CP Search

Guido Tack

CP=Propagation + Search

- Reminder:
  - **generate and test:**
    use backtracking search to generate assignments to variables, then check constraints
  - **constraint programming:**
    at every search node, **propagate** constraints to remove inconsistent values
- Constraint propagation **prunes** the search tree
- Now: **branching heuristics** determine the **choices** that are made at each node

Branching

- At each node, the solver needs to make a **choice**
- Usual case (in CP):
  - pick a **variable** x that still has at least two values
  - pick a **value** v from the domain of x
  - **branch** with $x=v$ and $x\neq v$
- Heuristic needed: **which** x and which v?
- Most CP solvers use this pattern

Branching and Exploration

- The choices define the **shape** of the tree
- An **exploration strategy** defines the order of exploration:
  - **DFS** depth-first left-to-right
  - BFS (breadth-first), LDS (limited discrepancy), ID (iterative deepening), A* (best-first search), parallel, …
- The two **interact**:
  - if heuristic tries to put solutions towards the left of the tree, we should use left-to-right exploration!

Branching Heuristics

- **Programmable** heuristics:
  - use domain knowledge (application specific search, e.g. scheduling, packing, geometric problems, routing)
  - require insight into problem structure
  - can be a big advantage of CP solvers
- **Universal** heuristics:
  - no user interaction needed ("black box" solver)
  - dominant in SAT, MILP
  - current research topic in CP
- **In between**:
  - commonly used (generic) CP heuristics

First-fail

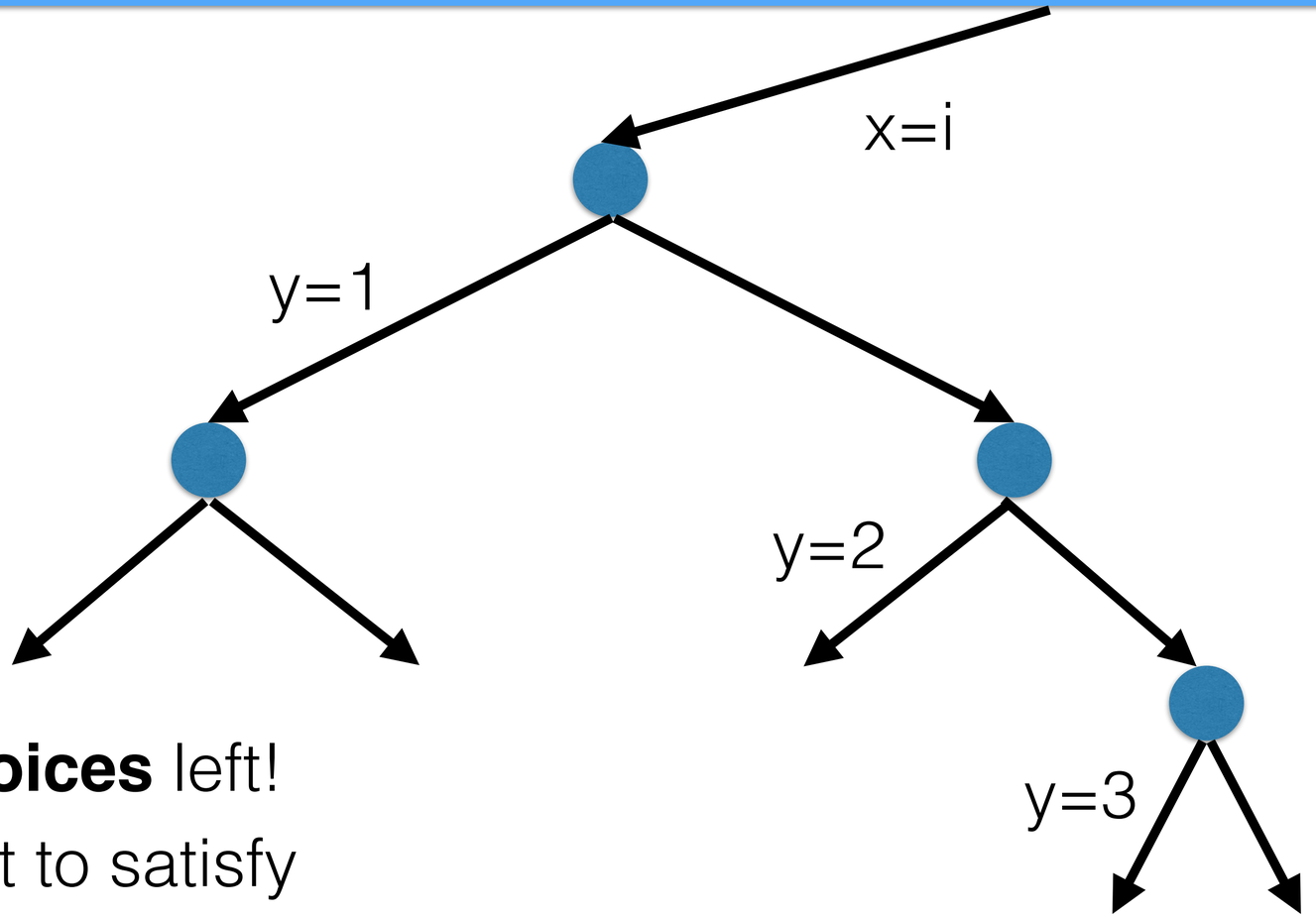
- One successful heuristic is the **first-fail principle:**
  "To succeed, try first where you are most likely to fail"
- At each step choose the variable with the smallest domain
- Do this dynamically based on the domain size after propagation
- **Doesn't sound like a good idea?**

Why first-fail works

variables x,y

x: large domain

y: small domain



assign x first: **few choices** left!

tight variables difficult to satisfy

hard to recover from failure deep in the tree

try to **keep tree shallow**!

First-fail Demo

- Try it yourself:

  n-queens for n=10
  use decomposed `alldifferent`

- search all solutions using

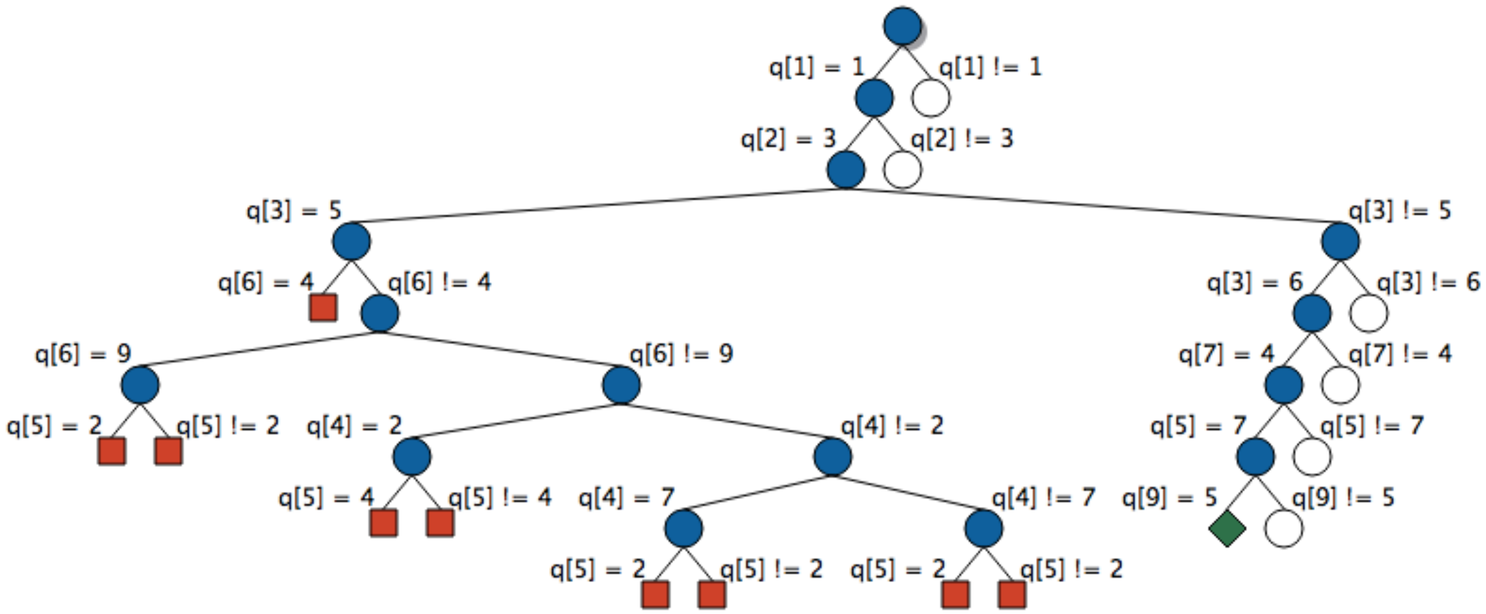
  - first_fail: 5707 choice points (G12 fd)

  - anti_first_fail: 134,443 choice points (G12 fd)

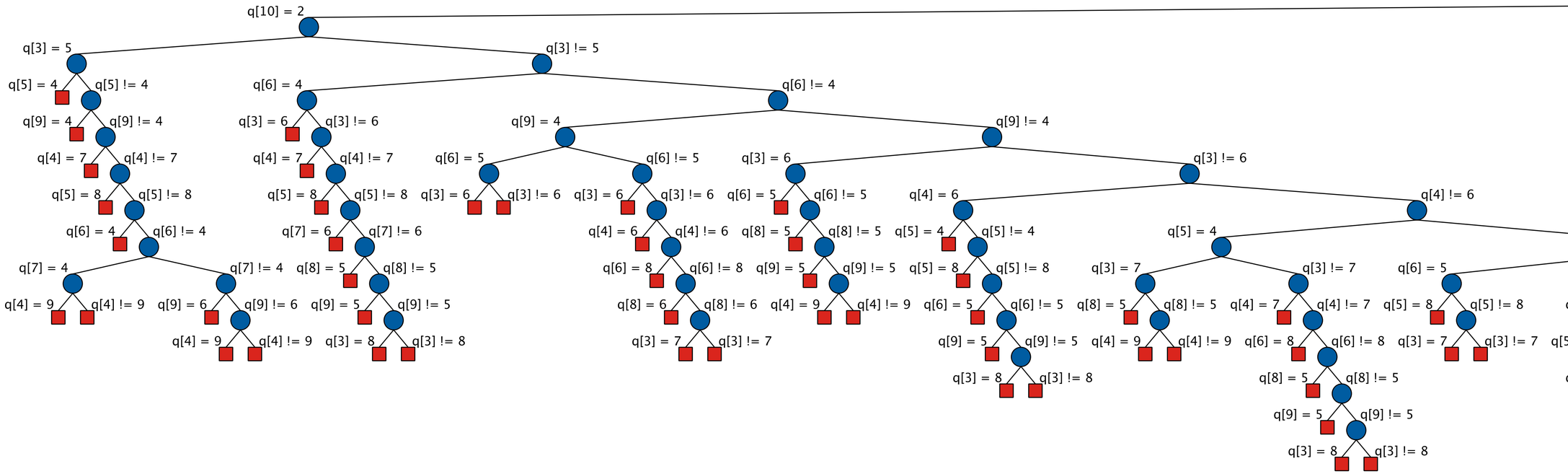
- In Gecode Gist:

  - visualise domains and choices

10 queens first-fail



10 queens anti-first-fail



Adaptations of first-fail

- Pick the **most constrained** variable
  - degree = how many constraints are attached (solver needs this anyway!)
  - order by domain size / degree
- Pick the **variable that was responsible** for most of the recent failures
  - keep track of which variables **cause failure**
  - weighted degree = degree weighted by failure counts
  - similar to standard heuristic in SAT (called VSIDS)

Heuristics in MiniZinc

- Search can be controlled using **annotations** on the solve item
- For integer variables:
  `int_search(vars, var_sel, val_sel, complete)`

  - `vars` is an array of variables

  - `var_sel` is the variable selection strategy
    `(input_order, first_fail, anti_first_fail, smallest, largest, occurrence, most_constrained, max_regret, dom_w_deg)`

  - `val_sel` is the value selection strategy
    `(indomain_min, indomain_max, indomain, indomain_split)`

Value selection

- Choice: x=n, x≠n
  - `indomain_{min,max}`: *n* = {smallest,largest} value in domain
  - `indomain_median`: *n* = median value in domain
  - `indomain_random`: *n* = random value from domain
- Choice: $x=d_1$, $x=d_2$, … $x=d_n$
  - `indomain`: try domain values from lowest to highest
- Choice: x≤d, x>d
  - `indomain_split`: d=(min(D,x) + max(D,x)) / 2
  - `indomain_reverse_split`: swap ≤ and >

Value selection

- Which one works?
  - typically depends on the problem.
  - minimise? perhaps try smaller values first.
    maximise? perhaps try larger values first.
  - splitting can work well with arithmetic constraints.

Complex Search

- MiniZinc supports only one complex search **combinator**: seq_search([s_1,…,s_n])
  - complete search s_i before starting search s_{i+1}
  - e.g. consider two arrays of variables:
    `vehicle[c]` maps customer c to a vehicle
    `arrival[c]` maps customer to arrival time of vehicle
  - possible strategy: first decide vehicle mapping, then arrival times:

```
seq_search([int_search(vehicle,…,…,…),
            int_search(arrival,…,…,…)]);
```

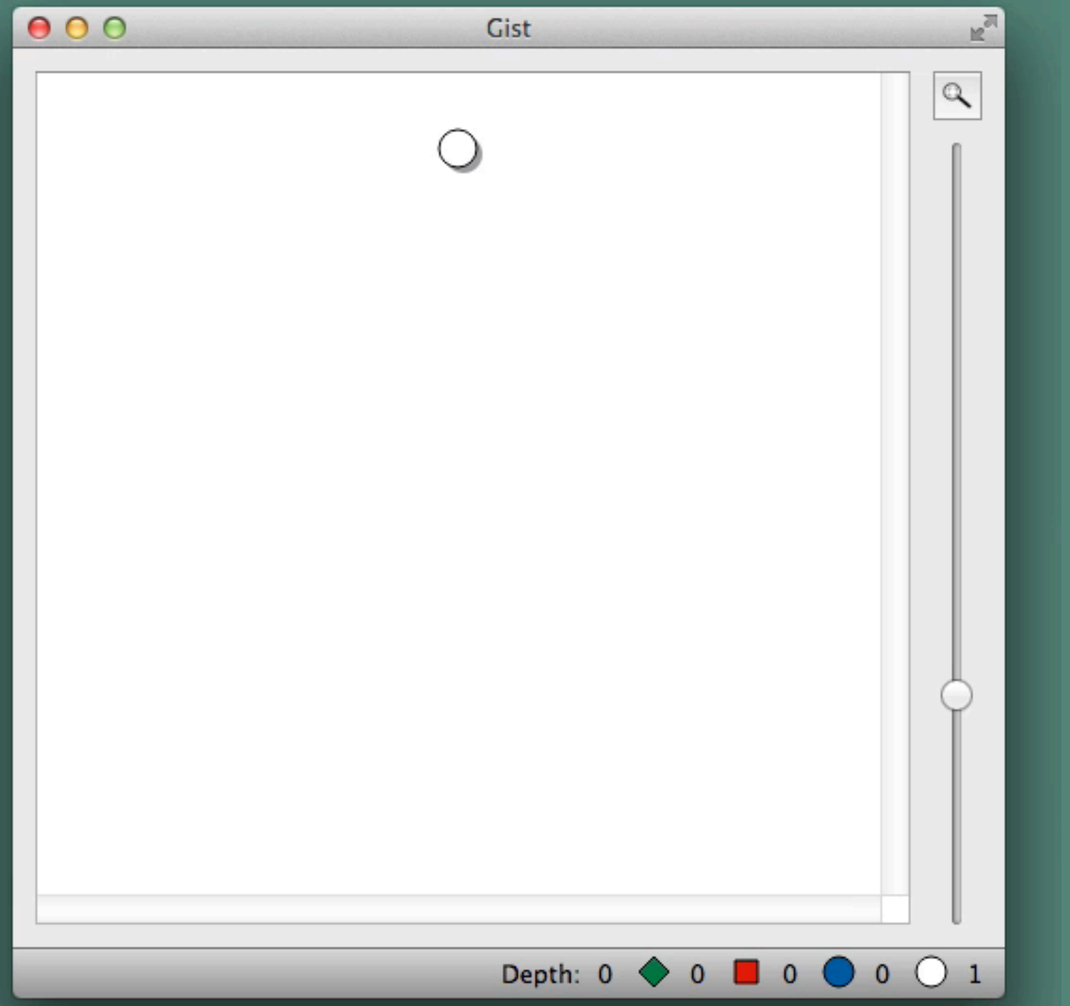
Backtracking Search for Optimisation

Solving Optimisation Problems

- So far: how to find a satisfying solution
- To solve optimisation problems, we use **branch and bound** search:
  - search for a solution
  - compute its objective value
  - post a constraint for the remaining search to only allow solutions with better objective value
  - repeat until no more solutions are found
- The final solution is guaranteed to be optimal

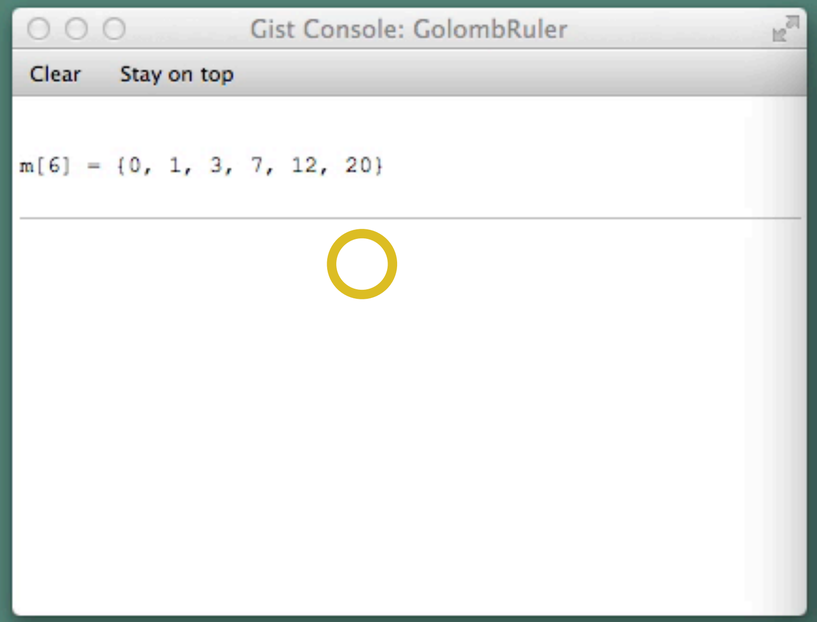
Example: golomb ruler

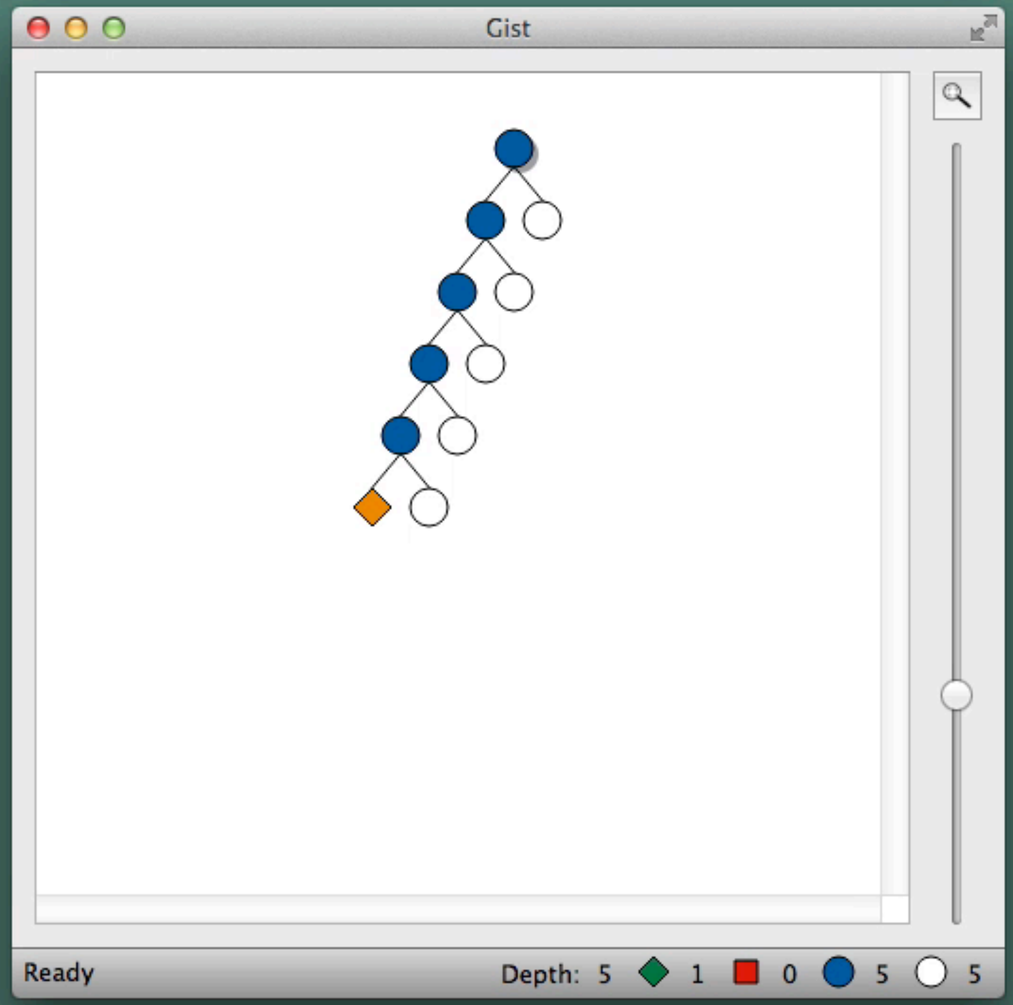




Add constraint:
objective < 20

Example: golomb ruler



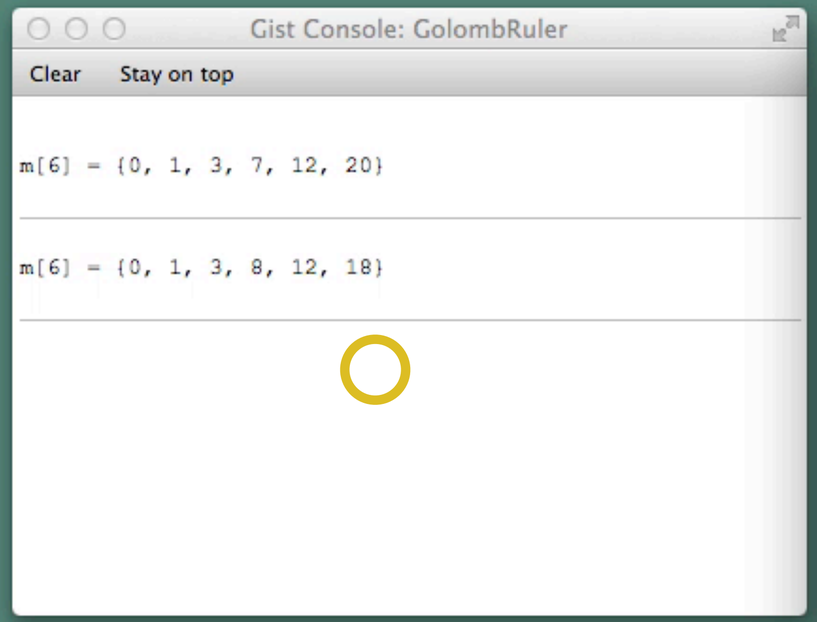


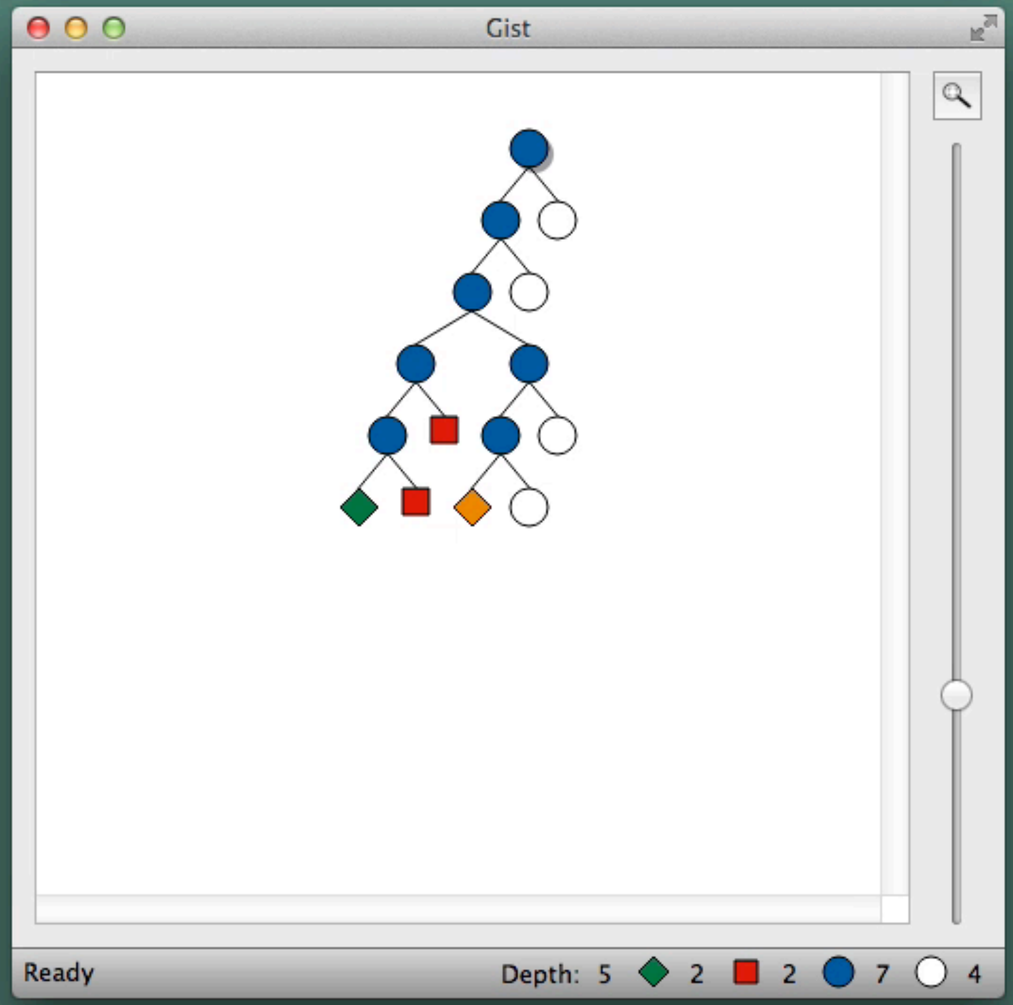
Add constraint:

objective < 20

objective < 18

Example: golomb ruler

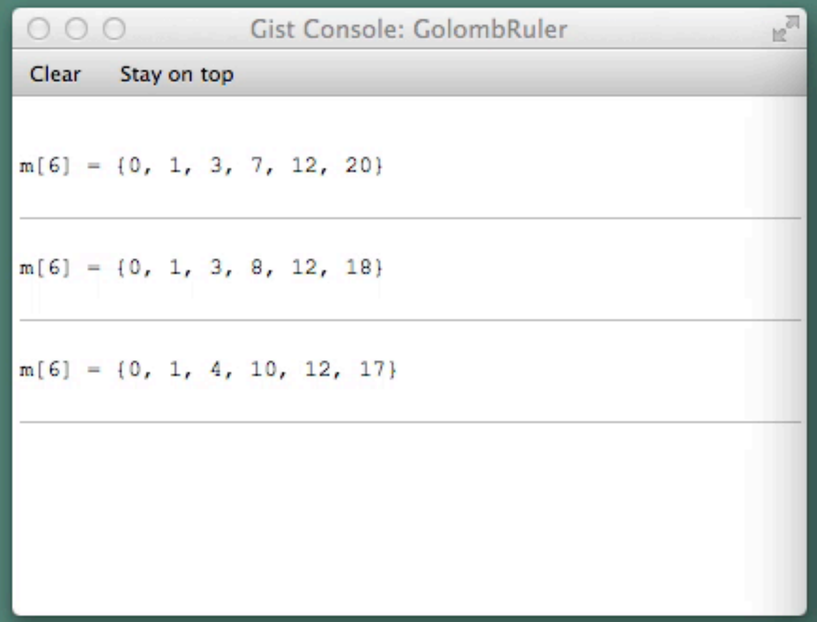


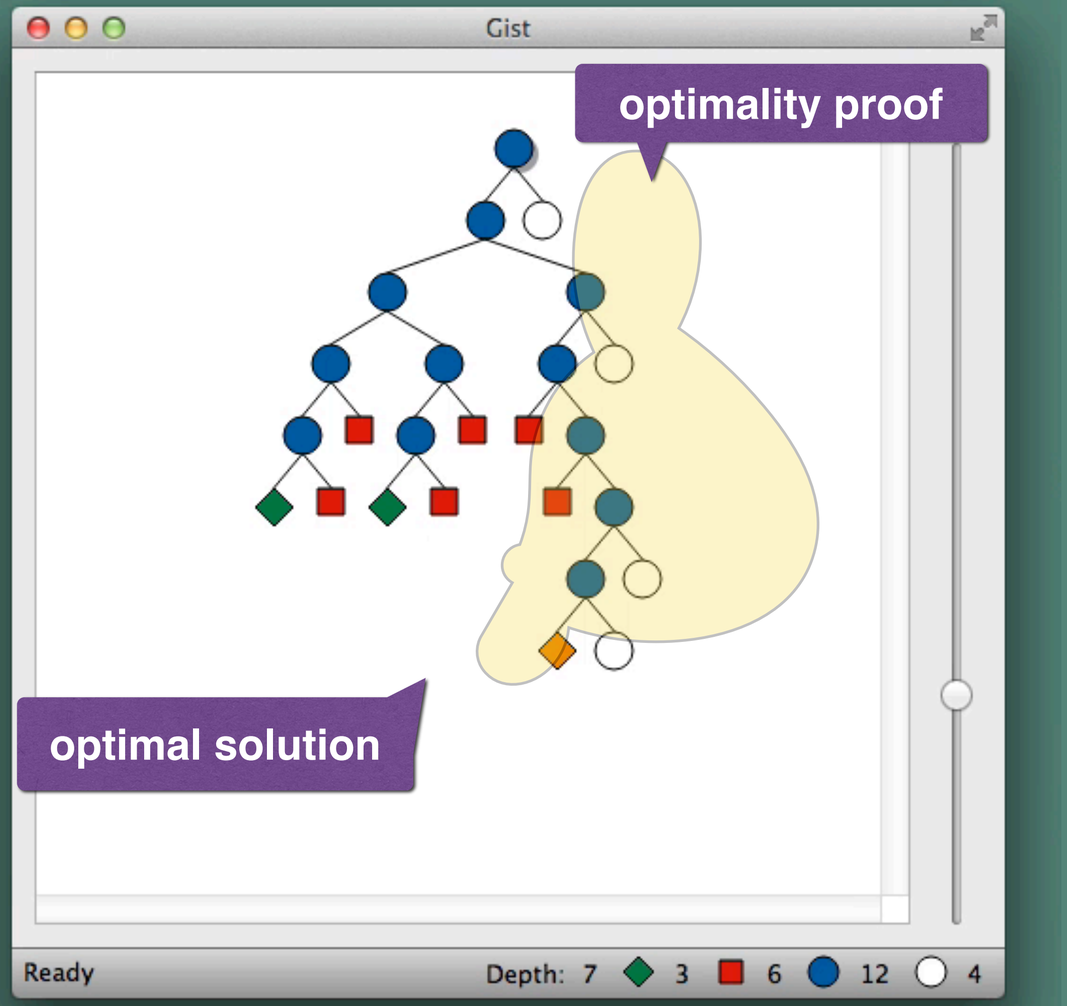


Add constraint:

objective < 20
objective < 18
objective < 17

Example: golomb ruler





Add constraint:

objective < 20
objective < 18
objective < 17

Restarts and Heavy-tailed behaviour

When search is random

- Let's assume our solver is **randomised** (e.g. picks a random value during search) and here's how it behaves:

75% probability:	finishes in	100 nodes
20% probability:	finishes in	10,000 nodes
5% probability:	finishes in	1,000,000 nodes

- Q: What is the **expected** number of nodes it needs?
- Now let's say **we give up after *c* nodes and restart**.

  Q: What is the **expected** number of nodes for c=50, c=100, c=10,000?

Is restarting worth it?

$$\mathbb{E}[\text{total}] = c \times \frac{1-p}{p} + \mathbb{E}[X \mid X \leq c], \qquad p = P(X \leq c)$$

c	p	expected nodes	
∞	1	75+2000+50000=52,075	no restarts
50	0	∞	never finishes in 50 nodes
100	0.75	100/3 + 100 =133.333	391x better
10,000	0.95	10000/19+2184=2,710.316	19x better

Heavy-tailed behaviour

- From a real experiment:

giving up after c=	1,000	10,000	100,000	1,000,000	10,000,000
avg. cost=	304	1,745	13,343	111,876	949,201

- **If you're prepared to wait longer, it is only going to take longer on average.**
- **The mean doesn't exist!**
- The distribution is heavy-tailed.
- Restarting can fix this problem!

Restart strategies

- Policy for when to restart
  - Constant restart – after using L resources
  - Geometric restart
    - restart after using L resources, with new limit α L
  - Luby restart
    - 1,1,2,1,1,2,4,1,1,2,1,1,2,4,8, …
    - "universally optimal" for randomised algorithms:
      - no worse than a log factor slower than optimal policy
      - not bettered by more than a constant factor by other universal policies

Restarts

- Restarts are ubiquitous in default search strategies
- Combined with dynamic variable selection strategies they have another advantage
  - A bad choice at the top requires exponential search to undo
  - Restarts avoid this, by throwing away the choice.

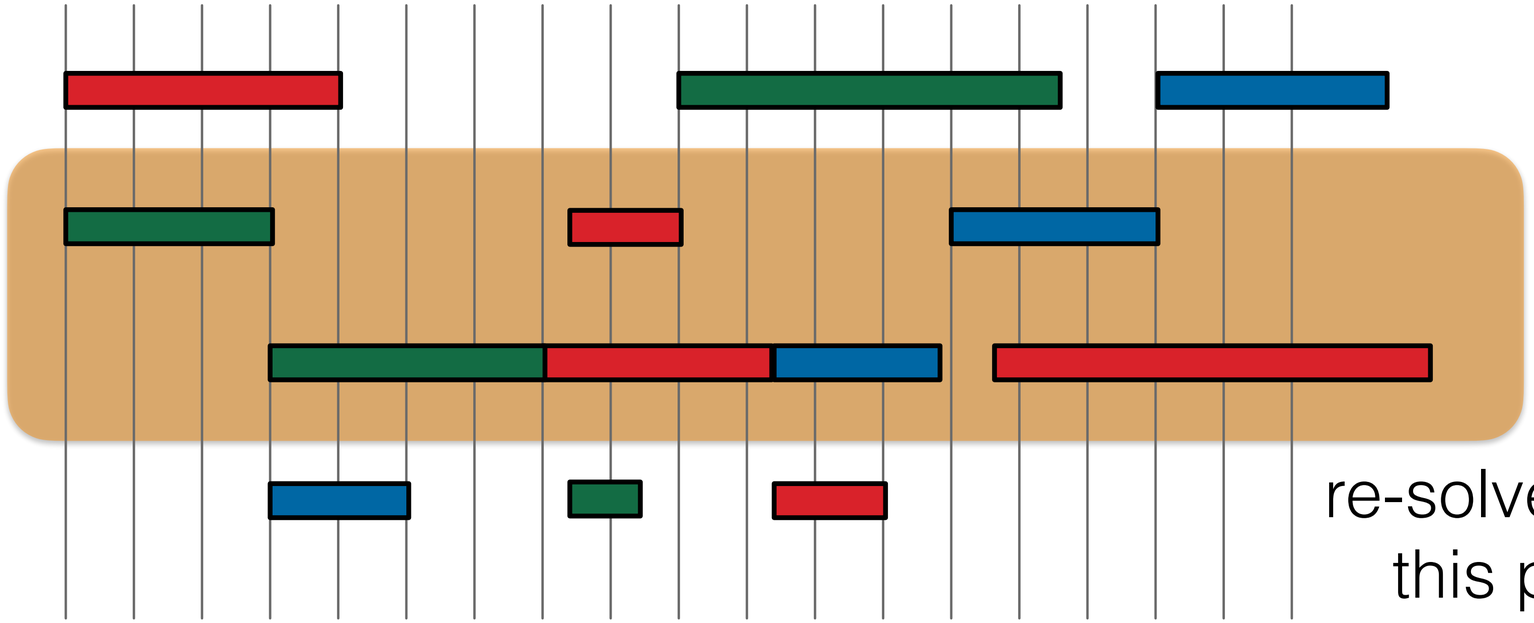
Large Neighbourhood Search

Large Neighbourhood Search

- Many real-world problems are too big to solve with CP
- But: we can find reasonable solutions in reasonable time
- Idea: use CP to ***improve*** solutions, but give up on global optimality
- Very influential paper:
  Paul Shaw: *Using constraint programming and local search methods to solve vehicle routing problems*. CP ’98

Large Neighbourhood Search

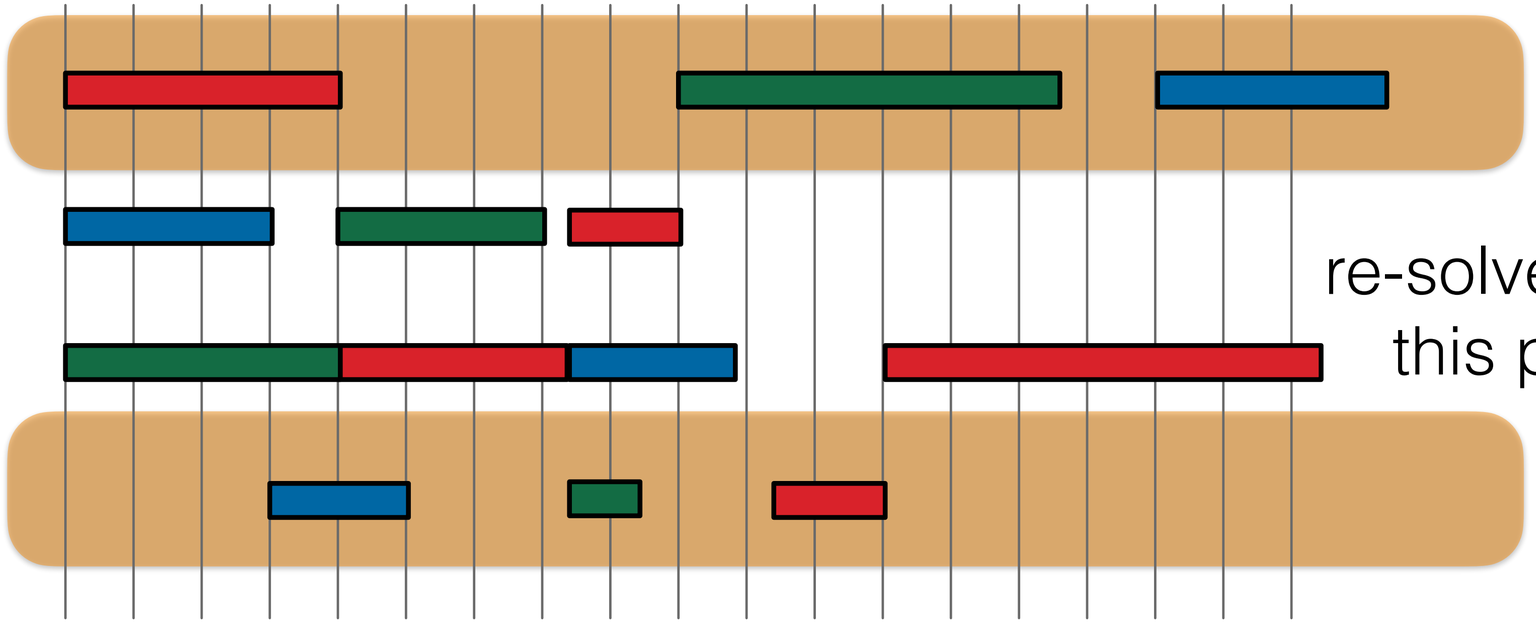
Initial solution:



re-solve only this part

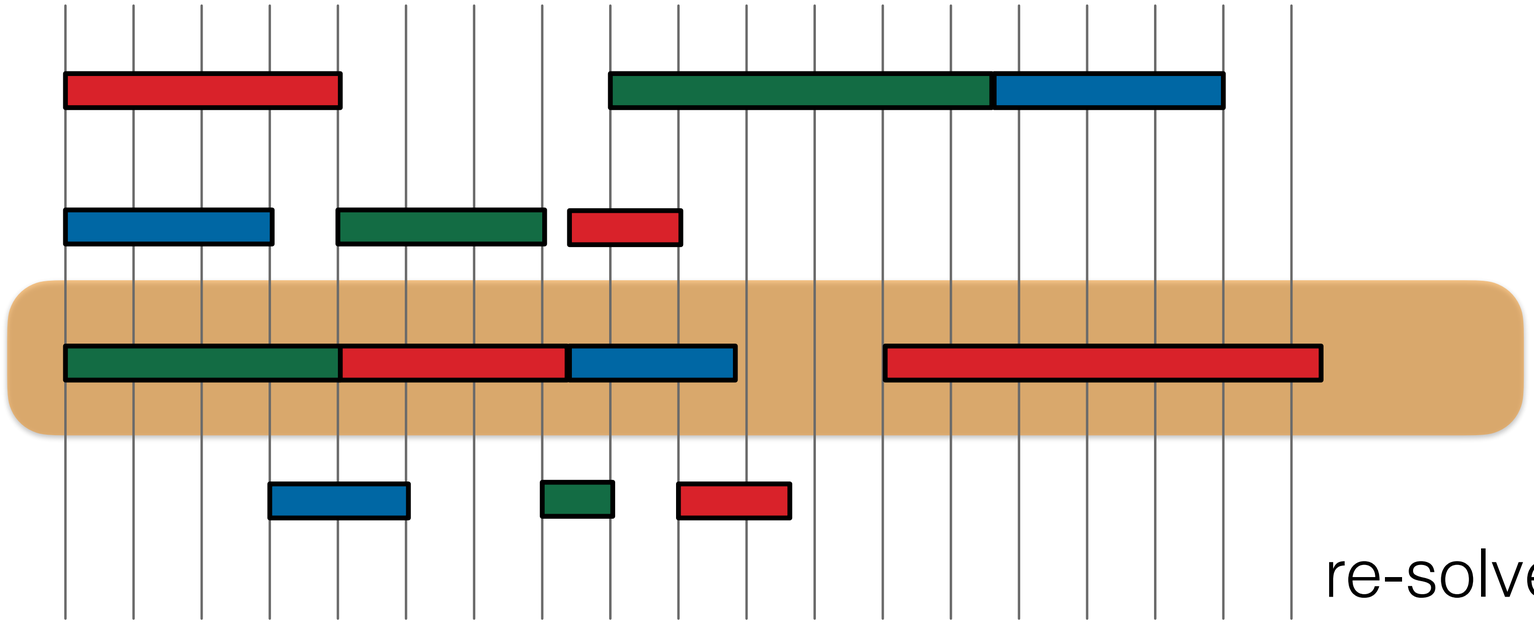
Large Neighbourhood Search

Next solution:



Large Neighbourhood Search

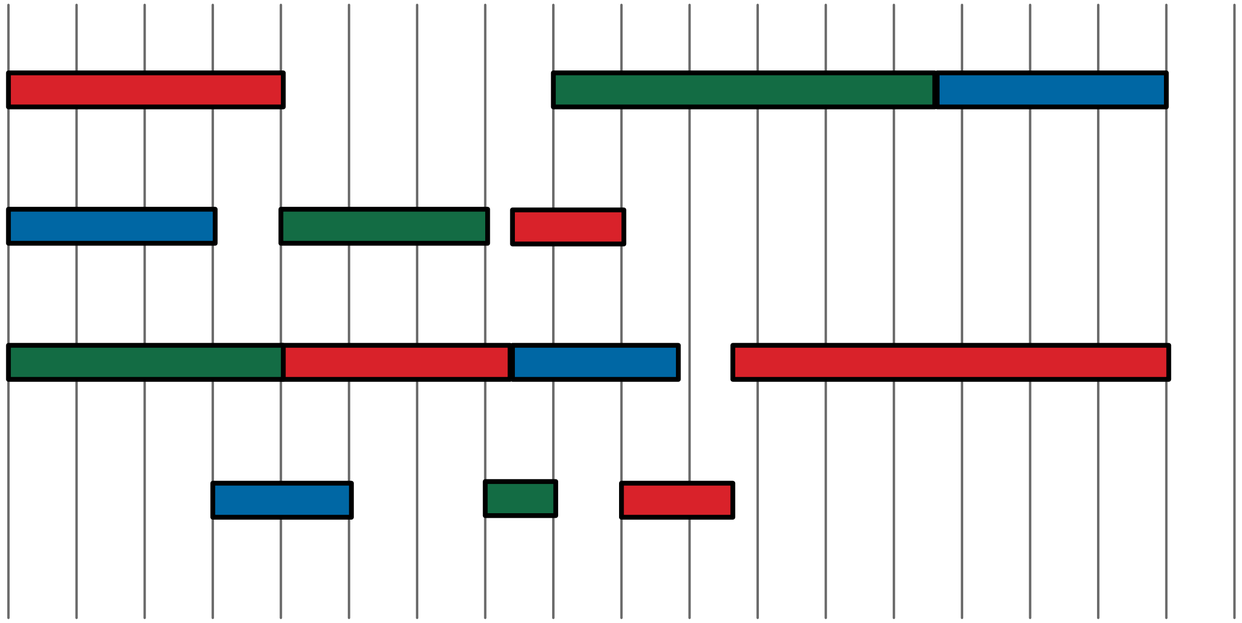
Next solution:



re-solve only this part

Large Neighbourhood Search

Next solution:



Repeat until solution quality is sufficient.

Large Neighbourhood Search

- Basic principle:

  - find initial solution (using CP, or any other method)

  - **destroy** parts of it (reset variables to full domains)

  - **repair** by re-solving

- Control overall search using a **meta heuristic** (e.g. hill climbing or simulated annealing)

- Has proven very successful for hard problems (e.g. vehicle routing, scheduling, rostering)

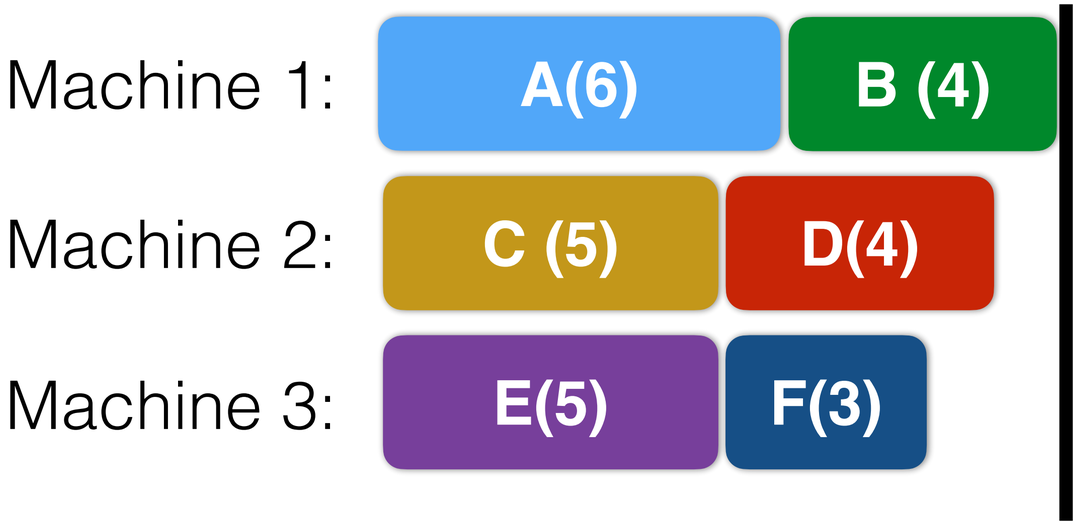
Bad neighbourhoods

This is our solution to a scheduling problem.

What if we destroy:

- {D,F}?
- {A,B}?

What do we have to destroy to improve the solution?



Total: 10 time units

How to destroy

- Simple heuristic:

  pick a random set of variables, fix them to their previous values

- But we can come up with problem-specific heuristics

Problem-specific LNS: VRP

- Let's assume we are solving a vehicle routing problem:

  Assign customers to trucks, and assign a route for each truck so that it visits its customers and returns to the depot. Each customer must be visited inside a given time window, and each truck has a fixed capacity.

- **Q:** come up with different heuristics: which decisions should we fix, and which ones should we re-solve?

Problem-specific LNS: Scheduling

- Let's assume we are solving a scheduling problem:

  We have a set of tasks that need to be scheduled on a set of machines. Each machine has a capacity (how many tasks it can do in parallel). Each task has a list of machines it needs to go through, a deadline for production, and which other tasks need to be finished before this one.

- **Q:** come up with different heuristics: which decisions should we fix, and which ones should we re-solve?

LNS with MiniZinc/Gecode

We can try out a very simple version of LNS in MiniZinc/Gecode:

- add search annotation

```
::relax_and_reconstruct(x, 85)
```

- With 85% probability, keep each variable in array x for each restart
- Combine with restarts to enable the search:

```
::restart_constant(3000)
::restart_geometric(1.5, 200)
```

restart after 3000 failures
restart with geometric series

Limitations

Limitations

CP works well for many types of problems, but there are limits:

- Very large number of variables and constraints
- Continuous variables
- Problems with strong LP relaxations
- Problems where finding solutions is easy (and there are very many) but finding *good* solutions is hard