

# Constraint Programming 2002

## Graduate Course

# HEURISTIC METHODS

Markus Böhlin  
[bola@sics.se](mailto:bola@sics.se)

Västerås, December 13, 2002



---

Constraint Programming 2002: *Heuristic Methods*

Västerås, December 13, 2002

## Outline

### Heuristic and Exact Methods

Introduction to Local Search

Initial Solutions

Cost Functions

Neighborhoods

Improvement Strategies

Plateaus and Local Optima

Algorithms

## Heuristic and Exact Methods

*Heuristic* – from the Greek *heuriskein*, meaning to find out, discover.

Exact Methods:

- Complete — guarantees to find a solution / optimum if one exists.
- A *proof of infeasibility* (to reject a partial label) can be expensive to find.
- To find a *proof of optimality* for optimization problems is very expensive.
- Solving time doesn't scale well with increased problem size.
- Examples: Backtrack search, CLP, Branch&Bound optimization.

Heuristic Methods:

- Based on “common sense” and what *seems* best to do.
- Can “undo” bad decisions during search.
- Answer for a satisfiability problem: “Yes”, or “Probably not”.
- Answer for an optimization problem: a solution hopefully close to the optimum.
- Conclusion: Heuristic methods are *not complete* and are not guaranteed to find a solution or an optimum.

## Local Search Applications

- The Traveling Salesman Problem (TSP)
- Job-Shop Scheduling
- Propositional Satisfiability (SAT)
- 0-1 Integer Optimization
- Numerous other problems
- Available in ILOG Solver
- Specialized languages: *Localizer* etc.

# Outline

Heuristic and Exact Methods

## Introduction to Local Search

Initial Solutions

Cost Functions

Neighborhoods

Improvement Strategies

Plateaus and Local Optima

Algorithms

## Introduction to Local Search

Local search is a common *algorithm structure* for heuristic methods.

1. Choose an initial possible solution  $s$  to the problem.
2. Choose another possible solution  $s'$  from  $N(s)$ , the *neighborhood* of  $s$ , and let  $s = s'$ .
3. Repeat (2) until  $s'$  satisfies a *stop criteria*.

- It is common to use a *cost function*  $f$  which grades possible solutions. We only select a new solution if it is better than the old one.
- A solution  $s$  can be either partial or complete. In pure local search, solutions are often complete.

## Good

- Avoids “thrashing”, bad choices can be repaired without backtrack.
- Search space in each iteration is small compared to the full search space.
- It is often possible to evaluate how good possible solutions are efficiently.
- Modification of one possible solution to another in the neighborhood is often possible to implement efficiently.
- Because of efficiency reasons, sometimes the only usable method to solve complex problems.

## Bad

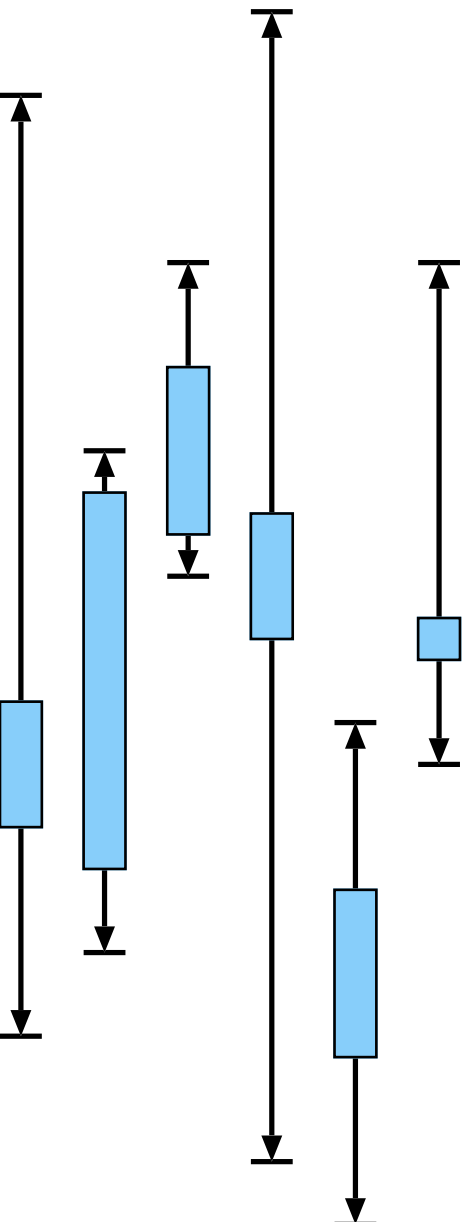
- Incomplete.
- Cannot use consistency.
- Problem specific solvers.
- Solver is often a “black box”.

However, several of these problems can be addressed using *hybrid methods*.

## Running Example: *Single-Capacity Resource Scheduling* (SCRS)

- Assume a set of activities (tasks)  $T = \{t_1, \dots, t_n\}$ .
- Each activity  $t_i$  has a *start time*  $s_i$  (domain variable) and a *duration*  $d_i > 0$  (constant). The *end time*  $e_i$  of any task  $t_i$  can be calculated as  $e_i = s_i + d_i$ .
- The goal is to schedule all activities in  $T$  on a machine (resource)  $m$ .
- $m$  can handle at most one activity at the same time.
- No task can be interrupted while processed on  $m$ .
- Equivalent to the serialized global constraint.

### A solution candidate for the SCRS



## Outline

Heuristic and Exact Methods

Introduction to Local Search

### Initial Solutions

Cost Functions

Neighborhoods

Improvement Strategies

Plateaus and Local Optima

Algorithms

## Initial Solutions

For satisfiability:

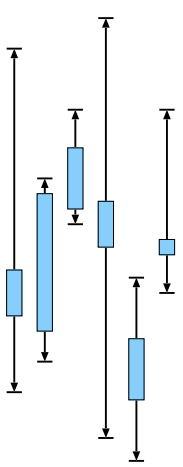
- Completely random.
- Approximation (use heuristic).
- Empty.

For optimization:

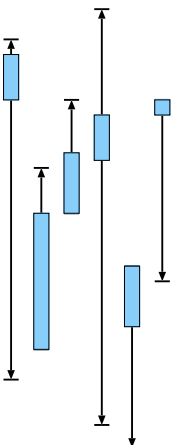
- Must (should) satisfy all constraints.
- Approximation of optimum (use heuristic).
- Random.
- Empty.
- etc.

## Initial Solutions for SCRS (satisfiability)

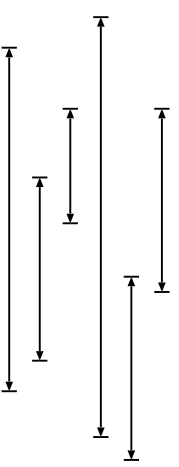
- Random:



- Approximation:



- Empty:



## Outline

- Heuristic and Exact Methods
- Introduction to Local Search
  - Initial Solutions
  - Cost Functions**
  - Neighborhoods
  - Improvement Strategies
  - Plateaus and Local Optima
  - Algorithms

## Cost Functions

The cost function is used to grade solutions, and to guide the search. The cost function should have the following three characteristics:

- Few solutions in  $N(s)$  should have the same  $f$ -value. Otherwise, we will get many plateaus and bad guiding.
- $f(s')$  –  $f(s)$  should be possible to calculate efficiently. This is used as a measure for the “goodness” of  $s'$  and calculated in the inner loop of the local search algorithm.
- If the problem is a combined optimization and satisfaction problem, the optimization part of  $f$  should be “comparable” with the satisfaction part of  $f$ .

### Some Good and Bad Cost Functions

#### Good:

- The number of violated soft constraints.
- The sum of the weights of the violated constraints
- The sum of the amount of individual violation of the constraints

#### Bad:

- The minimum number of variables that has to be changed in order to satisfy the soft constraints. Expensive to calculate, lots of plateaus.

## Cost Function for SCRS

Three good cost functions:

- *Use the number of tasks intersecting with other tasks.* Low granularity, easy to calculate.
- *Use the number of conflicts between two tasks.* Better granularity, easy to calculate. Can be directly combined with external binary constraints.
- *Use the total overlap time between all tasks.* High granularity, easy to calculate. Not trivial to compare to other costs.

## Outline

- Heuristic and Exact Methods
- Introduction to Local Search
  - Initial Solutions
  - Cost Functions
  - Neighborhoods**
  - Improvement Strategies
  - Plateaus and Local Optima
  - Algorithms

## Neighborhood

- Search space should be closed under the neighborhood definition so that all solutions can be reached. However, this is not always possible.
- Small neighborhood → fast search, bad local optima.
- Large neighborhood → slower search, better local optima.

### How do we find a good neighborhood?

- A given problem class usually has certain constraint characteristics.
- A good neighborhood is one that explores possibly improving states close by. This can be used to reduce the neighborhood.
- Satisfaction problems — try to reduce the number of conflicts.
- Optimization problems — try to explore the space of feasible solutions.

## Common Neighborhoods

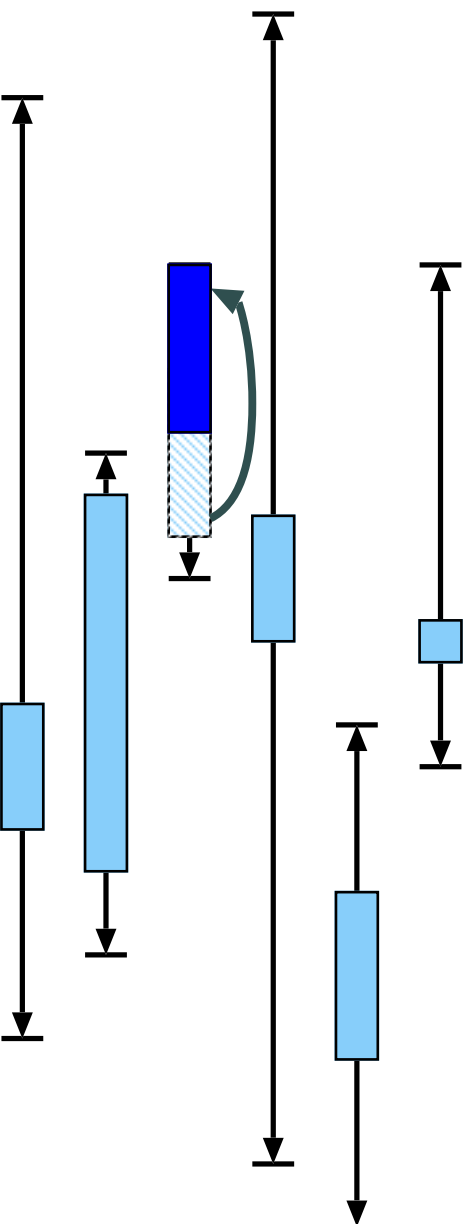
**$k$ -assign:** The solutions that can be reached by changing the value of at most  $k$  variables. Usable for general constraint satisfaction.

**$k$ -modify:** The solutions that can be reached by changing the value of the variables with at most  $k$ . Variant of  $k$ -assign. Usable for constraint satisfaction and boolean optimization.

**$k$ -swap:** The solutions that can be reached by swapping the value of at most  $k$  variables. Often useless for satisfaction problems. Usable for optimization, to keep a solution feasible.

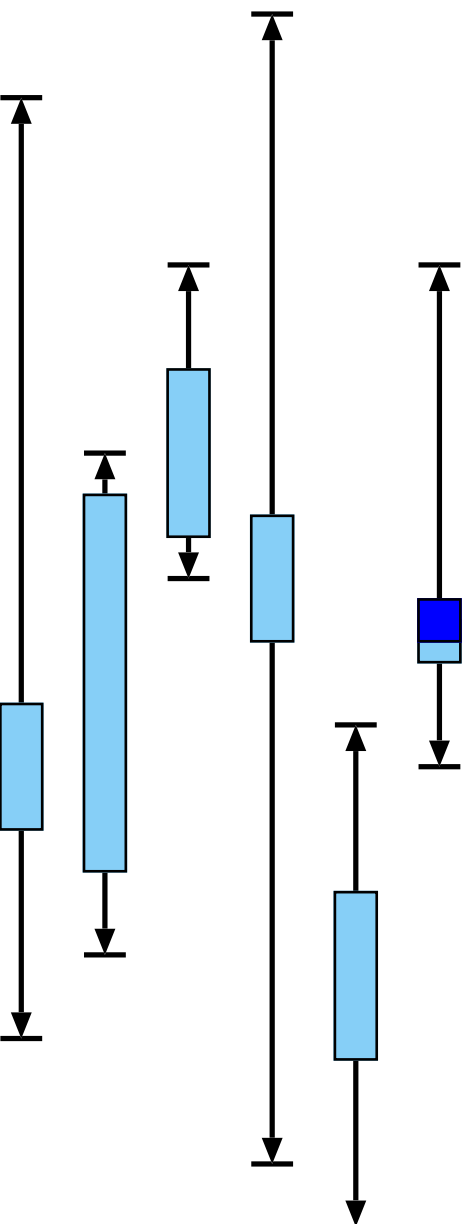
## 1-assign for SCRS

We assume integer domains and show the values resulting from changing the start time of one of the tasks in the problem.



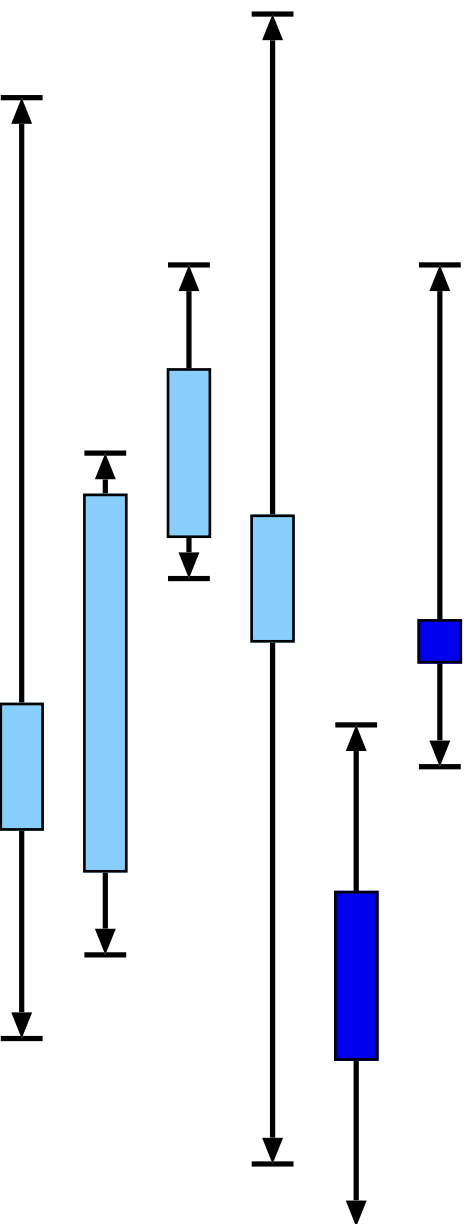
## 1-modify for SCRS

We again assume integer domains. This is the full example for the given state, where we modify each value by either increasing or decreasing it with 1.



## 1-swap for SCRS

In this neighborhood we exchange the values between two tasks. Several domain constraints are broken, and the number of conflicting tasks are not reduced.

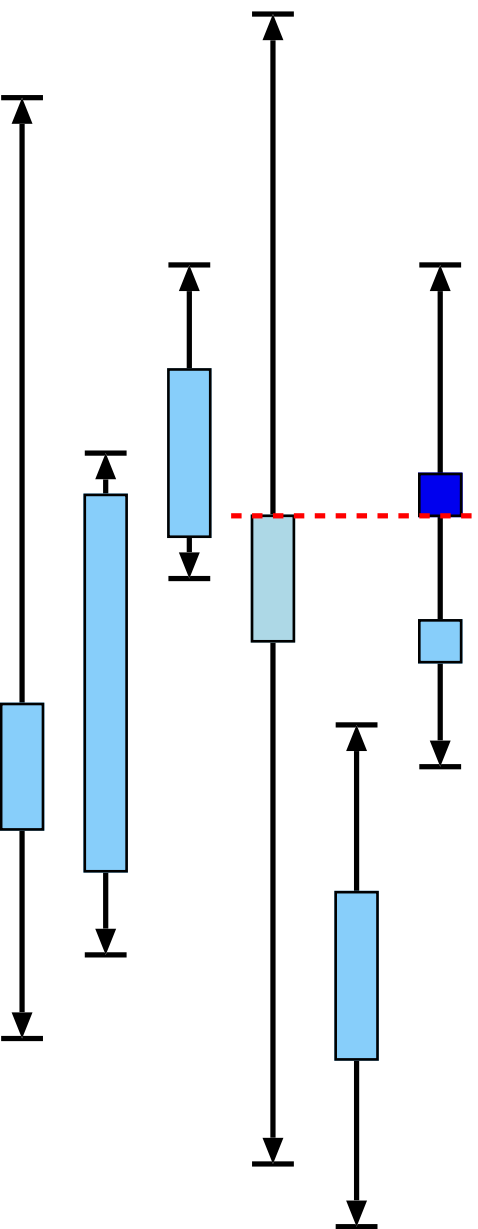


## An Adapted Neighborhood for SCRS

- The  $k$ -assign neighborhood is costly to compute,  $\mathcal{O}(dn)$ . (Impossible to compute efficiently if we have infinite domains).
- The  $k$ -modify neighborhood is not directly suitable.
- The  $k$ -swap neighborhood is just bad for this problem.
- For each task, there are  $2(n - 1)$  timepoints that clearly (can) reduce the number of conflicts.
- Try to move a task so that it is active immediately before or after another task.
- Complexity of neighborhood is  $\mathcal{O}(n^2)$ .

## Neighborhood Demo for SCRS

We try to move each task to the position either immediately before or immediately after another task. The example show this for task 1.



## Outline

- Heuristic and Exact Methods
- Introduction to Local Search
  - Initial Solutions
  - Cost Functions
  - Neighborhoods
- Improvement Strategies**
- Plateaus and Local Optima
- Algorithms

## Improvement Strategies

We assume a cost function  $f$  where lower values indicate better solution candidates. Three main strategies:

- First Descent
- Random Descent
- Steepest Descent

These are often combined with more exotic techniques to escape plateaus, diversify search, explore plateaus, etc.

### First Descent

1. Choose an initial possible solution  $s$  to the problem.
2. Choose the first possible solution  $s' \in N(s)$  for which  $f(s') < f(s)$  and let  $s = s'$ .
3. Repeat (2) until  $s'$  is a solution or is optimal, or until a plateau has been detected (there is no  $s'$  so that  $f(s') < f(s)$ ).

First Descent is based on how LS improvement is usually implemented. A list of neighbors is traversed, and the first one giving any improvement is selected as the new state.

## Random Descent

1. Choose an initial possible solution  $s$  to the problem.
2. Choose another random possible solution  $s' \in N(s)$  so that  $f(s') < f(s)$  and let  $s = s'$ .
3. Repeat (2) until  $s'$  is a solution or is optimal, or until a plateau has been detected (there is no  $s'$  so that  $f(s') < f(s)$ ).

Possible implementations:

- Randomly pick an element  $s'$  from  $N(s)$  until  $f(s') < f(s)$  (bad).
- Collect all  $P = \{s' \mid s' \in L \wedge f(s') < f(s)\}$ , select a random element from  $P$ .
- Variation: select element from  $P$  with a probability proportional to the cost decrease.

## Steepest Descent

Special version of random descent. Selects best candidate in neighborhood.

1. Choose an initial possible solution  $s$  to the problem.
2. Choose another possible solution  $s' \in N(s)$  so that  $f(s') < f(s)$ , and there is no other  $s'' \in N(s)$  so that  $f(s'') < f(s')$ . Let  $s = s'$ .
3. Repeat (2) until  $s'$  is a solution or is optimal, or until a plateau has been detected (there is no  $s'$  so that  $f(s') < f(s)$ ).

It is a good idea to randomize the choice of  $s'$  if several candidates are equally good.

## Outline

Heuristic and Exact Methods

Introduction to Local Search

Initial Solutions

Cost Functions

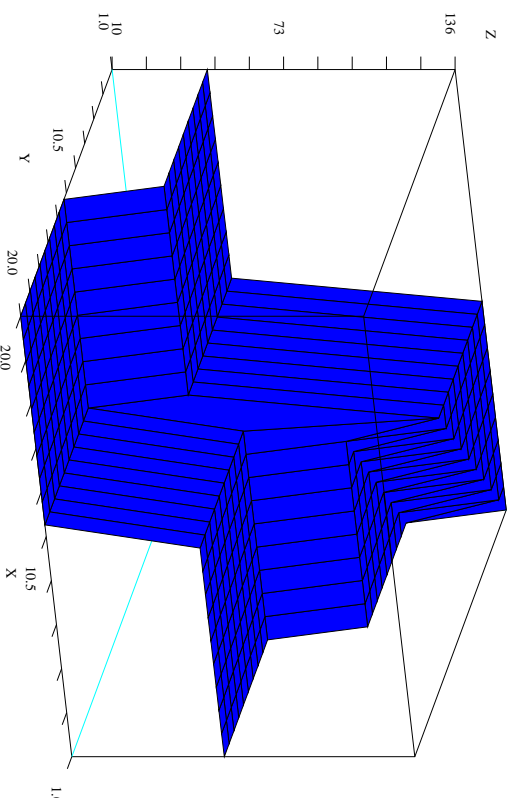
Neighborhoods

Improvement Strategies

**Plateaus and Local Optima**

Algorithms

## Escaping Plateaus and Local Minima



- All neighbors increase the cost — *local minimum*.
- No neighbor decrease the cost — *plateau* (generalization of local minimum).
- Basic Strategy: diversify search.

## Common Escape Techniques

- Randomization
- Horizon Extension
- Cost adaptation
- History-based Search
- Monte-Carlo Methods

## Randomization

- Randomize parameters in the search.
- Make a choice of neighbor with a probability inverse proportional to the cost.
- Restart from a random new solution.
- Randomly modify parts of the solution.
- Very common way to handle plateaus and local minima.
- Often crucial in achieving efficient search.
- Cheap in terms of implementation effort and complexity.

## Extension of Horizon

These techniques aim to increase the size of the neighborhood, and thus allowing the search to find states that improve the situation, escaping the plateau.

- Allow *flat moves* - if no better solution exist, pick one (at random) that at least does not increase cost.
- Temporary increase of the neighborhood size:

$k$ -assign  $\rightarrow (k + 1)$ -assign

$k$ -modify  $\rightarrow (k + 1)$ -modify

$k$ -swap  $\rightarrow (k + 1)$ -swap

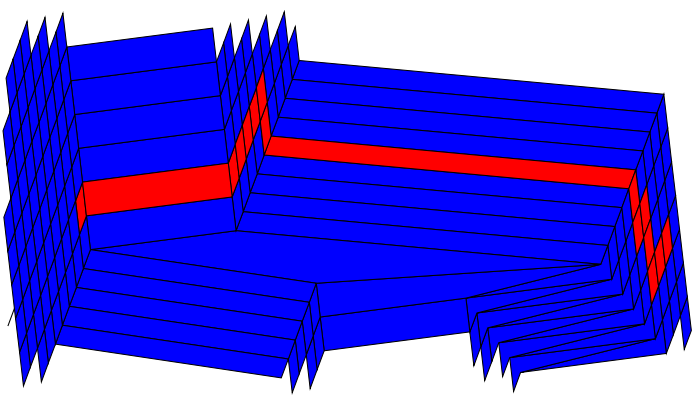
- Variant: Perform  $p$  successive improvements in order to find better solutions. Relax the cost constraint on moves.

## Cost adaptation

- Observation: Several constraints are violated more often than other constraints.
- Attach a *weight*  $w(C)$  to each constraint  $C$ . Initially  $w(C) = 1$  for all  $C$ .
- Increase weight on violated constraints:  $w(C) \leftarrow w(C) + 1$ .
- Decrease weight on satisfied constraints:  $w(C) \leftarrow w(C) - 1$ .
- Variation: No decrease, but *normalization* of weights after  $k$  iterations.
- Effect: Violated constraints will become more valuable to satisfy, and the search will thus be directed away from a local minimum / plateau.

## History-based search

- Avoid undoing moves that we have already committed to.
- If we remember each solution we've tried, we can achieve completeness!
- However, this means exponential space *and* time complexity of algorithm.
- Approach 1: remember  $n$  last changes made, and forbid undoing them (Tabu Search).
- Approach 2: Remember  $n$  last most general dead ends, and avoid them (No-goods).



## Tabu Search

- Keep track of the  $n$  last changes made in a *tabu list*, forbid undoing the changes in the tabu list.
- Parameter  $n$  is the *tabu tenure size*, critical to select right value for performance of search.
- If a change that is tabu decreases the cost, do it anyway (*aspiration criteria*).
- **Effect:** Diversification of search, possible to escape a local minimum / plateau.
- **Variation:** vary the size of  $n$  during the search (increase at plateaus).
- **Variation:** if a dead end is detected, decrease tabu list until we can go back to a previous state – A limited form of backtracking.
- **Very successful,** crucial in gaining local search performance for integer optimization, planning, scheduling, etc.

## Monte-Carlo based methods

- Local search that allows changes that “makes things worse”. with a probability inverse proportional to the increase of the cost.
- Special case: *Simulated Annealing*.
- Here, we decrease the probability (*temperature*) of a nondecreasing change as the search proceeds.
- Effect: More “irrational” behaviour, cost-increasing changes will be committed to and the search will (hopefully) escape from local minima and plateaus.
- If the temperature decrease is “sufficiently small”, convergence can be guaranteed. There is however no definition of “sufficiently”...

## Outline

- Heuristic and Exact Methods
- Introduction to Local Search
  - Initial Solutions
  - Cost Functions
  - Neighborhoods
  - Improvement Strategies
- Plateaus and Local Optima

## Algorithms

## Min-conflicts local search

- Local search on binary FD constraint solving.
- Uses the 1-assign neighborhood.
- Main loop of algorithm:
  1. Select a variable involved in a violated constraint.
  2. Choose a new value for the variable so that the number of violated constraints are minimized.
- Has been used to plan the usage of the Hubble space telescope.

## Min-conflicts Algorithm

```
for  $r \leftarrow 0, \dots, \text{MAX-TRIES}$  do  
   $v \leftarrow$  random assignment  
  for  $k \leftarrow 0, \dots, \text{MAX-MOVES}$  do  
    if  $f(v) = 0$  then  
      return “satisfied by  $v$ ”  
    end if  
     $v \leftarrow$  variable in  $N(v)$  that minimizes the violated constraints  
  end for  
return “no satisfying assignment found”
```

*$f$  uses the 1-assign neighborhood*

## Hybrid Approaches

Hybridizations of local search and systematic techniques aim to overcome the drawbacks with both techniques. Several approaches:

- Use heuristics to guide labeling in systematic techniques.
- Use propagation as a first step to reduce domains.
- Use labeling and propagation on domains, replace backtracking with a local improvement phase.
- Use local search on a constraint system similar to CP.

## Informed Backtrack

- An *exact* method based on min-conflicts.
- Starts with an given assignment in which one or more constraints are violated.
- The method uses two sets:  $O$ , variables with possible conflicts and  $P$ , a consistent tuple.
- The algorithm tries to find a sequence of steps so that no variable is repaired more than once and all constraints are satisfied.

## Informed Backtrack Algorithm

```
procedure informed-backtrack( $O, P$ )  
  if all constraints are satisfied then succeed.  
   $x \leftarrow$  a variable in  $O$  that is in conflict  
  for all  $v \in D_x$  by ascending number of conflicts with  $O$  do  
    if  $(x, v)$  does not conflict with any  $w \in P$  then  
      assign  $v$  to  $x$   
      informed-backtrack( $O \setminus \{x\}, P \cup \{x\}$ )  
    end if  
  end for  
endproc
```

**procedure** main()

```
   $O \leftarrow$  all variables assigned an initial value  
  informed-backtrack( $O, \emptyset$ )  
endproc
```

## Drawbacks of Backtrack Search

CLP backtrack search and IB has a serious problem: *Every incorrect choice of value is fatal*. A tuple not part of a solution cannot be rejected until it is *proven exhaustively* that this is the case.

- Weak-commitment is based on min-conflicts.
- All variables are assigned an initial value.
- A consistent tuple  $P$  is constructed variable for variable.
- New values are selected according to the min-conflicts heuristic.
- If  $P$  cannot be extended, it is rejected and  $P$  is stored as a *nogood*. The algorithm then restarts.

## Weak-commitment algorithm

```

procedure wcs( $O, P, C$ )
  if all constraints in  $C$  are satisfied then succeed.
   $x \leftarrow$  a variable in  $O$  that is in conflict
   $V \leftarrow$  values from  $D_x$  satisfying all constraints together with  $P$ 
  if  $V = \emptyset$  then
    if  $P = \emptyset$  then fail.
    return wcs( $O \cup P, \emptyset, C \cup \{\text{nogood}(P)\}$ )
  else
     $v \leftarrow v' \in V$  that minimizes the number of conflicts with  $O$ 
    assign  $v$  to  $x$ 
    wcs( $O \setminus \{x\}, P \cup \{x\}, C$ )
  end if
endproc

procedure main()
   $O \leftarrow$  all variables assigned an initial value
   $C \leftarrow$  all constraints
  wcs( $O, \emptyset, C$ )
endproc

nogood( $\{(x_1, v_1), \dots, (x_n, v_n)\}$ )  $\equiv \neg(x_1 = v_1 \wedge \dots \wedge x_n = v_n)$ 

```

## Local Probing

(Kämäräinen and el Sakkout, 2002)

- Probe backtrack search with a *prober* based on local search.
- Problem are described with *easy* and *hard* constraints.
- Easy constraints = linear constraints, hard constraints = disjunctive or cumulative scheduling constraints.
- The prober satisfies all easy constraints, and tries to optimize a cost function with local search.
- If the hard constraints are not satisfied, post new easy ones that *guides* the prober so that the hard constraints can be satisfied.
- If the easy constraints cannot be satisfied, we use backtracking and reverse a posting of an easy constraint.