

# Introduction to metaheuristics for combinatorial optimization

Gilles Trombettoni

Université de Montpellier ; <http://www.lirmm.fr/~trombetton/cours/local.pdf>

**Goal:** Find a solution **satisfying** the constraints and which is **optimal** with respect to a given criterion.

## Two problems:

- 1 Constrained optimization (optimisation sous contraintes): see above
- 2 **Optimization: just minimize a criterion**

## Two big approaches:

- 1 Complete/exact/guaranteed algorithms
- 2 **Incomplete/inexact algorithms  
(heuristics or metaheuristics)**

**Definition:** the cost function (criterion) to be optimized.  
The objective function can be evaluated on a complete instantiation.

(An estimate of the objective function can often be given on a partial instantiation.)

Examples of objective functions:

- Scheduling problems: minimizing the due date of the latest task in the problem.
- Resource allocation problems: minimizing the number of resources
- Configuration or design problems: minimizing the price of production
- MAX-CSP: minimizing the number of violated constraints (or a weighted sum of violated constraints)

- 1 The Descent (or Hill climbing) algorithm
- 2 Existing mechanisms for improving the search
- 3 A generic (and didactic ;-) metaheuristic
- 4 The Simulated annealing algorithm
- 5 The Tabu search algorithm
- 6 Candidate list strategies, IDWalk
- 7 Genetic algorithms
- 8 The reparation heuristic for CSP
- 9 The GSAT algorithm for SAT
- 10 The WalkSAT algorithm for SAT
- 11 Synthesis... and what to do with all these methods

Optimization heuristics work on a **current solution (configuration)**: a point of the search space (actual solution or not).

**Neighborhood**: the set of configurations which can be obtained by a local transformation of the current configuration. Examples:

- Graph-Coloring: change a color
- SAT: “flip” of one boolean variable
- CSP: modification of one variable value ( $n(d - 1)$  neighbors)

**Evaluation of a configuration**: cost of the configuration: value to be minimized during search

**Local search**: improve a current configuration by iterative local transformations

## Definitions

Examples : Sudoku, graph coloring, many industrial problems

**Solving** the problem by **minimizing** the conflicts in Max-SAT (minimizing the number violated clauses) and Max-CSP (minimizing the number of violated constraints)

# Descent algorithm (algo de descente)

Also called *hill climbing* or greedy algorithm

## Initial configuration

- random: any point in the search space, or
- given by a deterministic (greedy) algorithm: a not too bad initial point should lead to a good configuration

## Local search

While a halt criterion is not fulfilled and a better configuration is found do:

- 1 Search a better configuration (or the best one) in the neighboring of the current configuration.
- 2 Change the current configuration to the selected neighbor (if any).

## Drawback

The descent algorithm finds a **local optimum**.

- **Incompleteness:** the search is not systematic (all the possibilities are not tried)  $\implies$   
no guarantee that the best solution has been found  
(no proof of the best solution)
- **Local optimum:** a local search may be blocked within a local optimum.  
It may be blocked on a plateau and sometimes visits several times the same configurations.
- **Sensitivity to the initial configuration**



# Several improvements

Most of the following improvements aim at avoiding local minima and plateaus. More generally, they follow an **Intensification/Diversification** mechanism.

- Interrupt the current search and try again with other initial configurations  
⇒ **GSAT** (random initial configurations) or other **multi-start** strategies with selected relaunch points.
- Manage several configurations in parallel (a “population” of configurations)  
⇒ **genetic algorithms**, **GWW** (and ant and bees metaheuristics).
- Record the latest moves to avoid looping on the same configurations  
⇒ **tabu search (TS)**
- Sometimes accept a configuration which gives a worse configuration  
⇒ **simulated annealing (SA)**, threshold accepting (TA)
- Use only the neighbors management to intensify or diversify the search: candidate list strategies (CLS), including **IDWalk**.

# A generic metaheuristic

**Goal:** design (and explain) most of existing optimization heuristics

**Parameters:**

- **Max-Tries:** number of times a local search is performed (from different initial configurations)
- **Max-Moves:** maximum number of steps in every walk
- **Accepted?(x, x' : configurations) : boolean**  
The function checking whether the neighbor  $x'$  of  $x$  is an acceptable move
- **Max-Neighbors:** maximum number of neighbors which are visited for any move
- **Min-Neighbors:** minimum number of neighbors which are visited for any move
- **No-Acceptation:** value taken into account when no neighbor has been accepted (among **Max-Neighbors** ones). Can either be equal to:
  - **no-move:** no new neighbor is selected (and the current walk is stopped)
  - **one-neighbor:** any visited neighbor is selected (e.g. the last one)
  - **best-neighbor:** a “less bad neighbor” is selected

# A generic metaheuristic

**Algorithm GenericMetaheuristic(...)**    **Returns:** *a configuration*

```
best  $\leftarrow \perp$ 
for  $i=1$  to Max-Tries do
     $x \leftarrow$  Initial-Configuration(...)
     $j \leftarrow 0$ 
    best-walk  $\leftarrow \perp$ 
    while  $j <$  Max-Moves do
         $x \leftarrow$  Generic-Move( $x$ )
        best-walk  $\leftarrow$  Minimum(best-walk,  $x$ )
    end
    best  $\leftarrow$  Minimum(best, best-walk)
end
return best
end.
```

# Acceptation of a move

```
Algorithm Generic-Move(cur : current configuration) Returns: neighbor configuration
  i  $\leftarrow$  0
  best?  $\leftarrow$  (Min-Neighbors > 1) or (No-Acceptation=best-neighbor)
  best-cost  $\leftarrow$   $+\infty$ ; x-best  $\leftarrow$  cur; x  $\leftarrow$  cur; accepted?  $\leftarrow$  false
  while (i < Min-Neighbors) or (i < Max-Neighbors and not(accepted?))
  do
    x  $\leftarrow$  Generate-Neighbor(init)
    if Accepted?(cur, x) then accepted?  $\leftarrow$  true
    if best? and (cost(x) < best-cost) then
      | x-best  $\leftarrow$  x
      | best-cost  $\leftarrow$  cost(x)
    end
    i  $\leftarrow$  i + 1
  end
  if accepted? then
    if best? then
      | return x-best
    else
      | return x
    end
  end
  if No-Acceptation=best-neighbor then return x-best
  if No-Acceptation=one-neighbor then return x
  if No-Acceptation=no-move then return cur
end.
```

En français : algorithme du *recuit simulé*  
One of the oldest local search algorithms

## Deduced from Generic-Move:

- `Generate-Neighbor(cur)`: any neighbor (selected randomly)
- `Min-Neighbors` = number of neighbors  
(variant: `Min-Neighbors`=`Max-Neighbors`)
- `No-Acceptation` = no-move (+ interruption of the walk)
- `Accepted?(cur, x)` :  $\text{cost}(x) \leq \text{cost}(cur)$  **or**  
 $\text{Random}() < \exp(\frac{-\Delta}{T})$

# Simulated Annealing (SA) method

```
Algorithm SA-Move(cur: current configuration, T: a temperature,  
Min-Neighbors : number of neighbors) Returns: neighbor configuration  
    best-cost  $\leftarrow +\infty$ ; x-best  $\leftarrow cur$   
    i  $\leftarrow 0$ ; accepted?  $\leftarrow$  false  
    while (i < Min-Neighbors) do  
        x  $\leftarrow$  Generate-Neighbor(cur)  
        if cost(x)  $\leq$  cost(cur) or Random() <  $\exp(\frac{-\Delta}{T})$  then  
            | accepted?  $\leftarrow$  true  
        end  
        if cost(x) < best-cost then  
            | x-best  $\leftarrow x$   
            | best-cost  $\leftarrow$  cost(x)  
        end  
        i  $\leftarrow i + 1$   
    end  
    if accepted? then  
        | return x-best  
    else  
        | return cur  
    end  
end.
```

# Simulated Annealing (SA) method

The important parameter is the **temperature**  $T$ , inspired from the physical phenomenon of *annealing*:

- $\Delta$  represents the degree of deterioration of the criterion, e.g., the additional number of violated constraints in MAX-CSP.
- A high temperature  $T$  allows the algorithm to escape from local minima.
- A low temperature makes the algorithm a greedy algorithm.
- The temperature should smoothly decrease (annealing process):  
Theoretical *convergence* with a infinitely smooth decrease.  
*Variant*: the Metropolis algorithm with a constant temperature.

# The Tabu Search (TS) method

**Idea:** Management of a *tabu list*: list of constant length  $L$  which records the  $L$  latest moves (FIFO). For CSPs, a move  $x' - x$  in the tabu list is the modified variable (the value is not stored). The tabu list avoids looking several times at the same configuration.

## Definition:

- `Min-Neighbors = number of neighbors`  
(variant: `Min-Neighbors = Max-Neighbors`)
- `No-Acceptation = no-move` (+ interruption of the walk)
- `Accepted?(cur, x, tabu-list) : x - cur  $\notin$  tabu-list`  
or  $x$  has the best cost ever found (*aspiration*).

An important parameter is the length  $L$  of the tabu list.  
Question: what allows the algorithm to escape from local minima?



# The Tabu Search (TS) method

**Algorithm TS-Move**(*cur*: current configuration, in-out tabu-list, *L*: max length of tabu list, Min-Neighbors: number of neighbors) **Returns**: neighbor configuration

```
best-cost  $\leftarrow +\infty$ ; x-best  $\leftarrow$  cur
i  $\leftarrow$  0; accepted?  $\leftarrow$  false
while (i < Min-Neighbors) do
    x  $\leftarrow$  Generate-Neighbor(cur)
    if x - cur  $\notin$  tabu-list or cost(x) < best-cost (aspiration) then
        | accepted?  $\leftarrow$  true
    end
    if cost(x) < best-cost then
        | x-best  $\leftarrow$  x
        | best-cost  $\leftarrow$  cost(x)
    end
    i  $\leftarrow$  i + 1
end
if accepted? then
    | tabu-list.PushEnd(x-best)
    | if (size(tabu-list) > L) then tabu-list.PopFirst()
    | return x-best
else
    | return cur
end
end.
```

# Variants of the Tabu Search

Fred Glover has introduced a lot of mechanisms in the tabu search schema for solving miscellaneous operational research problems. Two variants:

- *Probabilistic tabu*: a probability of acceptance is associated to every neighbor (the sum equals to 1). The value depends on when a move has been pushed in the list, the quality of the neighbor, and so on.
- Dynamic tabu list: the length  $L$  of the tabu list is modified in the course of time:
  - $L$  follows a pattern (e.g., a sinosoid) modifying the ratio Intensification vs Diversification during time: *strategic oscillation*.
  - $L$  is *adaptive*, that is, changes according the difficulty to improve the current configuration.

# Candidate list strategies and IdWalk

**Principle:** Take a lot of care in the analysis of the candidates (neighbors) for the next move: encode most of the local search mechanisms (such as Intensification vs Diversification) inside the function **Generic-move!**

Example: the **Intensification Diversification Walk (IDW)** algorithm:

**Definition:**

- Accepted?(*cur*, *x*) :  $cost(x) \leq cost(cur)$  (greedy component: Intensification)
- Min-Neighbors = 0
- No-Acceptation = one-move or best-move (random component: Diversification)
- Main parameter to be tuned: Max-Neighbors with 3 roles:
  - limits the number of explored neighbors,
  - must be sufficiently large for intensifying the search,
  - must be sufficiently small for diversifying the search (with No-Acceptation).

**Algorithm IDW-Move**(*cur*: current configuration, Max-Neighbor: number of neighbors) **Returns:** neighbor configuration

```

    best-cost  $\leftarrow +\infty$  ; x-best  $\leftarrow$  cur
    i  $\leftarrow$  0 ; accepted?  $\leftarrow$  false
    while (i < Max-Neighbors and not(accepted?)) do
        x  $\leftarrow$  Generate-Neighbor(cur)
        if cost(x)  $\leq$  cost(cur) then
            | accepted?  $\leftarrow$  true
        end
        if cost(x) < best-cost then
            | x-best  $\leftarrow$  x
            | best-cost  $\leftarrow$  cost(x)
        end
        i  $\leftarrow$  i + 1
    end
    if accepted? then
        | return x
    else
        | return x-best /* Variant: return x */
    end
end.
```

# Genetic algorithms: guidelines

Management of a **population** of configurations, called **individuals**

## Algorithm *GA-schema*

```
while no satisfactory individuals in the population do  
    Select individuals in the population for reproduction  
    Apply different reproduction operators on selected  
    individuals:  
        • mutation: generation of a neighbor of an individual  
        • crossover: mixing two configurations (individuals)  
          to generate a new individual  
end  
Selection: keep a sub-set of the new population  
    (natural selection)  
end.
```

**Encoding:** an individual is made of a chromosome: a sequence of bits

# Another population algorithm: Go With the Winners

GWW manages several configurations (called *particles*) and a threshold (in French: *seuil*).

**Initialization:**  $B$  particles are randomly distributed; a threshold is placed at the cost of the worst particle.

**Main loop:** *repeat until no particle remains under or at the threshold:*

- 1 **Redistribution:** (bad) particles over the threshold are “redistributed”: a redistributed particle is replaced by a copy of another particle (under the threshold; randomly chosen).
- 2 **Randomization:** A random walk of length  $S$  is performed: every step of the walk moves a particle to a neighbor *which remains at or under the threshold*.
- 3 Lower the threshold value by 1.

The population is a set of configurations.

Evaluation function of an individual: number of conflicts, weighted sum of conflicts...

**Difficulty:** the crossover operator is not relevant  
⇒ difficult to generate a better individual.

- The crossover point does not take the number of conflicts into account.
- A chromosome loses the topology of the constraint system (This drawback is also true for most of the non-structured combinatorial problems.)

# Heuristic reparation by Minton (for CSPs)

**Initialization:** greedy instantiation of variables: iteratively choose the variable which produces a smallest number of conflicts with previous variables.

**Reparation:** While *there is a conflict* do:

- choose a variable  $x$  whose value gives at least one conflict
- change the value of  $x$  to a new value which minimizes the number of conflicts

Min-Neighbors? No-Acceptation  
? best-neighbor? Accepted? ? ...?



Initially developed for SAT (satisfiability of a boolean formula)

**Definition:** several trials of the descent algorithm

- Neighborhood: the  $n$  configurations where a single variable is “flipped”.
- `Max-Tries`  $\gg 1$
- `Max-Moves` is limited
- `Max-Neighbors` = number of neighbors
- `Min-Neighbors` = `Max-Neighbors`
- `No-Acceptation` = best-move  
(or no-move + interruption of the walk)
- `Accepted?( $x, x'$ )` :  $\text{cost}(x') \leq \text{cost}(x)$

`Max-Tries` allows GSAT to find several local minima, one being maybe a global minimum

Choose an **unsatisfied** clause  $C$

(Flipping any variable in  $C$  will at least fix  $C$ )

Compute a “break score”: for each var  $v$  in  $C$ , flipping  $v$  would break how many other clauses?

- 1 If  $C$  has any vars with break score 0:  
Pick at random among the vars with break score 0
- 2 else, with probability  $p$ :
  - Pick at random among the variables with minimum break score
  - else: Pick at random among all variables in  $C$  (large diversification)

# Example of experiments

le15c, le25c and flat28 are graph coloring instances (encoded as MAX-CSP instances); celar6, celar7, celar8 are radio link frequency assignment instances.

An entry contains the average cost (over 10 or 20 trials). The best cost over the 10 or 20 trials appears into parentheses.

The neighborhood definition is crucial! (not detailed here)

	le15c	le25c	flat28	celar6	celar7	celar8
# colors	15	25	31			
Time/trial	2 min	14 min	9 min	14 min	6 min	50 min
Metrop.	5.9 (2)	<b>3.1</b> (2)	0.9 ( <b>0</b> )	5048 (3906)	$6 \cdot 10^6$ ( $2.9 \cdot 10^6$ )	410 (300)
SA	9.6 ( <b>0</b> )	5.8 (4)	1.8 ( <b>0</b> )	4167 (3539)	$1.2 \cdot 10^6$ (456893)	281 (264)
Tabu	1.5 ( <b>0</b> )	3.7 (3)	2.5 (1)	4183 (3935)	$1.2 \cdot 10^6$ (620159)	373 (315)
IDWalk	0.5 ( <b>0</b> )	<b>3.1</b> ( <b>1</b> )	<b>0.8</b> ( <b>0</b> )	3447 ( <b>3389</b> )	373334 (343998)	291 (273)
GWW	536 (410)	17.1 (14)	6.6 (6)	3648 (3427)	583278 (456968)	276 (265)
GWW-idw	<b>0</b> ( <b>0</b> )	4 (3)	1.3 ( <b>0</b> )	<b>3405</b> ( <b>3389</b> )	<b>368452</b> ( <b>343600</b> )	<b>267</b> ( <b>262</b> )

Optimization heuristics are incomplete, but are less sensitive to bad choices than exact methods.

What is important if one wants to find, with local search, a good solution to a given optimization problem:

- 1 Try several ways of encoding the problem (definition of neighboring).
- 2 Understand the intuitions (i.e., the useful mechanisms) behind the main optimization heuristics.
- 3 Be pragmatic, have no strong belief: Try different heuristics, instead of tuning only one: use a library!
- 4 Favor the simplest ideas because they are faster to understand and experiment.
- 5 A fine tuning of the parameters generally does not pay.
- 6 Favor adaptive parameters.