Dynamic programming

Giovanni Righini

Dipartimento di Informatica
Polo Didattico e di Ricerca di Crema
Università degli Studi di Milano
Via Bramante 65, 26013 Crema, Italy

1 Introduction

In these notes I present some examples of discrete optimization problems solved by dynamic programming (D.P. for short). The basic steps in the design of a D.P. algorithm are the same in all examples:

1. Define a sequence of decisions, that corresponds to determining a solution.
2. Define the state. The state is the amount of information that one needs to know when some decisions in the sequence have already been taken and others are still to be taken. The information in the state must be enough to determine the feasibility and the cost of the remaining decisions.
3. Define a recursive extension function, i.e. how the cost of states can be computed from the cost of other states.

The execution of a D.P. algorithm resembles the search for an optimal path on an acyclic and weighted digraph, from a given source node correspondig to an empty solution (no decision taken) to a given target node corresponding to a complete solution (all decisions taken). The structure and the size of the digraph depend on the problem at hand and they determine the complexity of the D.P. algorithm. When the D.P. algorithm terminates, the label (cost to be minimized or value to be maximized) of the final state gives the optimal value of the objective function.

Besides computing the optimal value, one usually wants to reconstruct a feasible solution with that value, i.e. an optimal solution. This is achieved by scanning the sequence of the decisions backward, from the final state to the initial state. For each state its optimal predecessor is selected and for this purpose it is required that the optimal predecessor has been stored for each state.

For each example, I will provide the description of the problem, the description of one or more D.P. algorithms, the corresponding acyclic weighted digraphs, the complexity of the algorithm. Each problem is illustrated by a numerical example with its solution. I also provide the pseudo-code of the D.P. algorithm allowing for an efficient and straightforward implementation.

Refinements such as bi-directional D.P. and the use of suitable data-structures will be also discussed.

2 The shortest path problem on a weighted acyclic digraph

The problem. Given an acyclic digraph $\mathcal{D} = (\mathcal{N}, \mathcal{A})$ with $|\mathcal{N}| = n$ and $|\mathcal{A}| = m$ and a cost function $w : \mathcal{A} \mapsto \Re$, find the shortest path from a given node $s \in \mathcal{N}$ to a given node $t \in \mathcal{N}$. A digraph is acyclic if and only if it does not contain any circuit.

Step 1: sequencing the decisions. Since $\mathcal{D}$ is acyclic, it is possible to sort its nodes in topological order in $O(m)$. If $\mathcal{D}$ is also layered, it is possible to sort the layers and it is not necessary to sort the nodes in each layer (any order fits). We indicate by $Pred(j) \subset \mathcal{N}$ the set of predecessors of each node $j \in \mathcal{N}$:

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

Step 2: defining the state. The state consists of the last reached node. All paths from node s to node $i \in \mathcal{N}$ correspond to sub-policies leading to the same state. Hence states have the following (trivial) form: $\{i\}$, with $i \in \mathcal{N}$. The cost associated with each state $\{i\}$ is indicated by $c(i)$.

Step 3: state extension.

- Initialization: $c(s) := 0$.
- Extension: $c(j) := \min_{i \in Pred(j)}\{c(i) + w_{ij}\}$.

The optimal value is $c(t)$ when the algorithm stops.

Complexity. The acyclic weighted state-transition digraph of the D.P. algorithm is the digraph $\mathcal{D}$ itself. This is why this example is suitbale to be presented as the introductory problem to illustrate D.P..

The time complexity immediately follows: when extending states, each arc of the digraph is considered only once; hence the complexity is $O(m)$.

Since each node has a single *label* $c(i)$ and its value is computed only once, the resulting D.P. algorithm is a *label setting* algorithm.

Pseudo-code. The pseudo-code of the D.P. algorithm is shown in Algorithm 2.1. A vector π records the optimal predecessor state for each state, i.e. the optimal predecessor node for each node in this example.

The sub-routine $ComputePredecessors$ is needed when the digraph is given in input as a list of weighted arcs. In such a case the list is sequentially scanned and for each arc $(i, j) \in A$ node i is inserted in $Pred(j)$. The computational complexity of this operation is obviously $O(m)$.

The computational complexity of the label extension is $O(m)$, as already proven.

The computational complexity of the last part, where an optimal solution is produced, depends on the required output format: if a binary vector x is required, then the complexity is $O(m)$, as shown in Algorithm 2.1, because m binary variables must be assigned a value; if a set X of selected arcs is required, then the complexity is $O(n)$, because the initialization $X \leftarrow \emptyset$ takes constant time and no more than $n - 1$ arcs can belong to the $s - t$ path.

Algorithm 2.1 Shortest path problem on a weighted acyclic digraph

```
1:  TopologicalSort;                                         ▷ O(m)
2:  ComputePredecessors;                                     ▷ O(m)
3:  /* Neglect unreachable nodes (nodes before s), if any */
4:  for j = 0, ..., s − 1 do                                 ▷ O(n)
5:      c(j) ← ∞;
6:  /* Initialization */
7:  c(s) ← 0;
8:  /* Extension */
9:  for j = s + 1, ..., t do
10:     for i ∈ Pred(j) do
11:         if (c(i) + w_ij < c(j)) then
12:             c(j) ← c(i) + w_ij;
13:             π(j) ← i;
14: /* Optimal objective function value */
15: z* ← c(t);
16: /* Retrieval of an optimal solution */
17: for (i, j) ∈ A do                                        ▷ O(m)
18:     x*(i, j) ← 0;
19: j ← t;
20: while (j > s) do                                         ▷ O(n)
21:     x*(π(j), j) ← 1;
22:     j ← π(j);
    return z*, x*
```

Label correcting variation. A variation of the D.P. algorithm is obtained by extending the labels from each state to its successors. Let $Succ(i)$ be the set of all successors of node i, i.e.

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

The label extension part of Algorithm 2.1 could be replaced by the following Algorithm 2.2.

Algorithm 2.2 Label extension

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

The time complexity is the same, i.e. $O(m)$, because every arc is examined once. However, in this case the label of each state can be updated several times. Therefore this variation is a *label correcting* algorithm.

A numerical example. Consider the acyclic weighted digraph shown in Figure 1. The problem instance requires to compute a shortest path from node $s = 0$ to node $t = 9$.

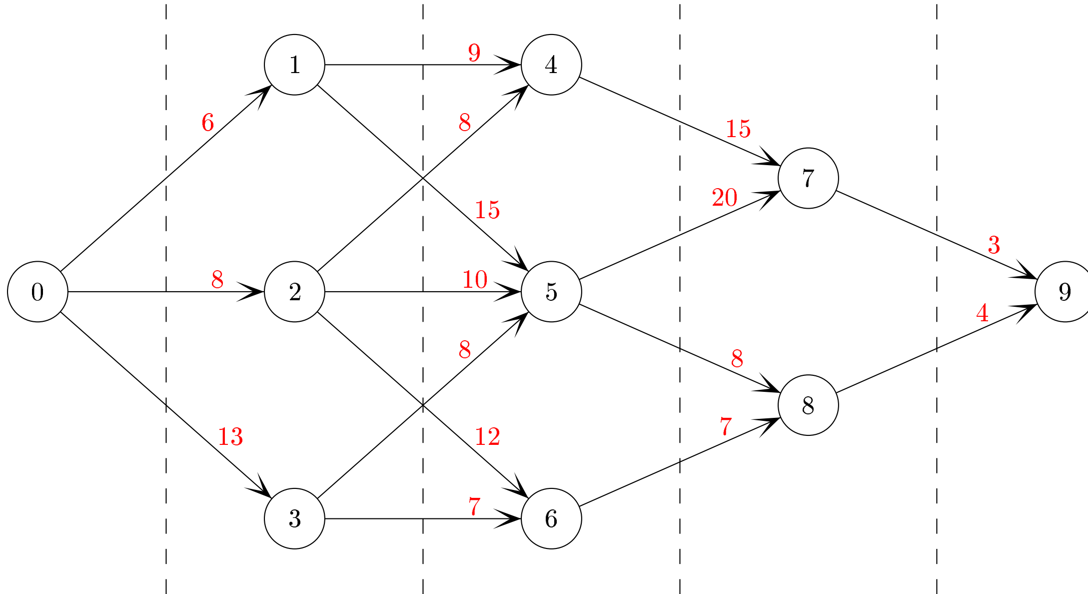

Figure 1: A weighted acyclic (and layered) digraph.

Figure 2 shows the iteration in which node 5 is labeled. Nodes 0 to 4 have already been labeled. Their associated costs are shown by the blue labels in Figure 2. Optimal predecessors are represented by red arcs. Node 5 is now labeled by comparing three predecessor states, yielding costs equal to 21 (from node 1), 18 (from node 2) and 21 (from node 3) and selecting the best option (the second one).

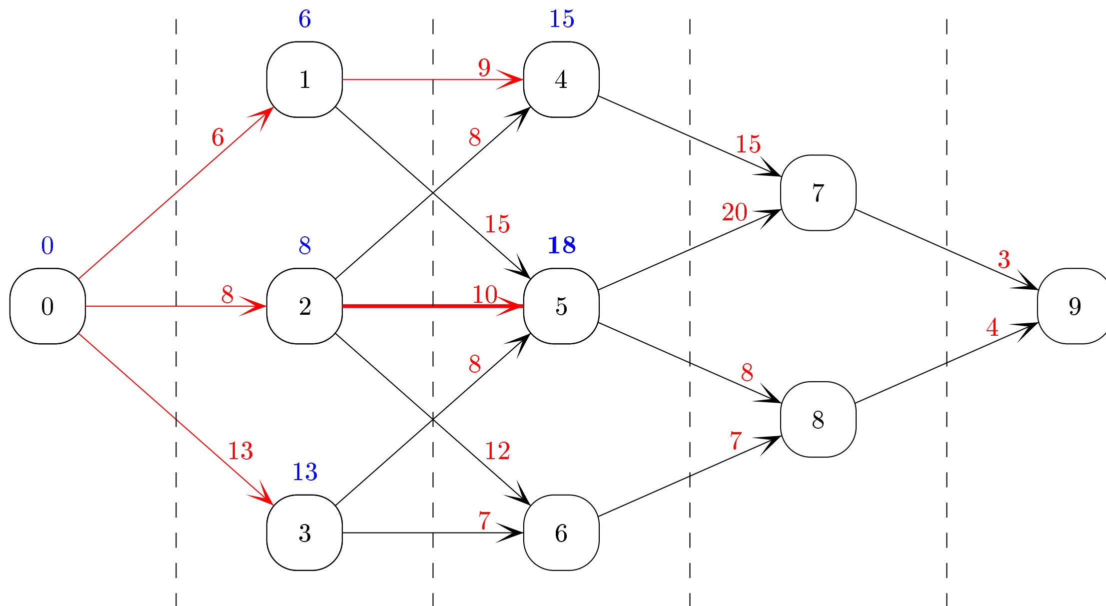

Figure 2: A state extension.

Figure 3 shows the solution. The optimal value is 30 and the corresponding optimal solution can be reconstructed by following the chain of predecessors π backward from node 9 to node 1: the result is represented by the thick red arcs.

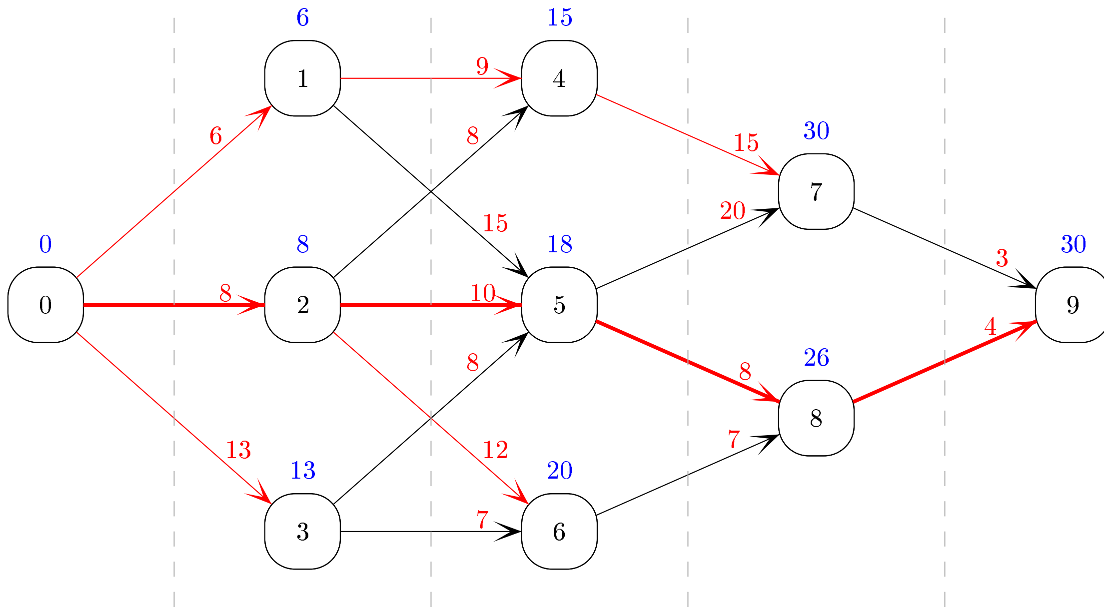

Figure 3: The example solved.

3 The shortest path problem on a weighted digraph

The problem. Given a digraph $\mathcal{D} = (\mathcal{N}, \mathcal{A})$ with $|\mathcal{N}| = n$ and $|\mathcal{A}| = m$ and a cost function $w : \mathcal{A} \mapsto \Re$, find the shortest path from a given node $s \in \mathcal{N}$ to a given node $t \in \mathcal{N}$. The digraph is not constrained to be acyclic as in the previous example; we consider now a generic digraph, possibly containing circuits. However circuits are guaranteed not to have negative cost.

Step 1: sequencing the decisions. The problem is very similar to the previous one, but now the digraph is not acyclic and therefore its nodes cannot be sorted as in the previous example. We can reformulate the problem on an acyclic and layered digraph $\mathcal{D}' = (\mathcal{N}', \mathcal{A}')$, where

- $\mathcal{N}'$ is made by a set $\mathcal{L}$ of n layers;
- each layer $\mathcal{L}_k$ contains a copy of each node in $\mathcal{N}$; hence each node in $\mathcal{N}'$ is indicated by a pair (i, k);
- for each arc $(i, j) \in \mathcal{A}$ there is an arc from node (i, k) to node $(j, k + 1)$ for each layer $k = 1, \dots, n - 1$.

We indicate by $Pred'(j, k) \subseteq \mathcal{N}'$ the set of predecessors of each node $(j, k) \in \mathcal{N}'$:

$$Pred'(j,k) = \{(i, k-1) \in \mathcal{N}' : (i,j) \in \mathcal{A}'\} \quad \forall k = 2, \dots, n.$$

The predecessors of each node in layer k belong to layer $k - 1$. The nodes in layer 1 have no predecessor.

The n layers of digraph $\mathcal{D}'$ represent the n stages of the D.P. algorithm. At each stage k the algorithm computes and compares paths made by $k - 1$ arcs. Since no feasible solution can contain more than $n - 1$ arcs, n layers are sufficient to implicitly enumerate all possible solutions.

After this reformulation, the shortest problem on a generic digraph with n nodes is translated into the shortest path problem on an acyclic digraph with $n(n-1)$ nodes. Therefore, the same D.P. algorithm shown in the previous example applies.

Step 2: defining the state. The state consists of the last reached node in $\mathcal{N}'$. All $s - i$ paths made by k arcs correspond to sub-policies leading to the same state. Hence labels have the following form: $\{i, k\}$, with $i \in \mathcal{N}$ and $k = 1, \dots, n$. The cost associated with each label $\{i, k\}$ is indicated by $c(i, k)$.

Step 3: label extension.

- Initialization: $c(s, 1) = 0 \quad c(i, 1) = \infty \; \forall i \neq s$.
- Extension: $c(j, k) = \min_{(i,k-1) \in Pred'(j,k)} \{c(i, k - 1) + w_{ij}, c(j, k - 1)\}$.

The optimal value is $c(t, n)$.

Remark 1. The algorithm can possibly terminate even before reaching stage n: if no change in labels is observed at a certain stage, no change can occur in the remaining stages either.

Remark 2. The algorithm computes the shortest path from s to *all* the other nodes in $\mathcal{N}$.

Complexity. When extending labels, each arc in $\mathcal{A}'$ is considered only once. The number of arcs in $\mathcal{A}'$ is $m(n - 1)$. Hence the complexity is $O(mn)$.

The algorithm is known as *Bellman-Ford algorithm*.

The label of a same node can be updated several times (once for each stage). For this reason the Bellman-Ford algorithm is a *label correcting* algorithm.

A numerical example. Figure 4 shows a weighted digraph containing circuits. We want to compute the shortest path from node 1 to node 6.

The solution process and the optimal solution are illustrated in Figure 5.

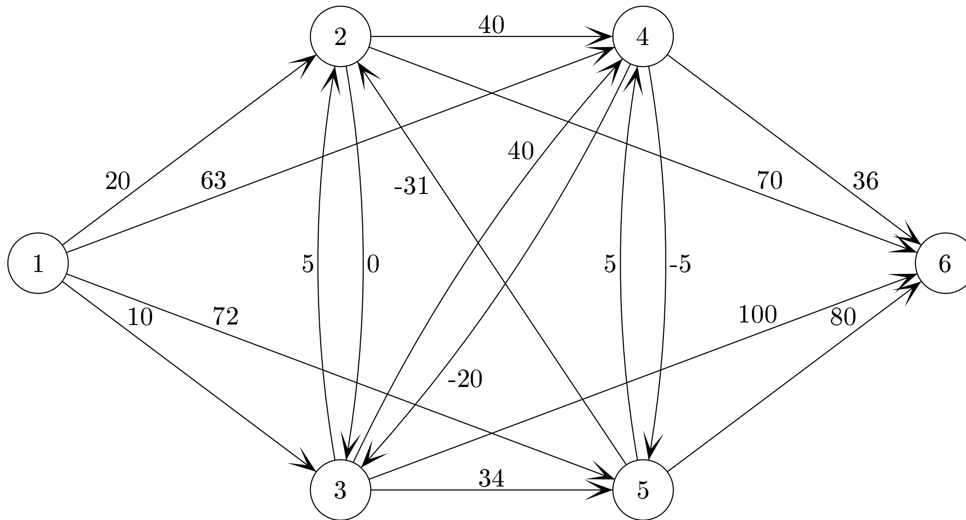

Figure 4: A weighted digraph.

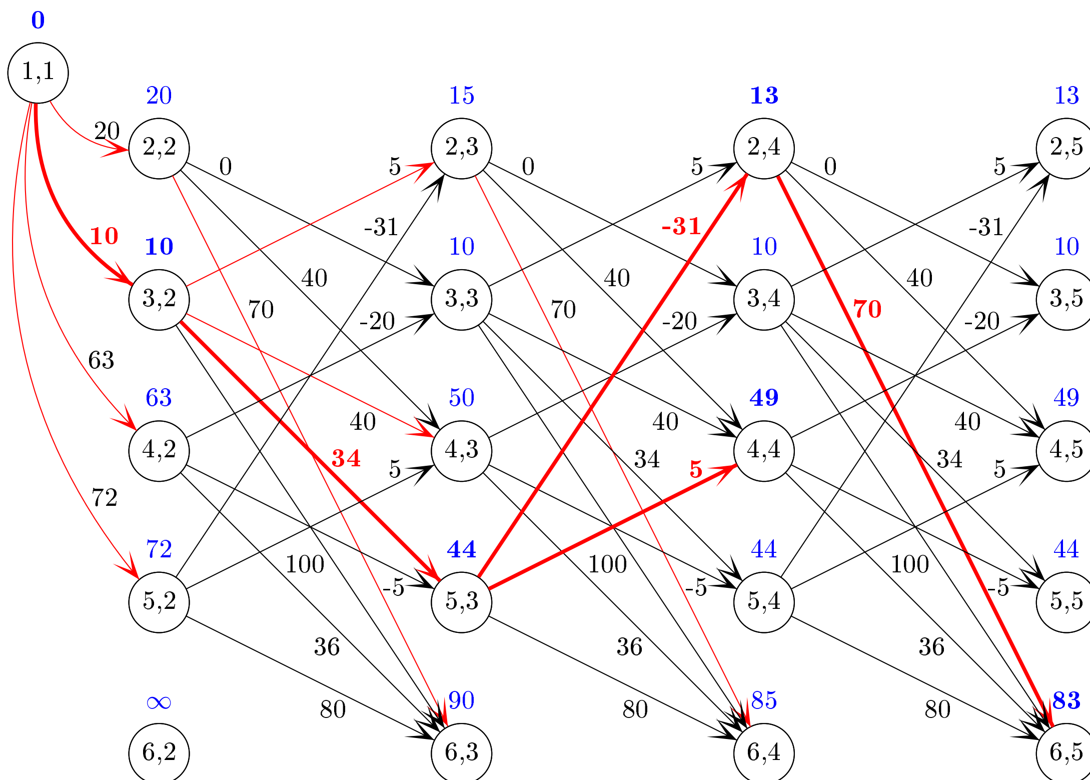

Figure 5: The state-transition graph and the state extensions. Costs are represented in blue; optimal predecessors are represented in red. The optimal solution is indicated by thick arcs and bolded numbers.

4 The shortest Hamiltonian path problem on a weighted digraph

The problem. Given a digraph $\mathcal{D} = (\mathcal{N}, \mathcal{A})$ with $|\mathcal{N}| = n$ and $|\mathcal{A}| = m$ and a cost function $w : \mathcal{A} \mapsto \Re$, find the shortest Hamiltonian path from a given node $s \in \mathcal{N}$ to a given node $t \in \mathcal{N}$.
A Hamiltonian path is a path that visits all nodes of the digraph once. Note that w.l.o.g. we can assume $\mathcal{A}$ to be complete; just consider missing arcs as arcs with a very large cost.

Step 1: sequencing the decisions. Similarly to the previous example, since $\mathcal{D}$ is not acyclic, it is not possible to sort its nodes and to set their labels permanently. We resort to an auxiliary acyclic and layered digraph $\mathcal{D}'$ using the same construction of the previous example.

Step 2: defining the state. The state does not consist only of the last reached node. In order to check the feasibility of the solution one must know which nodes have already been visited. All paths from node $(s, 1)$ to node $(i, k) \in \mathcal{N}'$ correspond to sub-policies leading to the same state if and only if they also visit the same subset of nodes. Hence labels have the following form: $\{i, k, S\}$, with $i \in \mathcal{N}$ and $S \subseteq \mathcal{N}$. Note that $k = |S|$, because one additional node is visited every time a path is extended. Therefore the information given by the layer index k is redundant and can be omitted. The cost associated with each label $\{i, S\}$ is indicated by $c(i, S)$.

Step 3: label extension.

- Initialization: $c(s, \{s\}) = 0$.
- Extension: $c(j, S) = \min_{i \in S}\{c(i, S \backslash \{j\}) + w_{ij}\}$.

The optimal value is $c(t, \mathcal{N})$.

Complexity. The number of states to be labeled is exponential in n. The number of distinct values of the possible subsets S is 2^n and each subset with k nodes appears in k different states (depending on which is the last visited node). Hence the number of states is $O(n2^n)$. When extending labels the number of predecessors for each state is $O(n)$. Hence the complexity of the D.P. algorithm is $O(n^2 2^n)$.

Dominance. Dominance consists of deleting some states from further consideration because there is a guarantee that they do not correspond to optimal sub-policies and therefore the corresponding partial solutions cannot be part of optimal solutions. In these examples the dominance criteria are implicitly applied when the label obtained from the best predecessor is selected and all other extensions (i.e. the labels produced by the other predecessors) are discarded. This is a special case of dominance: a state dominates another one when it has a smaller cost and all information is identical in both states (same last visited node, same layer, same subset of visited nodes,...).

A numerical example.

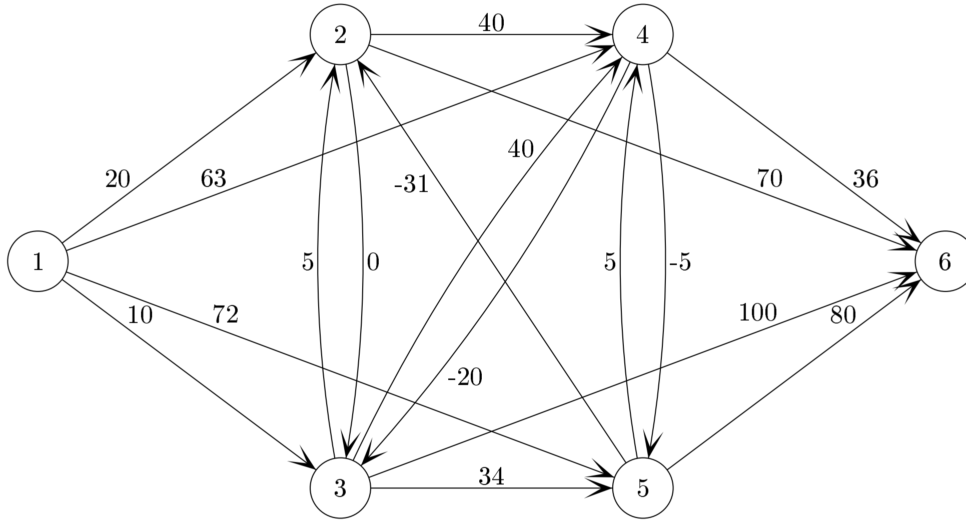

Figure 6: A weighted digraph. We want to find the minimum cost Hamiltonian path from node 1 to node 6.

Note that the digraph contains arcs with negative cost. However it does not contain circuits of negative cost. Circuits of zero cost are allowed and indeed there is one, containing nodes 4 and 5. Also note that not all arcs are present. This makes the instance easier to solve and the corresponding state-transition graph easier to represent: some states have a unique predecessor (see Figure 7).

The optimal solution is the path $(1, 3, 4, 5, 2, 6)$, whose cost is 84. Dominance can be observed on the right part of Figure 7, where some states have more than one predecessor: this means that they can be reached in different ways. For instance, state $\{5, 1345\}$ can be reached through the sequences $(1, 4, 3, 5)$ with cost 77 and $(1, 3, 4, 5)$ with cost 45. The former sub-policy is dominated by the latter one. Owing to dominance, the states generated by the D.P. algorithm are fewer than all possible sequences (sub-policies).

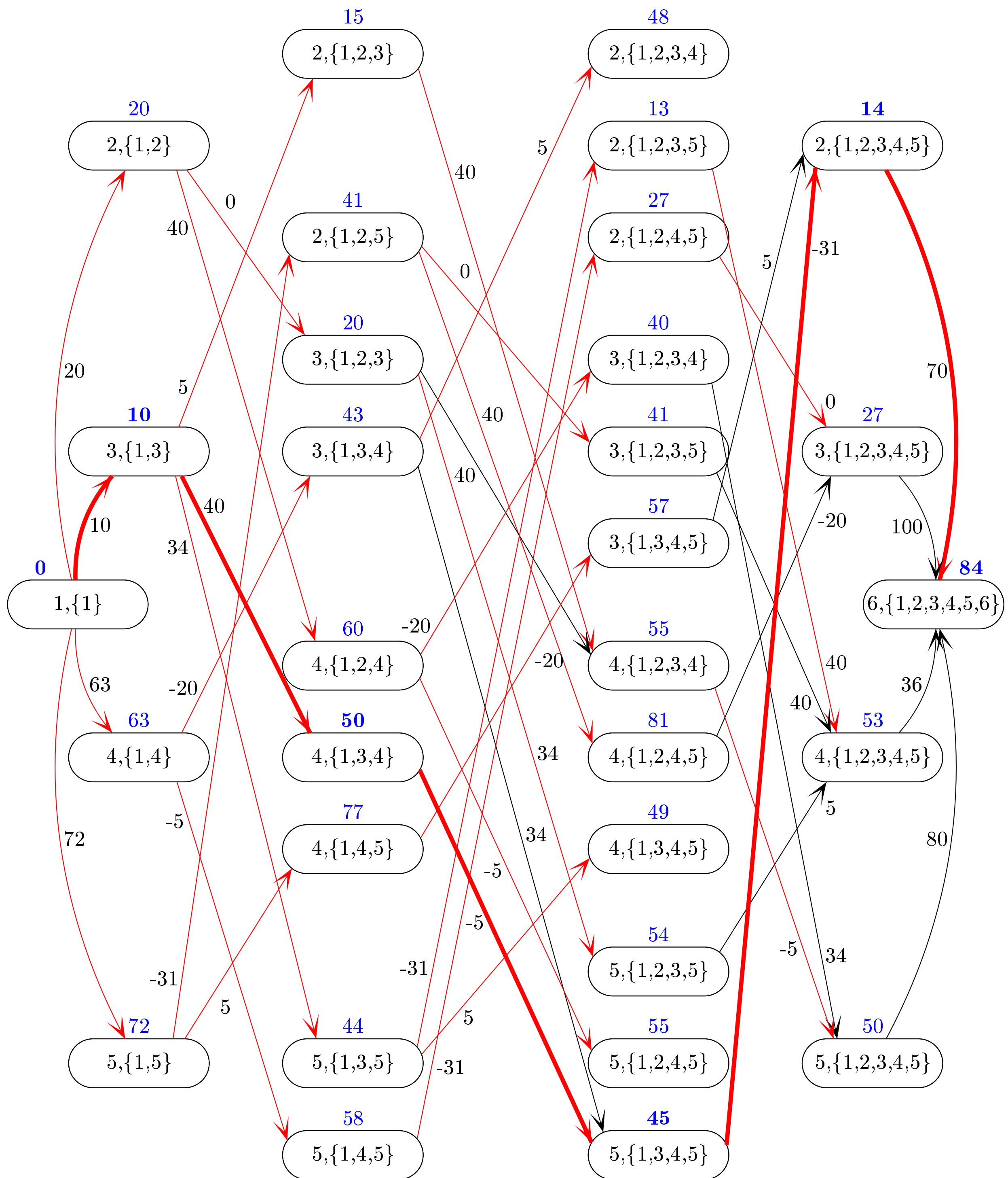

Figure 7: The state-transition graph and the state extensions. Costs are represented in blue; optimal predecessors are represented in red. The optimal solution is indicated by thick arcs and bolded numbers.

5 String matching problem

The problem. Given two strings S_1 and S_2, i.e. two sequences of characters taken from a given alphabet $\mathcal{A}$, given an additional character "X" not occurring in S_1 and S_2 and given a cost function $w : \mathcal{A}^+ \times \mathcal{A}^+ \mapsto \Re$, where $\mathcal{A}^+ = \mathcal{A} \cup \{``X''\}$, find the minimum cost alignment.

An alignment is given by inserting any number of characters "X" in any positions along the two sequences, so that the final sequences have the same length. The cost of an alignment is the sum of the values of the cost function computed for all positions along the resulting sequences: each pair of characters occurring in the same position is the argument of the function for that position.

Formally, let S_1^+ and S_2^+ be the two sequences after the insertion of the occurrences of the "X" character. Let L be their length. Let indicate by $S_i^+(k)$ the character in position k in sequence S_i^+. The cost of the alignment is $\sum_{k=1}^{L} w(S_1^+(k), S_2^+(k))$.

Remark. Obviously the function w is defined to penalize misalignments, i.e. the presence of different characters in the same position, and to reward alignments, i.e. the presence of identical characters in the same positions. The alignment of a character in $\mathcal{A}$ with a character "X" is usually penalized, but less than a misalignment. Aligning two characters "X" is never optimal: they can be deleted from both sequences, yielding a better solution.

Step 1: sequencing the decisions. In this example we consider two sequences of decisions, one for each string. For each position one has to decide whether to insert a character "X" or not in the string.

Step 2: defining the state. The state must represent which decisions have already been taken: since there are two sequences, the state contains two indices, say k_1 and k_2, indicating how many positions have already been scanned and aligned in each sequence. No further information is required to check the feasibility of the remaining decisions. Hence the states have the following form: $\{k_1, k_2\}$. The cost associated with each state $\{k_1, k_2\}$ is indicated by $c(k_1, k_2)$.

Step 3: label extension.

- Initialization: $c(0, 0) = 0$.
- Extension: $c(k_1, k_2) = \min\{c(k_1 - 1, k_2) + w_{S_1(k_1), ``X''}, c(k_1, k_2 - 1) + w_{``X'', S_2(k_2)}, c(k_1 - 1, k_2 - 1) + w_{S_1(k_1), S_2(k_2)}\}$.

The optimal value is $c(n_1, n_2)$, where n_1 and n_2 are the lengths of the given sequences.

Complexity. The number of states to be labeled is $(n_1 + 1)(n_2 + 1)$. When extending labels the number of predecessors for each state is at most equal to three. Hence the complexity of the D.P. algorithm is quadratic in the input size.

A numerical example.

Instance

$S1$: A A A B A B

$S2$: B A B A B B

Cost	A	B	X
A	0	2	1
B	2	0	1
X	1	1	-

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 X X

S_2 = B A X X B A B B

Cost = 4

Figure 8: A sample instance of the string matching problem with two feasible solutions.

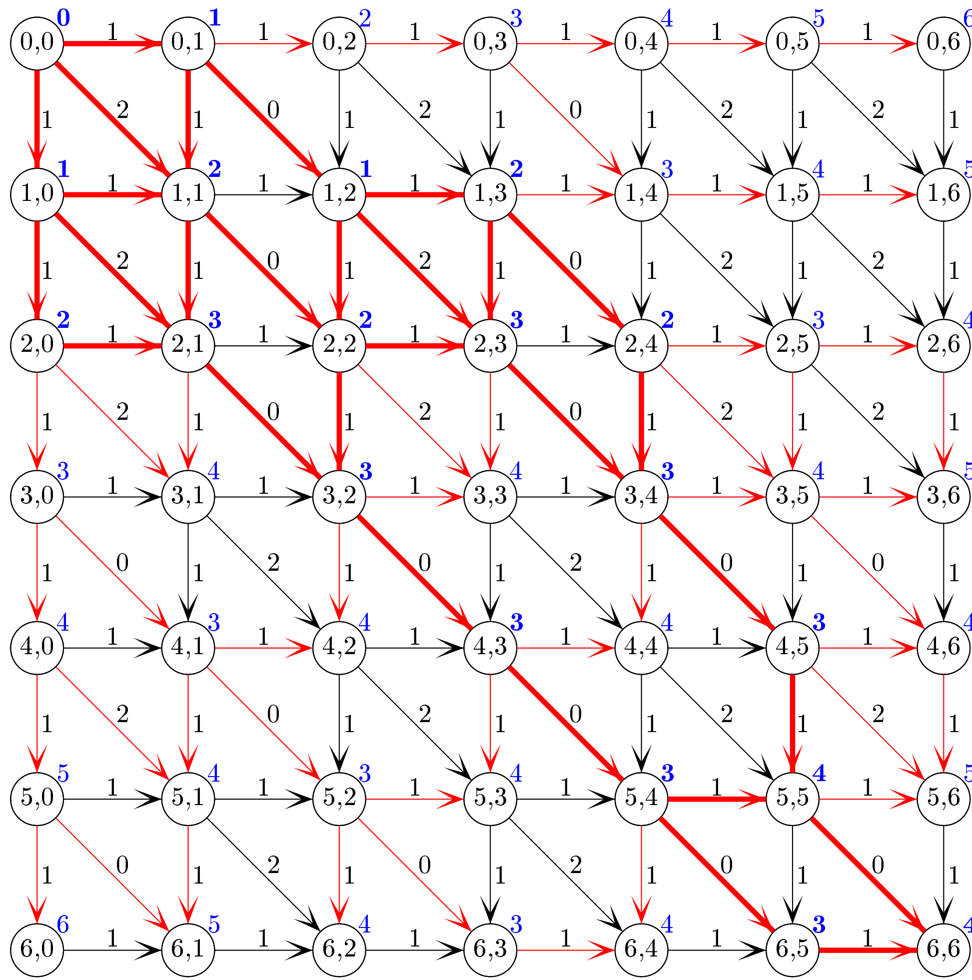

Figure 9: The state-transition graph. The graph is directed, acyclic and layered: each node can be reached from at most three predecessors in the previous two layers. Costs associated with the states are represented in blue. Optimal predecessors are represented by red arcs. Optimal solutions are indicated by bolded numbers and thick arcs. For this small instance multiple optimal solutions exist.

6 p-medians on a line

The problem. Given a straight line and a set $N = \{1, \ldots, n\}$ of points along it in fixed 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 ordered in the same order as they occur along the line.

Remark 1. The p-median problem is NP-hard on graphs, but this simplified version in which all points lie on a straight line is polynomially solvable by dynamic programming.

Remark 2. The 1-median problem on a line is easy (it is easy, i.e. polynomially solvable, even on general graphs). If the number of points is odd, the optimal location of the median coincides with the central point; if the number of points is even, the optimal location of the median is anywhere along the segment between the two central points. We indicate by $w(i, j)$ the optimal (minimum) cost of locating a single median to serve the points in the interval $[i, j]$, with $i \in N,\ j \in N, j \geq i$.

Step 1: sequencing 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.

Step 2: defining the state. The state represents which decisions have already been taken: at each stage in the decision process we need to know which is the last scanned point $i \in N$ and the number m of medians used up to that point. Hence the states have the following form: $\{i, m\}$. The cost associated with each state $\{i, m\}$ is indicated by $c(i, m)$: it indicates the minimum cost to serve the points in $[1..i]$ with m medians.

Step 3: label extension.

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

The optimal value is $c(n, p)$.

Complexity. The number of states to be labeled is given by np and since $p \leq n$ it is not larger than n^2. When extending labels, the number of predecessors for each state is $O(n)$. Hence the complexity of the D.P. algorithm is $O(n^2p)$ or $O(n^3)$.

A numerical example. Figure 10 represents an instance of the p-median problem on a line.

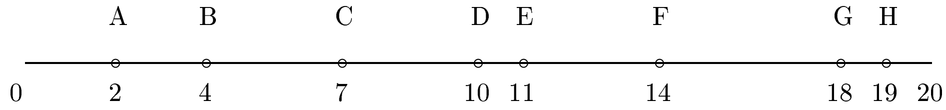

Figure 10: An instance of the p-median problem on a line. For better readability the indices $1, \dots, n$ of the given points have been replaced by letters. In this instance $p = 3$.

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

Algorithm 6.1

1: **for** $j = 1, \dots, n-1$ **do**
2: $\quad opt := j$;
3: $\quad i := j$;
4: $\quad w(j, i) := 0$;
5: $\quad paritybit := 1$;
6: $\quad$ **while** $i < n$ **do**
7: $\quad\quad i := i + 1$;
8: $\quad\quad paritybit := 1 - paritybit$;
9: $\quad\quad w(j, i) := w(j, i-1) + (x(i) - x(opt))$;
10: $\quad\quad$ **if** $(paritybit = 0)$ **then**
11: $\quad\quad\quad opt := opt + 1$;

The resulting cost matrix 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

The corresponding states-transitions graph, with the optimal solution, is represented in Figure 11.

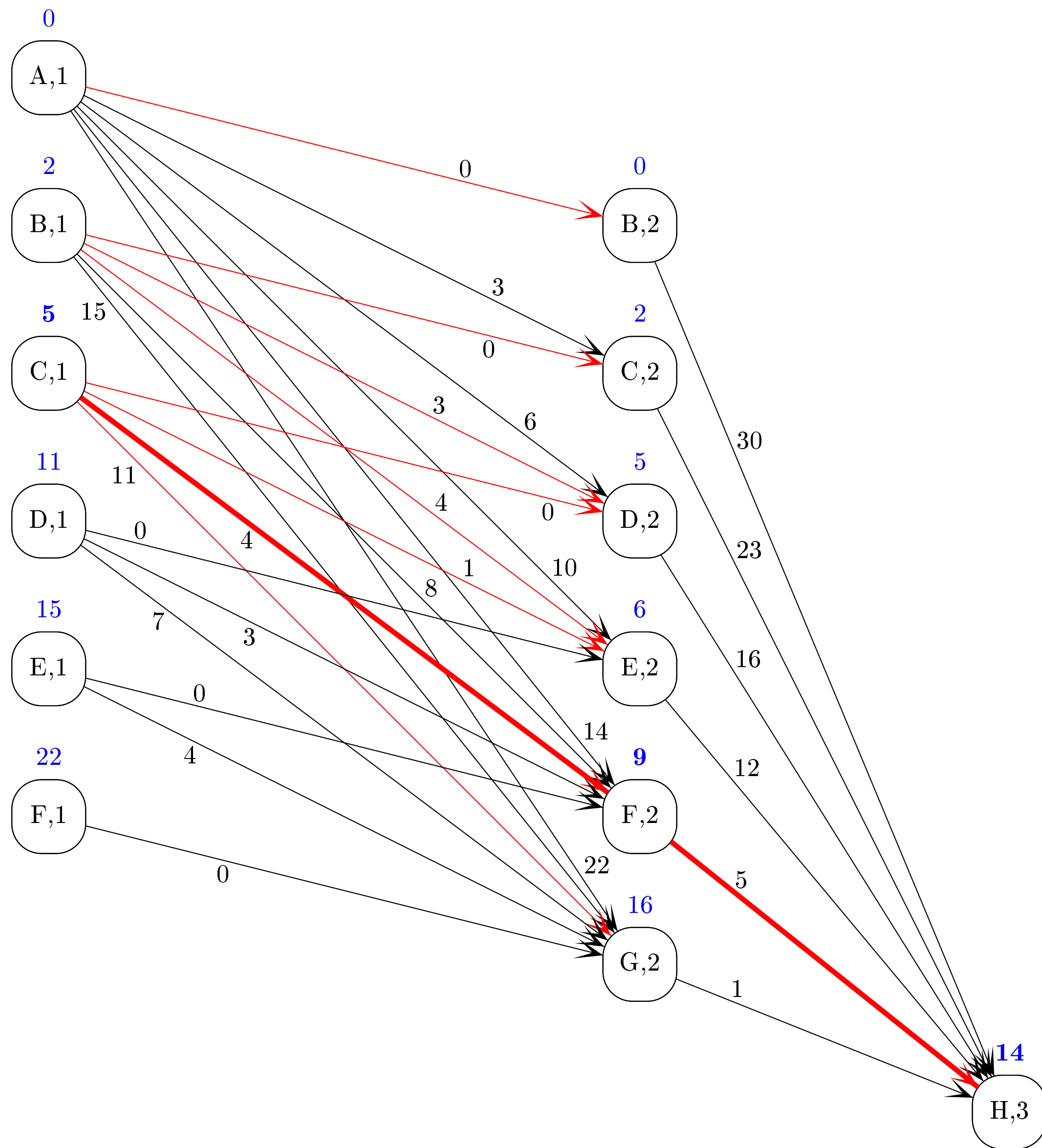

Figure 11: The states-transitions graph and the states extensions. Costs of the states are indicated in blue; costs of the transitions are indicated in red. The optimal solution is indicated by bolded costs and thick arcs.

7 The binary knapsack problem

The problem. Given a set $N = \{1, \dots, n\}$ of items with a value v_i $\forall i \in N$ and a weight w_i $\forall i \in N$, select the subset of items of maximum value such that the overall weight of the selected items does not exceed a given capacity W.

Step 1: sequencing the decisions. We can consider the items according to their numbering from 1 to n. For each item we have to decide whether to select it or not. Hence we have a sequence of n binary decisions.

Step 2: defining the state. At each point along the decision process we need to know the residual capacity which is left (or equivalently, the amount of capacity already used) after the already taken decisions. Hence the states have the following form: $\{i, q\}$, where i indicates the last item considered in the sequence and q indicates the capacity used. The value associated with each state $\{i, q\}$ is indicated by $z(i, q)$: it indicates the maximum value that can be achieved with the first i items, using an amount of capacity equal to q. A dummy initial state indicates that at the beginning no item has been considered and no capacity has been used.

Step 3: label extension.

- Initialization: $z(0, q) = 0 \ \ \forall q = 0, \dots, W$.
- Extension: $z(i,q) = \begin{cases} z(i-1,q) & \forall i \in N \ \forall q < w_i \\ \max\{z(i-1,q), z(i-1,q-w_i)+v_i\} & \forall i \in N \ \forall q = w_i, \dots, W. \end{cases}$

The optimal value is $\max_{q=0,\dots,W}\{z(n, q)\}$.

Complexity. Index i ranges in the interval $[0 \dots, n]$; the number of possible values for i is $n + 1$. Index q ranges in the interval $[0 \dots, W]$; the number of possible values for q is $W + 1$. Hence the number of states grows as $O(nW)$. When extending labels, the number of predecessors for each state is 2. Hence the worst-case time complexity of the D.P. algorithm is $O(nW)$. This complexity is not polynomial, because W does not determine the size of the instance, like n; rather it is the value of a datum of the instance. In this case, we say that the computational complexity is *pseudo-polynomial*. Dynamic Programming is a very powerful technique to design pseudo-polynomial complexity algorithms for NP-hard problems. If an NP-hard problem admits such an algorithm, it is classified as *weakly* NP-hard.

A numerical example. Table 1 shows a small instance of the binary knapsack problem. The corresponding optimal solution is reported in Table 7.

Implementation. A basic version of the D.P. algorithm described above directly comes from the definition of the extension function. It consists of filling the matrix shown in Figure 7 row after row.

However in this algorithm many iterations are wasted, because no 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

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

Table 1: A small instance. Items are sorted according to their efficiency v_i/w_i. The capacity is $W = 16$.

	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**

Table 2: The optimal solution (in bold): $z^* = 169$, $x^* = [1, 1, 0, 0, 0, 0, 1]$.

1: **procedure** KNAPSACK (BASIC VERSION)
2:     /* Inizialize */
3:     **for** $q = 0, \dots, W$ **do**
4:         $z[0, q] := 0$;
5:     /* Compute all states */
6:     **for** $i = 1, \dots, n$ **do**
7:         **for** $q = 0, \dots, w_i - 1$ **do**
8:             $z[i, q] := z[i-1, q]$;
9:             $flag[i, q] := 0$;
10:         **for** $q = w_i, \dots, W$ **do**
11:             **if** $(z[i-1, q-w_i] + v_i > z[i-1, q])$ **then**
12:                 $z[i, q] := z[i-1, q-w_i] + v_i$;
13:                 $flag[i, q] := 1$;
14:             **else**
15:                 $z[i, q] := z[i-1, q]$;
16:                 $flag[i, q] := 0$;
17:     /* Find the optimal value */
18:     $z^* := 0$;
19:     **for** $q = 0, \dots, W$ **do**
20:         **if** $(z[n, q] > z^*)$ **then**
21:             $z^* := z[n, q]$;
22:             $q^* := q$;
23:     /* Reconstruct the optimal solution */
24:     **for** $i = n, \dots, 1$ **do**
25:         $x^*[i] := flag[i, q^*]$;
26:         **if** $(flag[i, q^*] = 1)$ **then**
27:             $q^* := q^* - w_i$;
        return z^*,x^*

as a linked list. The algorithm starts from a single state of null value on row 0 and only existing states in row i generate successor states in row $i+1$. Every time a state (i, q) is generated, it is necessary to check whether another state already exists with the same value of q. If it exists, then a comparison is needed which of the two states dominate. The search for this state requires in general more steps than a direct access to a matrix entry. However this can be compensated by the sparsity of the data-structure: each linked list is likely to contain less states than a single row of the matrix z, especially in the earliest iterations.

The notation used in 7.1 is the following. Each row is a doubly linked list made of records with the following fields:

- *capac*: the value of the used capacity (q);
- *value*: the accumulated value (z);
- *left* and *right*: pointers to the adjacent records;
- *pred*: pointer to the optimal predecessor state;
- *item*: last element inserted into the knapsack.

An array *tail* contains the pointers to the rightmost elements of each row.

Algorithm 7.1 Knapsack (dynamic data-structure)

```
1: Initialize;
2: /* Compute all states */
3: for i = 1,...,n do
4:     CopyList(i);
5:     /* Initialize pointers to scan the list */
6:     p := tail[i];
7:     s := tail[i];
8:     repeat
9:         if (p^.capac + w_i <= W) then
10:            while (s^.capac > p^.capac + w_i) do
11:                s := s^.left;
12:            if (p^.capac + w_i > s^.capac) then
13:                CreateNewState(p, s, i);
14:            else
15:                /* Dominance test */
16:                if (s^.value < p^.value + v_i) then
17:                    s^.value := p^.value + v_i;
18:                    s^.item := i;
19:                    s^.pred := p;
20:        p := p^.left;
21:    until (p = nil);
22: RetrieveOptimalSolution;
```

Algorithm 7.2 *Initialize*

```
1: New(tail[0]);
2: tail[0]^.value := 0;
3: tail[0]^.capac := 0;
4: tail[0]^.left := nil;
5: tail[0]^.right := nil;
6: tail[0]^.pred := nil;
```

Algorithm 7.3 *CopyList(i)*

```
1:  p := tail[i − 1];
2:  tail[i] := nil;
3:  while (p <> nil) do
4:      if (tail[i] = nil) then
5:          New(tail[i]);
6:          s := tail[i];
7:          sˆ.capac := pˆ.capac;
8:          sˆ.value := pˆ.value;
9:          sˆ.item := pˆ.item;
10:         sˆ.right := nil;
11:         sˆ.left := nil;
12:         sˆ.pred := tail[i − 1];
13:     else
14:         New(sˆ.left);
15:         sˆ.leftˆ.capac := pˆ.capac;
16:         sˆ.leftˆ.value := pˆ.value;
17:         sˆ.leftˆ.item := pˆ.item;
18:         sˆ.leftˆ.right := s;
19:         sˆ.leftˆ.left := nil;
20:         sˆ.leftˆ.pred := p;
21:         s := sˆ.left;
22:     p := pˆ.left;
```

Algorithm 7.4 *CreateNew(p, s, i)*

```
1:  New(t);
2:  tˆ.left := s;
3:  tˆ.right := sˆ.right;
4:  sˆ.right := t;
5:  if (tail[i] = s) then
6:      tail[i] := t
7:  else
8:      tˆ.rightˆ.left := t;
9:  tˆ.capac := pˆ.capac + w_i;
10: tˆ.value := pˆ.value + v_i;
11: tˆ.item := i;
12: tˆ.pred := p;
```

Algorithm 7.5 *RetrieveOptimalSolution*

```
1:  z* := 0;
2:  for i = 1, . . . , n do
3:      x[i] := 0;
4:  p := tail[n];
5:  while (p <> nil) do
6:      if (pˆ.value > z*) then
7:          z* := pˆ.value;
8:          p* := p;
9:      p := pˆ.left;
10: while (p*ˆ.pred <> nil) do
11:     x[p*ˆ.item] := 1;
12:     p* := p*ˆ.pred;
    return z*,x*
```

	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**

Table 3: The sparse matrix in the implementation using dynamic data-structures: about one half of the states are not evaluated.

	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									

Table 4: The two sparse sub-matrices in the bi-directional implementation: about 75% of the states are not generated.

In this implementation a large part of the matrix (i.e. a large part of the state space) is not generated, as shown in Table 7.

However, considering how many elementary operations are needed in the second implementation compared to the first one, this saving may be insufficient to justify the use of dynamic data-structures to solve this toy instance. The advantage of the second implementation is likely to be meaningful only for very large instances (where computing time is also more significant).

An observation directly stemming from the analysis of Table 7 is that the number of states grows with index i: top rows in the matrix are sparser than bottom rows. This suggests a further improvement: bi-directional D.P.. In bi-directional D.P. non-dominated states are generated in both directions along the sequence of decisions; therefore we have to manage forward and backward states separately and independently. The extension of states stops when a suitable "half-way point" is reached. The definition of this stop criterion must satisfy a fundamental property: it must be possible to obtain any feasible solution by suitably combining a forward state and a backward state, i.e. two non-dominated sub-policies. In our example, a possible criterion is the use of the items: for instance, we define forward states as non-dominated combinations of items $1\ldots,3$ and backward states as non-dominated combinations of items $4\ldots,7$. Using the same extension rules of forward states for generating backward states, with the only difference that items are considered in reverse order, we obtain the result shown in Table 7.

Bi-directional D.P. allows to decrease significantly the number of states to be considered. On the other side, it requires a post-processing operation to join pairs of forward and backward states in order to obtain solutions. Pairs of states must satisfy the capacity constraint: each forward state with capacity consumption q can be joined with any backward state with capacity consumption not larger than $W-q$. The join step can be done in $O(n)$, as illustrated in Table 7.

In the join step each forward state is matched with the most convenient backward state, that is with the backward state with maximum consumption among those satisfying the capacity constraint. For instance, forward state with $q=5$ and $z=55$ can be feasibly matched with all backward states with consumption between 0 and 11: the most convenient one is the backward state with $q=8$ and $z=80$. This can be done in $O(n)$ as shown in Algorithm 7.6: the pseudo-code assumes an implementation with two linked lists, whose extreme elements are pointed by $tail^{fw}$ and $tail^{bw}$ on the right side of the matrix and $head^{fw}$ and $head^{bw}$ on

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
FW	0	(0)	(0)	(0)	45	55	(55)	(55)	87	100	(100)	(100)	(100)	142	(142)	(142)	(142)
BW		149	142	131	123	(80)	(80)	(80)	80	69	62	(0)	(0)	(0)	(0)	(0)	0
	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	149	149	142	131	168	135	135	135	167	**169**	162	100	100	142	142	142	162

Table 5: The join step requires to scan the forward and backward list of non-dominated states.

the left side. It returns the optimal value z^* and two pointers p^* and q^* to the optimal forward and backward state pair. Since the two lists are scanned only once, the worst-case time complexity of the join procedure is $O(n)$.

Algorithm 7.6 *Join*

```
1:  z* := 0;
2:  p := tail^fw;
3:  q := head^bw;
4:  while (p ≠ nil) and (q ≠ nil) do
5:      repeat
6:          if (p^.value + q^.value > z*) then
7:              z* := p^.value + q^.value;
8:              p* := p;
9:              q* := q;
10:         q := q^.right;
11:     until (q = nil) or (p^.capac + q^.capac > W)
12:     repeat
13:         p := p^.left;
14:     until (p = nil) or (p^.capac + q^.capac ≤ W)
    return z*,p*,q*
```

8 Dynamic system optimal control

The problem. We are given a discrete-time dynamic system, i.e. a system characterized by an input, a state and an output. In this example they are all very simple, just consisting 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 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. When the cost is negative, it represents a benefit. The whole set of input values is to be decided and the initial state $x(0)$ as well. We want to lead the system to a given final state $\overline{x}$ by a sequence of transitions of minimum cost (or maximum benefit).

Step 1: sequencing the decisions. In this problem there is an obvious correspondence between the decision process and the dynamic system. In this correspondence the sequence of decisions is the sequence of input values to be chosen. So there is a decision for each point in time $t \in 1, \ldots, T$.

Step 2: defining the state. Owing to the above mentioned correspondence, the state in dynamic programming corresponds with the state of the dynamic system. Hence the states have the following form: $\{x, t\}$, where x indicates the state of the system and t indicates the point in time. The cost associated with each state $\{x, t\}$ is indicated by $c(x, t)$: it indicates the minimum cost that must be paid for reaching state x at time t.

Step 3: label extension.

- Initialization: $c(x, 0) = 0 \ \ \forall x \in X$.
- Extension: $c(x, t) = \max_{u \in U}\{c(x-u, t-1) + f_t(x-u, u)\} \ \ \forall x \in X \ \forall t \in 1, \ldots, T$.

The minimum cost is $c(\overline{x}, T)$ (if it is negative, it represents a maximum benefit).

Complexity. The number of possible values for x is $|X|$, while the number of possible values for t is T. Hence the number of states grows as $O(|X|T)$. When extending labels, the number of predecessors for each state is $|U|$. Hence the worst-case time complexity of the D.P. algorithm is $O(|X||U|T)$. If $|X|$ and $|U|$ are given, then the computational complexity is *polynomial*. If it is part of the input, the complexity is *pseudo-polynomial*.

A numerical example. Tables 6 represents an instance of the problem with $X = \{1, 2, 3\}$, $U = \{-1, 0, 1\}$ and $T = 3$. The system is required to reach state $\overline{x} = 2$ at $t = 3$. The states-transitions graph is represented

	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

Table 6: The transition costs. Rows indicate the x value, columns indicate the u value.

in Figure 12 together with the labels and the optimal solution.

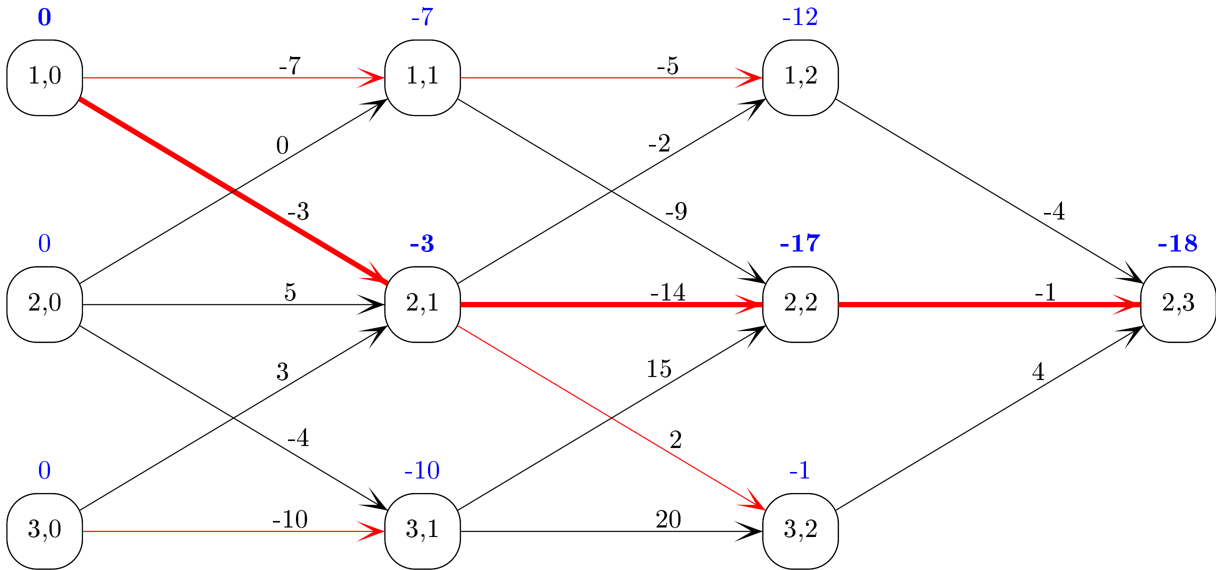

Figure 12: The state-transition graph and the state extensions. Costs are represented in blue; optimal predecessors are represented in red. The optimal solution is indicated by thick arcs and bolded numbers.

9 Optimal budget allocation

The problem. We are given a set $P = \{1, \ldots, n\}$ of projects and a budget R. We have to assign an investment to each project. Depending on the investment, each project is expected to yield a profit. No assumption is made about the kind of relationship between the investment and the profit, but for simplicity here we assume that the investments are integer and non-negative. For each project $i \in P$ the corresponding investment is represented by x_i and the corresponding expected profit by $f_i(x_i)$. The domain of x_i, i.e. the set of possible investments in project $i \in P$ is indicated by X_i. The objective is to maximize the overall expected profit without exceeding the budget.

Step 1: sequencing the decisions. In this problem we consider the projects in a sequence, arbitrarily. At each point in time during the decision process the first part of the sequence has already been scanned while the last part of the sequence is not, i.e. the first projects have been assigned an investment while the investments in the remaining projects are still to be decided. So a decision must be taken for each project $i \in P$.

Step 2: defining the state. The state must represent all relevant information at any generic step during the decision process. Obviously we need to know where we are in the decision process, i.e. which is the last decided investment. Furthermore, owing to the constraint on the limited budget R, we need to know how much resource is left. Hence the states have the following form: $\{i, r\}$, where i indicates the last project considered and r indicates the residual available budget. The profit associated with each state $\{i, r\}$ is indicated by $p(i, r)$: it indicates the maximum profit that can be achieved when reaching state $\{i, r\}$. An additional initial state $\{0, R\}$ corresponds to the beginning of the decision process, when no investment has been decided yet and the whole budget is still available.

Step 3: label extension.

- Initialization: $p(0, R) = 0$.
- Extension: $p(i, r) = \max_{x_i \in X_i}\{p(i-1, r + x_i) + f_i(x_i)\}\ \ \forall i \in P\ \forall r \in 0, \ldots, R$.

The maximum overall expected profit is $\max_{r=0}^{R}\{p(n, r)\}$. If the expected profits are larger than the investments (i.e. $f_i(x_i) \geq x_i\ \forall i \in P\ \forall x_i \in X_i$) and it is possible to invest all the resource (i.e. $R \leq \sum_{i \in P} \max_{x_i \in X_i}\{x_i\}$), then the maximum expected profit is certainly attained at state $(n, 0)$, because it is certainly optimal to invest the whole budget.

Complexity. The number of possible values for i is n, while the number of possible values for r is $R + 1$ (from 0 to R). Hence the number of states grows as $O(nR)$. When extending labels to each state of project $i \in P$, the number of predecessors for each state is at most $|X_i|$, which cannot be larger than $R+1$. Hence the worst-case time complexity of the D.P. algorithm is $O(nR^2)$. The computational complexity of this algorithm is *pseudo-polynomial*.

A numerical example. Tables 7 represents an instance of the optimal budget allocation problem with $P = \{1, \ldots, 4\}$ 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		

Table 7: The possible investments and the corresponding expected profits for each project.

The states-transitions graph is represented in Figure 13 together with the labels and the optimal solution.

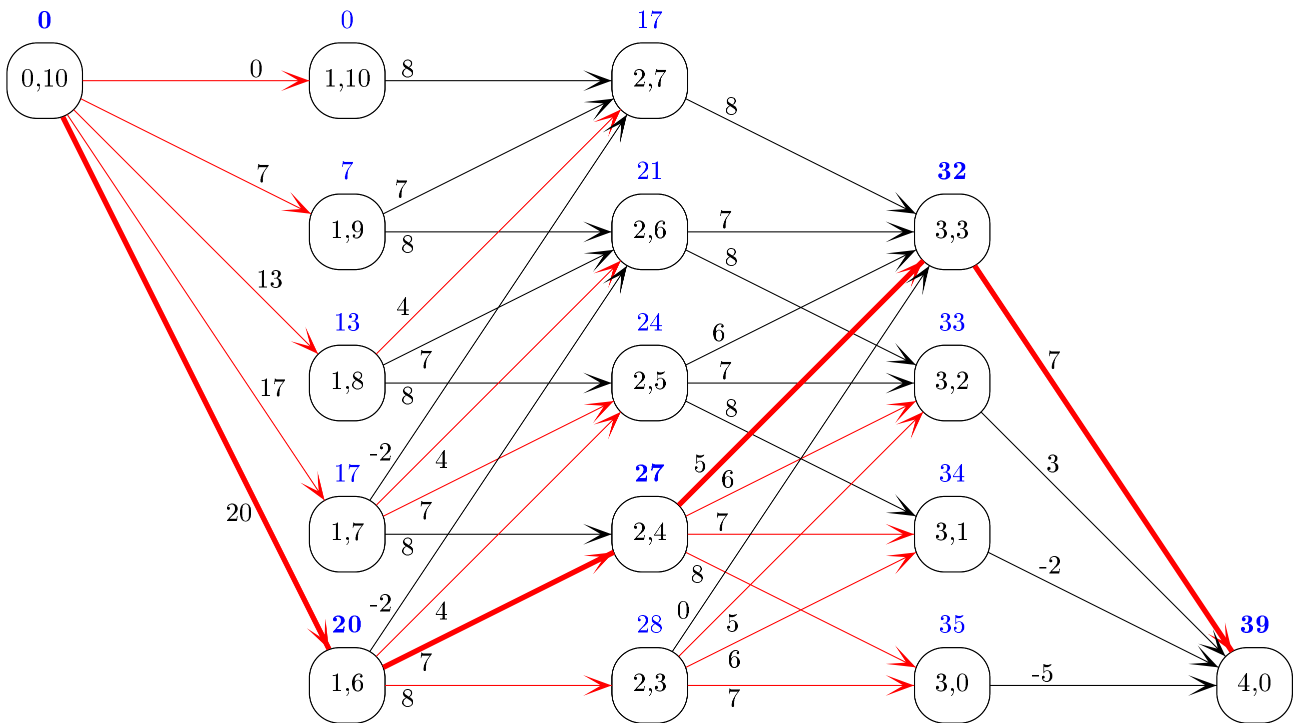

Figure 13: The state-transition graph and the state extensions. Costs are represented in blue; optimal predecessors are represented in red. The optimal solution is indicated by thick arcs and bolded numbers.

10 The max independent set problem on an interval graph

The problem. A very small car rental company there is only one car. The company has collected a set N of orders from potential customers and now it must decide which orders to satisfy in order to maximize its profits. Each order $i \in N$ is characterized by a start time s_i, an end time e_i and a profit p_i. Obviously, no two selected orders can overlap in time.

Remark. In graph terminology this problem is called *Max independent set problem on an interval graph*. We can define a graph where each vertex corresponds to an order and any two vertices i and j are connected by an edge if and only if the two corresponding orders overlap. For its particular structure, the resulting graph is called *interval graph*. Overlapping orders are incompatible, i.e. they cannot be both selected. This constraint translates into the search for an independent set, i.e. a subset of vertices such that they are not connected to one another by any edge. The subset of orders 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 order. The max independent set problem is NP-hard on general graphs, but it is polynomially solvable on interval graphs.

Step 1: sequencing the decisions. The orders can be sequenced according to their start time s. A binary decision must be taken for each of them (whether to select that order or not).

Step 2: defining the state. At each point along the decision process we need to know where we are, i.e. which is the last order considered and what constraints are propagated to the future decisions because of the decisions already taken. This is easily represented by the time when the car becomes available. Actually the time the car becomes available is not directly relevant in itself; what is relevant is the next order that can be selected. If we know which is the next order that can be selected we do not even need to know which is the last order considered. Hence the states have the following (very simple) form: $\{i\}$, where i indicates the next order that can be selected when the car becomes available again. The profit associated with each state $\{i\}$ is indicated by $f(i)$: it indicates the maximum profit that can be achieved when reaching state $\{i\}$. A dummy state $\{n+1\}$ represents the final state, when all orders have been decided. We set s_{n+1} to an arbitrary value larger than $\max_{i \in N}\{e_i\}$.

Step 3: label extension.

- Initialization: $f(1) := 0$.
- Extension: $f(i) := \max\{0, \max_{j \in N: e_j \leq s_i}\{f(j) + p_j\}\} \quad \forall i = 1, \ldots, n+1$.

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

Complexity. The number of possible values for i is $n+1$. When extending labels to each state $\{i\}$, the number of predecessors is at most n. Hence the worst-case time complexity of the D.P. algorithm is not worse than $O(n^2)$. However, it is redundant to explore all predecessors for each state. A more efficient implementation is obtained if we assign each state j a unique successor $succ(j)$ such that

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

The order $succ(j)$ is the first order that can be selected after order j. If we can determine $succ(j)$ for each $j \in N$, then we can construct a graph in which each state j has only two outgoing arcs: an arc with value 0 to the next state $j+1$ and an arc of value p_j to $succ(j)$. These two arcs represent the two alternatives: select j or do not select j. The resulting graph is directed, acyclic and layered and has only $O(n)$ arcs: therefore the label propagation algorithm takes $O(n)$. The bottleneck determining the worst-case time complexity is the construction of the graph. It can be done in linear time if the orders are sorted, as hown in Algorithm 10.1. If they are not, it takes $O(n \log n)$ to sort them and therefore this is the resulting time complexity of the algorithm.

A vector G contains $2n$ records, two records for each order, with the following fields:

Algorithm 10.1 *MaxIndependentSetonanIntervalGraph*

```
1: Initialize;
2: Sort;
3: ComputeLabels;
4: RetrieveOptimalSolution;
```

- *id*: the order corresponding to the record;
- *time*: the start or end time of order *id*;
- *tail*: a boolean flag indicating whether the value *time* is the start time ($tail = 1$) or the end time ($tail = 0$) of order *id*;
- *mate*: the position in the array G of the other time value of order *id*;

They are initialized as shown in 10.2.

Algorithm 10.2 *Initialize*

```
1: for i = 1, ..., n do
2:     G[i].id := i;
3:     G[i + n].id := i;
4:     G[i].time := s_i;
5:     G[i + n].time := e_i;
6:     G[i].tail := 1;
7:     G[i + n].tail := 0;
8:     G[i].mate := i + n;
9:     G[i + n].mate := i;
```

The sorting procedure can be implemented in $O(n \log n)$ keeping the consistency of the information stored in the record fields. In particular the fields *mate* can be updated in $O(1)$ every time the position in G of any record is changed.

The procedure that computes the optimal labels of the states (shown in 10.3) simply scans the vector G and propagates labels in two ways: from each state corresponding to the start time of an order to the state corresponding to the end time of the same order; and from each state to the next one. The first type of extension implies a profit, the second one does not.

Algorithm 10.3 *ComputeLabels*

```
1: for i = 1, ..., 2n do
2:     G[i].value := 0;
3:     G[i + n].pred := 0;
4: for i = 1, ..., 2n do
5:     /* Extend to the head */
6:     if (G[i].tail = 1) then
7:         if (G[i].value + p_{G[i].id} > G[G[i].mate].value) then
8:             G[G[i].mate].value := G[i].value + p_{G[i].id};
9:             G[G[i].mate].pred := i;
10:    /* Extend to the next */
11:    if (G[i].value > G[i + 1].value) then
12:        G[i + 1].value := G[i].value;
13:        G[i + 1].pred := i;
```

Finally the optimal solution can be retrieved starting from the dummy node $n + 1$ and going backward along the chain of pointers defined by the fields *pred*, as described in 10.4.

Algorithm 10.4 *RetrieveOptimalSolution*

1: $z^* := G[2n+1].value$;
2: **for** $i = 1, \ldots, n$ **do**
3: $\quad x^*[i] := 0$;
4: $i := 2n$;
5: **while** $(i \neq 0)$ **do**
6: $\quad$ **if** $(G[i].pred = G[i].mate)$ **then**
7: $\quad\quad x^*[G[i].id] := 1$;
8: $\quad i := G[i].pred$;
return z^*, x^*

A numerical example. Table 8 represents an instance of the problem with $n = 7$.

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

Table 8: The orders received by the car rental company.

The state-transitions graph is represented in Figure 14 together with the labels and the optimal solution.

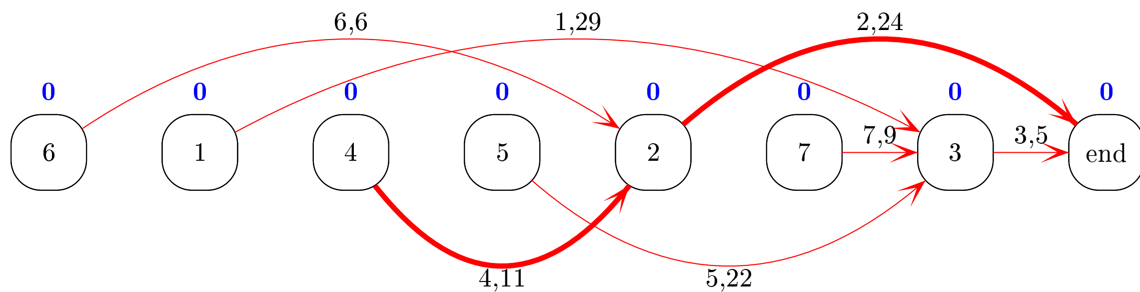

Figure 14: The state-transition graph and the state extensions. Costs are represented in blue; optimal predecessors are represented in red. The optimal solution is indicated by thick arcs and bolded numbers.

The matrix implementation described above would produce the following matrix.

Index	1	2	3	4	5	6	7	8	9	10	11	12	13	14
id	1	2	3	4	5	6	7	1	2	3	4	5	6	7
time	4	18	40	4	7	3	21	15	42	45	33	29	9	30
tail	1	1	1	1	1	1	1	0	0	0	0	0	0	0
mate	8	9	10	11	12	13	14	1	2	3	4	5	6	7

Table 9: The matrix after initialization.

Index	1	2	3	4	5	6	7	8	9	10	11	12	13	14
id	6	1	4	5	6	1	2	7	5	7	4	3	2	3
time	3	4	4	7	9	15	18	21	29	30	33	40	42	45
tail	1	1	1	1	0	0	1	1	0	0	0	1	0	0
mate	5	6	11	9	1	2	13	10	4	8	3	14	7	12

Table 10: The matrix after sorting.

Index	1	2	3	4	5	6	7	8	9	10	11	12	13	14
id	6	1	4	5	6	1	2	7	5	7	4	3	2	3
time	3	4	4	7	9	15	18	21	29	30	33	40	42	45
tail	1	1	1	1	0	0	1	1	0	0	0	1	0	0
mate	5	6	11	9	1	2	13	10	4	8	3	14	7	12
value	0	0	0	0	6	11	11	11	22	22	29	29	35	**35**
pred	0	1	2	3	1	2	6	7	4	9	3	11	7	13

Table 11: The matrix after labeling.

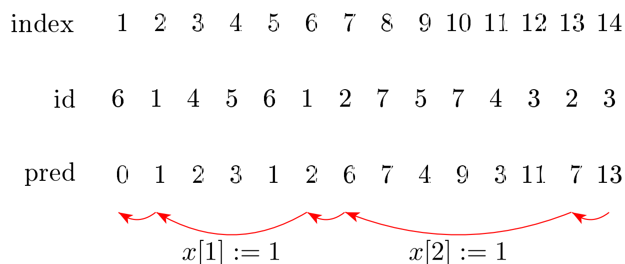

Figure 15: Reconstruction of the optimal solution.