Heuristic Algorithms for Combinatorial Optimization problems

Ph.D. course in Computer Science

Roberto Cordone

DI - Università degli Studi di Milano

E-mail: roberto.cordone@unimi.it

Web page: https://homes.di.unimi.it/cordone/courses/2023-haco/2023-haco.html

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 an 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 criterium $\varphi_A(B^+, x)$ 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 criterium $\varphi_A(B^+, B^-, x)$ 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

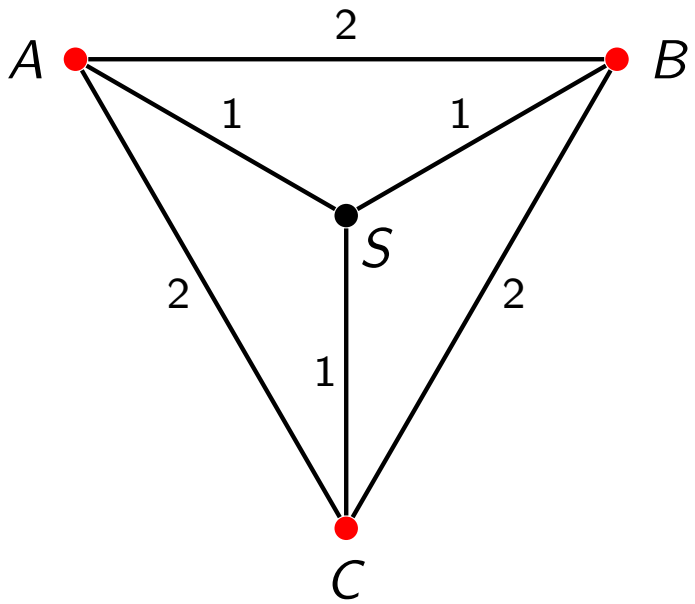
The fundamental problem is to define families of subsets such that optimizing the selection criterium is a polynomial problem

$$\min_{B^+ \subseteq B \setminus \{x\}, B^- \subseteq \{x\}} \varphi_A(B^+, B^-, x)$$

- subsets efficiently optimizable (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 $c : E \to \mathbb{N}$ 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

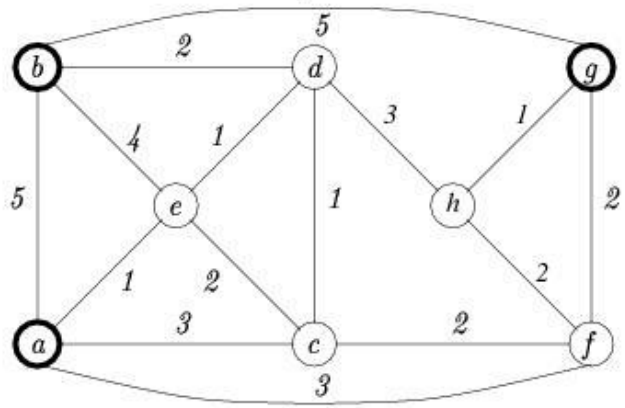
The *Distance Heuristic* adopts as search space $\mathcal{F}$
the collection of all trees including a given special vertex v_1 (*as in Prim*)

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

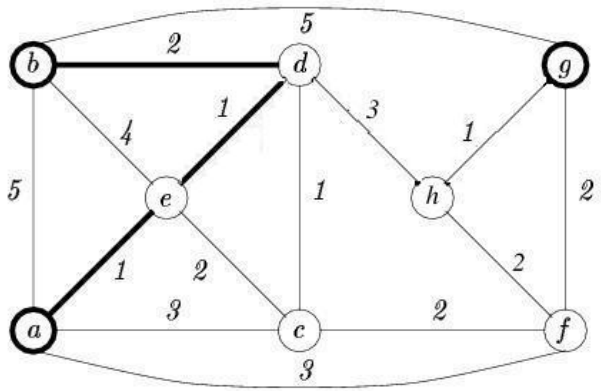
It terminates when all special vertices are connected

Example



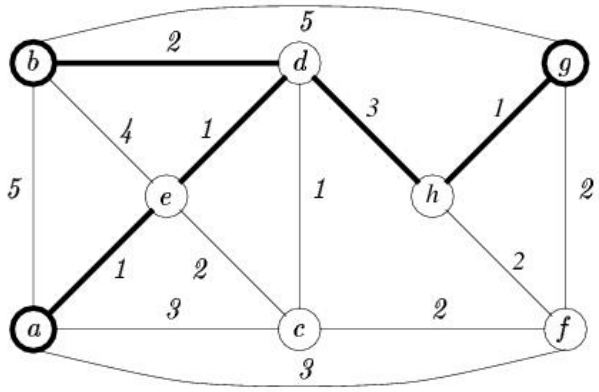
- start with a single special vertex a: $x := \emptyset$ (degenerate tree)
- add the closest special vertex (b) through path (a, e, d, b): $x = \{(a, e), (e, d), (d, b)\}$
- add the closest special vertex (g) through path (g, h, d): $x = \{(a, e), (e, d), (d, b), (g, h), (h, d)\}$
- all special vertices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

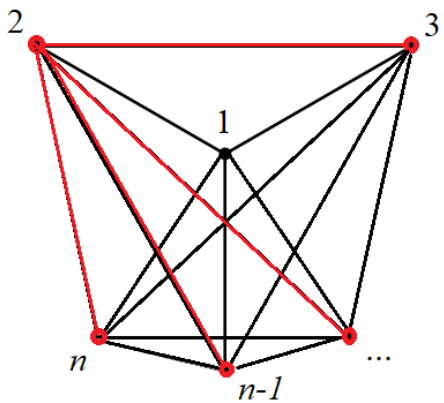
Consider a complete graph $G = (V, E)$ with $U = V \setminus \{1\}$ and cost

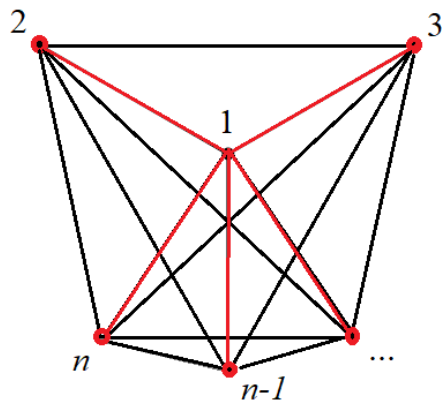
$$c_{uv} = \begin{cases} (1+\epsilon)M & \text{for } u \text{ or } v = 1 \\ 2M & \text{for } u, v \in U \end{cases}$$

(M *is just used to obtain integer costs for any* ϵ)

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

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





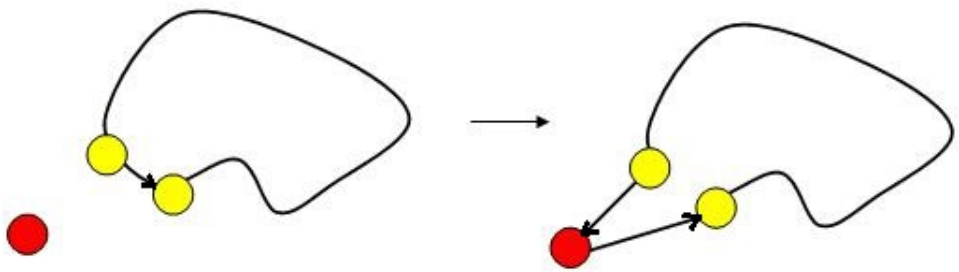
The approximation ratio is $\rho_{DH} = \dfrac{f_{\mathrm{DH}}}{f^*} = \dfrac{n-2}{n-1} \cdot \dfrac{2}{1+\epsilon} < 2$

and converges to 2 as n increases and ϵ 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; a circuit

- 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

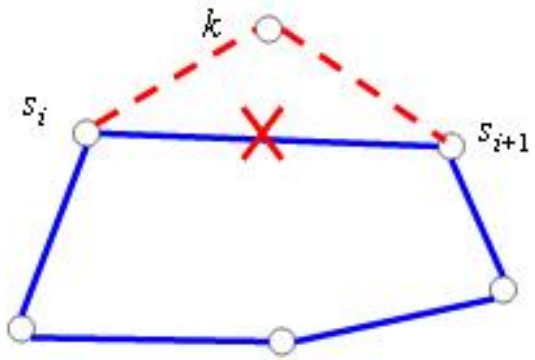
$$\varphi_A(B^+, B^-, x) = f(x \cup B^+ \setminus B^-)$$

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 φ_A decreases from $\Theta(n)$ to $\Theta(1)$



Cheapest Insertion heuristic for the *TSP*

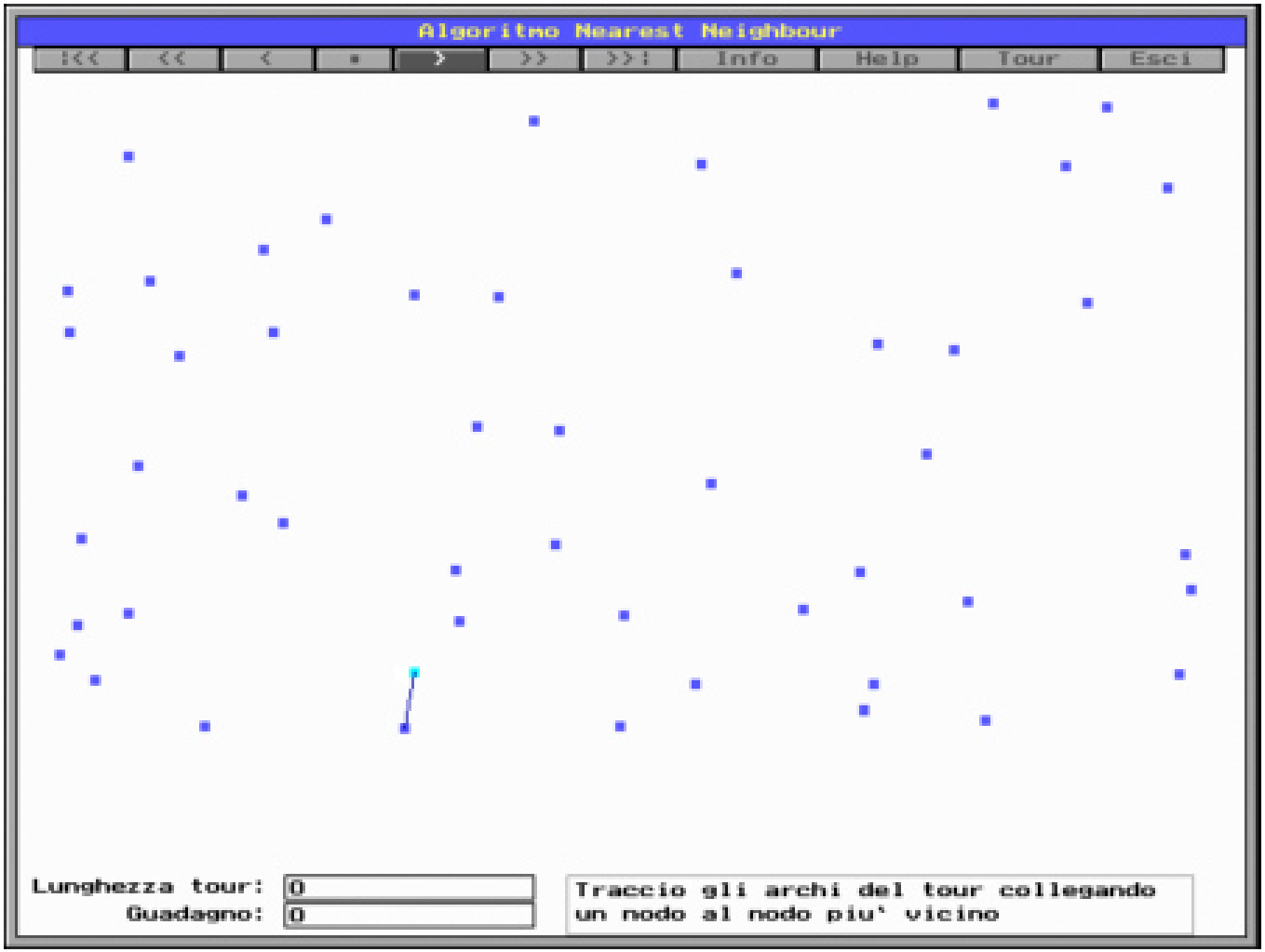
Algorithm *Cheapest Insertion*

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

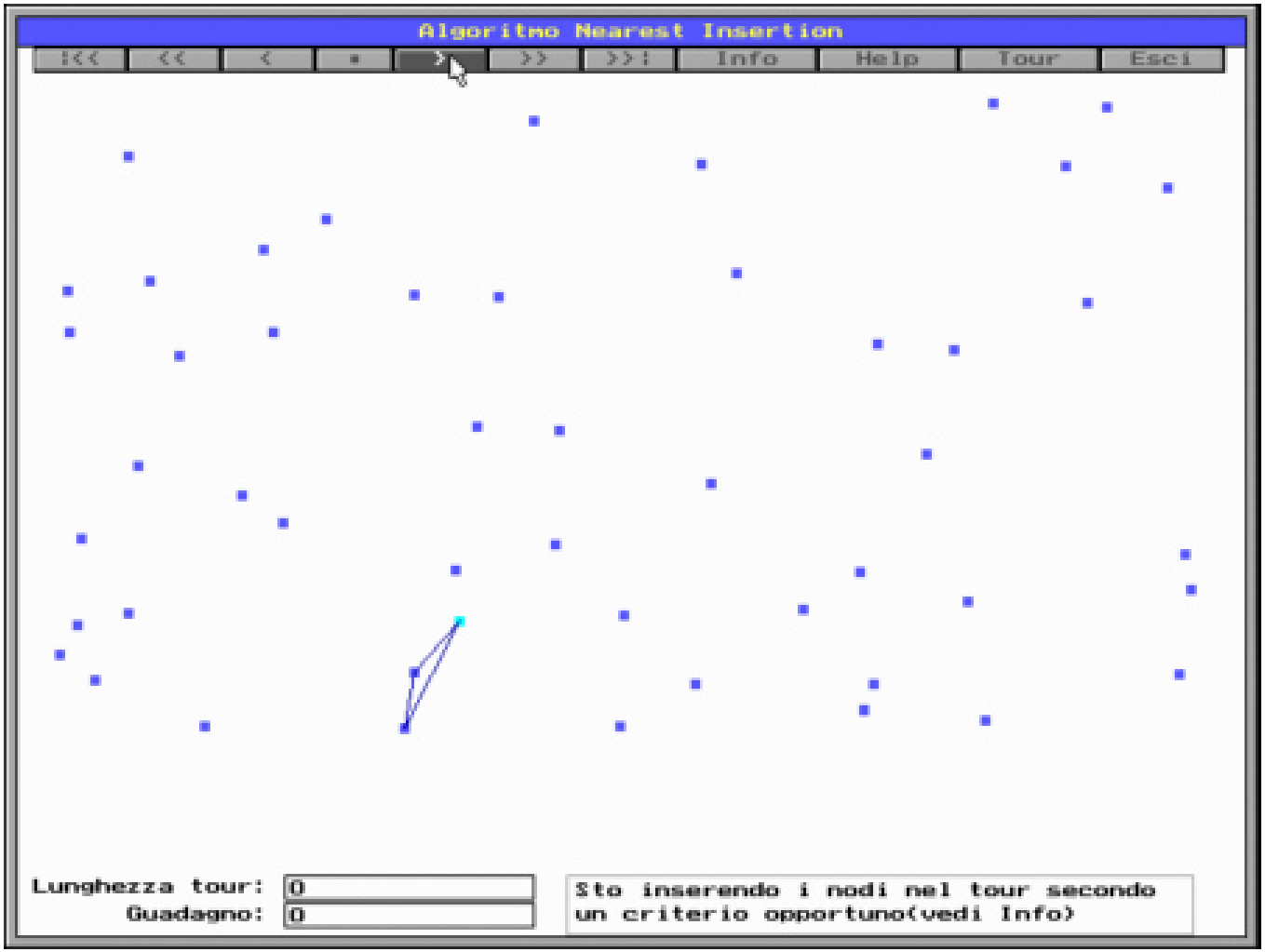
An example

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



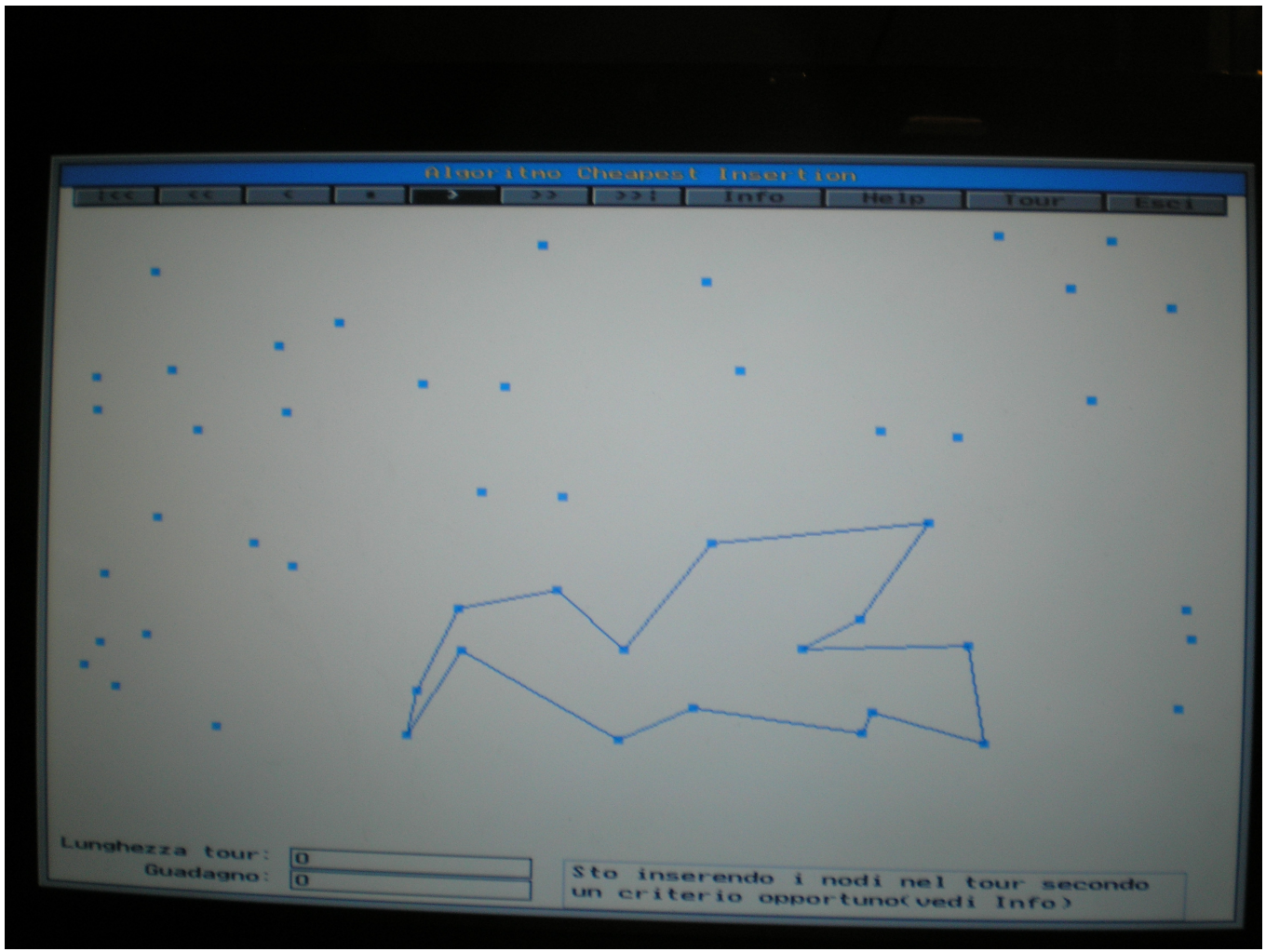
An example

Create a circuit (instead of a path)



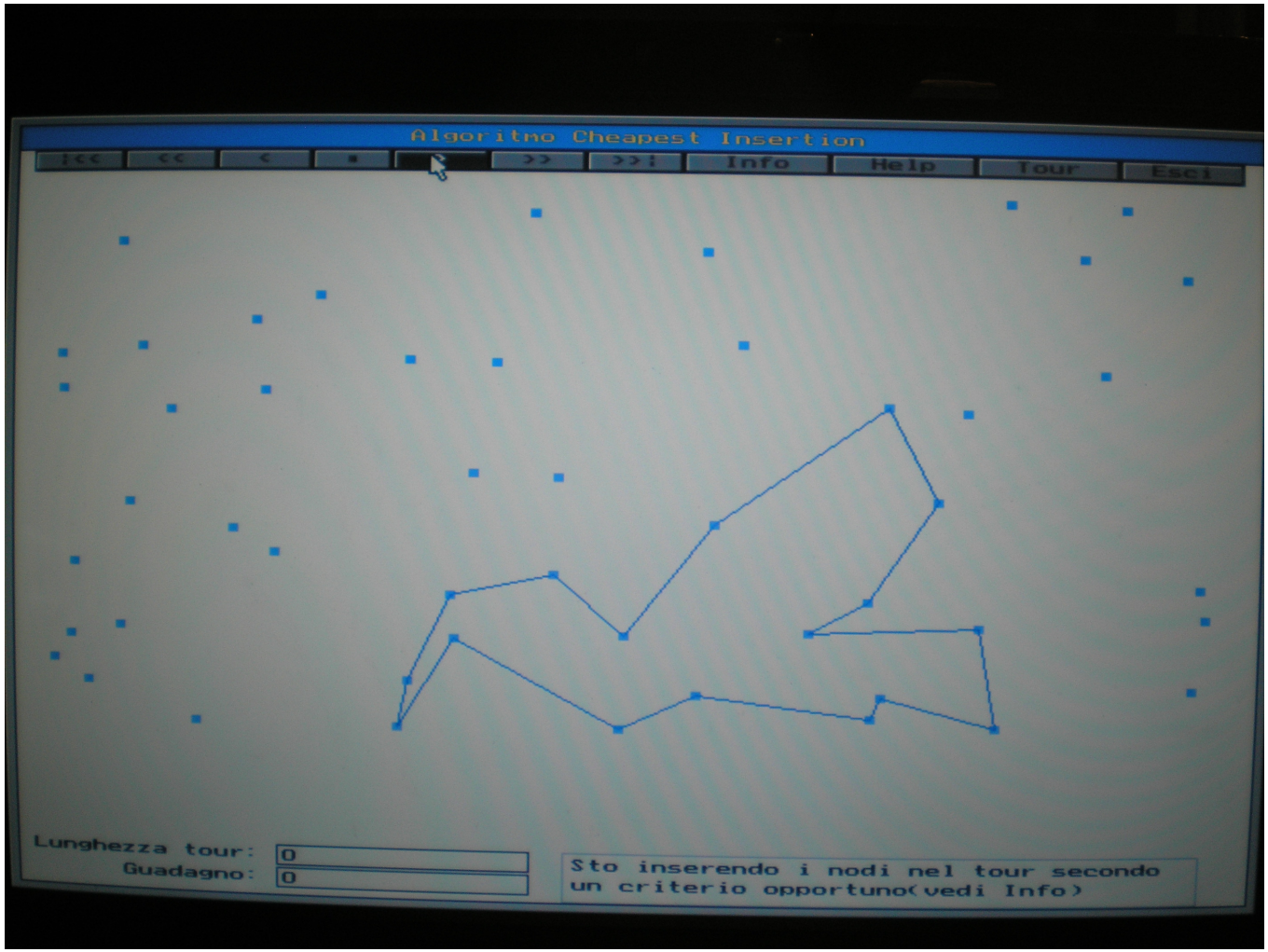
An example

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



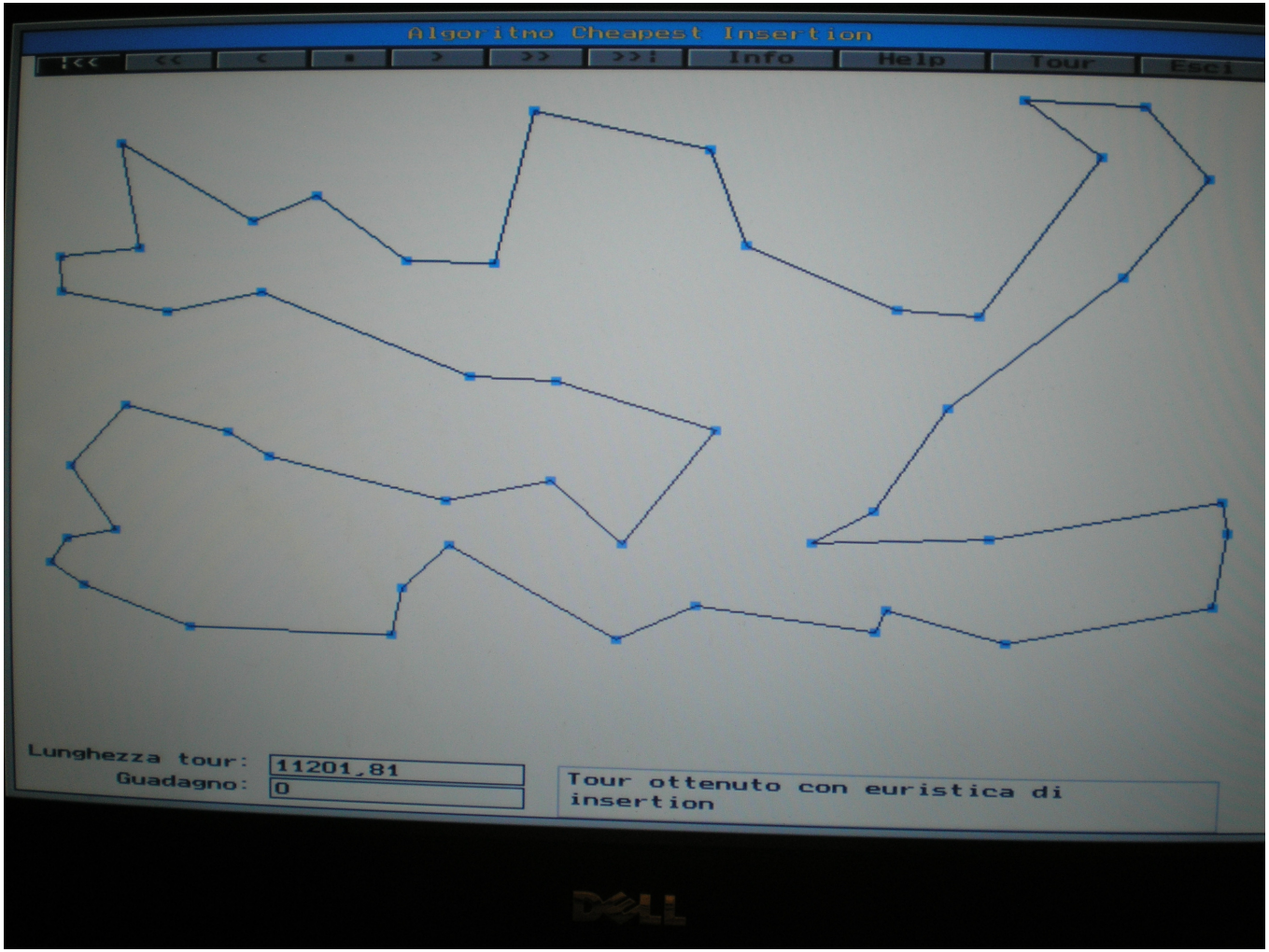
An example

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 $\Theta\left(n^3\right)$

It can be reduced to $\Theta\left(n^2 \log n\right)$ 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 φ_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

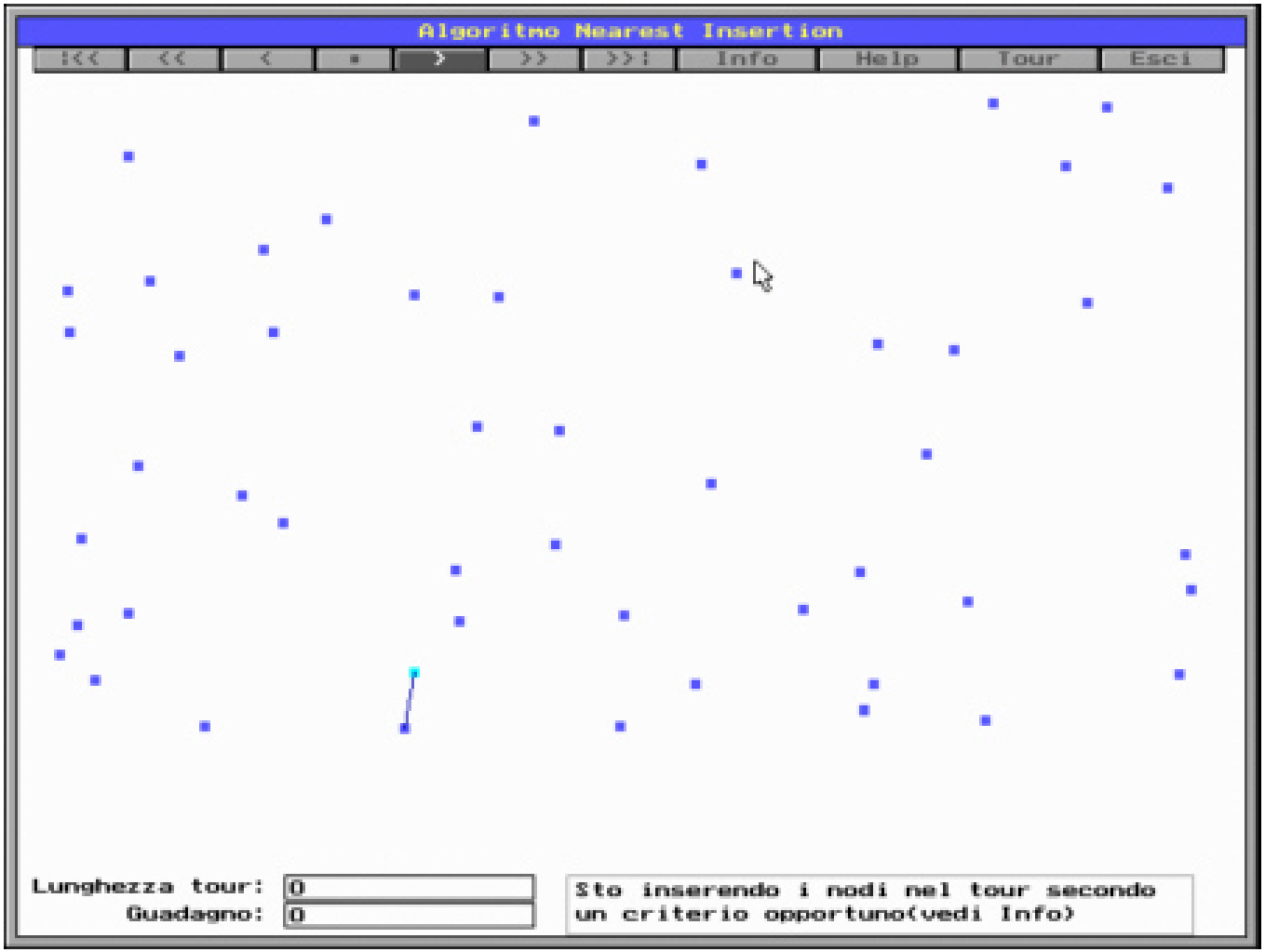
$$(s_i, s_{i+1}) = \arg \min_{(s_i,s_{i+1}) \in x} \left(c_{s_i,k} + c_{k,s_{i+1}} - c_{s_i,s_{i+1}} \right)$$

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

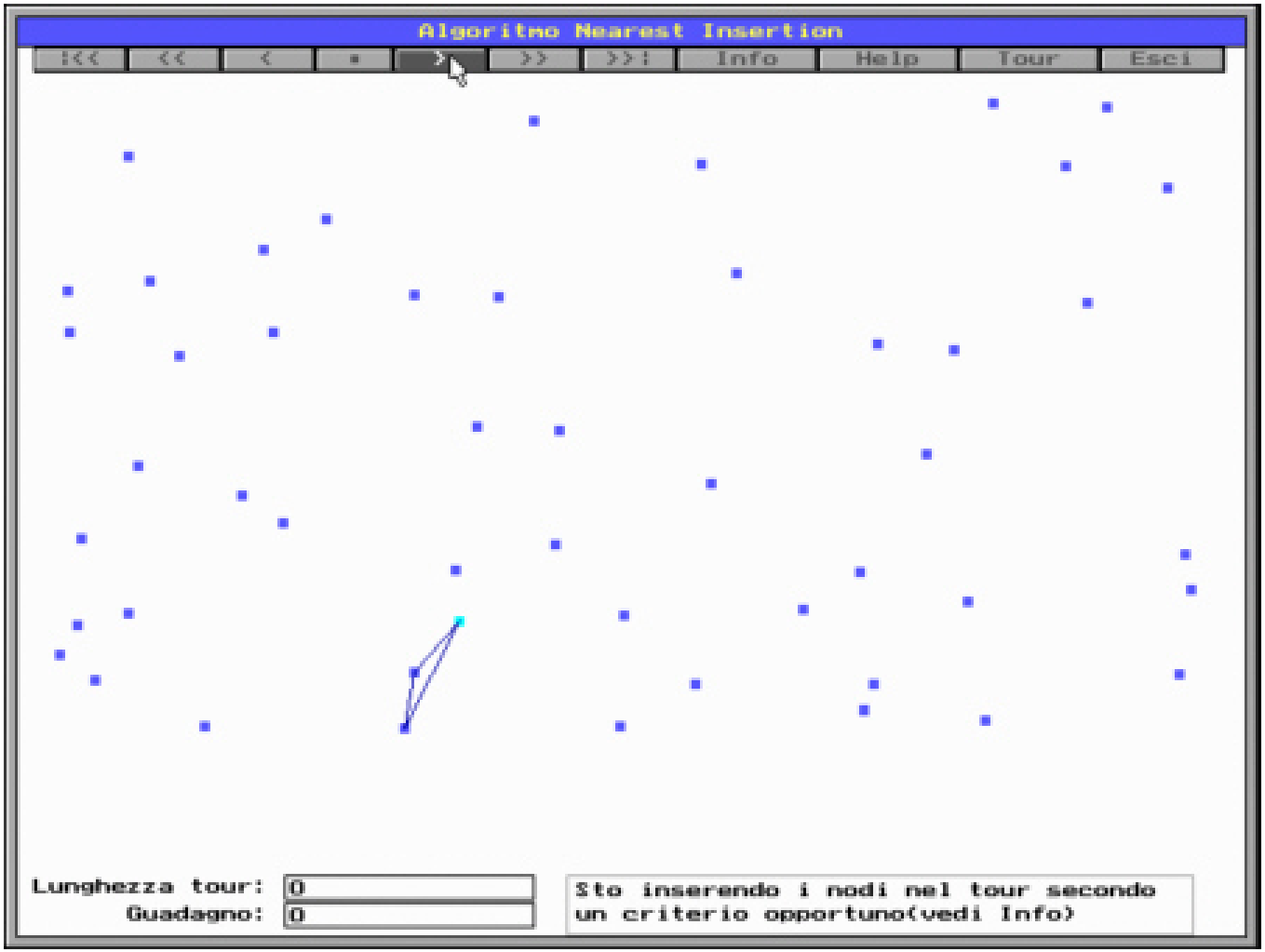
An example

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



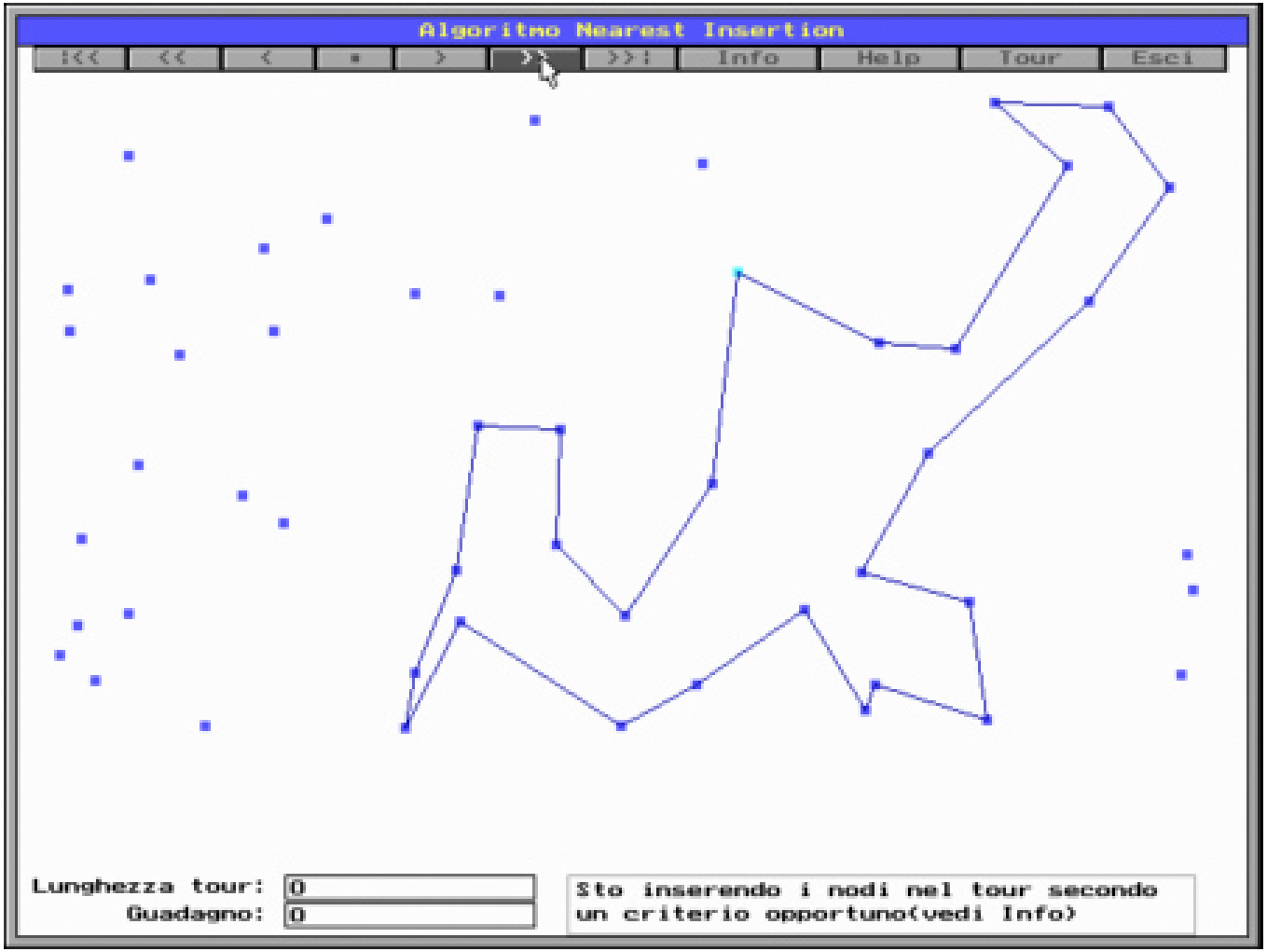
An example

Create a circuit (as in *CI*)



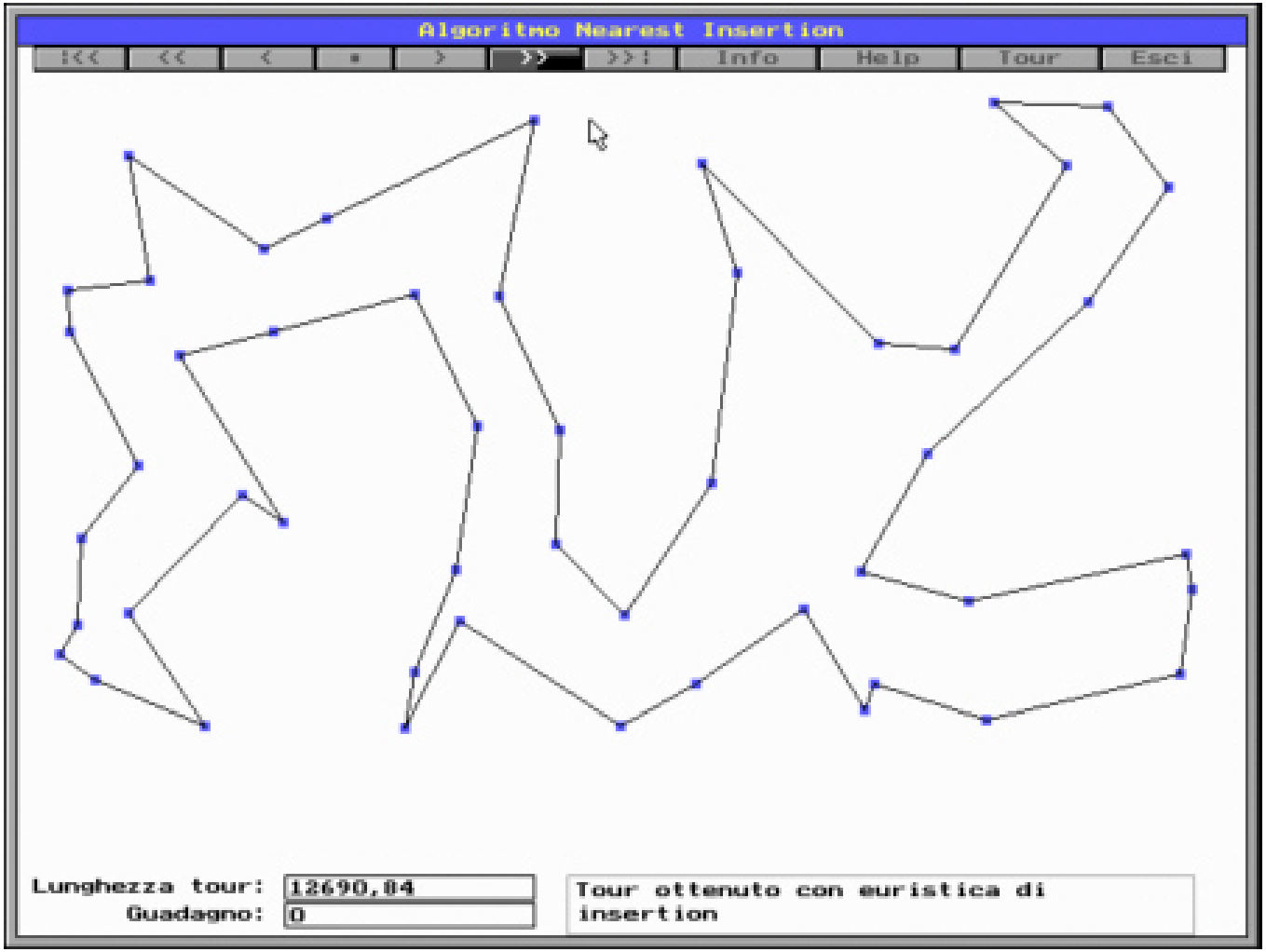
An example

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 $\Theta\left(n^3\right)$

It can be reduced to $\Theta\left(n^2\right)$ 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

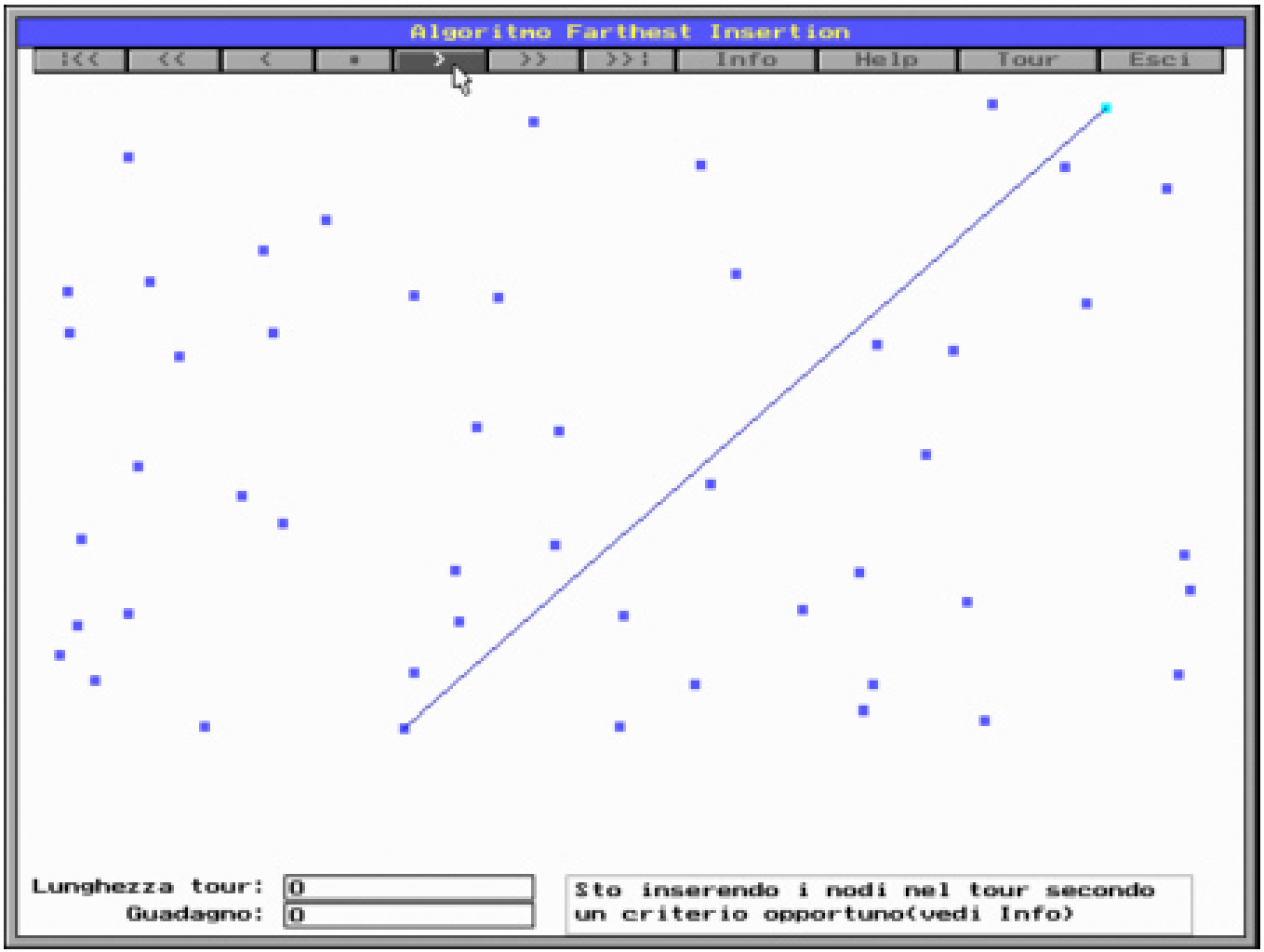
$$(s_i, s_{i+1}) = \arg \min_{(s_i,s_{i+1}) \in x} (c_{s_i,k} + c_{k,s_{i+1}} - c_{s_i,s_{i+1}})$$

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)

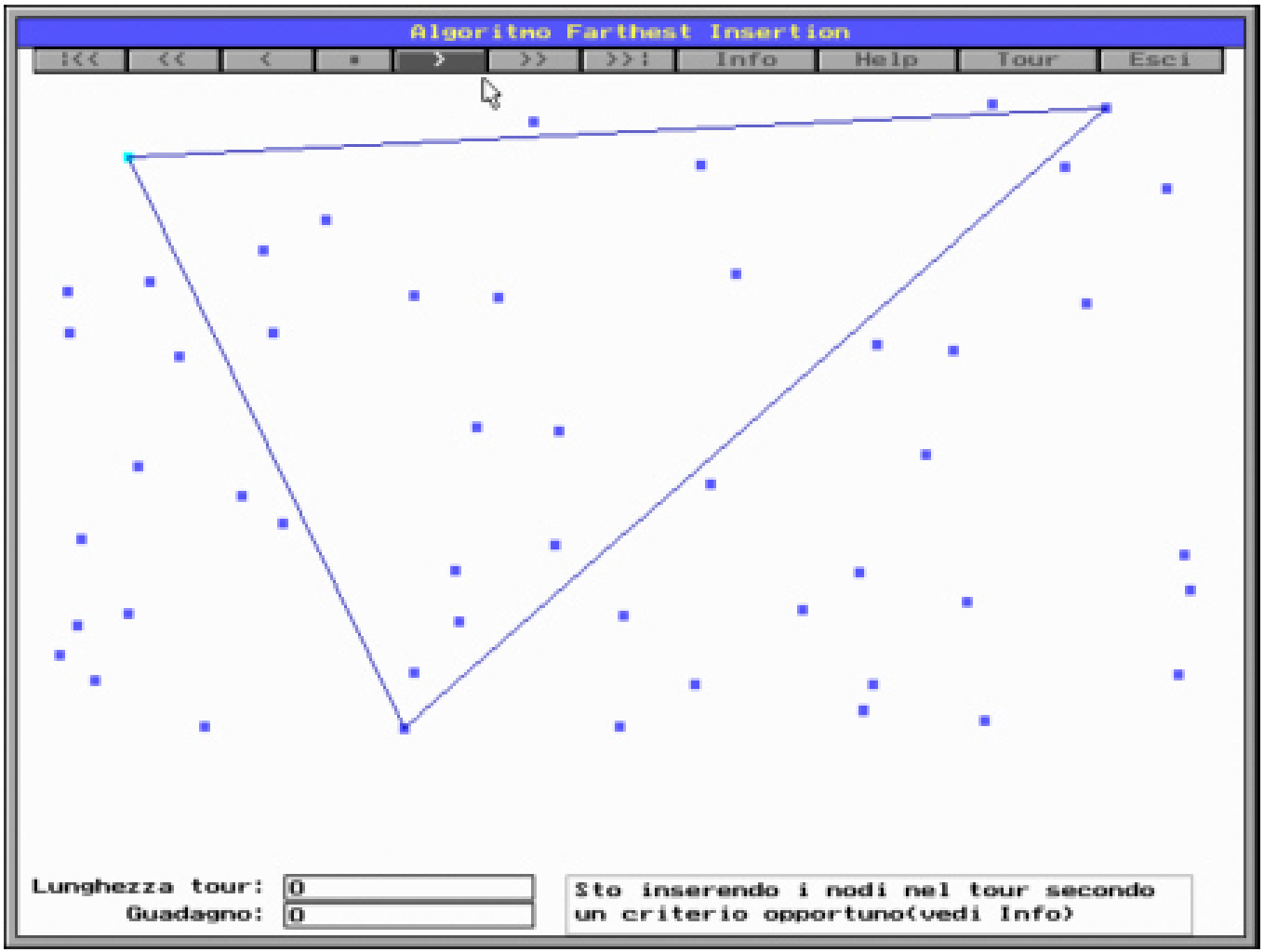
An example

Start reaching immediately the farthest node



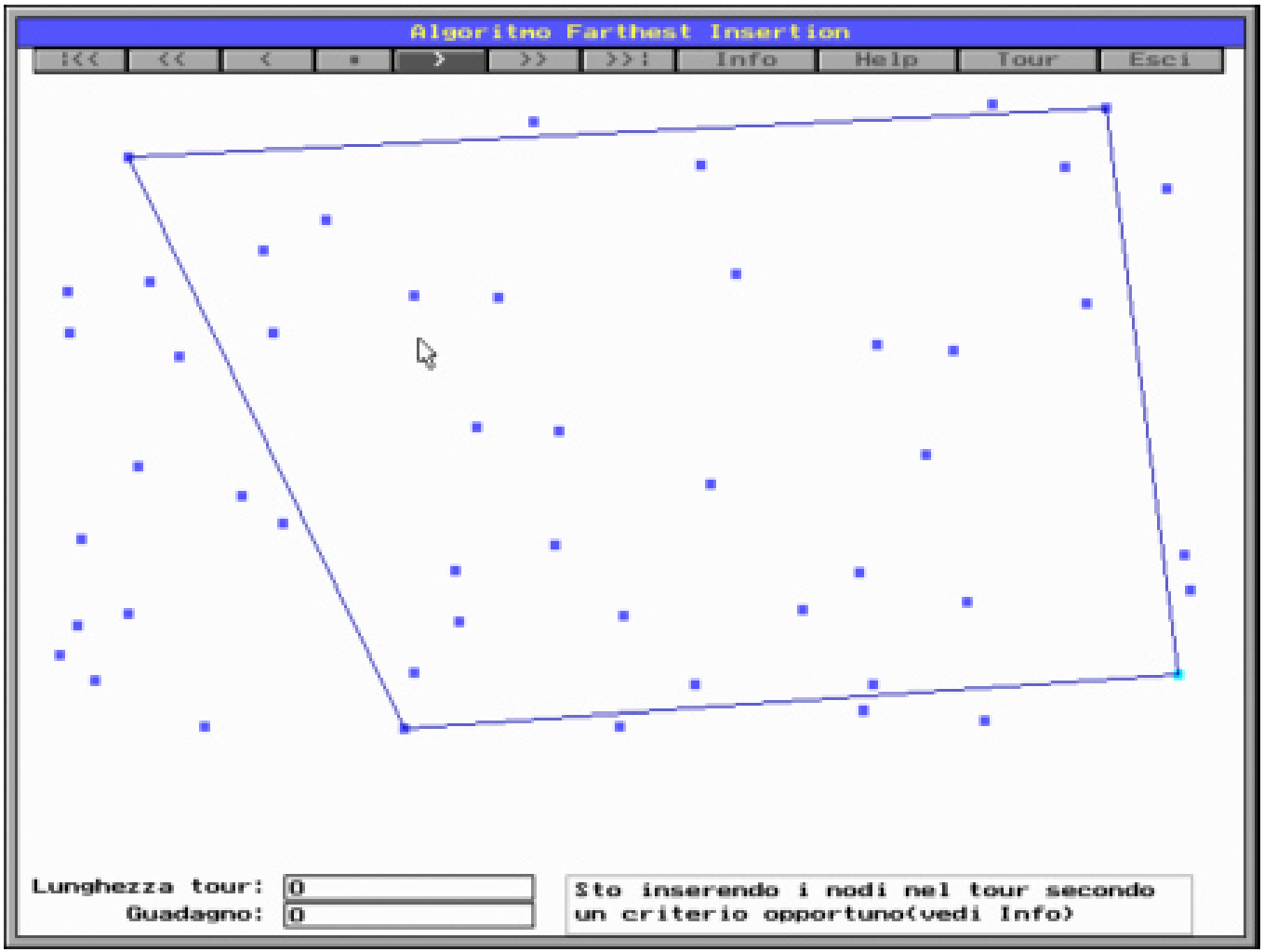
An example

And go on like that



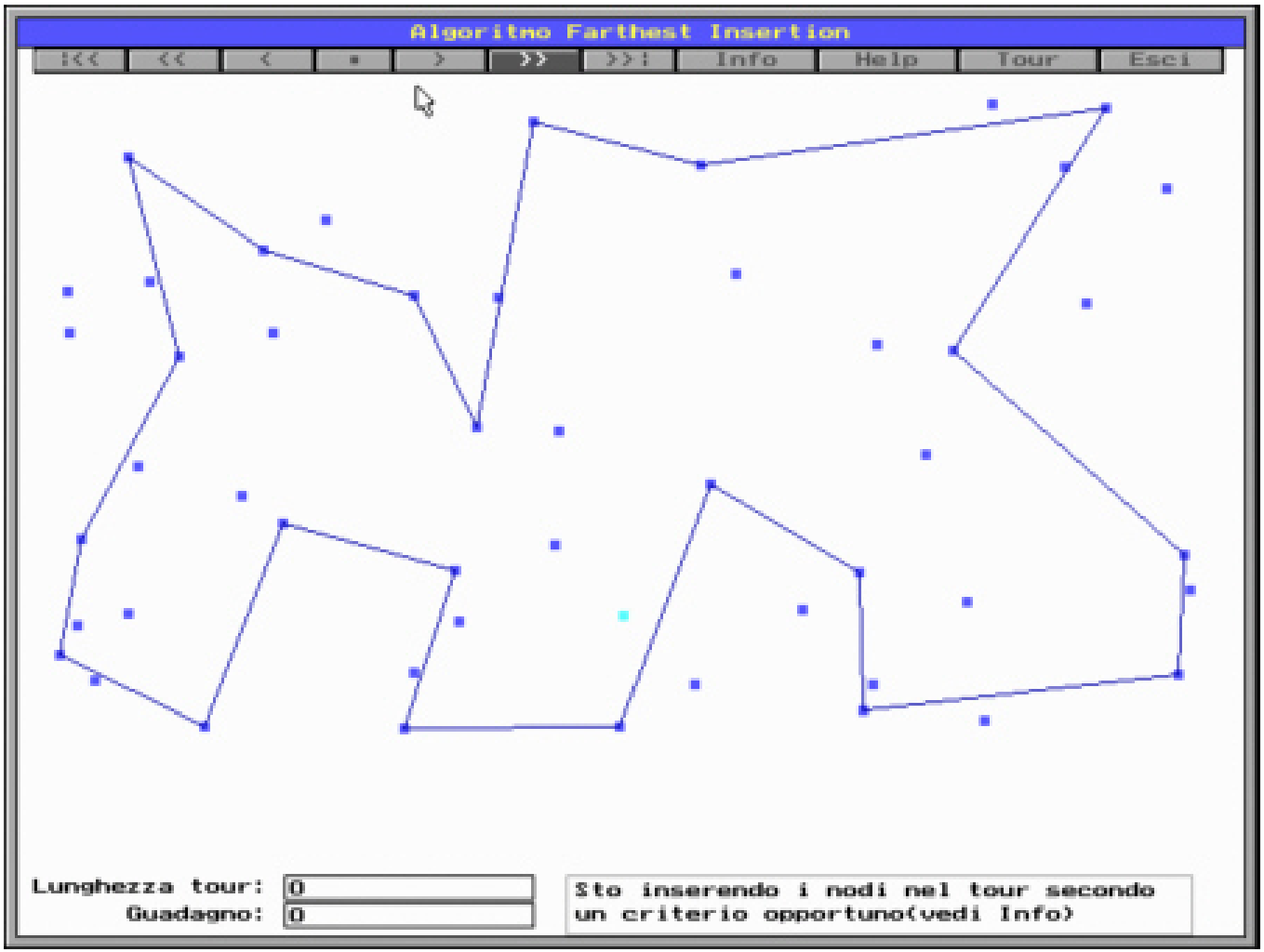
An example

But always inserting these nodes in the best possible way



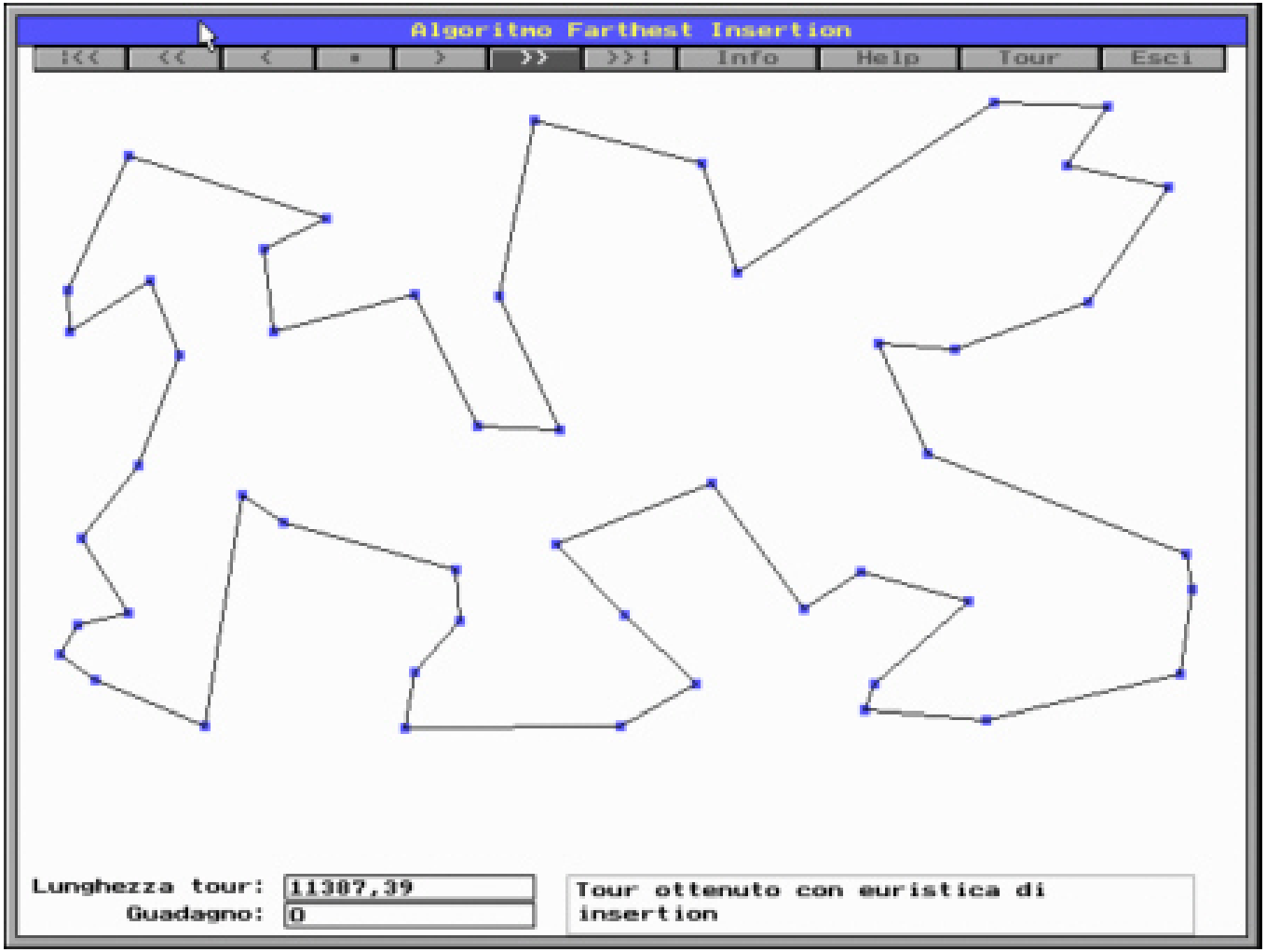
An example

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 $\Theta\left(n^3\right)$

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

Effectiveness and efficiency

A constructive algorithm performs at most $n = |B|$ steps consisting of

1. the construction of $\Delta_A^+(x)$
2. the evaluation of $\varphi_A(i, x)$ for each $i \in \Delta_A^+(x)$
3. the selection of the element i minimizing $\varphi_A(i, x)$
4. the update of x (and auxiliary data structures)

In general, the complexity is a polynomial of rather low order dominated by the first two components

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

General features of construction algorithms

Constructive algorithms

1. are intuitive
2. are simple to design, analyze and implement
3. are very efficient
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 scarse
   - on some problems they cannot even guarantee a feasible solution

Then, it is fundamental to study the problem before the algorithm

When are they used?

Constructive 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

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 criterium $\varphi_A(i, x)$ (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^*);

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	6	8	24	12

A				
	1	0	0	0
	1	0	0	0
	1	1	0	0
	0	1	1	0
	0	0	1	0
	0	0	0	1

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)

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

Regret-based constructive heuristics

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

Regret criterium

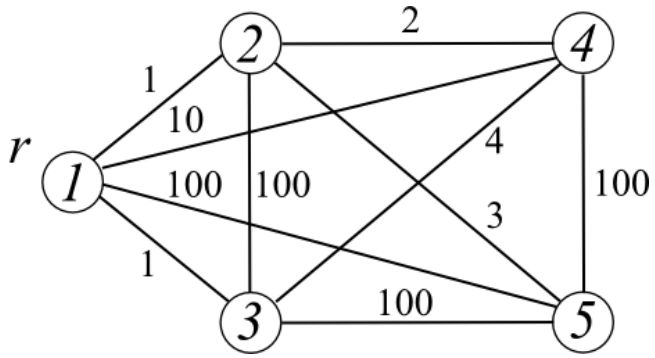
A typical regret-based heuristic consists in

- 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 each class
  This estimates the damage incurred by postponing the best choice
- choose the best choice of the class for which the regret is maximum

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, vertex 3 in subtree 2; 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 $i^{(t)}$

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

- 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, $T_{roA} = |B|^2 T_A$

Example: roll-out for the *SCP*

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

$$A = \begin{array}{|ccccc|} \hline 1 & 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 \\ 1 & 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 1 \\ \hline \end{array}$$

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 $A^{[1]}, \ldots, A^{[\ell]}$
- 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 $i^{(t)}$

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

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

Constructive metaheuristics

The constructive algorithms have strong limitations on many problems

What can be done, without abandoning the general scheme?

Iterate the scheme to generate many (potentially) different solutions

- the efficiency decreases: the computational times are summed
- the effectiveness increases: the best solution is returned

The trade-off must be carefully tuned

The iterated scheme can apply

- multi-start, that is different algorithm at each iteration $l = 1, \ldots, \ell$ (this requires to define multiple $\mathcal{F}_{A_l}$ and φ_{A_l})

but it is more flexible to apply metaheuristics, that exploit

- randomization (operations based on a random seed), as in the case of semigreedy algorithms, *GRASP* and *Ant System* (partly, *ART*)
- memory (operations based on the solutions of previous iterations), as in the case of *ART*, cost perturbation and *Ant System*

Termination condition

The iterated scheme can ideally proceed for an infinite time

In pratice, one uses termination conditions that can be "absolute"

1. a given total number of iterations of the basic scheme
2. a given total execution time
3. a given target value of the objective

or "relative" to the profile of f^*

1. a given number of iterations of the basic scheme without improving f^*
2. a given execution time without improving f^*
3. a given minimum ratio between the improvement of f^* and the number of iterations of the basic scheme or the execution time (*e.g.:* f^* *improves less than* 1% *in the last* 1 000 *iterations*)

Fair comparisons require absolute conditions

Constructive metaheuristics

The main constructive metaheuristics are

1. Adaptive Research Technique (*ART*) or Tabu Greedy:
   forbid some moves based on the solutions of the previous iterations

$$\min_{i:x\cup\{i\}\in\mathcal{F}^{[l]}} \varphi_A(i,x) \qquad \text{with } \mathcal{F}^{[l]} = \mathcal{F}^{[l]}\left(x_A^{[1]},\ldots,x_A^{[l-1]}\right) \subseteq \mathcal{F}$$

   This is much less popular than the other two

2. semigreedy and GRASP: use a randomized selection criterium

$$\min_{i:x\cup\{i\}\in\mathcal{F}} \varphi_A^{[l]}\left(i,x,\omega^{[l]}\right)$$

3. Ant System (AS): use a randomized selection criterium
   depending on the solutions of the previous iterations

$$\min_{i:x\cup\{i\}\in\mathcal{F}} \varphi_A^{[l]}\left(i,x,\omega^{[l]},x_A^{[1]},\ldots,x_A^{[l-1]}\right)$$

New information on the arcs of the construction graph guides the search

The *ART* uses memory, the *GRASP* randomization, the *AS* both

Adaptive Research Technique

It was proposed by Patterson et al. (1998) for the *CMSTP*

When deceivingly good elements are included in the first steps the final solution can be quite bad; to try and avoid that

- the roll-out approach makes a look-ahead on each possible element (*but a single step can be insufficient to identify the misleading ones*)
- the *ART* forbids some elements to drive subset x on the right path in the search space

(*how to identify the misleading elements?*)

The aim is diversification: forbidding elements of the previous solutions guarantees to obtain different solutions

The prohibitions are temporary, with an expiration time of L iterations; otherwise, building feasible solutions would become impossible

Adaptive Research Technique

Define a basic constructive heuristic A

Let T_i be the starting iteration of the prohibition for each element $i \in B$ and x^* be the best solution found

Set $T_i = -\infty$ for all $i \in B$ to indicate that no element is forbidden

At each iteration $l \in \{1, \ldots, \ell\}$

1. apply heuristic A forbidding all elements i such that $l \leq T_i + L$ (all prohibitions older than L iterations automatically expire); let $x^{[l]}$ be the resulting solution
2. if $x^{[l]}$ is better than x^*, set $x^* := x^{[l]}$ and save $T_i - l$ for all $i \in B$
3. decide which elements to forbid and set $T_i = l$ for them: each element is forbidden with probability π (*any better ideas?*)
4. make minor tweaks to L, π or T_i

At the end, return x^*

Example: *ART* for the *SCP*

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

$$A = \begin{array}{|ccccc|}\hline 1 & 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 \\ 1 & 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 1 \\ \hline\end{array}$$

Let $\pi = 0.15$ and pseudorandom sequence: $0.1, 0.9, 0.4, 0.5, 0.1, 0.2, \ldots$

1. the basic heuristic finds the solution $x^{[1]} = \{2, 3, 5, 4\}$ of cost $f\left(x^{[1]}\right) = 50$; forbid column 2 (because $0.1 \leq \pi < 0.9$, 0.4 and 0.5)
2. the basic heuristic finds the solution $x^{[2]} = \{3, 1\}$ of cost $f\left(x^{[2]}\right) = 33$ forbid column 3 (because $0.1 \leq \pi < 0.2$)
3. the basic heuristic finds the solution $x^{[3]} = \{1\}$ of cost $f\left(x^{[2]}\right) = 25$, that is optimal
4. ...

An unlucky sequence could forbid column 1 at step 2

Intensification

The *ART* has three basic parameters

- the total number of iterations ℓ (*tuned mainly by the available time*)
- the length L of the prohibition
- the probability π of the prohibition

An excessive diversification can hinder the discovery of the optimum

Intensification aims to focus the search on the more promising subsets

Diversification and intensification play complementary roles

Intensification can be obtained tuning the parameters based on

- problem data: assign to promising elements (e. g., cheapest)
  - a smaller probability π_i to be forbidden
  - a shorter expiration time L_i of the prohibition
- memory: for promising elements (e. g., appearing in the best known solutions)
  - reduce L_i (if $L_i = 0$, i is never forbidden)
  - periodically restart the algorithm with the $T_i - l$ values associated with the best known solution, instead of $T_i = -\infty$

Parameter tuning

How to assign effective values to the parameters?

The experimental comparison of different values is necessary but complex

1. it requires long experimental campaigns, because
   the number of configurations grows combinatorially with
   - the number of parameters
   - the number of tested values for each parameter
     (*the more sensitive the result, the more values must be tested*)
2. it risks overfitting, that is labelling as absolutely good
   values which are good only on the benchmark istances considered

The excess of parameters is an undesirable aspect, and often reveals an insufficient study of the problem and of the algorithm

More on this point later

Semi-greedy heuristics

A nonexact constructive algorithm has at least one step t which builds a subset $x^{(t)}$ not included in any optimal solution

Since the element selected is the best according to the selection criterium

$$i^* = \arg \min_{i \in \Delta_A^+(x)} \varphi_A(i, x)$$

necessarily $\varphi_A(i, x)$ is incorrect, but probably not completely wrong

The semi-greedy algorithm (Hart and Shogan, 1987) assumes that elements that lead to the optimum are very good for $\varphi_A(i, x)$, even if not strictly the best

How to know which one?

If it is not possible to refine $\varphi_A(i, x)$

- define a suitable probability distribution on $\Delta_A^+(x)$ favouring the elements with the best values of $\varphi_A(i, x)$
- select $i^*(\omega)$ according to the distribution function

Semi-greedy heuristics

Since the set of alternative choices is finite, this means to assign

- probability $\pi_A(i,x)$ to arc $(x, x \cup \{i\})$ of the construction graph (with a sum equal to 1 for the outgoing arcs of each node)

$$\sum_{i \in \Delta_A^+(x)} \pi_A(i,x) = 1 \quad \text{for all } x \in \mathcal{F}_A : \Delta_A^+(x) \neq \emptyset$$

- higher probabilities to the better elements for the selection criterium

$$\varphi_A(i,x) \leq \varphi_A(j,x) \Leftrightarrow \pi_A(i,x) \geq \pi_A(j,x)$$

for each $i, j \in \Delta_A^+(x), x \in \mathcal{F}_A$

This heuristic approach has important properties

- it can reach an optimal solution if there is a path from $\emptyset$ to X^* (*this is a basic condition*)
- it can be reapplied several times obtaining different solutions and the probability to reach the optimum grows gradually

(*each time the probability of always taking wrong ways decreases*)

Convergence to the optimum

The probability of

- following a path γ is the product of the probabilities on the arcs

$$\prod_{(y,y\cup\{i\}\in\gamma)} \pi_A(i,y)$$

- obtaining a solution x is the sum of those of the paths Γ_x reaching x

$$\sum_{\gamma\in\Gamma_x} \prod_{(y,y\cup\{i\}\in\gamma)} \pi_A(i,y)$$

This implies that the probability to reach the optimum:

1. is nonzero if and only if there exists a path of nonzero probability from $\emptyset$ to X^*
2. increases as $\ell \to +\infty$
   (*the probability of not reaching it decreases gradually*)

It tends to 1 for probabilistically approximatively complete algorithms

Convergence to the optimum

In this context, a *random walk* is a constructive metaheuristic in which all the arcs going out of the same node have equal probability

- it finds a path to the optimum with probability 1 (if one exists)
- the time required can be extremely long

The exhaustive algorithm is exact and requires finite time

A deterministic constructive heuristic sets all probabilities to zero except for those on the arcs of a single path

- it finds the optimum only if it enjoys specific properties
- it finds the optimum in a single run

Randomized heuristics that favour promising arcs and penalise the others

- accelerate the average convergence time
- decrease the guarantee of convergence in the worst case

There is a trade-off between expected and worst result

Arcs with zero probability can block the path to the optimum

Arcs with probability converging to zero reduce the probability to find it

Semi-greedy and GRASP

GRASP, that is Greedy Randomized Adaptive Search Procedure (Feo and Resende, 1989) is a sophisticated variant of the semi-greedy heuristic

- *Greedy* indicates that it uses a constructive basic heuristic
- *Randomized* indicates that the basic heuristic makes random steps
- *Adaptive* indicates that the heuristic uses an adaptive selection criterium $\varphi_A(i, x)$, depending also on x (not strictly necessary)
- *Search* indicates that it alternates the constructive heuristic and an exchange heuristic (differently from the semi-greedy approach)

The use of auxiliary exchange heuristics allows strongly better results

This aspect will be investigated in the following lessons

What probability function?

Several functions $\pi_A(i,x)$ are monotonous with respect to $\varphi_A(i,x)$

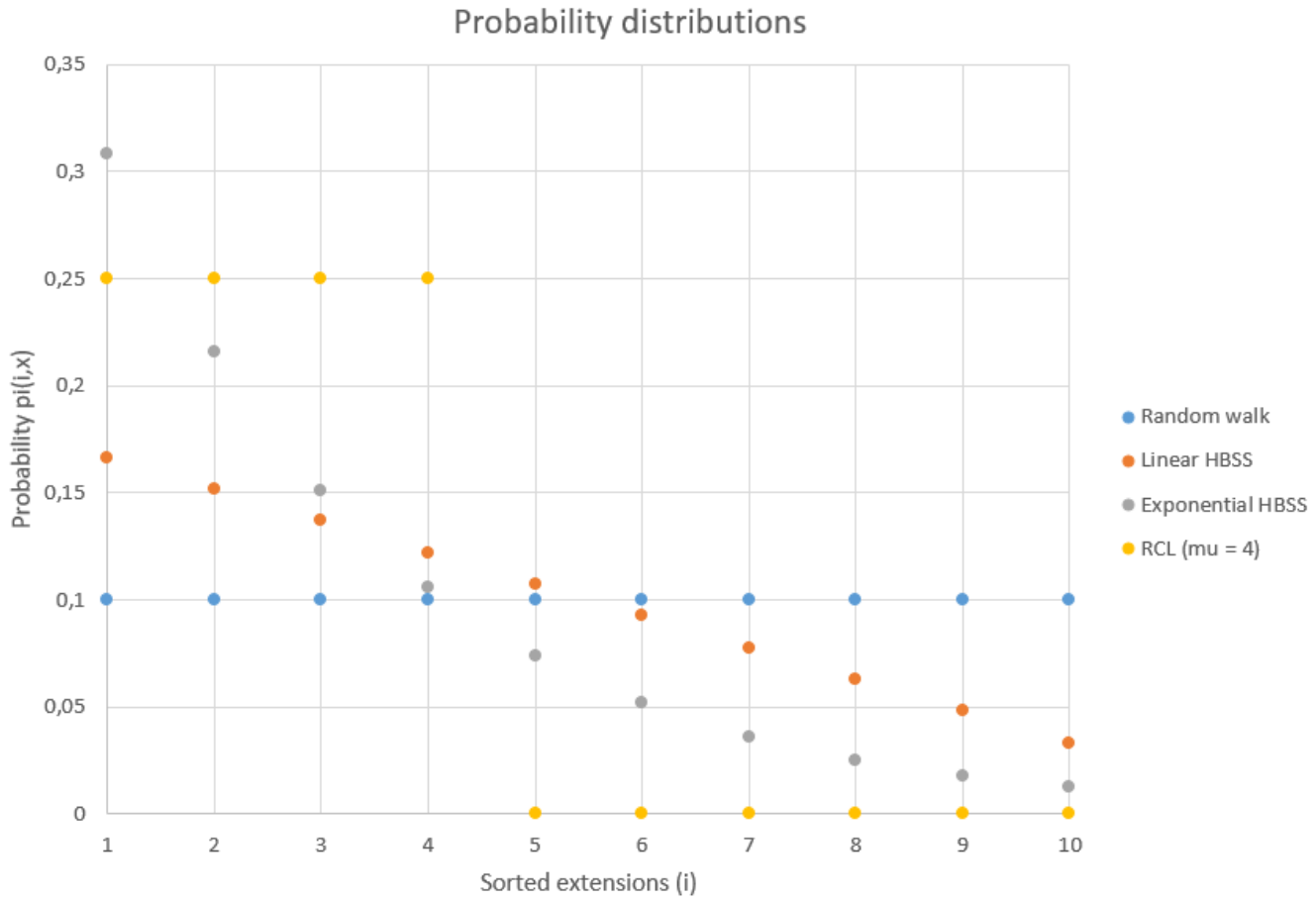
$$\varphi_A(i,x) \leq \varphi_A(j,x) \Leftrightarrow \pi_A(i,x) \geq \pi_A(j,x)$$

- uniform probability: each arc going out of x has the same $\pi_A(i,x)$; the algorithm performs a random path in $\mathcal{F}_A$ (random walk)
- Heuristic-Biased Stochastic Sampling (*HBSS*):
  - sort the arcs going out of x by nonincreasing values of $\varphi_A(i,x)$
  - assign a decreasing probability according to the position in the order based on a simple scheme (linear, exponential, ecc. . .)
- Restricted Candidate List (*RCL*):
  - sort the arcs going out of x by nonincreasing values of $\varphi_A(i,x)$
  - insert the best arcs in a list (*How many?*)
  - assign uniform probability to the arcs of the list, zero to the others

The most common strategy is the *RCL*, even if the zero probability arcs potentially cancel the global convergence to the optimum

Common probability functions

Suppose that at the current step $\left|\Delta_A^+(x)\right| = 10$ elements can be added



Definition of the RCL

Two main strategies are used to define the *RCL*

- cardinality: the RCL includes the best μ elements of $\Delta_A^+(x)$, where $\mu \in \left\{1, \ldots, \left|\Delta_A^+(x)\right|\right\}$ is a parameter fixed by the user
  - $\mu = 1$ yields the constructive basic heuristic
  - $\mu = |B|$ (i, e., $\left|\Delta_A^+(x)\right|$ for each x) yields the *random walk*
- value: the RCL includes all the elements of $\Delta_A^+(x)$ whose value is between $\varphi_{\min}$ and $(1-\mu)\,\varphi_{\min} + \mu\varphi_{\max}$ where

$$\varphi_{\min}(x) = \min_{i \in \Delta_A^+(x)} \varphi_A(i,x) \qquad \varphi_{\max}(x) = \max_{i \in \Delta_A^+(x)} \varphi_A(i,x)$$

  and $\mu \in [0;1]$ is a parameter fixed by the user
  - $\mu = 0$ yields the constructive basic heuristic
  - $\mu = 1$ yields the *random walk*

General scheme of *GRASP*

Algorithm GRASP(I)
$x^* := \emptyset$; $f^* := +\infty$; { Best solution found so far }
For $l = 1$ *to* ℓ *do*
  { Constructive heuristic with random steps }
  $x := \emptyset$;
  While $\Delta_A^+(x) \neq \emptyset$ *do*
    $\varphi_i := \varphi_A(i, x)$ for each $i \in \Delta_A^+(x)$
    $\pi :=$ AssignProbabilities$\left(\Delta_A^+(x), \varphi, \mu\right)$;
    $i :=$ RandomExtract$(\Delta_A^+(x), \pi)$;
    $x := x \cup \{i\}$;
  EndWhile;
  $x :=$ Search(x);
  If $x \in X$ and $f(x) < f^*$ *then* $x^* := x$; $f^* := f(x)$;
EndFor;
Return (x^*, f^*);

Example: *GRASP* for the *SCP*

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

$$A = \begin{array}{|ccccc|} \hline 1 & 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 \\ 1 & 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 1 \\ \hline \end{array}$$

Let $\mu = 2$ and the pseudorandom sequence be: $0.6, 0.8, \ldots$

1. start with the empty subset: $x^{(0)} = \emptyset$
2. build the *RCL* with columns 2 ($\varphi_2 = 2$) and 3 ($\varphi_3 = 4$); select column 3 (because $0.6 > 1/2$);
3. build the *RCL* with columns 2 ($\varphi_2 = 3$) and 1 ($\varphi_3 = 6.25$); select column 1 (because $0.8 > 1/2$);
4. the solution obtained is $x = \{3, 1\}$ of cost $f(x) = 33$

With $\mu = 2$, the optimal solution cannot be obtained; with $\mu = 3$ it can

The optimum is found with $\mu = 2$ if a destructive phase is applied

Reactive semi-greedy algorithm

Once again there are parameters to tune:

- the number of iterations ℓ
- the value μ determining the size of the *RCL*

An idea to exploit memory is to learn from the previous results

1. select m configurations of parameters $\mu_1, \ldots, \mu_m$ and set $\ell_r = \ell/m$
2. run each configuration μ_r for ℓ_r iterations
3. evaluate the mean $\bar{f}(\mu_r)$ of the results obtained with μ_r
4. update the number of iterations ℓ_r for each μ_r based on $\bar{f}(\mu_r)$

$$\ell_r = \frac{\frac{1}{\bar{f}(\mu_r)}}{\sum_{s=1}^{m} \frac{1}{\bar{f}(\mu_s)}} \ell \quad \text{for } r = 1, \ldots, m$$

   increasing it for the more effective configurations

5. repeat the whole process, going back to point 2, for R times

Other scheme use scores based on the number of best known results

Cost perturbation methods

Instead of forbidding/forcing some choices, or modifying their probability, it is possible to modify the appeal of the available choices

Given a basic constructive heuristic A, at each step of iteration l

- tune the selection criterium $\varphi_A(i,x)$ with a factor $\tau_A^{[l]}(i,x)$

$$\psi_A^{[l]}(i,x) = \frac{\varphi_A(i,x)}{\tau_A^{[l]}(i,x)}$$

- update $\tau_A^{[l]}(i,x)$ based on the previous solutions $x^{[1]},\ldots,x^{[l-1]}$

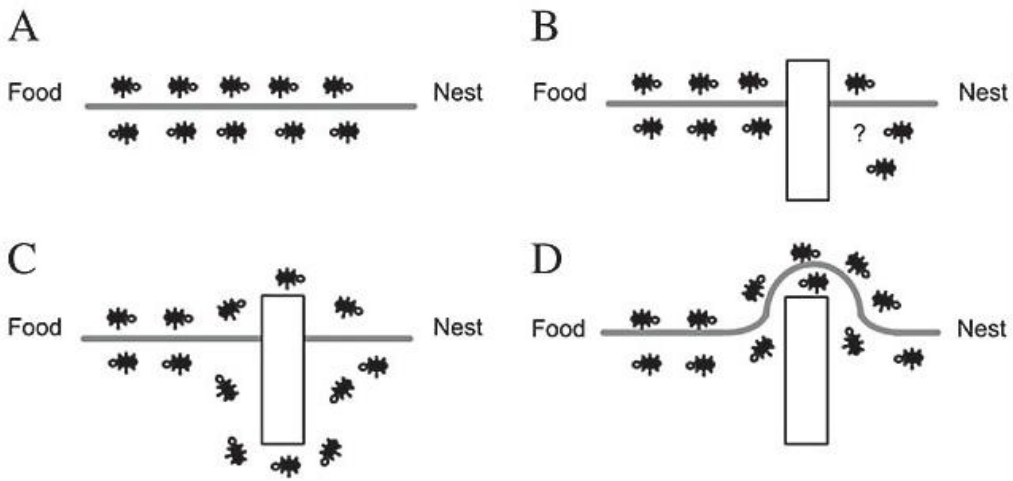
The elements with a better $\varphi_A(i,x)$ tend to be favoured,
but $\tau_A^{[l]}(i,x)$ tunes this effect, promoting

- intensification if $\tau_A^{[l]}(i,x)$ increases for the most frequent elements; this favours solutions similar to the previous ones
- diversification if $\tau_A^{[l]}(i,x)$ decreases for the most frequent elements; this favours solutions different from the previous ones

Ant Colony Optimization

It was devised by Dorigo, Maniezzo and Colorni in 1991
drawing inspiration from the social behaviour of ants

Stigmergy = indirect communication among different agents who are influenced by the results of the actions of all agents



Each agent is an application of the basic constructive heuristic

- it leaves a trail on the data depending on the solution generated
- it performs choices influenced by the trails left by the other agents

The choices of the agent have also a random component

Trail

As in the semi-greedy heuristic

- a basic constructive heuristic A is given
- each step performs a partially random choice

Differently from the semi-greedy heuristic

- each iteration l runs h times heuristic A (population)
- all the choices of $\Delta_A^+(x)$ are feasible (*there is no RCL*)
- the probability $\pi_A(i,x)$ depends on
  1. the selection criterium $\varphi_A(i,x)$
  2. auxiliary information $\tau_A(i,x)$ denoted as trail produced in previous iterations (sometimes by other agents in the same iteration)

The trail is uniform at first ($\tau_A(i,x) = \tau_0$), and later tuned

- increasing it to favour promising choices
- decreasing it to avoid repetitive choices

For the sake of simplicity, the trail $\tau_A(i,x)$ is not associated to each arc $(x, x \cup \{i\})$, but is the same for blocks of arcs (e.g., depending only on i)

Random choice

Instead of selecting the best element according to criterium $\varphi_A(i,x)$, i is extracted from $\Delta_A^+(x)$ with probability

$$\pi_A(i,x) = \frac{\tau_A(i,x)^{\mu_\tau}\ \eta_A(i,x)^{\mu_\eta}}{\sum_{j\in\Delta_A^+(x)} \tau_A(j,x)^{\mu_\tau}\ \eta_A(j,x)^{\mu_\eta}}$$

where

- the denominator normalizes the probability
- the visibility is the auxiliary function

$$\eta_A(i,x) = \begin{cases} \varphi_A(i,x) & \text{for maximisation problems} \\ \dfrac{1}{\varphi_A(i,x)} & \text{for minimisation problems} \end{cases}$$

The promising choices have larger visibility

- the parameters μ_τ and μ_η tune the weights of the two terms

Balancing given and learned information

The original Ant System tunes the probabilities

$$\pi_A(i,x) = \frac{\tau_A(i,x)^{\mu_\tau}\,\eta_A(i,x)^{\mu_\eta}}{\sum_{j\in\Delta_A^+(x)} \tau_A(j,x)^{\mu_\tau}\,\eta_A(j,x)^{\mu_\eta}}$$

with parameters μ_η and μ_τ that control the amount of randomness

- $\mu_\eta \approx 0$ and $\mu_\tau \approx 0$ push towards randomness
- large values of μ_η and μ_τ push towards determinism (favour $\arg\max_{i\in\Delta^+(x)} \tau_A(i,x)^{\mu_\tau}\,\eta_A(i,x)^{\mu_\eta}$)

and the relative weight of the data and of memory

- $\mu_\eta \gg \mu_\tau$ favours the data, simulating the basic constructive heuristic which makes sense when the known solutions are not very significant
- $\mu_\eta \ll \mu_\tau$ favours memory, keeping close to the previous solutions which makes sense when the known solutions are very significant

($\textit{assuming}\ \tau_A(i,x) > 1\ \textit{and}\ \eta_A(i,x) > 1$)

Balancing given and learned information

The Ant Colony System variant splits the selection into two phases

1. decide the selection procedure
   - with probability q, to choose i deterministically
   - with probability $(1-q)$, choose i stochastically

   where parameter q tunes the randomness
   - $q \approx 0$ favours random choices
   - $q \approx 1$ favours deterministic choices
2. apply the selection procedure
   - the deterministic one selects the best element

$$i^* = \arg \max_{i \in \Delta^+(x)} \tau_A(i,x)\, \eta_A(i,x)^{\mu_\eta}$$

   - the stochastic one select a random element with probabilities

$$\pi_A(i,x) = \frac{\tau_A(i,x)\, \eta_A(i,x)^{\mu_\eta}}{\sum_{j \in \Delta_A^+(x)} \tau_A(j,x)\, \eta_A(j,x)^{\mu_\eta}}$$

   where parameter μ_η tunes the relative weight of data and memory
   - $\mu_\eta \gg 1$ favours the data
   - $\mu_\eta \ll 1$ favours memory

(*setting* $\mu_\tau = 1$ *as a form of normalisation*)

Trail update

At each iteration ℓ

1. run h istances of the basic heuristic A
2. select a subset $\tilde{X}^{[l]}$ of the solutions obtained, in order to favour their elements in the following iterations
3. update the trail according to the formula

$$\tau_A(i,x) := (1-\rho)\,\tau_A(i,x) + \rho \sum_{y \in \tilde{X}^{[l]} : i \in y} F_A(y)$$

where

- $\rho \in [0;1]$ is an oblivion parameter
- $F_A(y)$ is a fitness function expressing the quality of solution y (such that $F > \tau$: e.g., $F(y) = Q/f(y)$ for a suitable constant Q)

The purpose of the update is to

1. increase the trail on the elements of specific solutions ($y \in \tilde{X}^{[l]}$)
2. decrease the trail on the other elements

The oblivion parameter

$$\tau_A(i,x) := (1-\rho)\,\tau_A(i,x) + \rho \sum_{y \in \tilde{X}^{[l]} : i \in y} F_A(y)$$

The oblivion parameter $\rho \in [0;1]$ tunes the behaviour of the algorithm:

- diversification: a high oblivion ($\rho \approx 1$) cancels the current trail based on the intuition that
  - the solutions obtained are not trustworthy
  - different solutions should be explored
- intensification: a low oblivion ($\rho \approx 0$) preserves the current trail based on the intuition that
  - the solutions obtained are trustworthy
  - similar solutions should be explored

Selection of the influential solutions

$\tilde{X}^{[l]}$ collects the solutions around which the search will be intensified

- the classical Ant System considers all the solutions of iteration $l-1$
- the elitist methods consider the best known solutions
  - the best solution of iteration $l-1$
  - the best solution of all iterations $< l$

The elitist methods

- find better results in shorter time
- require additional mechanisms to avoid premature convergence

Some variants of the Ant System

- $\mathcal{MAX} - \mathcal{MIN}$ Ant System: imposes on the trail a limited range of values $[\tau_{\min}; \tau_{\max}]$, experimentally tuned
- HyperCube Ant Colony Optimization (HC-ACO): normalizes the trail between 0 and 1
- Ant Colony System: updates the trail on two levels
  - the global update (already seen) modifies it at each iteration ℓ

    The purpose is to intensify the search
  - the local update updates the trail at each application g of the basic heuristic in order to discourage identical choices in the following

$$\tau_A(i,x) := (1-\rho)\,\tau_A(i,x) \qquad \text{for each } i \in x_A^{[l,g]}$$

The purpose is to diversify the search

General scheme of the *Ant System*

Algorithm AntSystem(I)
$x^* := \emptyset$; $f^* := +\infty$; { Best solution found so far }
For $l = 1$ *to* ℓ *do*
  $\tilde{X}^{[l]} := \emptyset$;
  For $g = 1$ *to* h *do*
    $x := A(I, \tau_A)$; { Basic heuristic with random steps and memory }
    $x :=$ Search(x); { Improvement heuristic }
    If $f(x) < f^*$ *then* $x^* := x$; $f^* := f(x)$;
    $\tau_A :=$ LocalUpdate(τ_A, x); { Local update of the trail }
    $\tilde{X}^{[l]} :=$ Update($\tilde{X}^{[l]}, x$);
  EndFor;
  $\tau_A :=$ GlobalUpdate($\tau_A, \tilde{X}^{[l]}$); { Global update of the trail }
EndFor;
Return (x^*, f^*);

Convergence to the optimum

Some variants of the Ant System converge to the optimum with probability 1 (Gutjahr, 2002)

The analysis is based on the construction graph

- the trail $\tau_A(i, x)$ is laid down on the arcs $(x, x \cup \{i\})$
- no information from the data is used, that is $\eta_A(i, x) \equiv 1$ (*this strange assumption simplifies the computation, but is not necessary*)
- $\tau^{[l]}$ is the trail function at the beginning of iteration l
- $\gamma^{[l]}$ is the best path on the graph at the end of iteration l,
- $\left(\tau^{[l]}, \gamma^{[l-1]}\right)$ is the state of a nonhomogeneous Markov process:
  - the probability of each state depends only on the previous iteration
  - the process is nonhomogeneous because the dependency varies with l

The proof concludes that for $\ell \to +\infty$, with probability 1

1. at least one run follows an optimum path in $\mathcal{F}$
2. the trail τ tends to a maximum along one of the optimal paths, to zero on the other arcs

First variant with global convergence

The trail is updated with a variable coefficient of oblivion

$$\tau^{[l]}(i,x) := \begin{cases} \left(1-\rho^{[l-1]}\right)\tau^{[l-1]}(i,x) + \rho^{[l-1]}\dfrac{1}{|\gamma^{[l-1]}|} & \text{if } (x, x\cup\{i\}) \in \gamma^{[l-1]} \\ \left(1-\rho^{[l-1]}\right)\tau^{[l-1]}(i,x) & \text{otherwise} \end{cases}$$

where $\gamma^{[l-1]}$ is the best path found in the graph up to iteration $l-1$ and $|\gamma^{[l-1]}|$ is the number of its arcs (*to normalise the trail*)

If the oblivion decreases slowly enough

$$\rho^{[l]} \leq 1 - \frac{\log l}{\log(l+1)} \quad \text{and} \quad \sum_{l=0}^{+\infty} \rho^{[l]} = +\infty$$

then with probability 1 the state converges to (τ^*, γ^*), where

- γ^* is an optimal path in the construction graph
- $\tau^*(i,x) = \dfrac{1}{|\gamma^*|}$ for $(x, x\cup\{i\}) \in \gamma^*$, 0 otherwise

Second variant with global convergence

Alternatively, if the oblivion ρ remains constant,
but the trail is forced a slowly decreasing minimum threshold

$$\tau(i,x) \geq \frac{c_l}{\log(l+1)} \quad \text{and} \quad \lim_{l \to +\infty} c_l \in (0;1)$$

then with probability 1 the state converges to (τ^*, γ^*)

Here the oblivion is restricted by the minimum threshold

In pratice, all algorithms proposed so far in the literature

- associate the trail to groups of arcs $(x, x \cup \{i\})$
  (e.g., to element i)
- use constant values for parameters ρ and $\tau_{\min}$

therefore do not guarantee convergence

The trail τ, and therefore π, can tend to zero on every optimal path