Constructive heuristics (2/2)

Outline (1/2)

The travelling salesman problem: append or insert?
- Nearest neighbour
- Multi-start nearest neighbour
- Insertion heuristics
- A comparison on a larger instance

Vertex coloring: greedy and DSATUR
- The greedy coloring
- DSATUR
- What the greedy coloring guarantees
- DSATUR as a heuristic

Randomized construction

Outline (2/2)

Look-ahead and regret
- Regret insertion
- Look-ahead: the pilot method

What constructive heuristics give, and what they do not

Typical mistakes

C++ implementation

In summary

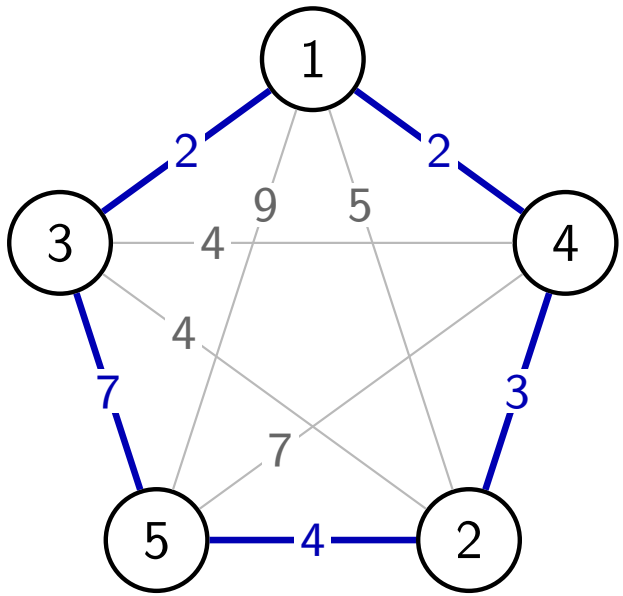
Where we were

- A **constructive heuristic** builds one feasible solution, one element at a time; a **greedy** algorithm adds the element that looks best now, and a choice, once made, is never revised.
- In the first part the order of the elements was the main design decision: efficiency for the KP, decreasing weights for the BPP (FFD, BFD), longest jobs first for $P \,||\, C_{\max}$ (LPT); no order is the best on every instance.
- In this second part the criteria are dynamic: tours for the TSP, colorings for the VCP; then randomized construction, regret and look-ahead.

Where we were: the TSP reference instance

The five cities are drawn on a pentagon, in the order $(1, 3, 5, 2, 4)$ of the optimal tour ($z(I) = 18$):

d_{ij}	1	2	3	4	5
1	—	5	2	2	9
2	5	—	4	3	4
3	2	4	—	4	7
4	2	3	4	—	7
5	9	4	7	7	—



The travelling salesman problem: append or insert?

Introduction: The travelling salesman problem: append or insert?

- In this section we build TSP tours: every permutation of the cities is feasible, so the whole design is in the selection criterion
- We study nearest neighbour, which appends the new city at the end of a path, its multi-start version, and the insertion heuristics, which insert it into a closed subtour
- We run them on the reference instance and compare them on a larger instance with 6 cities

Append or insert

- For the TSP every permutation of the cities is a feasible tour, so the feasibility test is trivial, and the whole design is in the selection criterion.

- Two families of constructive heuristics are natural, and they differ in **where** the new city is placed:
  - **append**: the partial solution is a path, and the new city is added at its end;
  - **insert**: the partial solution is a closed subtour, and the new city is inserted between two consecutive cities, wherever it costs least.

Nearest neighbour

- The *Nearest Neighbour Algorithm* (denoted by *NN*) is a constructive heuristic for the TSP.
- Starting from a given city s, it repeatedly moves from the last visited city to the *nearest* city not yet visited (in case of ties, the one with the smallest index).
- When all the cities have been visited, it goes back to s.

Algorithm: Nearest Neighbour

Input : TSP instance with n cities, distances $d[1..n][1..n]$; starting city s
Output : Tour $\sigma[1..n]$

```
1  function NearestNeighbour(n, d, s)
2      σ[1] ← s                                  // the path starts at s
3      visited := {s}
4      for len := 2 to n do
5          last := σ[len − 1]                    // last city of the path
6          k := the city j ∉ visited with minimum d[last][j] (smallest
             index)
7          σ[len] ← k                            // append k at the end
8          visited ← visited ∪ {k}
9      end
10     return σ          // the tour closes with the edge (σ[n], s)
11 end
```

The criterion of *NN* and its weakness

- The criterion is dynamic: the distances that matter are those from the last city, which changes at every step. *NN* runs in $O(n^2)$ time.
- **The weakness of** *NN* is visible in its last steps:
  - the cities that are left are those that were never the nearest to anything, often far from each other and from s;
  - the closing edge back to s is not chosen at all, it is forced.

Example: *NN* on the TSP reference instance (1/3)

Starting from city 1:

1. from city 1 the distances to $2, 3, 4, 5$ are $5, 2, 2, 9$: cities 3 and 4 are the nearest, and the tie is broken in favour of city 3;
2. from city 3 the distances to $2, 4, 5$ are $4, 4, 7$: city 2 (tie with 4);
3. from city 2 the distances to $4, 5$ are $3, 4$: city 4;
4. from city 4 only city 5 is left, at distance 7;
5. the tour closes with the edge $(5, 1)$, of length 9.

Example: *NN* on the TSP reference instance (2/3)

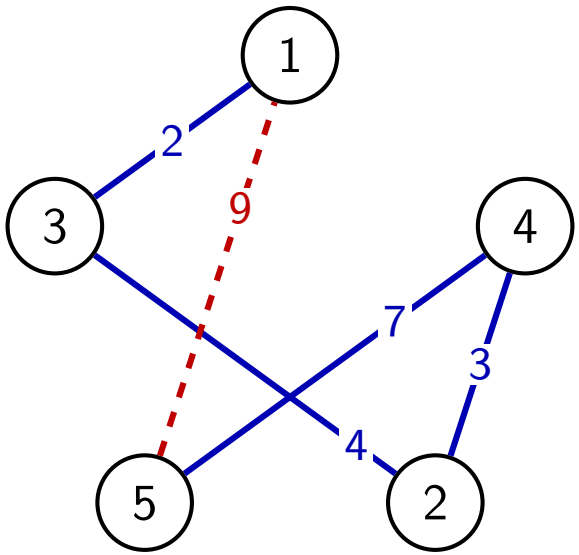
Hence $\sigma = (1, 3, 2, 4, 5)$ and

$$NN(I) = d_{13} + d_{32} + d_{24} + d_{45} + d_{51} = 2 + 4 + 3 + 7 + 9 = 25,$$

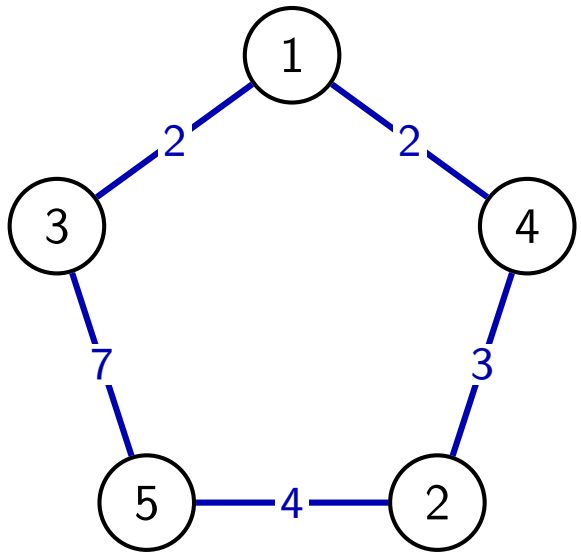
39% more than $z(I) = 18$.

- This is the tour used in the chapter «Heuristic algorithms: definitions» to illustrate the gap.
- The four edges chosen by the criterion have length $2 + 4 + 3 + 7 = 16$, and the forced closing edge alone costs 9, the largest distance of the instance.

Example: *NN* on the TSP reference instance (3/3)



NN from city 1: $L = 25$
(dashed: the closing edge)



optimal tour: $L = 18$

Multi-start nearest neighbour

- The result of *NN* depends on the starting city, which is an arbitrary choice: the tour is a cycle and has no natural first city.
- Since *NN* is fast, it can simply be run from every city.

The *multi-start nearest neighbour algorithm* runs *NN* once from every city $s \in \mathcal{V}$ and returns the shortest of the n tours obtained
(in case of ties, the first one found).

Algorithm: Multi-start Nearest Neighbour

Input : TSP instance with n cities, distances $d[1..n][1..n]$

Output: Tour $\sigma^{best}[1..n]$

```
1  function MultiStartNN(n, d)
2      L^best := +∞
3      for s := 1 to n do
4          σ := NearestNeighbour(n, d, s)
5          if L(σ) < L^best then
6              σ^best ← σ, L^best ← L(σ)  // new best tour
7          end
8      end
9      return σ^best
10 end
```

From an arbitrary choice to a search

- The running time is n times that of NN, i.e., $O(n^3)$.
- Multi-start is the simplest way to turn an arbitrary choice into a search: each start gives a different solution, and the best one is kept.

Example: multi-start *NN* on the TSP reference instance (1/3)

The five runs of *NN* give

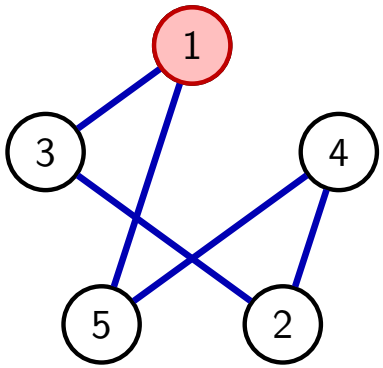
start s	1	2	3
tour	$(1, 3, 2, 4, 5)$	$(2, 4, 1, 3, 5)$	$(3, 1, 4, 2, 5)$
L	25	18	18

start s	4	5
tour	$(4, 1, 3, 2, 5)$	$(5, 2, 4, 1, 3)$
L	19	18

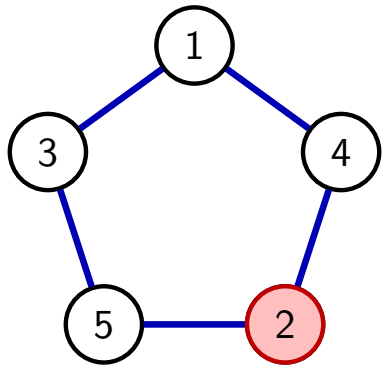
Example: multi-start *NN* on the TSP reference instance (2/3)

- The tours from cities 2, 3 and 5 are the optimal tour $(1, 3, 5, 2, 4)$, traversed from different cities and in different directions.
- The multi-start returns the tour from city 2, of length $18 = z(I)$.

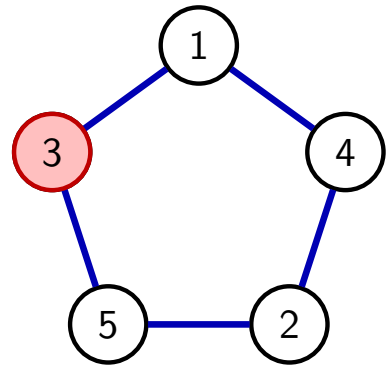
Example: multi-start *NN* on the TSP reference instance (3/3)



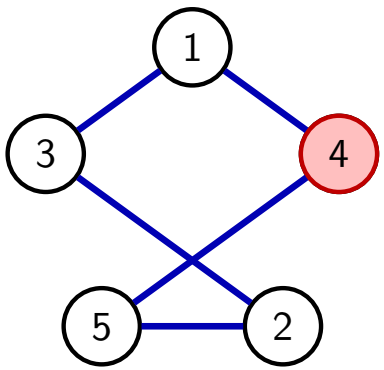
start 1: $L = 25$



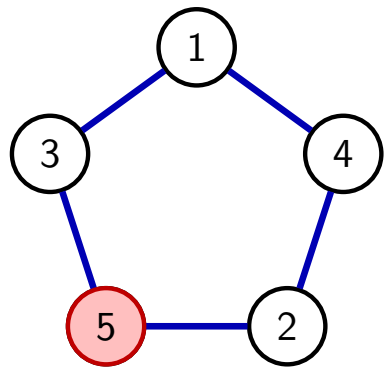
start 2: $L = 18$



start 3: $L = 18$



start 4: $L = 19$



start 5: $L = 18$

The insertion cost

- *NN* can only add a city at the end of a path. An **insertion heuristic** keeps a closed *subtour* T through some of the cities, and inserts the new city wherever it is cheapest.

- Inserting city k between two consecutive cities i and j of T replaces the edge (i, j) with the edges (i, k) and (k, j), and increases the length by the **insertion cost**

$$\Delta(i, k, j) = d_{ik} + d_{kj} - d_{ij} \quad \text{for every edge } (i, j) \text{ of the subtour } T.$$

On metric instances $\Delta(i, k, j) \geq 0$, by the triangle inequality.

Which city, and where

- Each step makes two decisions:
  - **which** city is inserted (the selection rule);
  - **where** (always the edge of minimum insertion cost).
- The three classical heuristics differ only in the selection rule.
- Let

$$d(q, T) = \min_{i \in T} d_{qi}$$

be the distance of city q from the subtour.

The insertion heuristics

The *insertion heuristics* for the TSP start from the subtour $T = (s)$ and, until all the cities are in T, select a city $k \notin T$ and insert it into the edge of T of minimum insertion cost:

nearest insertion (NI): k minimizes $d(q, T)$, the distance from the subtour

farthest insertion (FI): k maximizes $d(q, T)$

cheapest insertion (CI): k minimizes $\min_{(i,j)\in T} \Delta(i, q, j)$, the cost of its insertion

Ties are broken in favour of the smallest index, for the city and for the edge.

Algorithm: Insertion (NI, FI, CI)

Input : TSP instance with n cities, distances $d[1..n][1..n]$; starting city s; $rule \in \{NI, FI, CI\}$

Output : Tour $\sigma[1..n]$

```
1 function Insertion(n, d, s, rule)
2     σ := (s)                                  // subtour with one city
3     for len := 1 to n − 1 do
4         k := the city q ∉ σ selected by rule   // which city
5         a := the position a ∈ {1, ..., len} minimizing
               Δ(σ[a], k, σ[a+1]), with σ[len+1] = σ[1] // where
6         insert k between σ[a] and σ[a + 1]
7     end
8     return σ
9 end
```

Insertion: the first step, and append or insert

- **The subtour with one city.** The subtour (s) is a closed tour with a single "edge" (s, s) of length 0, so inserting k costs $d_{sk} + d_{ks} - 0 = 2d_{sk}$, and the subtour (s, k) goes from s to k and back. No special case is needed.

- **Append or insert.** Every tour built by *NN* could also be built by an insertion heuristic, but not conversely: inserting anywhere keeps the subtour closed at every step, so the length of the closing edge is paid, and controlled, from the beginning.

The intuition behind *FI*, and the running times

- **The intuition behind FI.** Taking the *farthest* city looks absurd for a greedy algorithm, but it builds first the outline of the tour, through the cities that are far apart, and then inserts the remaining cities, which are "on the way" and cost little.

- With the distances $d(q, T)$ and the minimum insertion costs updated incrementally after each insertion, *NI* and *FI* run in $O(n^2)$ time; a simple implementation of *CI* runs in $O(n^3)$.

Example: *NI* and *CI* on the TSP reference instance (1/4)

Starting from $T = (1)$. At each step we list, for every city $q \notin T$, its distance $d(q, T)$ (used by *NI*) and the city inserted:

subtour T	$d(q, T)$	inserted
(1)	$2{:}\,5,\ 3{:}\,2,\ 4{:}\,2,\ 5{:}\,9$	3 on $(1, 1)$, cost 4
$(1, 3)$	$2{:}\,4,\ 4{:}\,2,\ 5{:}\,7$	4 on $(1, 3)$, cost 4
$(1, 4, 3)$	$2{:}\,3,\ 5{:}\,7$	2 on $(4, 3)$, cost 3
$(1, 4, 2, 3)$	$5{:}\,4$	5 on $(2, 3)$, cost 7

Example: *NI* and *CI* on the TSP reference instance (2/4)

At each step we list, for every city $q \notin T$, its insertion costs on the edges of T, in the order of T (used by *CI*), and the city inserted:

subtour T	insertion costs	inserted
(1)	$2\colon 10,\ 3\colon 4,\ 4\colon 4,\ 5\colon 18$	3 on $(1,1)$, cost 4
$(1,3)$	$2\colon (7,7),\ 4\colon (4,4),\ 5\colon (14,14)$	4 on $(1,3)$, cost 4
$(1,4,3)$	$2\colon (6,3,7),\ 5\colon (14,10,14)$	2 on $(4,3)$, cost 3
$(1,4,2,3)$	$5\colon (14,8,7,14)$	5 on $(2,3)$, cost 7

Example: *NI* and *CI* on the TSP reference instance (3/4)

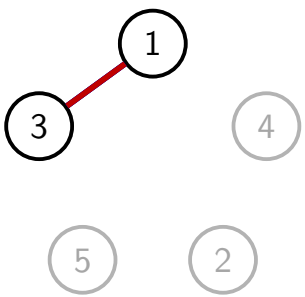
- *NI* selects the city with the smallest $d(q, T)$, *CI* the city with the smallest insertion cost.
- On this instance the two rules select the same cities at every step (city 3 at the first step by the tie with city 4).
- Both return $\sigma = (1, 4, 2, 5, 3)$, and

$$NI(I) = CI(I) = 4 + 4 + 3 + 7 = 18 = z(I),$$

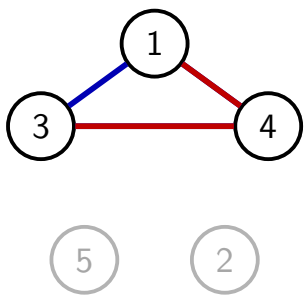
the optimal tour $(1, 3, 5, 2, 4)$ traversed in the opposite direction.

Example: *NI* and *CI* on the TSP reference instance (4/4)

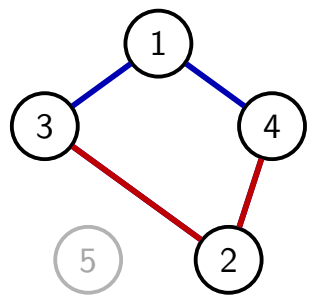
The subtours grow as follows (new edges in red):



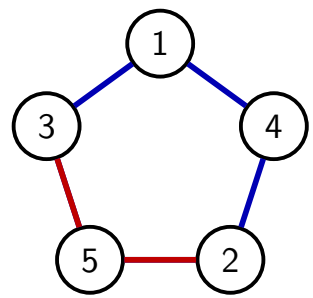
$L = 4$



$L = 8$



$L = 11$



$L = 18$

Example: *FI* on the TSP reference instance (1/3)

Starting from $T = (1)$:

subtour T	$d(q, T)$	insertion costs of the selected city	inserted
(1)	$2\colon 5,\ 3\colon 2,\ 4\colon 2,\ 5\colon 9$	$5\colon 18$	5 on $(1,1)$, cost 18
$(1,5)$	$2\colon 4,\ 3\colon 2,\ 4\colon 2$	$2\colon (0,0)$	2 on $(1,5)$, cost 0
$(1,2,5)$	$3\colon 2,\ 4\colon 2$	$3\colon (1,7,0)$	3 on $(5,1)$, cost 0
$(1,2,5,3)$	$4\colon 2$	$4\colon (0,6,4,4)$	4 on $(1,2)$, cost 0

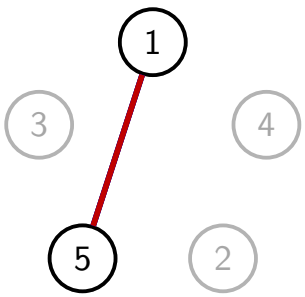
Example: *FI* on the TSP reference instance (2/3)

- At the first step *FI* takes city 5, the farthest from city 1, and the subtour $(1, 5)$ already has length $2d_{15} = 18$.
- Cities 2, 3 and 4 lie "on the way" and are inserted at zero cost, because

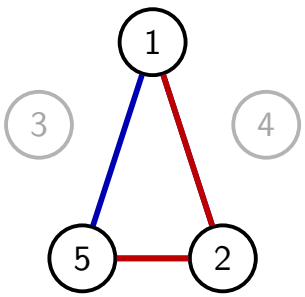
$$d_{12} + d_{25} = d_{15}, \qquad d_{53} + d_{31} = d_{51}, \qquad d_{14} + d_{42} = d_{12}.$$

- Hence $\sigma = (1, 4, 2, 5, 3)$ and $FI(I) = 18 = z(I)$: the same tour as *NI* and *CI*, built in a completely different order.

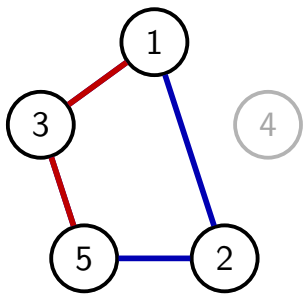
Example: *FI* on the TSP reference instance (3/3)



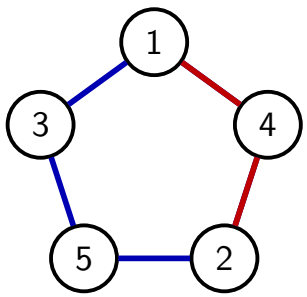
$L = 18$



$L = 18$



$L = 18$



$L = 18$

What is known in the worst case

- On metric instances, *NI* and *CI* never return a tour longer than twice the optimal one.
- The ratio of *NN* can grow like $\log n$, and only a logarithmic bound is known for *FI* (D.J. Rosenkrantz, R.E. Stearns, P.M. Lewis, *An analysis of several heuristics for the traveling salesman problem*, SIAM Journal on Computing 6 (1977) 563–581).
- In practice, on Euclidean instances, *FI* is usually the best of the insertion heuristics, and *NN* tours are typically about 25% longer than the optimum.

A comparison on a larger instance

- On the reference instance the three insertion heuristics, from every starting city, all find the optimal tour, and the differences between the rules do not show.
- To compare them we use a second TSP instance, with 6 cities in the plane and distances equal to the Euclidean distances rounded to the nearest integer.
- The instance is metric (checked on all the triples of cities).

Example: a TSP instance with 6 cities (1/4)

The cities have coordinates

$$1\colon (2,1), \quad 2\colon (5,5), \quad 3\colon (4,4), \quad 4\colon (0,5), \quad 5\colon (7,0), \quad 6\colon (4,1),$$

and distances

d_{ij}	1	2	3	4	5	6
1	—	5	4	4	5	2
2	5	—	1	5	5	4
3	4	1	—	4	5	3
4	4	5	4	—	9	6
5	5	5	5	9	—	3
6	2	4	3	6	3	—

Example: a TSP instance with 6 cities (2/4)

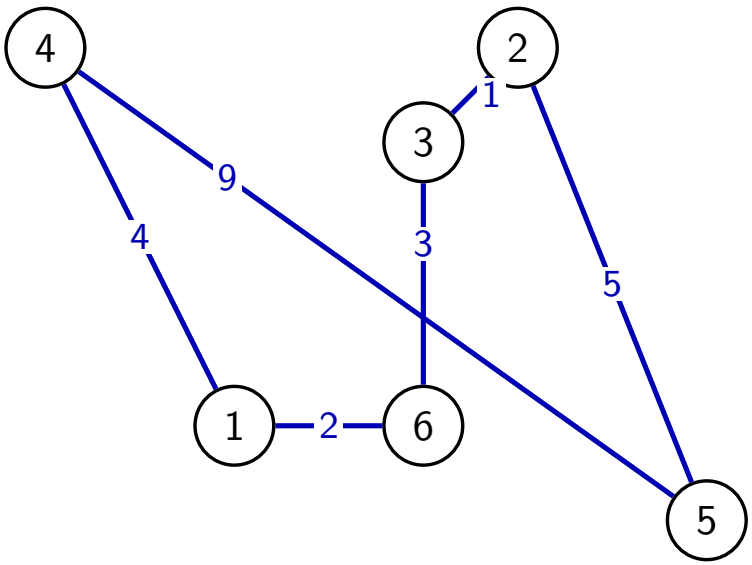
- The instance has $\frac{5!}{2} = 60$ distinct tours; enumerating them, the unique optimal tour is $(1, 4, 3, 2, 5, 6)$, with $z(I) = 19$, and only 7 tours have length smaller than 24.
- The heuristics of this section give:

	starting city					
	1	2	3	4	5	6
NN	25	24	25	24	24	24

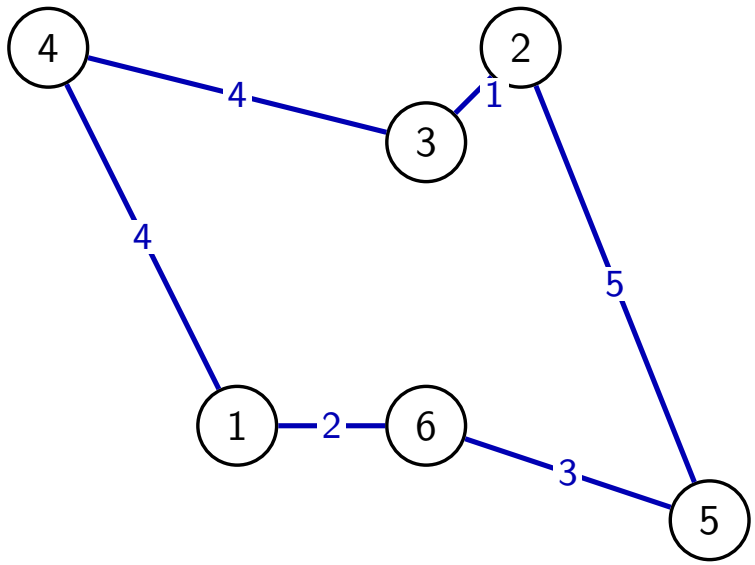
from city 1	L
NI	23
FI	19
CI	20

Example: a TSP instance with 6 cities (3/4)

The best tour of the multi-start *NN* has length 24, 26% above the optimum, while *FI* finds the optimal tour $(1, 6, 5, 2, 3, 4)$ (the optimal tour in the opposite direction):



multi-start *NN* (from city 2): $L = 24$



FI from city 1: $L = 19 = z(I)$

Example: a TSP instance with 6 cities (4/4)

- All six *NN* tours contain the edge $(4, 5)$, of length 9, the longest of the instance.
- Cities 4 and 5 are at opposite ends of the instance, and *NN*, from whatever city it starts, leaves at least one of them to the last steps, when the only way left is the long one.

Summary

Append: nearest neighbour:

- *NN* appends the nearest unvisited city; the closing edge is forced: $NN(I) = 25$ from city 1
- multi-start *NN* runs *NN* from every city and keeps the best tour: $18 = z(I)$

Insert: insertion heuristics:

- a closed subtour, the city inserted into the edge of minimum $\Delta(i, k, j) = d_{ik} + d_{kj} - d_{ij}$
- *NI*, *FI*, *CI* differ only in which city is selected; all give $18 = z(I)$ on the reference instance
- on the 6-city instance: multi-start *NN* 24, *NI* 23, *CI* 20, *FI* 19 $= z(I)$

Vertex coloring: greedy and DSATUR

Introduction: Vertex coloring: greedy and DSATUR

- In this section we color the vertices of a graph one at a time: the greedy coloring follows a given list, *DSATUR* builds the list on the fly
- We run both on the Petersen graph, and see that the order of the list can make a large difference
- We prove what the greedy coloring guarantees, and what it does not

The vertex coloring problem

- The problem of this section is the vertex coloring problem (VCP): its definition and its assumptions are on its page.
- The graph is $G = (V, E)$, and the optimal value is the chromatic number $\chi(G)$.

The greedy coloring

- The *Greedy Coloring Algorithm* (denoted by *GC*) is a constructive heuristic for the VCP.
- The colors are the integers $1, 2, 3, \ldots$
- The algorithm examines the vertices in the order of a given list λ and gives each vertex the smallest color not used by its neighbours already colored.

Algorithm: Greedy Coloring

Input : VCP instance $G = (V, E)$ with n vertices; list $\lambda[1..n]$

Output : Vertex-color assignment $col[1..n]$ and number of colors nc

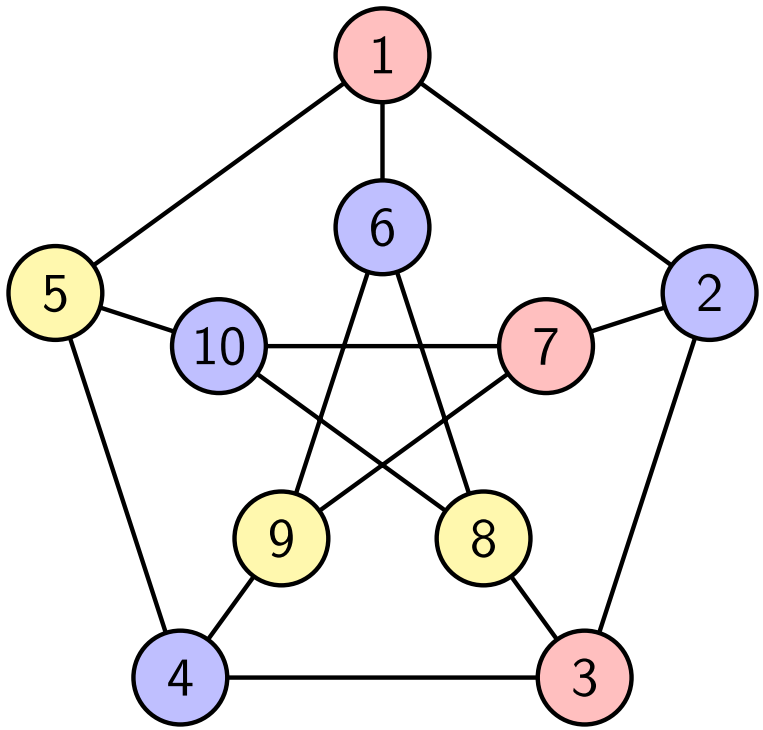
```
1 function GreedyColoring(n, E, λ)
2 |   col := new array[1..n], all entries 0      // 0 = not yet
  |    colored
3 |   for k := 1 to n do
4 |   |   v := λ[k]
5 |   |   U := { col[u] : u ∈ N(v), col[u] ≠ 0 }  // colors of
  |   |    the colored neighbours
6 |   |   col[v] ← min{ c ∈ ℤ_{≥1} : c ∉ U }  // smallest free
  |   |    color
7 |   end
8 |   return col, nc = max_v col[v]
9 end
```

The greedy coloring is feasible, and depends on the list

- The color classes $Q_c = \{v : col[v] = c\}$ are independent sets by construction, so GC always returns a feasible coloring.
- It runs in $O(n + |E|)$ time.
- As $LIST$, GC depends on the order of the list, and the order can make a large difference.

Example: GC on the Petersen graph, two orders (1/3)

The Petersen graph, with the colors given by GC with the list $\lambda = (1, 2, \ldots, 10)$:



Example: GC on the Petersen graph, two orders (2/3)

List $\lambda = (1, 2, \ldots, 10)$. The colors of the colored neighbours U and the color assigned are:

v	1	2	3	4	5	6	7	8	9	10
U	$\emptyset$	$\{1\}$	$\{2\}$	$\{1\}$	$\{1,2\}$	$\{1\}$	$\{2\}$	$\{1,2\}$	$\{1,2\}$	$\{1,3\}$
$col[v]$	1	2	1	2	3	2	1	3	3	2

- GC uses 3 colors (red $= 1$, blue $= 2$, yellow $= 3$ in the figure).
- The outer 5-cycle $1, 2, 3, 4, 5$ is an odd cycle and needs 3 colors, so $\chi(G) = 3$ and this coloring is optimal.

Example: GC on the Petersen graph, two orders (3/3)

List $\lambda = (1, 2, 3, 4, 5, 7, 8, 6, 9, 10)$**.** The first five vertices get the same colors as before; then

v	7	8	6	9	10
U	$\{2\}$	$\{1\}$	$\{1, 2\}$	$\{1, 2, 3\}$	$\{1, 2, 3\}$
$col[v]$	1	2	3	4	4

- GC uses 4 colors: vertex 8 takes color 2, which the first order gave to vertex 6.
- Vertices 9 and 10 then see all three colors among their neighbours.

DSATUR

- The *DSATUR Algorithm* (Degree of SATURation, denoted by *DSATUR*) is a constructive heuristic for the VCP. It does not use a fixed list.
- At each step it colors, with the smallest free color, an uncolored vertex of maximum **saturation degree**, i.e., with the largest number of *different* colors among its neighbours.
- Ties are broken by the largest degree in the subgraph of the uncolored vertices, and then by the smallest index.

Algorithm: DSATUR (1/2)

Input : VCP instance $G = (V, E)$ with n vertices
Output: Vertex-color assignment $col[1..n]$ and number of colors nc

```
1 function DSATUR(n, E)
2     col := new array[1..n], all entries 0
3     for k := 1 to n do
4         for each uncolored u do
5             sat[u] ← |{ col[w] : w ∈ N(u), col[w] ≠ 0 }|
                  // saturation degree
6             du[u] ← |{ w ∈ N(u) : col[w] = 0 }|
                  // degree among the uncolored
7         end
```

Algorithm: DSATUR (2/2)

function *DSATUR*(n, E) (continued)

    for $k := 1$ **to** n **do** (continued)

8        $v :=$ the uncolored vertex maximizing
          $(sat[v], du[v])$ lexicographically (smallest index)

9        $col[v] \leftarrow \min\{\, c \in \mathbb{Z}_{\geq 1} : c \neq col[w]\ \forall w \in N(v) \,\}$

10   **end**

11   **return** col, $nc = \max_v col[v]$

12 **end**

Why the saturation degree

- A vertex with many different colors around it has few colors left: if we postpone it, it may end up with none of the colors already in use.
- *DSATUR* colors first the vertices that are most constrained, the same idea as "the hardest items first" of FFD and *LPT*, but updated dynamically after each assignment.
- *DSATUR* is *GC* with a list built on the fly, so every property of *GC* that holds for every list holds for *DSATUR* too.
- A straightforward implementation runs in $O(n^2)$ time.

Example: *DSATUR* on the Petersen graph (1/2)

All the vertices have degree 3. The vertex chosen at each step, its saturation degree, its degree among the uncolored vertices and its color are:

step	1	2	3	4	5	6	7	8	9	10
v	1	2	3	4	5	6	8	7	9	10
$sat[v]$	0	1	1	1	2	1	2	1	2	2
$du[v]$	3	2	2	2	1	2	1	2	0	0
$col[v]$	1	2	1	2	3	2	3	1	3	2

Example: *DSATUR* on the Petersen graph (2/2)

- At step 1 all the vertices tie and vertex 1 is chosen.
- At step 2 the vertices 2, 5, 6 have saturation degree 1 and two uncolored neighbours each: the smallest index, 2, is chosen.
- At step 7 vertex 8 (neighbours 3 and 6, colors 1 and 2) has saturation degree 2 and goes before vertex 7.
- *DSATUR* uses $3 = \chi(G)$ colors, the same coloring as *GC* with the list $(1, 2, \ldots, 10)$.

An upper bound on the colors of GC

How many colors can GC use, whatever the list?

Proposition 1

For every graph G and every list λ,

$$GC(I) \leq \Delta(G) + 1.$$

Proof

When vertex v is colored, the set U of the colors of its colored neighbours has at most

$$|U| \leq \deg(v) \leq \Delta(G)$$

elements, so one of the colors $1, 2, \ldots, \Delta(G) + 1$ is not in U, and the smallest free color is at most $\Delta(G) + 1$. □

What the bound says (1/2)

- The bound holds for *DSATUR* too, which is *GC* with a particular list.
- It is attained by complete graphs ($\Delta = n - 1$, $\chi = n$) and odd cycles ($\Delta = 2$, $\chi = 3$), whatever the list.
- It is *not* a performance guarantee: it compares $GC(I)$ with $\Delta(G) + 1$, not with $\chi(G)$, and $\Delta(G) + 1$ can be much larger than $\chi(G)$.
- The next proposition shows that a bad list can really force that many colors on a graph that needs only two.

What the bound says (2/2)

- On the other hand, **some** list is always optimal.
- If $Q_1, \ldots, Q_{\chi(G)}$ are the color classes of an optimal coloring, GC with the vertices of Q_1 first, then those of Q_2, and so on, gives each vertex of Q_c a color at most c, so it uses at most $\chi(G)$ colors.
- The difficulty is entirely in finding the list.

A bad list

A bad list can force many colors on a graph that needs only two:

Proposition 2

For every $k \geq 3$ there is a graph G_k with $n = 2k$ vertices, $\chi(G_k) = 2$ and $\Delta(G_k) = k - 1$, and a list λ, such that

$$GC(I_k) = k = \frac{n}{2}.$$

Proof (1/2)

- G_k is the **crown graph**: the vertices are $u_1, \ldots, u_k$ and $v_1, \ldots, v_k$, and u_i is adjacent to v_j if and only if $i \neq j$ (the complete bipartite graph $K_{k,k}$ without a perfect matching).
- It is bipartite, with an edge, so $\chi(G_k) = 2$; every vertex has degree $k - 1$.
- The list is

$$\lambda = (u_1, v_1, u_2, v_2, \ldots, u_k, v_k).$$

Proof (2/2)

We show by induction on i that GC gives color i to both u_i and v_i.

- For $i = 1$ no neighbour is colored when u_1 is examined, and v_1 is not adjacent to u_1: both get color 1.
- When u_i is examined, its colored neighbours are $v_1, \ldots, v_{i-1}$, with colors $1, \ldots, i-1$ by induction, so u_i gets color i.
- When v_i is examined, its colored neighbours are $u_1, \ldots, u_{i-1}$ (u_i is not a neighbour), with colors $1, \ldots, i-1$, so v_i gets color i too.

Hence $GC(I_k) = k$. □

No constant guarantee for *GC*

- On the crown graphs

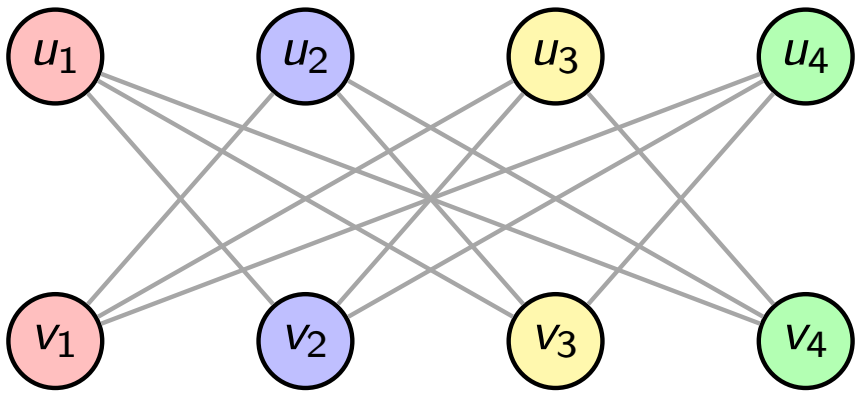
$$\frac{GC(I_k)}{\chi(G_k)} = \frac{k}{2} = \frac{n}{4},$$

  which grows without bound: with an arbitrary list, *GC* has no constant performance guarantee.
- The list $(u_1, \ldots, u_k, v_1, \ldots, v_k)$, the color classes first, gives 2 colors.

- The crown graphs satisfy the assumptions of the VCP for $k \geq 3$: no isolated vertices, and $N(u_i) = \{v_j : j \neq i\}$ is not contained in $N(u_{i'})$ nor in any $N(v_j) \subseteq \{u_1, \ldots, u_k\}$.

Example: *GC* and *DSATUR* on the crown graph G_4 (1/2)

Number the vertices $u_i = 2i - 1$ and $v_i = 2i$, so that $u_1, v_1, \ldots, u_4, v_4$ are $1, 2, \ldots, 8$, and $E = \{\{2i - 1, 2j\} : i \neq j\}$, with 12 edges.



With the list $(1, 2, \ldots, 8) = (u_1, v_1, \ldots, u_4, v_4)$, *GC* uses 4 colors, one for each pair $\{u_i, v_i\}$ (shown in the figure); with the list $(1, 3, 5, 7, 2, 4, 6, 8)$ it uses 2 colors.

Example: *GC* and *DSATUR* on the crown graph G_4 (2/2)

- *DSATUR* chooses vertex 1 (u_1, color 1), then the vertices 4, 5, 2, 3, 6, 7, 8, with colors $2, 1, 2, 1, 2, 1, 2$.
- The vertices u_i get color 1 and the vertices v_i color 2, and *DSATUR* uses $2 = \chi(G_4)$ colors.
- After the first step, every vertex adjacent to a colored vertex has saturation degree 1, and the vertex chosen is always on the side opposite to the last one colored.

DSATUR as a heuristic

- On bipartite graphs *DSATUR* always uses at most 2 colors, as on the crown graphs; this is proved in the paper that introduced it (Brélaz, 1979).
- On general graphs *DSATUR* has no performance guarantee, since the VCP is not approximable.
- It is nevertheless one of the most used constructive heuristics for the VCP, because on most graphs it uses few colors more than $\chi(G)$, often none.
- It is also the base of exact branch-and-bound algorithms for the VCP, which branch on the vertex chosen by the *DSATUR* rule.

Summary

Greedy coloring:

- each vertex of the list gets the smallest color not used by its colored neighbours
- always feasible, $O(n + |E|)$ time; $GC(I) \leq \Delta(G) + 1$ for every list
- a bad list can force $\frac{n}{2}$ colors on a bipartite graph (crown graphs); some list is always optimal

DSATUR:

- colors first the vertex with the most different colors around it: the most constrained first
- $3 = \chi(G)$ colors on the Petersen graph, 2 colors on bipartite graphs
- no performance guarantee, but one of the most used heuristics for the VCP

Randomized construction

Introduction: Randomized construction

- In this section we obtain many different, and still good, solutions from one constructive heuristic, by randomizing its choices
- We define the restricted candidate list (RCL) and a reproducible generator of random numbers
- We run the randomized nearest neighbour four times on the TSP reference instance

Randomize the choice

- A greedy heuristic is deterministic: run twice on the same instance, it returns the same solution.
- The multi-start *NN* obtains different solutions by changing the starting city, but the number of starts is limited to n.
- A more flexible way to obtain many different, and still good, solutions is to **randomize the choice**: at each step, instead of the best candidate, choose at random one of the *good* candidates.

The restricted candidate list

- The good candidates form the **restricted candidate list** (RCL).
- For a minimization criterion with values $g(q)$ of the candidates q, let $g^{\min}$ and $g^{\max}$ be the smallest and the largest value; the *value-based* RCL with parameter $\alpha \in [0, 1]$ is

$$RCL = \left\{ q : g(q) \leq g^{\min} + \alpha \left(g^{\max} - g^{\min} \right) \right\}.$$

- With $\alpha = 0$ the RCL contains only the best candidates, and the construction is greedy (with random tie-breaking).
- With $\alpha = 1$ it contains all the candidates, and the construction is purely random.

Cardinality-based RCL and GRASP

- An alternative is the *cardinality-based* RCL, made of the k best candidates for a fixed k.
- Randomized construction is the first ingredient of the metaheuristic GRASP (T.A. Feo, M.G.C. Resende, *Greedy randomized adaptive search procedures*, Journal of Global Optimization 6 (1995) 109–133).

Random numbers

- To make examples and exercises reproducible, the notes on heuristic algorithms use the *minimal standard* generator of Park and Miller, which produces a sequence of numbers $u_1, u_2, \ldots$ in $(0, 1)$ from an integer **seed**:

$$seed \leftarrow (16807 \cdot seed) \bmod 2147483647, \qquad u = \frac{seed}{2147483647}.$$

- The modulus is $2^{31} - 1$, a prime number; with 64-bit integers the product never overflows.
- The same seed gives the same sequence on every computer and in every programming language.

Drawing a candidate from the RCL

- A candidate is drawn from an RCL with $|RCL|$ elements, listed by increasing index, by taking the element in position

$$\lfloor u \cdot |RCL| \rfloor + 1.$$

- A random number is used only when $|RCL| \geq 2$.

Randomized nearest neighbour

- The *randomized nearest neighbour algorithm* (denoted by RNN_α) is a constructive heuristic for the TSP.
- Starting from city s, at each step it builds the RCL of the unvisited cities whose distance from the last city is at most

$$d^{\min} + \alpha\,(d^{\max} - d^{\min}),$$

and appends a city drawn uniformly at random from the RCL.

Algorithm: Randomized Nearest Neighbour (1/2)

Input : TSP instance with n cities, distances $d[1..n][1..n]$; starting city s; $\alpha \in [0,1]$; *seed*

Output : Tour $\sigma[1..n]$

```
1 function RandomizedNN(n, d, s, α, seed)
2 |   σ[1] ← s, visited := {s}
3 |   for len := 2 to n do
4 |   |   last := σ[len − 1]
5 |   |   d^min := min_{j∉visited} d[last][j],
  |   |     d^max := max_{j∉visited} d[last][j]
6 |   |   RCL := { j ∉ visited : d[last][j] ≤
  |   |     d^min + α (d^max − d^min) }          // by index
```

Algorithm: Randomized Nearest Neighbour (2/2)

function *RandomizedNN*$(n, d, s, \alpha, seed)$ (continued)

```
    for len := 2 to n do (continued)
7       if |RCL| ≥ 2 then
8           u := ParkMiller(seed)     // random number
              in (0, 1)
9           k := the element in position ⌊u · |RCL|⌋ + 1
              of RCL
10      else
11          k := the only element of RCL
12      end
13      σ[len] ← k, visited ← visited ∪ {k}
14  end
15  return σ
16 end
```

Quality and variety

- Repeating the construction with the same generator — the seed is *not* reset between two runs — gives different tours; the best one is kept.
- The parameter α balances quality and variety:
  - a small α gives tours close to the *NN* tour;
  - a large α gives very different but usually poor tours.

Example: $RNN_{0.3}$ on the TSP reference instance (1/5)

We run RNN_α four times from city 1, with $\alpha = 0.3$ and initial seed 1. The generator gives

u_1	u_2	u_3	u_4	u_5
0.0000	0.1315	0.7556	0.4587	0.5328

u_6	u_7	u_8	u_9
0.2190	0.0470	0.6789	0.6793

($u_1 = \frac{16807}{2147483647} \approx 0.0000078$). From city 1 the distances to $2, 3, 4, 5$ are $5, 2, 2, 9$, so the threshold is $2 + 0.3 \cdot (9 - 2) = 4.1$ and $RCL = \{3, 4\}$.

Example: $RNN_{0.3}$ on the TSP reference instance (2/5)

Run 1.

- u_1: position $\lfloor 0.0000 \cdot 2 \rfloor + 1 = 1$, city 3.
- From 3 the distances to $2, 4, 5$ are $4, 4, 7$, threshold $4 + 0.3 \cdot 3 = 4.9$, $RCL = \{2, 4\}$; u_2: position 1, city 2.
- From 2 the distances to $4, 5$ are $3, 4$, threshold 3.3, $RCL = \{4\}$: city 4, then city 5.

Tour $(1, 3, 2, 4, 5)$, $L = 25$: the NN tour.

Example: $RNN_{0.3}$ on the TSP reference instance (3/5)

Run 2.

- u_3: position $\lfloor 0.7556 \cdot 2 \rfloor + 1 = 2$, city 4.
- From 4 the distances to $2, 3, 5$ are $3, 4, 7$, threshold $3 + 0.3 \cdot 4 = 4.2$, $RCL = \{2, 3\}$; u_4: position 1, city 2.
- From 2 the distances to $3, 5$ are $4, 4$: $d^{\min} = d^{\max} = 4$ and $RCL = \{3, 5\}$; u_5: position $\lfloor 0.5328 \cdot 2 \rfloor + 1 = 2$, city 5; then city 3.

Tour $(1, 4, 2, 5, 3)$, $L = 18 = z(I)$.

Example: $RNN_{0.3}$ on the TSP reference instance (4/5)

Run 3. u_6: city 3; u_7: city 2; then cities 4 and 5. Tour $(1, 3, 2, 4, 5)$, $L = 25$.

Run 4.

- u_8: city 4; $RCL = \{2, 3\}$ and u_9: position 2, city 3.
- From 3 the distances to $2, 5$ are $4, 7$, threshold 4.9, $RCL = \{2\}$: city 2, then city 5.

Tour $(1, 4, 3, 2, 5)$, $L = 2 + 4 + 4 + 4 + 9 = 23$.

Example: $RNN_{0.3}$ on the TSP reference instance (5/5)

- The four runs give $25, 18, 25, 23$: the second one finds the optimal tour, which NN from city 1 misses.
- The randomization has turned a single deterministic answer into a small search.

Summary

Randomized construction:

- at each step, a candidate drawn at random from the restricted candidate list (RCL)
- value-based RCL: $g(q) \leq g^{\min} + \alpha\,(g^{\max} - g^{\min})$; $\alpha = 0$ greedy, $\alpha = 1$ purely random
- the first ingredient of GRASP

Reproducible random numbers:

- the Park–Miller generator: the same seed gives the same sequence everywhere
- $RNN_{0.3}$ on the reference instance: $25, 18, 25, 23$, and the optimum 18 is found

Look-ahead and regret

Introduction: Look-ahead and regret

- In this section we reduce the myopia of the greedy criterion, still taking every choice once and for all
- Regret insertion serves first the city that would lose most if postponed
- The pilot method evaluates each candidate by completing the solution with a fast heuristic

Two different questions

- A greedy criterion asks: *which choice is best now?* Its myopia can be reduced by asking two different questions:
  - **regret**: *what do I lose if I postpone this choice?* An element that has a single good option should be served before it loses it; an element with many equivalent options can wait;
  - **look-ahead**: *where does this choice lead?* Each candidate is evaluated by completing the partial solution with a fast heuristic, and the candidate with the best completed solution is chosen.
- Both ideas cost more time per step than the plain greedy criterion, and both keep the essential feature of a constructive heuristic: the choice made is final.

The regret of a city

- For an insertion heuristic, let $\Delta_q^1 \leq \Delta_q^2$ be the two smallest insertion costs of city q over the edges of the current subtour (with $\Delta_q^2 = \Delta_q^1$ if the subtour has a single edge).

- The **regret** of q is

$$reg_q \;=\; \Delta_q^2 - \Delta_q^1 \;\geq\; 0 :$$

the extra cost paid for q if its best edge is used by another city before q is inserted.

Regret-2 insertion

- The *regret-2 insertion algorithm* (denoted by REG_2) is a constructive heuristic for the TSP.
- Starting from the subtour (s), at each step it inserts the city with the *largest regret* into its edge of minimum insertion cost.
- In case of ties, it inserts the city with the smallest Δ_q^1, and then the one with the smallest index.

Algorithm: Regret-2 Insertion (1/2)

Input : TSP instance with n cities, distances $d[1..n][1..n]$; starting city s

Output: Tour $\sigma[1..n]$

```
1 function Regret2Insertion(n, d, s)
2 |   σ := (s)
3 |   for len := 1 to n − 1 do
4 |   |   foreach city q ∉ σ do
5 |   |   |   Δ¹_q, Δ²_q := the two smallest values of
  |   |   |    Δ(σ[a], q, σ[a + 1]), a = 1, . . . , len
6 |   |   |   reg_q := Δ²_q − Δ¹_q            // regret of q
7 |   |   end
```

Algorithm: Regret-2 Insertion (2/2)

function *Regret2Insertion*(n, d, s) (continued)

    for $len := 1$ **to** $n - 1$ **do** (continued)

8 $k :=$ the city with maximum reg_q (ties: minimum Δ_q^1, then smallest index)

9 insert k into its edge of minimum insertion cost

10 **end**

11 **return** σ

12 **end**

Regret-k heuristics

- Regret-k heuristics generalize the idea, summing the differences between the best option and the $k-1$ next ones.
- They are widely used for vehicle routing problems, where a customer may have a single feasible route left (J.-Y. Potvin, J.-M. Rousseau, *A parallel route building algorithm for the vehicle routing and scheduling problem with time windows*, European Journal of Operational Research 66 (1993) 331–340).
- The same idea applies to any assignment-like construction, such as items to bins.
- As long as all the regrets are 0, the tie-breaking rule makes REG_2 behave like CI.

Example: REG_2 against CI on the 6-city instance (1/4)

On the 6-city instance of the previous section, from city 1:

- in the first two steps all the regrets are 0 (the subtours (1) and $(1, 6)$ have one edge, or two copies of the same edge);
- both REG_2 and CI insert city 6 (cost 4) and then city 3 (cost 5), obtaining $(1, 3, 6)$ of length 9.

At the third step the insertion costs on the edges $(1, 3)$, $(3, 6)$, $(6, 1)$ are

city q	costs	Δ_q^1	Δ_q^2	reg_q
2	$(2, 2, 7)$	2	2	0
4	$(4, 7, 8)$	4	7	3
5	$(6, 5, 6)$	5	6	1

Example: REG_2 against CI on the 6-city instance (2/4)

- CI inserts city 2, the cheapest, on the edge $(1, 3)$.
- REG_2 inserts city 4, the one with the largest regret, on the edge $(1, 3)$.
- This edge is the only good place for city 4, while city 2 has two equally good places and can wait.

Example: REG_2 against CI on the 6-city instance (3/4)

The two constructions then continue:

	CI	REG_2
step 3	2 on $(1,3)$, cost 2: $(1,2,3,6)$	4 on $(1,3)$, cost 4: $(1,4,3,6)$
step 4	4 on $(1,2)$, cost 4: $(1,4,2,3,6)$	5 on $(3,6)$, cost 5: $(1,4,3,5,6)$
step 5	5 on $(3,6)$, cost 5: $(1,4,2,3,5,6)$	2 on $(3,5)$, cost 1: $(1,4,3,2,5,6)$
length	$4+5+2+4+5=20$	$4+5+4+5+1=19$

Example: REG_2 against CI on the 6-city instance (4/4)

- At step 4 of REG_2 the costs of city 2 on the edges of $(1, 4, 3, 6)$ are $(6, 2, 2, 7)$, regret 0, and those of city 5 are $(10, 10, 5, 6)$, regret 1: city 5 goes first.
- City 2, postponed twice, is finally inserted for 1 instead of 2, between cities 3 and 5.
- $REG_2(I) = 19 = z(I)$, while $CI(I) = 20$.

The pilot method

- The *pilot method* turns a constructive heuristic $\mathscr{H}$, the *pilot*, into a stronger one.
- At each step it tries every candidate: the partial solution extended with the candidate is completed by $\mathscr{H}$, and the candidate whose completed solution is best is added for good.
- With *NN* as pilot (denoted by *PILOT-NN*), the candidates are the next cities of the tour.

Algorithm: Pilot method with nearest neighbour

Input : TSP instance with n cities, distances $d[1..n][1..n]$; starting city s

Output : Tour $\sigma[1..n]$

```
1  function PilotNN(n, d, s)
2      σ := (s)
3      for len := 2 to n do
4          foreach city q ∉ σ do
5              τ_q := the path σ extended with q and completed
                by NN                                  // look ahead
6          end
7          k := the city q with minimum L(τ_q) (smallest index)
8          append k to σ                    // the choice is final
9      end
10     return σ
11 end
```

The cost of looking ahead

- The pilot method is due to C. Duin and S. Voß (*The pilot method: a strategy for heuristic repetition with application to the Steiner problem in graphs*, Networks 34 (1999) 181–191).
- Each step runs the pilot once per candidate: with *NN* as pilot the running time is $O(n^4)$ instead of $O(n^2)$.

The pilot method is never worse than its pilot

- At the first step, the candidate chosen by NN is among those tried, and its completion is the NN tour from s, so the best completion found has length at most $NN(I)$.
- At every later step, the completion of the chosen candidate is again among the completions tried at the next step (the candidate that NN would append), so the length of the best completion never increases.
- The final tour is the last completion, hence

$$PILOT\text{-}NN(I) \leq NN(I)$$

from the same starting city.

Example: *PILOT-NN* on the TSP reference instance (1/2)

From city 1, the candidates and their *NN* completions are:

partial path	candidates: completed tour and length	chosen
(1)	2: $(1,2,4,3,5)$ 28; 3: $(1,3,2,4,5)$ 25; 4: $(1,4,2,3,5)$ 25; 5: $(1,5,2,4,3)$ 22	5
$(1,5)$	2: $(1,5,2,4,3)$ 22; 3: $(1,5,3,2,4)$ 25; 4: $(1,5,4,2,3)$ 25	2
$(1,5,2)$	3: $(1,5,2,3,4)$ 23; 4: $(1,5,2,4,3)$ 22	4
$(1,5,2,4)$	3: $(1,5,2,4,3)$ 22	3

Example: *PILOT-NN* on the TSP reference instance (2/2)

- *PILOT-NN* goes first to city 5, the *farthest* from city 1, something *NN* would never do.
- It returns $(1, 5, 2, 4, 3)$ with $L = 9 + 4 + 3 + 4 + 2 = 22$, against $NN(I) = 25$ from the same city.
- The optimum 18 is not reached: the look-ahead is only as good as the pilot that performs it.
- On the 6-city instance of the previous section, *PILOT-NN* from city 1 gives 20, against 25 for *NN*.

Summary

Regret:

- $reg_q = \Delta_q^2 - \Delta_q^1$: what is lost if the insertion of q is postponed
- REG_2 inserts first the city with the largest regret; on the 6-city instance $REG_2(I) = 19 = z(I)$, $CI(I) = 20$

Look-ahead:

- the pilot method completes each candidate with a fast heuristic and keeps the best one
- *PILOT-NN* is never worse than *NN*: 22 against 25 on the reference instance
- both ideas cost more time per step, and the choices remain final

What constructive heuristics give, and what they do not

Introduction: What constructive heuristics give, and what they do not

- In this section we collect the results on the four reference instances and compare them with the dual bounds
- We draw the lessons of the chapter: a primal bound, final choices, the order of the choices
- We see why constructive heuristics are rarely used alone

Results on the reference instances

The table collects the results on the four reference instances, with the dual bounds of the chapter «Heuristic algorithms: definitions» (UB for the KP, LB for the other problems) and the gap of the best heuristic solution.

problem	heuristic	values	dual bound	gap	$z(I)$
KP	GR_p, GR, GR_w	17, 16, 10	$\mathrm{UB} = 17$	0%	17
BPP	FF, FFD, BFD	4, 3, 3	$\mathrm{LB} = 3$	0%	3
$P \mid\mid C_{\max}$	*LIST* (reverse list), *LPT*	12, 11	$\mathrm{LB} = 9$	22.2%	9
TSP	*NN*, multi-start *NN*, *NI*, *FI*, *CI*	25, 18, 18, 18, 18	$\mathrm{LB} = 17$	5.9%	18

A primal bound, and nothing more

- A constructive heuristic returns a feasible solution, and its value is a primal bound.
- For the KP and the BPP the best heuristic value meets the dual bound, and the solution is *proved* optimal.
- For the TSP the tour of length 18 is optimal, but the dual bound 17 cannot prove it.
- For $P \,||\, C_{\max}$ the gap of 22.2% is real.
- The heuristic itself never knows in which of these situations it is.

A choice is never revised

- Every heuristic of this chapter stops as soon as the solution is complete, and every decision taken along the way is final.
- In the *LPT* schedule, job 7 is put on machine 1 when the three loads are equal, and the makespan 11 is fixed at that moment.
- In the *NN* tour from city 1, the edge $(5, 1)$ of length 9 is the consequence of the very first choices.
- Looking at the Gantt chart or at the tour, a better solution seems close; but a constructive heuristic has no mechanism to go back.

The order decides everything

- The examples of this chapter show the same lesson on four problems: sorting the elements in the right order (efficiency, decreasing weights, longest jobs first) is the main design choice.
- Still, no order is the best on every instance (the three orders on the KP reference instance, and the example in which sorting hurts FFD).
- Running several orders, several starting points or several randomized constructions, and keeping the best solution, is a cheap and effective protection.

Cheap, and useful as a start

- Constructive heuristics are fast, and they are rarely used alone.
- Their solutions are the starting points of the methods that try to improve a complete solution.

A constructive heuristic builds one solution, one element at a time, choosing at each step the element that looks best now.

The order of the choices decides the quality of the result, and a choice, once made, is never revised.

Summary

What they give:

- a feasible solution, i.e., a primal bound: optimality is proved only if a dual bound meets it
- fast solutions, used as starting points of the improvement methods

What they do not give:

- a way back: every choice is final
- an order that is the best on every instance: several orders, starts or randomized constructions protect against it

Typical mistakes

Introduction: Typical mistakes

- We collect the mistakes that are made most often with constructive heuristics
- Two concern the orders, two the execution by hand, one the meaning of the value
- Check them before running a heuristic by hand

Typical mistakes (1/2)

Observation 1: Expecting the greedy by efficiency to be optimal

It can be arbitrarily bad, and so can each of the other two orders: the chapter contains a family of instances for each of them.

Observation 2: Believing that sorting always helps

FFD can be worse than FF (the example in which sorting hurts).

Typical mistakes (2/2)

Observation 3: Breaking ties in the wrong way

Every example uses one fixed rule (smallest index, first bin that fits, fullest bin for BF): changing it changes the solution.

Observation 4: Forgetting the closing edge

The length of a tour includes the edge from the last city back to the first one; the cost of an insertion is $d_{ik} + d_{kj} - d_{ij}$, and every position, closing edge included, must be tried.

Observation 5: Reading $\mathscr{A}(I)$ as a bound on $z(I)$

It is only a primal bound.

Summary

The orders:

- no order of the KP items is always optimal; sorting does not always help

The execution by hand:

- break the ties with the fixed rule of the heuristic
- count the closing edge, and try every position of an insertion

The value:

- $\mathscr{A}(I)$ is only a primal bound

C++ implementation

Introduction: C++ implementation

- In this section we show a few short excerpts of the C++ implementation of the heuristics of the chapter
- The complete code is in the folder of the chapter and in the notebooks
- The code follows the pseudo-codes step by step, with the same names

The code of the chapter

- The folder `CONSTR_HEUR_CODE` contains a C++ implementation of the heuristics of the chapter, one function per file, with the same names and the same steps as the pseudo-codes.
- Items, bins, jobs, machines and cities are indexed from 0; the algorithmic logic is unchanged, and ties are always broken in favour of the smallest index.
- The command `make` builds the program, and `make test` runs every example of the notes and every instance of the exercises, comparing each value and each solution with those computed by an independent implementation in Python: 202 checks, 0 errors.

The header (excerpt)

The header `Constr_heuristics.h` declares all the functions; for the TSP:

```
1  // TSP: d is the n x n distance matrix;
2  // every function returns the length of the tour
3  int TourLength(int n, int** d, int* tour);
4  int NearestNeighbour(int n, int** d, int s,
5                       int* tour);
6  int MultiStartNN(int n, int** d, int* tour);
7  int Insertion(int n, int** d, int s, int rule,
8                int* tour);  // 0 NI, 1 FI, 2 CI
9  int Regret2Insertion(int n, int** d, int s,
10                      int* tour);
11 int PilotNN(int n, int** d, int s, int* tour);
```

Sorting (excerpt)

- The orders of the KP items and the decreasing order used by FFD, BFD and *LPT* are computed by insertion sort, which keeps the index order among equal keys.
- The efficiencies are compared without divisions: $\frac{p_j}{w_j} > \frac{p_k}{w_k}$ if and only if $p_j w_k > p_k w_j$.

```
1 static bool Before(int i, int j, int* p, int* w,
2                    int criterion)
3 {
4     if (criterion == 0) return p[i] > p[j];
5     if (criterion == 1) return w[i] < w[j];
6     // p_i/w_i > p_j/w_j without divisions
7     return p[i] * w[j] > p[j] * w[i];
8 }
```

Nearest neighbour (excerpt)

The main loop of `NearestNeighbour`:

```
for (int len = 1; len < n; len++)
{
    int last = tour[len - 1];  // last city
    int k = -1;                // nearest unvisited city
    for (int j = 0; j < n; j++)
    {
        if (!visited[j] &&
            (k == -1 || d[last][j] < d[last][k]))
            k = j;
    }
    tour[len] = k;             // append k at the end
    visited[k] = true;
}
```

Insertion (excerpt)

- In `Insertion` the three selection rules share the same loop: the value of a candidate is its distance from the subtour (NI), the same distance with the sign changed, so that the minimum corresponds to the farthest city (FI), or its minimum insertion cost (CI).

```
1 key = d[q][tour[0]];    // distance from the subtour
2 for (int a = 1; a < len; a++)
3 {
4     if (d[q][tour[a]] < key) key = d[q][tour[a]];
5 }
6 if (rule == 1) key = -key;  // farthest: maximize
```

Randomized construction (excerpt)

- The threshold of the RCL is tested in integer arithmetic, with α given as a fraction $\frac{\alpha_{num}}{\alpha_{den}}$, so that no rounding error can change the list.

```
1 // d <= dmin + alpha (dmax - dmin),
2 // in integer arithmetic
3 if (!visited[j] && alpha_den * d[last][j]
4         <= alpha_den * dmin
5             + alpha_num * (dmax - dmin))
6 {
7     RCL[size] = j;
8     size++;
9 }
```

Summary

The code:

- one `C++` function per heuristic, with the names and the steps of the pseudo-codes
- indices from 0, ties broken in favour of the smallest index
- `make test`: 202 checks against an independent implementation in Python, 0 errors

Implementation details:

- insertion sort keeps the index order among equal keys; efficiencies compared without divisions
- one loop for *NI*, *FI* and *CI*; the RCL threshold in integer arithmetic

In summary

Introduction: In summary

- We first collect the key ideas of this second part, then those of the whole chapter on constructive heuristics
- The dashboard gathers the best constructive values on the four reference instances, with their dual bounds

Summary of the second part (1/2)

The travelling salesman problem:

- append: *NN* and its multi-start version; insert: *NI*, *FI*, *CI*, with the insertion cost $\Delta(i, k, j)$
- on the 6-city instance *FI* finds the optimum 19, multi-start *NN* only 24

Vertex coloring:

- *GC* follows a list, $GC(I) \leq \Delta(G) + 1$, but a bad list can be arbitrarily bad (crown graphs)
- *DSATUR* colors first the most constrained vertex

Summary of the second part (2/2)

Randomized construction:

- a random choice in the restricted candidate list turns one construction into a small search

Regret and look-ahead:

- REG_2 inserts first the city that would lose most if postponed
- the pilot method completes each candidate with a fast heuristic, and is never worse than its pilot

What constructive heuristics give:

- a primal bound, fast, with final choices: the starting point of the improvement methods

Summary (1/2)

- A constructive heuristic builds one solution, one element at a time, and never revises a choice.
- The order of the choices is the main design decision: efficiency for the KP, decreasing weights for the BPP, longest jobs first for $P \,||\, C_{\max}$; no order wins on every instance.
- Several orders, starting points or randomized constructions, keeping the best solution, are a cheap protection.
- The solutions are starting points for local search.

Summary (2/2)

Dashboard of the reference instances. Best constructive value and dual bound:

problem	best heuristic value	dual bound	gap	$z(I)$
KP (max)	17 (GR_p)	$\mathrm{UB}(I) = 17$	0%	17
BPP (min)	3 (FFD, BFD)	$\mathrm{LB}(I) = 3$	0%	3
$P \mid\mid C_{\max}$ (min)	11 (LPT)	$\mathrm{LB}(I) = 9$	22.2%	9
TSP (min)	18 (multi-start NN, NI, FI, CI)	$\mathrm{LB}(I) = 17$	5.9%	18