Metaheuristic Optimization Methods

Simulação e Otimização

Mestrado em Engenharia Informática
Mestrado em Robótica e Sistemas Inteligentes

Amaro de Sousa, Nuno Lau

DETI-UA, 2024/2025

Metaheuristic Optimization Methods
Organization of Classes

CLASS 1

- Introduction to optimization: selection of optimal nodes in a network

CLASS 2

- Basic definitions of optimization methods
- Single-solution based metaheuristic methods

CLASS 3

- Multi-start single-solution based metaheuristic methods

CLASS 4

- Population-based metaheuristic methods

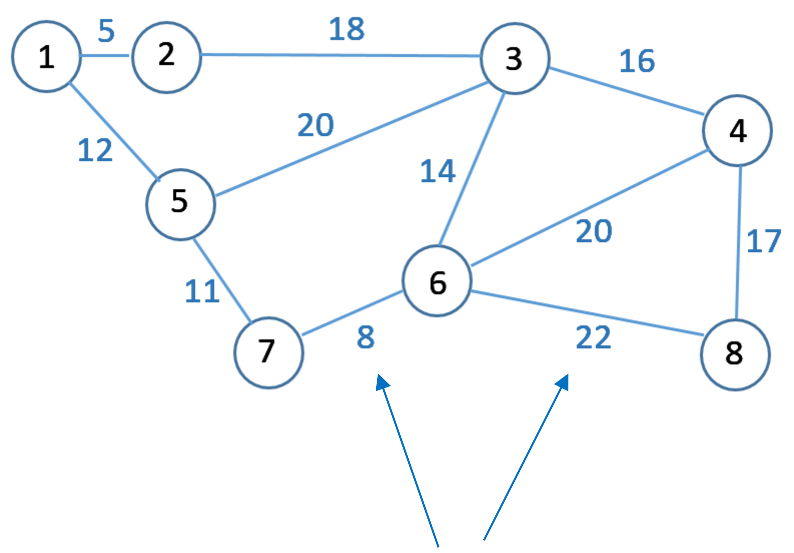
CLASS 1

Introduction to optimization: selection of optimal nodes in a network

A Network defined by a Directed Graph

Consider a network defined by a graph $G = (N,A)$:

- Set N of nodes identified as $i = 1, 2, \ldots, |N|$
- Set A of arcs identified as (i,j), with $i \in N$ and $j \in N$
- Each arc $(i,j) \in A$ with an associated parameter l_{ij} (for example, representing the length of the arc)



Arc lengths l_{ij}
(unless stated otherwise: $l_{ij} = l_{ji}$)

$N = \{1,2,3,4,5,6,7,8\}$
$|N| = 8$

$A = \{(1,2), (2,1), (1,5), (5,1),$
$(2,3), (3,2), (3,4), (4,3),$
$(3,5), (5,3), (3,6), (6,3),$
$(4,6), (6,4), (4,8), (8,4),$
$(5,7), (7,5), (6,7), (7,6),$
$(6,8), (8,6)\}$
$|A| = 22$

Selection of an Optimal Set of Nodes

Consider a network defined by a graph $G = (N,A)$:

- Nodes of set N are identified as $i = 1, 2, \ldots, |N|$
- Arcs of set A are identified as (i,j)
- Each arc $(i,j) \in A$ has an associated length l_{ij}

For a given positive integer number n, we aim to select a set of nodes $M \subset N$, with $|M| = n$, that optimize some objective.

Brute-force method to find an optimal solution: to evaluate all solutions and select the best one.

- Each combination of n out of $|N|$ is a possible solution
- There are $\frac{|N|!}{n! \times (|N|-n)!}$ combinations
- If $|N| = 100$ nodes, the number of solutions is:

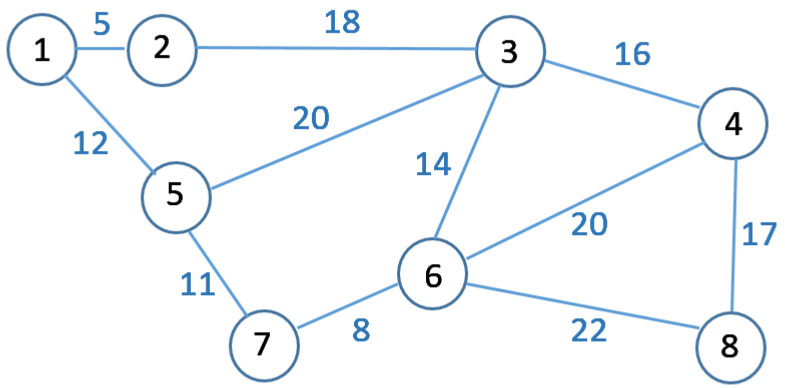
  * 1.86×10^{11}, for $n = 8$ * 1.73×10^{13}, for $n = 10$

So, the brute-force method is, in general, computationally expensive.

Optimization objective 1

Server Node Selection problem:

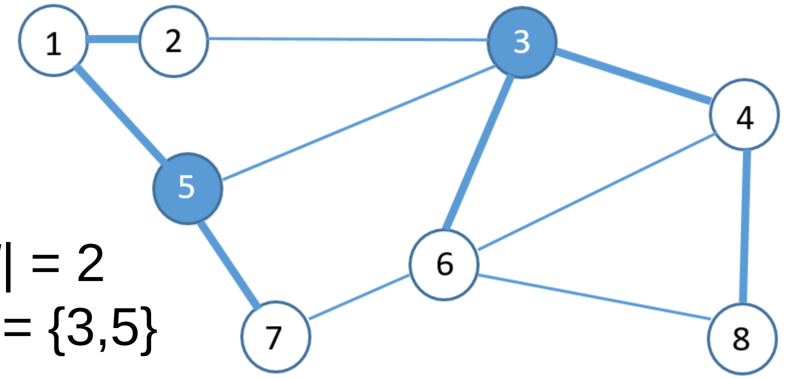
Select set *M*, with |*M*| = *n* nodes, aiming to minimize the average shortest path length from each node $i \in N$ to its closest node in set *M.*



Shortest paths and lengths:

Node 1:	$1 \to 5$	Length: 12
Node 2:	$2 \to 1 \to 5$	Length: 17
Node 3:	3	Length: 0
Node 4:	$4 \to 3$	Length: 16
Node 5:	5	Length: 0
Node 6:	$6 \to 3$	Length: 14
Node 7:	$7 \to 5$	Length: 11
Node 8:	$8 \to 4 \to 3$	Length: 33

One possible solution for *n* = 2:



|*M*| = 2

M = {3,5}

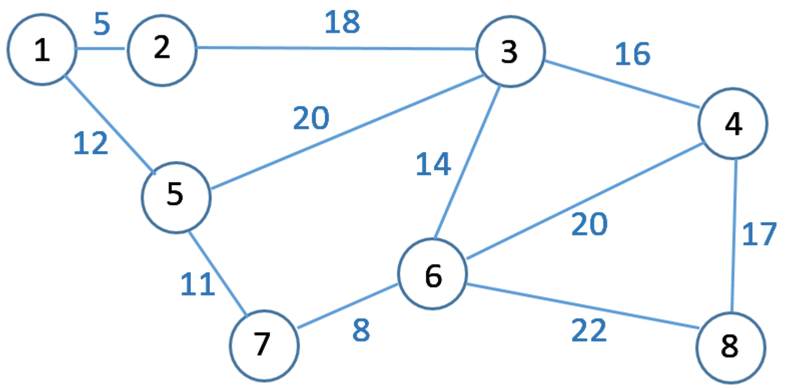
Average shortest path length of *M* = {3,5}:

(12+17+0+16+0+14+11+33)/8 = 12.875

Optimization objective 2

Critical Node Detection (CND) problem:

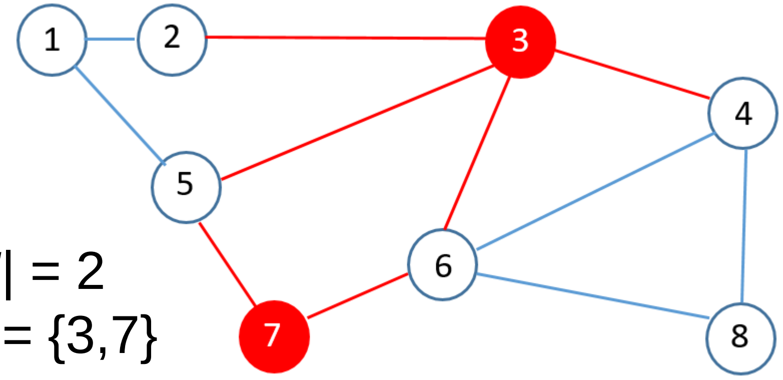
Select set *M*, with |*M*| = *n* nodes (named *critical nodes*) aiming to minimize the number of node pairs that can communicate when the critical nodes are eliminated (in this objective, the arc length l_{ij} are not considered).



Node pairs that can communicate:
{1,2},{1,5},{2,5},{4,6},{4,8},{6,8}

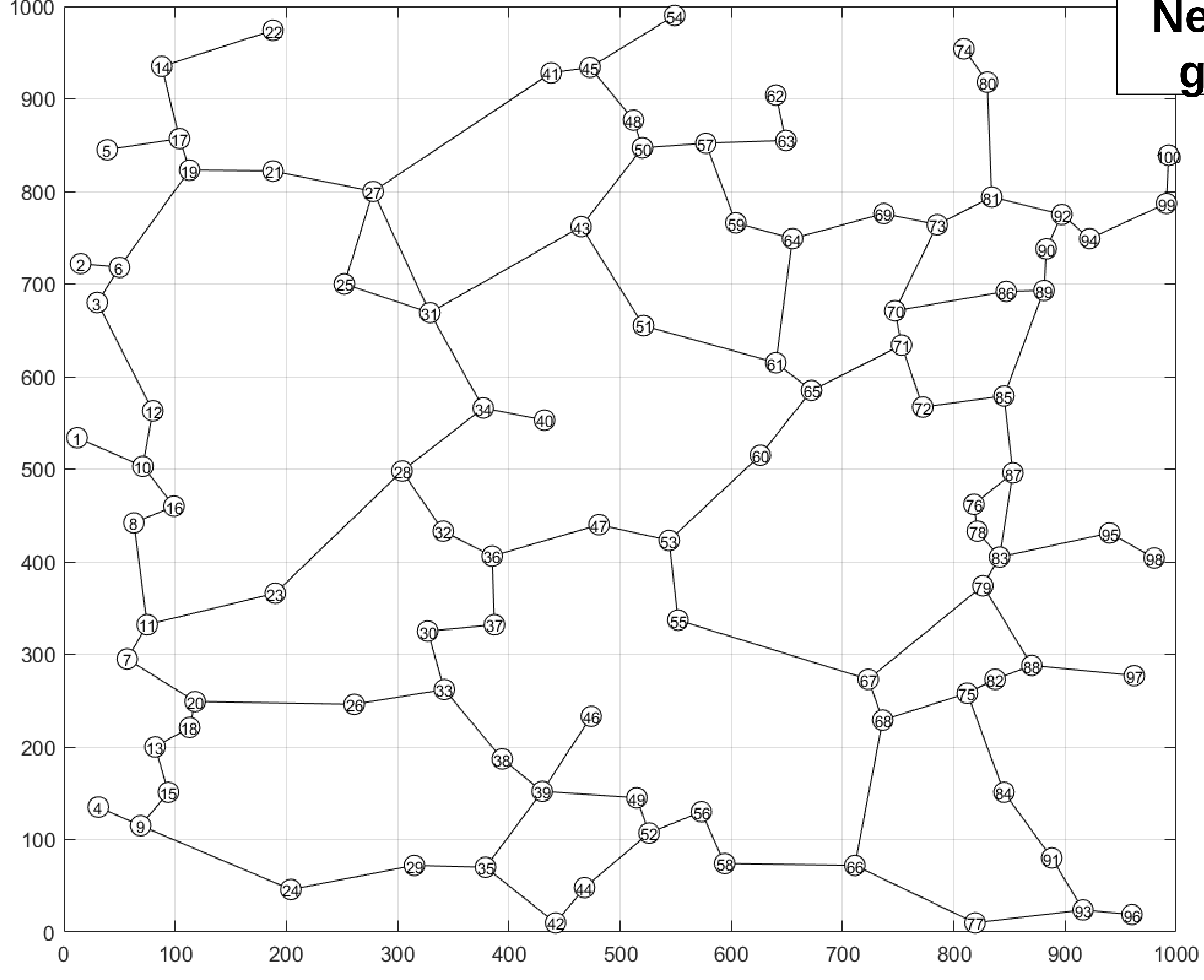
One possible solution for *n* = 2:

Number of nodes pairs of *M* = {3,7} is 6



|*M*| = 2
M = {3,7}

Network graph

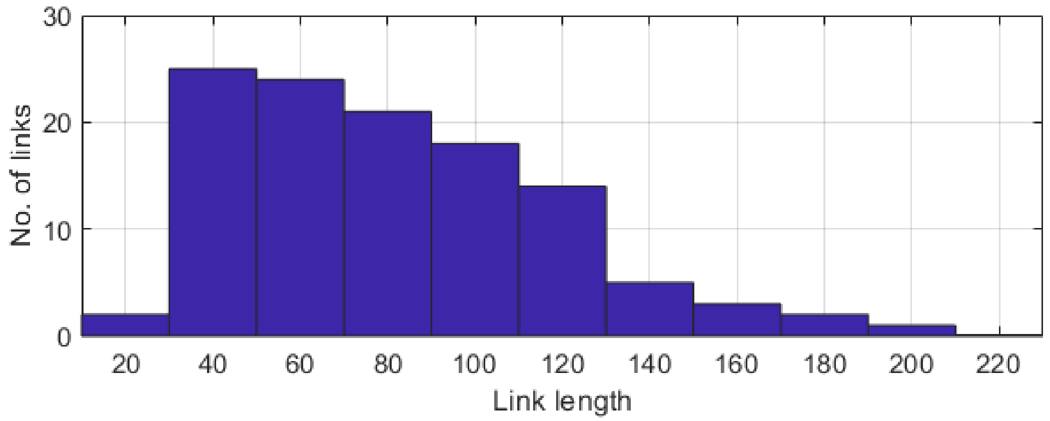


Network graph

Topology characteristics:

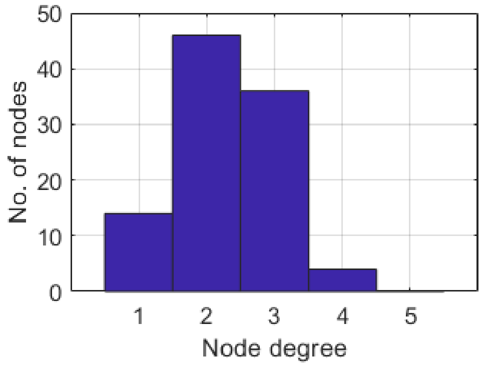
- 100 nodes
- 115 links (230 arcs)

Link length characterization:



Node degree – number of links connected to the node

Node degree characterization:



Input files and MATLAB supporting codes

Input files:

`Nodes.txt` – a matrix of 100 rows and 2 columns with the (x,y) coordinates of each node

`Links.txt` – a matrix of 115 rows and 2 columns with the node pairs of each link

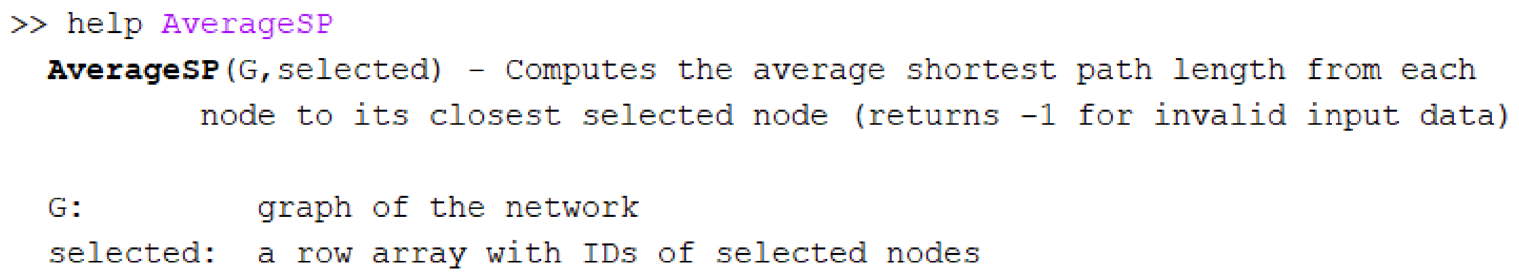
`L.txt` – a square matrix of 100x100 with the link length value l_{ij} for existing links (i,j) or 0 otherwise

Loading files, computing no. of nodes and links, creating graph:

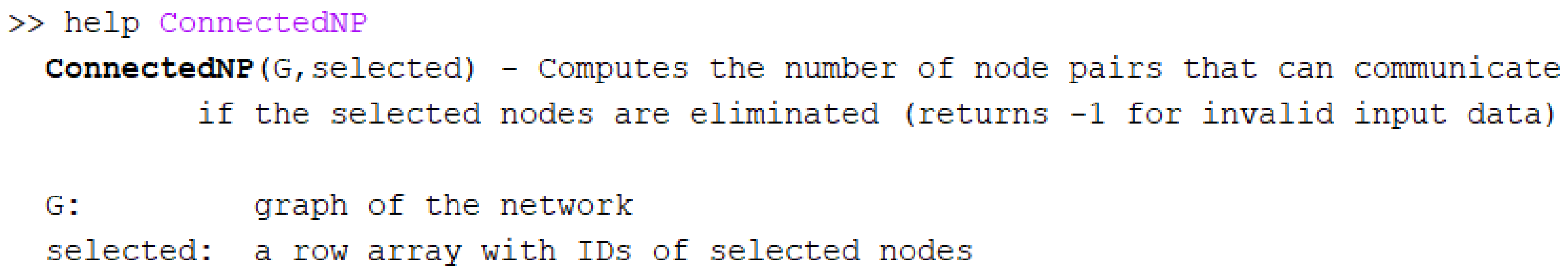
```
Nodes= load('Nodes.txt');      ← Creates Nodes with data from Nodes.txt
Links= load('Links.txt');      ← Creates Links with data from Links.txt
L= load('L.txt');              ← Creates L with data from L.txt
nNodes= size(Nodes,1);         ← Sets nNodes with the no. of rows of Nodes
nLinks= size(Links,1);         ← Sets nLinks with the no. of rows of Links
G= graph(L);                   ← Creates graph G based on matrix L
```

MATLAB supporting codes

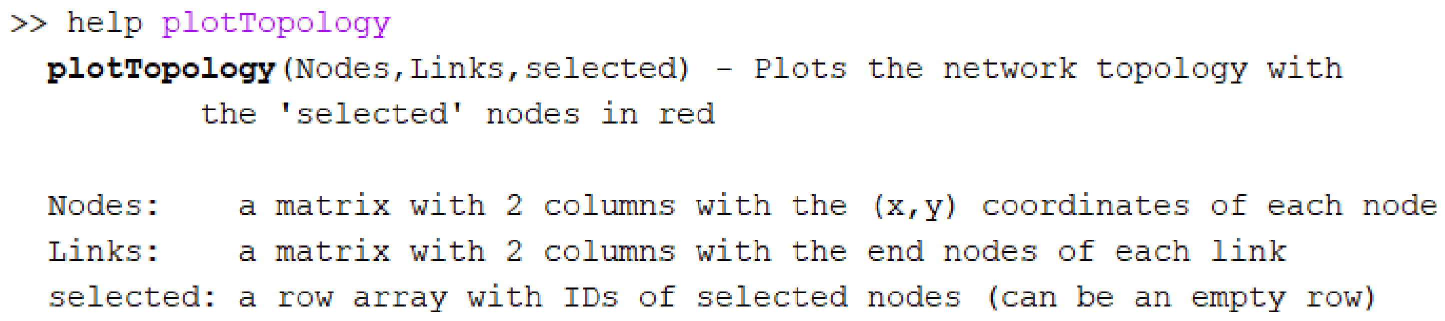
Provided MATLAB functions:



```
>> help AverageSP
  AverageSP(G,selected) - Computes the average shortest path length from each
          node to its closest selected node (returns -1 for invalid input data)

  G:          graph of the network
  selected:   a row array with IDs of selected nodes
```



```
>> help ConnectedNP
  ConnectedNP(G,selected) - Computes the number of node pairs that can communicate
          if the selected nodes are eliminated (returns -1 for invalid input data)

  G:          graph of the network
  selected:   a row array with IDs of selected nodes
```



```
>> help plotTopology
  plotTopology(Nodes,Links,selected) - Plots the network topology with
          the 'selected' nodes in red

  Nodes:    a matrix with 2 columns with the (x,y) coordinates of each node
  Links:    a matrix with 2 columns with the end nodes of each link
  selected: a row array with IDs of selected nodes (can be an empty row)
```

MATLAB supporting codes

Use of provided MATLAB functions:

```
Nodes= load('Nodes.txt');
Links= load('Links.txt');
L= load('L.txt');
nNodes= size(Nodes,1);
nLinks= size(Links,1);
G=graph(L);

% Plot the network:
figure(1)
plotTopology(Nodes,Links,[]);

% Plot the network with MATLAB plot function:
figure(2)
plot(G)

% Plot the network with server nodes:
figure(3)
selected= [11 27 66 87];
plotTopology(Nodes,Links,selected);
```

MATLAB supporting codes

Use of provided MATLAB functions:

```
Nodes= load('Nodes.txt');
Links= load('Links.txt');
L= load('L.txt');
nNodes= size(Nodes,1);
nLinks= size(Links,1);
G=graph(L);

selected= [11 27 66 87];

% Computing the average shortest path length from each
% node to its closest ‘selected’ node:
AvSP= AverageSP(G,selected)

% Computing the number of nodes pairs that can communicate
% when ‘selected’ nodes are eliminated:
ConNP= ConnectedNP(G,selected)
```

Practical Assignment 1

Analyze the provided network graph. Try to visually identify the best set of nodes *M* running the provided MATLAB functions (to compute the objective values):

- for each of the two optimization objectives
- considering *n* = 8 nodes

Optimization objective 1 (Server Node Selection):

Best set of nodes:

Objective value:

Optimization objective 2 (Critical Node Detection):

Best set of nodes:

Objective value:

Useful MATLAB information

```
A= 20; s= 6;
D= randperm(A,s);
```

← `D` is a row vector with a random permutation of 6 values from 1 to 20

```
t= tic;
while toc(t)<time
        (...)
end
```

← `while` cycle: runs during at least `time` seconds

```
n= 20;
for i = 1:n
    (...)
end
```

← `for` cycle: runs 20 iterations with `i` = 1, 2, …, 20 on each iteration

```
as= [2 4 5 9];
for i = as
    (...)
end
```

← `for` cycle: runs 4 iterations with `i` = 2, 4, 5 and 9 on each iteration

```
function [s p] = stat(a,b)
    s = a+b;
    p = [a b];
end
```

← syntax of a function with two input parameters (`a` and `b`) and two output parameters (`s` and `p`)

Practical Assignment 2

- Develop a function for each of the two optimization objectives that generates multiple random solutions during a given amount of time and outputs the best solution.
- The function must be in the form:

```
function [result,result_nodes] = xxx(G,n,time)
% Input:  G - graph of the network
%         n - no. of nodes of a solution
%         time - time to run the method (in seconds)
```

- Develop a script to run your function for *n* = 8 nodes and *time* = 30 seconds. At the end, plot the best solution found and register the obtained results.
- Compare the results with the ones of the Practical Assignment 1. Which ones are better?
- Run the developed script multiple times and check that the results are not always the same. What do you conclude?

Alternative method

- At the beginning, generate the best solution with a random solution.
- Then, repeat for a given amount of time:
  - generate a random neighbor of the best solution
  - if the neighbor is better than the best solution, the neighbor becomes the best solution; otherwise, the best solution remains the same
- Consider a neighbor solution as a solution that differs from the current best solution on a single selected node.

A random neighbor solution can be computed as:

```
N = numnodes(G);                % no. of nodes of graph G
n = 8;                          % no. of nodes of a solution
Nodes= [27 4 13 2 59 56 89 5];  % current solution
% Compute 'Others' with the nodes not in Nodes:
Others= setdiff(1:N,Nodes);
% Compute 'Neigh' with n-1 random nodes in 'Nodes' and
% one random node in 'Others':
Neigh= [Nodes(randperm(n,n-1)),Others(randperm(N-n,1))];
```

Practical Assignment 3

- Develop a function for each of the two optimization objectives that generates a solution by the alternative method (described in the previous slide) during a given amount of time.
- The input and output variables of the functions must be the same form as in the previous Practical Assignment 2.
- Develop a script to run your function for *n* = 8 nodes and *time* = 30 seconds. At the end, plot the best solution found and register the obtained results.
- Compare the results with the ones of Practical Assignments 1 and 2. What do you conclude?
- Run the developed script multiple times and check that the results are not always the same. What do you conclude?

CLASS 2

Basic definitions of optimization methods
Single-solution based metaheuristic methods

Optimization problem

- A (single objective) optimization problem is defined by:

S	a set of all feasible solutions $s \in S$ (also named *solution space S*)
$f(s)$	a function value of each solution $s \in S$

- A minimization problem aims to identify a solution $s' \in S$ such that

$$f(s') \leq f(s) \quad , s \in S$$

- A maximization problem aims to identify a solution $s' \in S$ such that

$$f(s') \geq f(s) \quad , s \in S$$

- In both cases, we say that solution *s'* is an optimal solution

- An optimization problem can have multiple optimal solutions but, in general, optimization methods aim to find one of them

Optimization methods

To solve an optimization problem, there are two types of methods:

- **Exact methods**: guarantee that the result at the end is an optimal solution
- **Heuristic methods**: aim to find a good solution in short time but do not guarantee that the solution at the end is optimal

Depending on the particular optimization problem of interest, the exact methods might be a non valid option when:

- they are not known;
- they do not run until the end due to (i) exaggerate running time or (ii) out-of-memory issues.

Heuristic methods are a valid option when:

- they are the only option;
- they provide better solutions than exact methods for the same running time;
- there is a need for a good solution in short running time.

Metaheuristic methods and metaheuristic algorithms

Metaheuristics are heuristic methods and, so, do not guarantee the optimality of the final solution

In both computer science and mathematical optimization domains, we distinguish two concepts:

- a **metaheuristic method** is a high-level search strategy aiming to provide a good solution to an optimization problem
- a **metaheuristic algorithm** is an optimization algorithm that adopts a metaheuristic method to a specific optimization problem

One broad classification of metaheuristic methods is:

- <u>Single-solution based methods</u>: focus on identifying, modifying and improving a single solution at each iteration
- <u>Population-based methods</u>: maintain and improve multiple solutions (named a population of solutions) at each iteration

Single-solution based metaheuristics

A single-solution based metaheuristic:

- focuses on a single current solution and explores its neighbor set (i.e., the set of solutions given by small modifications of the current solution) to find a better solution
- starts by an initial solution as the current solution and iteratively moves from the current solution to a neighbor solution until a stopping condition is met

---

Key aspects to be considered when applying the method:

- The identification of the initial solution
- The definition of the neighbor set (i.e., the set of solutions which are neighbors of a current solution on a specific optimization problem)
- The processing of the solutions of the neighbor set
- The move decision (i.e., which neighbor solution becomes the current solution)
- The stopping condition

Different single-solution based metaheuristics adopt different approaches to each of these key aspects.

Hill Climbing (or Local Search)

Hill Climbing (term used by the computer science community) or **Local Search** (term used by the mathematical optimization community) is a single-solution based metaheuristic method

- Consider a maximization problem with a set of feasible solutions S, a function $f(s)$, for each $s \in S$, and a set of neighbors of each solution $s \in S$ given by $V(s)$

```
s' ← Random(s ∈ S)
While not (stopping condition) do
    s ← Random(s ∈ V(s'))
    If  f(s) > f(s') do
        s' ← s
    EndIf
EndWhile
```

s' is set with a random solution

the algorithm stops when the stopping condition is reached

a random neighbor *s* of *s'* is selected

if *f*(*s*) is higher than *f*(*s'*), *s* becomes the current solution *s'*

The final result of the algorithm is solution *s'* whose function value is *f*(*s'*)

Hill Climbing (or Local Search)

Hill Climbing (term used by the computer science community) or **Local Search** (term used by the mathematical optimization community) is a single-solution based metaheuristic method

- Consider now a minimization problem with a set of feasible solutions S, a function $f(s)$, $s \in S$, and a set of neighbors of each solution $s \in S$ given by $V(s)$

```
$s' \leftarrow$ Random($s \in S$)
While not (stopping condition) do
    $s \leftarrow$ Random($s \in V(s')$)
    If  $f(s) < f(s')$ do
        $s' \leftarrow s$
    EndIf
EndWhile
```

if $f(s)$ is lower than $f(s')$, s becomes the current solution s'

In the minimization case, the move is when the neighbor solution s has a function value lower than (instead of higher than) the value of current solution s'

Steepest Ascent Hill Climbing (SA-HC)

Steepest Ascent Hill Climbing: Hill Climbing method where at each iteration the best neighbor solution becomes the current solution

- Consider an optimization problem with a feasible solution set *S*, a function *f*(*s*) and a set *V*(*s*) of neighbors of each $s \in S$

```
$s' \leftarrow$ Random($s \in S$)
$improved \leftarrow$ TRUE
While $improved$ do
    $s \leftarrow$ Best($s \in V(s')$)
    If $f(s)$ better than $f(s')$ do
        $s' \leftarrow s$
    Else do
        $improved \leftarrow$ FALSE
    EndIf
EndWhile
```

s' is set with a random solution

the algorithm stops when no improvement can be obtained by a neighbor solution

the best neighbor *s* of *s'* is selected, i.e., the neighbor *s* with the best value *f*(*s*)

if *f*(*s*) is better than *f*(*s'*), the neighbor solution *s* becomes the current solution *s'*

The final result of the algorithm is solution *s'* whose function value is *f*(*s'*)

Hill Climbing (HC) versus Steepest Ascent Hill Climbing (SA-HC)

Hill Climbing:

- Since at each iteration a random neighbor solution is evaluated, larger neighbor sets are usually better
- Stopping condition: (i) time limit, (ii) no. of iterations, (iii) no. of iterations without improvement, (iv) a combinations of the previous

Steepest Ascent Hill Climbing:

- Since at each iteration all neighbor solutions are evaluated, smaller neighbor sets are usually better (larger neighbor sets take too much runtime to be evaluated)
- Time efficiency depends on the initial solution (if the initial solution is bad, it runs many iterations until it stops)

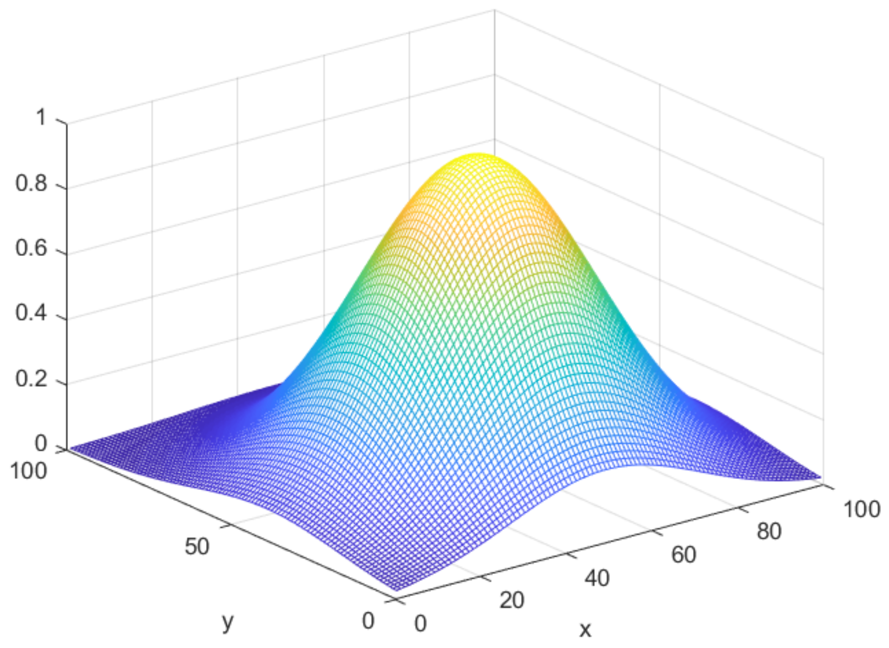
Both variants:

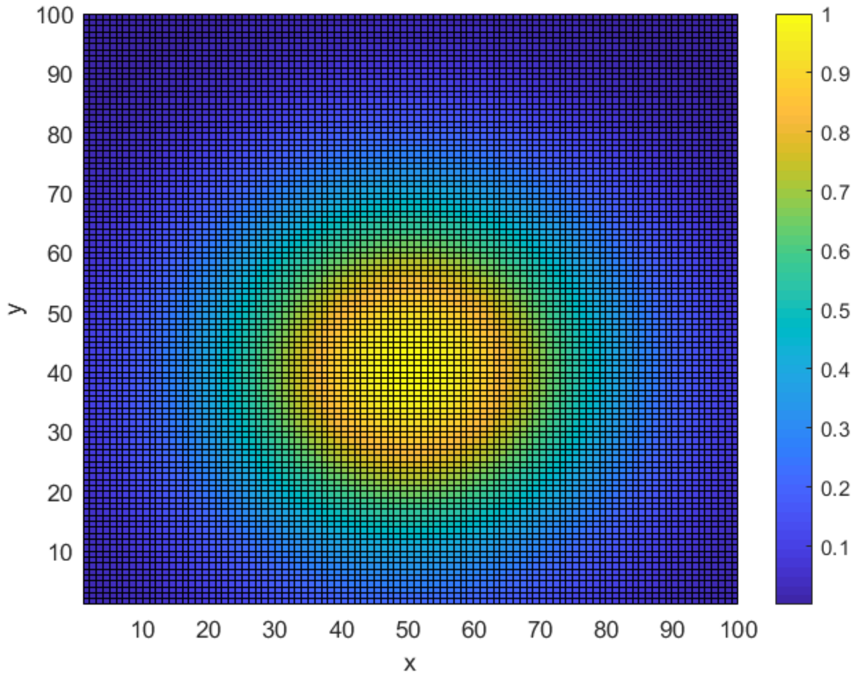
- change the current solution (i.e., move to a neighbor solution) only if the change improves the objective function
- consequently, they cannot improve further if the current solution is a local optimum solution (i.e., no neighbor solution is better)
- in general, don’t guarantee that the global optimum solution is found

Easiness of the objective function in Hill Climbing

Consider an optimization problem with 2 integer variables *x* and *y* and an objective function *f*(*x*,*y*) to be maximized.

- Hill Climbing guarantees that the optimal solution is found when *f*(*x*,*y*) is a convex function in the domain of the variables. Example:



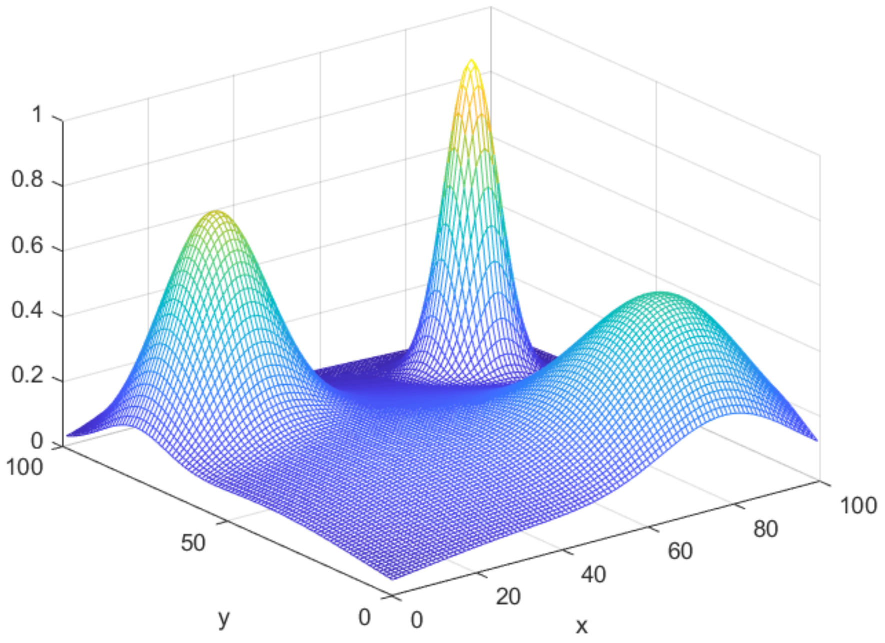


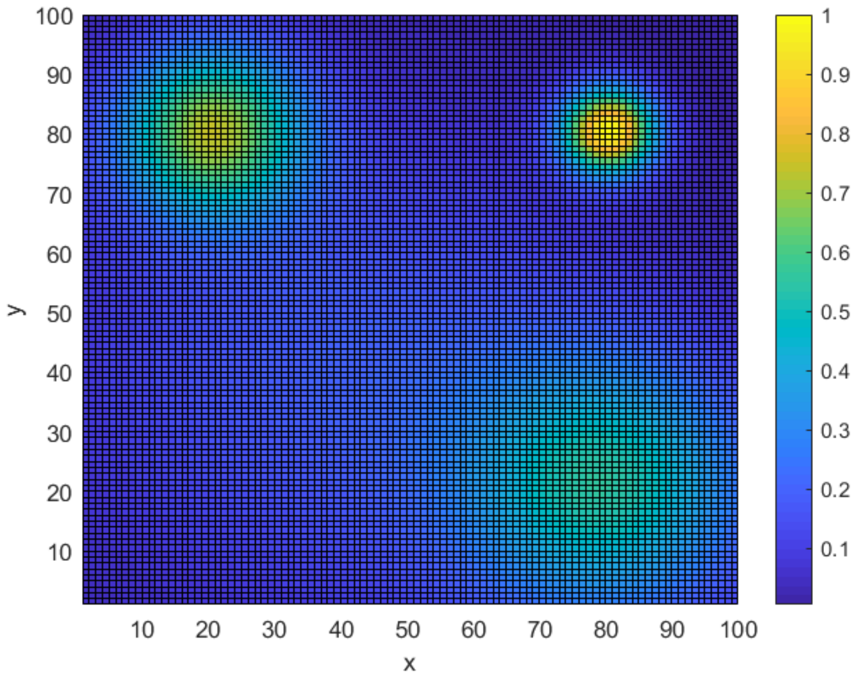
- There is only one local optimum solution which is, therefore, the global optimal solution.

Easiness of the objective function in Hill Climbing

Consider an optimization problem with 2 integer variables *x* and *y* and an objective function *f*(*x*,*y*) to be maximized.

- Hill Climbing does not guarantee to find the optimal solution when *f*(*x*,*y*) is not convex. Example:



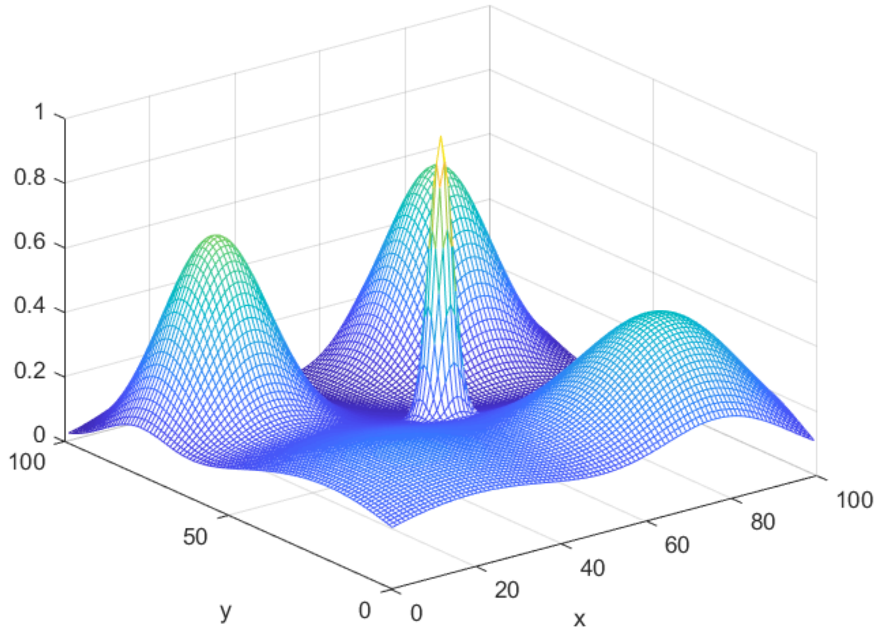


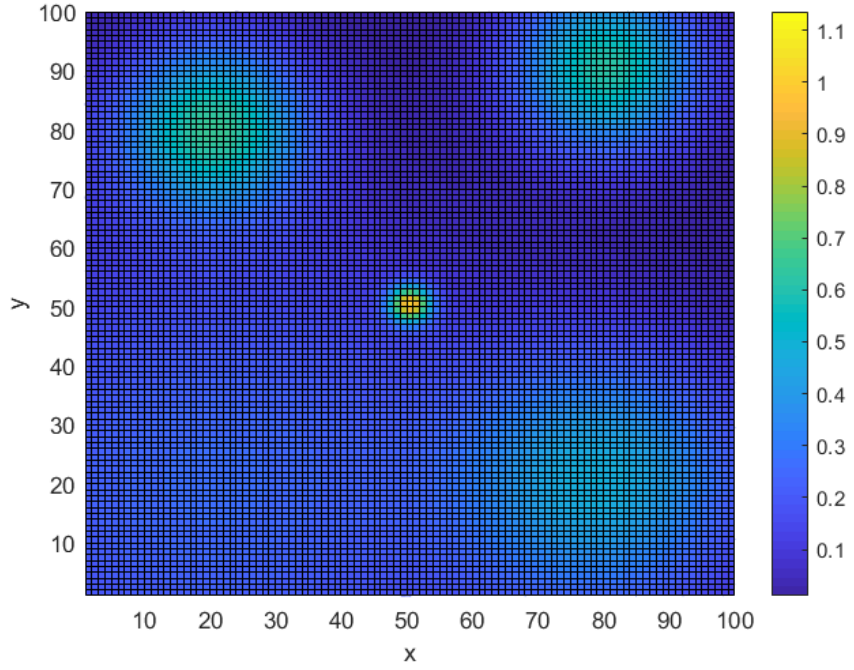
- There are multiple local optimum solutions and possibly a single global optimum solution.

Easiness of the objective function in Hill Climbing

Consider an optimization problem with 2 integer variables *x* and *y* and an objective function *f*(*x*,*y*) to be maximized.

- A harder example is when the set of solutions that let Hill Climbing to reach the global optimal solution is very small. Example:





Simulated Annealing (SA)

Simulated Annealing is a single-solution based metaheuristic method that lets the algorithm move to worst solutions as a means to escape from local optimum solutions

- The method is inspired on the annealing process in metallurgy. For example, to produce steel from iron:
  - first, the iron is heated to a very high temperature to reach a high energy state (liquid iron);
  - then, the iron is cooled very slowly to let the material reach the lowest possible energy state producing steel, a much more pure and resistant material than iron.
- The notion of slow cooling implemented in the SA method is interpreted as a slow decrease in the probability of accepting worst solutions as the algorithm evolves

Simulated Annealing

```
$s' \leftarrow$ Random($s \in S$)

$T \leftarrow T_0$
While not (stopping condition) do
    $s \leftarrow$ Random($s \in V(s')$)
    If $f(s)$ better than $f(s')$ do
        $s' \leftarrow s$
    ElseIf random([0,1]) $< e^{\frac{-|f(s)-f(s')|}{T}}$ do
        $s' \leftarrow s$
    EndIf
    $T \leftarrow$ Decrease($T$)
EndWhile
```

temperature T is set with the initial temperature T_0

if $f(s)$ is worse than $f(s')$, s becomes the current solution s' with a probability given by the Boltzmann distribution

temperature T is decreased

The final result of the algorithm is solution s' whose function value is $f(s')$

Simulated Annealing (with improvement in blue)

```
$s' \leftarrow \text{Random}(s \in S)$
$s_{best} \leftarrow s'$
$T \leftarrow T_0$
While not (stopping condition) do
    $s \leftarrow \text{Random}(s \in V(s'))$
    If $f(s)$ better than $f(s')$ do
        $s' \leftarrow s$
        If $f(s')$ better than $f(s_{best})$ do
            $s_{best} \leftarrow s'$
        EndIf
    ElseIf random([0,1]) $< e^{\frac{-|f(s)-f(s')|}{T}}$ do
        $s' \leftarrow s$
    EndIf
    $T \leftarrow \text{Decrease}(T)$
EndWhile
```

best solution s_{best} is set with current solution s'

temperature T is set with the initial temperature T_0

if $f(s')$ is better than $f(s_{best})$, the best solution s_{best} becomes the current solution s'

if $f(s)$ is worse than $f(s')$, s becomes the current solution s' with a probability given by the Boltzmann distribution

temperature T is decreased

The final result of the algorithm is solution s_{best} whose function value is $f(s_{best})$

Simulated Annealing Parameters

Boltzmann distribution:

$$e^{\frac{-|f(s)-f(s')|}{T}}$$

is a value in the interval [0,1].

- When temperature *T* is lower, the value decreases: the probability of moving to worst solutions decreases
- When the negative difference – |*f*(*s*) – *f*(*s'*)| is higher, the value decreases: moves are less likely for worst neighbors with larger differences to the current solution

In order to apply the SA method to a specific optimization problem, one must do three interrelated choices:

- The initial temperature T_0
- The decrease strategy of the temperature *T*
- The stopping condition

Simulated Annealing Parameters

Unfortunately, there are no choices that are good for all optimization problems, and there is no general way to find the best choices for each specific optimization problem

Some common guidelines:

- Select a desired probability at the beginning (for example 50%) and for the end (for example 0.1%) of the algorithm
- Use the value *f*(*s'*) of the initial solution *s'*
- Select T_0 such that:

$$e^{\frac{-f(s')}{T_0}} = 50\% \quad \Leftrightarrow \quad T_0 = \frac{-f(s')}{\ln(0.5)}$$

- Define a decreasing strategy in the form $T = \alpha \times T$ so that at the stopping condition the temperature T is around:

$$e^{\frac{-f(s')}{T}} = 0.1\%$$

Tabu Search (TS)

Tabu Search is also a single-solution based metaheuristic method:

- it has an associated Tabu List of last previous computed solutions
- it moves to the best neighbor that is not in the Tabu List, even if it is worst than the current solution.

Main characteristics:

- Like SA, it enables the algorithm to escape from local optimum solutions
- Like Steepest Ascent Hill Climbing, it evaluates all neighbors of a current solution
- The Tabu List prevents the search to repeat previous solutions avoiding search cycles
- In its basic version, the only required TS parameter is the size of the Tabu List (it cannot grow indefinitely because the computational effort becomes too high)

Tabu Search

```
$s' \leftarrow$ Random($s \in S$)
$s_{best} \leftarrow s'$
$T_{LIST} \leftarrow \{s'\}$
While not (stopping condition) do
    $V'(s') \leftarrow \{s \in V(s'): s \notin T_{LIST}\}$
    $s' \leftarrow$ Best($s \in V'(s')$)
    $T_{LIST} \leftarrow$ Update($s'$)
    If $f(s')$ better than $f(s_{best})$ do
        $s_{best} \leftarrow s'$
    EndIf
EndWhile
```

best solution s_{best} is set with current solution s'

Tabu List T_{LIST} is set with the current solution s'

$V'(s')$ is set with the neighbor solutions of s' that are not tabu (i.e., are not in the Tabu List)

the best solution in $V'(s')$ becomes the current solution s'

Tabu List T_{LIST} is updated by adding the current solution s' (and eliminating the oldest solution if the T_{LIST} size is violated)

if $f(s')$ is better than $f(s_{best})$, the best solution s_{best} becomes the current solution s'

The final result of the algorithm is solution s_{best} whose function value is $f(s_{best})$

Simulated Annealing (SA) versus Tabu Search (TS)

Simulated Annealing:

- Like Hill Climbing, a random neighbor is evaluated at each iteration and larger neighbor sets are usually better
- Requires hard parameter tuning (related with temperature) by testing on each specific optimization problem
- Usually, more appropriate when the initial solution is of poor quality

Tabu Search:

- At each iteration, all neighbors are evaluated and checked in the Tabu List (even harder than Steepest Ascent Hill Climbing)
- So, smaller neighbor sets are usually better (larger neighbor sets take too much runtime to be evaluated)
- Requires soft parameter tuning (size of Tabu List must be as large as possible without penalizing too much the overall runtime)
- Usually, TS is more appropriate when the initial solution is of good quality

Neighbor solutions in the node optimization challenges

Recall the two node optimization objectives on the first class:

For a given network defined by a graph $G=(N,A)$ and a given number n, we need to select a set $M \subset N$ of $|M| = n$ nodes:

1. Each arc (i,j) has an associated length l_{ij} and the aim is to minimize the average shortest path length from each node $i \in N$ to its closest node in set M
2. Critical Node Detection: to minimize the number of node pairs that can communicate when the nodes in set M are eliminated

For a current solution, two possible neighbor definitions are:

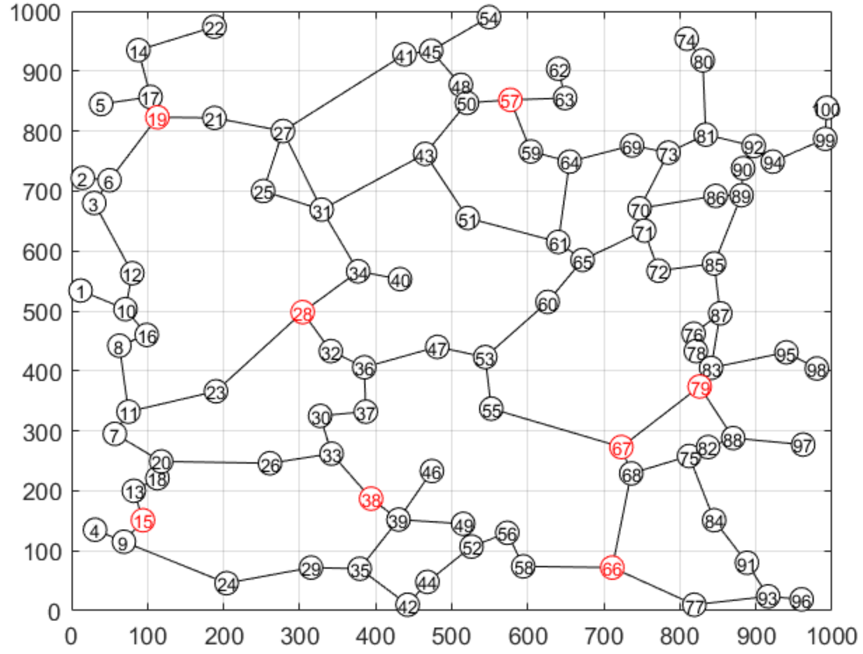
Definition 1: a neighbor solution is obtained by swapping a node in the current solution by a node not in the current solution

Definition 2: a neighbor solution is obtained by swapping a node in the current solution by a neighbor node not in the current solution

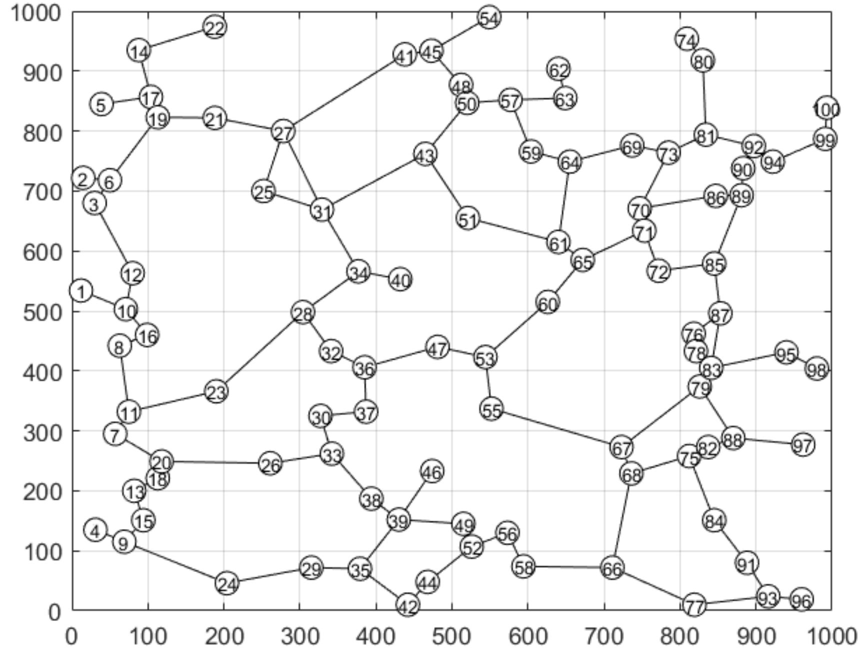
The swap is only between nodes with a direct link between them (it reduces the number of neighbor solutions)

Illustration of neighbor solutions in the node optimization challenges

Current solution:



Is a neighbor solution in both Definitions 1 and 2: the swapped nodes are 19 and 21 with a direct link between them



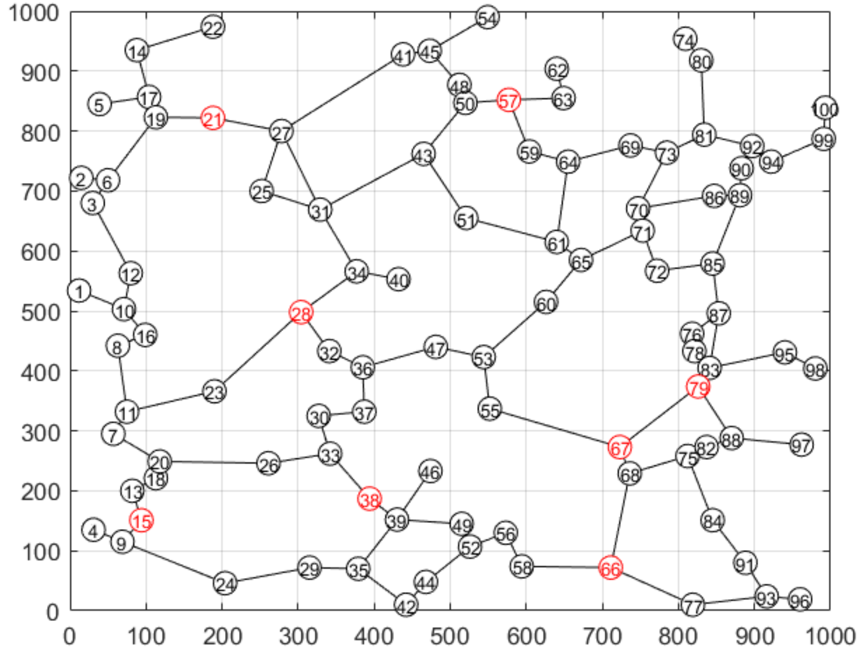
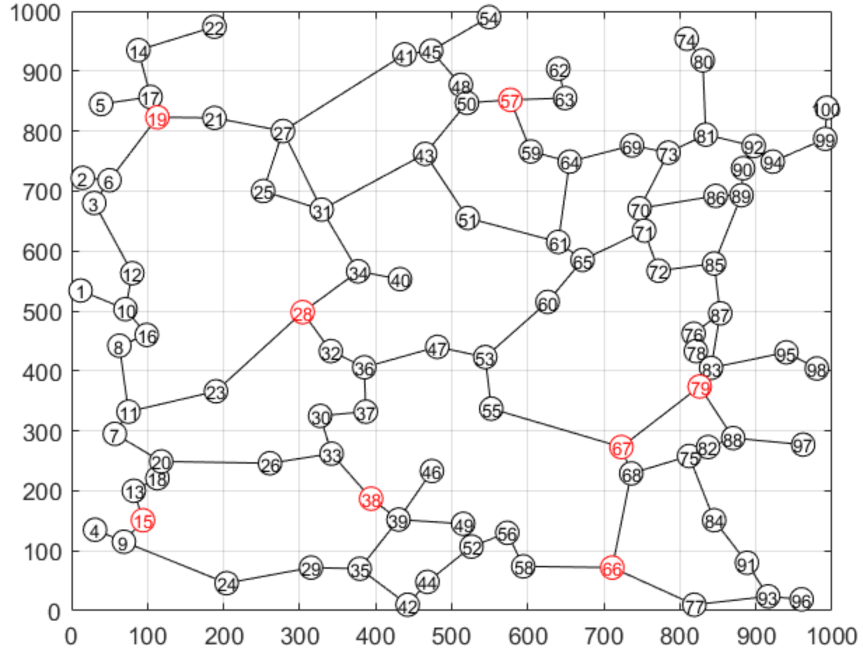
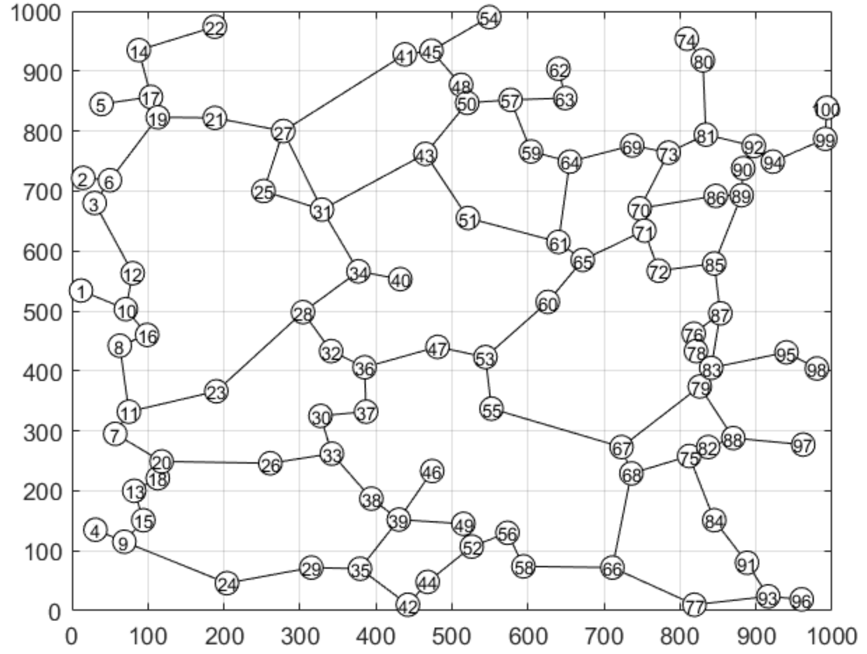


Illustration of neighbor solutions in the node optimization challenges

Current solution:



Is a neighbor solution in only Definition 1:
the swapped nodes are 19 and 27 without a direct link between them



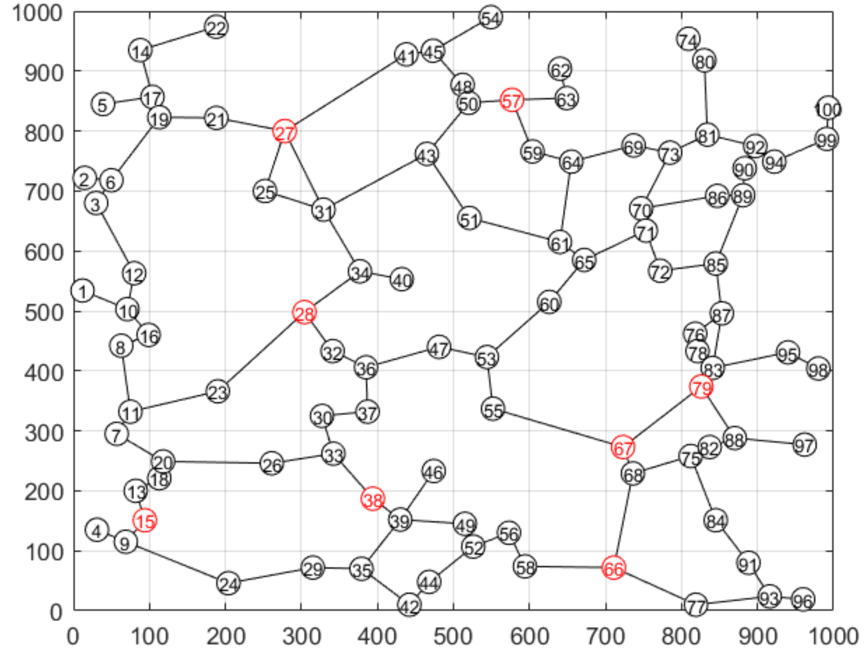
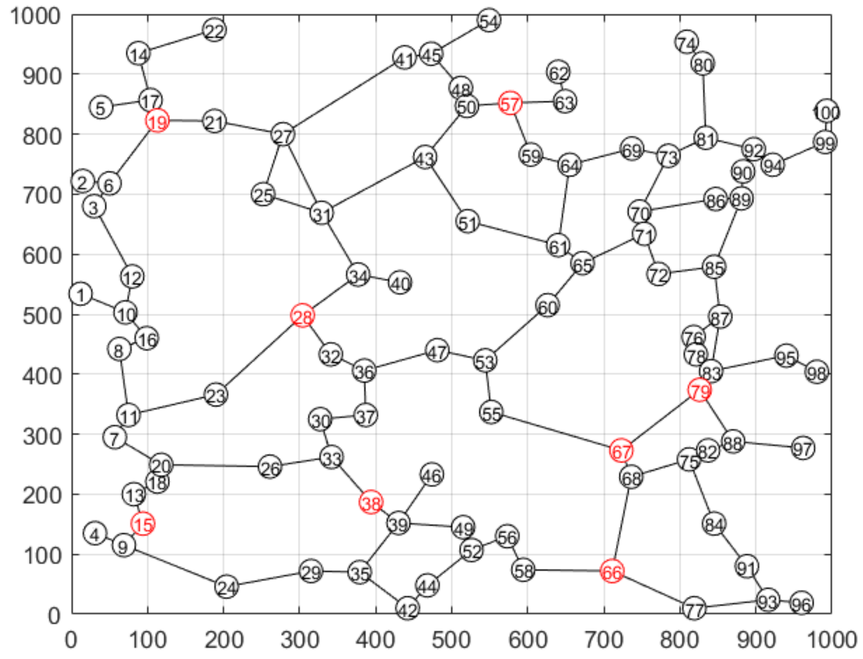
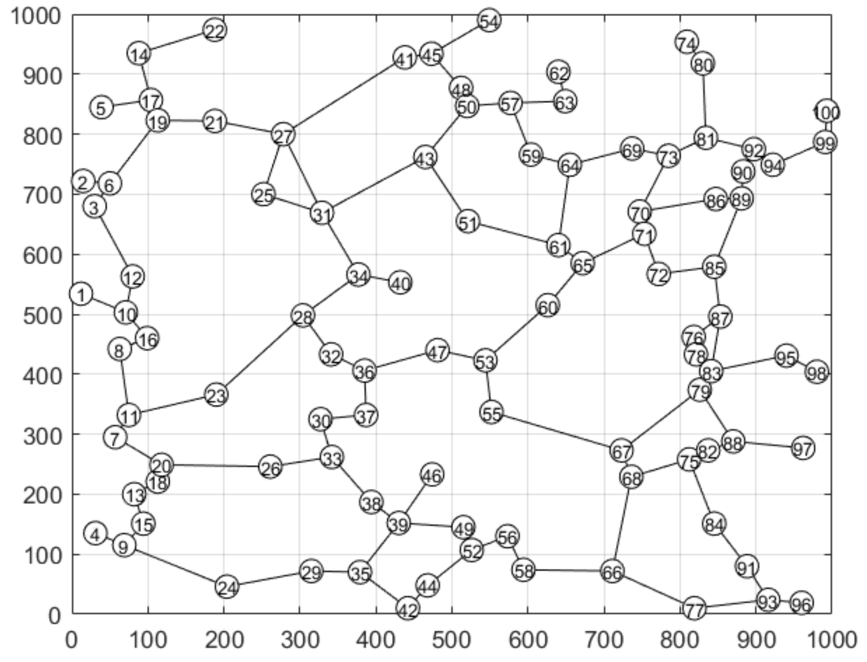
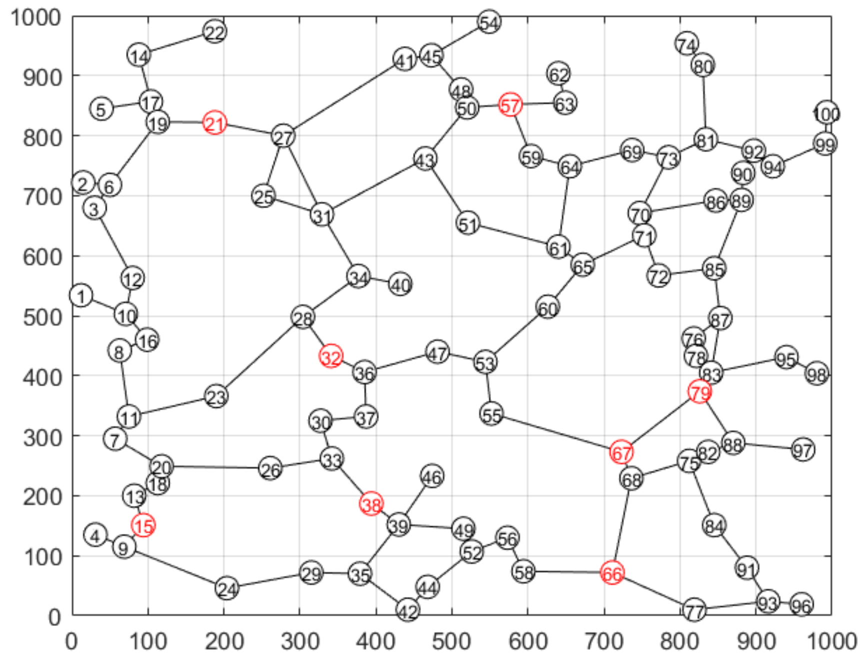


Illustration of neighbor solutions in the node optimization challenges

Current solution:







Is not a neighbor solution in any of the definitions:
there are two swaps (from 19 to 21 and from 28 to 32)

Computing a random neighbor solution in the selection of optimal nodes

Use of provided MATLAB functions:

```
Nodes= load('Nodes.txt');
Links= load('Links.txt');
L= load('L.txt');
nNodes= size(Nodes,1);
nLinks= size(Links,1);
G= graph(L);
n= 8;
```

Random neighbor solutions are required for Hill Climbing and SA and, therefore, Definition 1 (which defines the larger neighbor set) is used.

A random neighbor solution can be computed as (recall from slide 17):

```
current= [27 4 38 2 47 56 45 11]; % current solution
aux= setdiff(1:nNodes,current);   % nodes not in current solution
neig= [current(randperm(n,n-1)),aux(randperm(nNodes-n,1))];
```

Enumerating all neighbor solutions in the selection of optimal nodes

Use of provided MATLAB functions:

```
Nodes= load('Nodes.txt');
Links= load('Links.txt');
L= load('L.txt');
nNodes= size(Nodes,1);
nLinks= size(Links,1);
G= graph(L);
```

Definition 1: a neighbor solution is obtained by swapping a node in the current solution by a node not in the current solution

The set of neighbor solutions can be computed as:

```
current= [27 4 38 2 47 56 45 11]; % current solution
aux= setdiff(1:nNodes,current);   % nodes not in current solution
for a= current
    for b= aux
        neig= [setdiff(current,a) b];
        ...
    end
end
```

Enumerating all neighbor solutions in the selection of optimal nodes

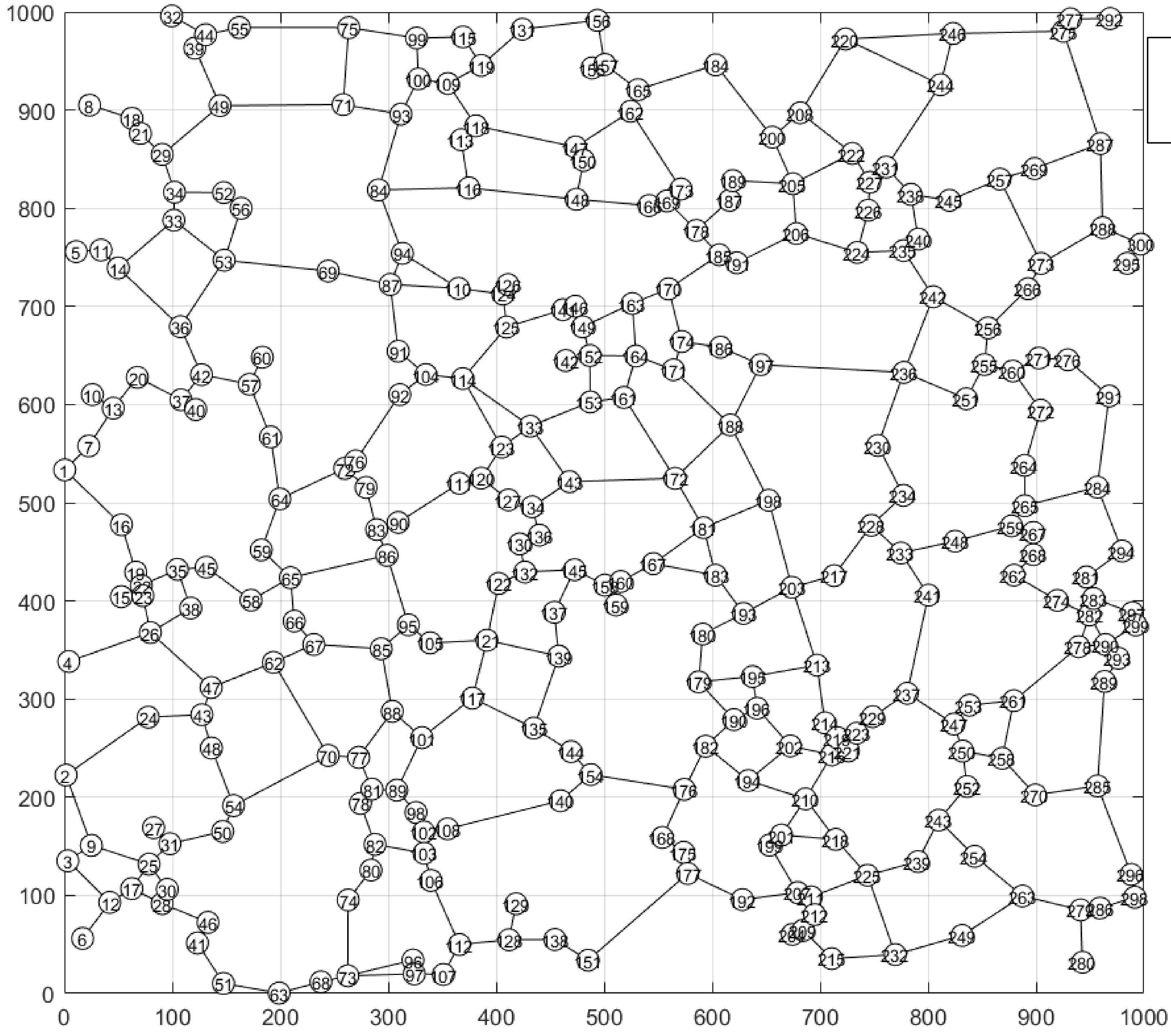
Use of provided MATLAB functions:

```
Nodes= load('Nodes.txt');
Links= load('Links.txt');
L= load('L.txt');
nNodes= size(Nodes,1);
nLinks= size(Links,1);
G= graph(L);
```

Definition 2: a neighbor solution is obtained by swapping a node in the current solution by a neighbor node not in the current solution

The set of neighbor solutions can be computed as:

```
current= [27 4 38 2 47 56 45 11]; % current solution
for a= current
    aux= setdiff(neighbors(G,a)’,current);
    for b= aux
        neig= [setdiff(current,a) b];
        ...
    end
end
```



Net300

Practical Assignment 4

In Practical Assignment 3, you have implemented a HC (Hill Climbing) algorithm for each of the two problems:

1. Server Node Selection: select set *M* to minimize the average shortest path length from each node $i \in N$ to its closest node in set *M*
2. Critical Node Detection: select set *M* to minimize the number of node pairs that can communicate when the nodes in set *M* are eliminated

---

- In Practical Assignment 4, develop two functions for the Server Node Selection problem (use the Practical Assignment 3 as starting point):
  - implementing the SA-HC (Steepest Ascent Hill Climbing) algorithm,
  - one function for each of the 2 neighbor definitions described in the previous slides,
  - besides the best solution and its objective value, each function must also output the number of iteration runs and the total running time.
- Develop a script to run the HC with a runtime of 30 seconds and the two SA-HC variants for n = 12 nodes in Net300 network topology (input files: `Nodes300.txt`, `Links300.txt` and `L300.txt`).
- Compare the results and the runtimes obtained by the algorithms.

Practical Assignment 5

- Using the Practical Assignment 3 implementation as starting point, develop a function for the Server Node Selection problem:
  - implementing the SA (Simulated Annealing) algorithm,
  - using as stopping criteria a runtime limit (defined as an input parameter),
  - besides the best solution found and its objective value, the function must also output the total number of iteration runs and the runtime when the best solution was found.
- Develop a script to run the SA algorithm for n = 12 nodes in Net300 network topology with a runtime of 30 seconds.
- Test different temperature settings to make the algorithm as efficient as possible and register the results and runtime values of the best settings.
- Compare the results and the runtime values obtained by the SA algorithm with the results and runtimes obtained in the previous Practical Assignment 4.

CLASS 3

Multi-start single-solution based metaheuristic methods

Multi-start approaches in single-solution methods

So far, the initial solution of the previously described methods was computed randomly. Recall that:

- Hill Climbing (HC) and Steepest Ascend Hill Climbing (SA-HC) methods cannot escape from local optimum solutions
- Simulated Annealing (SA) is hard to tune its parameters
- Tabu Search (TS) might take too many steps to escape from local optimum solutions

An alternative way to explore different regions of the set *S* of feasible solutions of an optimization problem is:

- to run one of the previous methods for different initial random solutions,
- keep the best among all found solutions.

Multi-start approaches in single-solution methods

```
$s' \leftarrow \text{Random}(s \in S)$
$s \leftarrow \text{Method}(s')$
$s_{best} \leftarrow s$
**While** not (stopping condition) **do**
    $s' \leftarrow \text{Random}(s \in S)$
    $s \leftarrow \text{Method}(s')$
    **If** $f(s)$ is better than $f(s_{best})$ **do**
        $s_{best} \leftarrow s$
    **EndIf**
**EndWhile**
```

s' is set with a random solution

s is set with the best solution found by Method starting on solution *s'*

best solution s_{best} is set with solution *s*

s' is set with a new random solution

s is set with the best solution found by Method starting on solution *s'*

if $f(s)$ is better than $f(s_{best})$, the best solution s_{best} becomes solution *s*

The final result of the algorithm is solution s_{best} whose function value is $f(s_{best})$

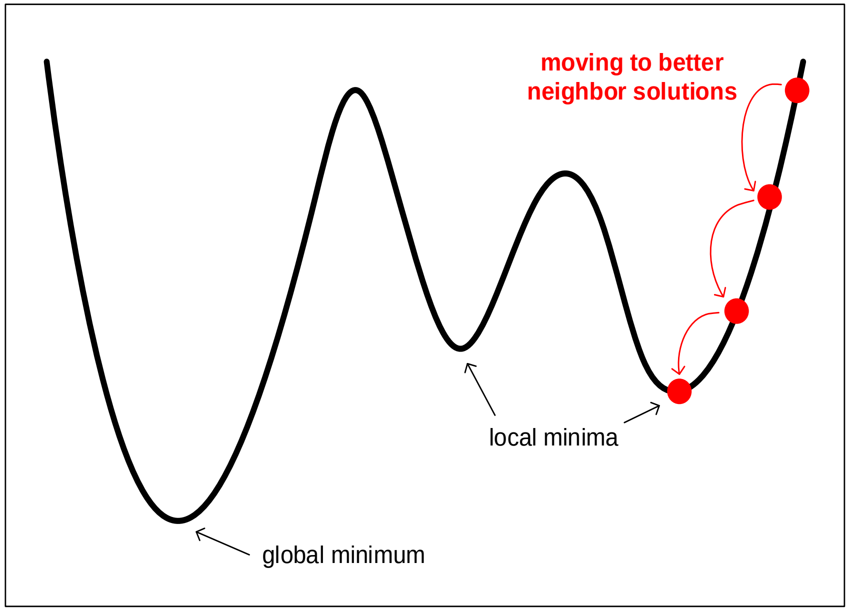
Multi-start approaches in single-solution methods

Although possible with any of the previous methods, multi-start approaches are frequently used with SA-HC:

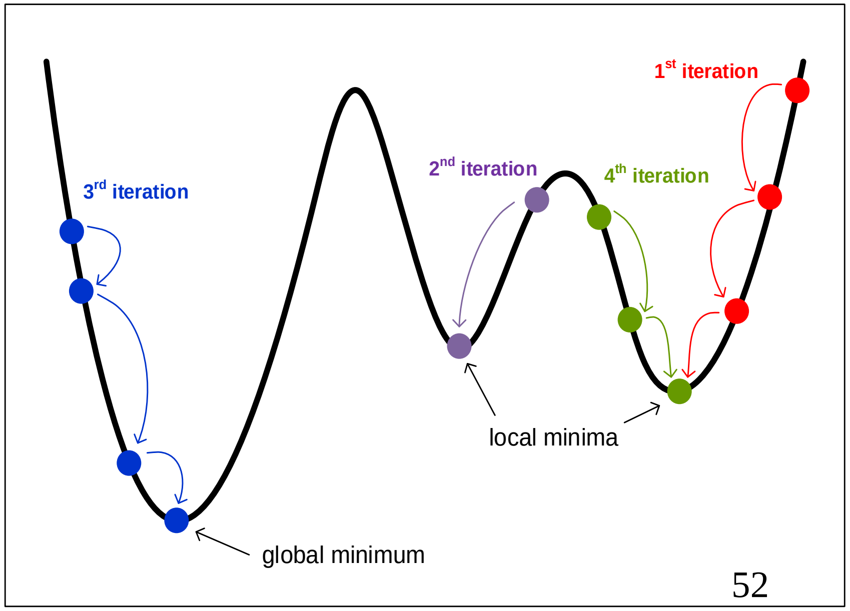
- each initial random solution starts in a different point of the solution space *S*,
- SA-HC finds the local optimum closest to the initial solution.

Illustration on a minimization problem:

SA-HC



Multi-start SA-HC



Multi-Start Hill Climbing Methods

Quality of an initial solution:

- how close its value (given by the optimization function) is from the value of its closest local optimum solution

Efficiency of a method:

- its ability to find good solutions in shorter running times

1. The quality of the initial solutions has a major influence on the efficiency of the multi-start HC methods
2. Pure random initial solutions might result in inefficient multi-start HC methods, when compared with SA or TS for example
3. To get more efficient multi-start HC methods, initial solutions must be computed:
   - on one hand, with randomness (to maintain the ability of the method to explore multiple regions of the feasible set *S*)
   - on the other hand, giving preference to good quality solutions

Greedy based methods to compute initial solutions

1. Consider an optimization problem with a solution space *S* and an optimization function *f*(*s*)
2. Assume that a solution can be defined as an ordered set of elements
   - in the optimization problems introduced in the first class, the aim is to select a given number of nodes
   - a solution can be defined by its first node, second node, etc…

A <u>greedy based method</u> to compute an (initial) solution is as follows:

- select iteratively each element
- at each iteration, select the next element taking into consideration the previous selected elements

<u>Greedy method</u>:

- at iteration *i*, select the i^{th} element that, together with the previous selected elements, gives the best partial solution

Greedy method

Consider an optimization problem characterized by:

- a set S of feasible solutions and an optimization function $f(s)$
- a solution $s \in S$ defined by n elements
- a set E with the set of elements that can be selected

s is initialized as empty

the cycle runs *n* times to select *n* elements

e is the element from set *E* whose partial solution $s \cup \{e\}$ has the best function value

```
$s \leftarrow \{\ \}$
For $i$ from 1 to $n$ do
    $e \leftarrow \{e \in E : f(s \cup \{e\})$ is the best value$\}$
    $s \leftarrow s \cup \{e\}$
    $E \leftarrow E \setminus \{e\}$
EndFor
```

element *e* is added to partial solution *s*

element *e* is removed from the set *E* of elements that can be selected

At the end, the Greedy solution is *s*

Greedy method with MATLAB in the node optimization problems

```
$s \leftarrow \{\,\}$
**For** $i$ from 1 to $n$ **do**
    $e \leftarrow \{e \in E : f(s \cup \{e\})$ is the best value$\}$
    $s \leftarrow s \cup \{e\}$
    $E \leftarrow E \setminus \{e\}$
**EndFor**
```

Consider a network defined by a graph G=(N,A) and each arc (i,j) has an associated length l_{ij}.

Consider the Server Node Selection problem: to select a number n of nodes minimizing the average shortest path length from each node to its closest selected node.

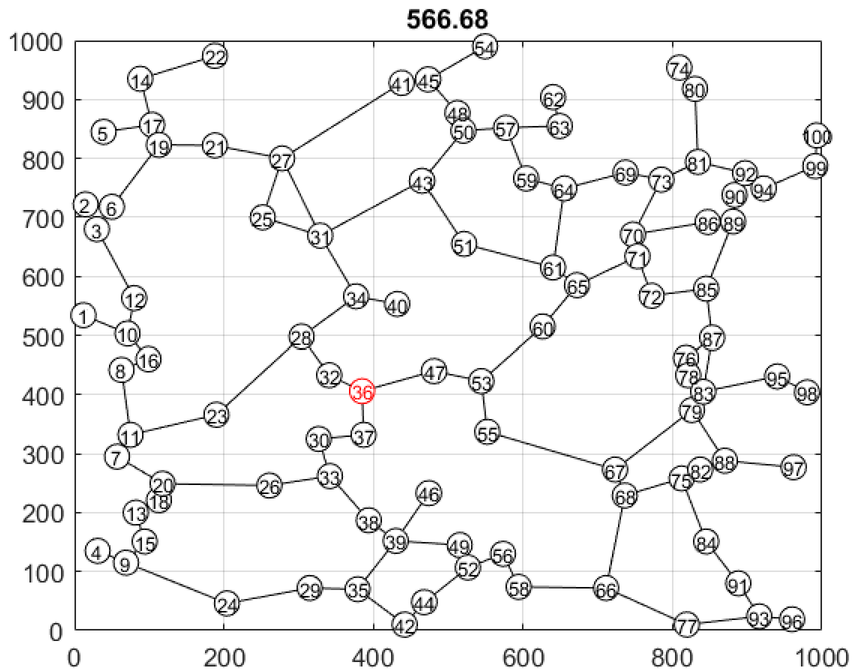
```
Nodes= load('Nodes.txt');
Links= load('Links.txt');
L= load('L.txt');
G= graph(L);
n= 10;

E= 1:numnodes(G);
s= [];
for i= 1:n
    best= inf;
    for j= E
        aux= AverageSP(G,[s j]);
        if aux < best
            best= aux;
            e= j;
        end
    end
    s= [s e];
    E= setdiff(E,e);
end
```

Illustration of the Greedy Algorithm in the Server Node Selection problem

Consider a network defined by a graph $G=(N,A)$ and each arc (i,j) has an associated length l_{ij}. The aim is to select a number n of nodes minimizing the average shortest path length from each node to its closest selected node.

Greedy algorithm for $n = 6$ using Network graph:

1st iteration: selected node 36



2nd iteration: selected node 85

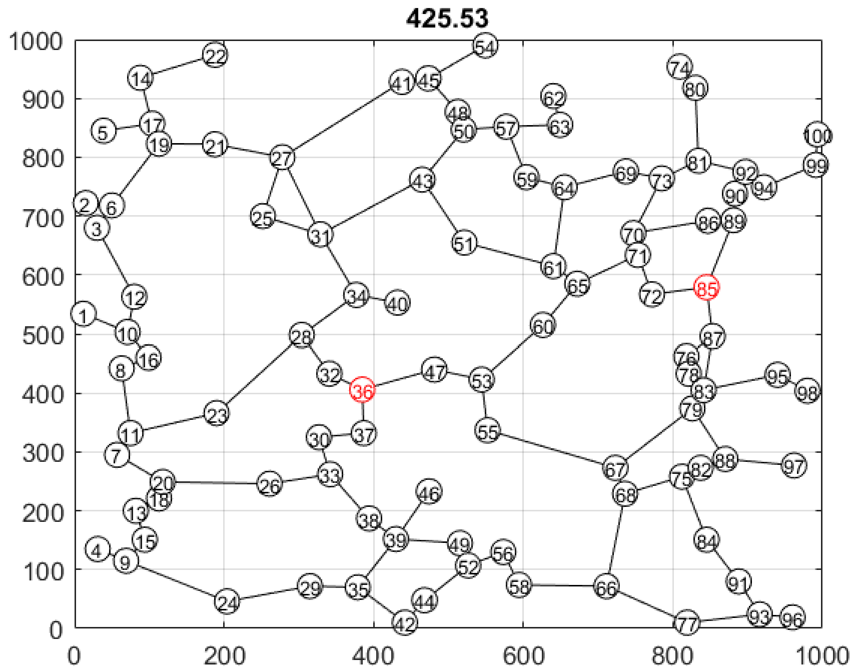
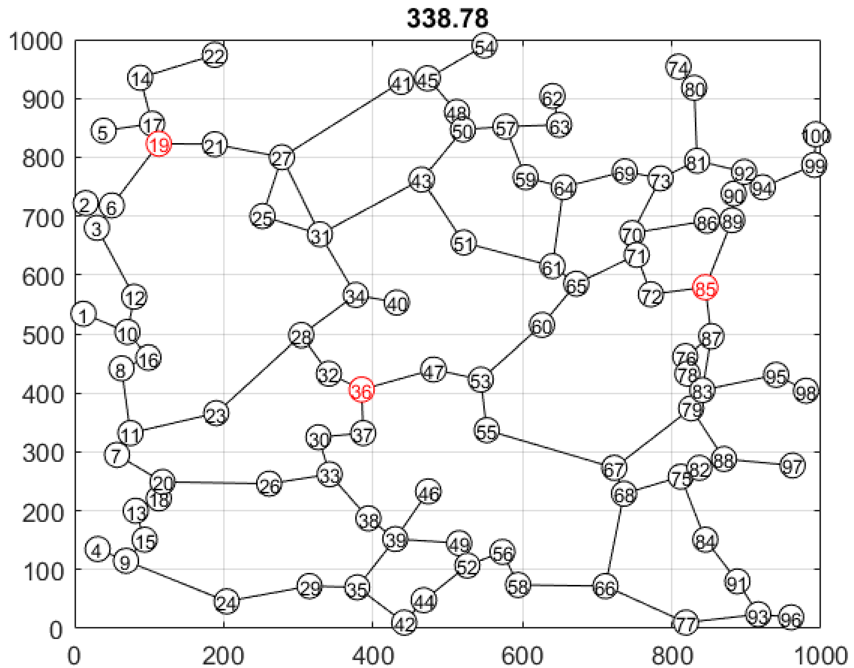


Illustration of the Greedy Algorithm in the Server Node Selection problem

Consider a network defined by a graph *G*=(*N*,*A*) and each arc (*i*,*j*) has an associated length l_{ij}. The aim is to select a number *n* of nodes minimizing the average shortest path length from each node to its closest selected node.

Greedy algorithm for *n* = 6 using Network graph:

3rd iteration: selected node 19



4th iteration: selected node 39

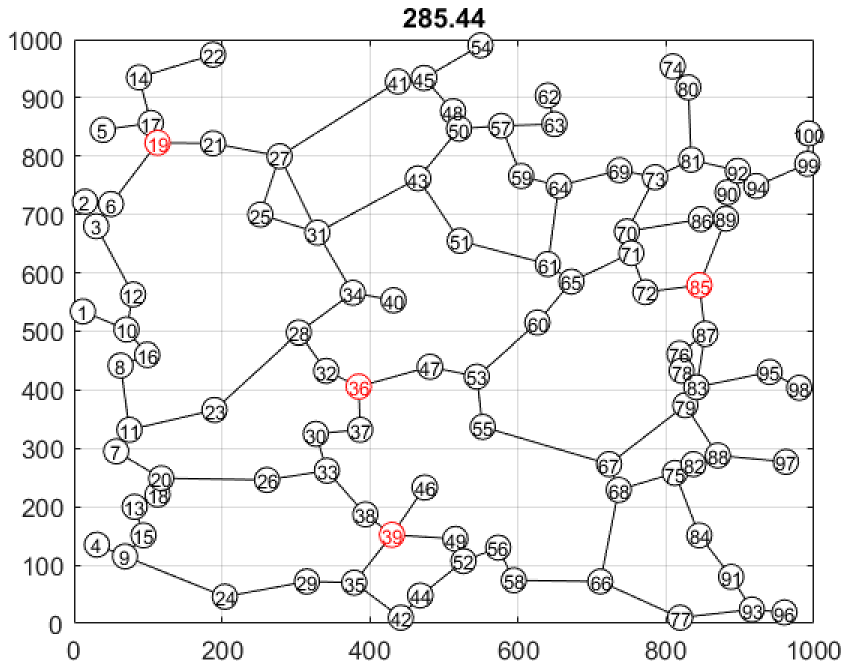
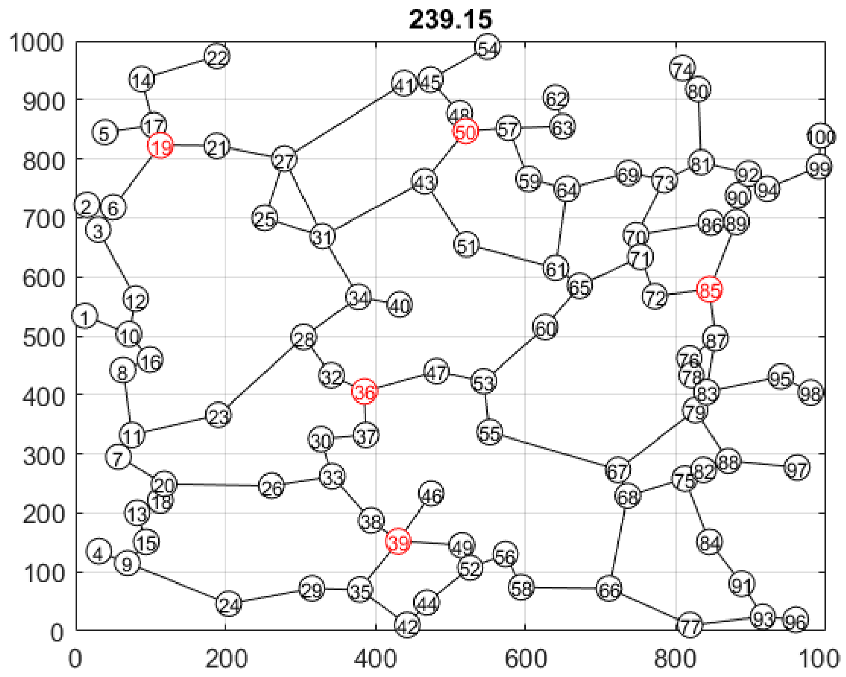


Illustration of the Greedy Algorithm in the Server Node Selection problem

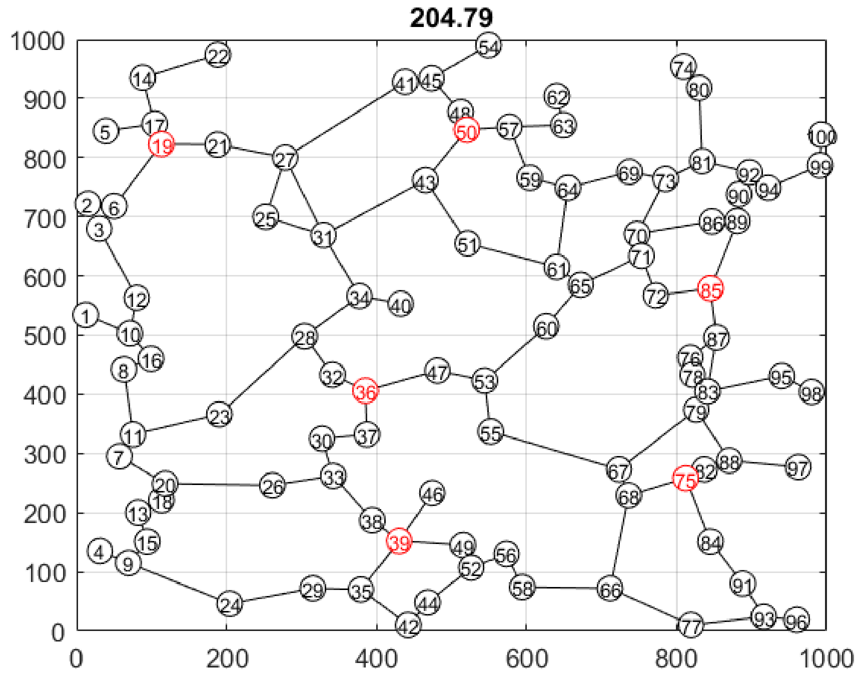
Consider a network defined by a graph G=(N,A) and each arc (i,j) has an associated length l_{ij}. The aim is to select a number n of nodes minimizing the average shortest path length from each node to its closest selected node.

Greedy algorithm for n = 6 using Network graph:

5th iteration: selected node 50



6th iteration: selected node 75



Computed solution with optimization value = 204.79

Greedy Randomized method

NOTE: the Greedy method always generates the same solution; so, it is useless in a multi-start approach (it requires the different initial solutions to be different)

Again, consider an optimization problem with a solution space *S* and an optimization function *f*(*s*), and assume that a solution can be defined as an ordered set of elements.

Greedy Randomized method to compute a “good” randomized solution:

- select iteratively each element
- at iteration *i*, compute the set *R* with the *r* elements that, each one together with the previous selected elements, gives the *r* (with $r \geq 2$) best partial solutions
- select randomly the i^{th} element from the elements of *R*

The integer *r* is a parameter of the Greedy Randomized method.

NOTE: the Greedy method is the particular case of the Greedy Randomized method with $r = 1$

Greedy Randomized method

Consider an optimization problem characterized by:

- a set S of feasible solutions and an optimization function $f(s)$
- a solution $s \in S$ defined by n elements
- a set E with the set of elements that can be selected

```
$s \leftarrow \{\ \}$
For $i$ from 1 to $n$ do
        $R \leftarrow \{e \in E$ such that $f(s \cup \{e\})$ is one of the $r$ best values$\}$
        $e \leftarrow$ random($R$)
        $s \leftarrow s \cup \{e\}$
        $E \leftarrow E \setminus \{e\}$
EndFor
```

s is initialized as empty

the cycle runs n times to select n elements

R contains the r elements e_1 , e_2 , … , e_r from set E whose partial solution $s \cup \{e\}$ has the best optimization value

element e is randomly selected from set R

element e is added to partial solution s

element e is removed from the set E of elements that can be selected

At the end, the Greedy Randomized solution is s

Greedy randomized method with MATLAB in the node optimization problems

```
$s \leftarrow \{\ \}$
**For** $i$ from 1 to $n$ **do**
    $R \leftarrow \{e \in E$ such that $f(s \cup \{e\})$ is one of the $r$ best values$\}$
    $e \leftarrow$ random($R$)
    $s \leftarrow s \cup \{e\}$
    $E \leftarrow E \setminus \{e\}$
**EndFor**
```

Consider a network defined by a graph $G=(N,A)$ and each arc (i,j) has an associated length l_{ij}.

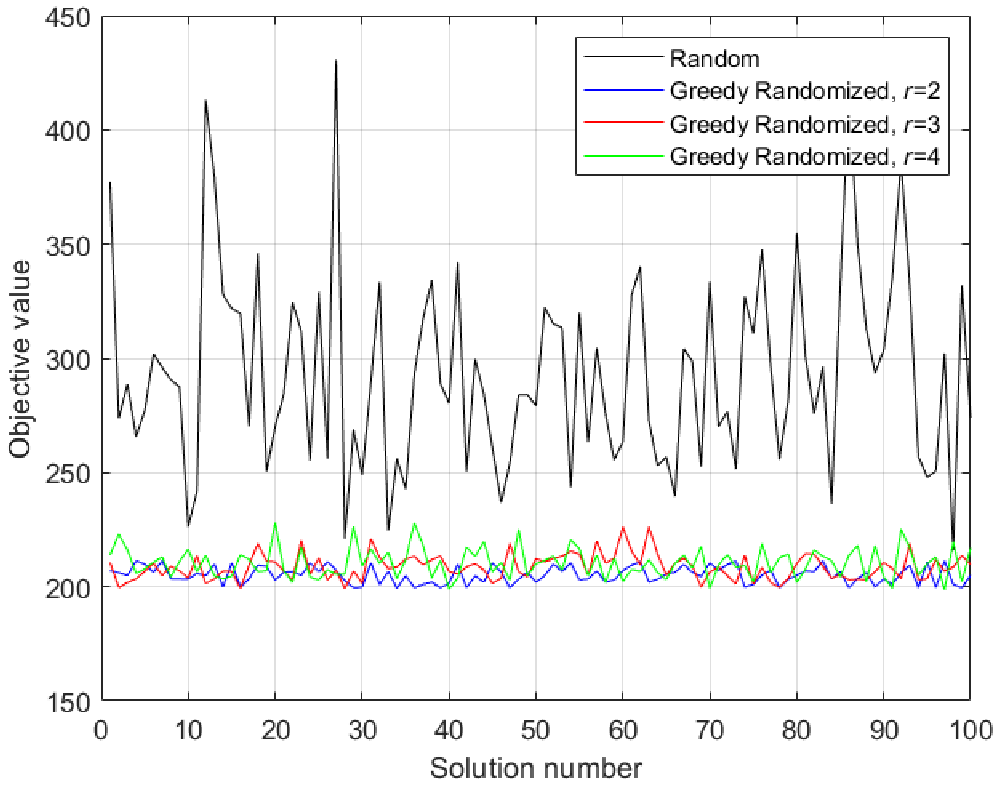
Consider the Server Node Selection problem: to select a number n of nodes minimizing the average shortest path length from each node to its closest selected node.

```
Nodes= load('Nodes.txt');
Links= load('Links.txt');
L= load('L.txt');
G= graph(L);
n= 10; r= 3;

E= 1:numnodes(G);
s= [];
for i= 1:n
    R= [];
    for j= E
        R= [R ; j AverageSP(G,[s j])];
    end
    R= sortrows(R,2);
    e= R(randi(r),1);
    s= [s e];
    E= setdiff(E,e);
end
```

Results of the Greedy and Greedy Randomized algorithms in the Server Node Selection problem

Consider a network defined by a graph G=(N,A) and each arc (i,j) has an associated length l_{ij}. Select a number n = 6 of nodes minimizing the average shortest path length from each node to its closest selected node.

Greedy Randomized algorithm, r = 2, 3 and 4, 100 runs, with Network graph:



Greedy solution: 204.79 (slide 59)

Greedy randomized solutions:

	Min.	Avg.	Max.
r = 2	199.25	205.12	211.37
r = 3	199.15	208.63	226.33
r = 4	198.50	210.42	228.22

- Higher values of r slightly decrease average solution quality but increase solution diversity
- Best approach depends on the specific optimization problem

GRASP
(*Greedy Randomized Adaptive Search Procedure*)

GRASP is a combination of:

- a Greedy Randomized method to compute good initial solutions
- an Adaptive Search method that starts from each initial solution aiming to find its closest local optimum solution.

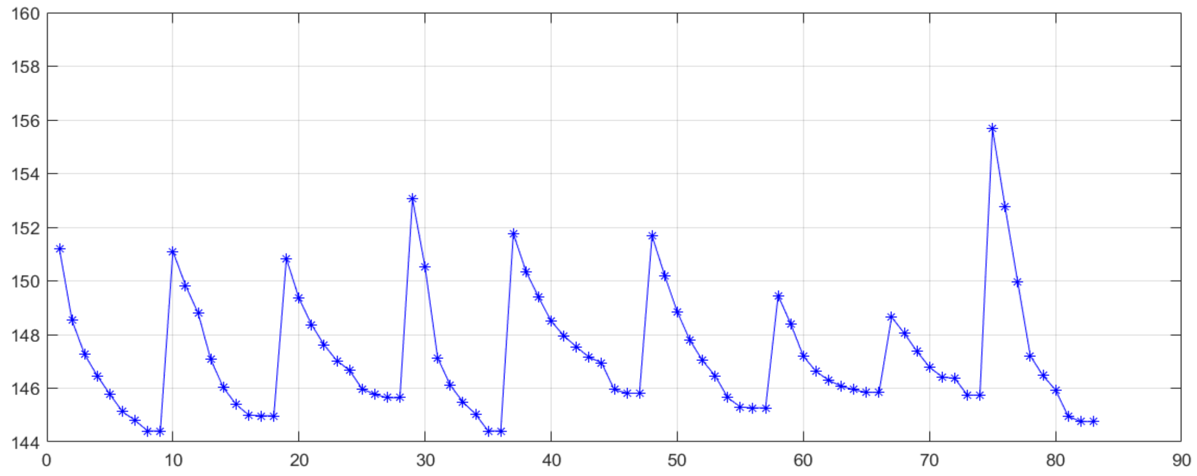
In its original proposal, the Adaptive Search method is the Steepest Ascent Hill Climbing method.

```
$s \leftarrow$ GreedyRandomized()
$s_{best} \leftarrow$ AdaptiveSearch($s$)
**While** not (stopping condition) **do**
    $s \leftarrow$ GreedyRandomized()
    $s \leftarrow$ AdaptiveSearch($s$)
    **If** $f(s)$ better than $f(s_{best})$ **do**
        $s_{best} \leftarrow s$
    **EndIf**
**EndWhile**
```

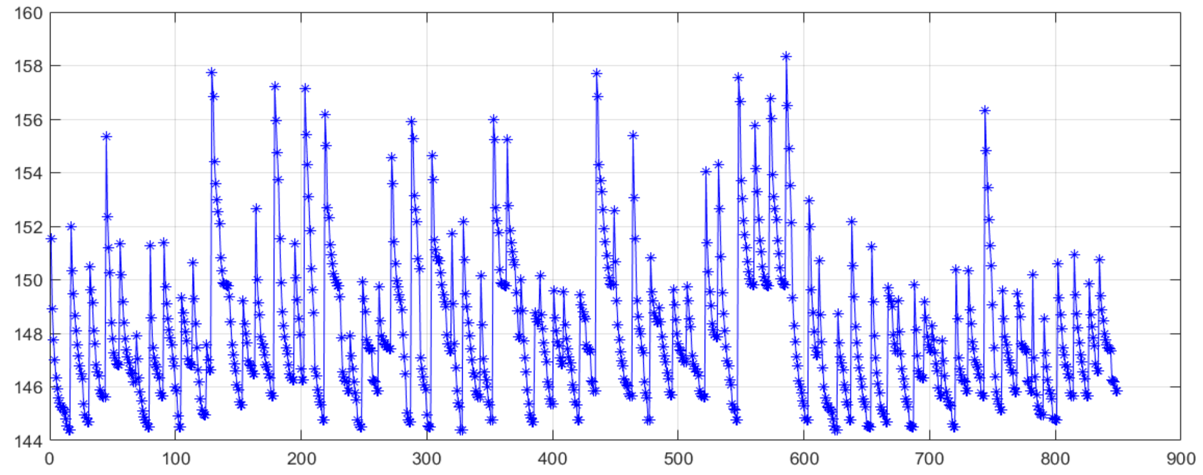
Illustration of GRASP in the Server Node Selection problem

Consider a network defined by a graph $G=(N,A)$ and each arc (i,j) has an associated length l_{ij}. Select a number n of nodes minimizing the average shortest path length from each node to its closest selected node.

GRASP: Net200 (with 200 nodes), n = 10, r = 3, runtime = 120 seconds:



Using neighbor **Definition 1**:
Best solution value = 144.4



Using neighbor **Definition 2**:
Best solution value = 144.4

Variable Neighborhood Metaheuristics

Variable Neighborhood metaheuristics systematically exploit the idea of neighborhood change, both:

- in descend to local minimum (ascent to local maximum): VND
- in escape from the local minimum (maximum): VNS

These metaheuristics rely upon the following observations:

1. A local minimum solution with respect to one neighborhood structure (i.e., definition) is not necessarily a local minimum solution for another (larger) neighborhood structure.
2. A global minimum solution is a local minimum solution with respect to all possible neighborhood structures.
3. For many problems, local minimum solutions with respect to one or several neighborhoods are relatively close to each other.

The last observation is empirical (but true in many known problems). It implies that a local minimum solution often provides some information about the global minimum solution.

Variable Neighborhood Descent (VND)

Consider a minimization problem with a set of feasible solutions S, a minimization function $f(s)$, $s \in S$, and:

- a finite set of pre-selected neighborhood structures, D_k , $k = 1, \ldots, k_{max}$
- $D_k(s)$ is the set of solutions of the kth neighborhood of s

---

The VND is the generalization of the Steepest Descent HC method for multiple neighborhood structures:

- when no better solution exists in a neighborhood structure D_k, move to neighborhood structure D_{k+1}
- when the best solution of a neighborhood structure D_k is better than the current solution, update the current solution and return to neighborhood structure D_k with $k = 1$
- finish if no better solution exists in the last neighborhood structure D_k with $k = k_{max}$ (the current solution is the local minimum solution with respect to all neighborhood structures)

Variable Neighborhood Descent (VND)

Consider a minimization problem with a set of feasible solutions S, a minimization function $f(s)$, $s \in S$, and:

- a finite set of pre-selected neighborhood structures, D_k , $k = 1, \ldots, k_{max}$
- $D_k(s)$ is the set of solutions of the kth neighborhood of s

---

```
$s' \leftarrow$ Random($s \in S$)
$k \leftarrow 1$
While $k \leq k_{max}$ do
    $s \leftarrow$ Best($s \in D_k(s')$)
    If $f(s) < f(s')$ do
        $s' \leftarrow s$, $k \leftarrow 1$
    Else do
        $k \leftarrow k + 1$
    EndIf
EndWhile
```

start with the first neighborhood structure

run until k becomes higher than the number of neighborhood structures

select the best solution s of the current neighborhood structure

if s is better than the current solution s', move s' to s and return to the first neighborhood structure

if s is not better than the current solution s', go to the next neighborhood structure

Variable Neighborhood Search (VNS)

Consider a minimization problem with a set of feasible solutions S, a minimization function $f(s)$, $s \in S$, and:

- a finite set of pre-selected neighborhood structures, V_k , $k = 1, \ldots, k_{max}$
- $V_k(s)$ is the set of solutions of the k^{th} neighborhood of s

---

The VNS exploits the fact that in many cases the local minimum solutions (and, potentially, the global minimum solution) are close to each other:

- take a random solution of the neighborhood structure V_1
- apply a Local Search method (usually Steepest Descend/Ascend HC or VND) to find a local minimum solution
- if the local minimum solution improves the current solution, update the current solution and return to neighborhood structure V_1
- otherwise, move to neighborhood structure V_{k+1}

As opposed to multi-start methods with initial random (or greedy randomized) solutions, VNS escapes from local minimum solutions selecting a new random solution closer to the current best one

Variable Neighborhood Search (VNS)

Consider a minimization problem with a set of feasible solutions S, a minimization function $f(s)$, $s \in S$, and:

- a finite set of pre-selected neighborhood structures, V_k , $k = 1, \ldots, k_{max}$
- $V_k(s)$ is the set of solutions of the kth neighborhood of s

---

```
$s' \leftarrow$ Random($s \in S$)
$k \leftarrow 1$
**While** $k \le k_{max}$ **do**
    $s \leftarrow$ Random($s \in V_k(s')$)
    $s \leftarrow$ LocalSearch($s$)
    **If** $f(s) < f(s')$ **do**
        $s' \leftarrow s$, $k \leftarrow 1$
    **Else do**
        $k \leftarrow k + 1$
    **EndIf**
**EndWhile**
```

start with the first neighborhood structure

run until k becomes higher than the number of neighborhood structures

select a random solution s of the current neighborhood structure

improve solution s with a Local Search method

if s is better than the current solution s', move s' to s and return to the first neighborhood structure

if s is not better than the current solution s', go to the next neighborhood structure

VND and VNS
Number of Neighborhood Structures

Recall:

- In VND, we must select a finite set of neighborhood structures D_k
- Similarly, in VNS, we must select a finite set of neighborhood structures V_k

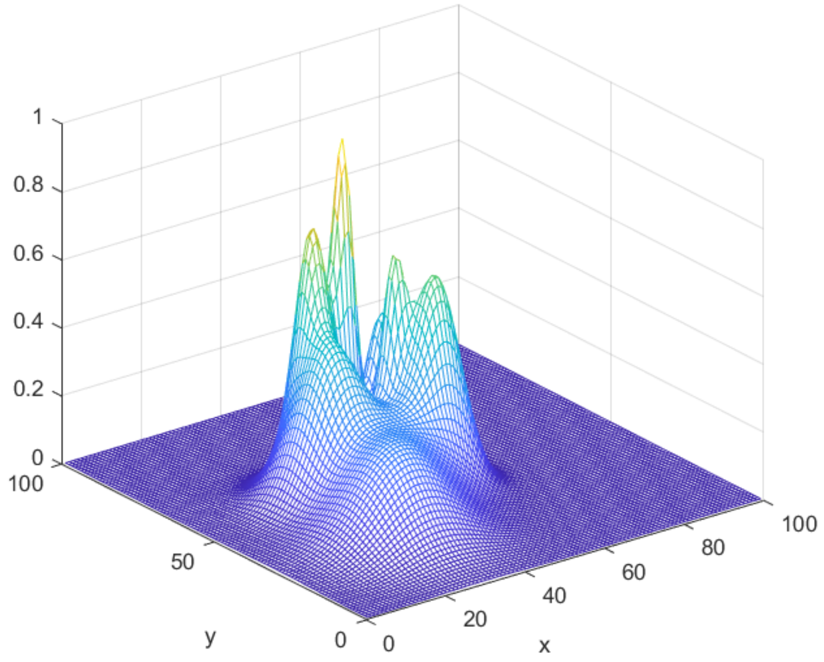
The best selection of the neighborhood structures depends on the specific optimization problem.

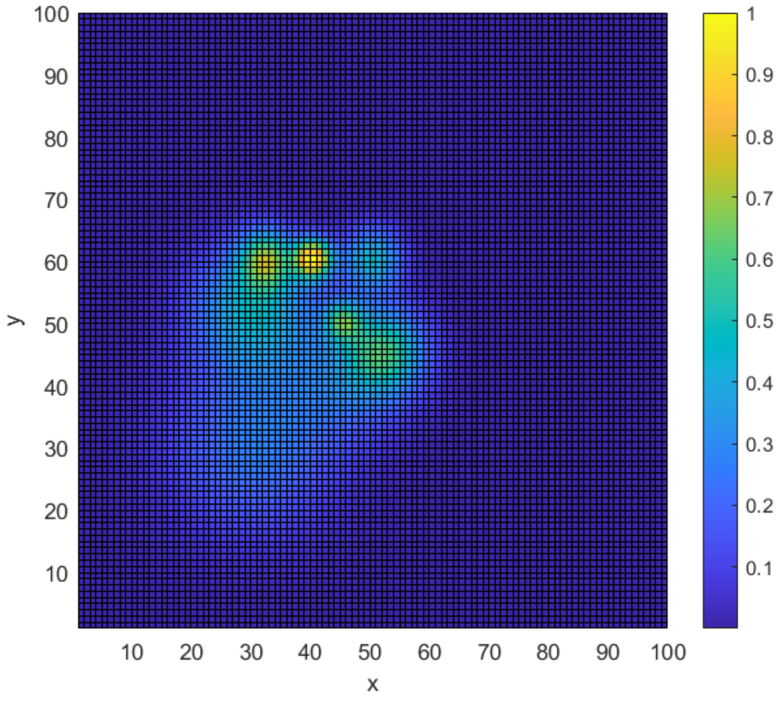
The neighborhood structures and their number are usually different for VNS and VND:

- usually, a small number (2 or 3) is used for VND to obtain efficient algorithms with reasonable runtime values
- depending on the specific optimization problem, a significant large number of neighborhood structures might be required to make VNS efficient in escaping from local minimum/maximum solutions

VND versus GRASP

Consider an optimization problem with 2 integer variables *x* and *y* (both between 0 and 100) and an objective function *f*(*x*,*y*) to be maximized:

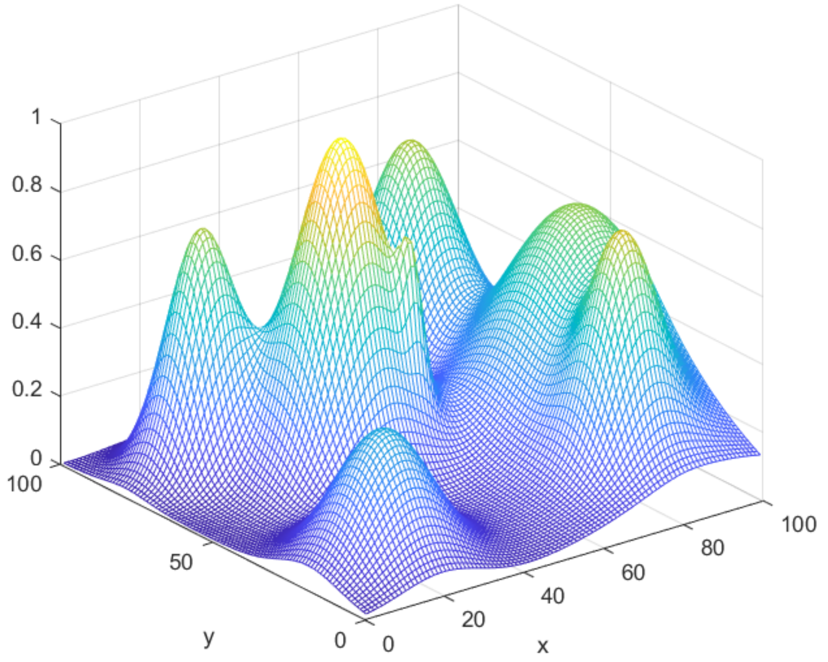


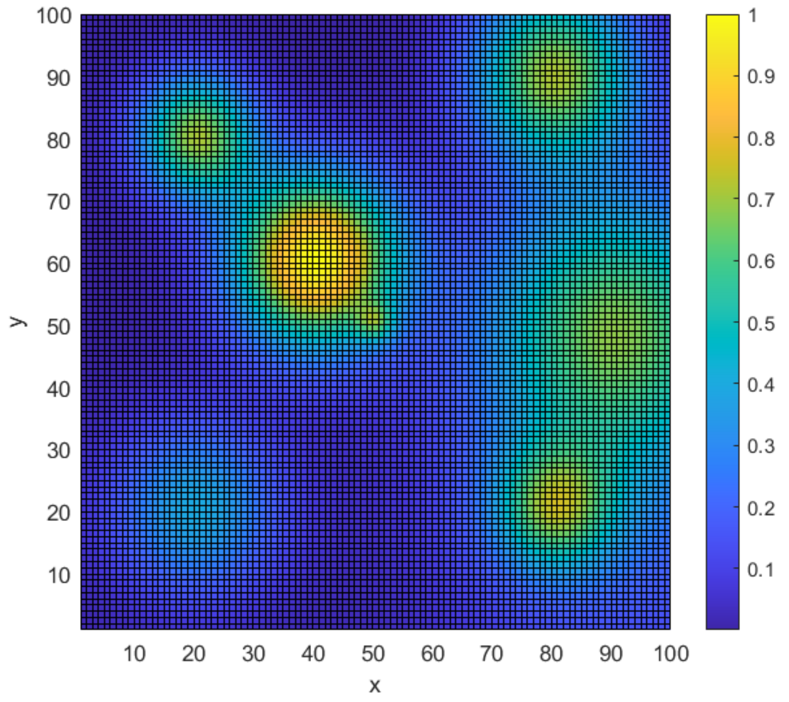


In this example, <u>VND is likely to be more efficient than GRASP</u> as most of the local optimum solutions are relative close to each other

VND versus GRASP

Consider an optimization problem with 2 integer variables *x* and *y* (both between 0 and 100) and an objective function *f*(*x*,*y*) to be maximized:





In this example, <u>GRASP is likely to be more efficient than VND</u> as the local optimum solutions are far away from each other

Usage of VND and VNS

Recall:

- VND is a local search method that improves a given initial solution by finding its closest local optimal solution
- When the search finds a local optimum solution, VNS is a method to compute a new random initial solution in a neighborhood set of the local optimum solution

The two methods can be combined between each other or combined with other methods:

- VNS/VND metaheuristic – VND is used as the Local Search method of VNS
- GRASP/VND metaheuristic – VND is used as the Adaptive Search method of GRASP

Practical Assignment 6

Consider the Server Node Selection problem: to select a number *n* of nodes minimizing the average shortest path length from each node to its closest selected node.

- Develop a function implementing a multi-start SA-HC algorithm (using neighbor Definition 1) with initial random solutions:
  - use as stopping criteria a runtime limit (defined as an input parameter),
  - besides the best solution found and its objective value, the function must also output the total number of iterations (the sum of the number of iterations of all SA-HC runs) and the runtime when the best solution was found.
- Develop a script to run the algorithm with a runtime limit of 30 seconds, for *n* = 12 nodes in Net300 graph (input files: `Nodes300.txt`, `Links300.txt` and `L300.txt`).
- Run the script and compare the obtained results with the ones obtained in the Practical Assignments 4 and 5.

Practical Assignment 7

Consider the Server Node Selection problem: to select a number *n* of nodes minimizing the average shortest path length from each node to its closest selected node.

- Develop a function implementing a GRASP algorithm (based on the SA-HC with neighbor Definition 1) with initial greedy randomized solutions (consider *r* as an input parameter):
  - use as stopping criteria a runtime limit (defined as an input parameter),
  - besides the best solution found and its objective value, the function must also output the total number of iterations run (the sum of the number of iterations of all SA-HC runs) and the runtime when the best solution was found.
- Extend the script developed in the previous assignment to run also the GRASP algorithm with a runtime limit of 30 seconds, for *n* = 12 nodes in Net300 graph and for three values of *r* (*r* = 2, 3 and 5).
- Run the script, compare the results and the runtimes obtained by the multi-start SA-HC and the GRASP algorithms and take conclusions.

CLASS 4

Population-based metaheuristic methods

Population-based metaheuristic methods

Recall:

- Single-solution based methods: focus on identifying, modifying and improving a single solution at each iteration
- Population-based methods: maintain and improve multiple solutions (named a population of solutions) at each iteration

Population-based metaheuristics covered in this course unit:

- Genetic algorithms
- Particle swarm algorithms
- Memetic algorithms

Genetic algorithm

A **genetic algorithm (GA)** is a metaheuristic inspired by the process of natural selection from Darwin's theory of evolution.

Consider an optimization problem with a set S of feasible solutions and an optimization function $f(s)$, $s \in S$.

At any moment, the algorithm considers a population of individuals, such that:

- an individual represents a solution $s \in S$ of the optimization problem and is characterized by a set of genes (each one with an assigned allele from a set of possible alleles);
- each individual s has a fitness value given by the objective function value $f(s)$.

Genetic algorithm

A **genetic algorithm (GA)** is a metaheuristic inspired by the process of natural selection from Darwin's theory of evolution.

Basic idea of a GA:

- The algorithm starts with a population of (usually) randomly generated individuals
- The algorithm is an iterative process where, at each iteration, a new population is generated based on the current population:
  - the individuals from the current population are mated to generate the individuals of the next population;
  - some of the most fit individuals might survive, i.e., might remain in the next population (elitist selection).
- The new population is then used in the next iteration of the algorithm.
- The algorithms ends when a stopping criteria is met (running time, total number of iterations, consecutive number of iterations without improvement, etc).

Genetic algorithm

There are 2 main genetic operators in generating new individuals:

Crossover: two parent individuals are selected and their genes are combined to generate the genes of an offspring individual.

Mutation: with a given probability, one gene of the offspring individual is randomly mutated (i.e., changed); the mutation probability should be small (otherwise, the method turns into a random search).

To generate an offspring individual, a pair of parent individuals is selected for breeding (using Crossover and Mutation):

– the parents are randomly selected (with higher probability to the most fit individuals);

– the generated offspring individual is a mix of the parent genes (but Mutation maintains genetic diversity between generations).

While generations evolve, GA is efficient if:

– on one hand, the average fitness of the generations improves;

– on the other hand, the individuals of each generation do not become too similar between them (premature convergence).

Genetic algorithm (basic)

```
$P \leftarrow$ Compute $|P|$ random individuals
**While** not (stopping condition) **do**
    $P' \leftarrow \{\}$
    **For** $i = 1 \ldots |P|$ **do**
        $s \leftarrow$ Crossover($P$)
        **If** random([0,1]) $< q$
            $s \leftarrow$ Mutation($s$)
        **EndIf**
        $P' \leftarrow P' \cup \{s\}$
    **EndFor**
    $P \leftarrow P'$
**EndWhile**
$s_{best} \leftarrow$ Best($P$)
```

Population *P* is initialized with |*P*| random individuals (|*P*| is a parameter of GA)

The next population *P'* is initialized empty

An individual *s* is generated by the Crossover operator

With probability *q*, individual *s* suffers a Mutation (*q* is a parameter of GA)

Individual *s* is added to the new population *P'*

The new population *P'* becomes the current population *P*

The best individual s_{best} of the last population is the final result of the algorithm

Genetic algorithm (improved)

```
$P \leftarrow$ Compute $|P|$ random individuals
$s_{best} \leftarrow$ Best($P$)
**While** not (stopping condition) **do**
    $P' \leftarrow \{\}$
    **For** $i = 1 \ldots |P|$ **do**
        $s \leftarrow$ Crossover($P$)
        **If** random([0,1]) $< q$
            $s \leftarrow$ Mutation($s$)
        **EndIf**
        $P' \leftarrow P' \cup \{s\}$
    **EndFor**
    $P \leftarrow P'$
    $s \leftarrow$ Best($P$)
    **If** $f(s)$ is better than $f(s_{best})$ **do**
        $s_{best} \leftarrow s$
    **EndIf**
**EndWhile**
```

Offspring individuals *s* are not necessarily better than parent individuals

Consequently, the best individual of the next population is not necessarily better than the best individual of the current population

In this algorithm variant, the final result s_{best} is guaranteed to be the best solution among all populations

Genetic algorithm (with elitist selection)

```
$P \leftarrow$ Compute $|P|$ random individuals
**While** not (stopping condition) **do**
    $P' \leftarrow \{\}$
    **For** $i = 1 \ldots |P|$ **do**
        $s \leftarrow$ Crossover($P$)
        **If** random([0,1]) $< q$
            $s \leftarrow$ Mutation($s$)
        **EndIf**
        $P' \leftarrow P' \cup \{s\}$
    **EndFor**
    ~~$P \leftarrow P'$~~
    $P \leftarrow$ Selection($P$,$P'$)
**EndWhile**
$s_{best} \leftarrow$ Best($P$)
```

Elitism: the most fit individuals of a current population survive to the next population.

Selection operator:

- Selects the $|P|$ individuals s from set $P \cup P'$ with the best value of $f(s)$
- limiting the number of elitist individuals (i.e., elements of P) to a maximum value m (m is a parameter of GA)

The best individual s_{best} of the last population is the final result of the algorithm

GA: Crossover operator

The Crossover operator (that combines the genes of two parent individuals to generate the genes of an offspring individual) is composed by two steps: parent selection and gene combination.

Parent selection: the aim is to select two random parents from the current population giving higher probability to the most fit individuals. Typical strategies:

- Fitness based selection: select each parent with a probability proportional to its fitness
- Tournament selection: for each parent, select two random individuals and choose the most fit among the two as the parent
- Rank based selection: rank all individuals by their fitness value and randomly select each parent with a probability proportional to its rank

Gene combination: compute an offspring whose genes are randomly set with the alleles of one of the parents.

Particle swarm algorithm

A **particle swarm algorithm (PSA)** is a population-based metaheuristic inspired by social behavior among individuals, where each individual:

- acts based on its current situation,
- is influenced by its own past experience,
- is influenced by the past experience of all individuals.

While GA is more appropriate to address discrete optimization problems (problems where decision variables can only assume a discrete set of values), PSA is more appropriate to address continuous optimization problems (problems where decision variables can assume continuous values).

Particle swarm algorithm

- The algorithm considers a swarm of particles (the number of particles of the swarm is a parameter of PSA).
- Each particle position of the swarm represents a solution of the optimization problem being solved.
- At each iteration, the algorithm moves the particles around in the solution space according to the particle's position and velocity.
- Each particle's movement is simultaneously guided by:
  - its local best known position, i.e., the best solution (of the optimization problem being solved), that the particle has found so far;
  - the best known position (i.e. solution) found by all particles of the swarm.
- This behavior is expected to move the swarm toward the global optimal solution.

Particle swarm algorithm

Consider an optimization problem with n continuous variables x_j, $j = 1…n$.

- The set of variables are represented in vector form as $\boldsymbol{x}$

The domain of each variable is: $x_{j,lo} \leq x_j \leq x_{j,hi}$.

- The lower and upper limits of the variables are represented in vector form as $\boldsymbol{x}_{lo}$ and $\boldsymbol{x}_{hi}$, respectively

Function $f(\boldsymbol{x})$ is the objective function to be minimized.

Consider the following PSA notation:

s	the number of particles (i.e., the swarm size)
$\boldsymbol{x}_i$	the current position vector of particle $i = 1…s$
$\boldsymbol{v}_i$	the current velocity vector of particle $i = 1…s$
$\boldsymbol{p}_i$	the best particle position (i.e., the best position found so far by particle $i = 1…s$)
$\boldsymbol{g}$	the best swarm position (i.e., the best position found so far among all particles of the swarm)

Particle swarm algorithm

```
For each particle $i = 1 \ldots s$ do
    $\boldsymbol{x}_i \leftarrow \text{Random}(\boldsymbol{x}_{lo}, \boldsymbol{x}_{hi})$
    $\boldsymbol{v}_i \leftarrow \text{Random}(-|\boldsymbol{x}_{hi} - \boldsymbol{x}_{lo}|, |\boldsymbol{x}_{hi} - \boldsymbol{x}_{lo}|)$
    $\boldsymbol{p}_i \leftarrow \boldsymbol{x}_i$
EndFor
$\boldsymbol{g} \leftarrow \text{Best}(\boldsymbol{p}_i, i = 1 \ldots s)$
While not (stopping condition) do
    For each particle $i = 1 \ldots s$ do
        $r_p \leftarrow \text{Random}(0,1)$
        $r_g \leftarrow \text{Random}(0,1)$
        $\boldsymbol{v}_i \leftarrow \omega\, \boldsymbol{v}_i + \phi_p\, r_p\, (\boldsymbol{p}_i - \boldsymbol{x}_i) + \phi_g\, r_g\, (\boldsymbol{g} - \boldsymbol{x}_i)$
        $\boldsymbol{x}_i \leftarrow \boldsymbol{x}_i + \boldsymbol{v}_i$
        If $f(\boldsymbol{x}_i) < f(\boldsymbol{p}_i)$ do
            $\boldsymbol{p}_i \leftarrow \boldsymbol{x}_i$
            If $f(\boldsymbol{p}_i) < f(\boldsymbol{g})$ do
                $\boldsymbol{g} \leftarrow \boldsymbol{p}_i$
            EndIf
        EndIf
    EndFor
EndWhile
```

The initial position $\boldsymbol{x}_i$ and velocity $\boldsymbol{v}_i$ vectors of each particle *i* are randomly generated

The best position $\boldsymbol{p}_i$ of each particle *i* is initialized with its initial position

The best swarm position $\boldsymbol{g}$ is initialized with the best of all initial positions

The velocity of each particle *i* is computed based on a weighted sum of (1) its current velocity, (2) a velocity vector towards its best position and (3) a velocity vector towards the best swarm position

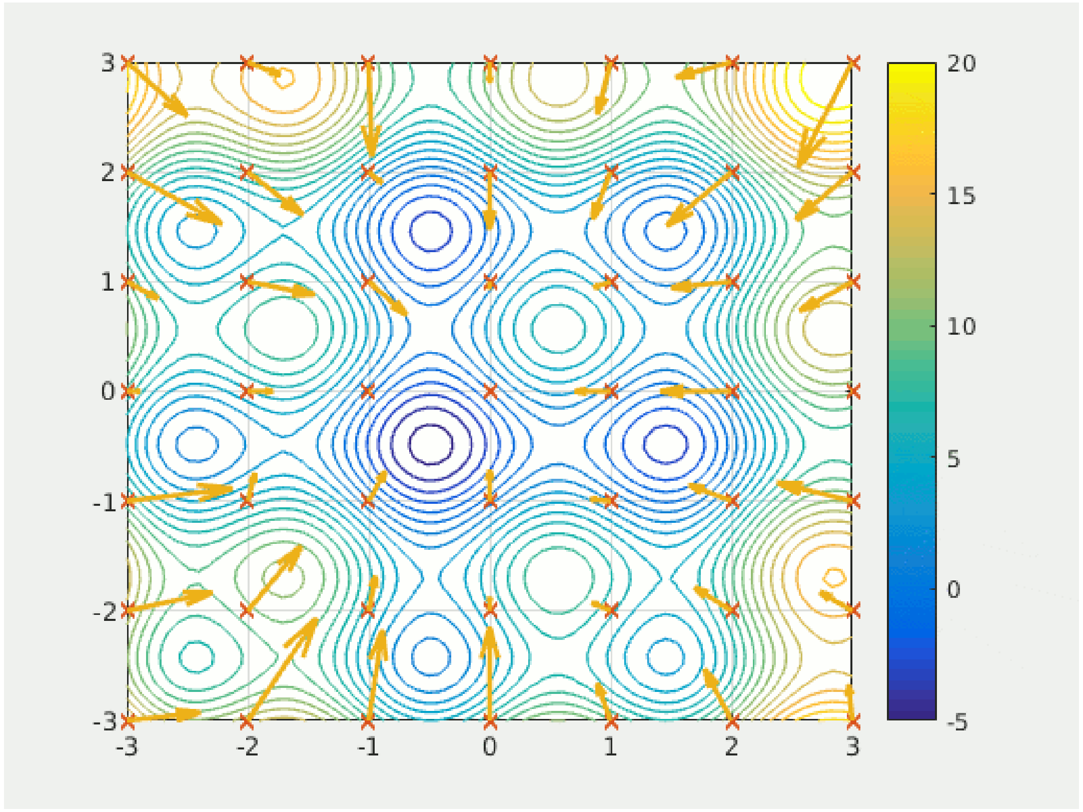
The position $\boldsymbol{x}_i$ of each particle *i* is updated

The best position $\boldsymbol{p}_i$ of each particle *i* is updated if new position $\boldsymbol{x}_i$ is better than the current $\boldsymbol{p}_i$

The best swarm position $\boldsymbol{g}$ is updated if $\boldsymbol{p}_i$ is updated and is better than the current $\boldsymbol{g}$

At the end, the final best swarm position $\boldsymbol{g}$ is the final result of the algorithm

Particle swarm algorithm (animation)



Source: https://en.wikipedia.org/wiki/Particle_swarm_optimization

PSA evolves with the expectation that the swarm explores the neighborhood of different local optimum solutions and, sooner or later, many of the particles concentrate their exploration on the neighborhood of the global optimal solution.

Particle swarm algorithm

- The weight parameters used to compute the velocity $\boldsymbol{v}_i$ vector of each particle $i = 1 \ldots s$ on each iteration:

  $$\boldsymbol{v}_i \leftarrow \omega\, \boldsymbol{v}_i + \phi_p\, \boldsymbol{r}_p\, (\boldsymbol{p}_i - \boldsymbol{x}_i) + \phi_g\, \boldsymbol{r}_g\, (\boldsymbol{g} - \boldsymbol{x}_i)$$

  can have a significant impact on the optimization performance:

  – the inertia weight ω controls the inertia of the velocity

  – the acceleration weight ϕ_p controls the importance of moving the particle towards its best found position (to further explore its neighbor solutions)

  – the acceleration weight ϕ_g controls the importance of moving the particle towards the best swarm position

- Selecting the number of particles and the weights that yield good performance require testing on each specific optimization problem.
- PSA is particular suitable for optimization problems with continuous variables and adaptations are required for problems with discrete variables.

Particle swarm algorithm: topologies

The topology of the swarm defines the subset of particles with which each particle exchanges information.

- The basic version of the algorithm, described in the previous slides, uses a **global topology**:
  - it considers that all particles communicate with all the other particles and, thus, the whole swarm shares the same best position.
- However, the global topology might trap the swarm into a local minimum solution (i.e., local best position).

In **local topologies**, each particle only shares information with a subset of other particles. Subset examples of local topologies:

- Geometrical subset: each particle shares information with the *k* nearest particles (*k* is a parameter of the algorithm)
- Social subset: the particles are initially clustered in social sets; then, each particle shares information with the particles of its social set:
  - equivalent to parallel runs of a global topology
  - however, different social sets can be associated with different inertia and acceleration weights

Memetic algorithm

A **memetic algorithm (MA)** is, in general terms, a metaheuristic which combines a population-based metaheuristic with a single-solution based metaheuristic.

- Usually, a MA takes as its basis a population-based metaheuristic and improves it with a single-solution based metaheuristic.
- The most usual MA is to improve a GA (Genetic Algorithm) with a local search heuristic like HC (Hill Climbing) or, more recently, VND (Variable Neighborhood Descent).
- One possible approach: all individuals of each population of a GA are optimized by local search.
- The aim is to increase the rate at which the populations of the GA converge to contain good solutions.

Memetic algorithm (with elitist selection)

```
For $i = 1 \ldots |P|$ do
    $s \leftarrow$ RandomIndividual()
    $s \leftarrow$ LocalSeach($s$)
    $P \leftarrow P \cup \{s\}$
EndFor
While not (stopping condition) do
    $P' \leftarrow \{\}$
    For $i = 1 \ldots |P|$ do
        $s \leftarrow$ Crossover($P$)
        If random(0,1) < $q$
            $s \leftarrow$ Mutation($s$)
        EndIf
        $s \leftarrow$ LocalSeach($s$)
        $P' \leftarrow P' \cup \{s\}$
    EndFor
    $P \leftarrow$ Selection($P$,$P'$)
EndWhile
$s_{best} \leftarrow$ Best($P$)
```

Each individual of the initial population *P* is first randomly generated and then improved by the local search heuristic.

Each offspring individual (generated by the Crossover and Mutation operators) is first improved by the local search heuristic before being inserted in the next population

Memetic algorithm

Properties of the MA approach presented in the previous slide:

In the GA perspective:

- it reduces the slow pace of population improvement given by only using random processes subjected to fitness-based selection,
- therefore, it increases the rate at which the populations converge to contain good solutions,
- moreover, it requires less overall iterations to obtain good solutions (but, on the other hand, each population takes longer running times as Local Search is applied to all individuals)

In the Local Search perspective:

- it allows the Local Search to benefit from the solution diversity given by a population of solutions (i.e., the GA is seen as a multi-start approach applied to the Local Search method)

Memetic algorithm

Threads of the MA approach presented in the previous slides:

- optimizing all individuals (by local search) of a population might be computational expensive,
- and might lead to populations whose individuals are too similar between them (premature convergence).

To minimize these threads:

- instead of improving all individuals of each population, only some individuals are selected to be improved;
- at each population, the individuals to be improved are randomly selected with higher probability to the most fit ones;
  - recall that good initial solutions improve the performance of single-solution based heuristics
- the selection of these individuals can use one of the methods described for the Crossover operator in a GA (fitness based method, tournament method or rank based method).

Practical Assignment 8

- Develop a function implementing a genetic algorithm (GA) with elitism to solve the Server Node Selection problem:
  - consider the population size $|P|$, the mutation probability q and the elitist parameter m as input parameters,
  - consider the tournament method for the Crossover operator and use as stopping criteria a runtime limit (also defined as an input parameter),
  - besides the best solution found and its objective value, the function must also output the total number of generated populations and the time instant at which the best solution was found.
- Develop a script to run the GA for n = 12 nodes in Net300 topology (input files: `Nodes300.txt`, `Links300.txt` and `L300.txt`) with a running time limit of 30 seconds, a population $|P|$ = 100 individuals, a mutation probability q = 10% and three different values for the elitist parameter (m = 1, 5 and 50).
- Compare the results obtained by the GA for the three values of m and take conclusions. Compare also these results and runtimes with the ones obtained in Practical Assignment 7 and take conclusions.