Bachelor's degree in Bioinformatics

Heuristic Algorithms for Combinatorial Optimization

Prof. Renato Bruni

bruni@dis.uniroma1.it

Department of Computer, Control, and Management Engineering (DIAG)

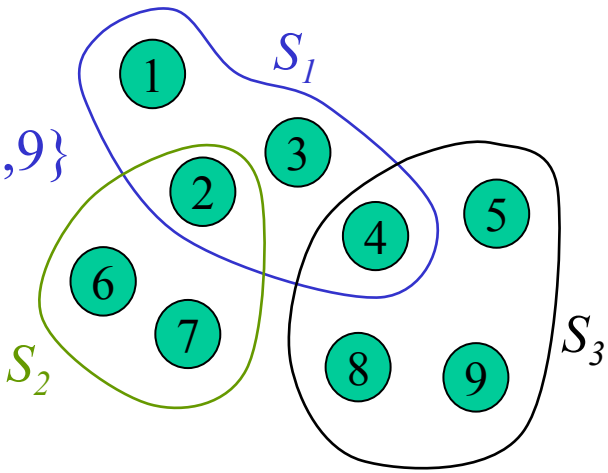
"Sapienza" University of Rome

Combinatorial Optimization

Combinatorial Optimization is a vast class of problems arising in many practical fields. They all share the following basic **mathematical structure**:

- There is a **Base set**: $B=\{b_1, \ldots, b_n\}$
  (elementary decisions, ex: project i activated, connection j established, etc.)
- Every **feasible solution** is a subset $S \subseteq B$ (set of elementary decisions which meet the feasibility conditions)
- The set of feasible solutions is $\mathcal{S}=\{S_1, \ldots, S_m\} \subseteq \mathcal{P}(B)$ (cardinality 2^n)
- Every element b_i of B has its cost c_i
- **Cost function** $c: S \rightarrow \mathrm{R}$ (often linear, and we speak of CO with linear objective: the sum of the c_i of the elements b_i in S)

$$c(S_1)=\sum_{i \in S_1} c_i$$

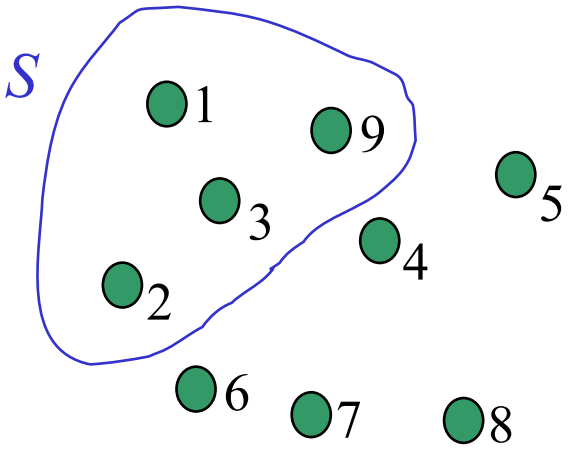


All in all: $\boxed{min\ \{c(S):\ S \in \mathcal{S}\}}$

OC can be modeled with PL01

- For example, the base set is $B=\{1,2,\ldots,9\}$
- a feasible solution is $S=\{1,2,3,9\}\in\mathcal{S}$
  - We **represent** S with a binary vector called incidence vector $x^S=$

$$\begin{bmatrix}1\\1\\1\\0\\0\\0\\0\\0\\1\end{bmatrix}$$



Then we have

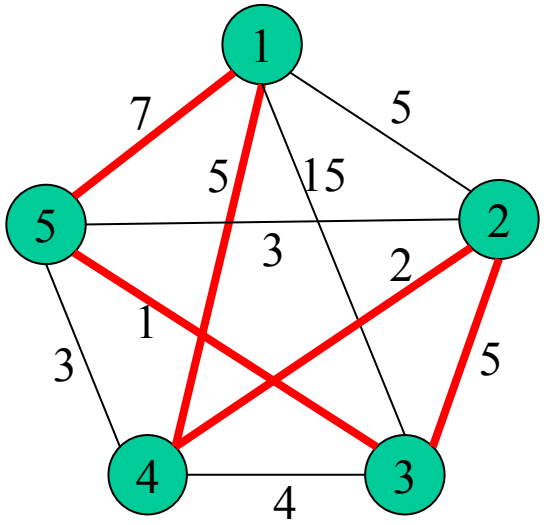
OC with linear objective and PL01

$\{subsets\}=\mathcal{S}$	$\{incidence\ vectors\}=V$
$c(S)$	c^Tx
$min\ \{c(S):\ S\in\mathcal{S}\}$	$min\ \{c^Tx : x\in V\subseteq\{0,1\}^n\}$

Every OC with linear obj can be written as PL01

Example: Traveling Salesman Problem

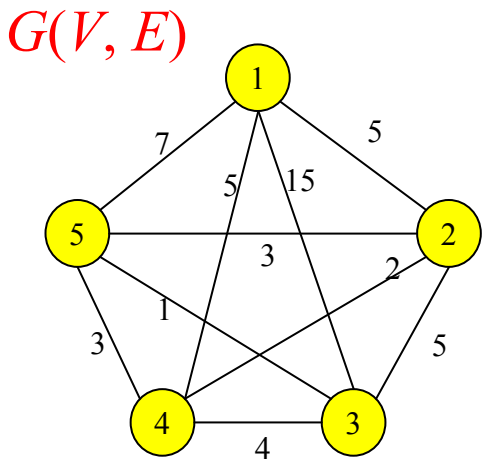
- A CO problem is the **Traveling Salesman Problem** (**TSP**): visit **all costumers** and come back to the **origin** following the **less expensive** route
- Many practical problems share this structure (ex: move a robot or a machinery to work on several points, pass a wire to connect several points, etc.)
- We have a **cost** for every possible "connection". It must represent what we really want to minimize (distance, time, etc.)
- The cost of the cycle is the sum of the costs of the used connections (in the example 7+5+2+5+1 = 20)



Minimum cost Hamiltonian Cycle

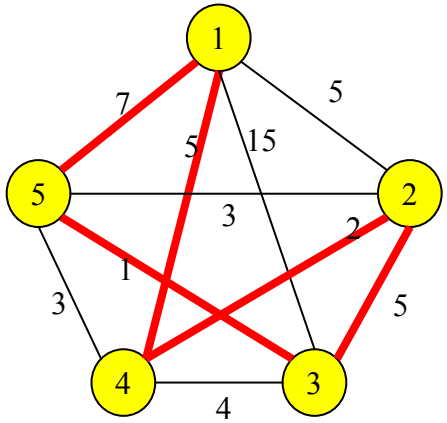
- Defined over a complete graph $G(N,E)$ with costs associated to arcs
- A **Hamiltonian** cycle is a cycle passing once for each node. The cost of the cycle is the sum of the costs of its arcs
- The Traveling Salesman Problem, **TSP** consists in finding a **Hamiltonian cycle with minimum cost**
- It is a computationally difficult problem (**NP-hard**) and it is usually solved by approximate methods called **heuristics**
- Heuristics cannot **guarantee** to find the optimal solution, they find a feasible solution which should be near to the optimum but there is no guarantee. However, this techniques are **fast**
- When a problem is **too difficult** (=requires too much computational time) to be solved by **exact algorithms** (= algorithms which guarantee to find the optimal solution) we can use **heuristics** (better than nothing)

TSP is CO with linear objective



It is a combinatorial optimization problem:

- the **base set** is the set of arcs
- every feasible solution (Hamiltonian cycle) is a **subset** of the base set
- the cost of a solution is the **sum of the costs of its components**



$T = \{(1,4), (4,2), (2,3), (3,5), (5,1)\}$

$c(T) = 5+2+5+1+7 = 20$

Greedy Algorithm (for min)

- We know the base set $B = \{1,2,\ldots,n\}$
- We can check if a solution is feasible $\mathcal{S} = \{T_1, T_2, \ldots, T_m\}$ $(T \subseteq B)$

We define "partial solution" a set of elements which can become a feasible solution by only adding elements: $H \subseteq T$

- *Cost of a partial solution* $H = \boxed{c(H) = \sum_{e \in H} c(e)}$

Greedy Algorithm (can be adapted to every CO problem)

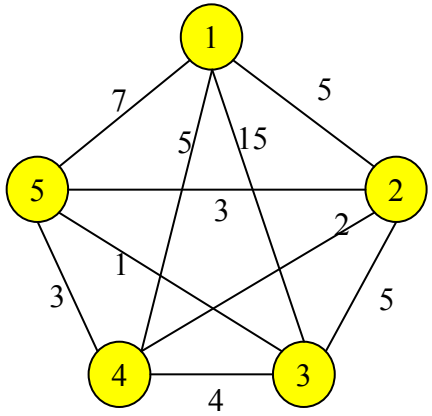
Build a sequence of partial solutions H_0 H_1 H_2 H_3 ... :

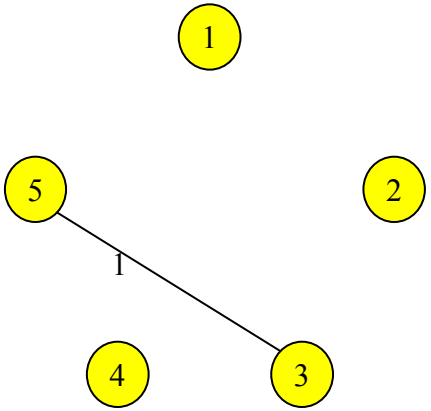
a. Starting from the empty set (*partial solution* H_0)

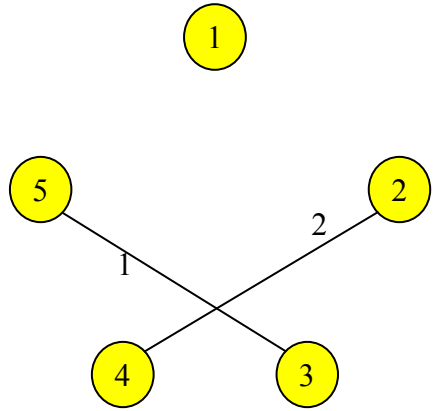
b. Adding, at each step, the less expensive element such that: it is not already taken and it still produces a partial solution

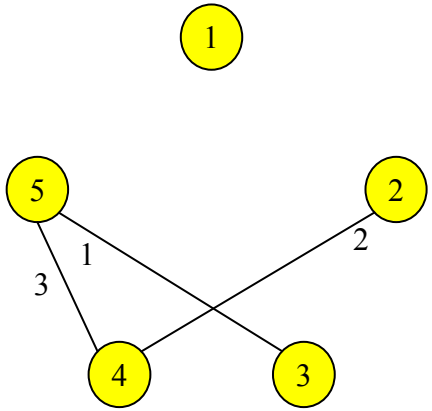
c. Stop when the partial solution becomes a feasible solution (or when it worsens, for problems where the empty solution is already feasible)

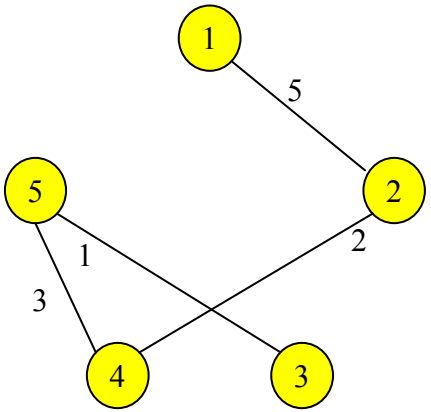
Example: Greedy algorithm for TSP

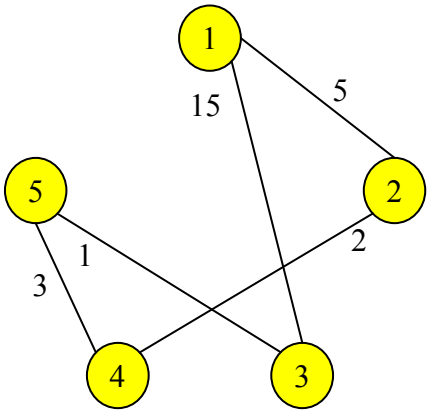












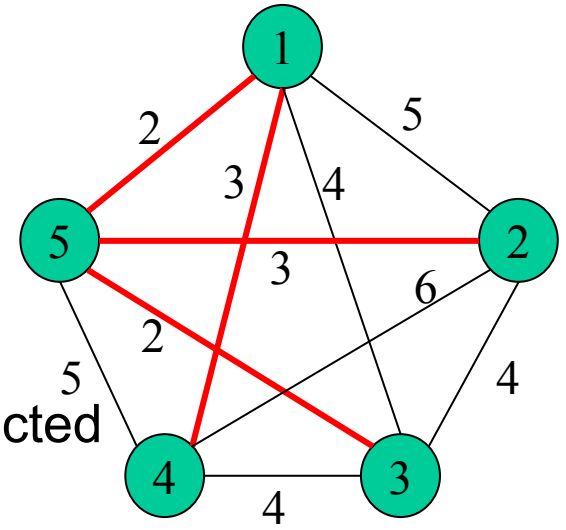
$c(T) = 26$

Observations on Greedy

- It is a **constructive** heuristic (it builds a solution)
- When an element is selected, it can never be **abandoned**
- This makes the algorithm **fast**, however choosing at each step the **cheaper** element may force to take **costly** elements later on (in the example, we are forced in the end to take 15, the most costly arc in the graph)
- So, **no guarantee** of optimality. Sometimes we are lucky and we have an optimal (but without knowing it) or almost optimal solution. In other cases, we may obtain poor solutions, but at least they are feasible ones (better than nothing)

Another example: Min. Spanning Tree

- Another important example of Combinatorial Optimization with linear objective function is the problem of **Minimum Spanning Tree** (**MST**): reach each customer by using the less expensive connections. This problem also represents many practical problems (e.g., pass electric wires, etc)



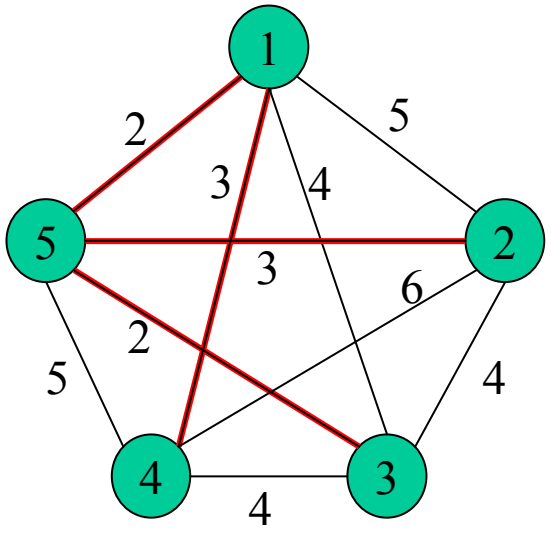
- Can be seen on a graph $G=(V,E)$ with costs c_i associated to arcs
- It is another **CO problem**, the base set is E
- A feasible solution S is a **spanning tree** (connected and acyclic subgraph reaching all nodes)
- The cost function is $\sum_{i \in T_k} c_i$ (in the example 10)

Greedy for the MST

A version of the Greedy algorithm called Kruskal algorithm:

- Order the arcs by increasing cost
- At every iteration, take an arc. If it **does not creates cycles** along with the arcs already selected, add it to the partial solution
- Stop when all nodes have been reached (= we have $|V| - 1$ arcs)

Example: select (1,5), (3,5), (1,4), (2,5) now we have 4 arcs: stop $c(S) = 10$

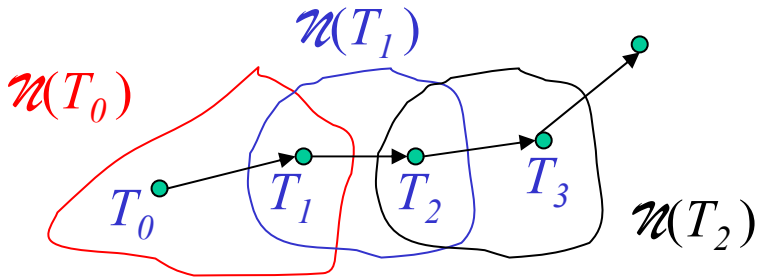


- MST has a particular mathematical structure called matroid. In this cases, greedy algorithm **guarantees optimality!**
- There exist other problems with this nice feature

Local Search (for min)

- Consider the set of feasible solutions $\mathcal{S} = \{T_1, T_2, \ldots, T_m\}$ $(T_i \subseteq B)$
- For every feasible solution T we define a *neighborhood* $\mathcal{N}(T) \subseteq \mathcal{S}$

(a set of feasible solutions **similar** to T)



Build a sequence of feasible solutions $T_0, T_1, T_2, T_3 \ldots$:

a. Starting from an initial feasible solution T_0

b. At every step k, we select the **best solution** T_k (the one with minimum value) in the neighborhood $\mathcal{N}(T_{k-1})$ of the current solution T_{k-1}

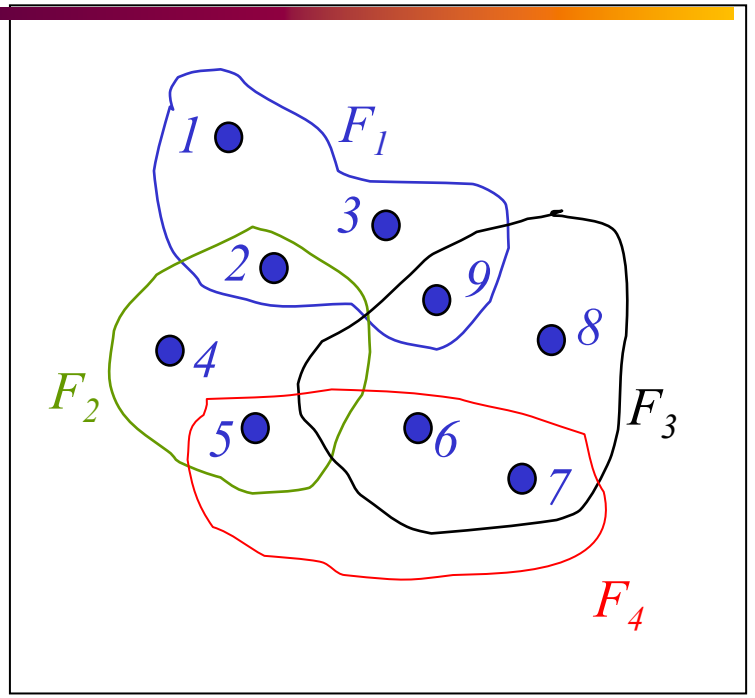
c. Stop when the best solution T_k of the neighborhood $\mathcal{N}(T_{k-1})$ is **worse** (greater) than T_{k-1} , that means $c(T_{k-1}) < c(T_k)$

Neighborhood relations

Examples

$\mathcal{N}(F_1)=\{F_2, F_4\}$ $\mathcal{N}(F_2)=\{F_1, F_4\}$

$\mathcal{N}(F_4)=\{F_1, F_2, F_3\}$ $\mathcal{N}(F_3)=\{F_2\}$



- We need to define a neighborhood system, that is a criterion to obtain solutions similar to a given one. There exist many of them:

$F \in \mathcal{N}_+(F_i) \Leftrightarrow F = F_i \cup \{k : k \notin F_i, F \in \mathcal{S}\}$ *greedy"* neighborhood

$F \in \mathcal{N}_-(F_i) \Leftrightarrow F = F_i - k : k \in F_i, F \in \mathcal{S}$ *"reverse greedy"*

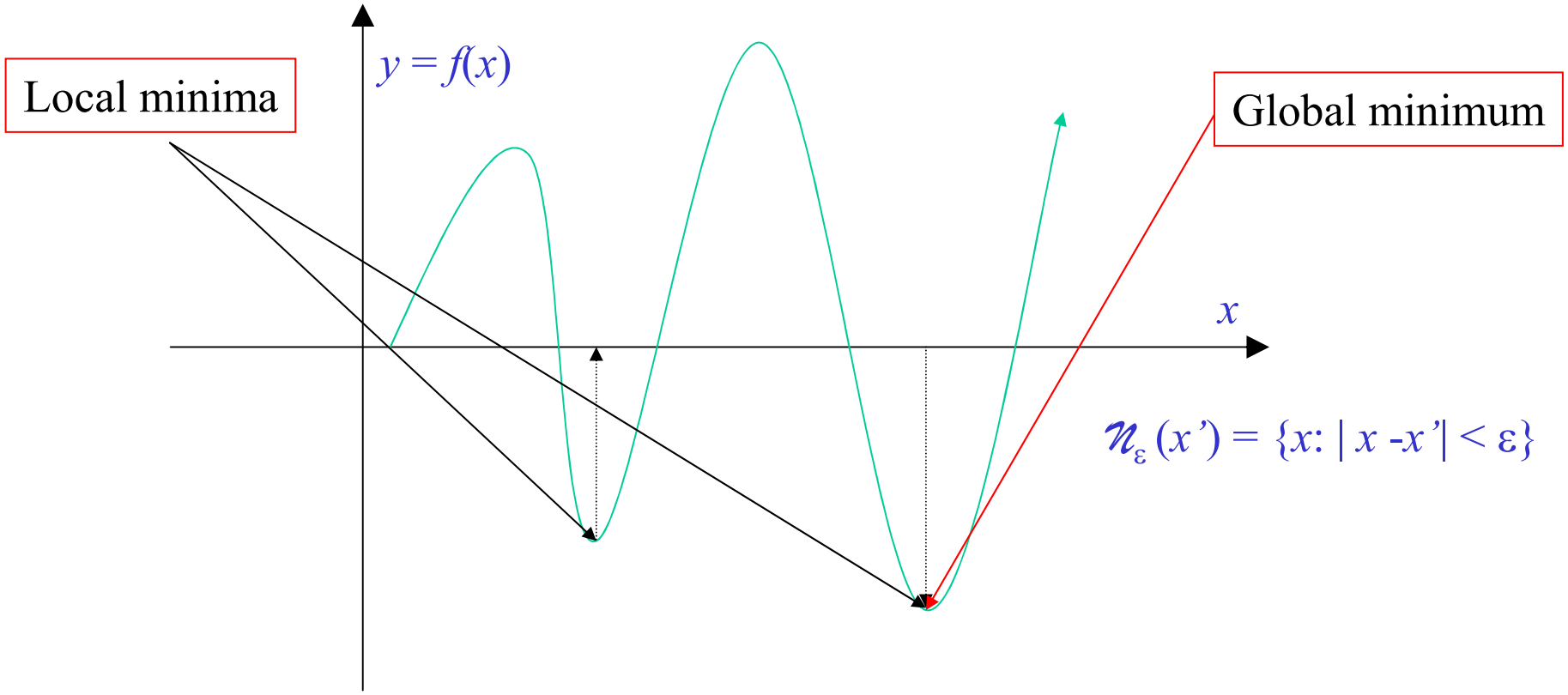
$F \in \mathcal{N}_s(F_i) \Leftrightarrow F_i - \{k\} \cup \{j\} : k \in F_i, j \notin F_i, F \in \mathcal{S}$ *"exchange"*

$F \in \mathcal{N}_s(F_i) \Leftrightarrow F_i - \{h,k\} \cup \{j,i\} : h,k \in F_i, j,i \notin F_i, F \in \mathcal{S}$ *"2-exchange"*

Global and Local minima

F^* global minimum $\Leftrightarrow$ $c(F^*) \leq c(F)$ *for every* $F \in \mathcal{S}$

F^* local minimum $\Leftrightarrow$ $c(F^*) \leq c(F)$ *for every* $F \in \mathcal{N}(F^*)$



Local Search stops in a **local minimum**

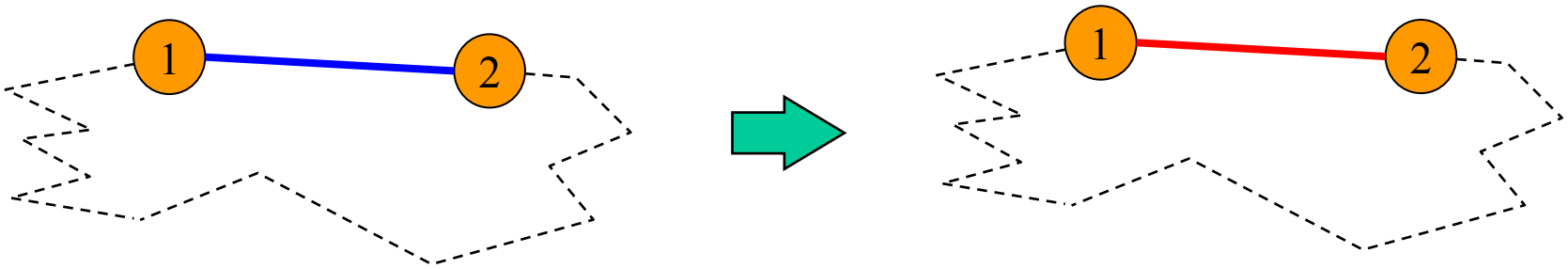
Which neighborhood for TSP?

When we have a Hamiltonian cycle, how can we build **other similar** Hamiltonian cycles?

•If we **add** an arc to a Hamiltonian cycle, we **don't** have another Hamilt.cycle

•If we **remove** an arc, again we **don't** obtain an Hamiltonian cycle

•If we **remove** one arc and **add** another arc, again we **don't** obtain it. We have a Hamiltonian cycle only if we add the same arc that was removed, but that is not a neighborhood: it's the same solution!

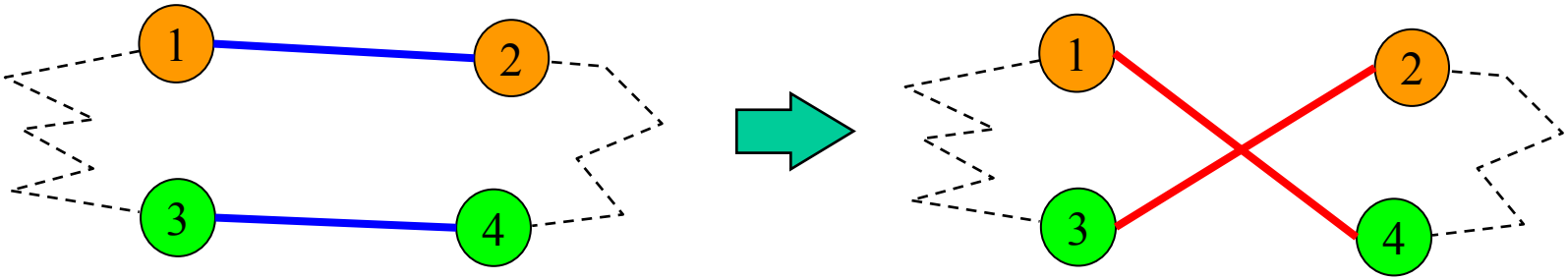


- So how can we build a neighborhood for TSP ?

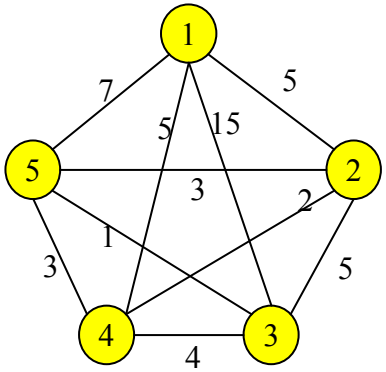
2-exchange neighborhood for TSP

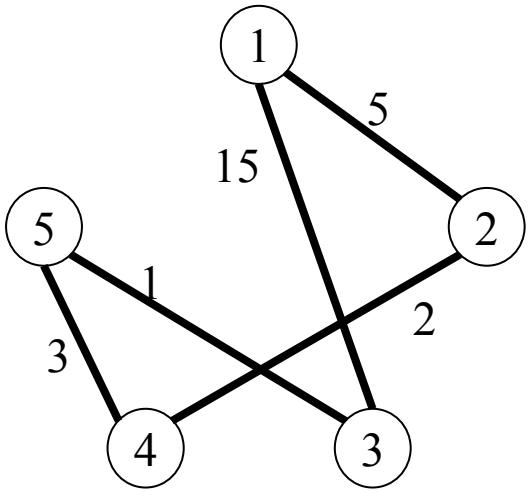
If we **remove 2 arcs** non contiguous and **add** the other 2 arcs **cross-connecting** the free extremes, then we have a Hamiltonian cycle!

This can be done in several ways: for each couple of non-contiguous arcs

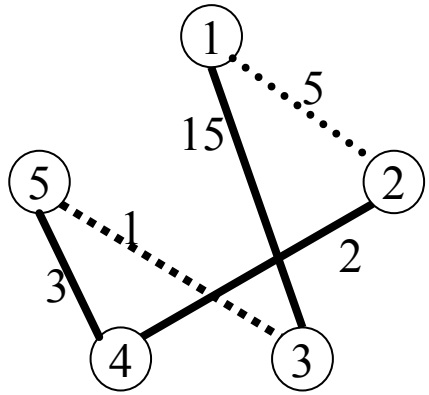


2-exchange (1/2)

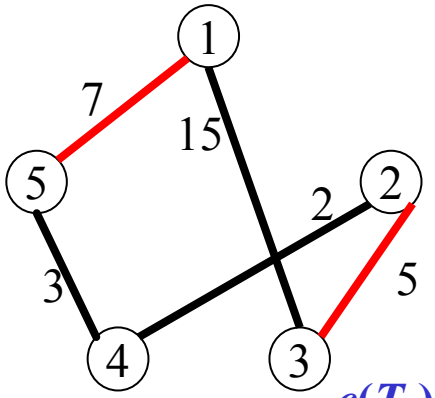




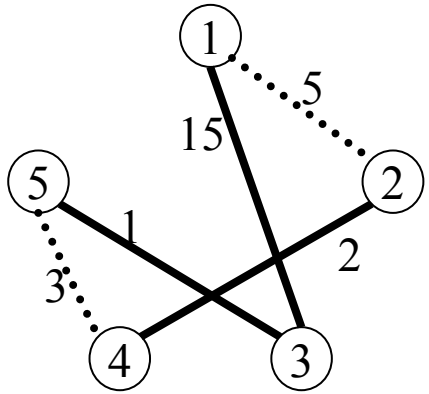
$c(T) = 26$



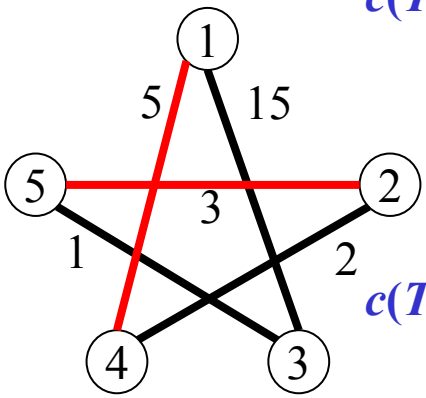
T_1



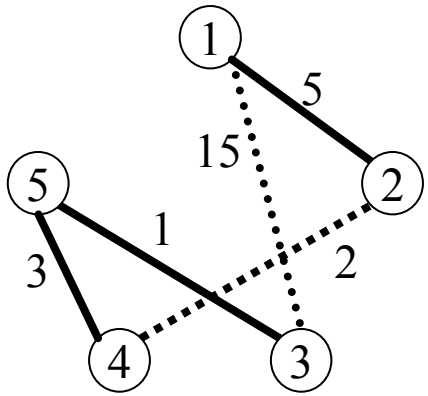
$c(T_1) = 32$



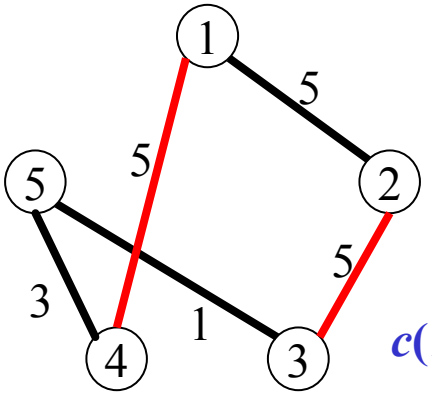
T_2



$c(T_2) = 26$

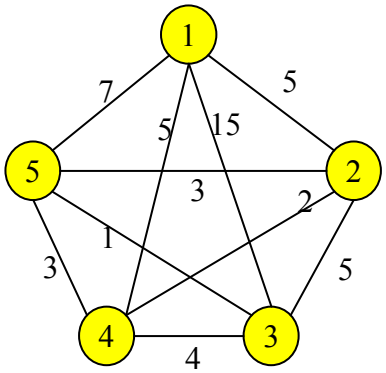


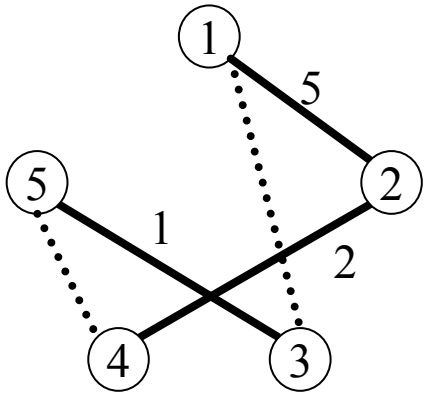
T_3



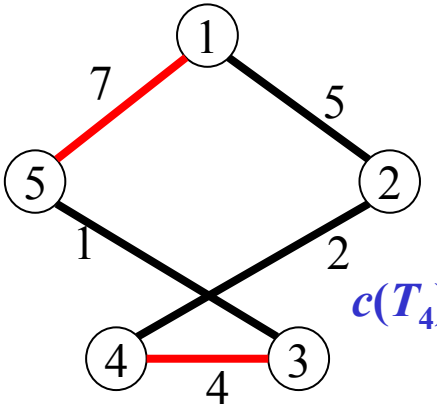
$c(T_3) = 19$

2-exchange (2/2)

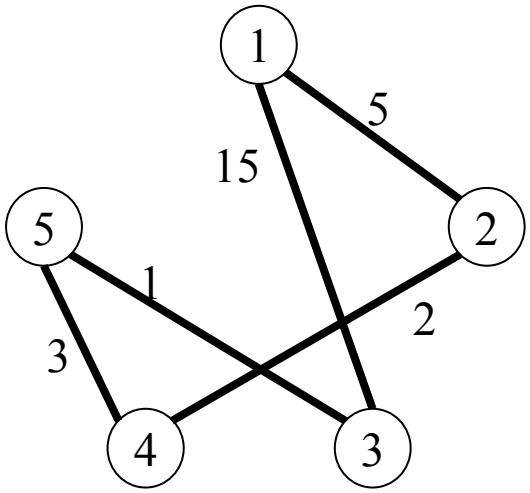




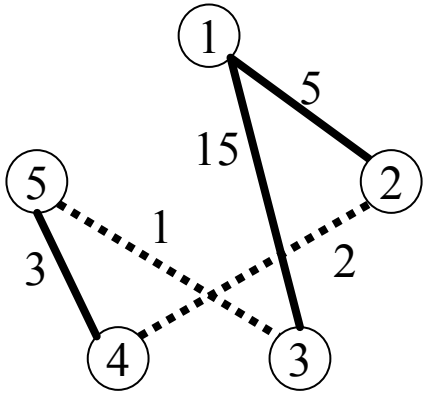
T_4



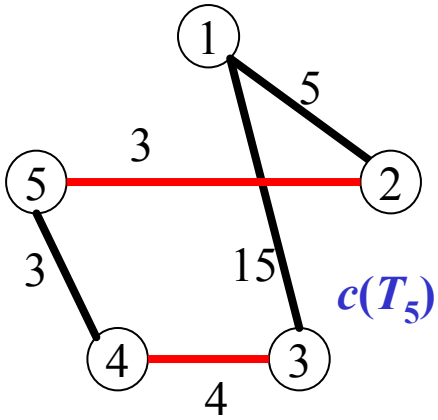
$c(T_4) = 19$



$c(T) = 26$



T_5



$c(T_5) = 30$

No other possibilities for 2-exchange (AKA **2-opt**): this is all $N(T)$

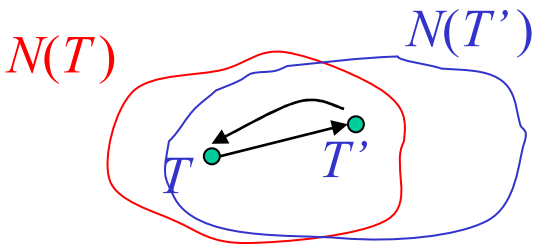
- Best solution in $N(T)$ is T_4 (or T_3, they are equivalent)

Observations on Local Search

- It is an **improvement** heuristic (needs an initial solution)
- Hence, often used after Greedy
- If the neighborhood is **small**, exploring it is **fast** but the probability of improvement is **small**
- On the contrary, If the neighborhood is **large**, the probability of improvement is **high** but exploring the neighborhood may be **slow**
- The choice also depends on the **time** we can **invest** in solving our problem

Improving Local Search?

- **What if** we don't stop when the best solution T' in the neighborhood $N(T)$ is worse than the current solution T ?
- We could do this, hoping to find **something even better** later on…
- It would take more time but may provide a better solution
- However, there is a problem: the best solution in the neighborhood $N(T')$ is very likely to be T, so we keep bouncing between solution T and solution T'



What can be done to avoid being trapped?

The so-called Taboo search

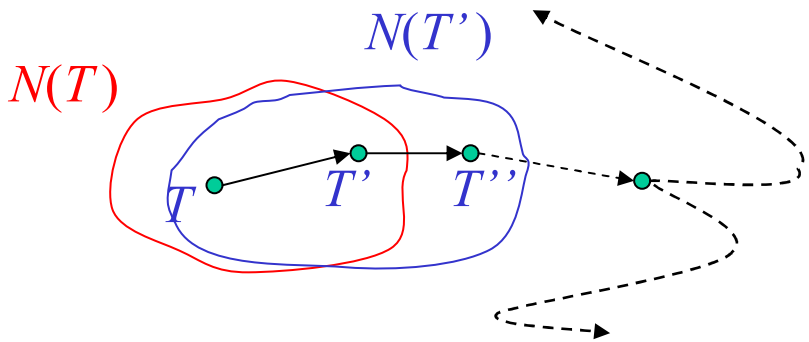
Taboo Search

- **Taboo Search**

  Identify the operation (move) used to pass from one solution to the next one (example, from *T* to *T'*). Then, undoing that move becomes forbidden (taboo) Not forever, otherwise we are too constrained in the end, but for a certain number of steps (we have a taboo list)

  Now, we can accept moving on worse solutions, under suitable conditions, in order to avoid the trap of local minima

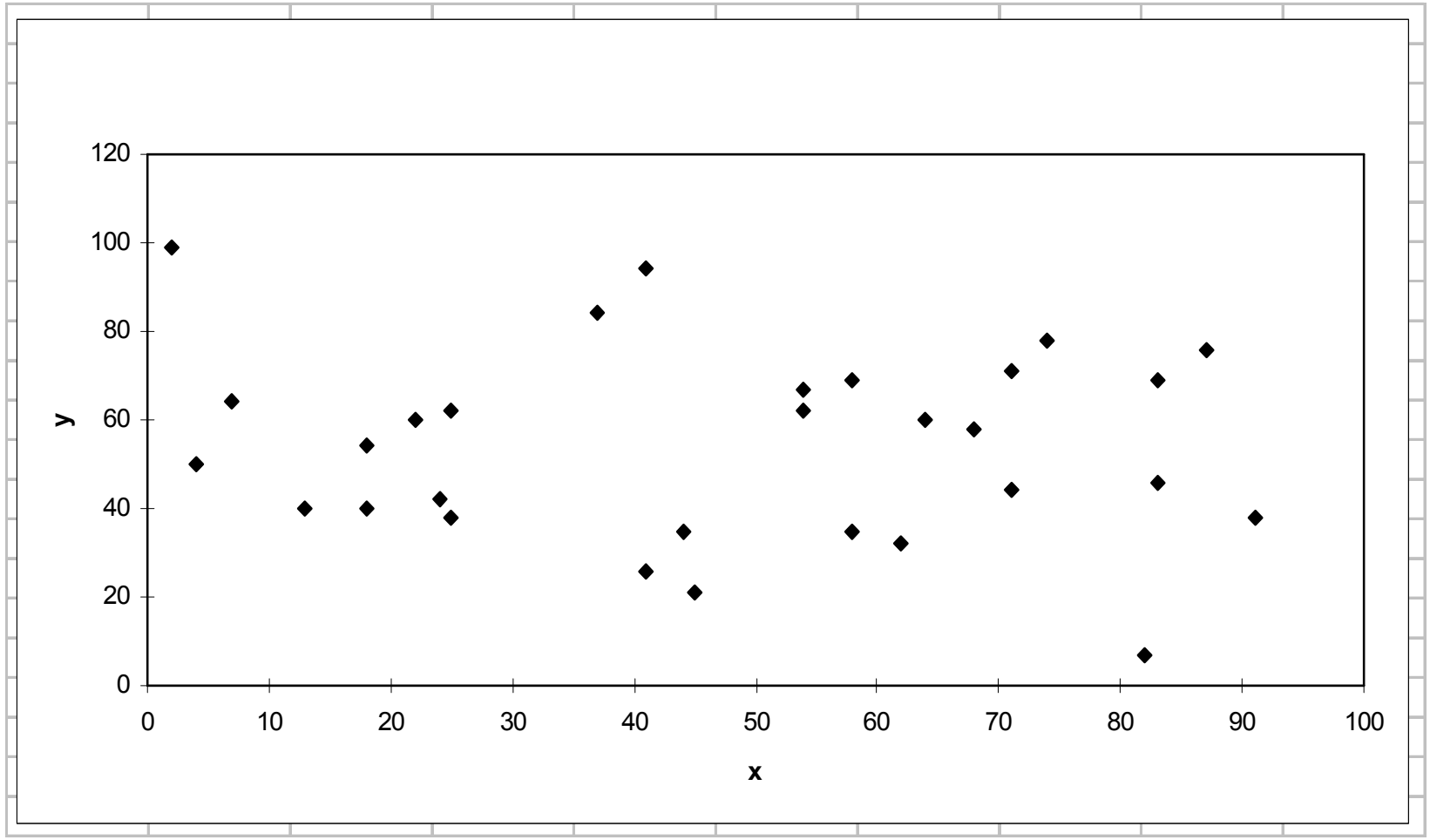
  But we need the taboo to keep exploring the solution space



How long must the taboo list be? Depends on the problem, we must find good values by doing experiments for our case…
Again, several modifications can be introduced

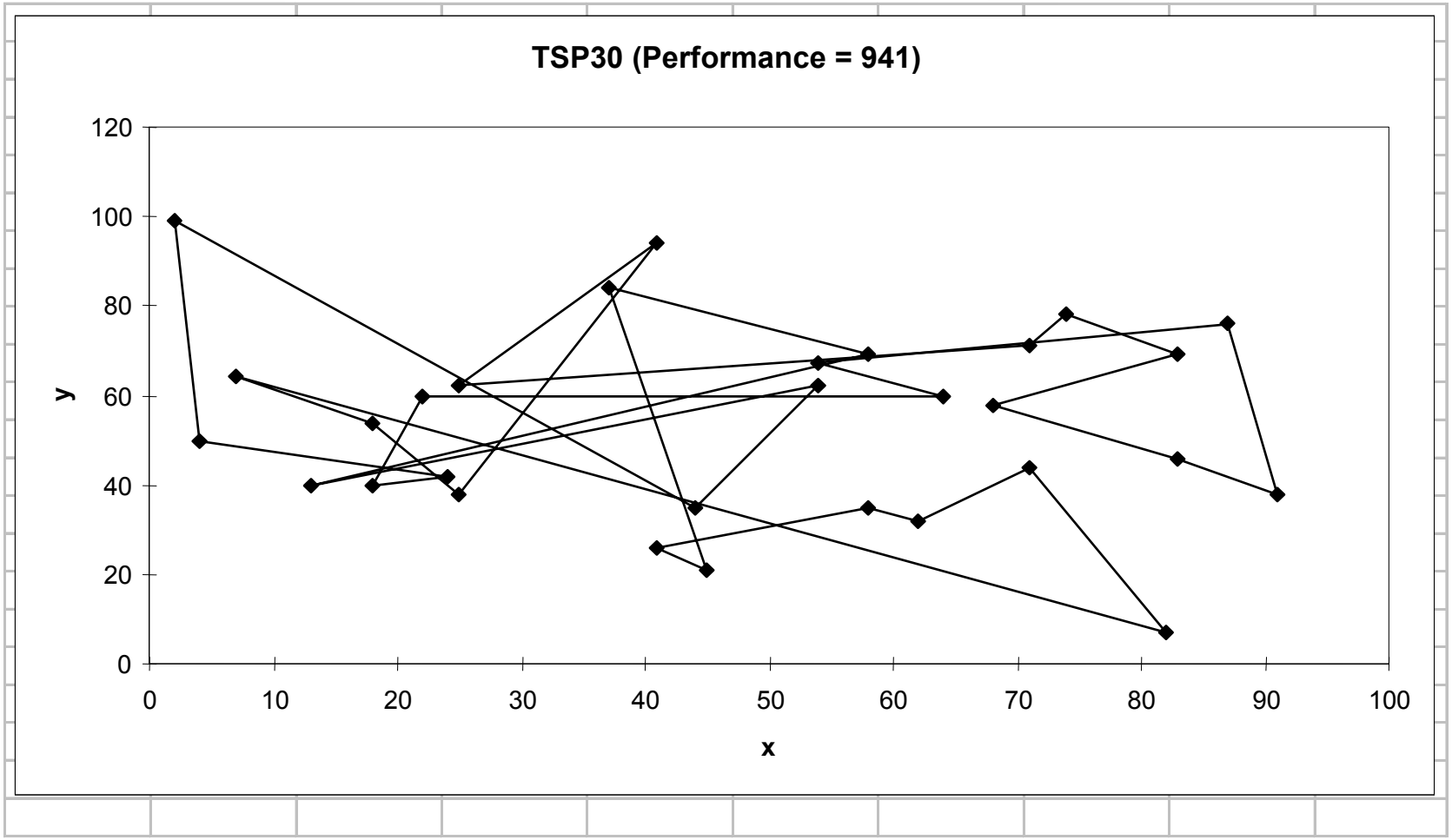
Example of application on TSP

TSP example with 30 cities



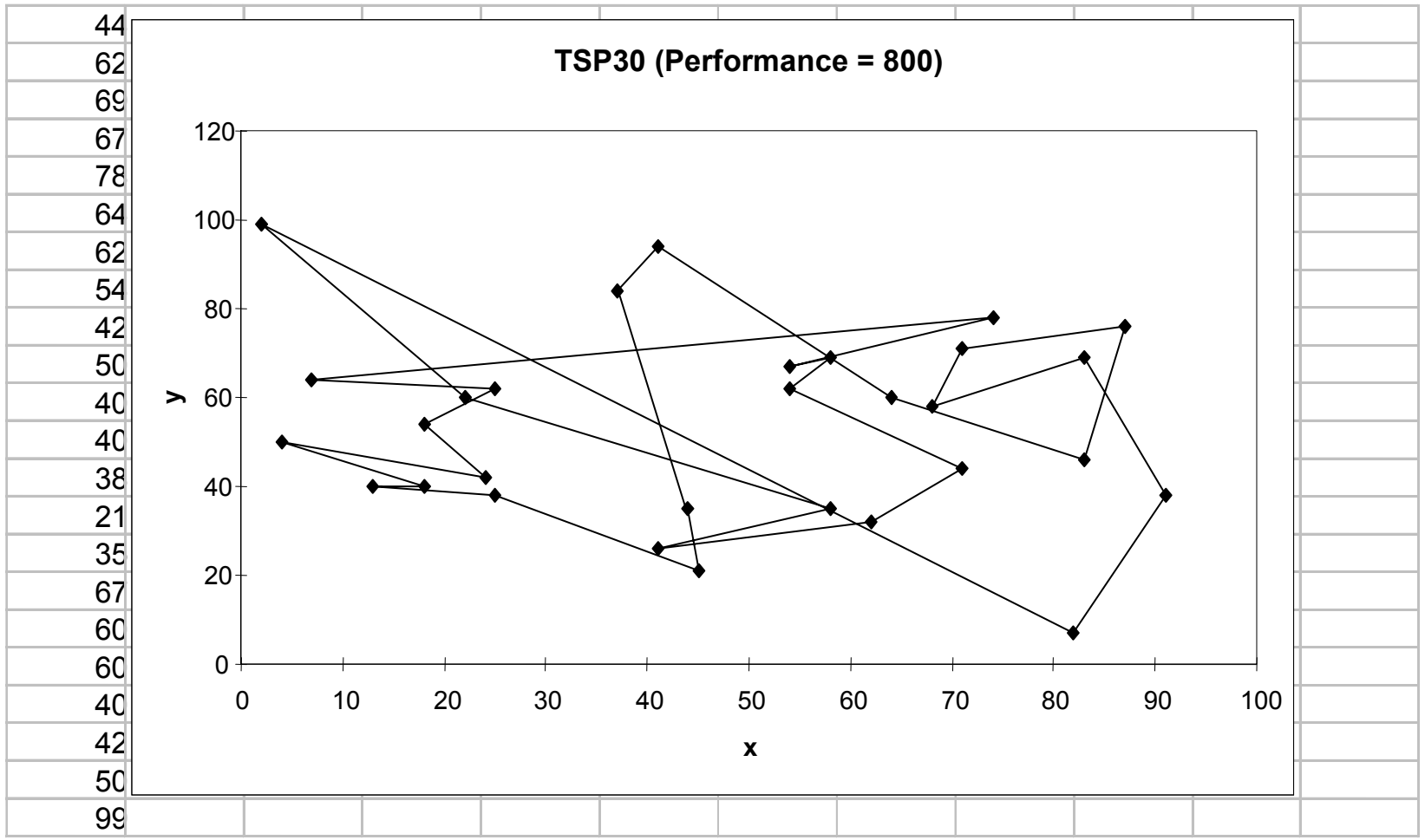
Example of application on TSP

A solution with total distance= 941



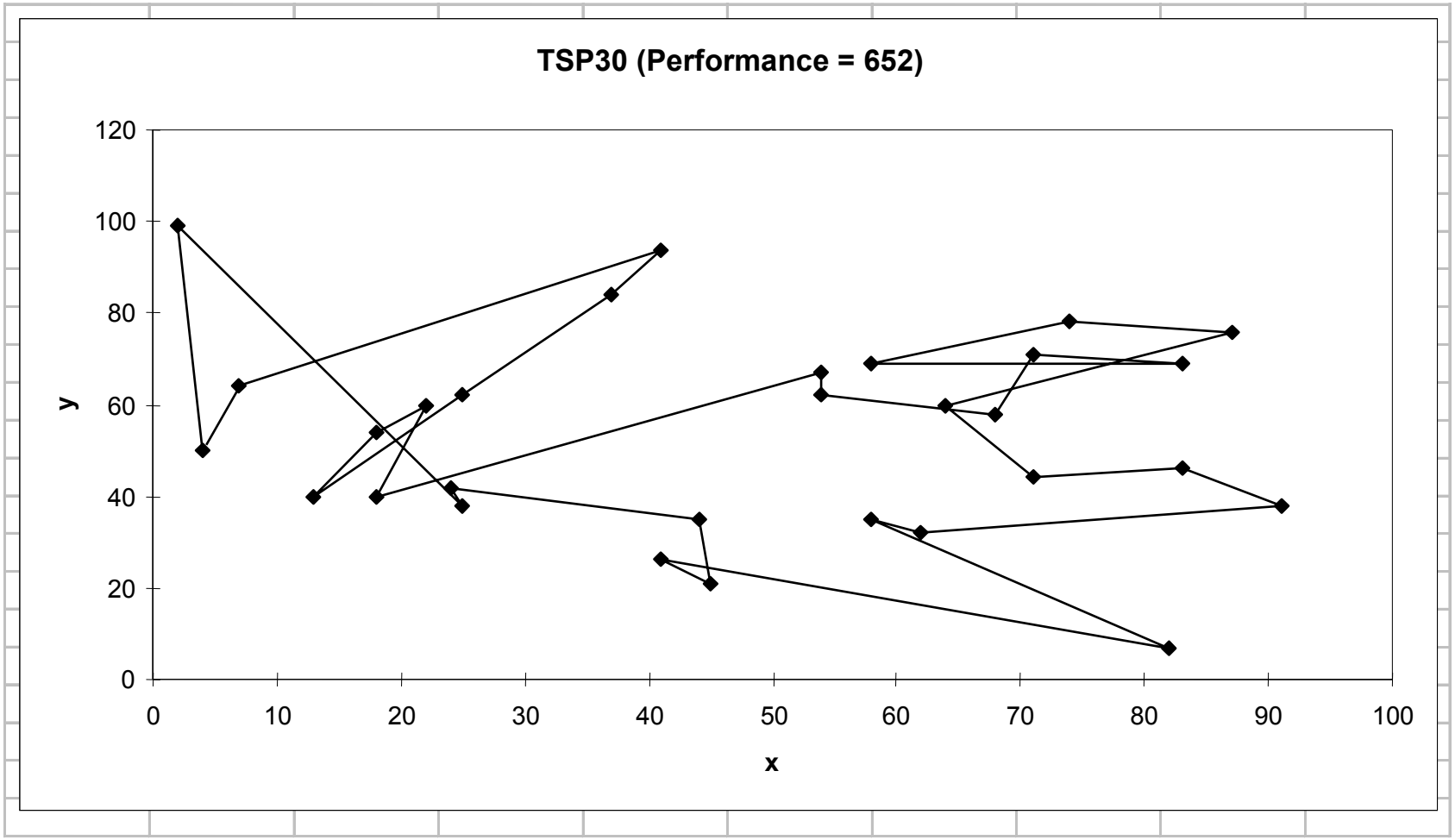
Example of application on TSP

A solution with total distance= 800



Example of application on TSP

A solution with total distance= 652



Example of application on TSP

The best solution with total distance= 420