An overview of

OscaR

OPERATIONAL RESEARCH IN SCALA

https://bitbucket.org/oscarlib/oscar

Optimization projects are increasingly complex!

OscaR = Scala in OR

- Optimization is more and more complex
- What you need is this for optimization:

Motivation

Modeling languages for optimization are

- Not open-source (Comet,AMPL,AIMS,OPL, etc)
- Dedicated languages
- Inefficient for something else than modeling

Looking for a language

- Flexible
- Mainstream
- Efficient

Good Candidates

- Not so flexible for DSL implementation (no continuations for instance)
- Dynamic
- Have to implement in C every bottleneck code

- Very flexible for implement DSL
- Strongly typed
- Good compromise in terms of speed (running on JVM, fully interoperable with Java and efficient)

A taste of Scala

```
scala> Array(1,6,9,3,2).filter(_%2 == 0).sum
res: Int = 8

scala> (1 to 8).map(_*2)
res: Vector(2, 4, 6, 8, 10, 12, 14, 16)

scala> class Person(name: String, age: Int)
scala> val p = new Person("Laurence",60)

scala> for (i <- 1 to 5) yield i+3
res: Vector(4, 5, 6, 7, 8)
```

OscaR is a tool to

- quickly make hybridizations (CP,LS,MIP…)
- try new ideas in a flexible way and visualize it
- open for extensions

Contributors (code, examples, suggestions):

Bertrand Cornelusse (n-side)
Cyrille Dejemeppe (ucl + n-Side internship)
Pierre-Yves Gousenbourger (ucl + n-Side internship)
Renaud De Landtsheer (cbls, cetic)
Renaud Hartert (ucl + n-Side internship)
Håkan Kjellerstrand (cp blog)
Hrayr Kostanyan (ulb + n-Side internship)
Gilles Meyer (n-side)
Sébastien Mouthuy (n-side)
Christophe Ponsard (cbls, cetic)
Pierre Schaus (cp blog, UCL)
Gilles Scouvart (n-side)

Tools currently available

Visualization

Derivative Free Optimization

Constraint Programming

Linear Programming

Constrained Based Local Search

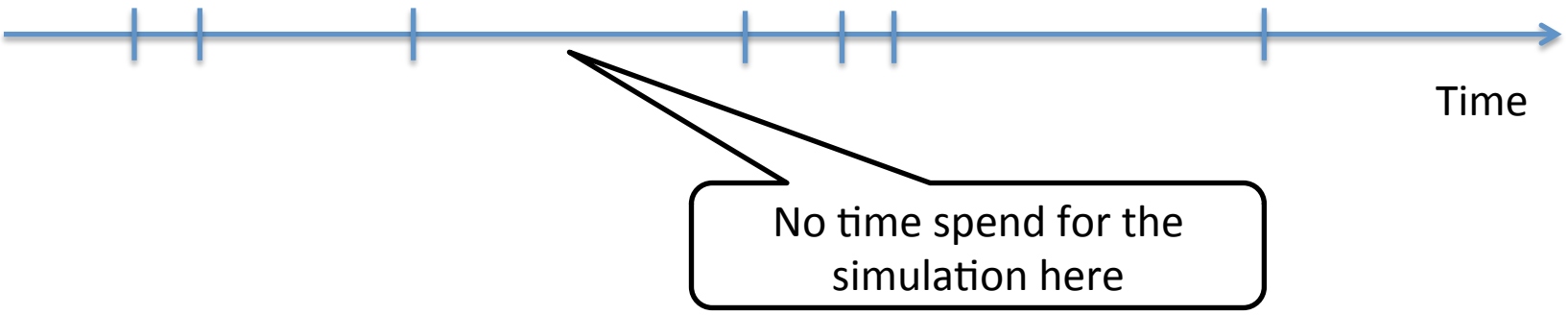
Discrete Event Simulation

Discrete Event Simulation

- Two machines can be broken, there is only one repair person that can fix it at a time,
- One of the machines must wait if the two machines are broken at the same time

Discrete Event Simulation

- The system is not continuously changing
- State changes only when an event occurs
- Don't want to simulate every time steps
  - complexity should be O(n), n = number of changes in the system*



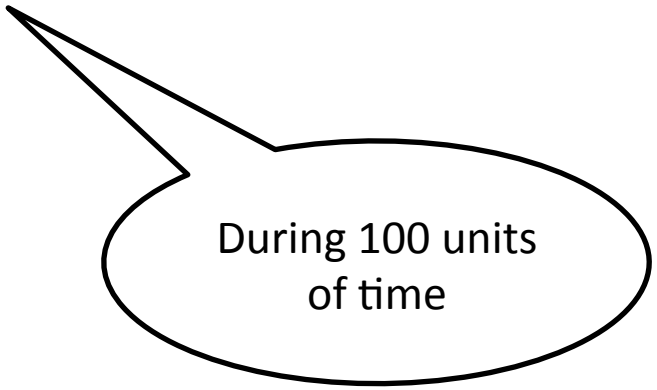
* could be slightly more because we use a heap to store pending events

Let's write the story ...

```
class Machine(m : Model, name: String, repairPerson: UnaryResource)
                                        extends Process(m,name) {

  override def start( = alive()
  def alive() = {
      println(name+" is alive")
      m.wait(liveDur.nextInt(10))
      broken()
   }
   def broken() = {
      println(name+" is broken waiting to be repaired")
      m.request(repairPerson)
      repair()
   }
   def repair() = {
      println(name+" being repaired")
      m.wait(repairDur.nextInt(3))
      m.release(repairPerson)
      alive()
    }
}
```

… and ask OscaR to tell it

```
object Machine extends App {
  val m = new Model()
  val repairPerson = new UnaryResource(m)
  new Machine(m,"machine1",repairPerson)
  new Machine(m,"machine2",repairPerson)
  m.simulate(100,true)
}
```



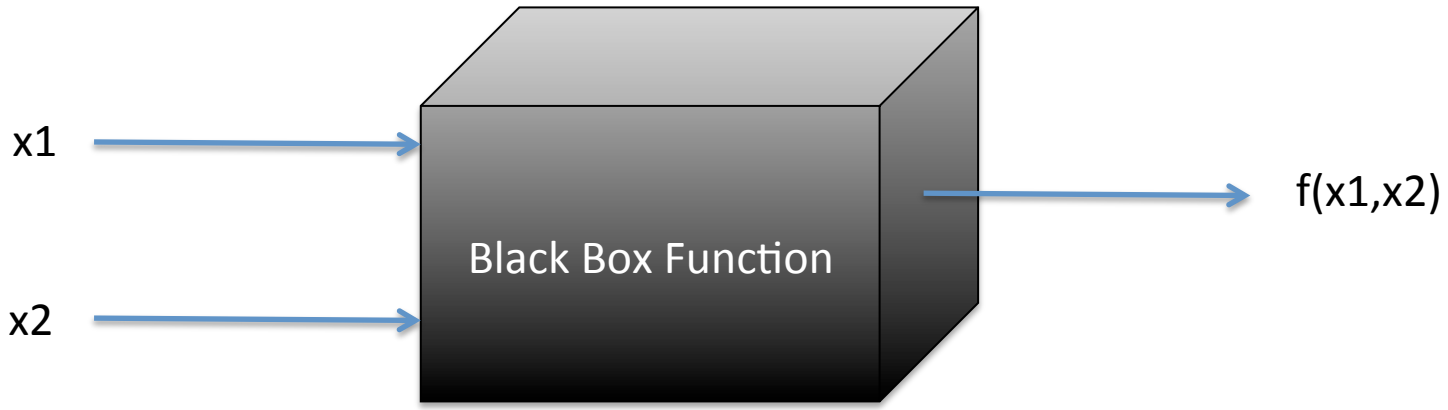
Optimization with OscaR

```
import oscar.X.modeling._
Object MyModel extends App {
  val s = XSolver() // Solver Object
  val y = XVar(...) // Variable Creation
  s.minimize(y) subjectTo {
    // some constraints
  }
}
```

X can be CP,MIP,LP,DFO

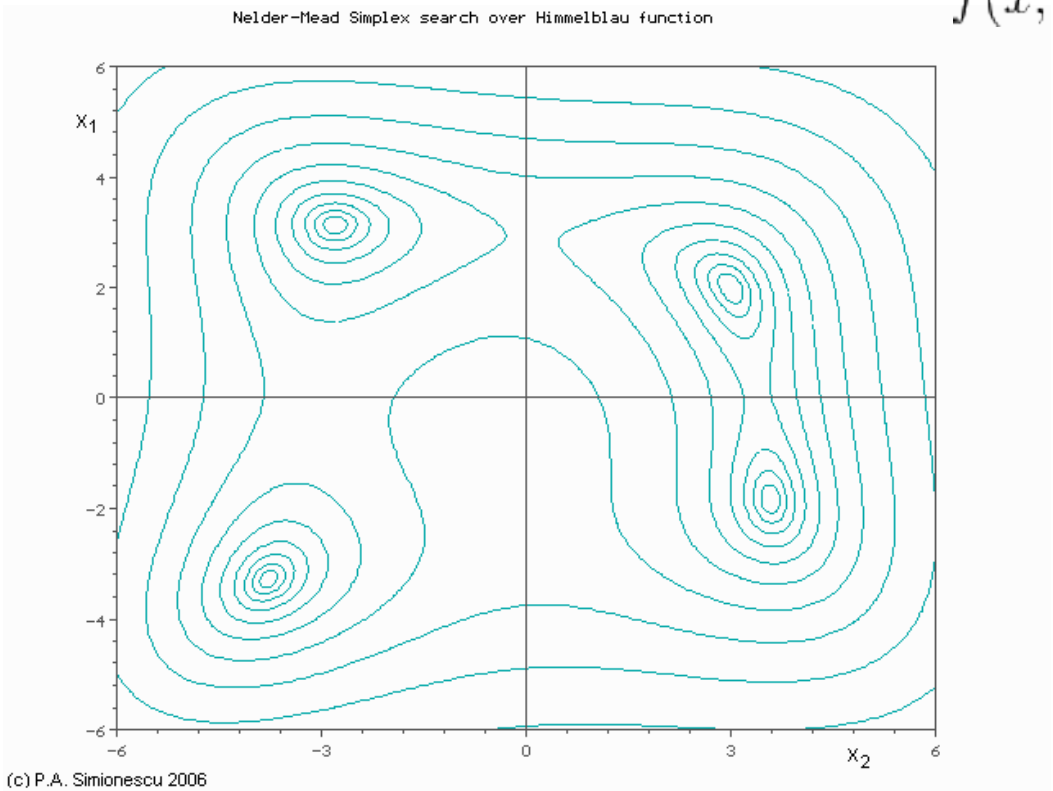
Derivative Free Optimization

- Sometimes you don't have the gradient info

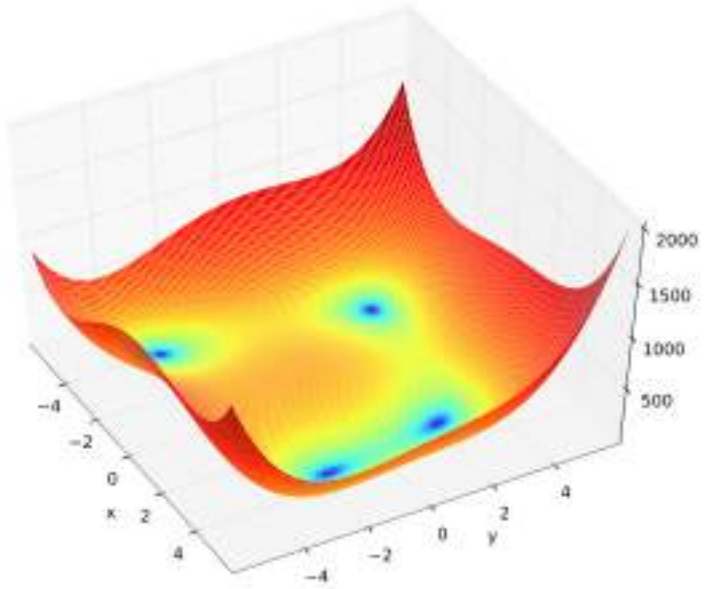


Derivative Free Optimization

- Comparative Method: use only f(x) < f(y)

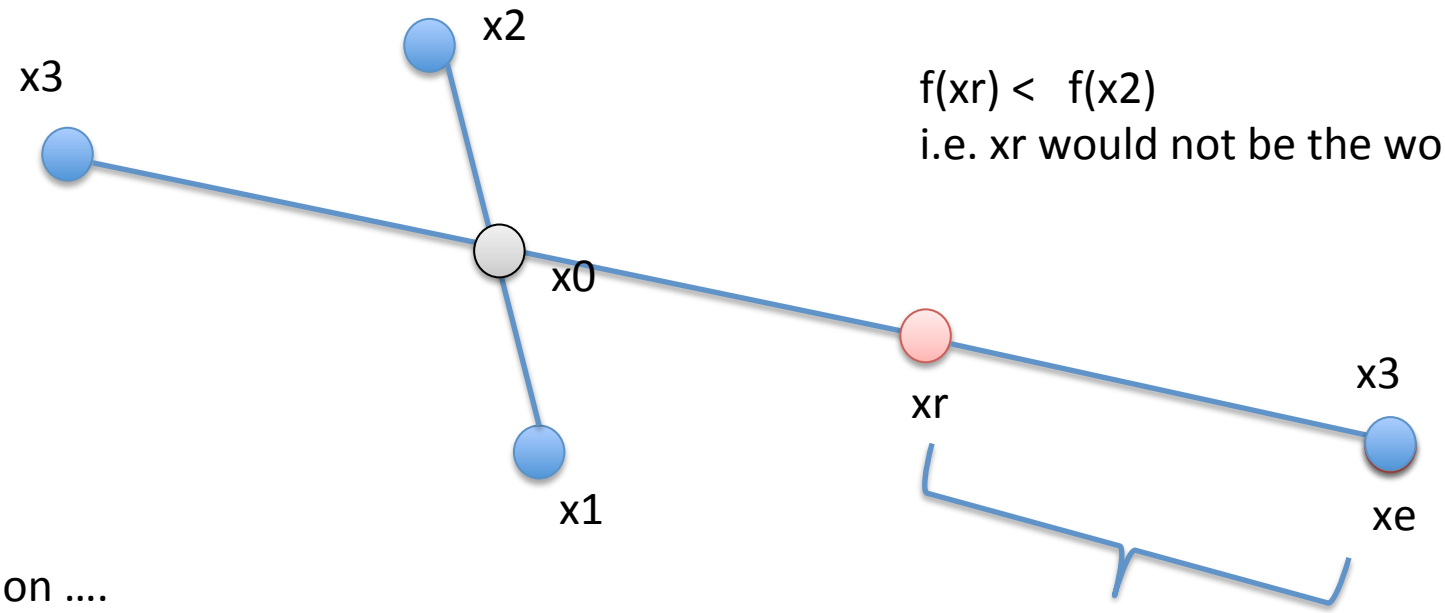


$$f(x,y) = (x^2 + y - 11)^2 + (x + y^2 - 7)^2.$$



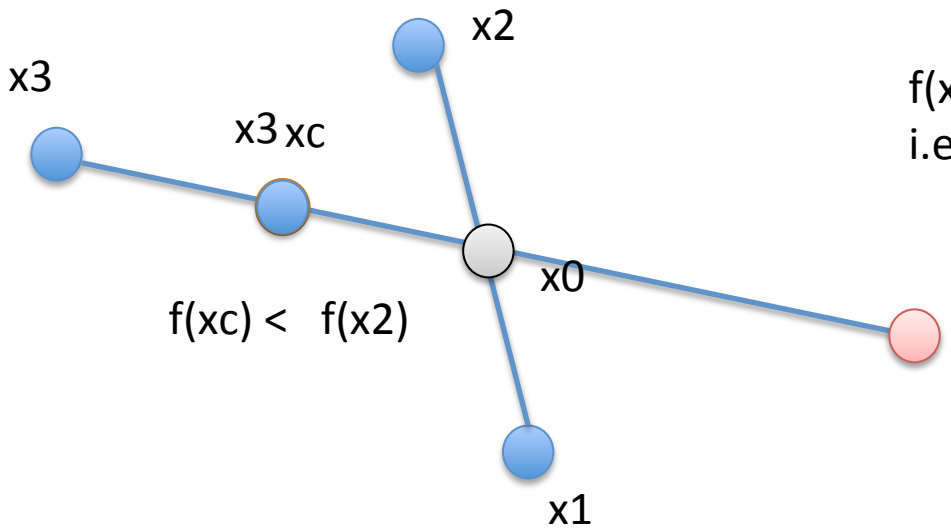
Nelder-Mead (simplex algo 2D)

- Chose and evaluate 3 points in space (triangle) index them such that f(x1) <= f(x2) <= f(x3)



Nelder-Mead (simplex algo 2D)

- f(x1) <= f(x2) <= f(x3)

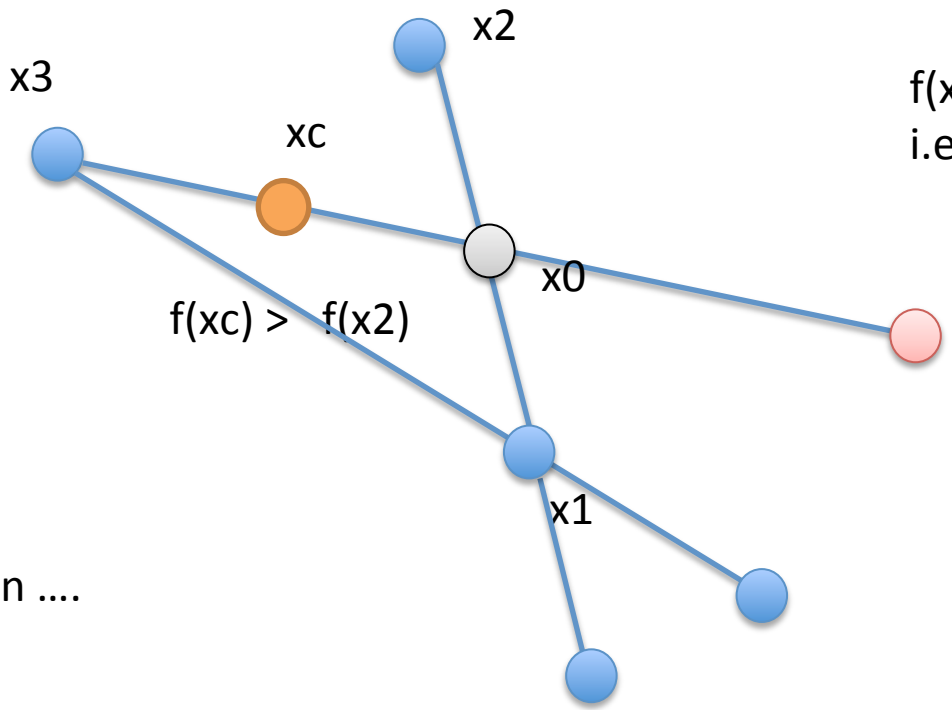


f(xr) > f(x2)
i.e. xr would be the worse

Contraction ….

Nelder-Mead (simplex algo 2D)

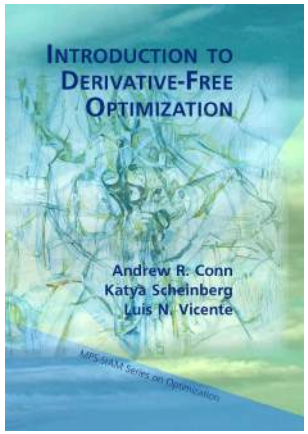
- f(x1) <= f(x2) <= f(x3)



f(xr) > f(x2)
i.e. xr would be the worse

Reflexion ….

Derivative Free Optimization



DDS, MDS, …

```
val dfo = DFOSolver(DFOAlgo.NelderMead)
val x = DFOVar(dfo, "x1", -4, +4)
val y = DFOVar(dfo, "x2", -4, +4)
val objective = (x*x + y - 11) * (x*x + y - 11) +
                (x + y*y - 7) * (x + y*y – 7)
dfo.minimize(objective)
```

Other Features of DFO package

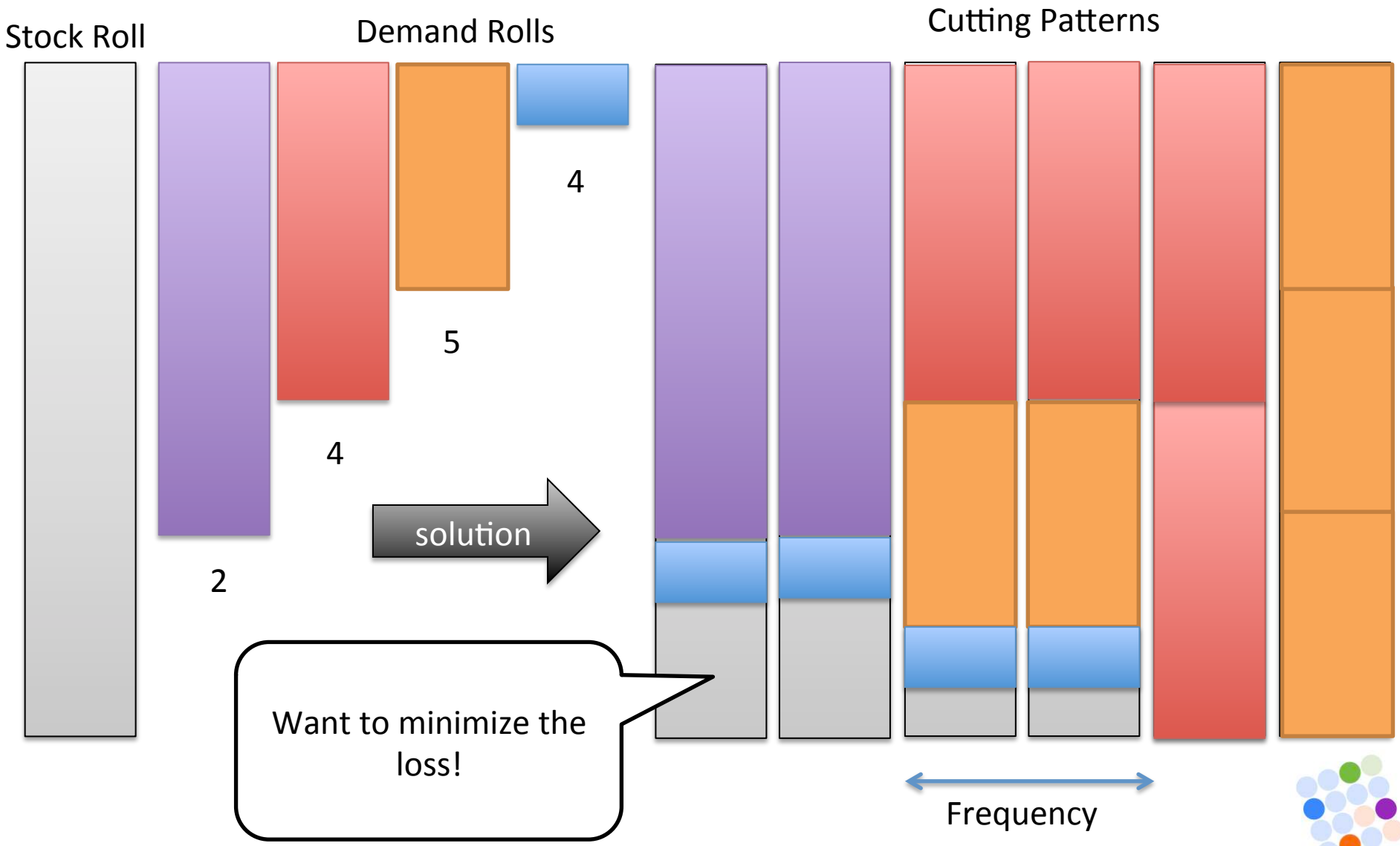
- No need to have an analytical function:
  - Just implement a comparison method:
    def compare(x1: Array[Double],x2:Array[Double])
- Precision for stopping criteria can be chosen
- Starting Point can also be chosen
- Pseudo Random Restarts
- Callbacks to print solution along optimization and stop it if necessary
- Multi-Objective DFO optimization

Linear Programing

```
val mip = MIPSolver(LPSolverLib.lp_solve)

val x0 = MIPVar(mip,"x0",0.0,40.0)   // continuous var
val x1 = MIPVar(mip,"x1",0 to 1000) // discrete var
val x2 = MIPVar(mip,"x2",0 until 18)
val x3 = MIPVar(mip,"x3",2.0,3.0)

mip.maximize(x0+2*x1+3*x2+x3) subjectTo {
      mip.add(-1*x0 + x1 + x2 + 10*x3 <= 20)
      mip.add(x0 - 3.0*x1 + x2 <= 30)
      mip.add(x1 - 3.5*x3 == 0 )
}
```

Cplex, gurobi, glpk

Cutting Stock



Imagine you have generated every possible patterns …

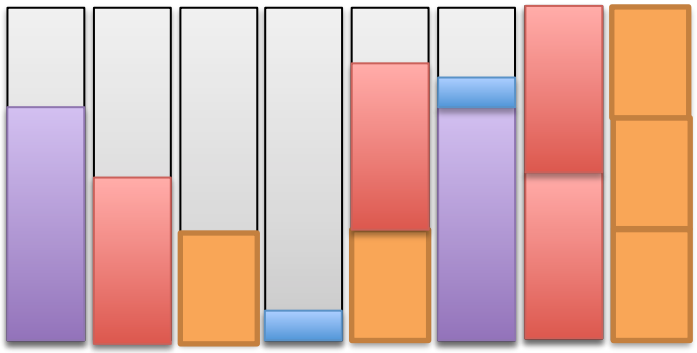
X_p = number of times pattern p is selected in solution

O_{pr} = number of roll type r in pattern p

D_r = demand of roll type r

$$\min \sum_p X_p$$

$$\sum_p O_{pr} \cdot X_p \geq D_r, \forall r$$



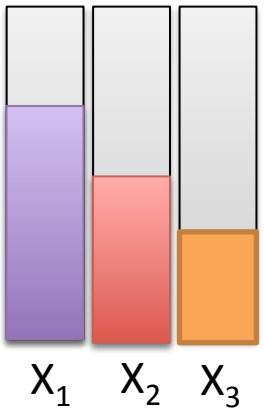
…

Too many columns!

Solution: Lazy generation of patterns

aka: Column Generation

1. Start with a limited number of patterns



$$\min \sum_p X_p$$

$$\sum_p O_{pr} \cdot X_p \geq D_r, \forall r$$

Dynamic intro of new variable, in the objective and in every constraints

2. Solve the master problem
3. Generate a new column with negative reduced cost, if not possible stop

Adding a new column in OscaR

1,0,0,2

Xnew =lp.addColumn(1,constraints,newPattern)

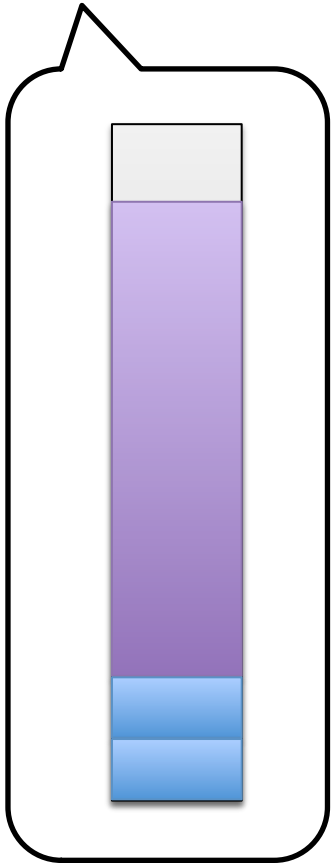
$$min\left(X_{new} + \sum_p X_p\right)$$

$$1 \cdot X_{new} + \sum_p O_{p1} \cdot X_p \geq D_1$$

$$0 \cdot X_{new} + \sum_p O_{p2} \cdot X_p \geq D_2$$

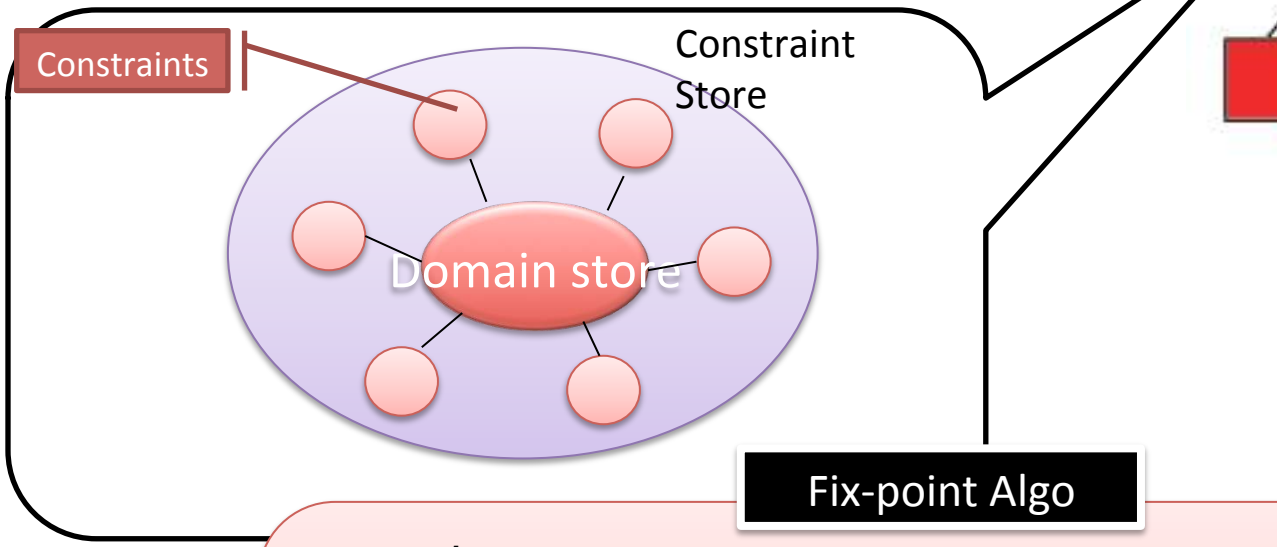
$$0 \cdot X_{new} + \sum_p O_{p3} \cdot X_p \geq D_3$$

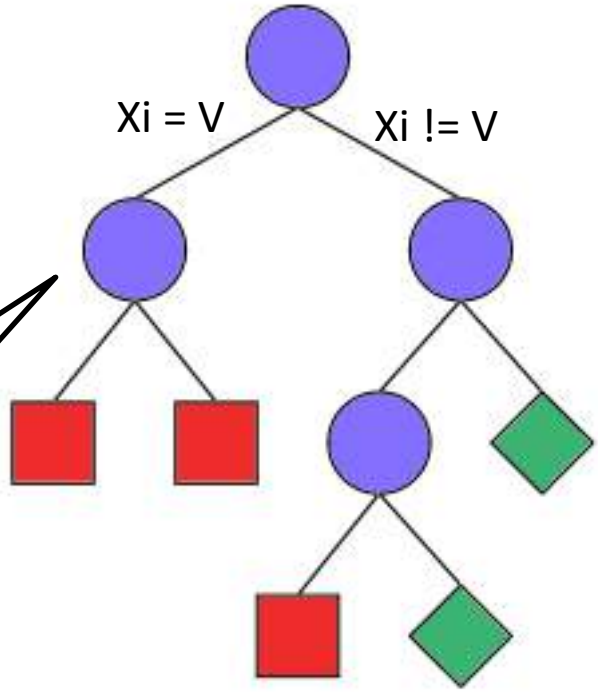
$$2 \cdot X_{new} + \sum_p O_{p4} \cdot X_p \geq D_4$$



CP in One Slide

- CP = Pruning + Search



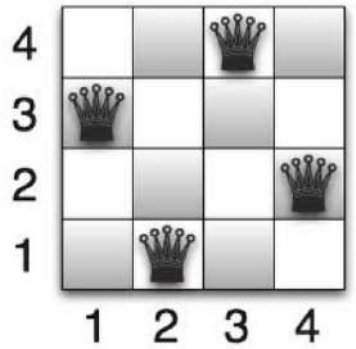


Fix-point Algo

```
repeat
    select a constraint c
    if c is infeasible wrt the domain store
        return infeasible
    else
        apply pruning algorithm of c
until no value can be removed
```

n-Queens (= CP Hello World)

```
val cp = CPSolver()
val n = 8 //number of queens
val Queens = 0 until n
val queens = for(i <- Queens) yield CPVarInt(cp,1 to n) //variables
var nbsol = 0
cp.solveAll subjectTo {
   cp.add(alldifferent(queens))
   cp.add(alldifferent(for(i <- Queens) yield queens(i) + i))
   cp.add(alldifferent(for(i <- Queens) yield queens(i) – i))
} exploration {
   cp.binaryFirstFail(cp)
    nbsol += 1
}
```



CP Search is

- Very important to get quickly good solutions
- Often Rebuttal/Difficult for new CP modeler (recursive thinking, …)

Non Deterministic Search

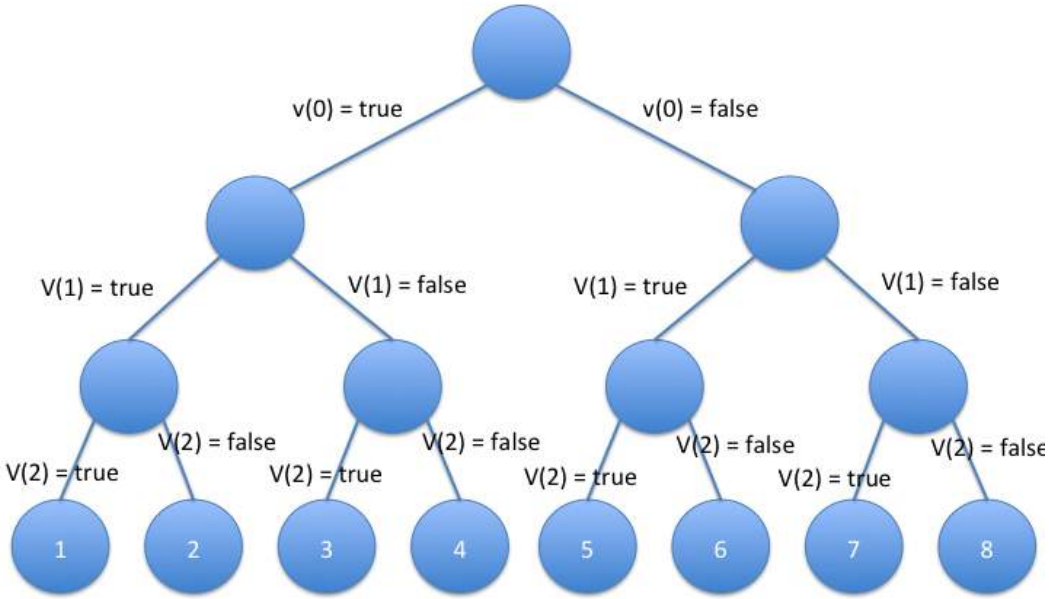
A declarative (magic) expression of search trees:

- How it looks like
- Not it's exploration order

Nondeterministic control for hybrid search, 2006
(P. Van Hentenryck, L. Michel)

Reversible Boolean, recover it's value on backtracking

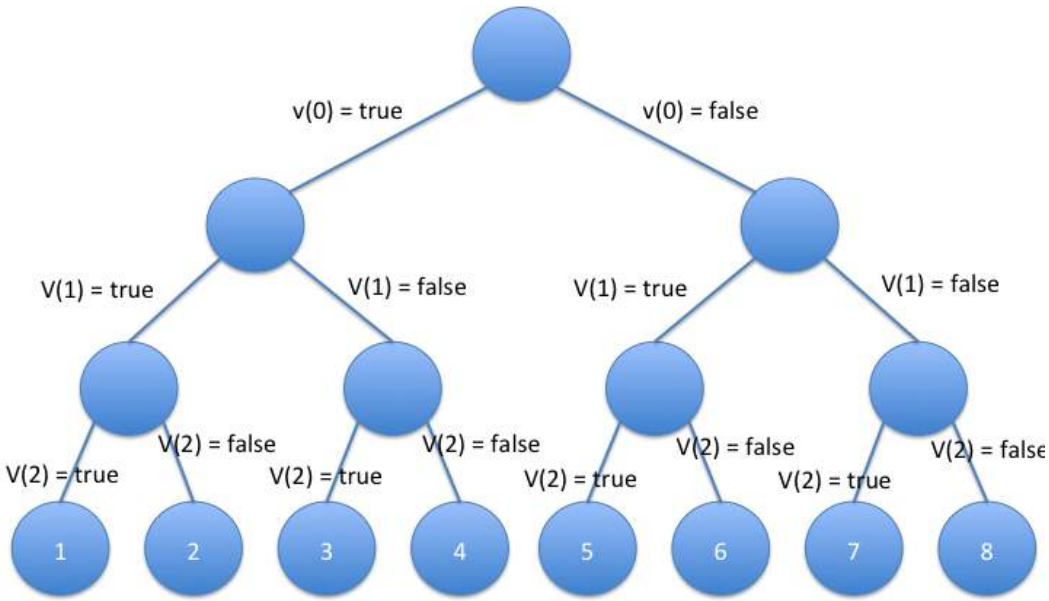
```
val cp = CPSolver()
val v = Array.tabulate(3)(i => new ReversibleBool(cp))

cp.exploration {
  cp.branch { v(0).value = true }
            { v(0).value = false}
  cp.branch { v(1).value = true }
            { v(1).value = false}
  cp.branch { v(2).value = true }
            { v(2).value = false}
  println(v.mkString(","))
}
```



DFS exploration (default)

Reversible Boolean, recover it's value on backtracking

```
val cp = CPSolver()
val v = Array.tabulate(3)(i => new ReversibleBool(cp))
cp.sc = new IDSSearchController(cp,2) // iterative discrepancy
cp.exploration {
  cp.branch { v(0).value = true }
            { v(0).value = false}
  cp.branch { v(1).value = true }
            { v(1).value = false}
  cp.branch { v(2).value = true }
            { v(2).value = false}
  println(v.mkString(","))
}
```



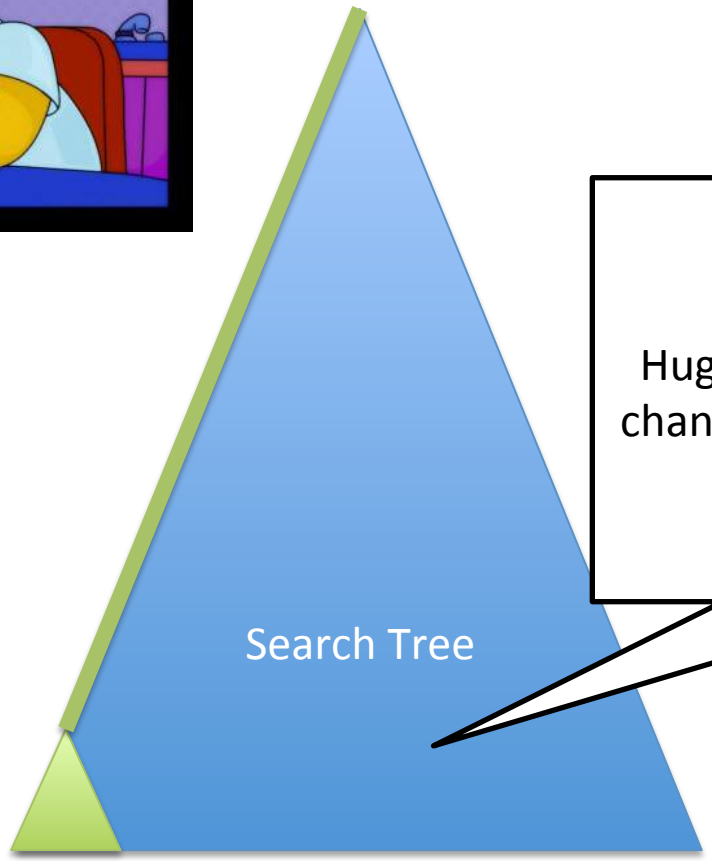
1 2 4 5 3 6 7 X

Binary First Fail Search

X is the uninstanciated var with min dom size

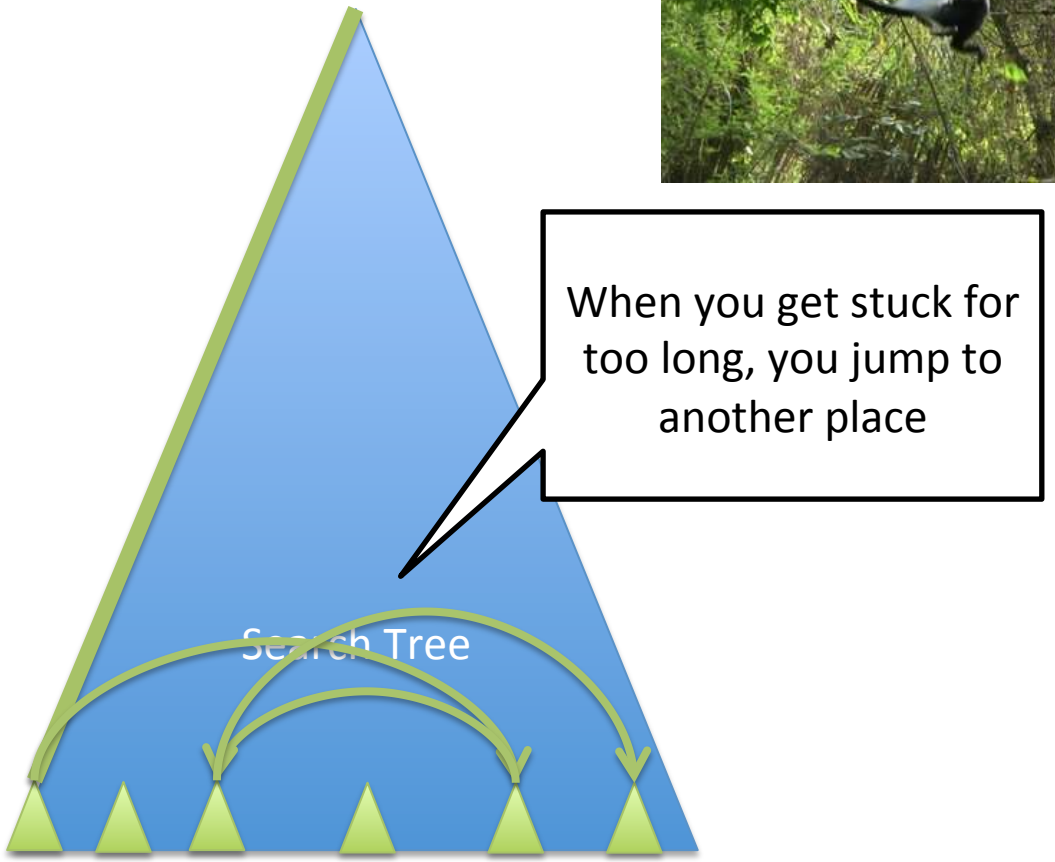
```
cp.exploration {
 while (!allBounds(vars)) {
        val unbound = vars.filter(!_.isBound)
        val minDomSize = unbound.map(_.size).min
        val x = unbound.filter(_.size == minDomSize).first
        cp.branch(post(x == x.min)) // left alternative
                  (post(x != x.min))  // right alternative
 }
 println("solution found")
}
```

Weakness of CP on some problems



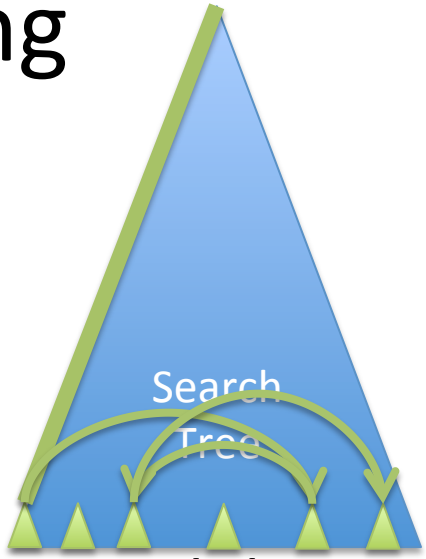
Huge search tree, you'll never get a chance to get there ? You are stuck …

Idea: Jump In The Search Tree



Jump In The Search Tree
Is exactly what LNS is doing

Fix, relax and restart …

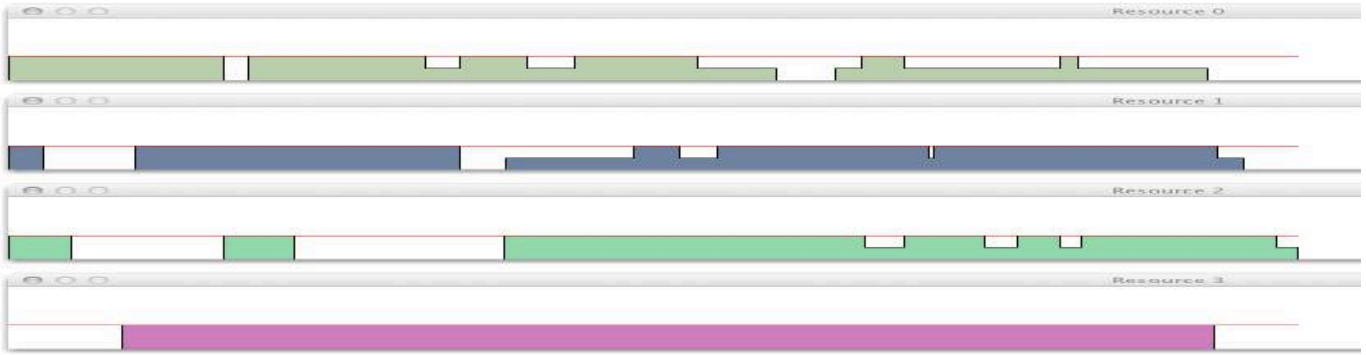


1. Find a first initial solution, S*
2. Randomly relax S* and re-optimize with search limit
   - Relax = fix some variables to their values in S*
3. Replace S* by the best solution found

CP Scheduling: Job Shop

```
val cp = CPScheduler(horizon)
val activities =
                Array.tabulate(nActivities)(I => Activity(cp, durations(i)))
val resources  = Array.tabulate(nResources)(r => UnitResource(cp))
cp.minimize(maximum(activities)(a => a.end)) subjectTo {
    for (i <- Activities)
        activities(i) needs resources(machines(i))
    for (i <- 0 until nActivities - 1; if (jobs(i) == jobs(i + 1)))
        activities(i) precedes activities(i + 1)
} exploration {
    for (r <- (0 until nResources))
        resources(r).rank()
}
```

Scheduling With Capacity




```
val resources  = Array.tabulate(nResources)(m => MaxResource(cp, 2))
for (i <- Activities)
      activities(i) needs 1 ofResource resources(machines(i))
cp.minimize(makespan) subjectTo {
      for (i <- Activities; if (jobs(i) == jobs(i + 1)))
            activities(i) precedes activities(i + 1)
} exploration {
      cp.setTimes(activities)
}
```

Other Features of CP Scheduling

- Disjunctive
- Reservoirs:
  - Variable capacity
  - Activities generate/consume permanently capa
- State Resources (soon)
- Transition Times (soon)
- Dedicated LNS Relaxation (Partial Order)

Quadratic Assignment Problem (QAP)

```
val cp = CPSolver()
var w: Array[Array[Int]]  // weight matrix
var d: Array[Array[Int]]  // distance matrix
val x = N map (v => CPVarInt(cp, 0 until n))
val bestSol = Array.fill(n)(0)
val rand = new scala.util.Random(0)
```

```
cp.minimize(sum(N, N)((i, j) => D(x(i))(x(j)) * w(i)(j))) subjectTo {
   cp.add(alldifferent(x))
} exploration {
  cp.binaryFirstFail(x)
  (0 until n).foreach(i => bestSol(i) = x(i).value)
}
```

Record the current best solution each time a new one is discovered

QAP with LNS

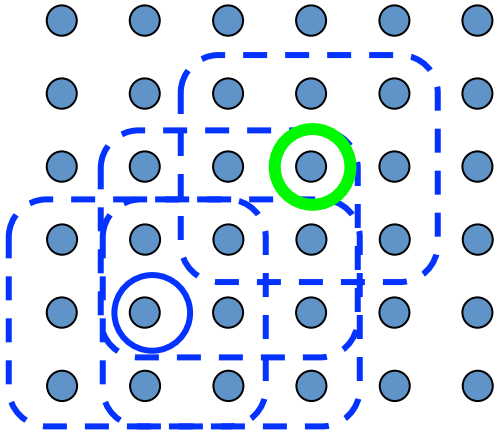
```
val cp = CPSolver()
var w: Array[Array[Int]]  // weight matrix
var d: Array[Array[Int]]  // distance matrix
val x = N map (v => CPIntVar(0 until n))
val bestSol = Array.fill(n)(0)
val rand = new scala.util.Random(0)
cp.lns(300,50) {
  cp.post((0 until n).filter(i => rand.nextInt(100) < 50).map(i => x(i) == bestSol(i)))
}
cp.minimize(sum(N, N)((i, j) => D(x(i))(x(j)) * w(i)(j))) subjectTo {
  cp.add(alldifferent(x))
} exploration {
  cp.binaryFirstFail(x)
  (0 until n).foreach(i => bestSol(i) = x(i).value)
}
```

300 restarts, 50 backtracks max

relax 50% of the variables

Constraint-based local search

- Perform a descend in the solution space; repeatedly move from one solution to a better one
- Next solution identified via neighborhood exploration
  - TSP: moving a city from one position to another one in the current circuit
  - NQueens: moving a queen to a position where the overall degree of violation is decreased the most



- Constraint-based local search is about
  - Quickly evaluating objective function on neighbors
  - Quickly moving to a neighbor
  - Ensuring a declarative front end for defining problem

CBSL: Model for N-Queens

```
val m: Model = new Model
val Queens = 0 until N
val queens = Array.tabulate(N)
                    (q => IntVar(m, Queens, q, "queen" + q))

val c: ConstraintSystem = new ConstraintSystem(m)

c.add(alldifferent(for ( q <- Queens) yield (queens(q) + q)))
c.add(alldifferent(for ( q <- Queens) yield (queens(q) – q)))

c.close
m.close
```

Initialized on the diagonal

CBSL: Tabu for N-Queens

```
val maxIter = 10000
var it = 0
val tabu = Array.fill(N)(-1)

while ((c.violation.value > 0) && (it < maxIter)) {
    val allowedQueens = Queens.filter(q => tabu(q) < it)
    val (q1,q2) = selectMin(allowedQueens,allowedQueens)
                        ((q1,q2) => c.swapDelta(queens(q1),queens(q2)),
                         (q1,q2) => q1 < q2)
    queens(q1) :=: queens(q2)
    tabu(q1) = it + tabuLength
    tabu(q2) = it + tabuLength
    it += 1
}
```

Filter the not tabu queens

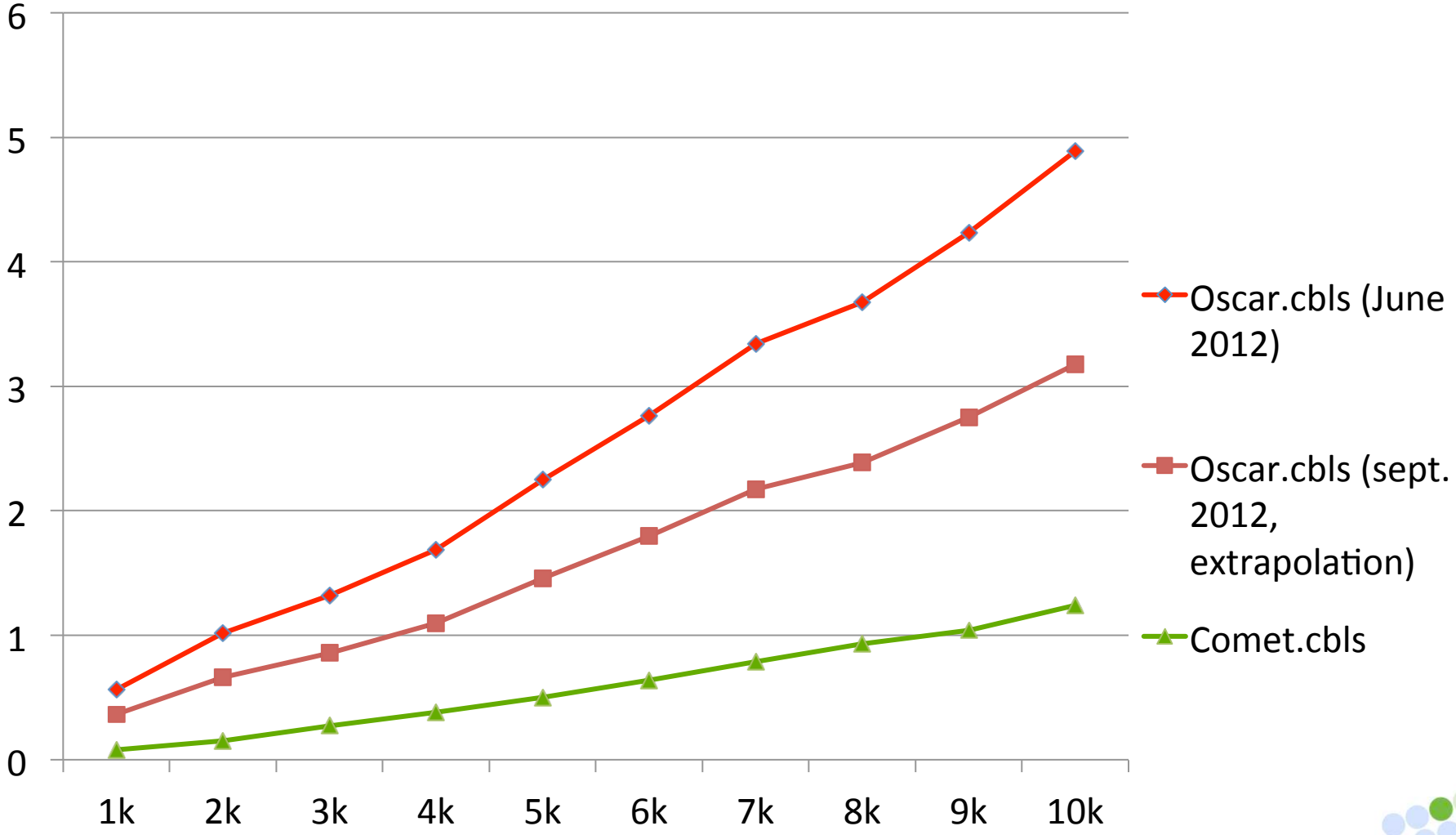
Select the swap with the best gradient

Do the swap

Set the queens from the swap tabu during some iteration

N-Queens Performances

Thanks to Jean-Noël Monette from Upsala



Features CBLS

- Libraries of standard invariants and constraints

- Several mechanisms to speed up search

... Requires more time to present properly, so check the next presentation…

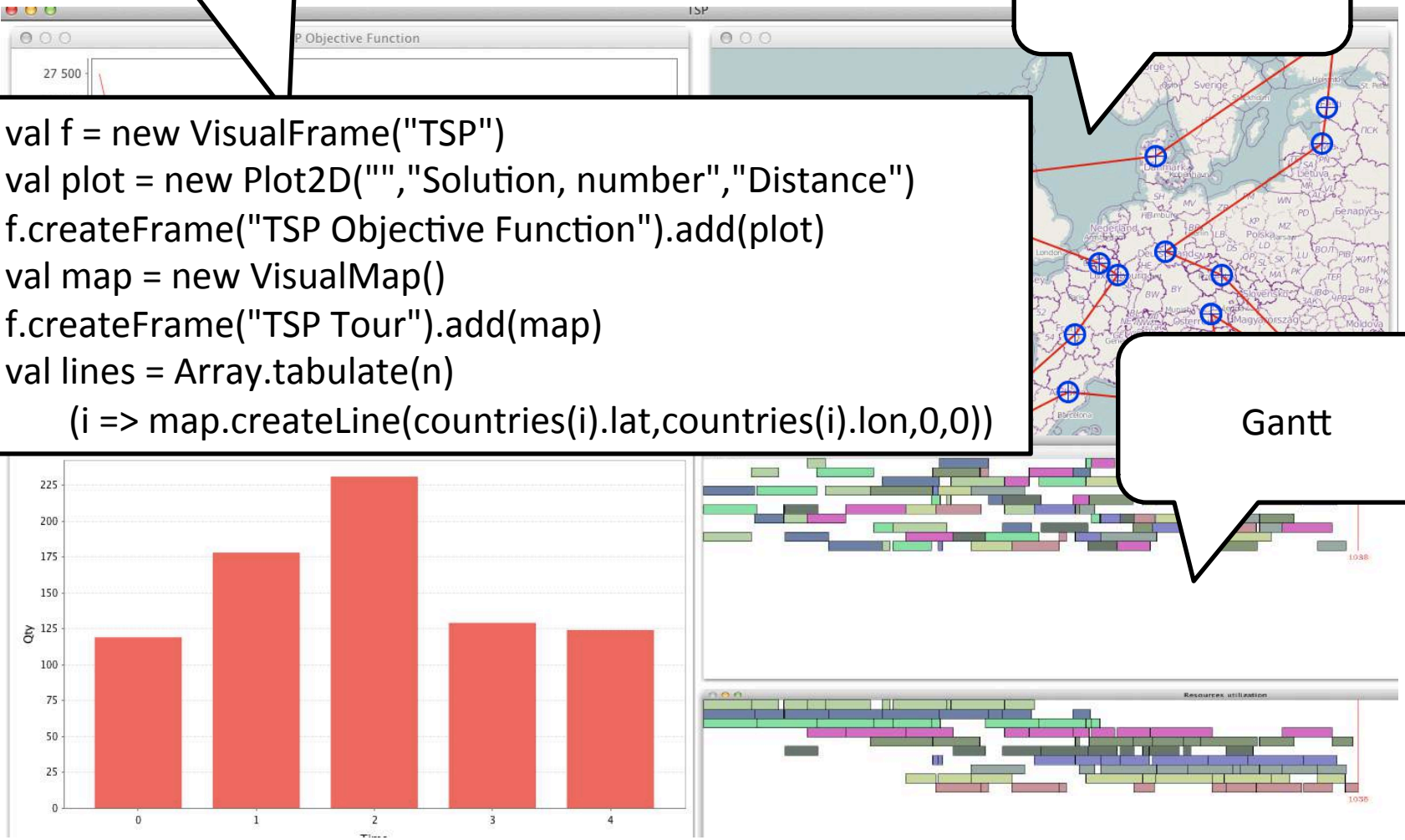
OscaR Visu

Jfree Charts Wrappers

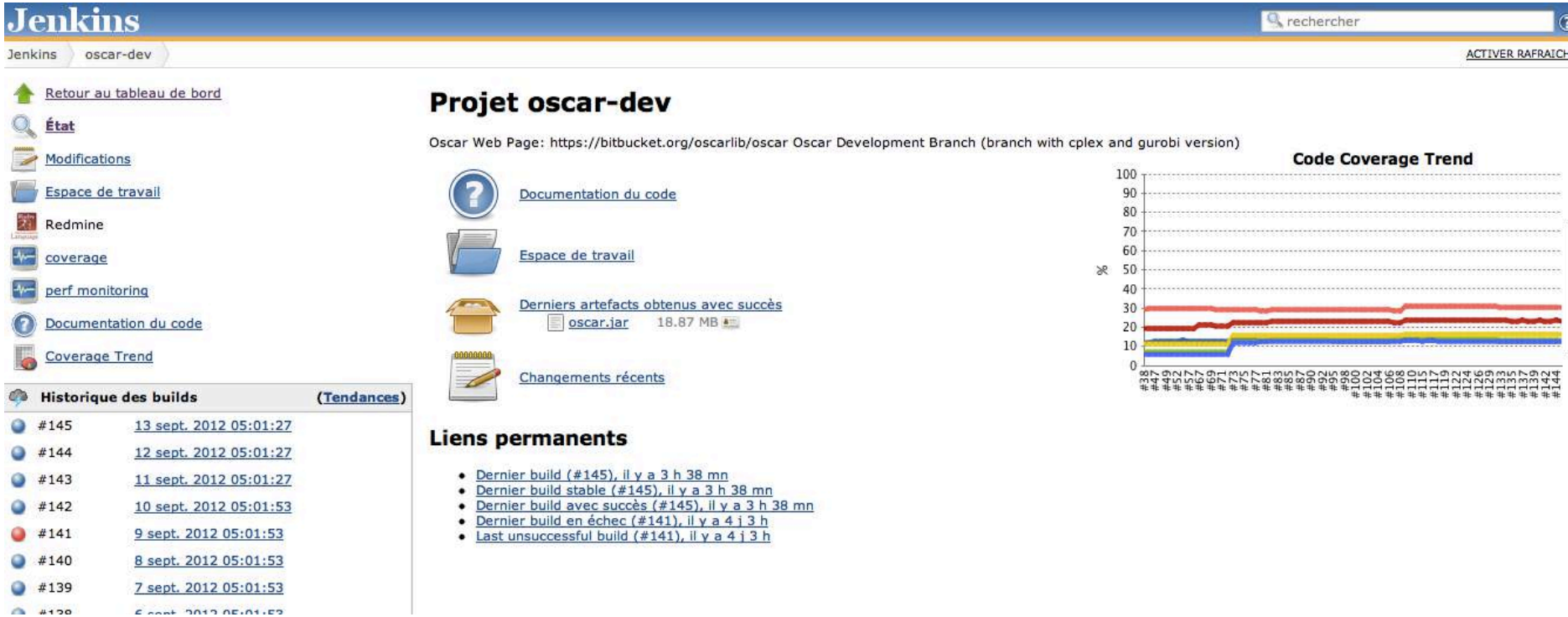
Open Street Map

Gantt

```
val f = new VisualFrame("TSP")
val plot = new Plot2D("","Solution, number","Distance")
f.createFrame("TSP Objective Function").add(plot)
val map = new VisualMap()
f.createFrame("TSP Tour").add(map)
val lines = Array.tabulate(n)
      (i => map.createLine(countries(i).lat,countries(i).lon,0,0))
```

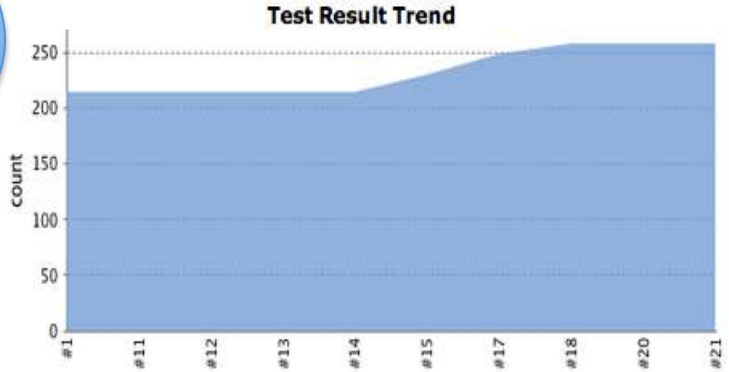


Nightly Build &Tests with CI Server

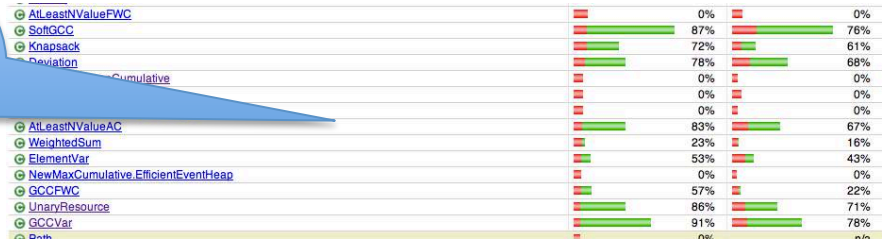


(soon ISO … certified ;-))

Performance monitoring



Code coverage monitoring



+25% unit tests

An overview of

OscaR

OPERATIONAL RESEARCH IN SCALA

https://bitbucket.org/oscarlib/oscar