8. Heuristic and Approximation Algorithms

- CSP versus MCSP
- Local Search Techniques

CSP versus MCSP

- A **Constraint Satisfaction Problem** (CSP) consist of:
  - a set of variables $X = \{x_1, \ldots, x_n\}$,
  - for each variable x_i, a finite set D_i of possible values (its domain),
  - and a set of constraints restricting the values that the variables can simultaneously take.
- A **solution to a CSP** is an assignment of a value from its domain to every variable, in such a way that every constraint is satisfied. We may want to find:
  - just one solution, with no preference as to which one,
  - all solutions,
  - an optimal, or at least a good solution, given some objective function defined in terms of some or all of the variables.
- The **Maximal Constraint Satisfaction Problem** is an optimization problem consisting in looking for an assignment that satisfy the maximal number of constraints.

Resolution Techniques

- CSP
  - Exact methods : local consistency + backtrack search.
- MCSP
  - Exact methods : Partial Constraint Satisfaction Techniques based on the branch and bound algorithm.
  - Approximation methods : Local search
    * Hill-Climbing
    * Min-Conflicts
    * Min-Conflicts-Random-Walk
    * Steepest-Descent-Random-Walk
    * Tabu-Search
    * GSAT

Partial Constraint Satisfaction Techniques

- Perform Direct Arc Consistency Algorithms in the pre-processing phase.
  - Count the number of inconsistencies counts associated with each variable value.
- Backtrack-Search based on Branch and Bound algorithm.
  - The cost function corresponds to the number of violated constraints.

Hill-Climbing

```
procedure hill-climbing(Max_Flips)
   restart: s <- random valuation of variables;
   for j:=1 to Max_Flips do
     if eval(s)=0 then return s endif;
     if s is a strict local minimum then
         goto restart
     else
       s <- neighborhood with smallest evaluation value
     endif
    endfor
    goto restart
   end hill-climbing
```

Hill-Climbing

Disadvantage of the method

- The hill-climbing algorithm has to explore all neighbors of the current state before choosing the move. This can take a lot of time.

Min-Conflicts

- To avoid exploring all neighbors of the current state some heuristics were proposed to find a next move.
- Min-conflicts heuristics chooses randomly any conflicting variable, i.e., the variable that is involved in any unsatisfied constraint, and then picks a value which minimizes the number of violated constraints (break ties randomly). If no such value exists, it picks randomly one value that does not increase the number of violated constraints (the current value of the variable is picked only if all the other values increase the number of violated constraints).

Min-Conflicts

```
procedure MC(Max_Moves)
 s <- random valuation of variables;
 nb_moves <- 0;
 while eval(s)>0 & nb_moves<Max_Moves do
   choose randomly a variable V in conflict;
   choose a value v’ that minimizes the
           number of conflicts for V;
   if v’ # current value of V then
     assign v’ to V;
     nb_moves <- nb_moves+1;
   endif
 endwhile
  return s
end MC
```

Min-Conflicts

Disadvantage of the method

- The pure min-conflicts algorithm presented above is not able to leave local-minimum. In addition, if the algorithm achieves a strict local-minimum it does not perform any move at all and, consequently, it does not terminate.

Min-Conflicts-Random-Walk

- Because the pure min-conflicts algorithm cannot go beyond a local-minimum, some noise strategies were introduced in MC. Among them, the random-walk strategy becomes one of the most popular.

- For a given conflicting variable, the random-walk strategy picks randomly a value with probability p, and apply the MC heuristic with probability 1-p.

Min-Conflicts-Random-Walk

```
procedure MCRW(Max_Moves,p)
  s <- random valuation of variables;
  nb_moves <- 0;
  while eval(s)>0 & nb_moves<Max_Moves do
    if probability p verified then
     choose randomly a variable V in conflict;
     choose randomly a value v' for V;
    else
     choose randomly a variable V in conflict;
     choose a value v' that minimizes the number of conflicts for V;
    endif
    if v' # current value of V then
      assign v' to V;
      nb_moves <- nb_moves+1;
    endif
  endwhile
  return s
end MCRW
```

Min-Conflicts-Random-Walk

This algorithm is controlled by the random probability p, it should be clear that the value for this parameter has a big influence on the performance of the algorithm. The preliminary studies determined the following feasible ranges of parameter values $0.02 \leq p \leq 0.1$.

Steepest-Descent-Random-Walk

- Stepest-Descent algorithm is a hill-climbing version of the min-conflicts algorithm. Instead of selecting the variable in conflict randomly, this algorithm explores the whole neighborhood of the current configuration and selects the best neighbor according to the evaluation value.
- Again, the algorithm can be randomized by using random-walk strategy to avoid getting stuck at ”local optima”.

Steepest-Descent-Random-Walk

```
procedure SDRW(Max_Moves,p)
  s <- random valuation of variables;
  nb_moves <- 0;
  while eval(s)>0 & nb_moves<Max_Moves do
    if probability p verified then
      choose randomly a variable V in conflict;
      choose randomly a value v' for V;
    else
      choose a move <V,v'> with the best performance
    endif
    if v' # current value of V then
      assign v' to V;
      nb_moves <- nb_moves+1;
    endif
  endwhile
  return s
end SDRW
```

Tabu-Search

- Tabu search (TS) is another method to avoid cycling and getting trapped in local minimum. It is based on the notion of tabu list, that is a special short term memory that maintains a selective history, composed of previously encountered configurations or more generally pertinent attributes of such configurations.
- A simple TS strategy consist in preventing configurations of tabu list from being recognized for the next k iterations (k, called tabu tenue, is the size of tabu list).
- Such a strategy prevents Tabu from being trapped in short term cycling and allows the search process to go beyond local optima.
- Tabu restrictions may be overridden under certain conditions, called aspiration criteria.
- Aspiration criteria define rules that govern whether next configuration is considered as a possible move even it is tabu. One widely used aspiration criterion consists of removing a tabu classification from a move when the move leads to a solution better than that obtained so far.

Tabu-Search

```
procedure tabu-search(Max_Iter)
  s <- random valuation of variables;
  nb_iter <- 0;
  initialize randomly the tabu list;
  while eval(s)>0 & nb_iter<Max_Iter do
   choose a move <V,v'> with the best performance among the non-
    tabu  moves  and the moves satisfying the aspiration criteria;
    introduce <V,v> in the tabu list,
              where v is the current values of V;
    remove the oldest move from the tabu list;
    assign v' to V;
    nb_iter <- nb_iter+1;
  endwhile
  return s
end tabu-search
```

Tabu-Search

- The performance of Tabu Search is greatly influenced by the size of tabu list tl. A preliminary studies determined the following feasible range of parameter values $10 \leq tl \leq 35$.

GSAT

- GSAT is a greedy local search procedure for satisfying logic formulas in a conjunctive normal form (CNF). Such problems are called SAT or k-SAT (k is a number of literals in each clause of the formula) and are known to be NP-c (each NP-hard problem can be transformed to NP-complex problem).
- The procedure starts with an arbitrary instantiation of the problem variables and offers to reach the highest satisfaction degree by succession of small transformations called repairs or flips (flipping a variable is a changing its value).

GSAT

```
procedure GSAT(A,Max_Tries,Max_Flips)
  A: is a CNF formula
  for i:=1 to Max_Tries do
    S <- instantiation of variables
    for j:=1 to Max_Iter do
      if A satisfiable by S then
        return S
      endif
      V <- the variable whose flip yield the most important
       raise in the number of satisfied clauses;
      S <- S with V flipped;
   endfor
  endfor
  return the best instantiation found
end GSAT
```

GSAT

- Several heuristics have been implemented within GSAT in order to efficiently solve structured problems.

Random-Walk The implementation of random walk within the GSAT algorithm is similar to MCRW.

Clause weight This technic result from the observation that for some problems, several resolution attempts reach the same unsatisfied final set of clauses. So, each clause has not the same weight on the resolution, some clauses will be much harder to solve. The resolution process must offer more importance to these "hard" clauses. A way to deal with this kind of problems is to associate a weight to each clause, in order to modify its influence on the global score. Thanks to this weight heuristic, the participation of a satisfied "hard" clause is more important. Furthermore the weight can be automatically found using the following method:

1. initialize each clause weight to '1'
2. at the end of each tries, add '1' to each unsatisfied clause weight.

Averaging in previous near solutions After each attempt, GSAT restarts with a random initial problem variables. This heuristic offers to reuse parts of the best assignments issued from the two previous states. Therefore, the starting variables vector for i-th attempt is computed from the bitwise($*$) of the two best reached states during the attempts (i-2)-th and (i-1)-th.

($*$) : identical bits representing identical variable are reused, the other ones are randomly chosen.

Local Search

- All above algorithms are based on common idea known under the notion local search.
- In local search, an initial configuration (valuation of variables) is generated and the algorithm moves from the current configuration to a neighborhood configurations until a solution (decision problems) or a good solution (optimization problems) has been found or the resources available are exhausted.

Local Search

```
procedure local-search(Max_Moves,Max_Iter)
  s <- random valuation of variables;
  for i:=1 to Max_Tries while Gcondition do
    for j:=1 to Max_Iter while Lcondition do
      if eval(s)=0 then
        return s
      endif;
      select n in neighborhood(s);
      if acceptable(n) then
        s <- n
      end if
    endfor
    s <- restartState(s);
  endfor
  return s
end local-search
```