Dynamic programming for *NP*-hard problems

Discrete Optimization

Giovanni Righini

UNIVERSITÀ DEGLI STUDI DI MILANO

D.P. for *NP*-hard problems

Many interesting and relevant discrete optimization problems are *NP*-hard: unless $P = NP$, there is no hope of designing a polynomial-time exact optimization algorithm.

It makes sense to design heuristics, but one should not dismiss exact optization.

Often, the best heuristic is a suitably modified exact optimization algorithm (matheuristics).

Strong and weak *NP*-hardness

Strongly *NP*-hard problems $\Leftrightarrow$ Exponential-time algorithms.

Weakly *NP*-hard problems $\Leftrightarrow$ Pseudo-polynomial-time algorithms.

Pseudo-polynomial complexity depends on the numeric value of some input datum (not only on the size of the instance).

Very useful for problems with a known limit on the range of some input values.

Approximation

NP-hard problems can also be approximated.

- K-approximation: $\frac{z - z^*}{z^*}$ bounded by a costant factor;
- $g(n)$-approximation: $\frac{z - z^*}{z^*}$ bounded by a function of n;
- ϵ-approximation: $\frac{z - z^*}{z^*}$ bounded by an arbitrarily small factor.

We are interested in polynomial-time approximation algorithms.

FPTAS: the computational complexity polynomially depends on $1/\epsilon$.

The binary knapsack problem

Data

- an indexed set $\mathcal{N}$ of n items,
- a value c_j for each item $j \in \mathcal{N}$,
- a weight a_j for each item $j \in \mathcal{N}$,
- a capacity b of a container (knapsack).

Find a maximum value subset of items such that their total weight does not exceed the capacity.

$$\max z = \sum_{j \in \mathcal{N}} c_j x_j$$

$$\text{s.t.} \sum_{j \in \mathcal{N}} a_j x_j \leq b$$

$$x_j \in \{0, 1\} \qquad \forall j \in \mathcal{N}.$$

Complexity

The number of possible solutions is 2^n and, even worse, the problem is *NP*-hard.

Assume that all data are integer.

The problem is weakly *NP*-hard and D.P. allows to design exact optimization algorithms with pseudo-polynomial complexity.

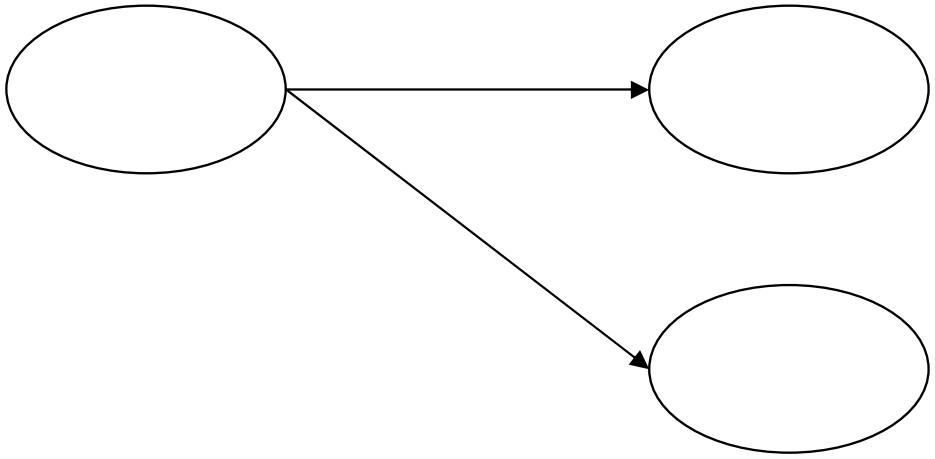
This is obtained, for instance, by enumeration of all feasible values for the capacity consumption.

A D.P. algorithm

- **Policy**: sort the items (the variables) from x_1 to x_n.
- **State**:
  - **Feasibility** depends on the residual capacity;
  - **Cost** does not depend on previous decisions.

  Hence the state is given by the last item considered (j) and the capacity used so far (u).
- **Resource Extension Function**:
  - **Initialization**: $z(0,0) = 0$;
  - **Extension**: $z(j,u) = \max\{z(j-1,u), z(j-1,u-a_j) + c_j\}$.

The state-transition graph



The state graph has a layer for each item (variable) $i \in \mathcal{N}$ and $b + 1$ nodes per layer.

Complexity: The graph has $O(nb)$ nodes and each of them has only two predecessors. Then the D.P. algorithm has complexity $O(nb)$, which is pseudo-polynomial.

Approximating the KP

The binary knapsack problem can also be solved to optimality with a D.P. algorithm based on the following recursion:

$$\begin{cases} u(j,0) = 0 \\ u(j,z) = \min\{u(j-1, z-c_j) + a_j, u(j-1,z)\} \quad \forall z = 1, \ldots, z^* \end{cases}$$

where $u(j,z)$ is the minimum capacity needed to achieve profit z using the first j elements of $\mathcal{N}$.

This algorithm takes $O(nz^*)$ time.

Bounds on z^*

Observation. Denoting the largest value of the profit vector c by $\overline{c} = \max_{j \in \mathcal{N}}\{c_j\}$, we have:

$$\overline{c} \leq z^* \leq n\overline{c}.$$

The first inequality is true under the obvious assumption that all solutions with only one item are feasible (items with $a_j > b$ can be identified and discarded at pre-processing time in $O(n)$).

The second inequality is true because

$$z^* = \sum_{j \in \mathcal{N}} c_j x_j^* \leq \sum_{j \in \mathcal{N}} c_j \leq \sum_{j \in \mathcal{N}} \overline{c} = n\overline{c}.$$

Scaling

Select a scale factor k and define modified costs $c'_j = \lfloor \frac{c_j}{k} \rfloor$.
The scaled problem is

$$KP') \text{ maximize } \quad z' = \sum_{j \in \mathcal{N}} c'_j x_j$$
$$\text{s.t. } \sum_{j \in \mathcal{N}} a_j x_j \leq b$$
$$x \in \mathcal{B}^n$$

This is still a binary knapsack problem and it can be solved to optimality with the same D.P. algorithm in $O(nz^{*'})$ time.

Define $\overline{c'} = \max_{j \in \mathcal{N}} \{c'_j\}$. Then $\overline{c'} \leq z^{'*} \leq n\overline{c'}$.

Therefore the time complexity for solving KP' is $O(n^2 \frac{\overline{c}}{k})$.

Relationship between z^* and $z^{*'}$

Let $X^{*'}$ be the set of items with $x = 1$ in the optimal solution of KP'.
Let X^* be the set of items with $x = 1$ in the optimal solution of KP.

We can now establish a relationship between $z(X^*)$ and $z(X^{*'})$.

$$z(X^{*'}) = \sum_{j \in X^{*'}} c_j \geq \sum_{j \in X^{*'}} k \left\lfloor \frac{c_j}{k} \right\rfloor \geq \sum_{j \in X^*} k \left\lfloor \frac{c_j}{k} \right\rfloor$$
$$\geq \sum_{j \in X^*} (c_j - k) = \sum_{j \in X^*} c_j - k|X^*| \geq z(X^*) - kn$$

The absolute error is bounded by kn.
The relative error is bounded by $\frac{kn}{z(X^*)}$, i.e. by $\frac{kn}{\overline{c}}$.

Relationship between z^* and $z^{*\prime}$

So, if we solve the scaled problem KP' instead of the original problem KP,

- we need $O(n^2 \frac{\overline{c}}{k})$ computing time;
- we achieve an approximation factor $\epsilon = \frac{kn}{\overline{c}}$.

Therefore the computational complexity of the approximation algorithm is

$$O\left(\frac{n^3}{\epsilon}\right).$$

This provides a fully polynomial time approximation scheme (FPTAS) for problem KP.

Label setting algorithm (part 1)

```
/* Inizialize */
for u = 0, ..., b do
    z[0, u] ← 0;
/* Compute all states */
for j = 1, ..., n do
    for u = 0, ..., a_j − 1 do
        z[j, u] ← z[j − 1, u];
        pred[j, u] ← 0;
    for u = a_j, ..., b do
        if (z[j − 1, u − a_j] + c_j > z[j − 1, u]) then
            z[j, u] ← z[j − 1, u − a_j] + c_j;
            pred[j, u] ← 1;
        else
            z[j, u] ← z[j − 1, u];
            pred[j, u] ← 0;
/* Find the optimal value */
/* Reconstruct the optimal solution */
```

Label setting algorithm (part 2)

/* Inizialize */
/* Compute all states */
/* Find the optimal value */
$z^* \leftarrow 0$;
for $u = 0, \ldots, b$ **do**
  if $(z[n, u] > z^*)$ **then**
    $z^* \leftarrow z[n, u]$;
    $u^* \leftarrow u$;
/* Reconstruct the optimal solution */
for $j = n, \ldots, 1$ **do**
  $x^*[j] \leftarrow pred[j, u^*]$;
  if $(pred[j, u^*] = 1)$ **then**
    $u^* \leftarrow u^* - a_j$;
Return(z^*, x^*)

Label correcting

In this label setting implementation many iterations are wasted, because not all entries of the matrix z are needed.

A possibly more effective implementation is based on pointers, where every row of the matrix z is implemented as a linked list.

The algorithm starts from a single state of value 0 on row 0 and only existing states in row i generate successor states in row $i + 1$.

Advantage. Sparsity of the data-structure: each linked list is likely to contain fewer states than a complete row of the matrix z, especially in the earliest iterations.

Drawback. Every time a state (j, u) is generated with value $z'(j, u)$, it is necessary to check whether another state (j, u) already exists with value $z''(j, u)$, to check for dominance.

KP: an example

i	c_i	a_i
1	45	4
2	55	5
3	42	4
4	62	6
5	61	6
6	80	8
7	69	7

Capacity: $b = 16$.

Label setting vs. label correcting

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
1	0	0	0	0	**45**	45	45	45	45	45	45	45	45	45	45	45	45
2	0	0	0	0	45	55	55	55	55	**100**	100	100	100	100	100	100	100
3	0	0	0	0	45	55	55	55	87	**100**	100	100	100	142	142	142	142
4	0	0	0	0	45	55	62	62	87	**100**	107	117	117	142	149	162	162
5	0	0	0	0	45	55	62	62	87	**100**	107	117	123	142	149	162	168
6	0	0	0	0	45	55	62	62	87	**100**	107	117	125	142	149	162	168
7	0	0	0	0	45	55	62	69	87	100	107	117	125	142	149	162	**169**

Label setting.

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
1	0				**45**												
2	0				45	55				**100**							
3	0				45	55			87	**100**				142			
4	0				45	55	62		87	**100**	107	117		142	149	162	
5	0				45	55	62		87	**100**	107	117	123	142	149	162	168
6	0				45	55	62		87	**100**	107	117	125	142	149	162	168
7	0				45	55	62	69	87	100	107	117	125	142	149	162	**169**

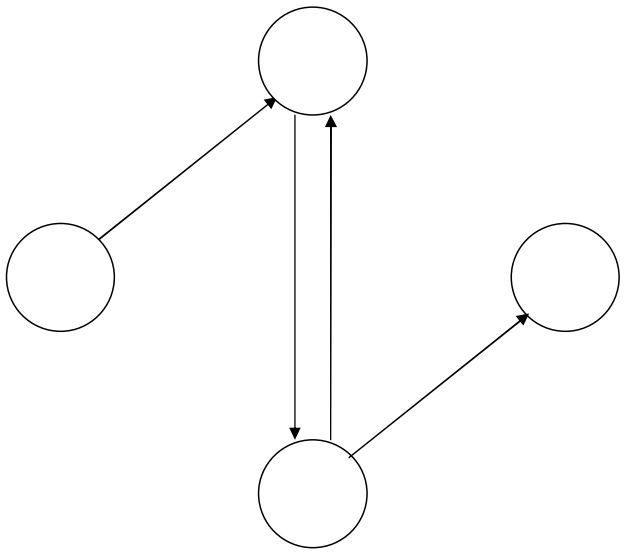
Label correcting.

The (Elementary) Resource Constrained Shortest Path Problem

Find a shortest path by car from Milan to Rome with a budget of 15 €.

It is an instance of an *NP*-hard problem!

Find the elementary shortest path from s to t in a weighted directed cyclic graph with general arc costs.



Negative cost cycles $\Rightarrow$ elementarity constraint.

RCESPP from VRP pricing

Branch-and-price is the most effective technique to solve vehicle routing problems.

A set of n customers (nodes of a graph) must be visited by a fleet of V vehicles, starting from a depot and going back to it.

Vehicle routes must typically comply with some constraints on limited resources (capacity, time, fuel,...).

Branch-and-price

A branch-and-price algorithm is a branch-and-bound algorithm, where dual bounds are obtained from the linear relaxation of an extended formulation (restricted master problem) of the VRP.

$$
\begin{aligned}
\text{minimize } z = & \sum_{k \in K} c_k \theta_k & & & \\
\text{s.t. } & \sum_{k \in K} y_{ik} \theta_k \geq 1 & & \forall i \in N & [\lambda_i] \\
& \sum_{k \in K} \theta_k \leq V & & & [\mu] \\
& \theta_k \in \{0, 1\} & & \forall k \in K &
\end{aligned}
$$

where

- K is the subset of available routes;
- θ_k indicates whether route k is selected;
- y_{ik} indicates whether route k visits node i.

Column generation

The route subset K is restricted, because the cardinality of the whole route set is exponential in n.

Column generation: additional routes are generated and inserted in the route subset, if they have negative reduced cost.
Then the master problem is re-optimized.

Reduced cost:

$$c_k - \sum_{i \in N} \lambda_i y_{ik} + \mu.$$

Pricing problem

Pricing problem: find a minimum reduced cost feasible route.

$$\text{minimize } r = \sum_{(i,j)\in A} c_{ij}x_{ij} - \sum_{i\in N} \lambda_i y_i$$

$$\text{s.t. } 0 \leq \sum_{j\in N} x_{ij} = y_i = \sum_{j\in N} x_{ji} \leq 1 \qquad \forall i \in N$$

$$y_0 = 1$$

resource constraints

$$x_{ij} \in \{0,1\} \qquad \forall (i,j) \in A.$$

Find a min cost path from the depot (0) to the depot, with

- costs c on the arcs,
- rewards λ on the nodes.

RCESPP for pricing

Rewards λ are assigned to the arcs, decreasing their cost:

$$c'_{ij} = c_{ij} - \frac{\lambda_i}{2} - \frac{\lambda_j}{2}.$$

Negative cost arcs and cycles can be produced.

The pricing problem turns out to be a RCESPP.

The most common technique to solve it is dynamic programming:

- heuristic pricing (approximation)
- exact pricing (optimization)

Example 1: Capacitated VRP

Each vehicle has a given capacity Q.
Each node $i \in N$ has a given demand d_i.

Capacity constraint:

$$\sum_{i \in N} d_i y_i \leq Q.$$

Dynamic programming uses a resource (capacity) consumption q.

Extension along (i, j) from (q', i) to (q'', j):

$$q'' = q' + d_j$$

Feasibility: $q'' \leq Q$.

Example 2: VRP with distribution and collection (VRPDC)

$\forall i \in N$, p_i and d_i are the amounts to be picked-up and delivered. Each vehicle has a positive integer capacity Q.

Two resources: the amounts of load that the vehicle can pick-up/deliver after node i.

The two resources interact: the maximum amount the vehicle can deliver after visiting i cannot be greater than the maximum amount it can pick-up after visiting i.

Initialization: $\pi = \delta = 0$.
Extension along (i, j) from (π', δ', i) to (π'', δ'', j):

$$\pi'' = \pi' + p_j \quad \delta'' = \max\{\delta' + d_j, \pi' + p_j\}.$$

Feasibility: $(\pi'' \leq Q) \wedge (\delta'' \leq Q)$ (the latter one implies the former one).

Example 3: VRP with time windows (VRPTW)

A time window $[a_i, b_i]$ is associated with each node $i \in N$.
If the vehicle arrives at i before a_i, it waits.
The travel time along (i, j) is t_{ij}.

One resource: τ_i, time elapsed up to start time of service at i.

Initialization: $\tau_i = 0$.
Extension along (i, j) from (τ', i) to (τ'', j):

$$\tau'' = \max\{\tau' + t_{ij}, a_j\}.$$

Feasibility: $\tau'' \leq b_j$.

Elementarity constraints

The graph can contain negative cost cycles but the route must be elementary (a reward λ_i cannot be counted more than once).

A suitable binary vector S of resources indicates which nodes have already been visited.

- **State:** (i, S) (visited subset, last reached node);
- **Initialization:** $c(\{s\}, s) = 0$;
- **Extension:** $c(j, S) = \min_{i \in S \setminus \{j\}} \{c(i, S \setminus \{j\}) + c_{ij}\} \quad \forall j \neq 0$.

Dominance

$L' = (i', q', S')$ dominates $L'' = (i'', q'', S'')$ only if:

- $i' = i''$
- $q' \leq q''$
- $S' \subseteq S''$
- $c(L') \leq c(L'')$ (assuming minimization).

Complexity: the n. of states grows exponentially with the size of the graph.

Continuous resources prevent from label setting.

Some resources can be renewable.

Unreachable nodes (Feillet et al., 2004)

A node j is unreachable from a state (i, q, S) when $d_{ij}^k + d_{j0}^k > Q^k - q^k$ for some resource k, where

- d_{ij} is the minimum resource consumption for reaching j from i;
- Q^k is the total amount of available resource k;
- q^k is the consumption of resource k in state (i, q, S).

. Non-visited unreachable nodes U can be considered as if they had already been visited: (i, q', S', U') dominates (i, q'', S'', U'') if

$$(q' \leq q'') \wedge (S' \cup U' \subseteq S'' \cup U'') \wedge (c(i, q', S', U') \leq c(i, q'', S'', U''))$$

Dynamic programming for *NP*-hard problems

- Exponential growth in the number of labels
- Exponential number of possible states, if the path must be elementary
- Non-enumerable states, if there are continuous resources
- Need for quick but good heuristic solutions (heuristic pricing, primal bounding)
- Need for quick but good relaxations (dual bounding)

Good algorithmic ideas are needed!

Bi-directional extension (Righini and Salani, 2006)

Bi-directional D.P.: labels are extended both forward from vertex s to its successors and backward from vertex t to its predecessors.

Backward states, extension functions and dominance rules are symmetrical.

A path from s to t is detected each time

- a forward state reaching i and a backward state reaching j can be feasibly joined through arc (i, j) (join on arcs);
- a forward and a backward state reaching i can be feasibly joined in node i (join in nodes).

Join

Let $L^{fw} = (S^{fw}, q^{fw}, i)$ be a forward path and $L^{bw} = (S^{bw}, q^{bw}, j)$ be a backward path.

Join on arcs ($i \neq j$).

Feasibility conditions:

- $S^{fw} \cap S^{bw} = \emptyset$
- $q^{fw} + q^{bw} \leq Q$

Cost: $c(L^{fw}) + c_{ij} + c(L^{bw})$.

Join on nodes ($i = j$).

Feasibility conditions:

- $S^{fw} \cap S^{bw} = \{i\}$
- $q^{fw} + q^{bw} - q_i \leq Q$

Cost: $c(L^{fw}) + c(L^{bw})$.

Example 1: CVRP

Backward state: (j, q^{bw}).

- Node j: last node reached by backward extensions from the final depot.
- Backward resource consumption q^{bw}: amount of capacity consumed by nodes between j (included) and the final depot.

Backward extension from $(j, q^{bw'})$ to $(i, q^{bw''})$ along (i, j):

$$q^{bw''} = q^{bw'} + d_i.$$

Feasible join between (i, q^{fw}) and (j, q^{bw}):

$$q^{fw} + q^{bw} \leq Q.$$

Example 2: VRPDC

Backward state: $(j, \pi^{bw}, \delta^{bw})$.

- Node j: last node reached by backward extensions from the final depot.
- Backward resource consumption:
  - δ^{bw}: amount of load delivered between j and the final depot;
  - π^{bw}: maximum total load on board between j and the final depot.

Backward extension from $(j, \pi^{bw'}, \delta^{bw'})$ to $(i, \pi^{bw''}, \delta^{bw''})$ along (i, j):

$$\delta^{bw''} = \delta^{bw'} + d_i \quad \pi^{bw''} = \max\{\delta^{bw'} + d_i, \pi^{bw'} + p_i\}.$$

Feasible join between $(i, \pi^{fw}, \delta^{fw})$ and $(j, \pi^{bw}, \delta^{bw})$:

$$(\pi^{fw} + \pi^{bw} \leq Q) \wedge (\delta^{fw} + \delta^{bw} \leq Q).$$

Example 3: VRPTW

Define $T = \max_{i \in N}\{b_i + t_{i0}\}$.

Backward state: (j, τ^{bw}).

- Node j: last node reached by backward extensions from the final depot.
- Backward resource consumption τ^{bw}: time elapsed between the departure from j and time T.

Backward extension from $(j, \tau^{bw'})$ to $(i, \tau^{bw''})$ along (i, j):

$$\tau^{bw''} = \max\{\tau^{bw'} + t_{ij}, T - b_i\}.$$

Feasible join between (i, τ^{fw}) and (j, τ^{bw}):

$$\tau^{fw} + t_{ij} + \tau^{bw} \leq T.$$

Stopping at a half-way point

Extensions in one direction are stopped, when there is the guarantee that the remaining part of the path is generated in the other direction and therefore no optimal solution is lost.

Possible stop criteria:

- stop when $n/2$ extensions have been done;
- stop when half the overall available amount of a non-renewable critical resource has been consumed:
  - CVRP: do not extend (i, q) if $q \geq Q$.
  - VRPDC: do not extend (i, π, δ) if $\pi + \delta \geq Q$.
  - VRPTW: do not extend (i, τ) if $\tau \geq T/2$.

A suitable combination of resources can be used as a critical resource.

Guessing the critical resource (Bezzi et al., 2020)

When several resources are used, a correct guess of the critical resource has a huge impact on the number of states.

In general, the ”best” critical resource is the most binding one.

Sometimes it can be guessed from the input data (supervised learning).

Sometimes it can be estimated from the states generated in the early iterations (predictive heuristics).

Supervised learning

Assumption: there are features of the instance that are predictive of the best critical resource.

- Generate many problem instances.
- For each instance create a data object, measuring and recording several features.
- Solve each instance once for every choice of critical resource and record the CPU time.
- Label each data object with the corresponding best critical resource for that instance.
- Use the labelled data objects to train classification models, mapping pricing instance features to critical resource labels.

The classification model can then be invoked to predict the best critical resource for any new instance.

Heuristics: partial inspection

Assumption: the number of labels produced by extensions is a good proxy for the overall CPU time required by the D.P. algorithm.

- The extension of a label is stopped when half of *any* resource has been consumed.
- Extensions are stopped when a certain number L of labels have been generated.
- When all extensions have been stopped, the best critical resource is guessed to be the one stopping the largest number of extensions.

Heuristics: partial inspection

for each resource r **do**
  $\overline{K}_r \leftarrow 0$
$t \leftarrow 0$
Queue $\leftarrow \{$initial state$\}$
while $t < \mathrm{L}$ **do**
  $\mathcal{L} \leftarrow$ Queue$[t]$
  if $\mathcal{L}$ is an open label **then**
    for Each node i reachable from $\mathcal{L}$ **do**
      Try to extend $\mathcal{L}$ to i creating $\mathcal{M}$
      for each resource r **do**
        if $\mathcal{M}$ blocked by resource r **then**
          $\overline{K}_r \leftarrow \overline{K}_r + 1$
      if $\mathcal{M}$ is not blocked **then**
        Add $\mathcal{M}$ to Queue
        Check dominance
  $t \leftarrow t + 1$
Return argmax$_r\{\overline{K}_r\}$

Solution uniqueness (Righini and Salani, 2006)

Goal: avoid generating the same path more than once.

Accept a path only when it is produced by the join of a forward state and a backward state for which the forward and backward consumptions of the critical resource are as close as possible to half the overall consumption along the path.

Let ρ^{fw} and ρ^{bw} be the critical resource consumptions in forward and backward paths.

We require $|\rho^{fw} ? \rho^{bw}|$ to be $\leq$ to its value in the next and the previous position along the path.

The test can be done in constant time.

Bi-directional D.P. for the KP

Forward states with items $1, \ldots, 3$.
Backward states with items $4, \ldots, 7$.

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
1	0				45												
2	0				45	55				100							
3	0				45	55			87	100				142			
4	0						62	69	80				123	131	142	149	
5	0						61	69	80					130	141	149	
6	0							69	80							149	
7	0							69									

The two sparse sub-matrices in the bi-directional implementation: about 75% of the states are not generated.

Each forward state with capacity consumption q is joined with the backward state with maximum consumption not larger than $b - q$.

For instance, $q^{fw} = 5$ and $z^{fw} = 55$ is matched with $q^{bw} = 8$ and $z^{bw} = 80$.

Join complexity: $O(n)$.

Bounding for fathoming states

Bounding is used as in branch-and-bound algorithms, to detect states that are not worth extending, because they cannot lead to optimal solutions.

Given a state $L = (S, q, i)$, compute a lower bound $LB(L)$ (completion bound) on the cost for completing the path with the residual amount $Q - q$ of resources.

$$\text{minimize } LB = \sum_{j \in \mathcal{N} \setminus \mathcal{S}} b_j y_j$$

$$\text{subject to } \sum_{j \in \mathcal{N} \setminus \mathcal{S}} d_j y_j \leq Q - q$$

$$y_j \in \{0, 1\} \qquad \forall j \in \mathcal{N} \setminus \mathcal{S}$$

where

- b_j is a lower bound on the (possibly ≤ 0) cost of visiting $j \notin S$;
- d_j is the minimum resource consumption for visiting j.

If $c(L) + LB \geq UB$, then fathom L (UB is an upper bound).

State space relaxation (Christofides et al., 1981)

The state space $\mathcal{S}$ explored by the D.P. algorithm is projected onto a lower dimensional space $\mathcal{T}$, so that each state in $\mathcal{T}$ retains the minimum cost among those of its corresponding states in $\mathcal{S}$ (assuming minimization).

$SSR : \mathcal{S} \mapsto \mathcal{T}$ such that $c(t) = \min_{s \in \mathcal{S}: SSR(s)=t}\{c(s)\}$.

In this way:

- the number of states to be explored is drastically reduced;
- some infeasible states s in $\mathcal{S}$ can be projected onto a state t corresponding to a feasible solution in $\mathcal{T}$.

The D.P. algorithm exploring $\mathcal{T}$ instead of $\mathcal{S}$ is faster and it does not guarantee to find an optimal solution, but rather a dual bound.

SSR of the set of visited nodes

We map each state (S, q, i) onto a new state (σ, q, i), where $\sigma = |S|$ represents the number of nodes visited (excluding s).

Dominance condition $S' \subseteq S''$ is replaced by $\sigma' \leq \sigma''$.

Since $\sigma \leq N$ the D.P. has pseudo-polynomial time complexity.

Since the state does no longer keep information about the set of already visited vertices, cycles are no longer forbidden; the solution

- is guaranteed to be feasible with respect to the resource constraints;
- is not guaranteed to be elementary.

This technique can also be applied to bi-directional search.

Decremental state space relaxation (Righini and Salani, 2006)

Decremental SSR allows to tune a trade-off between D.P. with SSR (using σ) and exact D.P. (using S):

- a set $\mathcal{V} \subseteq \mathcal{N}$ of critical nodes is defined;
- S is now a subset of $\mathcal{V}$;
- σ counts the overall number of visited nodes.
- For $\mathcal{V} = \mathcal{N}$, DSSR is equivalent to exact D.P..
- For $\mathcal{V} = \emptyset$, DSSR is equivalent to D.P. with SSR.

The algorithm is executed multiple times and after each iteration some nodes visited more than once are inserted into $\mathcal{V}$.

The algorithm ends when the optimal solution is also feasible (the path is elementary). An increasingly better lower bound is produced at each iteration.

Computational experiments show that in many cases a critical set containing about 15% of the nodes is enough (!).

Critical set

Policies to update V (Boland et al., 2006):

- insert one node among those visited the maximum number of times;
- insert all nodes visited the maximum number of times;
- insert all nodes visited more than once.

Policies to initialize V, based on the "cycling attractiveness" (CA) of a node (Righini and Salani, 2009):

$$f_{ij} = \lambda_i/(t_{ij} + t_{ji})$$

Sort the nodes by

- Highest CA: $\max_{j \in \mathcal{N} \setminus \{i\}} \{f_{ij}\}$
- Total CA: $\max_{j \in \mathcal{N} \setminus \{i\}} \{f_{ij}\}$
- Other criteria depending, for instance, on time windows width.

Initialize V with the first k nodes in the ordering.

ng-routes relaxation (Baldacci et al., 2011)

With each node $i \in N$ a neighborhood N_i is associated, such that $i \in N_i$.
Consider a path $P = (P_1, P_2, \ldots, P_l)$ with $P_k \in N \; \forall k = 1, \ldots, l$.
Its critical set is

$$\Pi_P = \{P_k \in P : P_k \in \bigcap_{h=k+1}^{l} N_{P_h}\} \cup \{P_l\}.$$

This set contains all nodes visited by P that are "close to" all nodes visited after them.

A forward *ng*-path (NG, i) is a path that

- ends at node i
- has $NG = \Pi_P$ with $i \notin P'$, where P' is P without the last arc.

The extension of a path P to any node in Π_P is forbidden, because it would produce a cycle.

ng-routes relaxation

A different state is kept for each subset NG and each i.
The number of states can grow expnentially with the size of the neighborhoods N.

Extension rule:

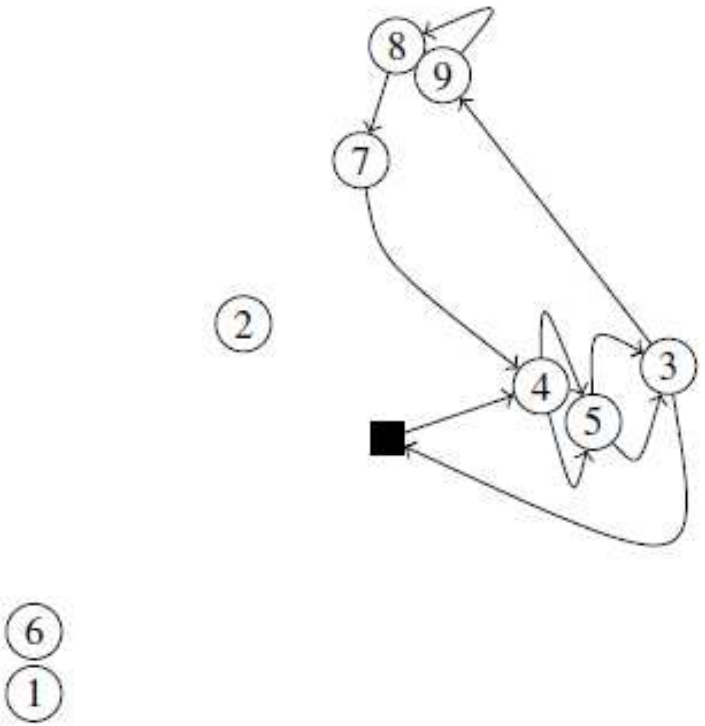
$$z(NG, i) = \min_{(NG', j) \in \mathcal{P}} \{z(NG', j) + c_{ji}\}$$

$$z(\emptyset, 0) = 0$$

Rule of thumb: each N_i contains 15 nearest neighbors of each node i.

Example

Each neighborhood is initialized with the 4 nearest neighbors.



$N_1 = \{1, 2, 4, 6\}$
$N_2 = \{2, 7, 8, 9\}$
$N_3 = \{3, 4, 5, 7\}$
$N_4 = \{3, 4, 5, 7\}$
$N_5 = \{3, 4, 5, 7\}$
$N_6 = \{1, 2, 6, 7\}$
$N_7 = \{2, 7, 8, 9\}$
$N_8 = \{2, 7, 8, 9\}$
$N_9 = \{2, 7, 8, 9\}$

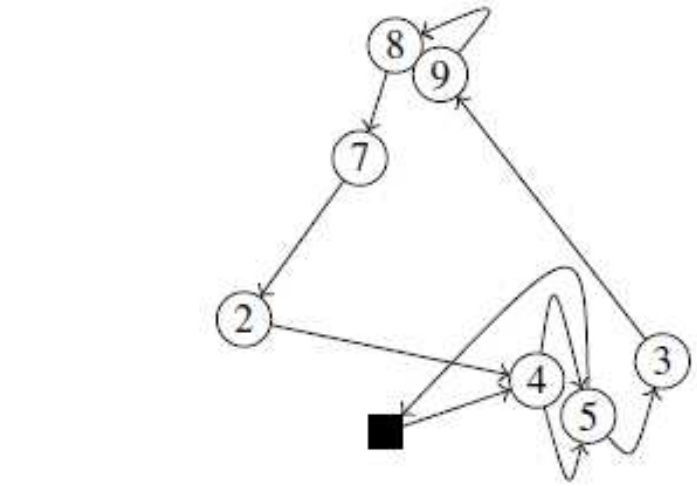
$lb^1 = 12,191$

Figure 2 First Iteration of the `Dng-Path` Relaxation
Note. A least-cost ng-tour is (0, 4, 5, 3, 9, 8, 7, 4, 5, 3, 0) of cost 12,191.

The solution contains a cycle, with $C_1 = 4$.
Node 4 does not belong to N_7, N_8 and N_9.

Example

Node 4 has been inserted in N_7, N_8 and N_9.



$N_1 = \{1, 2, 4, 6\}$
$N_2 = \{2, 7, 8, 9\}$
$N_3 = \{3, 4, 5, 7\}$
$N_4 = \{3, 4, 5, 7\}$
$N_5 = \{3, 4, 5, 7\}$
$N_6 = \{1, 2, 6, 7\}$
$N_7 = \{2, 4, 7, 8, 9\}$
$N_8 = \{2, 4, 7, 8, 9\}$
$N_9 = \{2, 4, 7, 8, 9\}$

$lb^2 = 12,707$

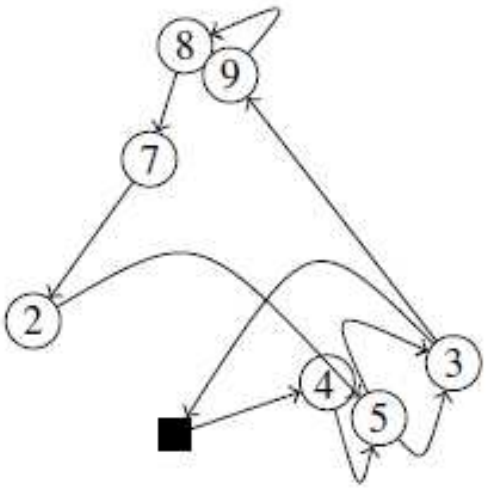
Figure 3 Second Iteration of the Dng-Path Relaxation

Note. A least-cost ng-tour is (0, 4, 5, 3, 9, 8, 7, 2, 4, 5, 0) of cost 12,707.

The solution contains a cycle, with $C_1 = 4$.
Node 4 does not belong to N_2.

Example

Node 4 has been inserted in N_2.



$N_1 = \{1, 2, 4, 6\}$
$N_2 = \{2, 4, 7, 8, 9\}$
$N_3 = \{3, 4, 5, 7\}$
$N_4 = \{3, 4, 5, 7\}$
$N_5 = \{3, 4, 5, 7\}$
$N_6 = \{1, 2, 6, 7\}$
$N_7 = \{2, 4, 7, 8, 9\}$
$N_8 = \{2, 4, 7, 8, 9\}$
$N_9 = \{2, 4, 7, 8, 9\}$

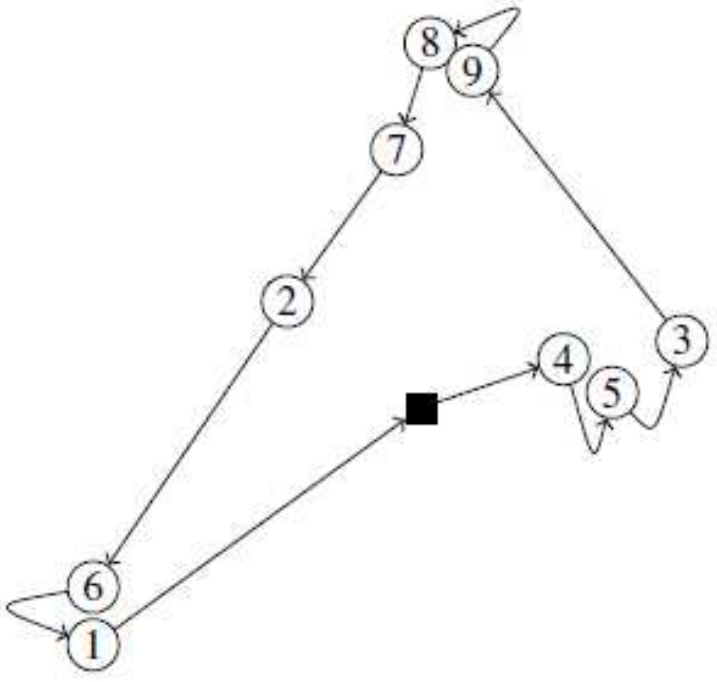
$lb^3 = 13{,}012$

Figure 4 Third Iteration of the `Dng-Path` Relaxation
Note. A least-cost ng-tour is (0, 4, 5, 3, 9, 8, 7, 2, 5, 3, 0) of cost 13,012.

The solution contains a cycle, with $C_1 = 5$.
Node 5 does not belong to N_2, N_7, N_8 and N_9.

Example

Node 5 has been inserted into N_2, N_7, N_8 and N_9.



$N_1 = \{1, 2, 4, 6\}$
$N_2 = \{2, 4, 5, 7, 8, 9\}$
$N_3 = \{3, 4, 5, 7\}$
$N_4 = \{3, 4, 5, 7\}$
$N_5 = \{3, 4, 5, 7\}$
$N_6 = \{1, 2, 6, 7\}$
$N_7 = \{2, 4, 5, 7, 8, 9\}$
$N_8 = \{2, 4, 5, 7, 8, 9\}$
$N_9 = \{2, 4, 5, 7, 8, 9\}$

$lb^4 = 13{,}323$

Figure 5 Last Iteration of the `Dng-Path` Relaxation
Note. The least-cost ng-tour (0, 4, 5, 3, 9, 8, 7, 2, 6, 1, 0) is elementary, and 13,323 is the optimal solution cost.

The solution is cycle-free and hence optimal.

Dynamic *ng*-routes relaxation (Roberti and Mingozzi, 2014)

The size of the neighborhoods is initially set to Δ' and it is iteratively enlarged (up to a maximum size Δ'') according to the cycles detected.

Every time a lower bound is computed, a minimum cardinality cycle $C = (C_1, C_2, \ldots, C_l, C_1)$ is searched such that

$$|N_{C_k}| < \Delta'' \quad \forall C_k : C_1 \notin N_{C_k}$$

If no such cycle exists, then the algorithm stops.
Otherwise C_1 is inserted into the neighborhoods N_{C_k} not including it and a new iteration is executed.

Heuristic D.P.

From exact to heuristic D.P. by relaxing the dominance test.

For instance, in the RCESPP replace

$$
\begin{aligned}
& i' = i'' \\
& \mathcal{S}' \subseteq \mathcal{S}'' \\
& r' \leq r'' \\
& C(i, r', \mathcal{S}') \leq C(i, r'', \mathcal{S}'')
\end{aligned}
$$

with

$$
\begin{aligned}
& i' = i'' \\
& |\mathcal{S}'| \leq |\mathcal{S}''| \\
& r' \leq r'' \\
& C(i, r', \mathcal{S}') \leq C(i, r'', \mathcal{S}'')
\end{aligned}
$$

still keeping the constraint $j \notin \mathcal{S}$ when extending $(i, \mathcal{S}, r)$ along (i, j).

The final solution will be feasible, but not necessarily optimal.

Continuous resources: the VRPSTW

VRP with Soft Time Windows:

- a graph $G = (V, A)$, with n customers and a depot,
- a demand q_i for each node $i \in N$,
- V identical vehicles of capacity Q,
- non-negative weights t_{ij} (time) and c_{ij} (cost) $\forall (i, j) \in A$,
- a service time $\theta_i \geq 0$ and a time window $[a_i, b_i]$ $\forall i \in N$.

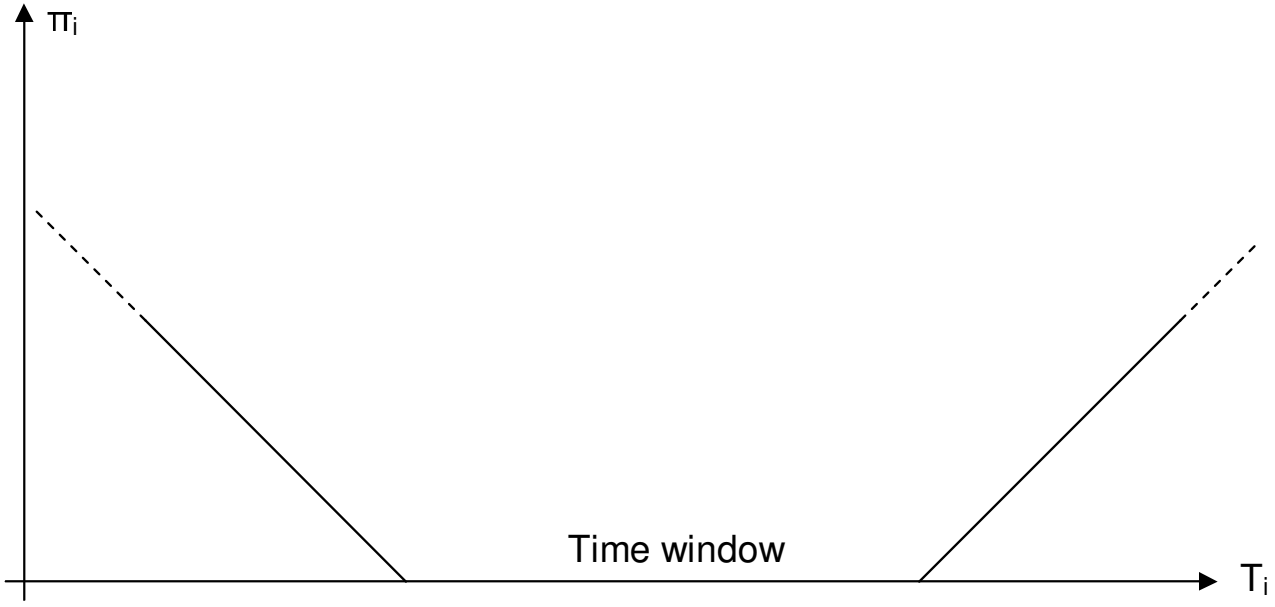
If the service at $i \in N$ starts before a_i or after b_i, a linear penalty is paid, through non-negative coefficients α_i (advance) and β_i (delay).

Indicating with T_i the starting time of service at $i \in N$, the penalty term $\pi_i(T_i)$ is:

$$\pi_i(T_i) = \begin{cases} \alpha_i(a_i - T_i) & \text{if } T_i < a_i \\ 0 & \text{if } a_i \leq T_i \leq b_i \\ \beta_i(T_i - b_i) & \text{if } T_i > b_i. \end{cases}$$

Vehicles are allowed to wait at no cost.

Soft time windows



A soft time window at node $i \in \mathcal{N}$: a linear penalty π_i is incurred depending on the service start time T_i.

States and labels

When a vehicle reaches a node $i \in N$ before a_i, it can start the service immediately or it can wait and start the service at a later time, in order to reduce the penalty.

Therefore, from each feasible state an infinite set of feasible states is generated.

The dynamic programming algorithm must take into account an infinite set of non-dominated states: this is done by grouping them into labels.

Each label corresponds to an infinite set of states associated with the same path.

States and labels

A label associated with node $i \in \mathcal{N}$ is a tuple $L_i = (S, i, r, C(T_i))$, where

- S is a binary vector indicating the nodes visited along the path,
- i is the last reached node,
- r is the amount of capacity consumed up to i,
- C is the cost of the path,
- T_i is the time at which the service at node i starts.

In each label the continuous function $C(T_i)$ describes the trade-off between cost and time.

This function is piecewise linear and convex, because it is the sum of piecewise linear and convex functions.

Extension

When $(S, i, r, C(T_i))$ is extended along (i, j) generating $(S', j, r', C'(T_j))$,

$$S'_k = \begin{cases} S_k + 1 & k = j \\ S_k & k \neq j \end{cases}$$

$$r' = r + q_j$$

$$C'(T_j) = C(T_j - (\theta_i + t_{ij})) - \lambda_i/2 + c_{ij} - \lambda_j/2 + \pi_j(T_j),$$

where $\lambda_0 = -\mu$ (dual variable of the convexity constraint).

Feasibility: $(S_k \leq 1 \ \forall k \in \mathcal{N}) \wedge (r \leq Q)$.

The cost function C of the predecessor is evaluated at $T_i = T_j - (\theta_i + t_{ij})$, which is the latest point in time at which the service at node i can start, to allow starting the service at node j at time T_j.

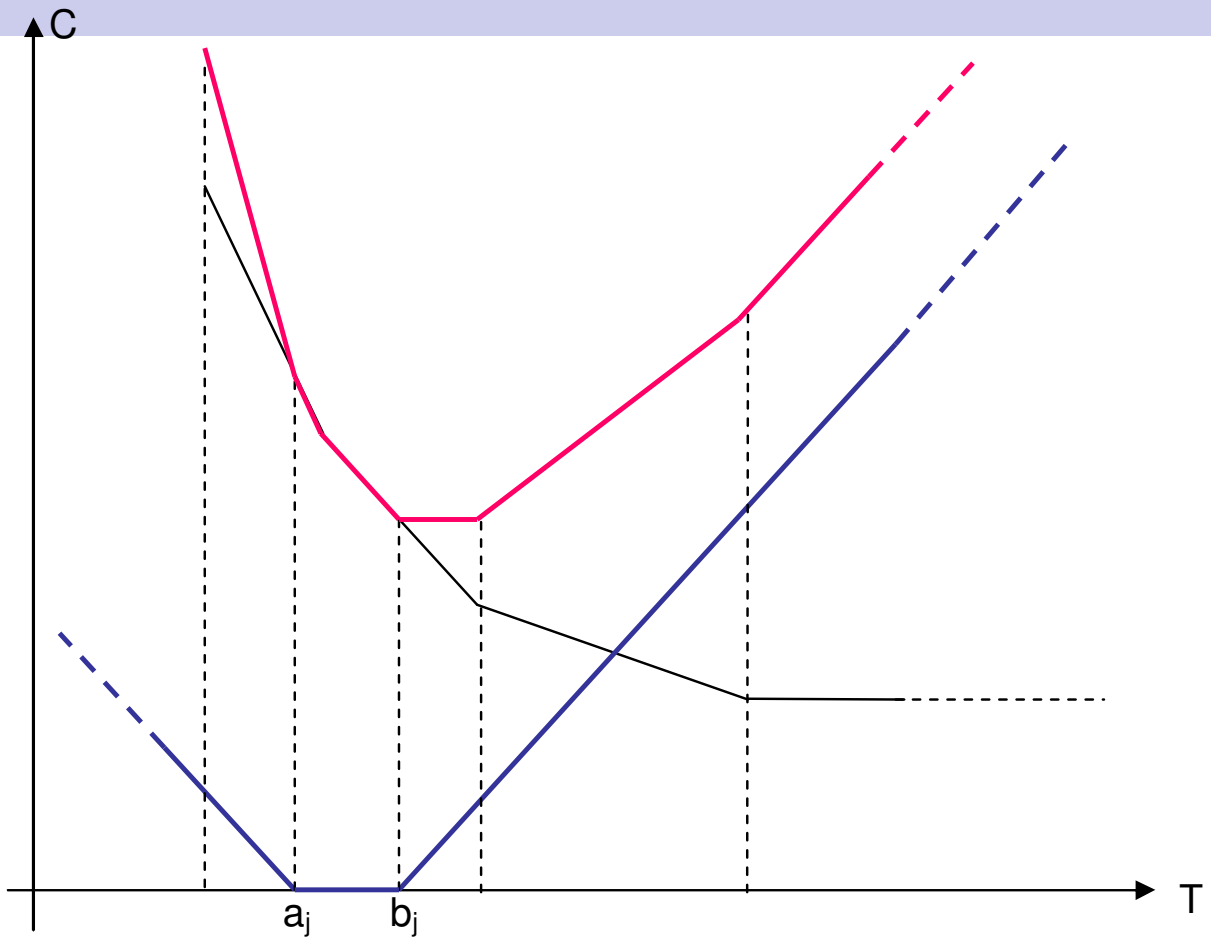


Figura: Forward extension from node i to node j. The $C'(T_j)$ function is the sum of the $C(T_i)$ function suitably right-and-up-shifted and the penalty function $\pi_j(T_j)$.

Dominance 1

Since waiting at no cost is allowed, if state with cost C and time T is feasible, all states with the same cost C and time larger than T are also reachable.

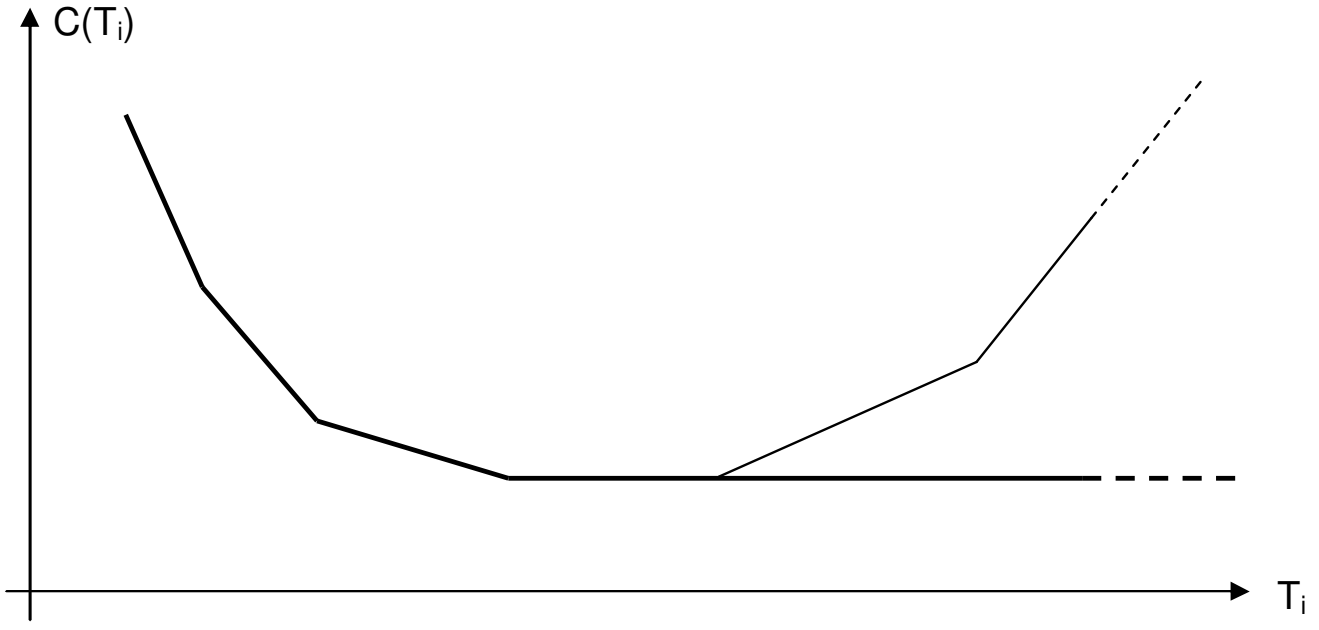


Figura: States on the ascending part of the piecewise linear function are dominated: the same value in time can be reached at a smaller cost.

Dominance 2

Dominance test.

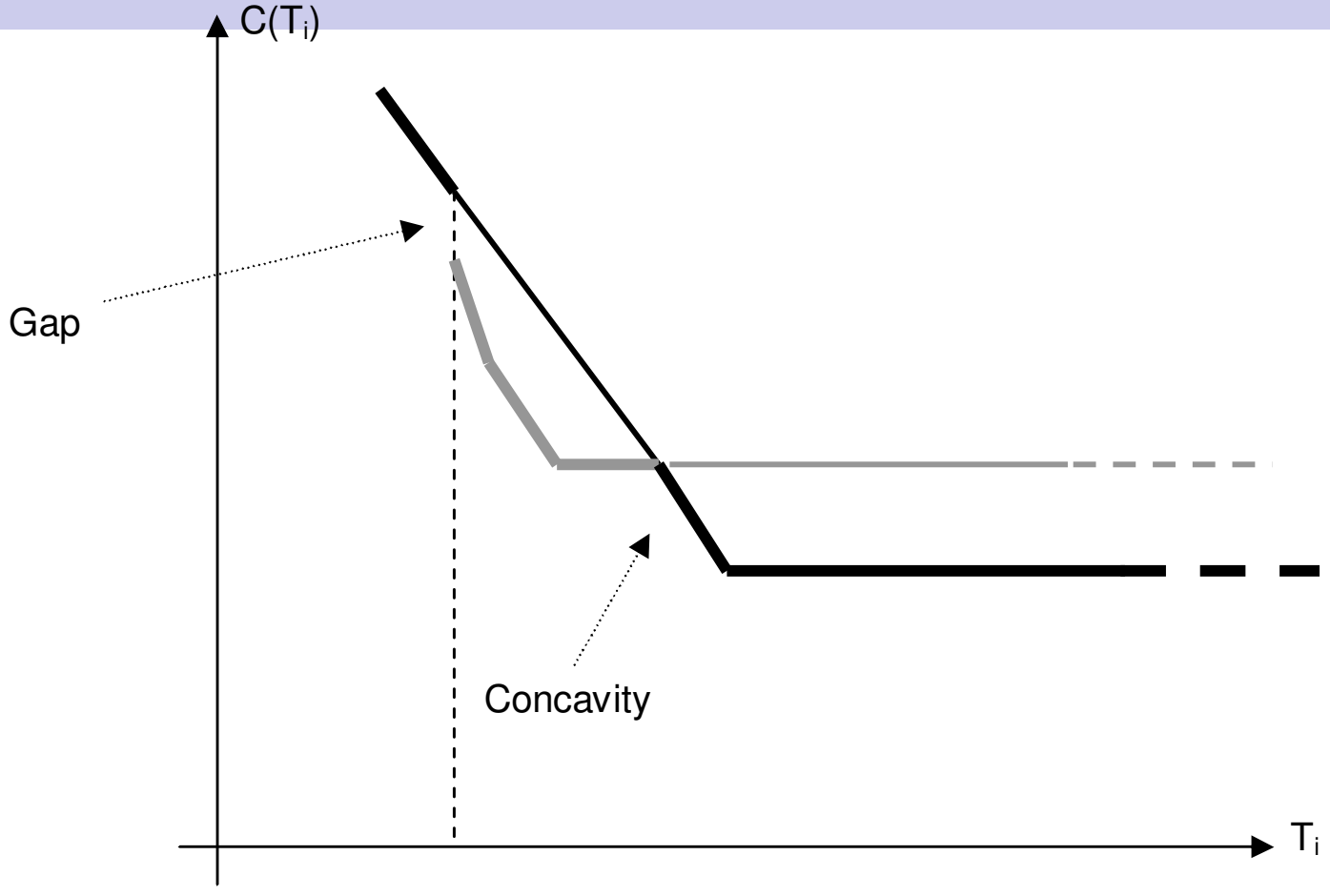
Let (S', i, r', C', T_i') and $(S'', i, r'', C'', T_i'')$ be two states belonging to two labels L' and L''.

Then the former dominates the latter only if

$$S' \leq S''$$
$$r' \leq r''$$
$$C' \leq C''$$
$$T_i' \leq T_i''$$

and at least one of the inequalities is strict.

The effect is to delete some parts of the piecewise linear functions.



Non-dominated states (strong lines).

The resulting piecewise linear functions may have gaps and are not convex in general,

Dominance

Comparing a new generated label L' with an existing one L'':

if $S' = S''$, $i' = i''$ and $r' = r''$, then the two labels are merged;

otherwise, some states in one of the two piecewise linear functions can be dominated; hence, the resulting piecewise linear function can have also horizontal gaps.

Join

Forward label: $L^{fw} = (S^{fw}, i, r^{fw}, C^{fw}(T_i))$.
Backward label: $L^{bw} = (S^{bw}, j, r^{bw}, C^{bw}(T_j))$.

Feasibility test:

$$S_k^{fw} + S_k^{bw} \leq 1 \quad \forall k \in \mathcal{N}.$$

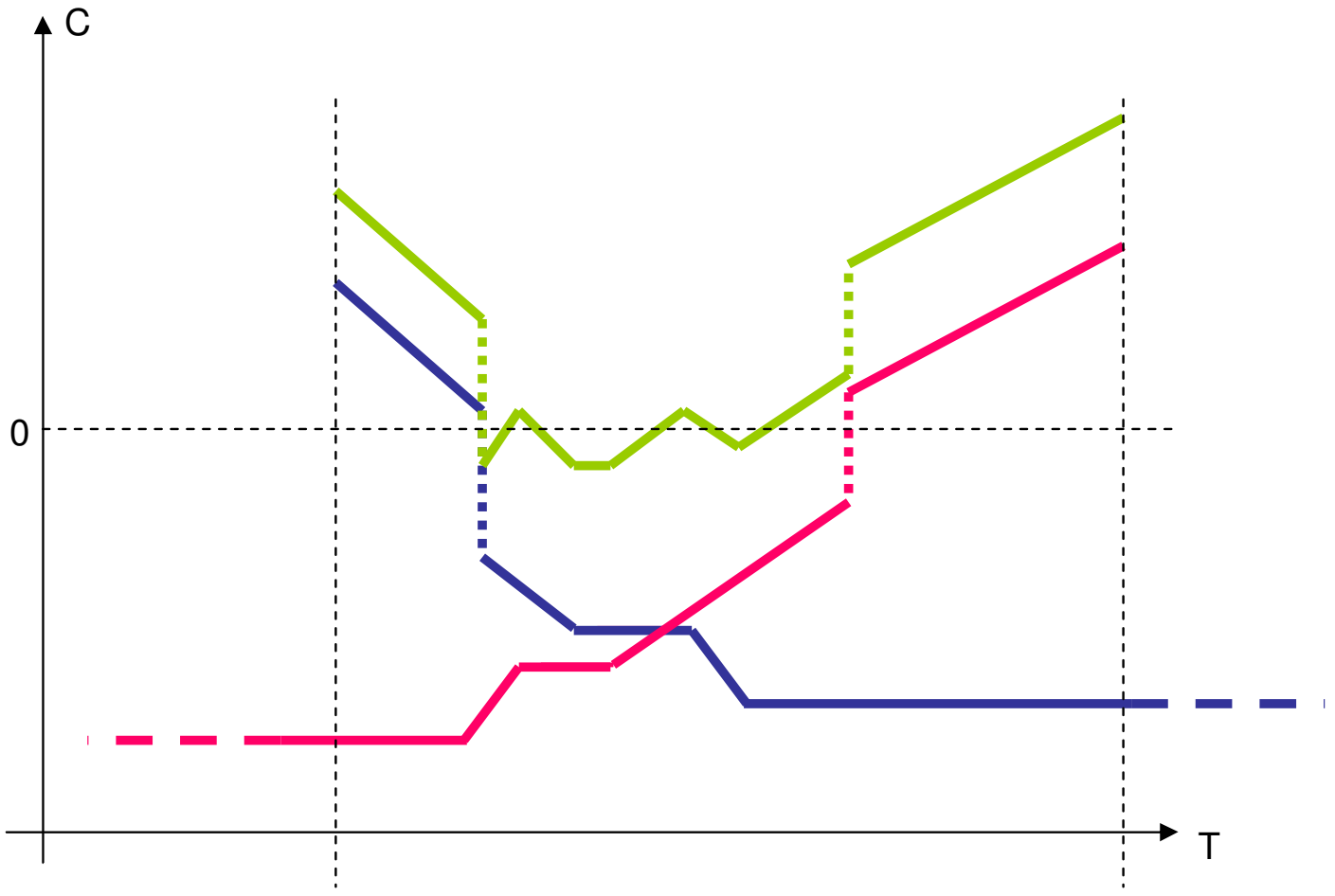
$$r^{fw} + r^{bw} \leq Q.$$

Cost:

$$C(T) = C^{fw}(T) - \lambda_i/2 + c_{ij} - \lambda_j/2 + C^{bw}(T + \theta_i + t_{ij}).$$

This function may have several local minima.
Finding the global minimum takes time linear in the number of discontinuity points of the two piecewise linear functions (merge two sorted lists).

Join



Blue: forward cost. Red: backward cost. Green: total cost.