Imperial College London

Repair and Local Search in ECL^{i}PS^e

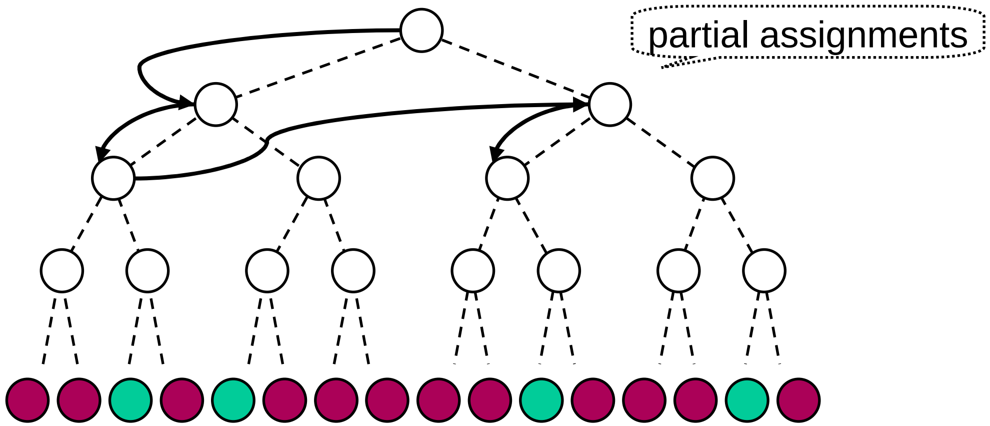
Joachim Schimpf

IC-Parc

Overview

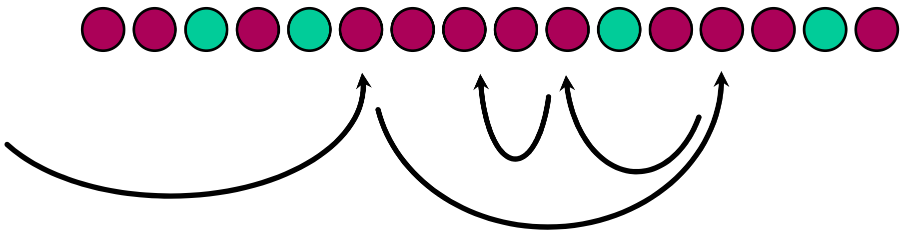
- Tree Search vs. Local Search
- The ECLiPSe CLP system
- The *repair* library
- Classical Local Search with the *repair* library
- Repair Methods

Exploring search spaces



CLP Tree search:

- constructive
- partial/total assignments
- systematic
- complete or incomplete

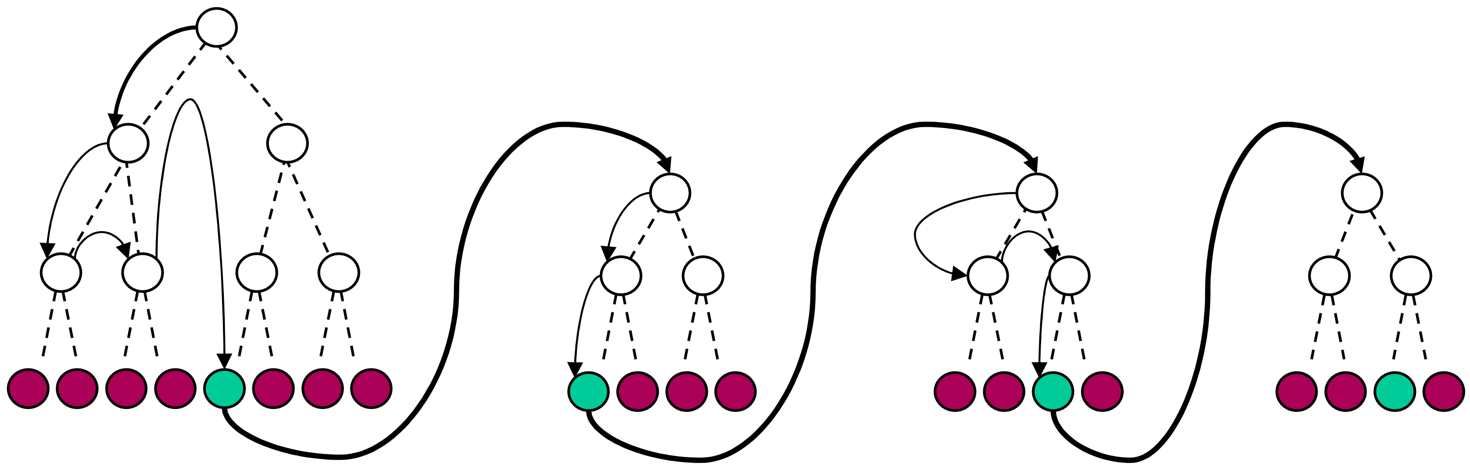


"Local" search:

- move-based (trajectories)
- only total assignments
- usually random element
- incomplete

Hybrids in CLP without special support

- E.g. Shuffle Search
  - tree search within subtrees
  - “local moves” between trees, preserving part of the previous solution’s variable assignments



- Pesant & Gendreau, Neighbourhood Models

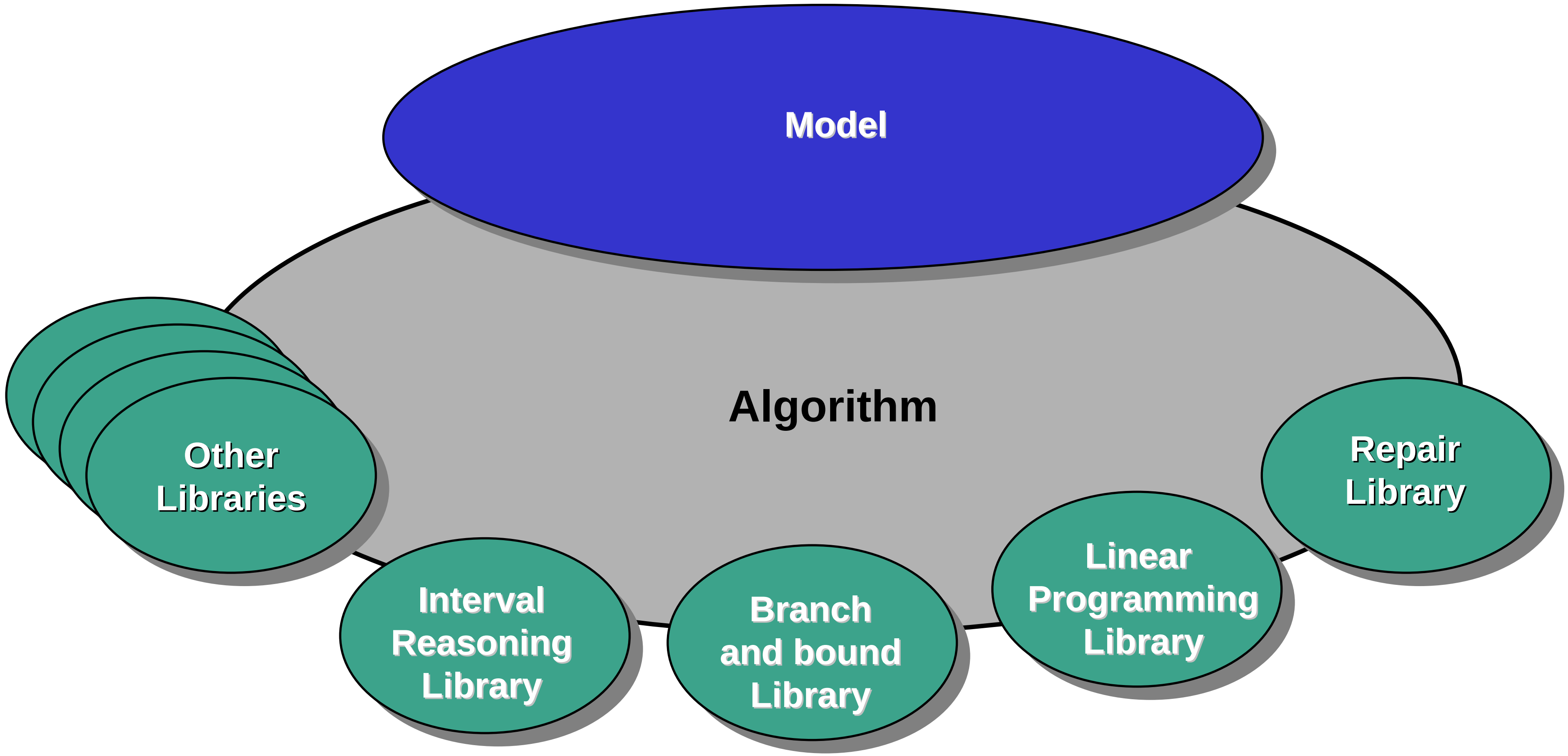
Issues with Classical Local Search in CP

- Efficient in CP implementation:
  - small monotonic change, e.g. single variable instantiation, domain narrowing
  - the reverse operation on backtracking

- Inefficient in CP framework:
  - small non-monotonic change, e.g. change value of a single variable
  - requires potentially deep backtracking and many re-instantiations

- Required for Local Search:
  - efficient small non-monotonic changes

ECLiPSe for Modelling and Solving



ECLiPSe Programming Language

- Logic Programming based
  - Predicates over Logical Variables   X#>Y, integers([X,Y])
  - Disjunction via backtracking   X=1 ; X=2
  - Metaprogramming (e.g. constraints as data)
- Modelling extensions
  - Arrays and structures
  - Iteration/Quantification   (foreach(X,Xs) do …)
- Constraint support
  - Attributed variables   X{1..5}
  - Data-driven computation (propagation)   suspend(…)
  - Solver libraries   :- lib(ic).

The *repair* library – Tentative Values

- Tentative values

  ```
  X::1..5, X tent_set 3
  ```

  X{1..5, tent:3}

  In addition to other attributes (e.g. domain).

  Tentative value can be changed freely (unlike domain)

  Change can trigger computation (like domain change)

- Conflict variables

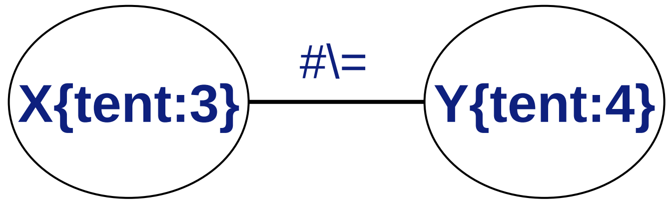
  Tentative value not in domain

  X{1..5, tent:7}

The *repair* library – Monitoring Constraints

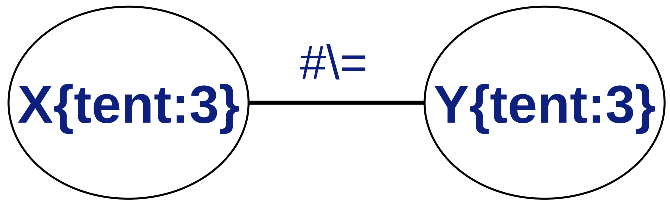
- Annotating arbitrary constraints

  ```
  X #\= Y r_conflict ConfSet
  ```



- Conflict constraint

  If not satisfied with current tentative values



- Conflict set

  ```
  conflict_constraints(ConfSet, Constrs)
  ```

  Set of conflict constraints, dynamically maintained.

  Constraints as data structures.

The *repair* library – Tentative Propagation

- Data-driven computation with tentative values

  Suspend until tentative value changes, then execute

- Arithmetic

  `Z tent_is X+Y`

  Update tentative value of Z *whenever* tentative value of X or Y changes (automatic and incremental)

- General

  `tent_call([X,Y], Z, Z is X+Y)`

  Recompute and update tentative value of Z *whenever* tentative value of X or Y changes

Local Search with *repair* library

```
:- lib(repair).

knapsack(N, Profits, Weights, Capacity, Opt) :-
    length(Vars, N),                                        % N booleans
        Capacity  >=  Weights*Vars r_conflict cap,          % constraint
        Opt  tent_is  Profits*Vars,                         % the objective
    local_search(cap, Vars, Opt).                           % search
```

Local Search

Local Search - algorithm template

local_search:

```
set starting state
while global_condition
    while local_condition
        select a move
        if acceptable
            do the move
            if new optimum
                remember it
    endwhile
    set restart state
endwhile
```

Different parameters:

- hill climbing
- simulated annealing
- tabu search
- ... and many variants

E.g. Hill Climbing

```
try_move(Vars, ProfitVar, OldBest, NewBest) :-
    (
        ProfitVar tent_get OldProfit,
        flip_random(Vars),                          % do a move
```

undo move

tentative value propagation

```
        ProfitVar tent_get NewProfit,
        NewProfit > OldProfit,                      % uphill?
        conflict_constraints(cap, [])               % solution?
    ->
        NewBest is max(OldBest,NewProfit)           % accept
    ;
        NewBest = OldBest                           % reject move
    ).
```

Techniques used here

- Move operation and acceptance test:
  – If the acceptance test fails (no solution or objective not improved) the move is automatically undone by backtracking!
- Detecting solutions:
  – Constraint satisfaction is checked by checking whether the conflict constraint set is empty
- Monitoring cost/profit:
  – Retrieve tentative value of Profit-variable <u>before</u> and <u>after</u> the move to check whether it is uphill
  – Since the move changes the tentative values of some variable(s), tent_is/2 will automatically and incrementally update the Profit variable!

Computing Violatedness

- Conflict monitoring not ideal for LS

  Generic, works for any constraint (r_conflict annotation).

  But many LS algorithms need measure of violatedness.

- E.g. capacity constraint

```
cap_con(Cap, Vars, Weights, Viol) :-
    Viol tent_is max(0, Vars*Weights – Cap).
```

- Simple constraint (0..1 violations)

```
differ(X, Y, Viol) :-
    tent_call([X,Y], Viol, (X\=Y -> Viol=0;Viol=1)).
```

Repair Techniques

- One Basic Technique
  - Start with “good” inconsistent assignment
  - Increase consistency incrementally

- Applications
  - **Repair Problems**

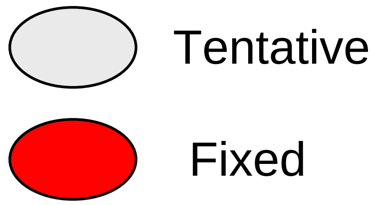
    “good” inconsistent assignment: ***the previous solution***
  - **Repair-Based Constraint Satisfaction**

    “good” inconsistent assignment: ***the partially consistent soln. found by heuristics***
  - **Repair-Based Constraint Optimization**

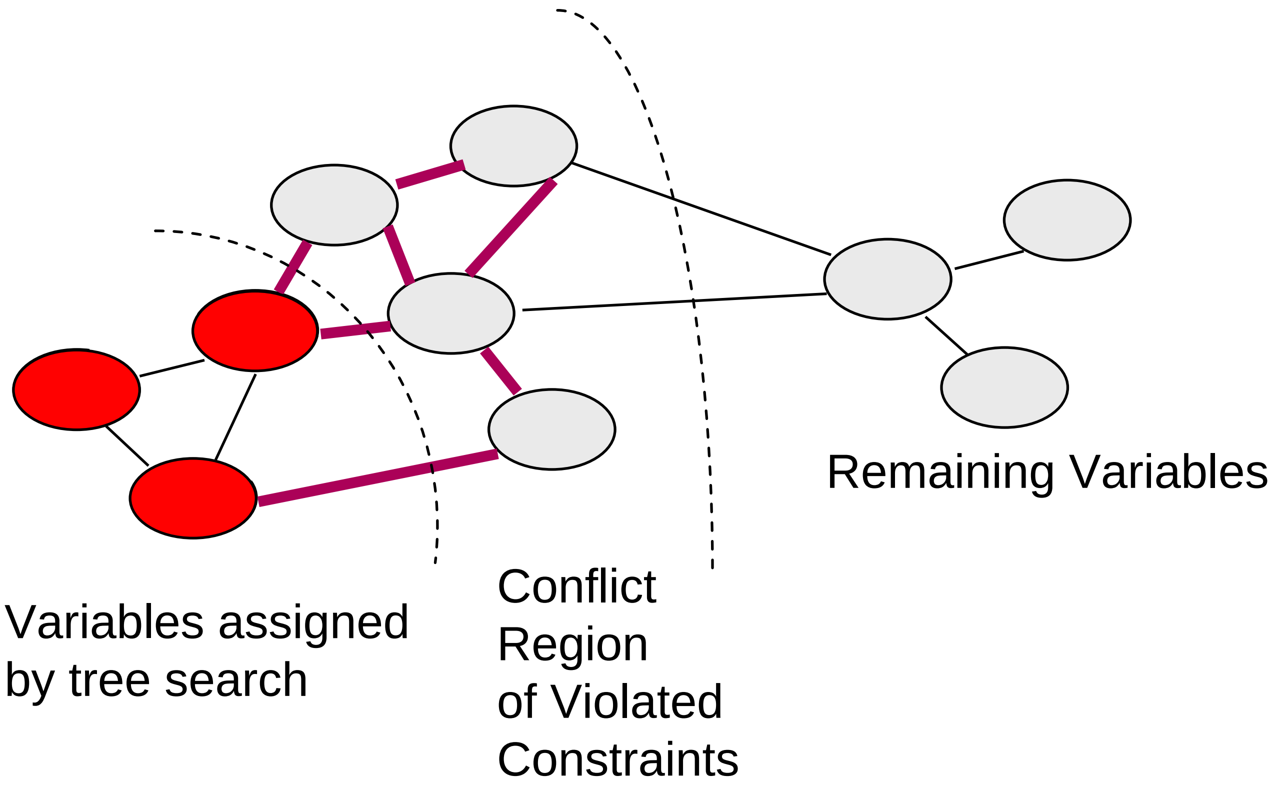
    “good” inconsistent assignment: ***but good with respect to optimization function***
  - **Hybridization (e.g. Probing)**

    “good” inconsistent assignment: ***a good solution produced by a partial solver***

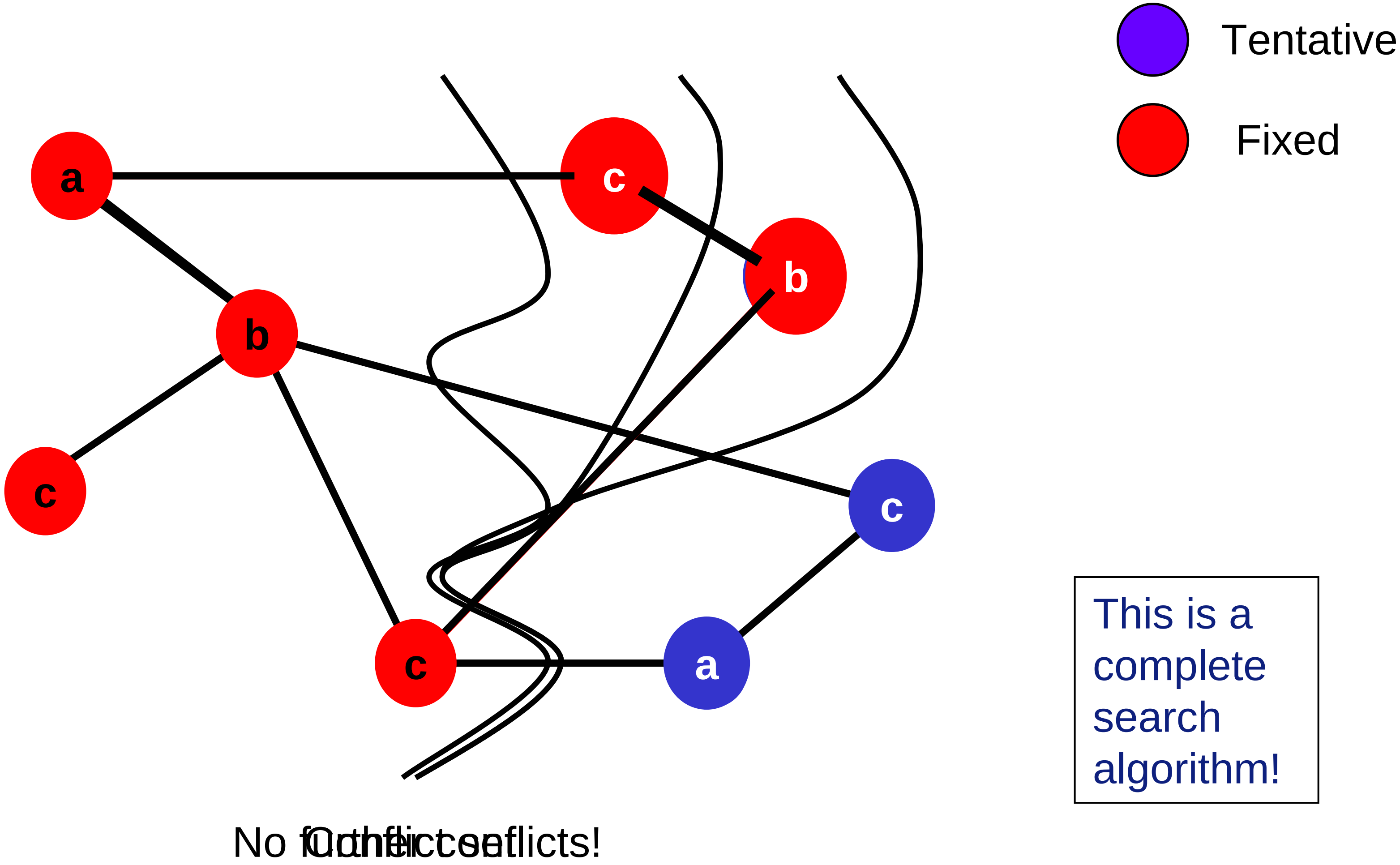
Repairing a Tentative Assignment



- The Conflict Region



Repairing a Tentative Assignment - detail



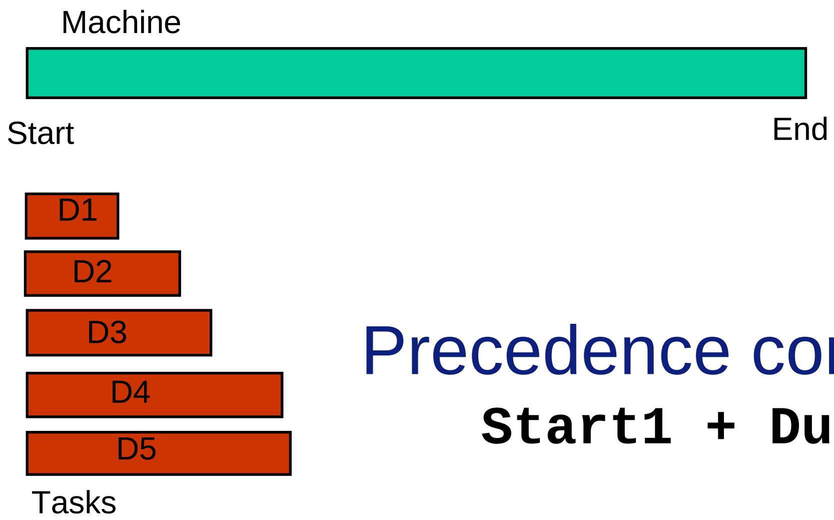
This is a complete search algorithm!

Local Search with Tentative Assignments
Tree Search with Tent. Ass. and Domains

```
model(…) :-
     Vars :: Domain,
     Vars tent_set StartingSolution,
     …
     ic: <Constraint>,
     <Constraint> r_conflict cs,
     …

search :-
     ( find_var_in_conflict_constraint(cs, V) ->
          indomain(V),
          search
     ;
          true
     ).
```

Repairing solutions from partial solvers



One machine scheduling

Precedence constraints (temporal,easy)

```
Start1 + Duration1 =< Start2
```

Resource constraints (disjunctive, hard)

```
noclash(S1,D1,S2,D2) :-
        S1 >= S2+D2  ;  S2 >= S1+D1.
```

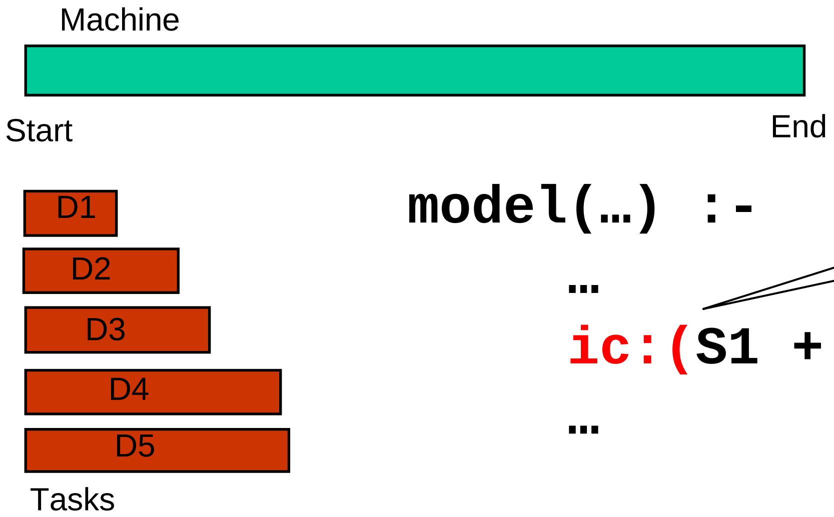
disjunction

Algorithmic Idea

- Temporal constraints
  - handled by interval propagation without search:
  - lower domain bounds are valid solutions for the temporal subproblem!
  - we use these values as tentative values

- Resource constraints
  - initially only monitored for conflicts (with respect to the solution of the temporal subproblem)
  - when in conflict, make a choice for the disjunction
  - in each branch a temporal constraint is added

Annotated Model



Precedence constraints handled by interval solver

```
model(…) :-
        …
        ic:(S1 + D1 =< S2)
        …

        noclash(S1,D1,S2,D2) r_conflict cs,
        …

noclash(S1,D1,S2,D2) :-
        ic:(S1 >= S2+D2) ; ic:(S2 >= S1+D1).
```

Resource constraints monitored for conflict

Choice between two additional precedence constraints

(Quite generic) search routine

```
repair_label :-
      conflict_vars(CVs),
      tent_set_to_min(CVs),
```

tentative value propagation

```
      conflict_constraints(cs,CCs),
      ( CCs == [] ->
            true
      ;
            CCs = [Constraint|_],
            call(Constraint),
```

interval propagation

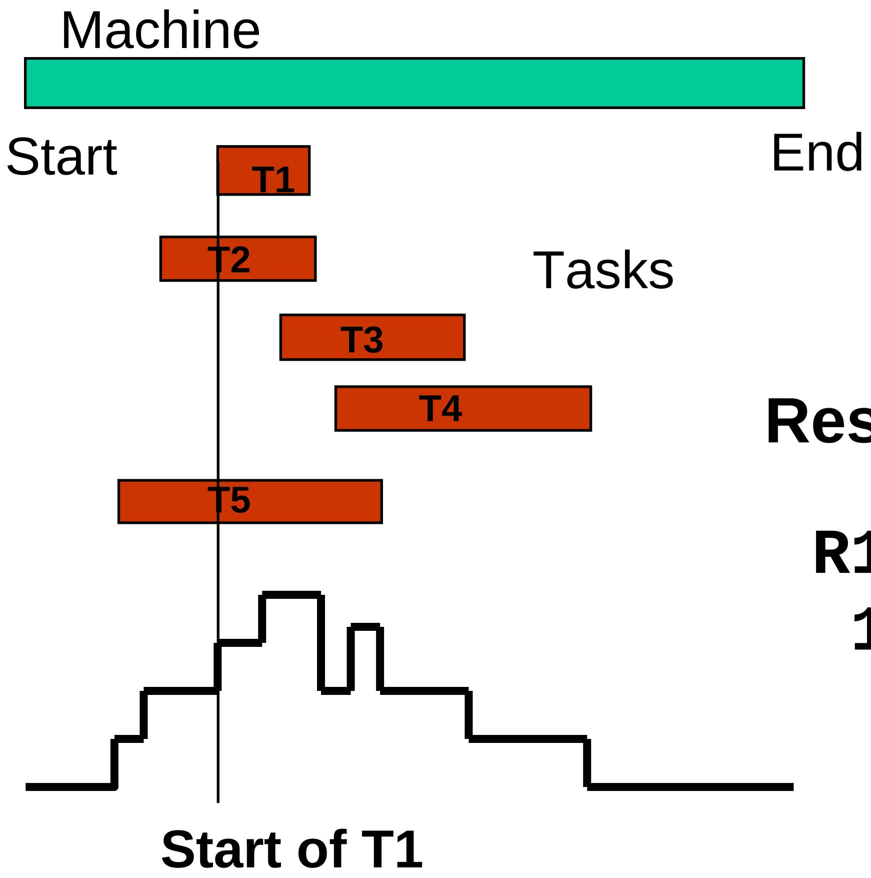
```
            repair_label
      ).
```

Set tentative values to sub-problem solution (lower domain bound)

Select constraint to repair

Address the constraint:
Create choice point
Add temporal constraint in each branch

Repair heuristics: max overlap



Resource usage at start of T1:

```
R1 tent_is
 1 + B12 + B13 + B14 + B15
```

```
overlap(S1,S2,D2,B12) :-
    B12 tent_is (S1>=S2 and S1=<S2+D2).
```

Variant: Probing with linear subproblem

- Scenario
  - more complex, but linear subproblem
  - more complex objective, e.g. minimal perturbation

- Algorithm
  - Send linear constraints to simplex
  - Simplex solves subproblem for given objective (*Probe*)
  - Set tentative values to simplex solution
  - Propagate tentative values to overlap variables
  - Identify bottleneck (maximum overlap)
  - Add precedence constraint on two bottleneck tasks

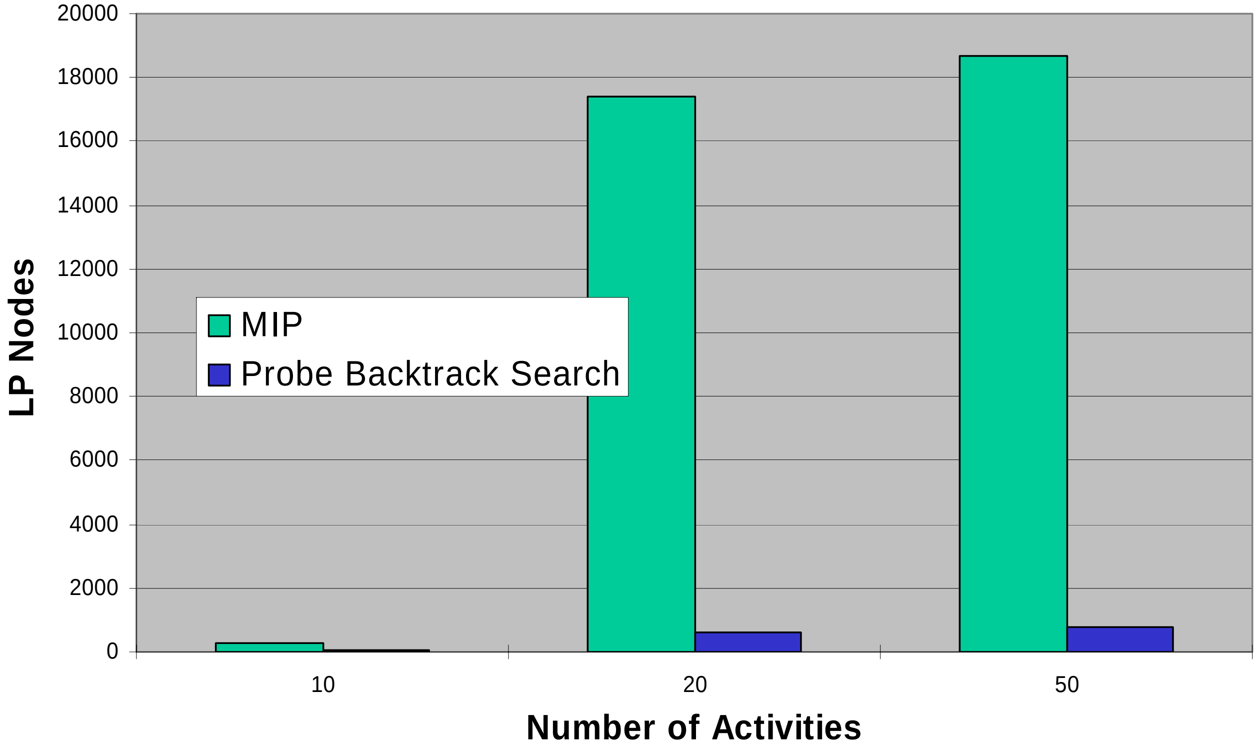
Setting Tentative Values via *eplex*

```
:- lib(eplex), lib(repair).

eplex_to_tent(Expr, Opt) :-
    eplex_solver_setup(Expr, Opt, [], 0
           [new_constraint,post(set_ans_to_tent)]).

set_ans_to_tent :-
    eplex_get(vars,Vars),
    eplex_get(typed_solution,Solution),
    Vars tent_set Solution.
```

MIP vs Probe Backtrack Search



ECLiPSe

- Web site

  http://www.icparc.ic.ac.uk/eclipse

- Free academic licence

- Successor for *repair* library is planned