Parallel Cooperative Optimization Research Group

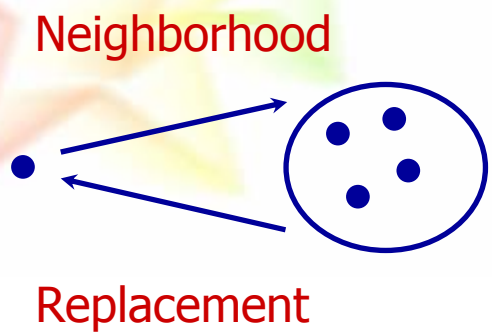
Single Solution-based Metaheuristics

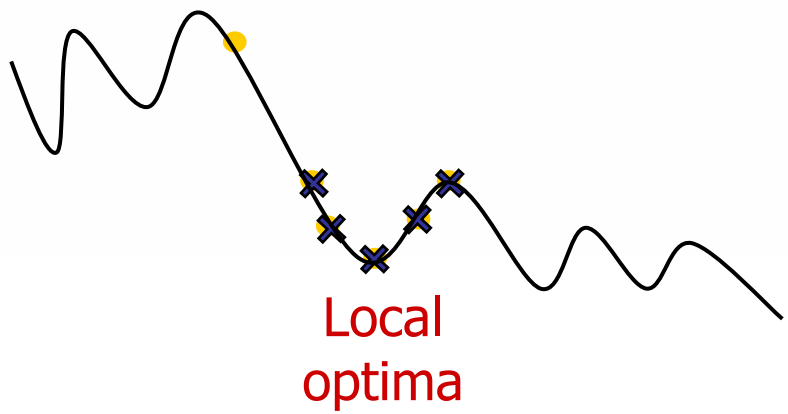
E-G. Talbi, S. Cahon

Laboratoire d'Informatique Fondamentale de Lille

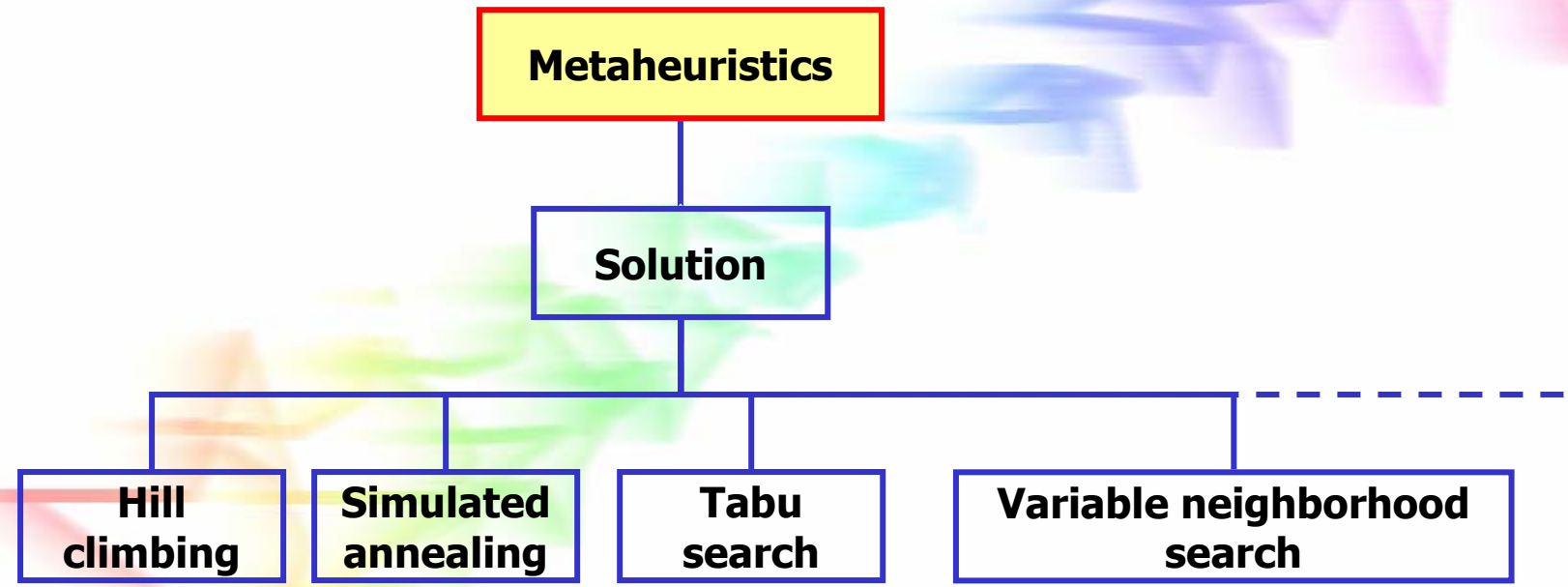
Solution-based metaheuristics

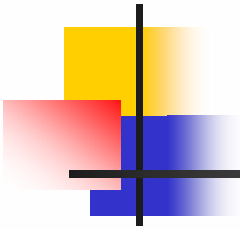
- "Improvement" of a solution
- Exploitation oriented
  - Based on the descent
  - Exploration of the neighborhood (intensification)





Taxonomy (Population-based metaheuristics)





Outline

- Hill Climbing
- Simulated Annealing
- Tabu Search
- Variable Neighborhood search

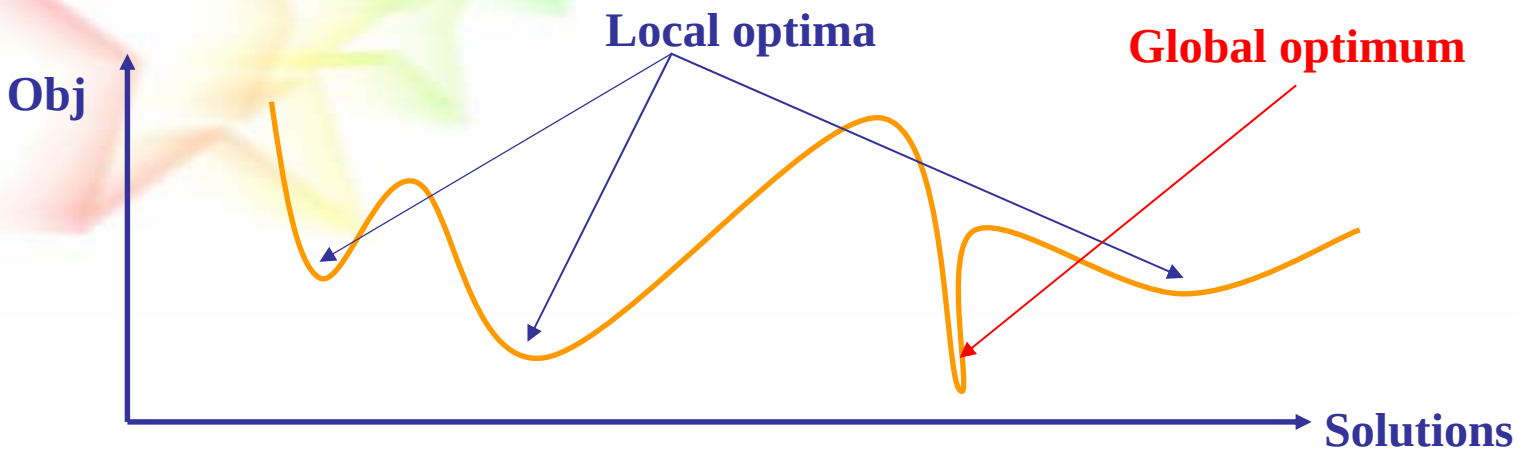
Hill Climbing

Outline

- Hill Climbing
  - Main characteristics,
  - Algorithm,
  - Trajectory guidance,
  - Efficient evaluation/application of moves
- Simulated Annealing
- Tabu Search
- Variable Neighborhood search

Hill Climbing

- Easy to implement,
- It only leads to local optima,
- The found optima depends on the initial solution,
- No mean to estimate the relative error from the global optimum,
- No mean to have an upper bound of the computation time,



Strategies for the selection of a neighbour

- Deterministic/full: choosing the best neighbor (i.e. that improves the most the cost function)
- Deterministic/partial: choosing the first processed neighbour that is better than the current solution,
- Stochastic/full: processing the whole neighborhood and applying a random better one
- …

Hill Climbing algorithm

Function HILL_CLIMBING (*Problem*)
returns a solution state

- **Inputs**: *Problem*, a problem
- **Local Variables**:
  - *Current*, a state
  - *Next*, a state
  - *Local*, a boolean

Hill Climbing algorithm

- *Current* = MAKE-NODE(INITIAL STATE[*Problem*])
- *Local = false*
- **Do**
  - *Next* = a neighbour of *Current,* selected according to a guidance strategy (det/stoch, full/partial, etc)
  - **If(***Next* is better than *Current***) then**
    - *Current* = *Next*
    - Local = false
  - **Else**
    - Local = true
- **Until not local**
- **Return** *Current*

Efficiency issues

- The evaluation function is calculated for every candidate neighbour,
- Often the cost function is the most expensive part of the algorithm
- Therefore,
  - We need to evaluate the cost function as efficiently as possible,

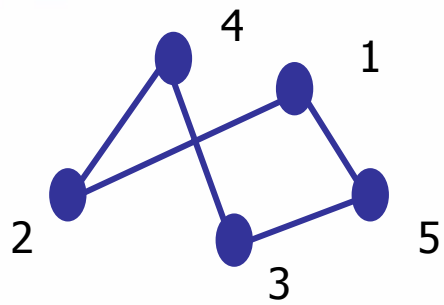
  → Use Incremental (Delta) Evaluation

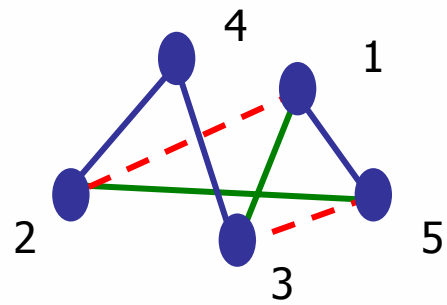
Application to TSP

- Example: “Two-opt”.
  - Two points within the permutation are selected and the segment between them is inverted. This operator put in two new edges in the tour

2	1	5	3	4

2	5	1	3	4





Delta = - d(2,1) – d (5,3) + d(2, 5) + d(1, 3)

Hill Climbing Pros & Cons

- Pros
  - Easy to implement and fast at execution
- Cons
  - Get stuck at local optima,
  - Possible solutions
    - Try several runs, starting at different positions,
    - Increase the size of the neighborhood
      - In TSP try 3-opt rather than 2-opt

Simulated Annealing

Outline

- Hill Climbing
- Simulated Annealing
  - Main features,
  - Criterion of move acceptation,
  - Cooling schedule.
- Tabu Search
- Variable Neighborhood search

Simulated Annealing

- Motivated by the physical annealing process
- Material is heated and slowly cooled into a uniform structure
- Simulated annealing mimics this process
- The first SA algorithm was developed in 1953 (Metropolis)
- Kirkpatrick (1982) applied SA to optimisation problems
  - Kirkpatrick, S , Gelatt, C.D., Vecchi, M.P. 1983. *Optimization by Simulated Annealing*. Science, vol 220, No. 4598, pp 671-680

Simulated Annealing

- Compared to hill climbing the main difference is that SA allows downwards steps,
- Simulated annealing also differs from hill climbing in that a move is selected at random and its acceptation is conditional
  - Yet, moves that improve the cost function are always accepted,

The decision criterion in accepting a given move

- The law of thermodynamics states that at temperature, *t*, the probability of an increase in energy of magnitude, δE, is given by

$$P(\delta E) = \exp(-\delta E / kt)$$

- Where *k* is a constant known as Boltzmann's constant

Accepting a move or not

$$P = \exp(-c/t) > r$$

- Where
  - c is change in the evaluation function
  - t the current temperature
  - r is a random number between 0 and 1
- Example

Change	Temp	exp(-C/T)		Change	Temp	exp(-C/T)
0.2	0.95	0.810157735		0.2	0.1	0.135335283
0.4	0.95	0.656355555		0.4	0.1	0.018315639
0.6	0.95	0.53175153		0.6	0.1	0.002478752
0.8	0.95	0.430802615		0.8	0.1	0.000335463

Accepting a move or not

- The probability of accepting a worse state is a function of both the temperature of the system and the change in the cost function
- As the temperature decreases, the probability of accepting worse moves decreases
- If t=0, no worse moves are accepted (i.e. hill climbing)

Simulated Annealing algorithm

Function SIMUL_ANNEALING(*Problem*) **returns** a solution state

- **Inputs**:
  - *Problem*, a problem
  - *Schedule*, a mapping from time to temperature
- **Local Variables**:
  - *Current*, a node
  - *Next*, a node
  - *T*, a "temperature" controlling the probability of downward steps
  - *num_steps*, giving the number of moves performed for a given temperature value

Simulated Annealing algorithm

- $T = T_{init}$
- *Current* = MAKE-NODE(INITIAL-STATE[*Problem*])
- **Do**
  - **For** k from 1 to num_steps do
    - *Next* = a randomly selected successor of *Current*
    - ΛE = VALUE[*Next*] – VALUE[*Current*]
    - **If** $\Lambda E > 0$ **then** *Current* = *Next*
    - **else** *Current* = *Next* only with probability $\exp(-\Lambda E/T)$
  - **Done**
  - $T = Schedule[T]$
- **Until** $T < T_{min}$
- **Return** Current

The cooling schedule

- The cooling schedule is *hidden* in this algorithm - but it is important (more later)
- The algorithm assumes that annealing will continue until temperature is zero - this is not necessarily the case
  - Starting temperature,
  - Final temperature,
  - Temperature update,
  - Iterations at each temperature

Starting temperature

- It must be *hot* enough to allow moves to *almost* neighbourhood state (else we are in danger of implementing hill climbing)
- Must *not* be so hot that we conduct a random search for a period of time
- Problem is finding a suitable starting temperature

Choosing the starting temperature

- If we know the maximum change in the cost function we can use this to estimate,
- Start high, reduce quickly until about 60% of worse moves are accepted. Use this as the starting temperature,
- Heat rapidly until a certain percentage are accepted the start cooling

Choosing the final temperature

- It is usual to let the temperature decrease until it reaches zero. However, this can make the algorithm run for a lot longer, especially when a geometric cooling schedule is being used,
- In practice, it is not necessary to let the temperature reach zero because the chances of accepting a worse move are almost the same as the temperature being equal to zero,
- Therefore, the stopping criteria can either be a suitably low temperature or when the system is “frozen” at the current temperature (i.e. no better or worse moves are being accepted)

Choosing the temperature decrement

- Theory states that we should allow enough iterations at each temperature so that the system stabilises at that temperature,
- Unfortunately, theory also states that the number of iterations at each temperature to achieve this might be exponential to the problem size,
- We need to compromise
- We can either do this by doing a large number of iterations at a few temperatures, a small number of iterations at many temperatures or a balance between the two

Choosing the temperature decrement

- Decrement function in general :

  $temp = temp * a$

  - Experience has shown that a should be between 0.8 and 0.99, with better results being found in the higher end of the range. Of course, the higher the value of a, the longer it will take to decrement the temperature to the stopping criterion.

Iterations at each temperature

- A constant number of iterations at each temperature
- Another method, first suggested by (Lundy, 1986) is to only do one iteration at each temperature, but to decrease the temperature *very* slowly.
  - The formula used is $t = t/(1 + \beta t)$ where β is a suitably small value

Iterations at each temperature

- An alternative is to dynamically change the number of iterations as the algorithm progresses
- At lower temperatures it is important that a large number of iterations are done so that the local optimum can be fully explored
- At higher temperatures, the number of iterations can be less

References

- E. Aarts, J. Korst, « Simulated Annealing and Boltzmann Machines », *John Wiley, New York, 1989.*
- E.H.L. Aarts, J.K. Lenstra (eds), « Local Search in Combinatorial Optimization », *John Wiley, Chichester, 1997.*
- M. Duflo, « Random Iterative Models », *Springer-Verlag, Berlin, 1997.*
- M. Johnson (ed), « Simulated Annealing and Optimization », *American Sciences Press, Columbus, OH, 1988.*
- P.J.M. Van Laarhoven, E.H.L. Aarts, « Simulated Annealing: Theory and Applications », *D. Reidel, Dordrecht, 1987.*

Tabu Search

Outline

- Hill Climbing
- Simulated Annealing
- Tabu Search
  - Main features
  - Management of tabu moves/solutions
  - Aspiration criterion
  - Example
- Variable Neighborhood search

Tabu Search

Proposed independently by Glover (1986) and Hansen (1986)

- "a meta-heuristic superimposed on another heuristic. The overall approach is to avoid entrapment in cycles by forbidding or penalizing moves which take the solution, in the next iteration, to points in the solution space previously visited (hence *tabu*)."

Main characteristics

- It behaves like a steepest ascent Hill Climbing algorithm
- But it accepts non-improving solutions in order to escape from local optima (where all the neighbouring solutions are non-improving)
- Deterministic algorithm

Main characteristics

- After exploring the neighbouring solutions, we accept the best one even if it decreases the cost function

  $\rightarrow$ A worse move may be accepted

- Two goals in the use of memory
  - prevent the search from revisiting previously visited solutions
  - explore the unvisited areas of the solution space

Main characteristics

- This metaheuristic uses past experiences to improve current decision making
- By using memory (a “tabu list”) to prohibit certain moves – TS is classified a global optimizer rather than a local optimizer

Issues in the management of Tabu solutions/moves

- We don't want to re-visit exact the same solution that we've been *before*,
- We may want to discourage some patterns in solution
  - in TSP problem, tabu a state that has the towns listed in the same order that we've seen before
  - If the size of problem is large, a lot of time is wasted just checking if a certain state has been reached before

Issues in the management of Tabu solutions/moves

- We don't want to return to the state that the search has come from
- Efficiency/tractability
  - Remembering a few solutions
  - Smaller data structure in tabu list
    - In TSP problem, tabu the two towns just been considered in the last move (two-opt, city-swap)

Issues in the management of Tabu solutions/moves

- In Tabu Search
  - What neighbourhood is
  - What constitute a tabu list (strongly dependent of the concept of neighbourhood)

Issues related to memory

- Efficient usage of memory resources
  - Design of efficient data structures to record and access the recorded data efficiently
    - Hash table, Binary tree, etc
- Discarding information which should not be remembered
- Just collecting the relevant and necessary information
  - Clear understanding of which attributes of solutions are crucial
  - Limited selection of attributes of solutions to be memorised
  - Clear strategy on usage of information or their disposal when not needed

Tabu Search algorithm

Function TABU_SEARCH(*Problem*) **returns** a solution state

- **Inputs**: *Problem*, a problem
- **Local Variables**:
  - *Current*, a state
  - *Next*, a state
  - *BestSolutionSeen*, a state
  - *H*, a history of visited states

Tabu Search algorithm

- *Current* = MAKE-NODE(INITIAL-STATE[*Problem*])
- **While not terminate**
  - *Next* = a highest-valued successor of *Current*
  - **If(not** Move_Tabu(*H,Next*) **or** Aspiration(*Next*)**) then**
    - *Current* = *Next*
    - Update *BestSolutionSeen*
    - *H* = Recency(*H* + *Current)*
  - Endif
- **End-While**
- **Return** *BestSolutionSeen*

Features of Tabu search

- Memory related - recency (How recent the solution has been reached)
  - Tabu List (short term memory): to record a limited number of attributes of solutions (moves, selections, assignments, etc) to be discouraged in order to prevent revisiting a visited solution
  - Tabu tenure (length of tabu list): number of iterations a tabu move is considered to remain tabu

Features of Tabu Search

- Memory related – recency (How recent the solution has been reached)
  - Tabu tenure
    - List of moves does not grow forever – restrict the search too much
    - Restrict the size of list
    - FIFO
    - Other ways: dynamic

Features of Tabu Search

- Memory related – frequency
  - Long term memory: to record attributes of elite solutions to be used in:
    - Diversification: Discouraging attributes of elite solutions in selection functions in order to diversify the search to other areas of solution space;
    - Intensification: giving priority to attributes of a set of elite solutions (usually in weighted probability manner)

Features of Tabu Search

- If a move is good, but it's tabu-ed, do we still reject it?
- Aspiration criteria: accepting an improving solution even if generated by a tabu move
  - Similar to SA in always accepting improving solutions, but accepting non-improving ones when there is no improving solution in the neighbourhood;

Example: TSP using Tabu Search

In our example of TSP:

	V_1	V_2	V_3	V_4
V_1	X			
V_2		X		
V_3			X	
V_4				X

- Short term memory:
  - Maintain a list of *t* edges and prevent them from being selected for consideration of moves for a number of iterations;
  - After a number of iterations, release those edges
  - Make use of a squared table to decide if an edge is tabu or not

Example: TSP using Tabu Search

In our example of TSP:

- Long term memory:
  - Maintain a list of *t* edges which have been considered in the last *k* best (worst) solutions
  - encourage (or discourage) their selections in future solutions
  - using their frequency of appearance in the set of elite solutions and the quality of solutions which they have appeared in our selection function

Example: TSP using Tabu Search

In our example of TSP:

- Aspiration:
  - If the next moves consider those moves in the tabu list but generate better solution than the current one
  - Accept that solution anyway
  - Put it into tabu list

Tabu Search Pros & Cons

- Pros
  - Generate generally good solutions for optimisation problems compared with other solution based metaheuristics
- Cons
  - Tabu list construction is problem specific
  - Long terme memory is problem specific
  - Different parameters ...

References

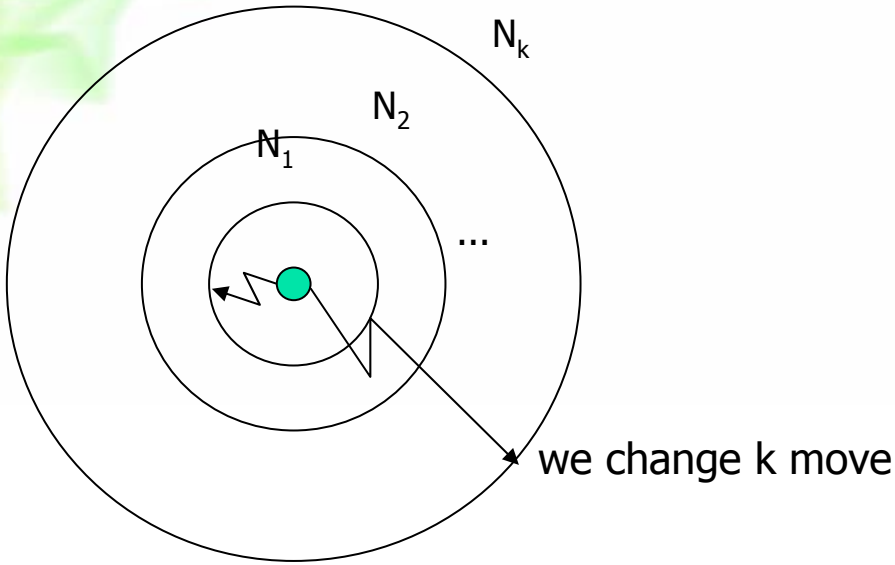
- Glover, F. 1989. *Tabu Search – Part I*. ORSA Journal on Computing, Vol. 1, No. 3, pp 190-206.
- Glover, F. 1990. *Tabu Search – Part II*. ORSA Journal on Computing, Vol. 2, No. 1, pp 4-32.
- Glover, F., Laguna, M. 1998. *Tabu Search*. Kluwer Academic Publishers
- Rayward-Smith, V.J., Osman, I.H., Reeves, C.R., Smith, G.D. 1996*. Modern Heuristic Search Methods*, John Wiley & Sons.
- Russell, S., Norvig, P. 1995. *Artificial Intelligence A Modern Approach*. Prentice-Hall
- R. Qu, "Artificial Intelligence methods: Tabu search"

Variable Neighborhood-search

- Inspired from the Hill Climbing
- But manages several neighborhoods (hence different move concepts)

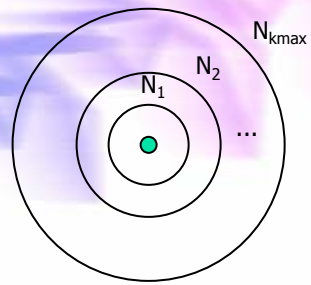
Variable Neighborhood-search

- Definitions:
  - Nk → Neighborhood in k distance
    - Distance: how many locations we change
  - Nk(x) → Set of solutions in the kth neighbourhood of x

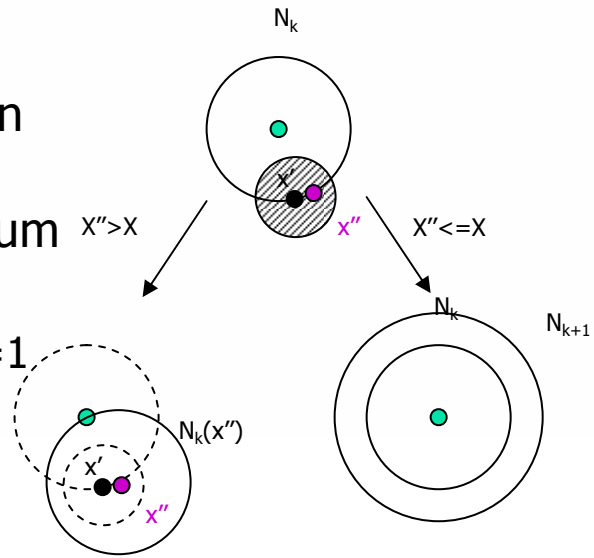


Variable Neighborhood-search

- **Initialisation**
- Select the set of neighbourhood structures Nk
- Find an initial solution x



- **Repeat** until stopping condition is met
  - Set K=1
  - **Repeat** until k=kmax
    - *Shaking*: Generate a random point X' in Nk(x)
    - *Local Search*: x" is the obtained optimum
    - Move or not:
      - If x" is better than x then x=x" and k=1
      - Otherwise k=k+1



Variations of Variable Neighborhood Search

- Reduced Variable Neighborhood Search (RVNS)
  - We omit the local search step
    - ADV: we accelerate drastically the VNS
    - DISV: we deteriorate the best solution
- Others variations:
  - Fast Interchange: Search candidates for deleting and adding in a faster way (Hansen, Mladenovic, Perez-Brito 2001)
  - We can start with a good solution (not random)