

# Heuristic Algorithms

Master's Degree in Computer Science/Mathematics

Roberto Cordone

DI - Università degli Studi di Milano



Schedule: **Thursday 14.30 - 16.30 in classroom 503**

**Friday 14.30 - 16.30 in classroom 503**

Office hours: **on appointment**

E-mail: **[roberto.cordone@unimi.it](mailto:roberto.cordone@unimi.it)**

Web page: **<https://homes.di.unimi.it/cordone/courses/2024-ae/2024-ae.html>**

Ariel site: **<https://myariel.unimi.it/course/view.php?id=4466>**

# Extensions of the basic constructive scheme

The basic scheme of constructive algorithms can be enhanced using

- ① a **more effective construction graph**
  - add more than one element to the current subset  $x$
  - add elements to  $x$ , but also remove elements from  $x$
- ② a **more sophisticated selection criterium**, such as
  - a **regret-based function** that estimates potential future losses associated with element  $i$
  - a **look-ahead function** that estimates the final value of the objective obtained adding  $i$  to  $x$

# Extensions of the construction graph

The constructive algorithm adds one element at a time to the solution

It is possible to generalize this scheme with algorithms that **at each step**

- 1 **add more than one element**: the selection criterion  $\varphi_A(B^+, x)$  identifies a subset  $B^+ \subseteq B \setminus x$  to add, instead of a single element  $i$
- 2 **add elements, but also remove a smaller number of elements**: the selection criterion  $\varphi_A(B^+, B^-, x)$  identifies a subset  $B^+ \subseteq B \setminus x$  to add and a subset  $B^- \subseteq x$  to remove, with  $|B^+| > |B^-|$

These algorithms **build an acyclic construction graph on the search space**, so that they never revisit any subset

The fundamental problem is to **define a family  $\Delta_A^+(x)$  of subset pairs such that optimising the selection criterion is a polynomial problem**

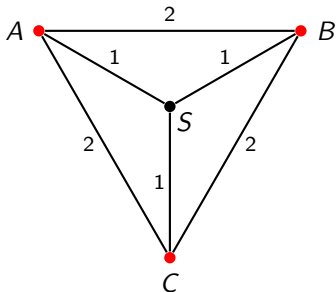
$$\min_{(B^+, B^-) \in \Delta_A^+(x)} \varphi_A(B^+, B^-, x)$$

that is

- subsets **efficiently optimisable** (minimum paths, ...)
- subsets of **limited size** (e. g.,  $|B^+| = 2$  and  $|B^-| = 1$ )

# The Steiner Tree Problem (*STP*)

Given an undirected graph  $G = (V, E)$ , a cost function  $c : E \rightarrow \mathbb{N}$  on the edges and a subset of **special vertices**  $U \subset V$ , find a tree connecting at minimum cost all special vertices



The minimum tree spanning the special vertices is not necessarily optimal  
(*and it might not even exist*)

# The *Distance Heuristic* (*DH*) for the *STP*

A basic constructive algorithm could adopt the same search spaces as

- Kruskal's algorithm: the set of all forests
- Prim's algorithm: the set of all trees including a (special) vertex

but adding one edge at a time

- returns solutions with redundant edges, therefore expensive
- has a hard time distinguishing useful and redundant edges

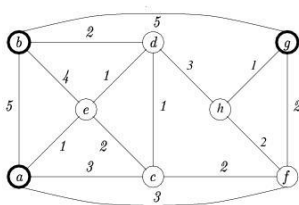
The *Distance Heuristic* adopts as **search space  $\mathcal{F}$**   
the **collection of all trees including a given special vertex  $v_1$**  (as in *Prim*)

**It iteratively adds a path  $B^+$  between  $x$  and a special vertex**  
instead of a single edge, so that

- $x$  remains a tree
- $x$  spans a new special vertex
- the minimum cost path can be computed efficiently at each step

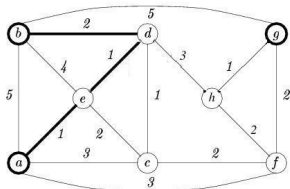
**It terminates when all special vertices are connected**

# Example



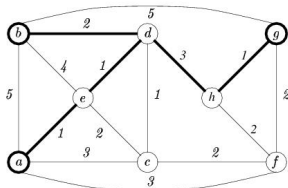
- start with a single special vertex  $a$ :  $x := \emptyset$  (degenerate tree)
- add the closest special vertex ( $b$ ) through path  $(a, e, d, b)$ :  
 $x = \{(a, e), (e, d), (d, b)\}$
- add the closest special vertex ( $g$ ) through path  $(g, h, d)$ :  
 $x = \{(a, e), (e, d), (d, b), (g, h), (h, d)\}$
- all special vertices are in the solution: terminate

# Example



- start with a single special vertex  $a$ :  $x := \emptyset$  (degenerate tree)
- add the closest special vertex ( $b$ ) through path  $(a, e, d, b)$ :  
 $x = \{(a, e), (e, d), (d, b)\}$
- add the closest special vertex ( $g$ ) through path  $(g, h, d)$ :  
 $x = \{(a, e), (e, d), (d, b), (g, h), (h, d)\}$
- all special vertices are in the solution: terminate

# Example



- start with a single special vertex  $a$ :  $x := \emptyset$  (degenerate tree)
- add the closest special vertex ( $b$ ) through path  $(a, e, d, b)$ :  
 $x = \{(a, e), (e, d), (d, b)\}$
- **add the closest special vertex ( $g$ ) through path  $(g, h, d)$ :**  
 $x = \{(a, e), (e, d), (d, b), (g, h), (h, d)\}$
- all special vertices are in the solution: terminate  
*(this time, the solution is optimal)*

The *Distance Heuristic* algorithm is **2-approximated**

It is equivalent to computing a minimum spanning tree on a graph with

- vertices reduced to the special vertices
- edges corresponding to the minimum paths



# Counterexample to optimality

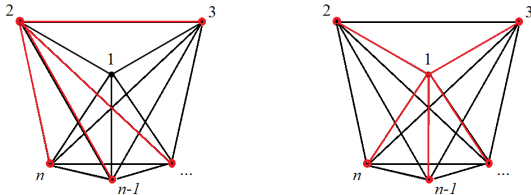
Consider a complete graph  $G = (V, E)$  with  $U = V \setminus \{1\}$  and cost

$$c_{uv} = \begin{cases} (1 + \epsilon)M & \text{for } u \text{ or } v = 1 \\ 2M & \text{for } u, v \in U \end{cases}$$

( $M$  is just used to obtain integer costs for any  $\epsilon$ )

The  $DH$  returns a star spanning the special vertices:  $f_{DH} = (n - 2) \cdot 2M$

The optimal solution is a spanning star centred in 1:  $f^* = (n - 1) \cdot (1 + \epsilon)M$



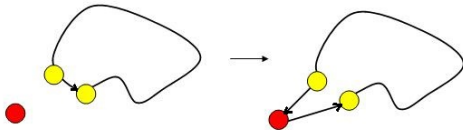
The approximation ratio is  $\rho_{DH} = \frac{f_{DH}}{f^*} = \frac{n-2}{n-1} \cdot \frac{2}{1+\epsilon} < 2$

and converges to 2 as  $n$  increases and  $\epsilon$  decreases

# Insertion algorithms for the TSP

Several heuristic algorithms for the TSP define the search space  $\mathcal{F}_A$  as the set of all circuits of the graph including a given node;

- cannot be obtained from another one by adding a single arc
- can be obtained adding two arcs  $(i, k)$ ,  $(k, j)$  and removing one  $(i, j)$



- 1 Start with a zero-cost self-loop on node 1:  $x^{(0)} = \{(1, 1)\}$   
*It is not very different from an empty set*
- 2 Select a node  $k$  to be added and an arc  $(i, j)$  to be removed
- 3 If the circuit does not visit all nodes, go back to point 2; otherwise terminate

Such a scheme never visits again the same solution and builds a feasible solution in  $n - 1$  steps (each step adds a new node)

# Insertion algorithms for the TSP

The selection criterium  $\varphi_A(B^+, B^-, x)$  must choose an arc and a node; there are  $(n - |x|) |x| \in O(n^2)$  alternatives

- $|x|$  possible arcs  $(s_i, s_{i+1})$  to remove
- $n - |x|$  possible nodes  $k$  to add through the arcs  $(s_i, k)$  and  $(k, s_{i+1})$

The *Cheapest Insertion* (CI) heuristic uses as a selection criterium

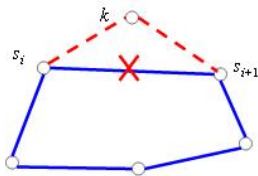
$$\varphi_A(B^+, B^-, x) = f(x \cup B^+ \setminus B^-)$$

Objective function  $f(x)$  is additive, hence extensible to the whole of  $\mathcal{F}_A$

Since  $f(x \cup B^+ \setminus B^-) = f(x) + c_{s_i, k} + c_{k, s_{i+1}} - c_{s_i, s_{i+1}}$

$$\arg \min_{(B^+, B^-)} \varphi_A(B^+, B^-, x) = \arg \min_{i, k} (c_{s_i, k} + c_{k, s_{i+1}} - c_{s_i, s_{i+1}})$$

The computational cost of evaluating  $\varphi_A$  decreases from  $\Theta(n)$  to  $\Theta(1)$



# Cheapest Insertion heuristic for the TSP

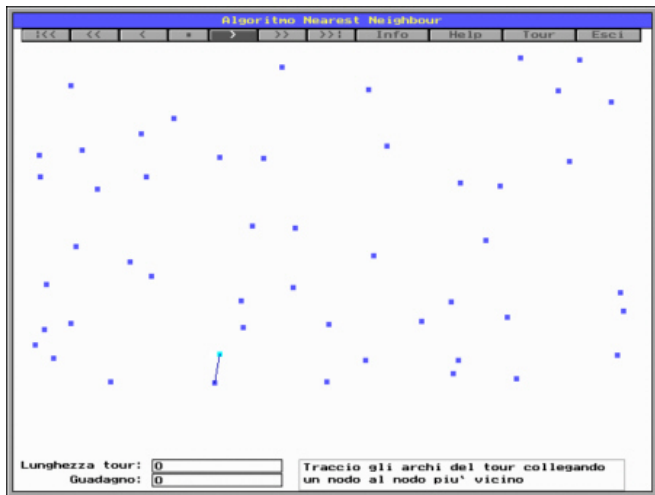
## Algorithm *Cheapest Insertion*

- 1 start with a zero-cost self-loop on node 1:  $x^{(0)} = \{(1, 1)\}$   
*It is also like starting with a single node*
- 2 select the arc  $(s_i, s_{i+1}) \in x$  and the node  $k \notin N_x$  such that  $(c_{s_i, k} + c_{k, s_{i+1}} - c_{s_i, s_{i+1}})$  is minimum
- 3 if the circuit does not visit all nodes, go back to point 2; otherwise terminate

It is not exact, but **2-approximated**, under the triangle inequality

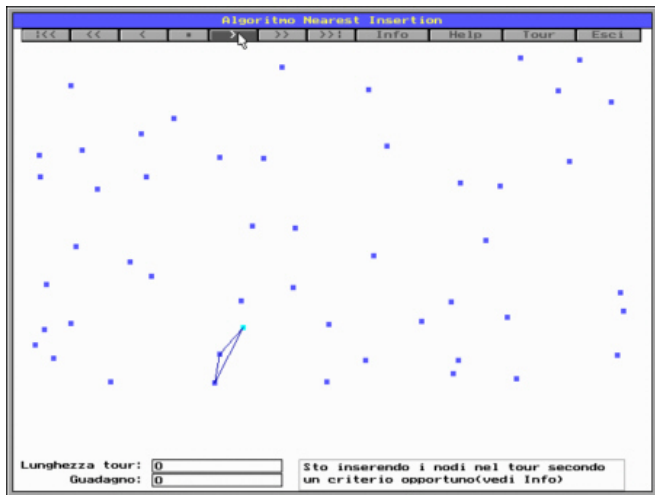
# An example

Start with a single node (as in the *NN* heuristic)



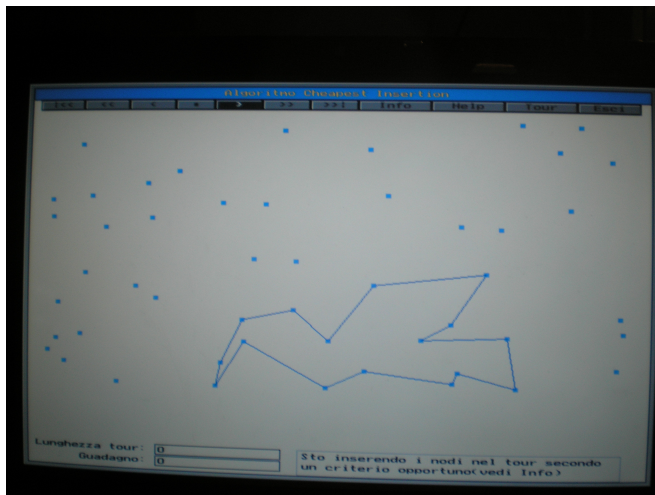
# An example

Create a circuit (instead of a path)



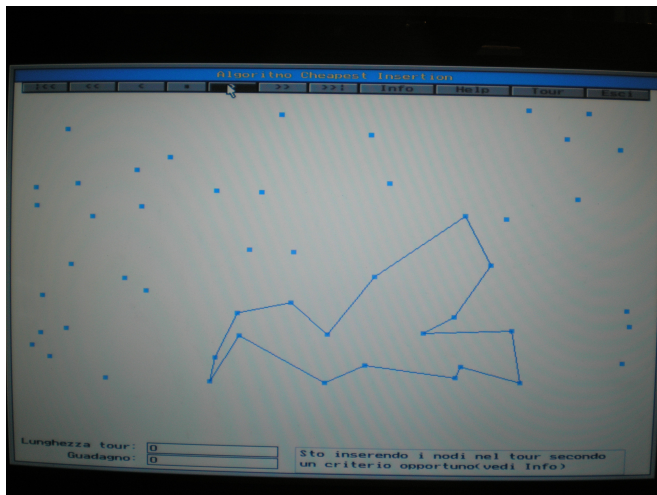
# An example

Add at each step the node that minimally increases the circuit cost



# An example

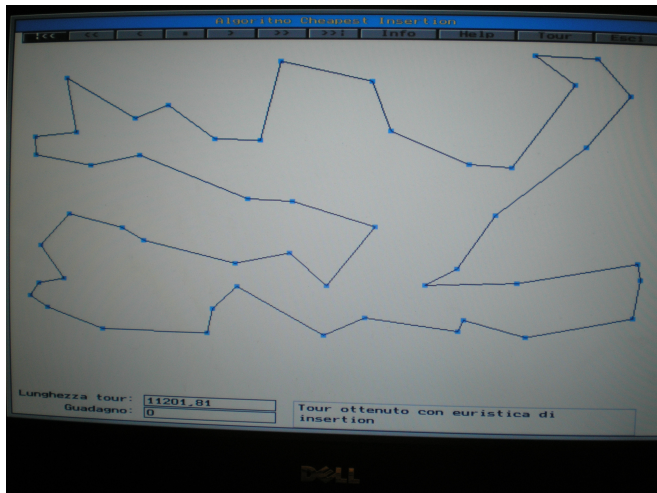
Add at each step the node that minimally increases the circuit cost





# An example

Terminate when the circuit visits all nodes



# Cheapest Insertion heuristic for the TSP

The *CI* algorithm performs  $n - 1$  steps: at each step  $t$

- it evaluates  $(n - t) t$  node-arc pairs
  - each evaluation requires constant time
  - each evaluation possibly updates the best move
- it performs the best addition/removal
- it decides whether to terminate

The overall complexity is  $\Theta(n^3)$

It can be reduced to  $\Theta(n^2 \log n)$  collecting in a *min-heap* the insertion costs for each external node: each of the  $n$  steps

- selects the best insertion in  $O(n)$  time and performs it
- creates two new insertions and removes one for each external node, and updates their heaps in  $O(n \log n)$  time

# Nearest Insertion heuristic for the TSP

Algorithm *Cheapest Insertion* tends to select nodes close to circuit  $x$ :  
minimizing  $c_{s_i,k} + c_{k,s_{i+1}} - c_{s_i,s_{i+1}}$  implies that  $c_{s_i,k}$  and  $c_{s_{i+1},k}$  are small

To accelerate, one can **decompose criterium  $\varphi_A$  into two phases**

Algorithm *Nearest Insertion* (NI)

- 1 start with a zero-cost self-loop on node 1:  $x^{(0)} = \{(1, 1)\}$
- 2 **Add criterium**: select the **node  $k$  nearest to circuit  $x$**

$$k = \arg \min_{\ell \notin N_x} \left( \min_{s_i \in N_x} c_{s_i, \ell} \right)$$

- 3 **Delete criterium**: select the **arc  $(s_i, s_{i+1})$  that minimises  $f$**

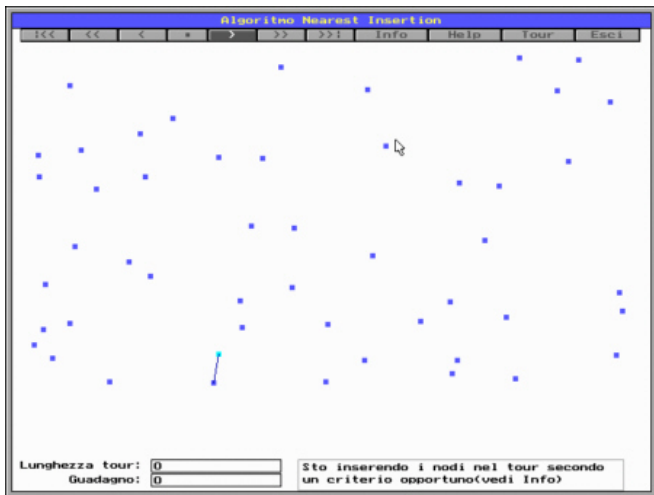
$$(s_i, s_{i+1}) = \arg \min_{(s_i, s_{i+1}) \in X} (c_{s_i, k} + c_{k, s_{i+1}} - c_{s_i, s_{i+1}})$$

- 4 If the circuit does not visit all nodes, go back to point 2; otherwise terminate

It is not exact, but **2-approximated**, under the triangle inequality

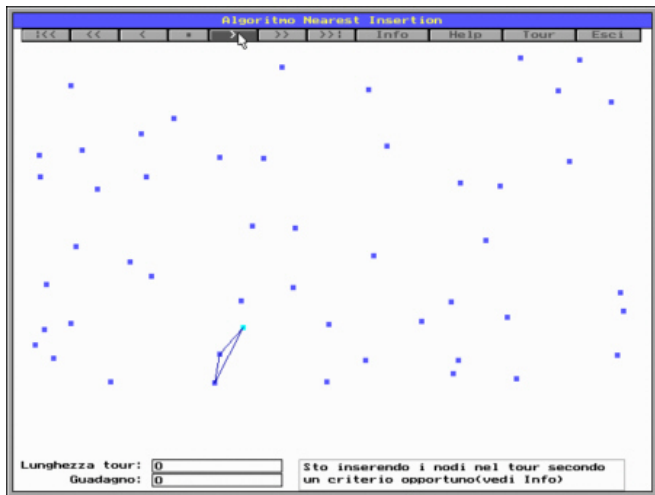
# An example

Start with a single vertex (as in *NN* and *CI*)



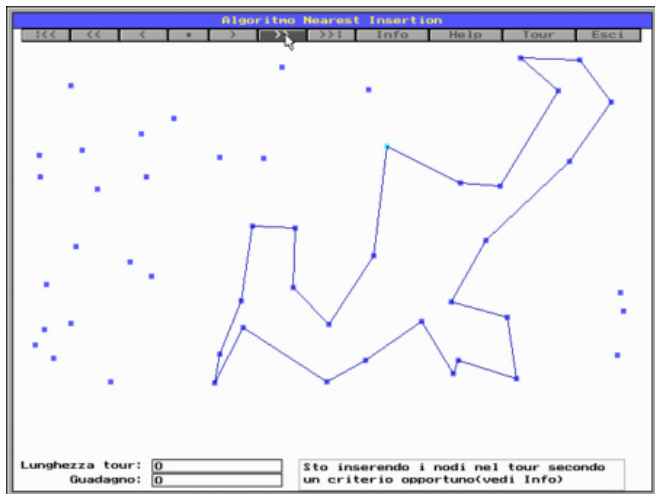
# An example

Create a circuit (as in  $C1$ )



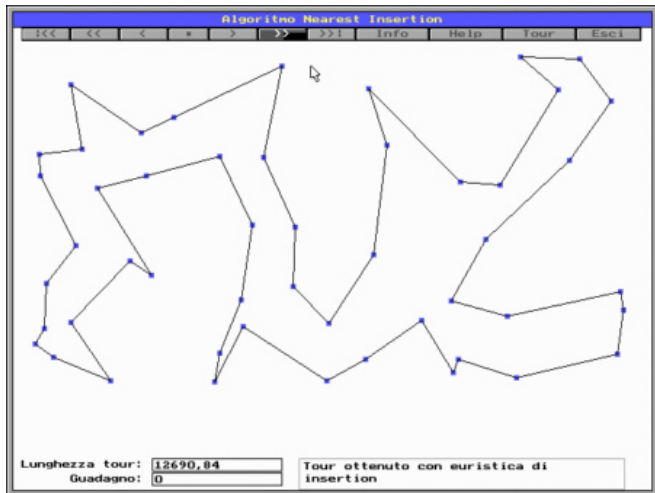
# An example

The circuit grows differently, always adding the closest node, even if this increases the cost more than another node



# An example

Terminate when the circuit visits all nodes



# Nearest Insertion heuristic for the TSP

The *NI* algorithm performs  $n - 1$  steps: at each step  $t$

- it evaluates the distance of  $(n - t)$  nodes from the circuit, each one in  $\Theta(t)$  time
- it selects the node at minimum distance
- it evaluates the removal of  $t$  arcs, each one in  $\Theta(1)$  time
- it performs the best addition/removal
- it decides whether to terminate

The overall complexity is  $\Theta(n^3)$

It can be reduced to  $\Theta(n^2)$  collecting in a vector for each external node the closest internal node: each of the  $n - 1$  steps

- selects the closest node in  $O(n)$  time
- finds the insertion point in  $O(n)$  time
- inserts the node creating a new internal node for each external node, which possibly becomes the closest saved in the vector; each of the  $O(n)$  updates takes  $O(1)$  time



# Farthest Insertion heuristic for the TSP

The choice of the closest node to the cycle is natural, but misleading: since all nodes must be visited, it is preferable to service in the best way the most problematic ones (i. e., the farthest ones)

Algorithm *Farthest Insertion* (*FI*)

- 1 start with a zero-cost self-loop on node 1:  $x^{(0)} = \{(1, 1)\}$
- 2 **Add criterium**: select the node  $k$  farthest from cycle  $x$

$$k = \arg \max_{\ell \notin N_x} \left( \min_{s_i \in N_x} c_{s_i, \ell} \right)$$

(the node that is farthest from the closest node of the cycle)

- 3 **Delete criterium**: select the arc  $(s_i, s_{i+1})$  minimizing

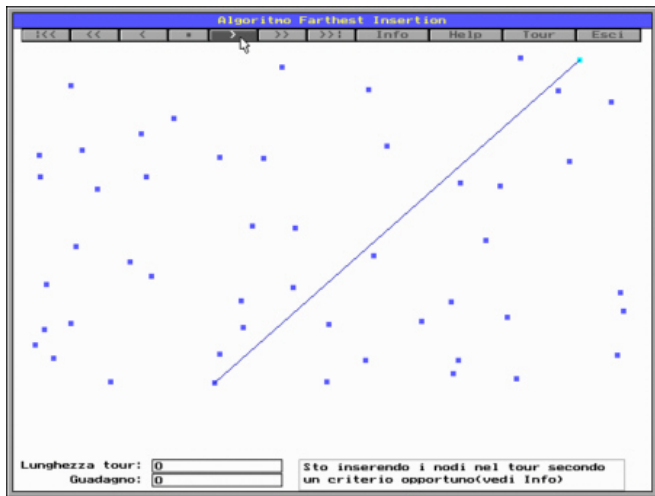
$$(s_i, s_{i+1}) = \arg \min_{(s_i, s_{i+1}) \in X} (c_{s_i, k} + c_{k, s_{i+1}} - c_{s_i, s_{i+1}})$$

- 4 If the circuit does not visit all nodes, go back to point 2; otherwise terminate

It is **log  $n$ -approximated** under the triangle inequality, hence worse than the previous ones in the worst-case (but often experimentally better)

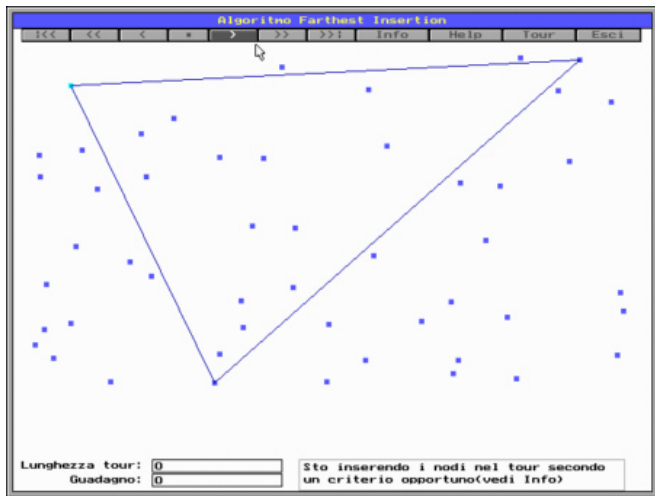
# An example

Start reaching immediately the farthest node



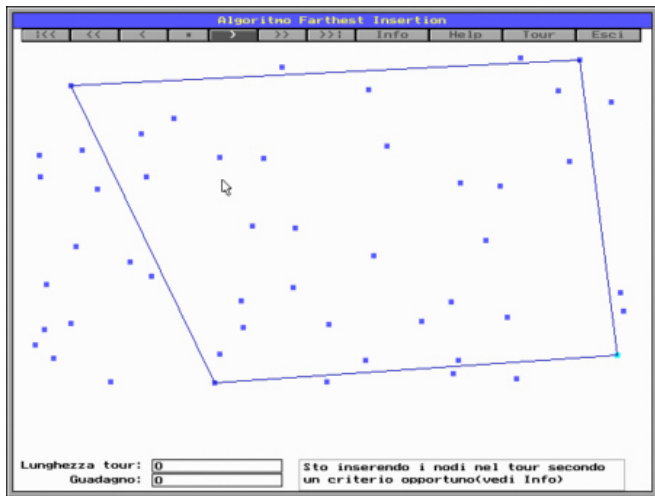
# An example

And go on like that



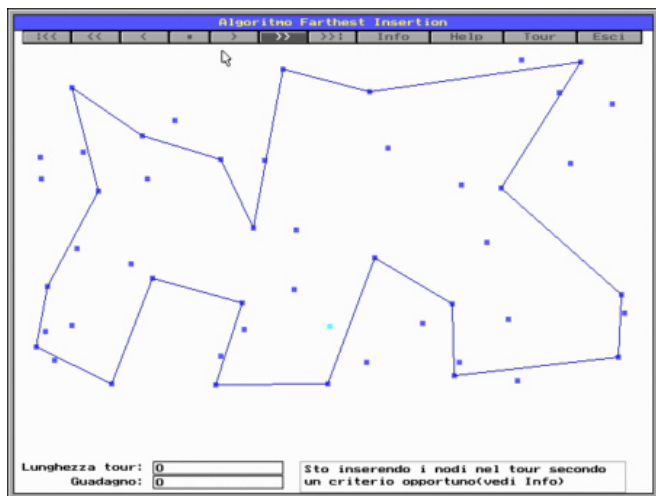
# An example

But always inserting these nodes in the best possible way



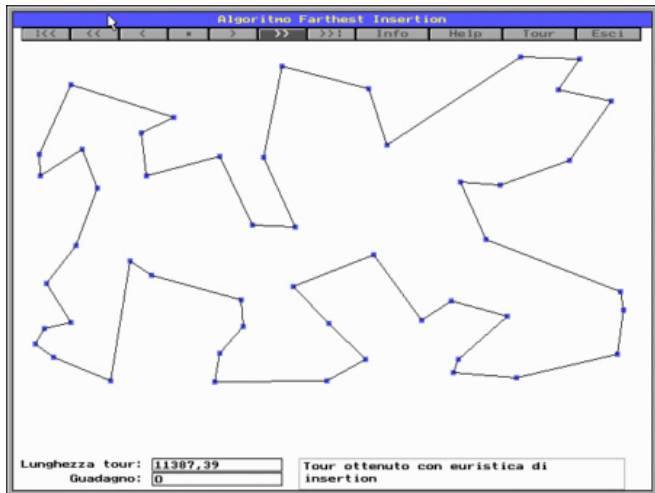
# An example

The circuit grows more regularly, with much less crossings and twists



# An example

Terminate when the circuit visits all nodes



# Farthest Insertion heuristic for the TSP

The *FI* algorithm performs  $n - 1$  steps: at each step  $t$

- it evaluates the distance of  $(n - t)$  nodes from the circuit, each one in  $\Theta(t)$  time
- select the node at maximum distance
- it evaluates the removal of  $t$  arcs, each one in  $\Theta(1)$  time
- it performs the best addition/removal
- it decides whether to terminate

The overall complexity is  $\Theta(n^3)$

It can be reduced to  $\Theta(n^2)$  as in the *NI* heuristic

# Extensions of constructive algorithms

The basic scheme of constructive algorithms can be enhanced using

- ① a **more effective construction graph**
  - add more than one element to the current subset  $x$
  - add elements to  $x$ , but also remove elements from  $x$
- ② a **more sophisticated selection criterium**, such as
  - a **regret-based function** that estimates potential future losses associated with element  $i$
  - a **look-ahead function** that estimates the final value of the objective obtained adding  $i$  to  $x$



Decisions taken in early steps can severely restrict the feasible choices in later steps due to the **constraints** of the problem

- *BPP*: all objects must be put into a container, but early assignments could make some containers unavailable for later objects
- *TSP*: all nodes must be visited, but early routing decisions could make the visit of later nodes more expensive  
(*even impossible, if the graph is noncomplete*)
- *CMST*: all vertices must be linked to the root through a subtree, but early links could make some subtrees unavailable for later vertices

The selection criterium can take it into account implicitly

- *BPP*: the Decreasing First-Fit heuristic assigns the larger objects first
- *TSP*: the Farthest Insertion heuristic visits the farther nodes first

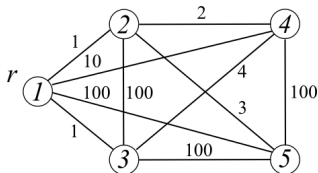
**Some selection criteria aim explicitly to leave larger sets of good choices**

A typical **regret-based heuristic** consists in

- partitioning  $\Delta_A^+(x)$  into disjoint classes of choices  
(*the assignments of each object, the edges incident in each vertex*)
- compute a basic selection criterium for all choices
- compute for each class the **regret**, i. e. the **difference between**
  - the **second-best choice**
  - the **average of the other choices** (possibly weighted)**and the best choice** in order to estimate the **damage incurred by postponing the best choice until it becomes impossible**
- choose **the best choice of the class for which the regret is maximum**  
*This is effective when a single choice per class must be taken*

# Example

Consider the *CMSTP* and ground set  $B = V \times T$  (*(vertex, subtree) pairs*)  
Let the weights be uniform ( $w_v = 1$  for all  $v \in V$ ) and capacity  $W = 2$



Let the search space  $\mathcal{F}$  include all partial solutions

The greedy algorithm puts vertex 2 in subtree 1, vertex 3 in subtree 2;  
then vertex 4 in subtree 1 and finally vertex 5 in subtree 3:

$$c(x) = 1 + 1 + 2 + 100 = 104$$

The regret algorithm puts vertex 2 in subtree 1; now:

- the regret of vertex 3 is the difference  $c(3, 3) - c(3, 2) = 1 - 1 = 0$
- the regret of vertex 4 is the difference  $c(4, 2) - c(4, 1) = 10 - 2 = 8$
- the regret of vertex 5 is the difference  $c(5, 2) - c(5, 1) = 100 - 3 = 97$

The algorithm puts vertex 5 in subtree 1

Then, it proceeds putting vertices 2 and 4 in subtree 2:

$$c(x) = 1 + 3 + 1 + 4 = 9$$

# Roll-out heuristics

They are also known as **single-step look-ahead** constructive heuristics and were proposed by Bertsekas and Tsitsiklis (1997)

Given a **basic constructive heuristic**  $A$

- start with an empty subset:  $x^{(0)} = \emptyset$
- at each step  $t$ 
  - **extend the subset in each feasible way**:  $x^{(t-1)} \cup \{i\}, \forall i \in \Delta_A^+(x)$
  - **apply the basic heuristic to each extended subset** and compute the resulting solution  $x_A(x^{(t-1)} \cup \{i\})$
  - **use the value of the solution as the selection criterium** to choose  $i^{(t)}$

$$\varphi_A(i, x) = f(x_A(x^{(t-1)} \cup \{i\}))$$

- terminate when  $\Delta_A^+(x)$  is empty

*Try every feasible move, look at the result, go back and choose the move*

**The result of the roll-out heuristic dominates that of the basic heuristic**  
(*under very general conditions*)

**The complexity remains polynomial, but is much larger:**  
in the worst case,  $T_{\text{ro}(A)} = |B|^2 T_A$

# Example: roll-out for the SCP

$$c \quad \begin{array}{|c|c|c|c|c|} \hline 25 & 6 & 8 & 24 & 12 \\ \hline \end{array}$$

$$A \quad \begin{array}{|c|c|c|c|c|} \hline 1 & 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 \\ 1 & 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 1 \\ \hline \end{array}$$

- 1 start with the empty subset:  $x^{(0)} = \emptyset$
- 2 for each column  $i$ , apply the constructive heuristic starting from subset  $x^{(0)} \cup \{i\} = \{i\}$ 
  - for  $i = 1$ , obtain  $x_A(\{1\}) = \{1\}$  of cost  $f_A(\{1\}) = 25$
  - for  $i = 2$ , obtain  $x_A(\{2\}) = \{2, 3, 5, 4\}$  of cost  $f_A(\{2\}) = 50$
  - for  $i = 3$ , obtain  $x_A(\{3\}) = \{3, 2, 5, 4\}$  of cost  $f_A(\{3\}) = 50$
  - for  $i = 4$ , obtain  $x_A(\{4\}) = \{4, 2, 5\}$  of cost  $f_A(\{4\}) = 43$
  - for  $i = 5$ , obtain  $x_A(\{5\}) = \{5, 2, 3, 4\}$  of cost  $f_A(\{5\}) = 50$
- 3 the best solution is the first one, therefore  $i^{(1)} = 1$
- 4 all rows are covered: the algorithm terminates

# Generalised roll-out heuristics

The scheme can be generalised

- applying several basic heuristics  $A^{[1]}, \dots, A^{[\ell]}$
- increasing the number of look-ahead steps, i. e., using  $x^{(t-1)} \cup B^+$  with  $|B^+| > 1$

The result improves and the complexity worsens further

The overall scheme does not change significantly

- start from the empty subset:  $x^{(0)} = \emptyset$
- at each step  $t$ 
  - for each possible extension  $B^+ \in \Delta_A^+(x^{(t-1)})$   
apply each basic algorithm  $A^{[l]}$  starting from  $x^{(t-1)} \cup B^+$
  - the selection criterium is  $\min_l f_{A^{[l]}}(x^{(t-1)} \cup B^+)$
  - use the value of the best solution as the selection criterium for  $i^{(t)}$

$$\varphi_A(i, x) = \min_{l=1, \dots, \ell} f(x_A(x^{(t-1)} \cup \{i\}))$$

- when  $\Delta_A^+(x)$  is empty, terminate

# Destructive heuristics

It is an approach exactly complementary to the constructive one

- start with the full ground set:  $x^{(0)} := B$
- remove an element at a time, selected
  - so as to remain within the search space  $\mathcal{F}_A$

$$\Delta_A^+(x) = \{i \in x : x \setminus \{i\} \in \mathcal{F}_A\}$$

- maximizing a selection criterion  $\varphi_A(i, x)$  (usually a cost reduction)
- terminate when  $\Delta_A^+(x) = \emptyset$  (there is no way to remain in  $\mathcal{F}_A$ )

A destructive heuristic (for a minimization problem) can be described as

*Algorithm Stingy(I)*

$x := B; x^* := B;$

*If*  $x \in X$  *then*  $f^* := f(x)$  *else*  $f^* := +\infty;$

*While*  $\Delta_A^+(x) \neq \emptyset$  *do*

$i := \arg \max_{i \in \Delta_A^+(x)} \varphi_A(i, x);$

$x := x \setminus \{i\};$

*If*  $x \in X$  *and*  $f(x) < f^*$  *then*  $x^* := x; f^* := f(x);$

*Return*  $(x^*, f^*);$

*It is optimal for the Minimum Spanning Tree Problem!*

# Why are they less used?

When the solutions are much smaller than the ground set ( $|x| \ll |B|$ )  
a destructive heuristic

- requires a larger number of steps
- is more likely to make a wrong decision at an early step
- sometimes requires more time to evaluate  $\Delta_A^+(x)$  and  $\varphi_A(i, x)$

When a constructive heuristic returns redundant solutions, it is useful to append a destructive heuristic at its end as a post-processing phase

This auxiliary destructive heuristic

- starts from the solution  $x$  of the constructive heuristic, instead of  $B$
- adopts as a search space the feasible region:

$$\mathcal{F}_A = X \Rightarrow \Delta_A^+(x) = \{i \in x : x \setminus \{i\} \in X\}$$

- adopts as the selection criterium the objective function:

$$\varphi_A(i, x) = f(x \setminus \{i\})$$

- terminates after very few steps



# Constructive/destructive heuristic for the SCP

$$c \quad \begin{array}{|c|c|c|c|} \hline 6 & 8 & 24 & 12 \\ \hline \end{array}$$

$$A \quad \begin{array}{|c|c|c|c|} \hline 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ \hline \end{array}$$

- 1 The constructive heuristic selects, in order, columns 1, 2, 4 and 3  
(each one covers new rows)
- 2 The solution is redundant: column 2 can be removed  
(the following columns also cover already covered rows)
- 3 The auxiliary destructive heuristic removes column 2 and provides the optimal solution  $x^* = \{1, 3, 4\}$   
(columns 1, 3 and 4 are essential to cover rows 1, 2, 5 and 6)

# Summary about constructive and destructive algorithms

## Constructive and destructive algorithms

- 1 are **intuitive**
- 2 are **simple** to design, analyze and implement
- 3 are **very efficient** (low-order polynomials)

$$T_A(n) \in O(n(T_{\Delta_A^+}(n) + T_{\varphi_A}(n)))$$

where

- $T_{\Delta_A^+}(n)$  is the cost to identify  $\Delta_A^+(x)$
  - $T_{\varphi_A}(n)$  is the cost to evaluate  $\varphi_A(i, x)$  for each  $i \in \Delta_A^+(x)$
  - the selection of  $\arg \min \varphi_A(i, x)$  and update of  $x$  (and auxiliary data structures) are dominated
- 4 have a **strongly variable effectiveness**
    - on some problems they guarantee an optimal solution
    - on other problems they provide an approximation guarantee
    - on most problems they provide solutions of extremely variable quality, often scarce
    - on some problems they cannot even guarantee a feasible solution

It is fundamental to **study the problem before the algorithm**

# When are they used?

Constructive and destructive algorithm are used

- ① when **they provide the optimal solution**
- ② when **the execution time must be very short**  
(e.g., for **on-line problems**: schedulers, on-call services, ...)
- ③ when **the problem has a huge size** or requires heavy computations  
(e.g., some data are obtained by simulation)
- ④ **as component of other algorithms**, for example as
  - **starting phase** for exchange algorithms
  - **basic procedure** for recombination algorithms