiki`.

Example (8 Queens: Comet CBLS Model)

```
import cotls;
Solver<LS> m();
int n = 8;
range Size = 1..n;
UniformDistribution distr(Size);
var{int} R[Size](m,Size) := distr.get();
ConstraintSystem<LS> S(m);
S.post(alldifferent(R));
S.post(alldifferent(all(c in Size) R[c]-c));
S.post(alldifferent(all(c in Size) R[c]+c));
m.close();
```

Define an array `R` of 8 variables and initialise each variable with a random (possibly repeated) value in the domain 1..8.

Better: Make the constraint `alldifferent(R)` implicit, by using a random permutation of 1..8 as initial assignment.

Example (8 Queens: Comet CBLS Search)

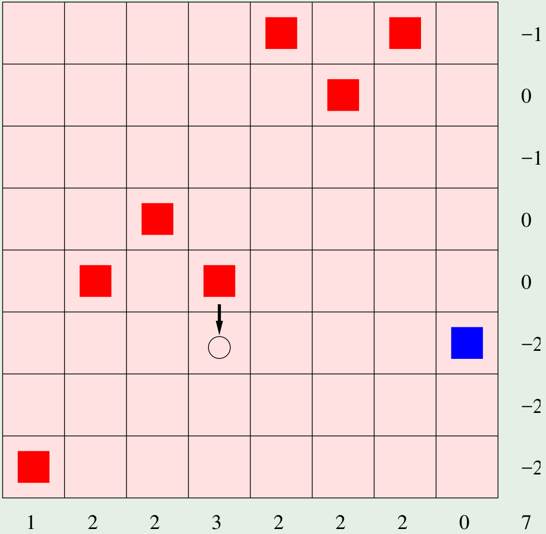
```
int k = 0;
while (S.violations() > 0 && k < 50 * n) {
  selectMax(c in Size)(S.violations(R[c]))
    selectMin(r in Size)(S.getAssignDelta(R[c],r))
      R[c] := r;
  k++;
}
```

In words:

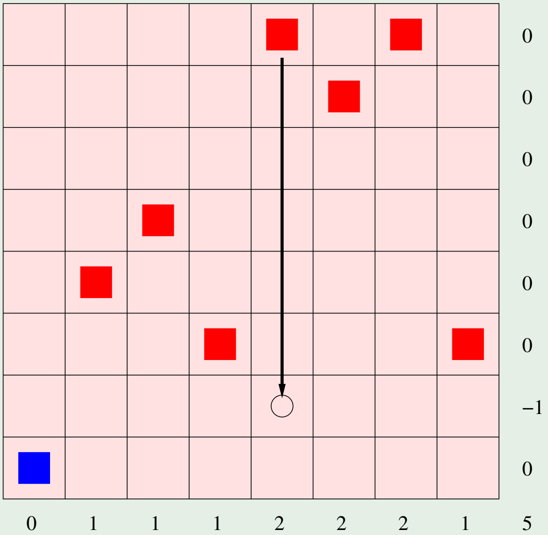
initialise the iteration counter to zero
while there are a violated constraint in system `S` and iterations left **do**
select a variable `R[c]` with the maximum violation in system `S`
select a value `r` with the minimum assignment delta for `R[c]` in `S`
assign value `r` to decision variable `R[c]`
increment the iteration counter

Better (continued): Keep the row constraint satisfied by a neighbourhood of swap moves `R[c] :=: R[c']`.

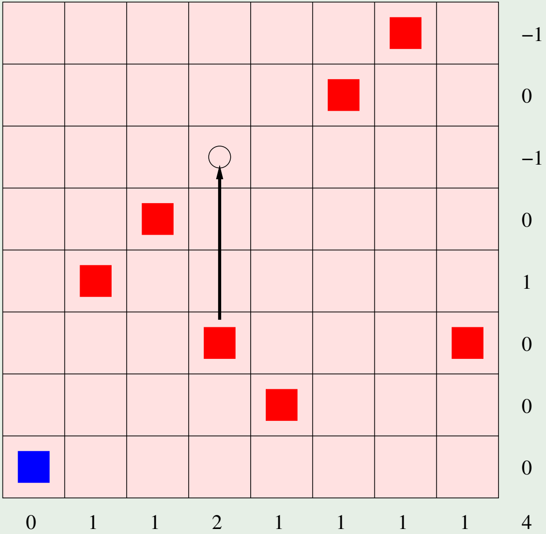
Example (8 Queens: Sample Run)



Example (8 Queens: Sample Run)



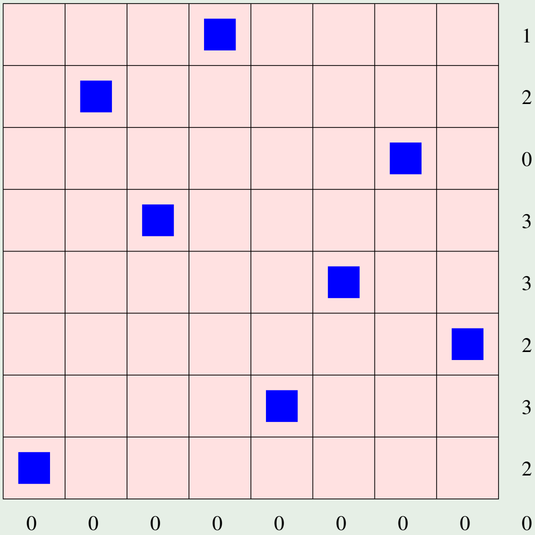
Example (8 Queens: Sample Run)



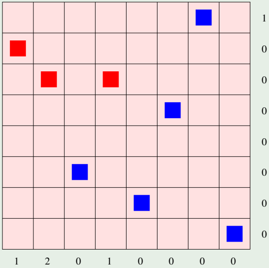
Example (8 Queens: Sample Run)

… and so on, until …

Example (8 Queens: Sample Run)



Example (8 Queens: Local Minimum)



- Queen 2 is selected, as the only most violating queen.
- Queen 2 is placed on one of rows 2 to 8, as the system violation will increase by 1 if she is placed on row 1.
- Queen 2 remains the only most violating queen!
- Queen 2 is selected over and over again.

A meta-heuristic is needed to escape this local minimum.

Outline

Hybridising Systematic and Local Search

For $\langle V, D, C, f \rangle$, and recall the generic algorithm of slide 16:

Example (Large Neighbourhood Search (Shaw, 1998))

$P := \langle V, D, C \rangle$ where all variables have their full domains
$s :=$ First(Solutions(P)) // systematic search
$k := 0; s^* := s$ // s^* is the so far best assignment
while $k < \mu$ **do**
  $k := k + 1$
  $P := \langle V, D, C \cup \{f(V) < f(s^*(V))\}, f \rangle$ but where some variables are frozen (e.g., fixed to their values in s^*) and the other variables are thawed (or: relaxed) (e.g., have their full domains, as per D)
  $s :=$ Best(Solutions(P), _) // limited systematic search
  if s exists **then** $s^* := s$
return s^*

Outline

UPPSALA UNIVERSITET

Reference

- Hoos, Holger H. and Stützle, Thomas. Stochastic Local Search: Foundations & Applications. Elsevier / Morgan Kaufmann, 2004.
- Glover, Fred W. and Laguna, Manuel. Tabu Search. Kluwer Academic Publishers, 1997.
- Michel, Laurent and Van Hentenryck, Pascal. Localizer. *Constraints* 5(1–2):43–84, 2000.
- Van Hentenryck, Pascal and Michel, Laurent. Constraint-Based Local Search. The MIT Press, 2005.
- Shaw, Paul. Using constraint programming and local search methods to solve vehicle routing problems. Proceedings of CP 1998, *Lecture Notes in Computer Science*, volume 1520, pages 417–431, Springer, 1998.