Dynamic programming for polynomial-time problems

Discrete Optimization

Giovanni Righini

UNIVERSITÀ DEGLI STUDI DI MILANO

Implicit enumeration

Combinatorial optimization problems are in general *NP*-hard and we usually resort to implicit enumeration to solve them to optimality (this is also useful for approximation purposes).

Two main methods for implicit enumeration are:

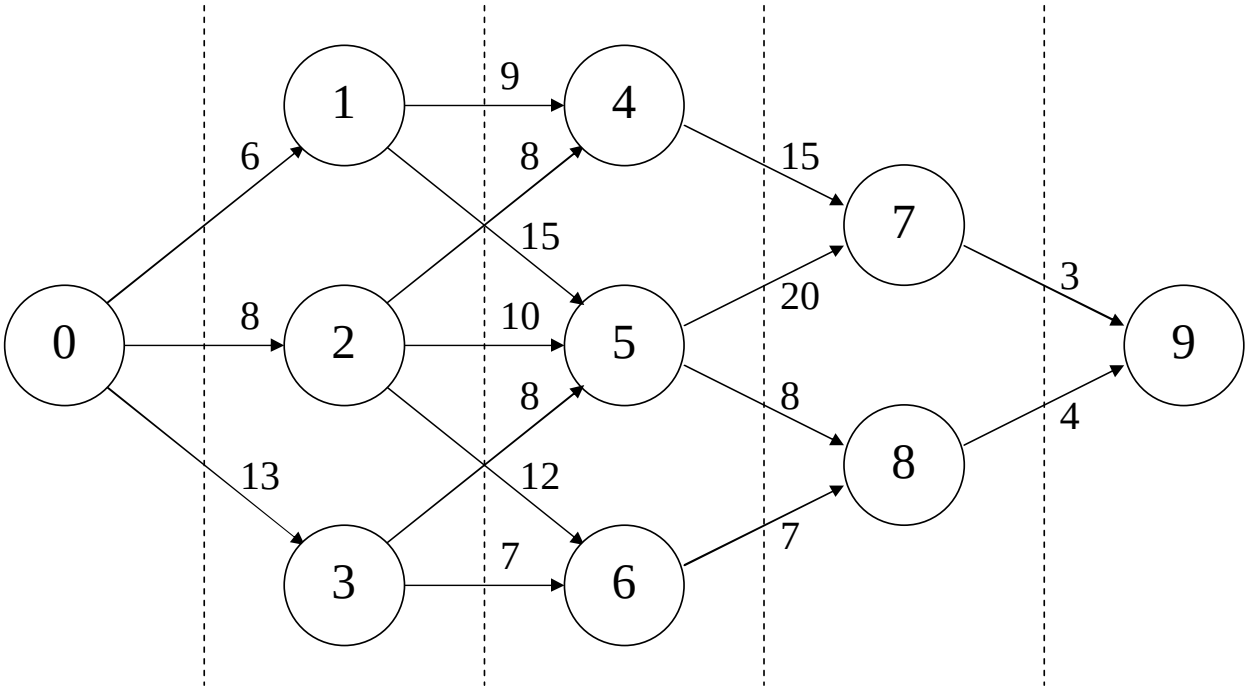
- branch-and-bound,
- dynamic programming.

Dynamic programming is also an algorithmic framework to solve polynomial time problems efficiently.

It also allows to devise pseudo-polynomial time algorithms as well as approximation algorithms.

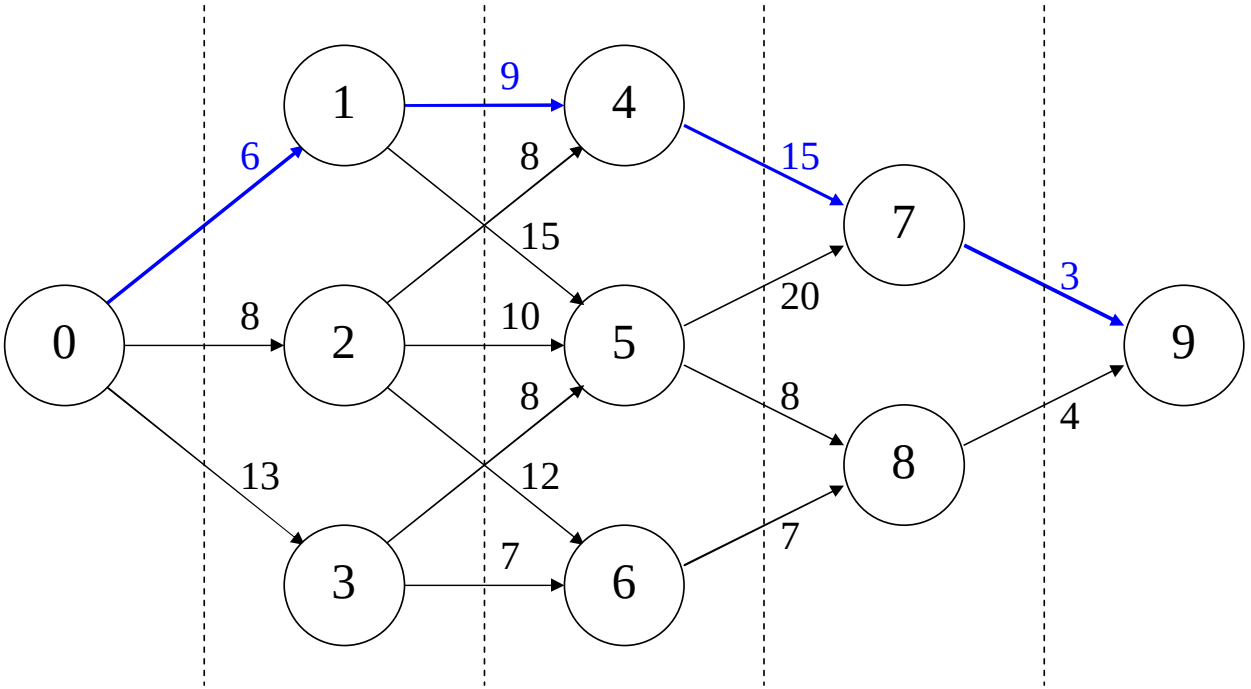
An introductory example

Consider the problem of finding a shortest path from node 0 to 9 on this graph, which is directed, acyclic and layered.



An introductory example

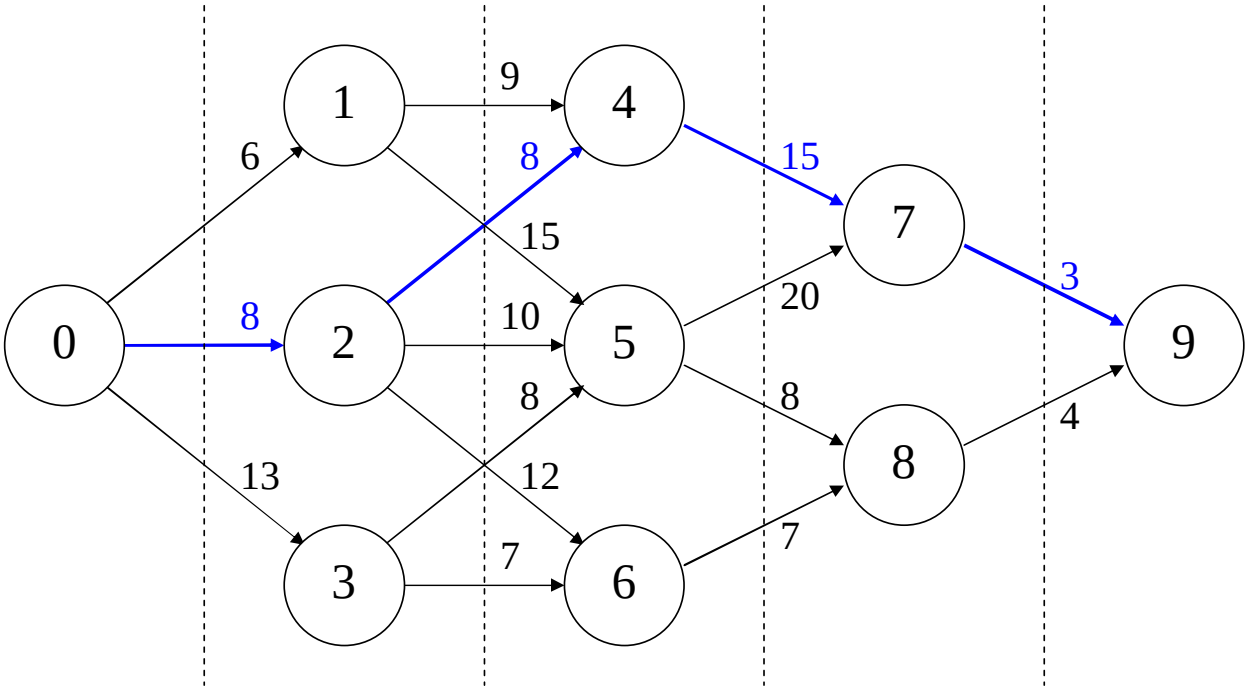
Consider the problem of finding a shortest path from node 0 to 9 on this graph, which is directed, acyclic and layered.



A greedy algorithm from 0 would produce a path of cost 33.

An introductory example

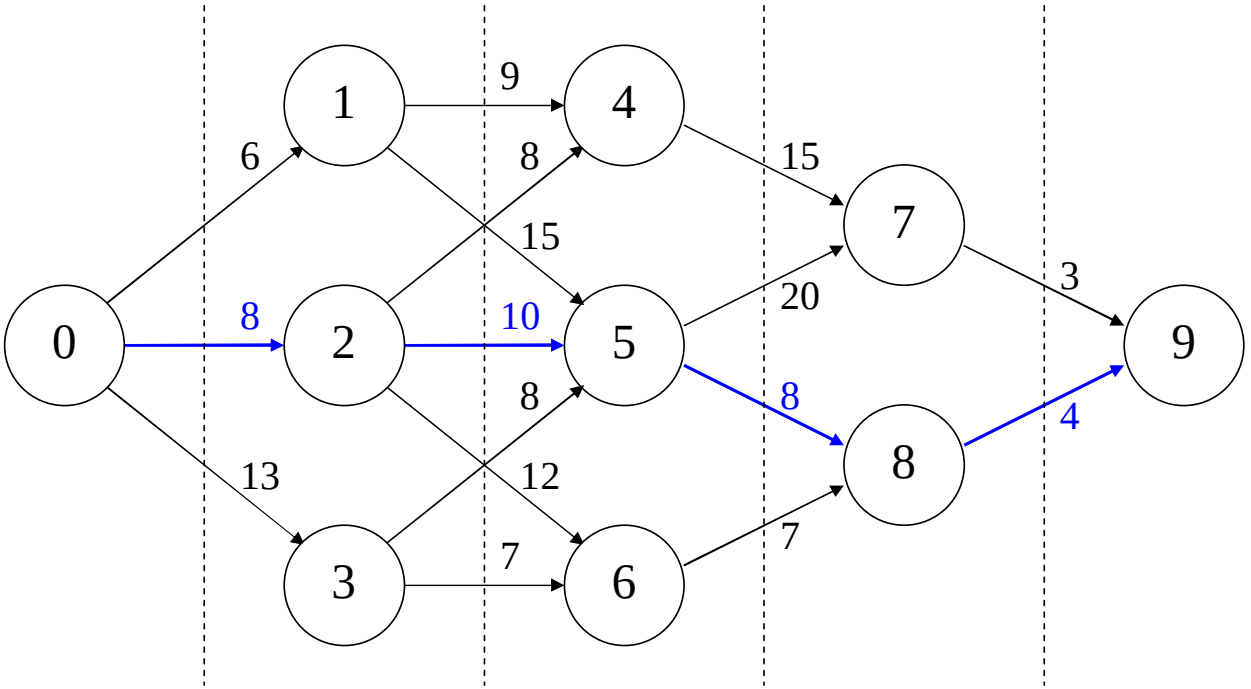
Consider the problem of finding a shortest path from node 0 to 9 on this graph, which is directed, acyclic and layered.



A greedy algorithm from 9 would produce a path of cost 34.

An introductory example

Consider the problem of finding a shortest path from node 0 to 9 on this graph, which is directed, acyclic and layered.



The optimal path has cost 30.

Bellman's optimality principle (1953)

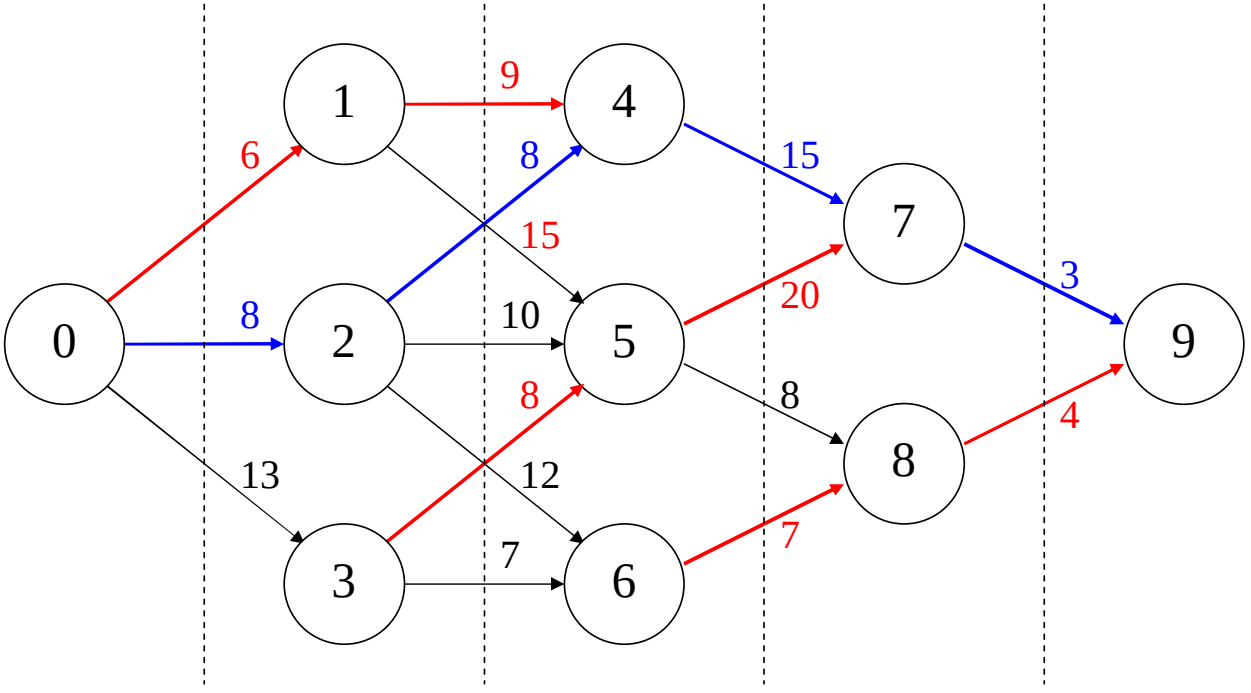
An optimal policy is made of a set of optimal sub-policies.

By policy we mean a sequence of decisions, i.e. of value assignments to the variables).

A sub-policy is a sub-sequence of decisions, i.e. of value assignments to a subset of the variables.

Richard E. Bellman
(New York, 1920 - 1984)

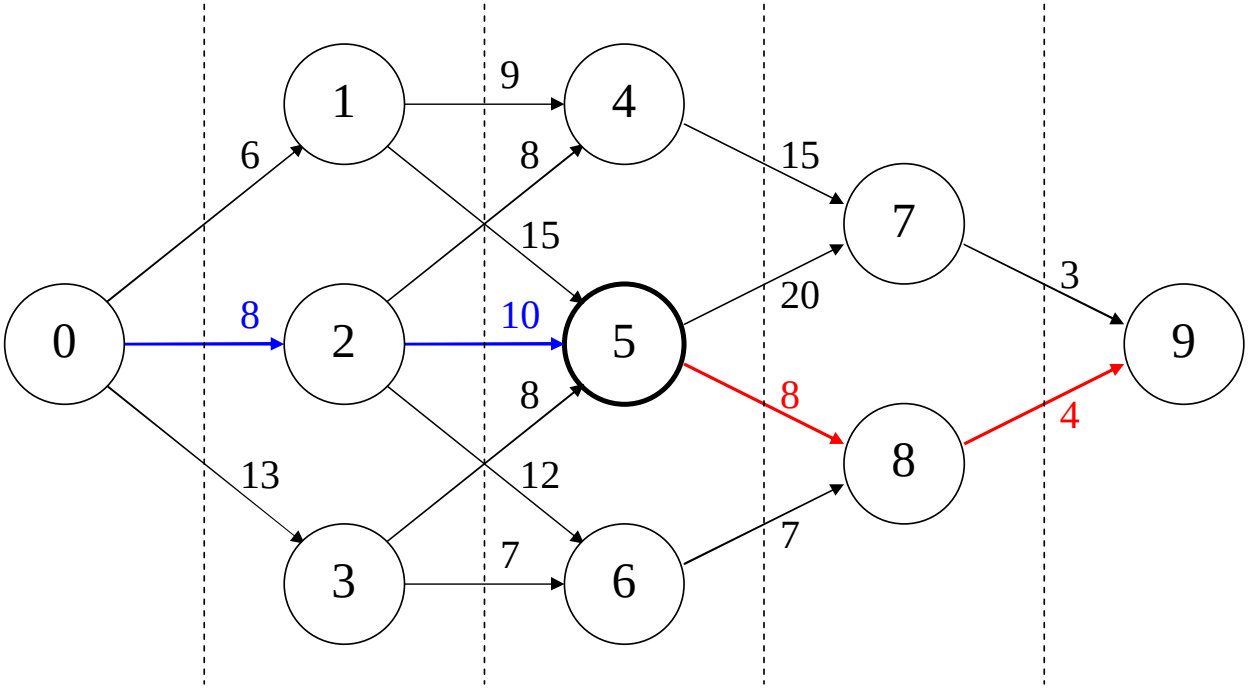
The example revisited



A decision must be taken every time the path is extended from one layer to the next.

A policy is a path from 0 to 9. A sub-policy is any other path.

The example revisited



The **optimal policy** is made of pairs of optimal sub-policies of the form (0, *i*) and (i,9).

We only need to store **optimal sub-policies**. All the other sub-policies can be disregarded.

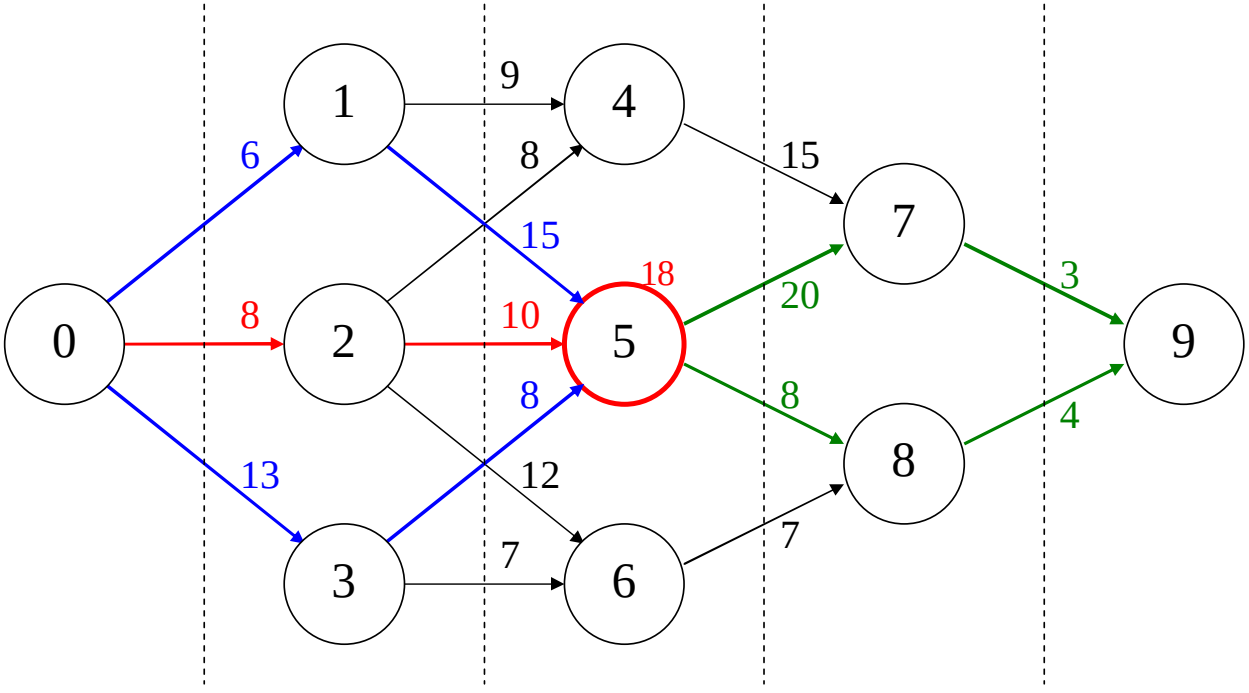
Dominance

Given two sub-policies S' and S'', S' dominates S'' only if

- all sub-policies that can be appended to S'' can also be appended to S', with no greater cost;
- the cost of S' is less than the cost of S''.

When this occurs, S'' can be neglected from further consideration: it cannot be part of an optimal policy.

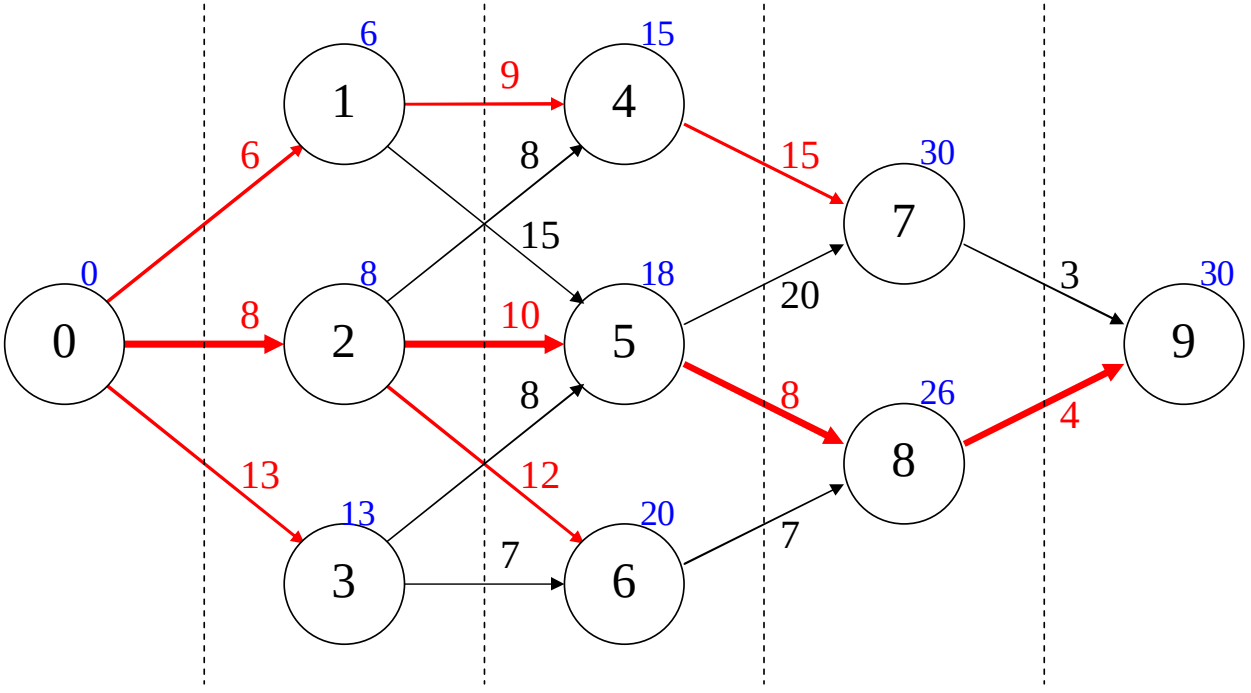
The example revisited



In our example, all sub-policies leading to the same node (paths of the form $(0, i)$), can be completed by appending to them the same sub-policies (paths of the form $(i, 9)$).

Then we need to store only an optimal one among them. All the others are dominated.

The example finally solved



Let $\mathcal{L} = \{0, \ldots, L\}$ be the set of layers.
Let $\mathcal{N}_l$ the subset of nodes in layer $l \in \mathcal{L}$.
Let w be the weight function of the arcs of the graph.

- $c(0) = 0$
- $\forall l \in \mathcal{L}, l \geq 1 \quad \forall j \in \mathcal{N}_l \quad c(j) = \min_{i \in \mathcal{N}_{l-1}} \{c(i) + w_{ij}\}$

Terminology and correspondence

- Nodes in the graph are **states**.
- Arcs of the graph are **state transitions**.
- Paths in the graphs are **(sub-)policies**.
- Values $c(i)$ (costs of shortest paths from 0 to i) are **labels**.

The solution process applies these two rules:

- **Initialization (recursion base)**: $c(0) = 0$
- **Label extension (recursive step)**:
  $\forall l \in \mathcal{L}, l \geq 1 \quad \forall j \in \mathcal{N}_l \quad c(j) = \min_{i \in \mathcal{N}_{l-1}} \{c(i) + w_{ij}\}$.

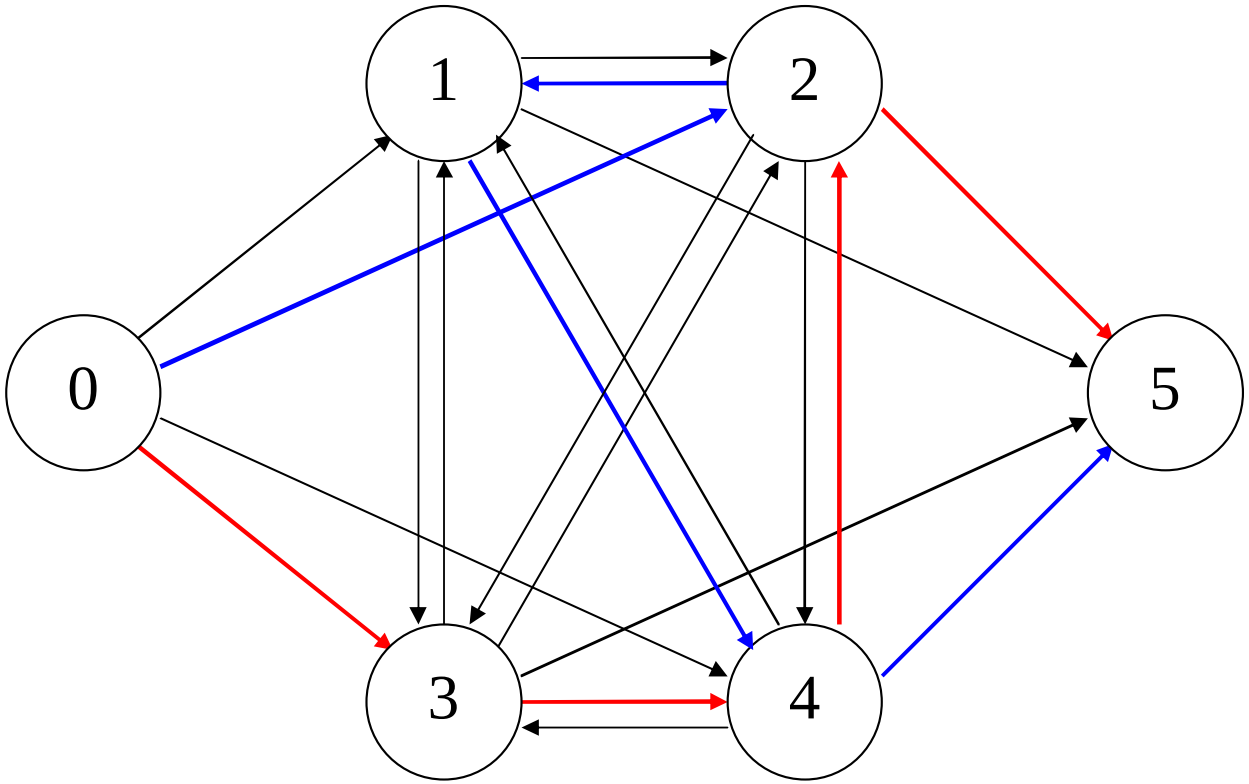
It resembles recursive programming, but it is bottom-up instead of top-down.

The execution of a **dynamic programming algorithm** resembles the evolution of a discrete dynamic system (automaton).

The recursive step requires to solve a very easy optimization problem.

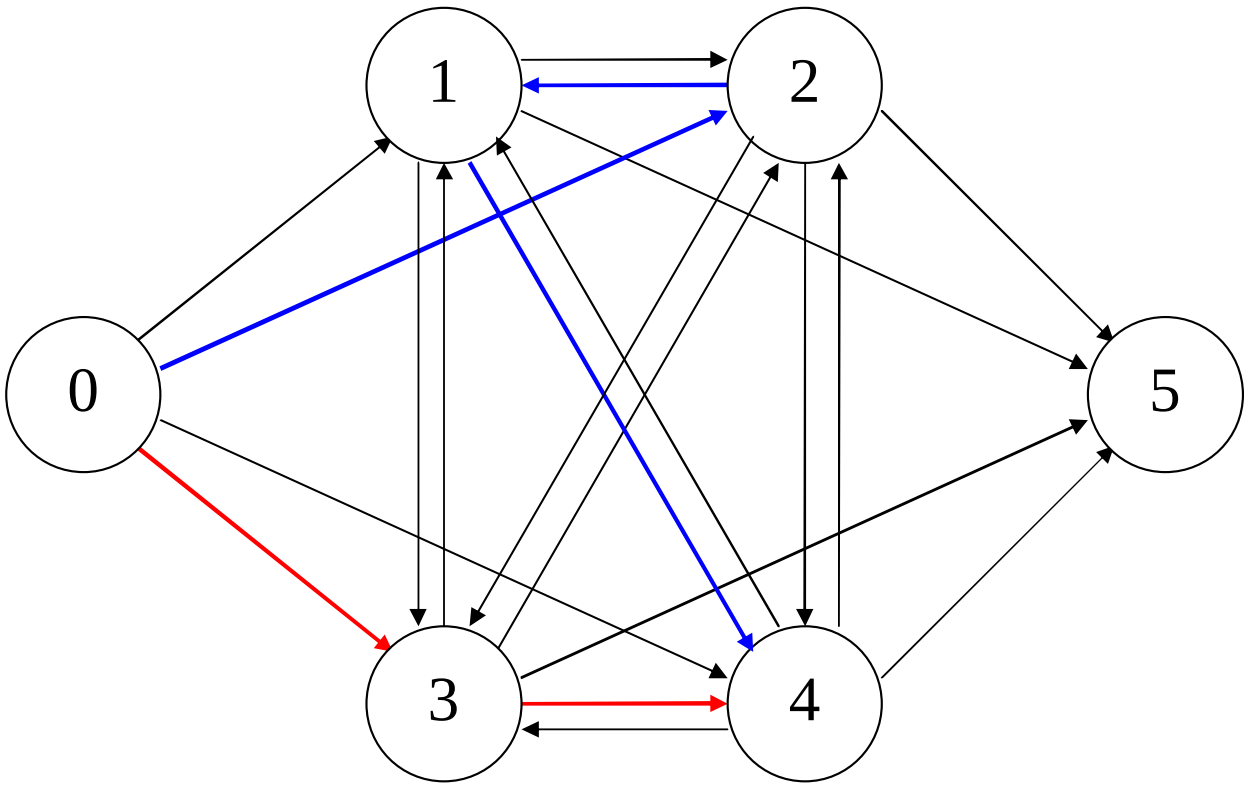
Another example

Consider the problem of finding a shortest Hamiltonian path from node 0 to node 5 on this graph, that is directed but cyclic.



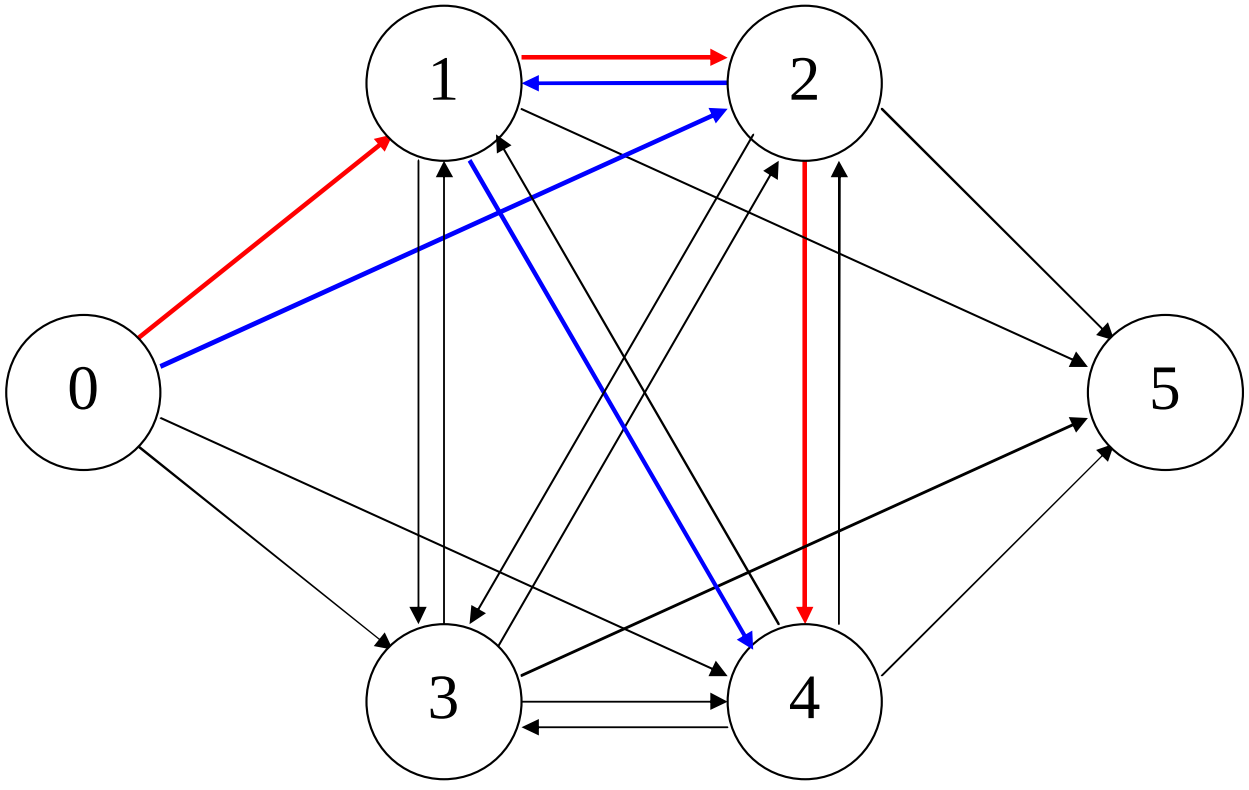
A policy (i.e. a path from 0 to 5) is feasible if and only if it visits all nodes exactly once.

Dominance?



Reaching the same node is no longer sufficient for a path (sub-policy) to dominate another: they do *not* reach the same state.

Dominance



Only if they have also visited the same subset of nodes, then they reach the same state.

- **Initialization:** $c(0, \{0\}) = 0$
- **Extension:** $c(j, S) = \min_{i \in S \setminus \{j\}} \{c(i, S \setminus \{j\}) + w_{ij}\} \quad \forall j \neq 0.$

Complexity

The two D.P. algorithms have different complexity.

Ex. 1: n. states = n. of distinct values of j = n. of nodes in the graph.

- **State:** (i) *[last reached node]*;
- **Initialization:** $c(0) = 0$;
- **Extension:** $c(j) = \min_{i \in \mathcal{N}_{l-1}} \{c(i) + w(i,j)\} \;\; \forall l \geq 1 \in \mathcal{L}, \; \forall j \in \mathcal{N}_l$.

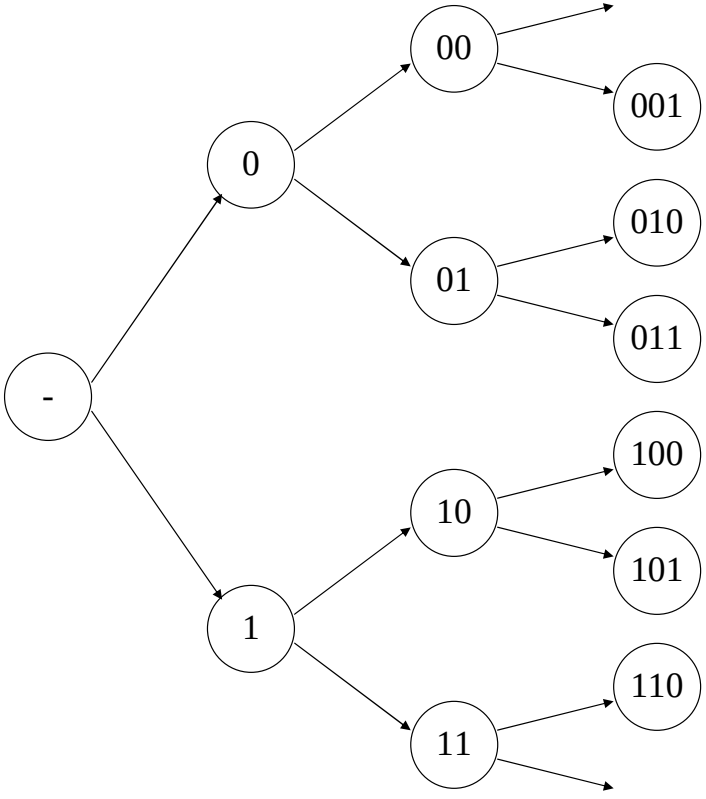
Ex. 2: n. states = n. distinct values of (j, S) = exponential number in the n. of nodes in the graph.

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

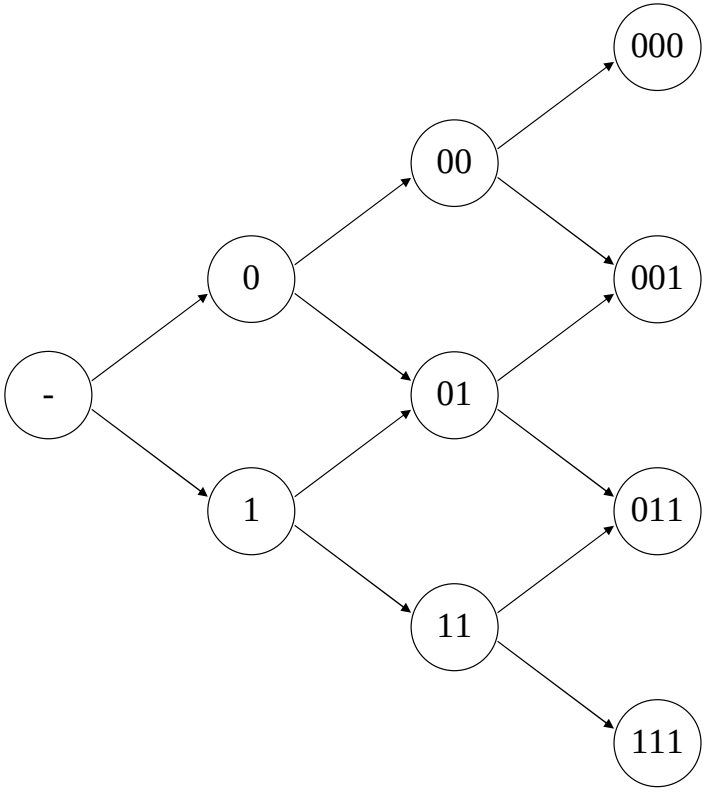
This is equivalent to solve the same problem as in Example 1 but on a larger state graph (one node for each distinct value of (j, S)).

The complexity of the D.P. algorithm depends on the number of arcs in the state graph.

Comparison with branch-and-bound



B&B: sub-policies only diverge.



D.P.: sub-policies sometimes converge.

In both cases graphs are directed, acyclic and layered.
But one is an arborescence, the other can be not.

Dynamic Programming in three steps

In order to design a D.P. algorithm we need to:

1. put the decisions (i.e. the variables) in a sequence (policy);
2. define the state, i.e. the amount of information needed to transform a sub-policy into a complete policy;
3. find the recursive extension function (REF) to compute the labels of the states, following the sequence.

The state of a dynamic system at time t summarizes the past history of the system up to time t, such that the evolution of the system after t depends on the state at time t but not on how it has been reached.

Solving a problem with D.P. amounts at defining a suitable state-transition graph, on which we search for an optimal path.

This graph is directed, acyclic and layered.

The number of its nodes and arcs determines the complexity of the D.P. algorithm.

Shortest path on an acyclic digraph: pseudo-code

TopologicalSort;
ComputePredecessors;
for $j = 0, \ldots, s-1$ **do**
  $c(j) \leftarrow \infty$;
$c(s) \leftarrow 0$;
for $j = s+1, \ldots, t$ **do**
  for $i \in Pred(j)$ **do**
    if $(c(i) + w_{ij} < c(j))$ **then**
      $c(j) \leftarrow c(i) + w_{ij}$;
      $\pi(j) \leftarrow i$;
$z^* \leftarrow c(t)$;
$X^* \leftarrow \emptyset$;
$j \leftarrow t$;
while $(j > s)$ **do**
  $X^* \leftarrow X^* \cup \{(\pi(j), j)\}$;
  $j \leftarrow \pi(j)$;
return z^*, X^*

$c(i)$: minimum cost to reach i.
$\pi(i)$: optimal predecessor of i.
z^*: minimum $s-t$ cost.
X^*: optimal solution.

Complexity.

TopologicalSort: $O(m)$.
ComputePredecessors: $O(m)$.
Neglect unreachable nodes: $O(n)$.
Label extension: $O(m)$.
Rebuild optimal solution: $O(n)$.

Complexity: $O(m)$.

Label correcting variation

A variation of the D.P. algorithm is obtained by extending the labels from each state to its successors.

$$Succ(i) = \{j \in N : (i,j) \in A\}.$$

Algorithm 1 Label setting.

```
...
for j = s + 1, ..., t do
   for i ∈ Pred(j) do
      if (c(i) + w_ij < c(j)) then
         c(j) ← c(i) + w_ij;
         π(j) ← i;
...
```

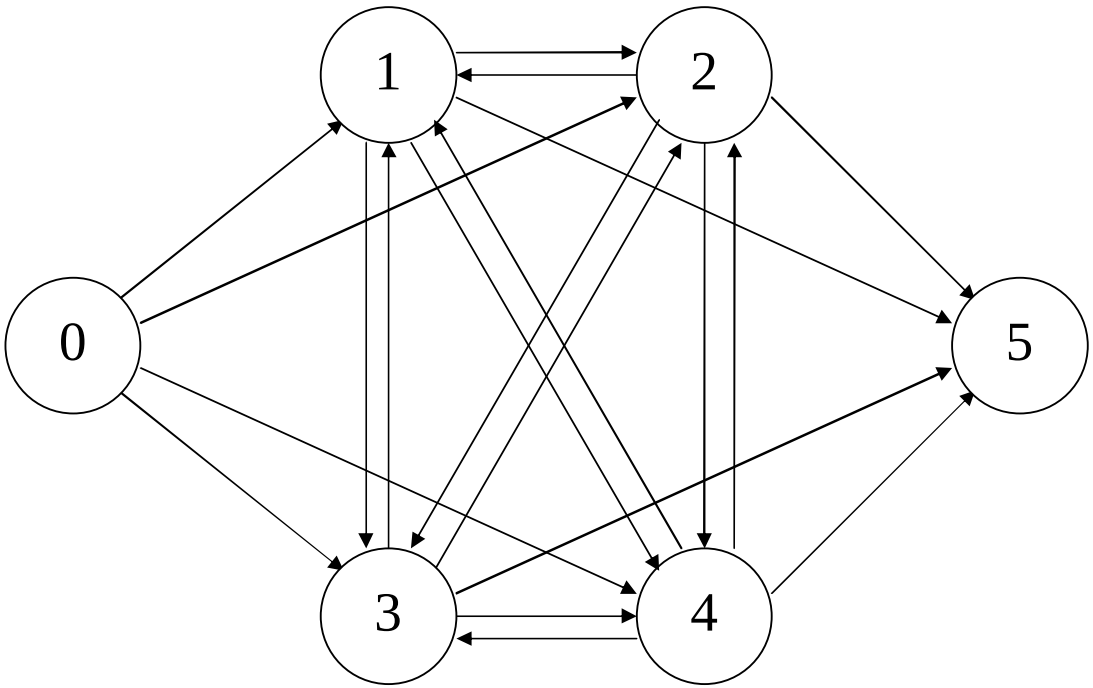
Algorithm 2 Label correcting.

```
...
for i = s, ..., t − 1 do
   for j ∈ Succ(i) do
      if (c(i) + w_ij < c(j)) then
         c(j) ← c(i) + w_ij;
         π(j) ← i;
...
```

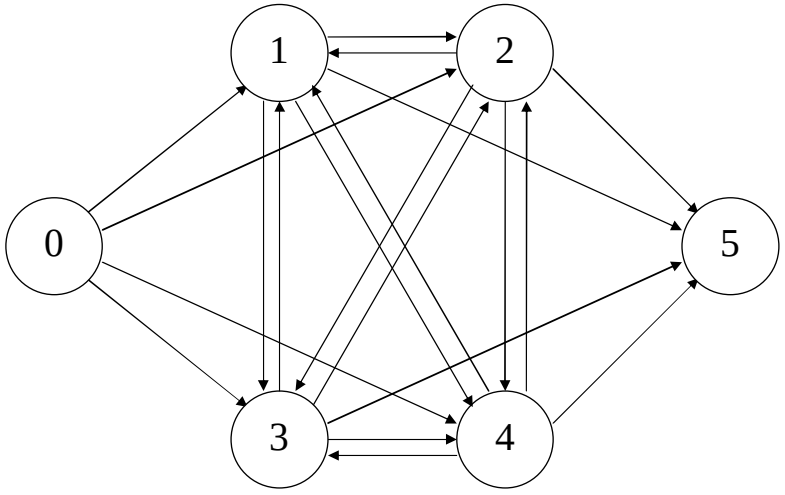
The time complexity is the same, i.e. $O(m)$, because every arc is examined once.

Shortest path on a weighted digraph

Find a shortest path from node 0 to node 5 on a weighted cyclic digraph with no negative cost cycles.



Shortest path on a weighted digraph



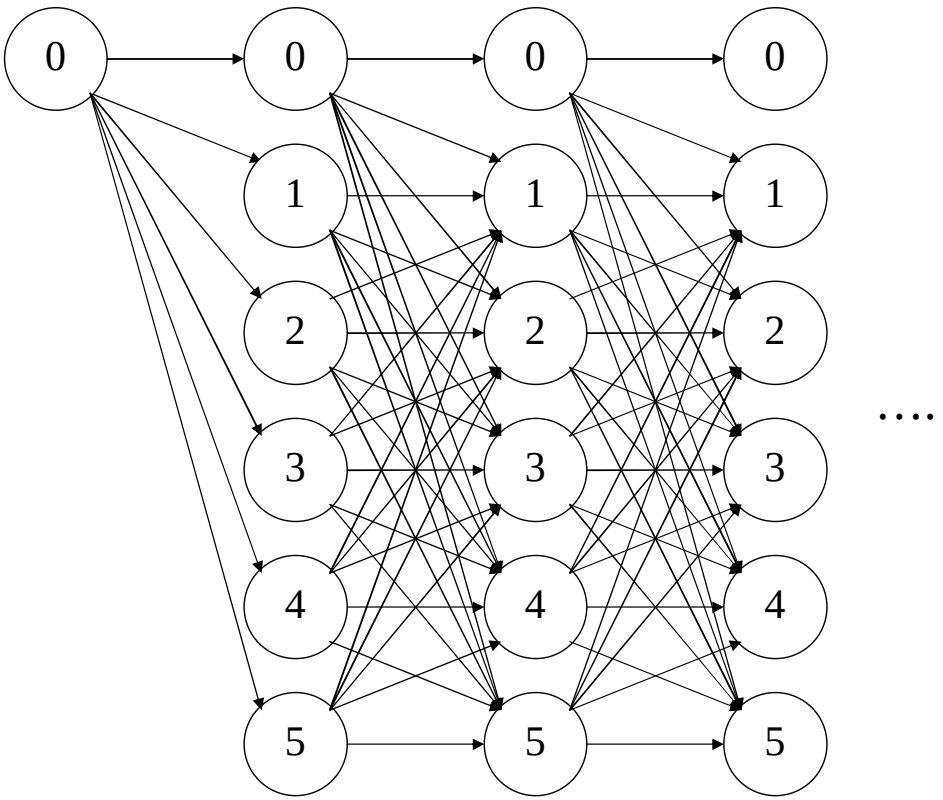
The digraph is not acyclic; its nodes cannot be sorted as in the previous example.

However no optimal path can take more than $n - 1$ arcs (in this case, 5 arcs).

The labels we compute at each extension are optimal for each number of arcs of the path.

Shortest path on a weighted digraph

We reformulate the problem on a directed, acyclic and layered graph, with n layers.



- **State:** (k, i) *[n. extensions, last reached node]*;
- **Initialization:** $c(0, 0) = 0$;
- **Extension:** $c(k, j) =$

  $= \min_{i \in \mathcal{N}} \{c(k-1, i) + w_{ij}\}$

  $\forall k \geq 1 \in \mathcal{L},\ \forall j \in \mathcal{N}.$

Bellman-Ford algorithm (1958)

The label correcting algorithm obtained in this way is known as the Bellman-Ford algorithm.

Original digraph: n nodes and m arcs.

- **Time complexity:** $O(mn)$, i.e. the number of arcs in the layered graph.
- **Space complexity:** $O(n^2)$, to represent the graph. For each node in $\mathcal{N}$ we need to store a cost and a predecessor.

No label can be considered permanent until:

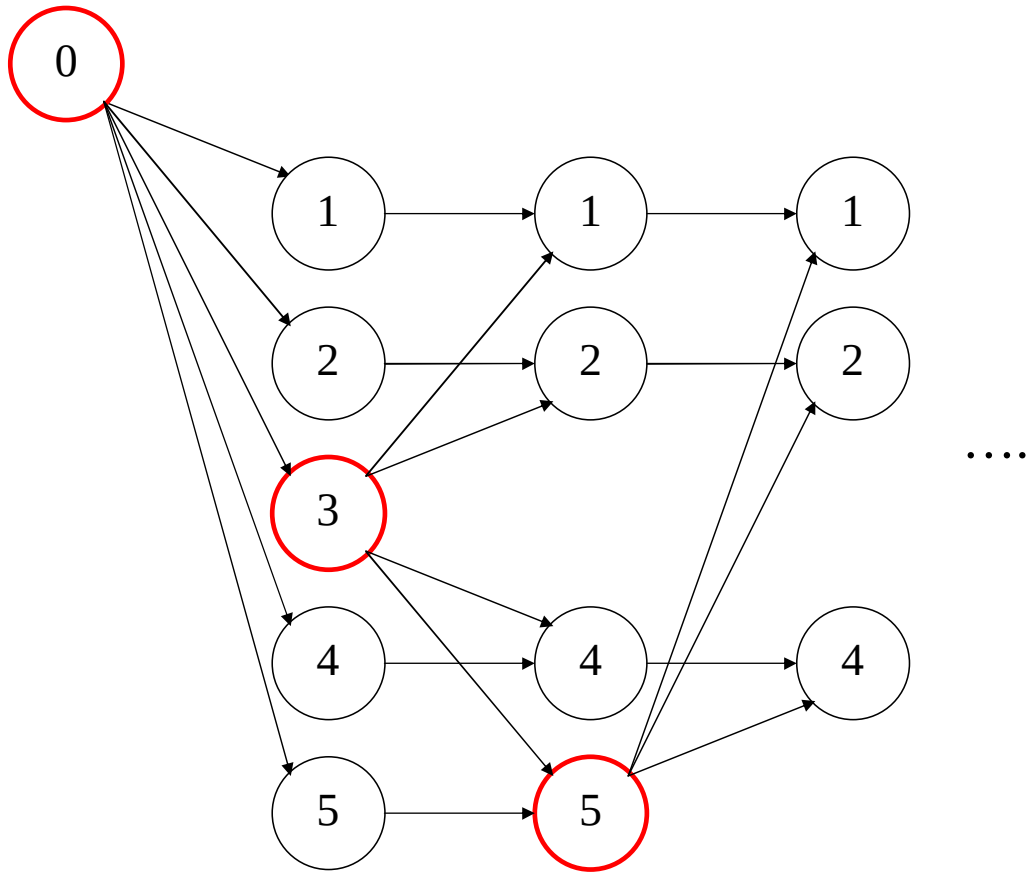
- either all layers have been labeled,
- or no label update has occurred from one layer to the next one.

Space complexity can be reduced to $O(n)$, because

- only one optimal predecessor must be stored for each node;
- only labels of layer $k-1$ are needed to compute those of layer k.

Dijkstra algorithm (1959)

Under the hypothesis that arc weights are non-negative, we can permanently set the label of minimum cost in each layer. The label setting algorithm obtained in this way is known as the Dijkstra algorithm.



Dijkstra algorithm (1959)

1. **State:** (k, i) *[layer, node]*;
2. **Initialization:** $c(0,0) = 0$; $last(0) = 0$; $Permanent(0) = \{0\}$;
3. **Extension:**
   - $c(k,j) = \min\{c(k-1,j), c(k-1, last(k-1)) + w(last(k-1), j)\}$ $\forall k \geq 1 \in \mathcal{L}, \forall j \notin Permanent(k-1)$;
   - $last(k) = \{\min_{j \notin Permanent(k-1)}\{c(k,j)\}\}$.
   - $Permanent(k) = Permanent(k-1) \cup \{last(k)\}$;

Every node has only two predecessors: the time complexity is $O(n^2)$.

Shortest trees and arborescences

Bellman-Ford's and Dijkstra's algorithms compute the shortest paths arborescence, that contains all shortest paths from the root node to all the others (for the optimality principle).

With a slight modification we can obtain Prim's algorithm to compute the minimum spanning tree on weighted (unoriented) graphs.

All these are examples of dynamic programming algorithms for polynomial-time combinatorial optimization problems.

Dijkstra and Prim algorithms

Algorithm 3 Dijkstra algorithm.

for $i \in \mathcal{N}$ **do**
  $c(i) \leftarrow \infty$
$c(s) \leftarrow 0$
$k \leftarrow s$
$P \leftarrow \{s\}$
while $|P| < |\mathcal{N}|$ **do**
  for $i \notin P$ **do**
    if $c(k) + w(k,i) < c(i)$ **then**
      $c(i) \leftarrow c(k) + w(k,i)$
      $\pi(i) \leftarrow k$
  $k \leftarrow \arg\min_{i \notin P}\{c(i)\}$
  $P \leftarrow P \cup \{k\}$

Algorithm 4 Prim algorithm.

for $i \in \mathcal{N}$ **do**
  $c(i) \leftarrow \infty$
$c(s) \leftarrow 0$
$k \leftarrow s$
$P \leftarrow \{s\}$
while $|P| < |\mathcal{N}|$ **do**
  for $i \notin P$ **do**
    if $w(k,i) < c(i)$ **then**
      $c(i) \leftarrow w(k,i)$
      $\pi(i) \leftarrow k$
  $k \leftarrow \arg\min_{i \notin P}\{c(i)\}$
  $P \leftarrow P \cup \{k\}$

Example: string matching

Given two sequences S_1 and S_2 of characters 'A' and 'B', insert blank characters in them in order to minimize the cost of the misalignments.

Instance:

- S_1 = A A A B A B
- S_2 = B A B A B B

Cost	A	B	blank
A	0	2	1
B	2	0	1
blank	1	1	-

For instance:

Solution 1:

- S_1 = A A A B A B
- S_2 = B A B A B B

Cost = 8

Solution 2:

- S_1 = A A A B A B _ _
- S_2 = B A _ _ B A B B

Cost = 4

Example: string matching

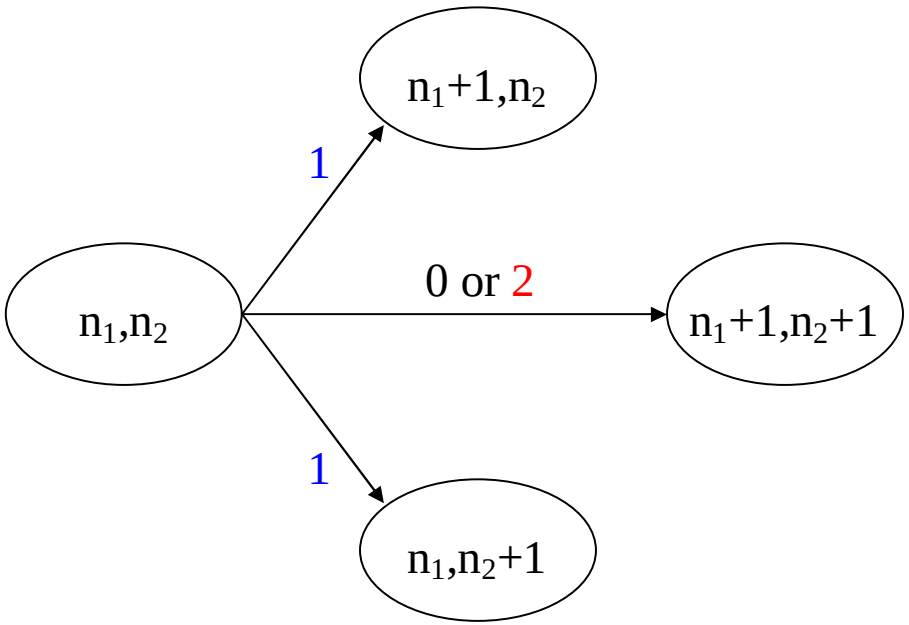
The problem is to decide how many blanks to insert in each of the $n+1$ positions of each sequence. A discrete choice must be made for each position: the number of distinct solutions is exponential in n.

Dynamic Programming:

1. **Sequence.** Align the sequences from left to right: a sub-policy is a partial alignment of the left part of S_1 with the left part of S_2.
2. **State.** All distinct ways of aligning the first n_1 characters of S_1 with the first n_2 characters of S_2 lead to the same state; the remaining decisions (their feasibility and their cost) do not depend on how the state has been reached.

Example: string matching

3. Extension is possible in three ways:



- Initialization: $c(0,0) = 0$.
- Extension:
  $c(n_1, n_2) = \min\{c(n_1 - 1, n_2) + w_{S_1(n_1), \text{“blank”}}, c(n_1, n_2 - 1) + w_{\text{“blank”}, S_2(n_2)}, c(n_1 - 1, n_2 - 1) + w_{S_1(n_1), S_2(n_2)}\}$.

where w is the cost function.

The optimal value is $c(|S_1|, |S_2|)$.

Example: string matching

Time complexity.

There are as many states as the n. of distinct pairs (n_1, n_2), i.e. n^2.

Each state has 3 predecessors: each label $c(n_1, n_2)$ can be computed in $O(1)$ time.

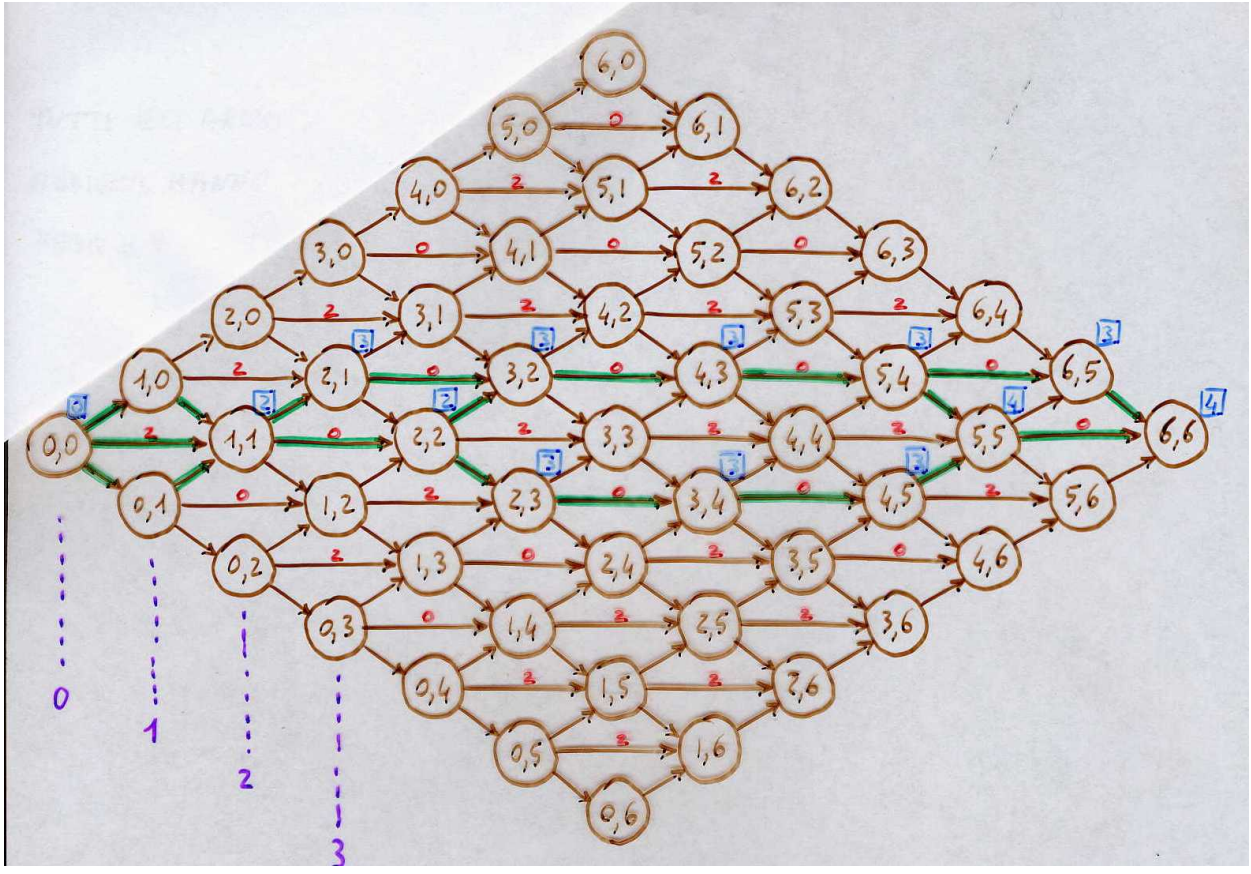
Then the D.P. algorithm has polynomial complexity $O(n^2)$.

Space complexity.

A cost label and a predecessor must be stored for each state: $O(n^2)$.

If the optimal solution is not needed, it can be reduced to $O(n)$, since only layers $k-2$ and $k-1$ are needed to compute labels of layer k.

Example: string matching



The graph is directed, acyclic and layered.
Each layer is reached only from the last two.

Spreadsheet implementation

Dynamic programming algorithms whose state-transitions graph is a matrix can be implemented in a spreadsheet.

The states-transitions graph is explored depth-first, owing to the extension function.

Branch-and-bound trees are usually visited depth-first or best-first, because a good primal bound is needed.

Bounding can be used in D.P. if

- a primal bound is known;
- dual bounds (completion bounds) can be (efficiently) computed for each state.

p-medians on a line: the problem

Given a straight line and a set $N = \{1, \ldots, n\}$ of points along it in given positions $x_i \; \forall i \in N$, find the optimal position along the line for p additional points, called "medians", such that the sum of the distances between each point in N and its closest median is minimized.

W.l.o.g. we assume that the points in N are numbered according to their order along the line.

Remark 1. The *p*-median problem is *NP*-hard on graphs.

Remark 2. The 1-median problem is polynomially solvable (even on general graphs).
n odd: central point;
n even: anywhere between the two central points.

Let $w(i, j)$ be the minimum cost for locating a single median to serve the points in the interval $[i, j]$, with $i \in N, \; j \in N, j \geq i$.

p-medians on a line: a D.P. algorithm

1. **Sequence the decisions.** All solutions induce a partition of N into non-overlapping intervals, such that all points within a same interval have the same closest median. If such partition is given, it is easy to optimally locate the median in each interval (see Remark 2). Hence, we scan the sequence of the points along the line and we decide how many medians are used to serve the points encountered.
2. **Define the state.** States: $\{i, m\}$, where $i \in N$ is the last scanned point; m is the number of medians used up to that point. Cost associated with each state: $c(i, m)$. It is the minimum cost to serve the points in $[1..i]$ with m medians.
3. **Label extension.**
   - $c(i, 1) = w(1, i) \ \ \forall i \in N$.
   - $c(i, m) = \min_{j<i}\{c(j, m-1) + w(j+1, i)\} \ \ \forall i \in \mathcal{N} : i \geq 2 \, \forall m : 2 \leq m \leq min\{i, p\}$.

The cost of the optimal solution is $c(n, p)$.

p-medians on a line: complexity

Time complexity.

N. of states: np, which is not larger than n^2, because $p \leq n$.

N. of predecessors for each state: $O(n)$.

Time complexity: $O(n^2 p)$, which is not worse than $O(n^3)$.

Space complexity.

We need to store a cost and a predecessor for each state: $O(np)$.
We also need to store a cost matrix w: $O(n^2)$.

p-medians on a line: an example

Figure 1 represents an instance of the *p*-median problem on a line.

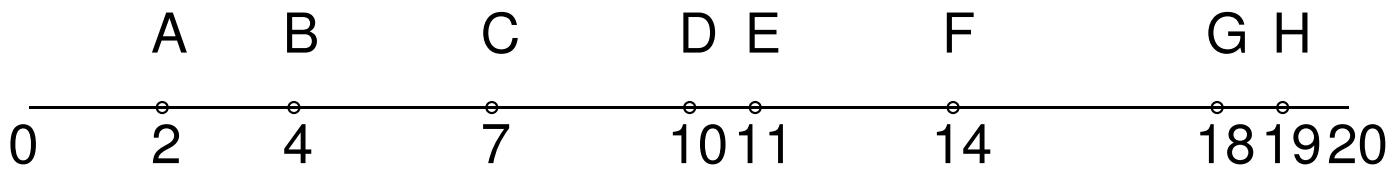


Figura: An instance of the *p*-median problem on a line. For better readability the indices $1, \ldots, n$ of the given points have been replaced by letters. In this instance $p = 3$.

p-medians on a line: computing *w*

The costs $w(i,j)$ for all pairs of points can be computed in $O(n^2)$ exploiting the property outlined in Remark 1 by the following algorithm, whose time complexity and space complexity is $O(n^2)$.

---

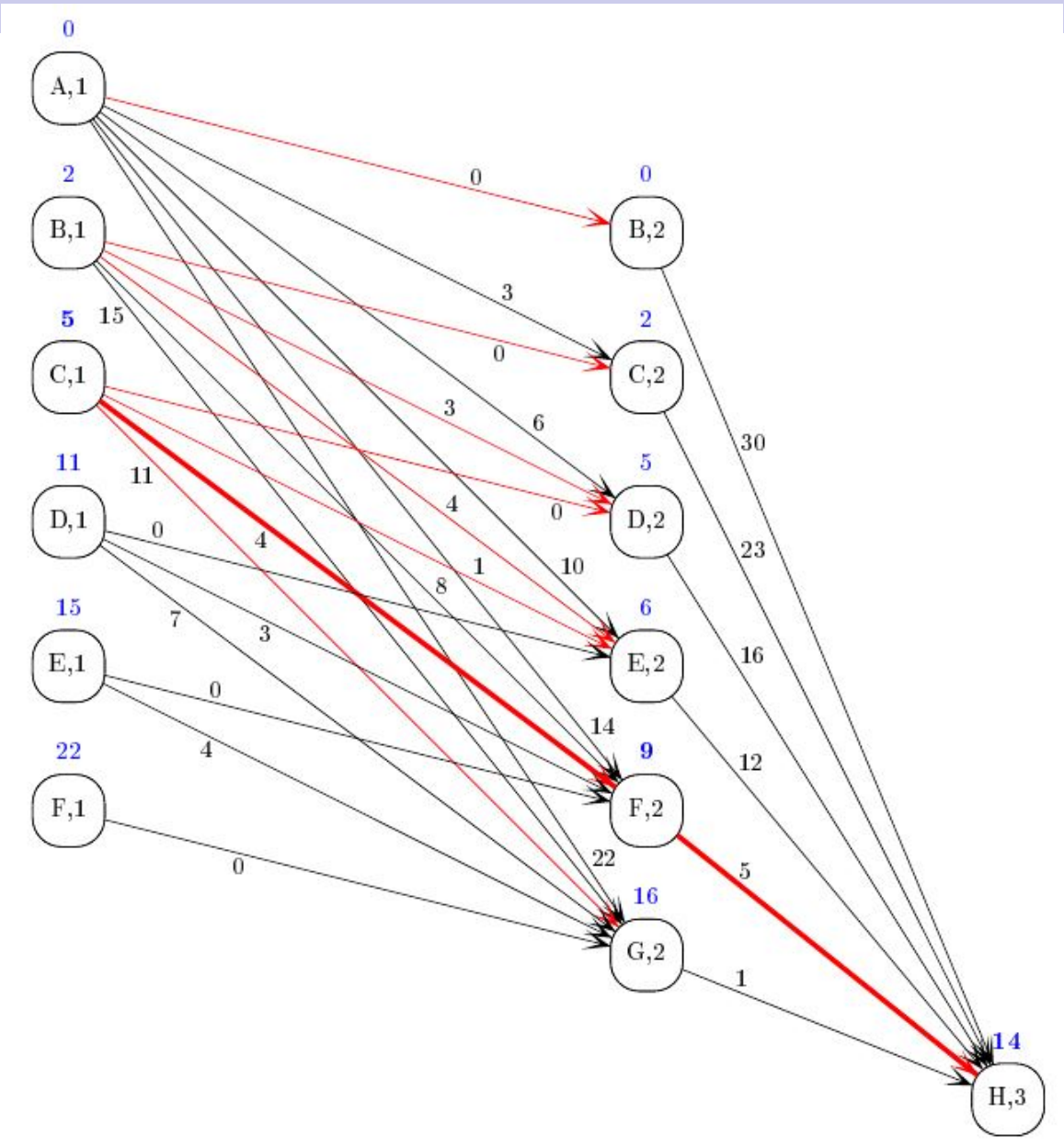
for $i = 1, \ldots, n-1$ **do**
  $opt \leftarrow i$;
  $j \leftarrow i$;
  $w(i,j) \leftarrow 0$;
  $parity \leftarrow 1$;
  while $i < n$ **do**
    $j \leftarrow j + 1$;
    $parity \leftarrow 1 - parity$;
    $w(i,j) \leftarrow w(i,j-1) + (x(j) - x(opt))$;
    if $(parity = 0)$ **then**
      $opt \leftarrow opt + 1$;

---

p-medians on a line: the cost matrix *w*

The resulting cost matrix *w* is as follows.

w	A	B	C	D	E	F	G	H
A	0	2	5	11	15	22	30	39
B		0	3	6	10	14	22	30
C			0	3	4	8	15	23
D				0	1	4	11	16
E					0	3	7	12
F						0	4	5
G							0	1
H								0


0
A,1
2
B,1
5
C,1
11
D,1
15
E,1
22
F,1
0
B,2
2
C,2
5
D,2
6
E,2
9
F,2
16
G,2
14
H,3
0
3
15
0
3
6
11
4
0
4
1
10
8
7
3
0
4
14
22
0
30
23
16
12
5
1

Observations

In a recursive algorithm based on

- $c(i,1) = w(1,i) \;\; \forall i \in N$
- $c(i,m) = \min_{j<i}\{c(j,m-1) + w(j+1,i)\} \;\; \forall i \in \mathcal{N} : i \geq 2 \; \forall m : 2 \leq m \leq min\{i,p\}$

the main program would have called $c(n,p)$ initially.

The value $c(i,m)$ for a same pair of parameters (i,m) would have been computed several times.

Pseudo-code

```
for i = 1, ..., n − p + 1 do
    c[i, 1] ← w[1, i];
for m = 2, ..., p do
    for i = m, ..., n − (p − m) do
        c[i, m] ← ∞
        for j = m − 1, ..., i − 1 do
            if c[j, m − 1] + w[j + 1, i] <
            c[i, m] then
                c[i, m] ← c[j, m − 1] +
                w[j + 1, i];
```

Dynamic programming

```
if m = 1 then
    return w[1, i];
else
    mincost ← ∞;
    for j = m − 1, ..., i − 1 do
        x ← c(j, m − 1);
        if x + w[j + 1, i] < mincost
        then
            mincost ← x + w[j + 1, i];
    return mincost;
```

Recursive algorithm $c(i, m)$ (branching).

Dynamic system optimal control

We are given a discrete-time dynamic system, i.e. a system characterized by an input, a state and an output.

In this simple example they consist of a scalar value.

In a discrete set of T points in time $t = 1, \ldots, T$ the state $x(t)$ evolves according to the equation

$$x(t) = x(t-1) + u(t)$$

where $u(t)$ is the input at time t.

The domains U and X of u and x are given discrete intervals.

A cost $f_t(x(t-1), u(t))$ is associated with each transition occurring from a state $x(t-1)$ with an input value $u(t)$ at time t.

The whole set of input values is to be decided and the initial state $x(0)$ as well.

The system must reach a given final state $\overline{x}$ by a sequence of transitions of minimum cost.

Dynamic programming algorithm

1. **Sequencing the decisions.** The sequence of decisions is the sequence of input values to be chosen: there is a decision for each $t \in 1, \ldots, T$.
2. **Defining the state.** The state in dynamic programming corresponds with the state of the dynamic system: $\{x, t\}$, where x is the state of the system and t indicates the point in time. The cost $c(x, t)$ is the minimum cost to reach state x at time t.
3. **Label extension.**
   - $c(x, 0) = 0 \;\; \forall x \in X$.
   - $c(x, t) = \max_{u \in U}\{c(x-u, t-1) + f_t(x-u, u)\} \;\; \forall x \in X \, \forall t \in 1, \ldots, T$.

The minimum cost is $c(\overline{x}, T)$.

Complexity

Time complexity.

The number of possible values of x is $|X|$.
The number of possible values of t is T.
Hence, the number of states grows as $O(|X|T)$.
N. of predecessors for each state: $|U|$.

Time complexity: $O(|X||U|T)$.

If $|X|$ and $|U|$ are given, then the complexity is *polynomial*.
If they are part of the input, the complexity is *pseudo-polynomial*.

Space complexity.

The number of states grows as $O(|X|T)$.
The data f grow as $O(|U|T)$.

Space complexity: $O(|X|T + |U|T)$.

An example

$X = \{1, 2, 3\}$, $U = \{-1, 0, 1\}$, $T = 3$.

	f_1			f_2			f_3		
	-1	0	1	-1	0	1	-1	0	1
1	(2)	-7	-3	(1)	-5	-9	(0)	-6	-4
2	0	5	-4	-2	-14	2	1	-1	0
3	3	-10	(-3)	15	20	(-8)	4	-9	(-2)

Tabella: Transition costs. Rows: x; columns: u.

Final state: $\overline{x} = 2$ at $t = 3$.

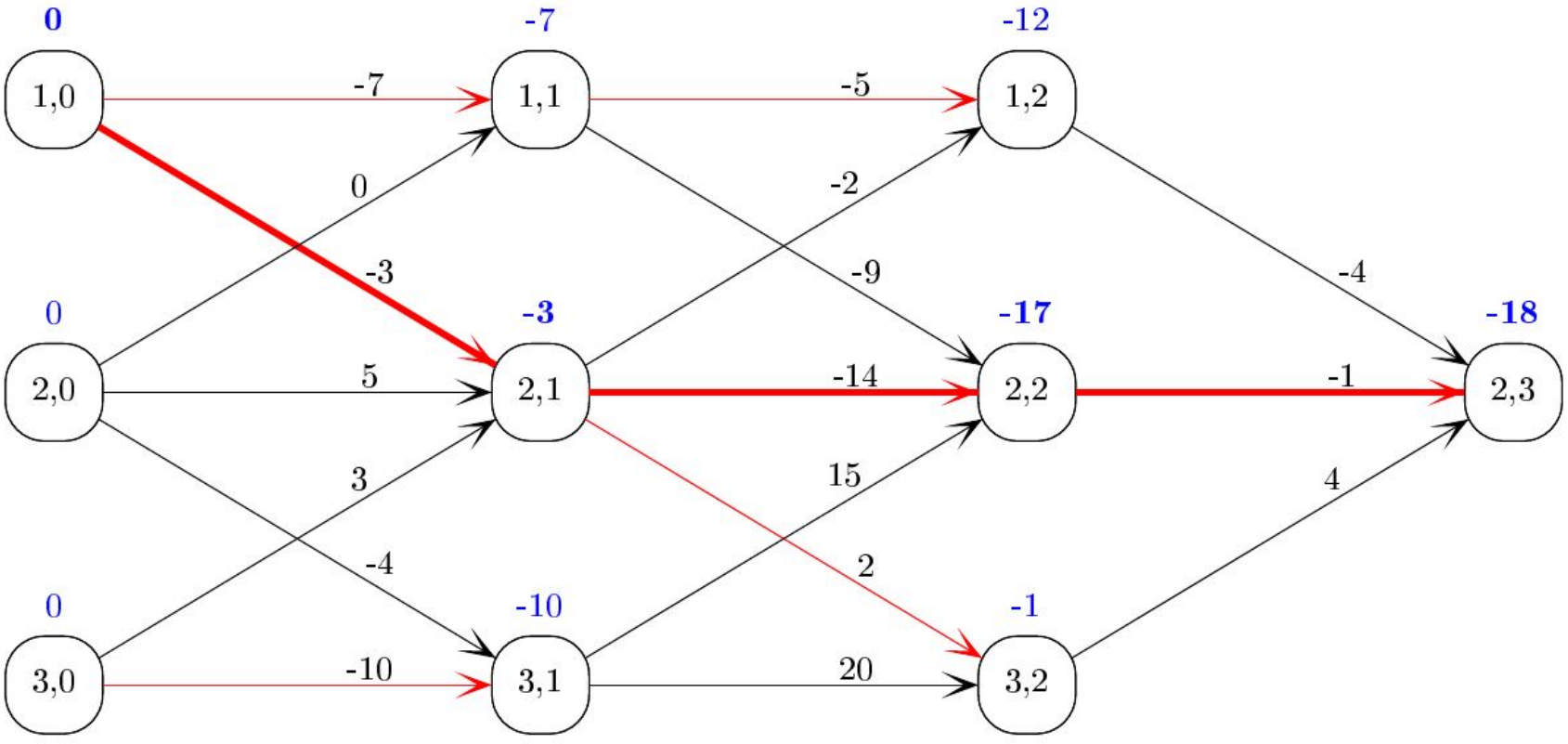


Figura: The state-transition graph and the optimal solution. Costs are represented in blue; optimal predecessors are represented in red. The optimal solution is indicated by thick arcs and bolded numbers.

Optimal budget allocation

We are given a set $P = \{1, \ldots, n\}$ of projects and a budget R.

We have to assign an investment x_i to each project $i \in P$.

Depending on the investment x_i, each project i is expected to yield a profit $f_i(x_i)$.
No assumption is made about the kind of relationship between the investment and the profit. For simplicity, we assume that the investments are integer and non-negative.

The domain of x_i is indicated by X_i for ech $i \in P$.

Objective: maximize the overall expected profit, without exceeding the budget.

Dynamic programming

1. **Sequencing the decisions.** Arbitrarily sort the projects. A decision must be taken for each project $i \in P$.
2. **Defining the state.** Each decision consumes a certain amount of a limited resource.
   States have the form $\{i, r\}$, where i is the last project considered and r is the residual available budget.
   Profit associated with each state: $p(i, r)$.
   A state $\{0, R\}$ corresponds to the beginning of the decision process.
3. **Label extension.**
   - $p(0, R) = 0$.
   - $p(i, r) = \max_{x_i \in X_i}\{p(i-1, r+x_i) + f_i(x_i)\} \;\; \forall i \in P \; \forall r \in 0, \ldots, R$.

The maximum expected profit is $\max_{r=0}^{R}\{p(n, r)\}$.

Remark. If $f_i(x_i) \geq x_i \; \forall i \in P \; \forall x_i \in X_i$ and $R \leq \sum_{i \in P} \max_{x_i \in X_i}\{x_i\}$, then the maximum expected profit is $p(n, 0)$, because it is optimal to invest the whole budget.

Complexity

Time complexity.
The n. of possible values of i is n.
The n. of possible values of r is $R+1$ (from 0 to R).
Hence, the n. of states is nR.
The n. of predecessors for each state is at most $|X_i|$, which is not larger than $R+1$.

Time complexity: $O(nR^2)$ (*pseudo-polynomial*).

Space complexity.
For profits and predecessors of states: $O(nR)$.
For data: $O(nR)$, since $|X_i| \leq R+1 \ \forall i \in N$.

Space complexity: $O(nR)$ (*pseudo-polynomial*).

An example

$P = \{1, \ldots, 4\}$ and $R = 10$.

x_1	f_1	x_2	f_2	x_3	f_3	x_4	f_4
0	0	0	-2	0	0	0	-5
1	7	1	4	1	5	1	-2
2	13	2	7	2	6	2	3
3	17	3	8	3	7	3	7
4	20			4	8		

Tabella: The possible investments and the corresponding expected profits for each project.

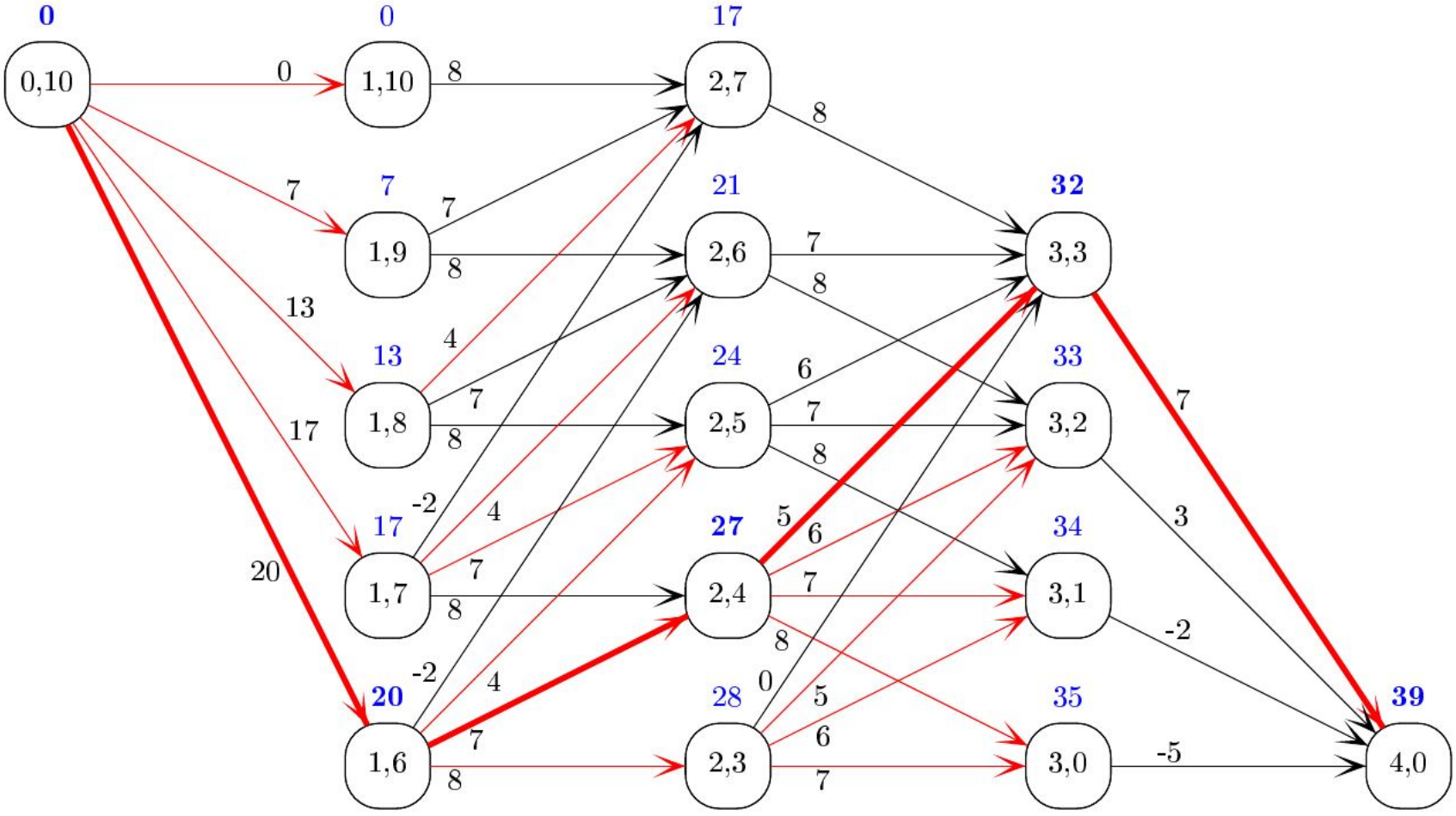


Figura: The state-transition graph and the optimal solution. Costs are represented in blue; optimal predecessors are represented in red. The optimal solution is indicated by thick arcs and bolded numbers.

The max independent set problem on an interval graph

The owner of an AirB&B apartment has collected a set N of n reservation requests from potential customers; she must decide which requests to accept in order to maximize her profits.

Each request $i \in N$ has a check-in day s_i, a check-out day e_i and a profit p_i.

Obviously, no two accepted requests can overlap in time.

The max independent set problem on an interval graph

Remark. In graph terminology this problem is called *Max independent set problem on an interval graph*.

Define a graph where

- each vertex corresponds to a request
- two vertices i and j are connected by an edge, if and only if the two corresponding requests overlap.

Subsets of compatible requests correspond to independent sets, i.e. a subset of vertices not connected to one another by any edge.

The subset of requests yielding the maximum profit corresponds to a maximum weight independent set, after assigning each vertex i a weight equal to the profit p_i of the corresponding request.

The max independent set problem is *NP*-hard on general graphs, but it is polynomially solvable on interval graphs.

Dynamic programming

1. **Sequencing the decisions.** Sort the requests according to their check-in date s.
   A binary decision must be taken for each of them (whether to accept it or not).
2. **Defining the state.** At each stage we only need to know the next request that must be considered, $i \in N$.
   Profit associated with each state $\{i\}$: $f(i)$.
   A dummy state $\{n+1\}$ represents the final state, when all requests have been decided: s_{n+1} is set to a value larger than $\max_{i \in N}\{e_i\}$.
3. **Label extension.**
   - $f(1) := 0$.
   - $f(i) := \max\{0, \max_{j \in N: e_j \leq s_i}\{f(j) + p_j\}\} \quad \forall i = 1, \ldots, n+1$.

The maximum profit is $f(n+1)$.

Complexity

Time complexity.
The n. of possible values of i is $n+1$.
The n. of predecessors for each state is at most n.

Time complexity: $O(n^2)$ (*polynomial*).

Improvement. Assign each state j a unique successor $succ(j)$ such that

$$succ(j) = \text{argmin}_{iinN:s_i \geq e_j}\{s_i\}.$$

From *succ* we can construct a graph in which each state j has only two outgoing arcs:

- an arc with value 0 to the next state $j+1$ (refuse j),
- an arc of value p_j to $succ(j)$ (accept j).

The resulting graph is directed, acyclic and layered and has only $O(n)$ arcs: labelling its nodes takes $O(n)$.

Constructing the graph takes $O(n)$ after sorting the requests (in $O(n \log n)$).

An example

i	s_i	e_i	p_i
1	4	15	11
2	18	42	24
3	40	45	5
4	4	33	29
5	7	29	22
6	3	9	6
7	21	30	9

Tabella: The list of the requests.

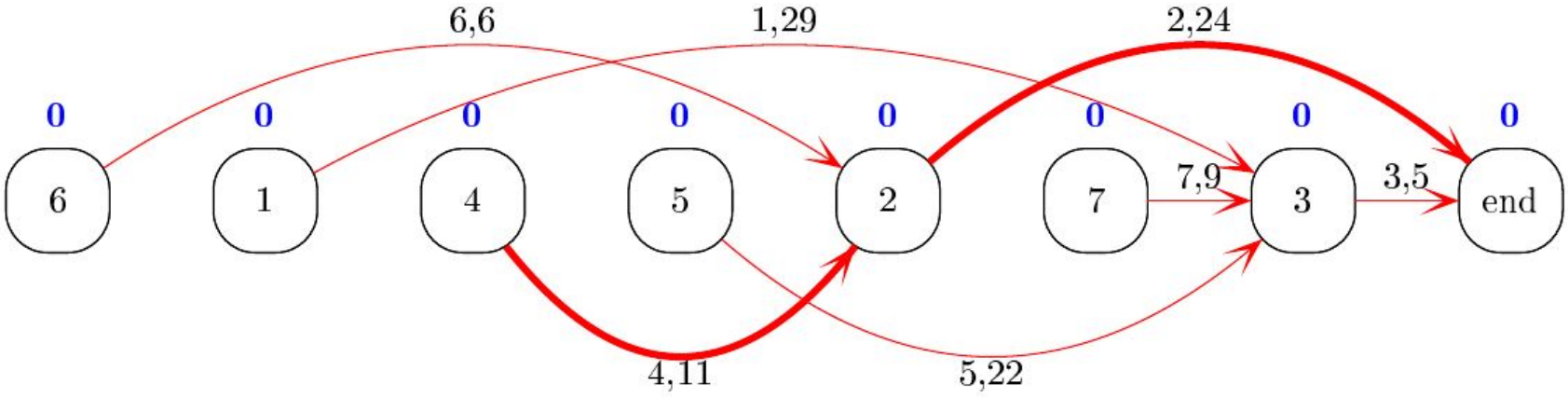


Figura: The state-transition graph and the state extensions. Costs are represented in blue; optimal predecessors are represented in red. The optimal solution is indicated by thick arcs and bolded numbers.