Network Optimization Models and Methods

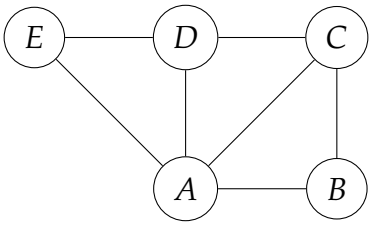
Shixuan Zhang

ISEN 320-501, Fall 2023

1 Graphs and Networks

A graph $G = (N, E)$ consists of a set of *nodes* N and a set of *edges* E, which are represented by pairs of nodes. For example, $N = \{1, 2, \ldots, n\}$ and $E = \{(1,2), (1,3), \ldots, (1,n)\}$. We remark that there could be multiple edges represented by the same pair of nodes. If there is at most one edge between a pair of nodes, then G is called a *simple graph*.

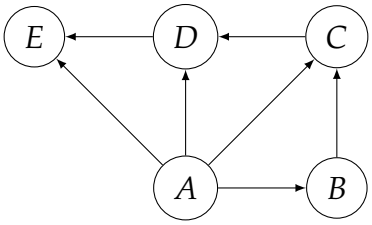
Example 1. *Here is a graph with 5 nodes.*



The nodes are denoted by A, B, C, D, E, and the edges are represented by the unordered pairs (A, B), (A, C), (A, D), (A, E), (B, C), (C, D), (D, E).

A graph is undirected if the edges have no "directions," i.e., (i, j) is the same as (j, i) for any $i, j \in N$. Otherwise the graph is directed and sometimes called a *digraph*. The nodes and edges in directed graphs are often called *vertices* and *arcs*.

Example 2. *Here is a directed graph with 5 vertices.*



The vertices are denoted by A, B, C, D, E, and the arcs are represented by the ordered pairs (A, B), (A, C), (A, D), (A, E), (B, C), (C, D), (D, E).

An undirected graph can be viewed as a directed graphs where we have arcs with both directions for each edge.

We say that a node $j \in N$ is *adjacent* to $i \in N$ if $(i, j) \in E$. An edge (or arc) $(i, j) \in E$ is said to be *incident* to nodes $i, j \in N$. When the graph is directed, we also say that it *emanates* from node i and *terminates* at node j; it is an *outgoing* arc of i and *incoming* arc of j. The *degree* of a node is the number of its incident edges. The *in-degree* of a vertex is the number of its incoming arcs, and the *out-degree* of a vertex is the number of its outgoing arcs.

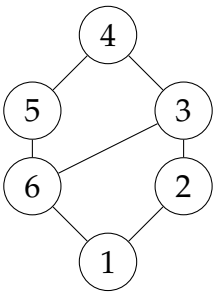
A *walk* is a sequence of alternating nodes and edges

$$i_1, e_1, i_2, e_2, \ldots, i_{k-1}, e_k, i_k,$$

where $i_1, \ldots, i_k \in N$ and $e_j = (i_j, i_{j+1}) \in E$ for $j = 1, \ldots, k-1$. It is called *closed* if $i_1 = i_k$. We also denote a walk by $i_1, i_2, \ldots, i_k$ or $e_1, \ldots, e_k$. A *trail* is a walk where $e_1, \ldots, e_k$ are distinct. A *path* is a walk where $i_1, \ldots, i_k$ are distinct. A path is always a trail, but the converse is not necessarily true. These definitions apply to directed graphs as well.

A *circuit* is a closed trail. A *cycle* is a circuit $i_1, i_2, \ldots, i_k, i_1$ where $i_1, \ldots, i_k$ are distinct. A graph without any cycles is called *acyclic*. A graph is *connected* if for any pair of nodes, there is a path connecting them. A connected, acyclic graph is called a *tree*.

Example 3. *In the following graph,*



$1, 2, 1$ *is a walk but not a trail;* $3, 2, 1, 6, 5, 4, 3, 6$ *is a trail but not a path;* $1, 2, 3, 4, 5, 6, 1$ *is a circuit and a cycle. The graph is connected but not a tree.*

An *Eulerian trail* is a trail $i_1, e_1, i_2, \ldots, i_{k-1}, e_{k-1}, i_k$ such that each edge e_j appears only once in the trail for $j = 1, \ldots, k-1$. An *Eulerian circuit* is an Eulerian trail that is also a circuit. Historically, one of the first graph theory problems studied was the Eulerian trail/circuit problem, motivated by the Königsberg seven bridge problem. It can be formulated as the existence of an Eulerian trail/path in the following (nonsimple) graph (Figure 1).

The seven bridge problem has a simple answer using the above definitions: a connected undirected graph admits an Eulerian trail (resp. circuit) if at most two (resp. none) of the nodes have odd degrees. This is because in an Eulerian trail, except for

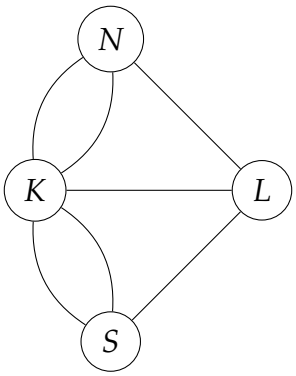


Figure 1: Königsberg seven bridge problem

the starting and ending nodes, we must arrive at and leave from each node the same amount of times, so the degree must be even. Conversely, if none of the nodes has an odd degree, then each time we arrive at the node, there is an edge we can use to leave from this node. The same argument applies to Eulerian trail by connecting the two odd-degree nodes (if there are any). Therefore, we can say that the Eulerian trail/circuit problem is easy as we only need to count the degrees of the nodes.

The following *vertex coloring* problem can be much more challenging: given a graph $G = (V, E)$, we want to color the vertices (nodes) such that no adjacent vertices share the same color. The minimum number of colors we need is called the *chromatic number* of the graph. A *heuristic* method to find a vertex coloring is as follows.

(i) Start with a vertex and use color $C = \{1\}$.

(ii) Go to a vertex $i \in N$.

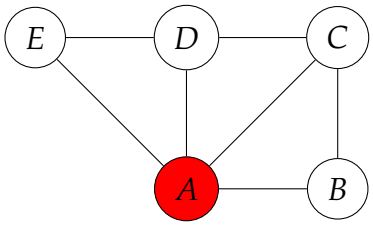
  (a) If there is a color $c \in C$ that is not used by any adjacent node j of i, color i with c;

  (b) otherwise add a new color to c' to C and color i with c'.

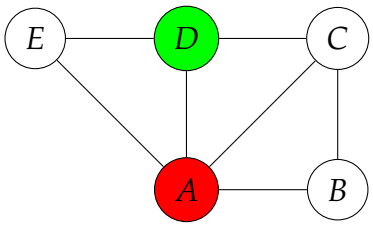
(iii) Repeat Step 2 until all vertices are colored.

Example 4. *The following example illustrates the heuristic way of coloring the vertices.*

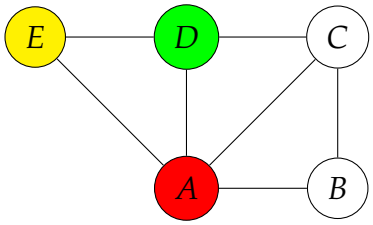
Step 1. *Used color: red*



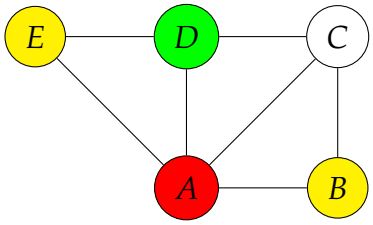
Step 2. *Used colors: red, green*



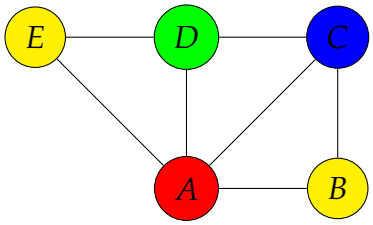
Step 3. *Used color: red, green, yellow*



Step 4. *We can color node B with the yellow color.*

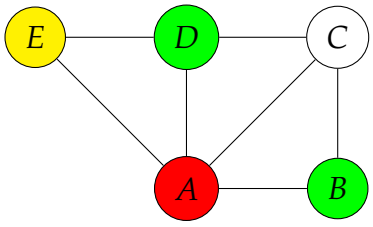


Step 5. *Used color: red, green, yellow, blue*

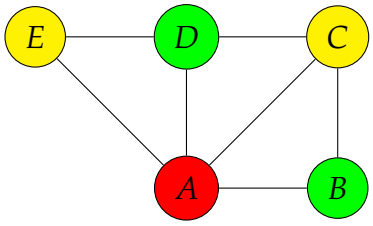


In this way, we can color the graph with 4 colors. However, depending on the ordering of the colors, we may be able to use fewer colors. For example, in step 4, if we color node B with the green color, then we would get the following.

Step 4. *Used color: red, green, yellow*



Step 5. *Used color: red, green, yellow*



This means that we only need at most 3 colors for this graph.

The chromatic number for the graph in Example 4 is indeed 3. To see this, note that the graph contains triangles, such as $\{A, D, E\}$ and $\{A, B, C\}$. Each of these triangles would require 3 different colors, as each vertex is adjacent to the other two. Therefore, the graph would require at least 3 colors. This argument can be generalized by the notion of *cliques* or *complete subgraphs*, meaning a subset of the vertices that are all adjacent to each other.

We are fortunate in finding the chromatic number in Example 4 by the above heuristic. In general, such heuristic can only be used for an upper bound on the chromatic number, which may not equal the lower bound provided by cliques. Nevertheless, it can be used to formulate the vertex coloring problem as a integer linear optimization as follows. Suppose we need at most c colors, which can be found by the above heuristic. For $i \in N$ and $k = 1, \ldots, c$, let

$$x_{ik} = \begin{cases} 1, & \text{if the vertex } i \text{ is colored by } k, \\ 0, & \text{otherwise,} \end{cases}$$

and

$$y_k = \begin{cases} 1, & \text{if the color } k \text{ is used,} \\ 0, & \text{otherwise.} \end{cases}$$

The vertex coloring problem can be formulated as

$$\begin{aligned} \min \quad & \sum_{k=1}^{c} y_k \\ \text{s. t.} \quad & \sum_{k=1}^{c} x_{ik} = 1, && i \in N, \\ & x_{ik} + x_{jk} \leq 1, && (i,j) \in E,\ k = 1, \ldots, c, \\ & x_{ik} \leq y_k, && i \in N,\ k = 1, \ldots, c, \\ & x_{ik}, y_k \in \{0, 1\}, && i \in N,\ k = 1, \ldots, c. \end{aligned}$$

The first constraint ensures that we are coloring each vertex with exactly one color; the second constraint ensures that adjacent vertices do not use the same color; the third constraint checks if a color is used on any of the vertices. The objective function is the number of colors used on the graph.

2 Network Flow Problems

A directed graph $G = (N, A)$ can be used to represent pipelines or transportation networks. Each arc $(i, j) \in A$ is associated with a *flow* variable x_{ij}. At each node $i \in N$, we have the *flow balance* constraint

$$\sum_{j:(j,i)\in A} x_{ji} - \sum_{j:(i,j)\in A} x_{ij} = f_i,$$

where f_i is the extraction/injection of the flow at node $i \in N$.

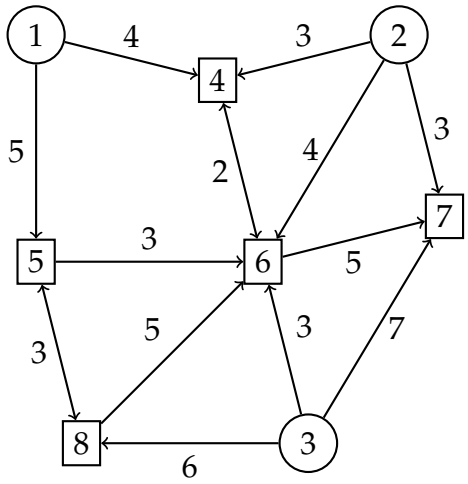
Example 5. *A water pipeline is built to deliver water to residential locations in a village. The water sources are listed below*

source	*1*	*2*	*3*
capacity	*100*	*100*	*80*

and the residential demands are listed below.

residence	*4*	*5*	*6*	*7*	*8*
demand	*50*	*60*	*40*	*30*	*70*

Each pipeline has its unit cost of delivering water, due to geographical differences. Below is the water network, where circle nodes are water sources and square nodes are residential locations. The arrows indicate the directions and the numbers indicate the unit costs for the pipelines. The



goal is to find the minimum total water delivery cost while satisfying the residential demands.

Let c_{ij} denote the cost on the arc $(i, j) \in A$, d_i denote the supply/demand at the node $i \in N$. We define variables

$$x_{ij} \geq 0 : \text{ water flow in the directed pipeline } (i, j) \in A,$$

and

$$y_i \in \mathbb{R} : \text{ water flowing out of the network at the node } i \in N.$$

Note that it is possible to have both x_{ij} and x_{ji}. If $y_i < 0$, then it means water flows into the network at the node $i \in N$. We have the water flow balance constraint

$$\sum_{j:(j,i)\in A} x_{ji} - \sum_{j:(i,j)\in A} x_{ij} = y_i, \forall\, i \in N.$$

At water sources, we have

$$d_i \le y_i \le 0, \quad i = 1,2,3.$$

At residential locations, we have

$$y_i \ge d_i, \quad i = 4,5,\ldots,8.$$

Then objective function is

$$\min \quad \sum_{(i,j)\in A} c_{ij} x_{ij}.$$

We code this LO model in `model_water_network.py` *and get the following results.*

```
The minimum cost is 1120.00.
The flow in each pipeline is shown below.
x(1, 4) = 10.000000000000004
x(1, 5) = 60.0
x(5, 6) = 0.0
x(8, 5) = 0.0
x(5, 8) = 0.0
x(8, 6) = 0.0
x(3, 8) = 70.0
x(3, 6) = 9.999999999999991
x(3, 7) = 0.0
x(6, 4) = 0.0
x(4, 6) = 0.0
x(2, 4) = 40.0
x(2, 6) = 30.000000000000007
x(2, 7) = 30.0
x(6, 7) = 0.0
```

If we have limits on the flows, then it may not be possible to transport any amount from the source to the target (or *sink*). We only inject flow at the source $s \in N$ and extract flow at the sink $t \in N$ and solve the above network flow problem. To simply

the notation, we can create an artificial arc (t, s) and

$$\begin{aligned} \max \quad & x_{ts} \\ \text{s.t.} \quad & \sum_{j:(j,i)\in A'} x_{ji} - \sum_{j:(i,j)\in A'} x_{ij} = 0, \quad \forall\, i \in N, \\ & 0 \le x_{ij} \le u_{ij}, \qquad\qquad \forall\, (i,j) \in A, \end{aligned} \tag{1}$$

where $A' = A \cup \{(t, s)\}$ and u_{ij} is the given limit for arc $(i, j) \in A$. The dual problem

$$\begin{aligned} \min \quad & \sum_{(i,j)\in A} u_{ij} y_{ij} \\ \text{s.t.} \quad & z_t - z_s \ge 1, \\ & z_i - z_j + y_{ij} \ge 0, \qquad \forall\, (i,j) \in A, \\ & y_{ij} \ge 0, \qquad\qquad \forall\, (i,j) \in A, \end{aligned} \tag{2}$$

has a nice interpretation, known as the *minimum cut* problem. Given a graph $G = (N, A)$, an *s-t cut* is a partition of $N = S \cup T$, $S \cap T = \varnothing$, such that $s \in S$ and $t \in T$. Given the capacity of each arc u_{ij}, the capacity of the cut S, T is the sum

$$\sum_{i\in S, j\in T, (i,j)\in A} u_{ij}.$$

To find a cut with minimum capacity, we can formulate it as an integer linear optimization. For each $i \in N$, let

$$z_i = \begin{cases} 1 & \text{if } i \in T, \\ 0 & \text{if } i \in S, \end{cases}$$

and for each $(i, j) \in A$, let

$$y_{ij} = \begin{cases} 1 & \text{if } i \in S \text{ and } j \in T, \\ 0 & \text{otherwise.} \end{cases}$$

Clearly we must have constraints

$$y_{ij} \ge z_j - z_i, \quad \forall\, (i,j) \in A.$$

Since we are looking for an *s-t* cut, we should have

$$z_t = 1,\ z_s = 0 \iff z_t - z_s = 1.$$

The objective is to minimize the sum

$$\min \quad \sum_{(i,j)\in A} u_{ij} y_{ij}.$$

To summarize, the minimum cut problem can be written as

$$\begin{aligned} \min \quad & \sum_{(i,j)\in A} u_{ij} y_{ij} \\ \text{s.t.} \quad & y_{ij} \geq z_j - z_i, && \forall\, (i,j) \in A, \\ & z_t - z_s = 1, \\ & y_{ij} \in \{0,1\}, && \forall\, (i,j) \in A, \\ & z_i \in \{0,1\}, && \forall\, i \in N. \end{aligned} \tag{3}$$

In fact, it can be proved that the MILO problem (3) is equivalent to the LO problem (2). The main idea is that all of the *minors* of the constraint coefficient matrix in (2) is ± 1 and therefore the simplex tableau will only consist of coefficients ± 1. Hence, the basic variable values will be either 0 or 1, as the right-hand side constants.

Theorem 1. *For any network $G = (N, A)$, the maximum flow in (1) equals the minimum cut (3).*

The theorem is connected to the *Ford-Fulkerson* algorithm, that directly calculates the maximum flow by iterative improvement on the current flows until reaching the capacities in some arcs. To be precise, we introduce two basic operations in the algorithm.

- Labeling path:
  - Label the source.
  - Given a labeled node i, label the node j if either
    - the arc (i,j) is a *forward arc*: $0 \leq x_{ij} < u_{ij}$; or
    - the arc (j,i) is a *backward arc*: $0 < x_{ij} \leq u_{ij}$.
  - Repeat the previous step until reaching the sink or no more nodes can be labeled.
- Augmenting flow:
  - Decrease the flow that would be feasible for all backward arcs:

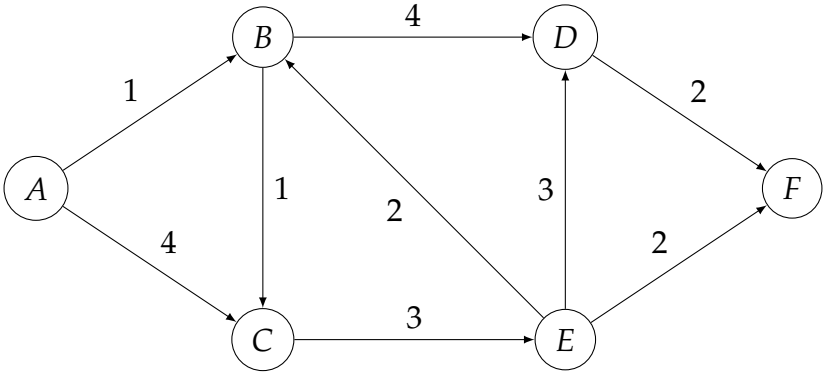
$$\Delta_{\mathrm{b}} := \min\{x_{ij} : (i,j) \text{ is a backward arc}\}.$$

  - Increase the flow that would be feasible for all forward arcs:

$$\Delta_{\mathrm{f}} := \min\{u_{ij} - x_{ij} : (i,j) \text{ is a forward arc}\}.$$

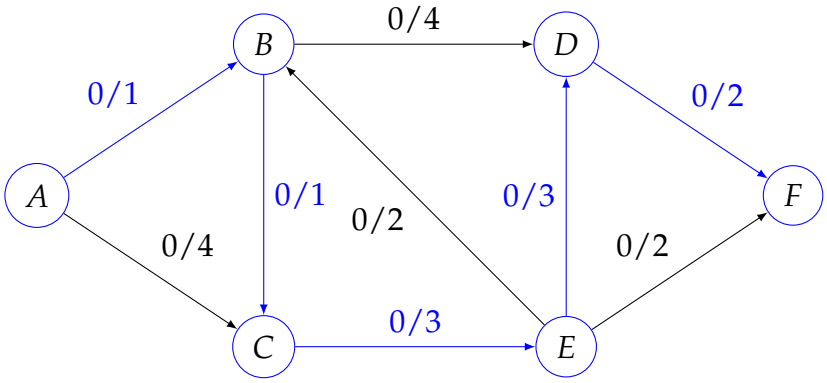
- Let $\Delta := \min\{\Delta_b, \Delta_f\}$. We can then improve the current solution by sending Δ units of flow from source to sink via the augmenting path:
  - increase the flow for all forward arcs in the path by Δ, and
  - decrease the flow for all backward arcs in the path by Δ.

Example 6. *We want to find the maximum flow from A to F, where the capacities are labeled on the arcs.*



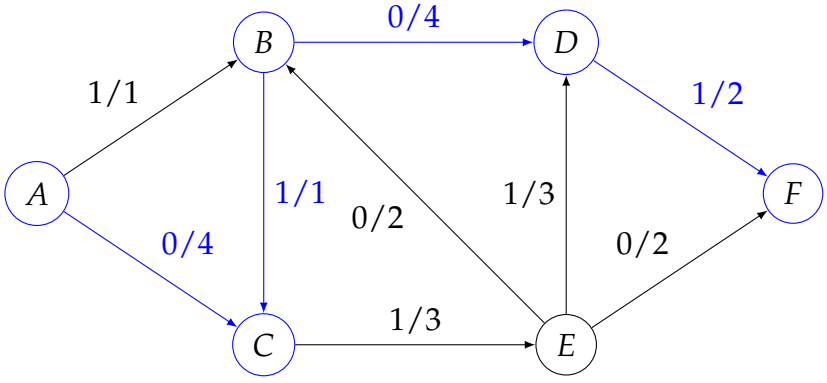
The iterations of Ford-Fulkerson algorithm are executed as follows.

Iteration 1: Labeling path A,B,C,E,D,F



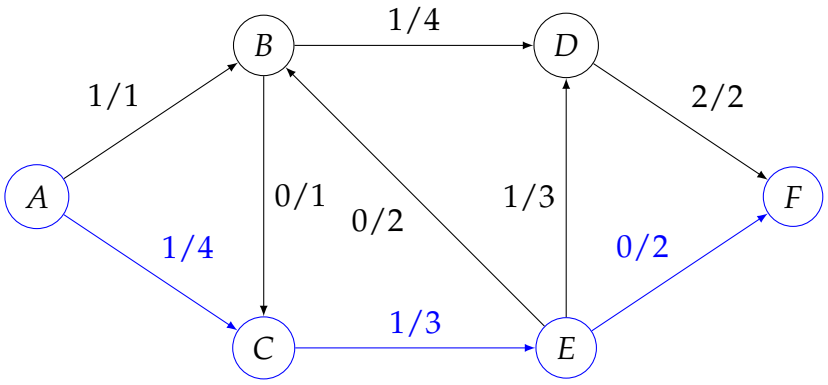
Augmenting flow $\Delta = \min\{1, 1, 3, 3, 2\} = 1$

Iteration 2: Labeling path A,C,B,D,F



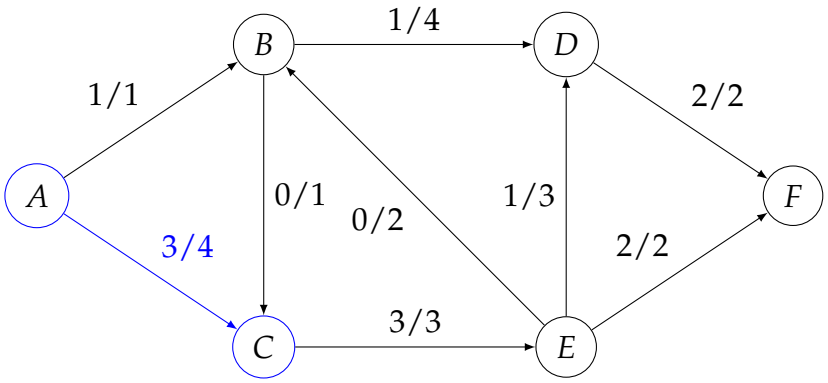
Augmenting flow $\Delta = \min\{4, 1, 4, 1\} = 1$

Iteration 3: Labeling path A,C,E,F



Augmenting flow $\Delta = \min\{3, 2, 2\} = 2$

Iteration 4: Labeling path A,C, and no more nodes can be labeled



Terminate the algorithm. The maximum flow from A to F is 4.

To see the connection to the minimum cut problem, we construct the cut S, T when the algorithm terminates. Let S be all the nodes that is still connected to the source with the remaining capacities. Clearly, S does not contain the sink. All arcs from S to T contain all the flows from the source to the sink, the sum of which equal to the capacity of this cut. Thus by LO duality, we know that S, T is a minimum cut.

3 Shortest Path Problem

Given a directed graph $G = (N, A)$ with each arc (i, j) associated with some *costs* c_{ij}, we want to find a path from a node $s \in N$ to a node $t \in N$ with the minimum total cost. Here we use the term "cost" instead of just distance because the shortest path problem can be used for modeling some general decision problems.

Example 7. *Suppose we have purchases a new machine for \$24,000 at time 0. The cost of maintaining the machine during a year and its trade-in price are given in the table below.*

Age	*Annual Maintenance Cost ($)*	*Age*	*Trade-in Price ($)*
0	*4,000*	*1*	*14,000*
1	*8,000*	*2*	*12,000*
2	*10,000*	*3*	*4,000*
3	*18,000*	*4*	*2,000*
4	*24,000*	*5*	*0*

Assume that at any time it costs $24,000 to purchase a new machine. Our goal is to minimize the net cost incurred during the next five years. We can model this problem using a graph.

- *Our network will have six nodes corresponding to the beginning of years 1-6.*
- *Node i is the beginning of year i and for $i < j$, an arc (i, j) corresponds to purchasing a new machine at the beginning of year i and keeping it until the beginning of year j.*
- *The cost on the arc (i, j) is the total net cost incurred from years i to j:*

$$\begin{aligned} c_{ij} = &\text{ maintenance cost incurred during years } i, i+1, \ldots, j-1 \\ &+ \text{ cost of purchasing a machine at the beginning of year } i \\ &- \text{ trade-in value received at the beginning of year } j. \end{aligned}$$

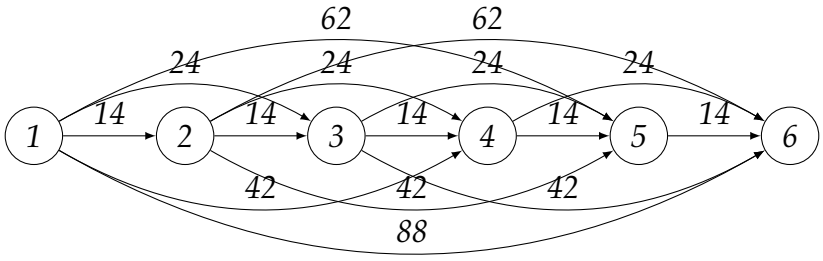
For example, (in $1,000)

$$c_{12} = 4 + 24 - 14 = 14,$$

and

$$c_{26} = 4 + 8 + 10 + 18 + 24 - 2 = 62.$$

The network is then plotted as follows.



Then the maintenance problem reduces to finding a shortest/cheapest path from node 1 *to node* 6.

Any shortest path problem can be formulated as a MILO model. Define for each $(i, j) \in A$,

$$x_{ij} = \begin{cases} 1 & \text{if the arc } (i, j) \text{ is selected in the path,} \\ 0 & \text{otherwise,} \end{cases}$$

and for each $i \in N \setminus \{s, t\}$

$$y_i = \begin{cases} 1 & \text{if the node } (i, j) \text{ is visited in the path,} \\ 0 & \text{otherwise.} \end{cases}$$

There is one outgoing arc for the starting node

$$\sum_{j:(s,j)\in A} x_{sj} = 1,$$

one incoming arc for the terminating node

$$\sum_{j:(j,t)\in A} x_{jt} = 1,$$

and one incoming arc and one outgoing arc for each visited node

$$\sum_{j:(j,i)\in A} x_{ji} = y_i,$$
$$\sum_{j:(i,j)\in A} x_{ij} = y_i.$$

The objective is to minimize the total cost

$$\min \quad \sum_{(i,j)\in A} c_{ij} x_{ij}.$$

Alternatively, we can also solve the shortest path problem efficiently through specialized algorithms.

3.1 Dijkstra's Algorithm

The algorithm can be described as follows.

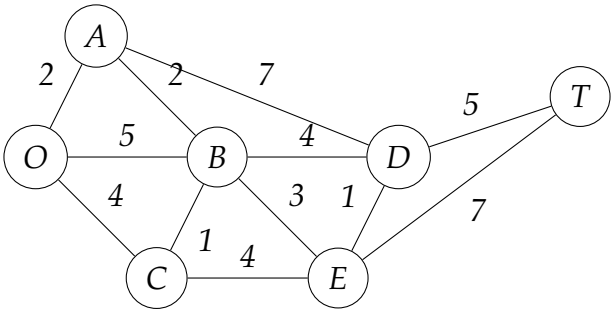
(i) Mark s as solved with cost 0, and all other nodes as unsolved.

(ii) For any node adjacent to the last solved node, calculate the sum of the edge cost and the minimum cost of the last solved node, and
- if it is not a candidate, set it to be a candidate with tentative cost being the sum;
- otherwise, if the sum is smaller than the incumbent tentative cost, update the tentative cost and predecessor to be the sum and the last solved node.

(iii) Pick a candidate node with the smallest tentative cost and mark it as solved.
- If the destination is solved or if there is no more candidate node, terminate the algorithm.

- Otherwise, go back to Step 2.

Example 8. *We want to find a shortest path from node O to node T. The cost of each edge is marked on the graph.*



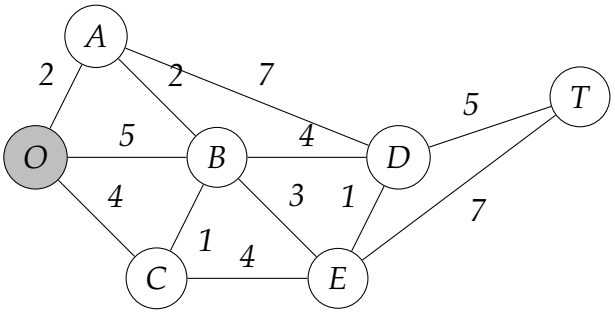
The iterations are illustrated as follows.

Iteration 1: solved nodes (with min. cost, predecessor)

- *O*, $(0, N/A)$

candidate nodes (with tentative cost, predecessor)

- *A (2, O)*
- *B (5, O)*
- *C (4, O)*

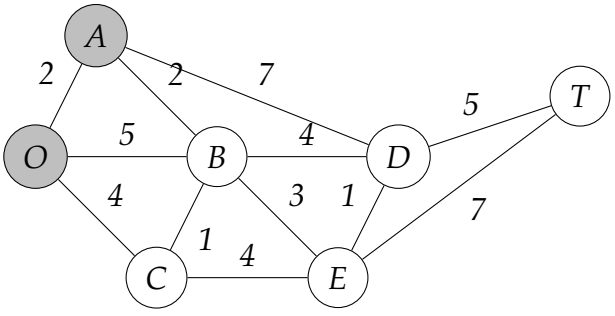


Iteration 2: solved nodes

- *O (0, N/A)*
- *A (2, O)*

candidate nodes

- *B (5, O)* $\rightarrow$ *(4, A)*
- *C (4, O)*
- *D (9, A)*

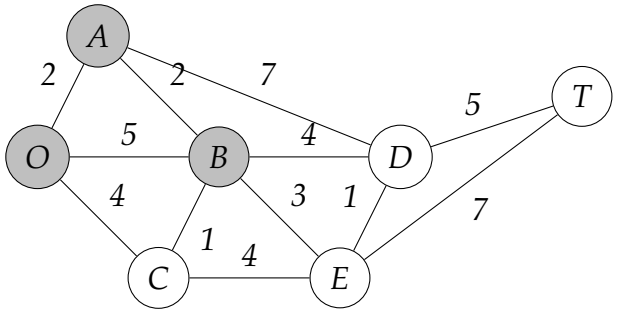


Iteration 3: solved nodes

- *O (0, N/A)*
- *A (2, O)*
- *B (4, A)*

candidate nodes

- *C (4, O)*
- *D (9, A)* $\rightarrow$ *(8, B)*
- *E (7, B)*

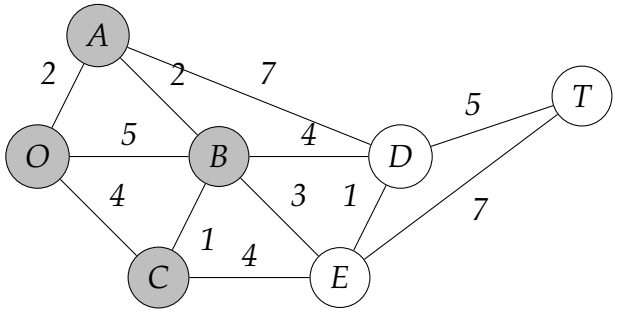


Iteration 4: solved nodes

- *O (0, N/A)*
- *A (2, O)*
- *B (4, A)*
- *C (4, O)*

candidate nodes

- *D (8, B)*
- *E (7, B)*



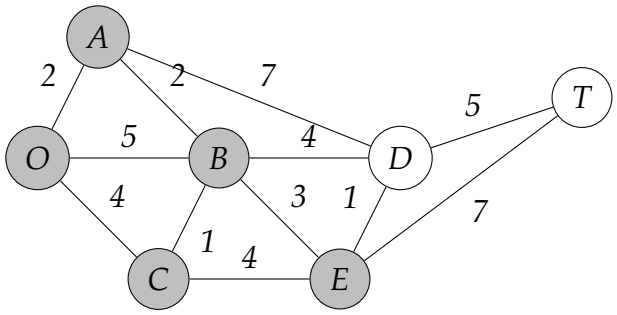
Iteration 5: solved nodes

- *O (0, N/A)*
- *A (2, O)*
- *B (4, A)*
- *C (4, O)*
- *E (7, B)*

candidate nodes

- *D (8, B)*

- *T (14, E)*

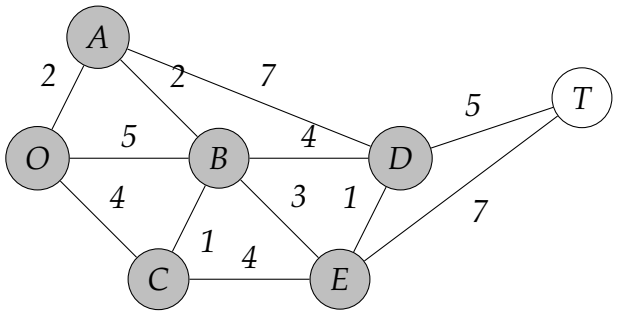


Iteration 6: solved nodes

- *O (0, N/A)*
- *A (2, O)*
- *B (4, A)*
- *C (4, O)*
- *E (7, B)*
- *D (8, B)*

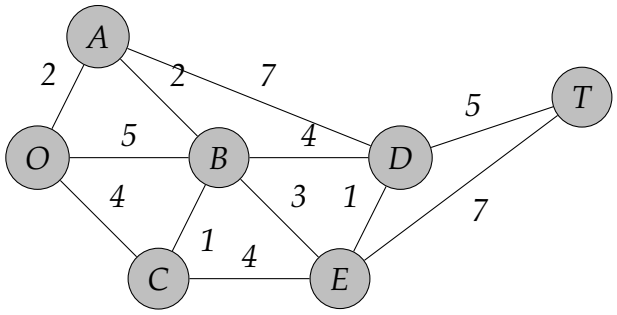
candidate nodes

- *T (14, E) → (13, D)*



Iteration 7: solved nodes

- *O (0, N/A)*
- *A (2, O)*
- *B (4, A)*
- *C (4, O)*
- *E (7, B)*
- *D (8, B)*
- *T (13, D)*



Terminate the algorithm as the target node is solved.

We remark that when there is negative cost on the edges/arcs, Dijkstra's algorithm may terminate prematurely and fail to give the correct answer. Thus we need another algorithm to handle the more general case.

3.2 Bellman-Ford Algorithm

The algorithm can be described as follows.

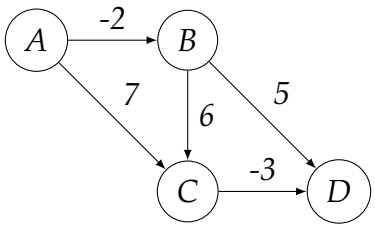
(i) Mark the cost of s as 0, and those of other nodes as $+\infty$.

(ii) Repeat the following step for $|V| - 1$ times.

- For each $(i, j) \in A$, if the current total cost of j is greater than the sum of the current total cost of i plus the cost on the arc (i, j),
  - update the cost of j to be the sum and set its predecessor to be i.

The cost we calculate in each step k is the minimum cost of node of a path from s to i connected with at most k arcs.

Example 9. *We want to find a cheapest path from node A to node D in the following directed graph.*



Iteration 0:

Node	*Cost*	*Predecessor*
A	0	N/A
B	$+\infty$	N/A
C	$+\infty$	N/A
D	$+\infty$	N/A

Iteration 1:

Node	*Cost*	*Predecessor*
A	0	N/A
B	-2	A
C	7	A
D	$+\infty$	N/A

Iteration 2:

Node	*Cost*	*Predecessor*
A	0	N/A
B	-2	A
C	4	B
D	3	B

Iteration 3:

Node	*Cost*	*Predecessor*
A	0	N/A
B	-2	A
C	4	B
D	1	C

We see that the minimum cost from A to D is 1, and the path is A, B, C, D. Note that the Dijkstra's algorithm, if applied in this case, would falsely terminate at the second iteration with a total cost 3.

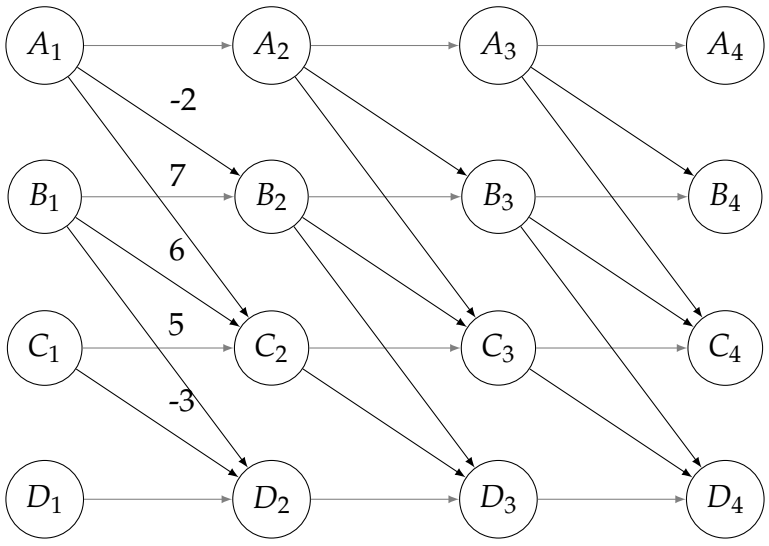
In general, the Bellman-Ford algorithm can be applied to any (directed) graph without a negative-cost cycle. We see below an alternative explanation why we need to execute the calculation step for $|V| - 1$ times.

4 Dynamic Programming

The shortest path problem can be reformulated on a *directed acyclic graph* (DAG). Let $\tilde{G} = (\tilde{V}, \tilde{A})$ such that

- $\tilde{V}$ consists of v_k for each $v \in V$ and $k = 1, \ldots, K := |V|$;
- the arcs $\tilde{A}$ consists of
  - (v_{k-1}, v_k) with cost 0 for each $v \in V$ and $k = 2, \ldots, K$,
  - and (v_{k-1}, u_k) with the cost c_{uv} for each $(u, v) \in A$ and $k = 2, \ldots, K$.

For example, for Example 9, the DAG can be plotted as follows.



We now want to find a path with minimum cost from s_1 to t_K in $\tilde{G}$. To do this, notice that the vertices are now grouped into *stages* $k = 1, \ldots, K$, such that arcs emanating from stage k vertices terminate in stage $k+1$. For each vertex v_k in stage $k = 2, \ldots, K$, we can write

$$c(v_k) = \min\{c_{uv} + c(u_{k-1}) : u \in V\}.$$

This is known as the *Bellman equation of dynamic programming* and corresponds to exactly what we did in the Bellman-Ford algorithm.

Dynamic programming (DP) is a very useful algorithmic framework for many optimization problems, where

- the decisions are made in different *stages* $t = 1, \ldots, T$;
- in each stage there are multiple *states* $v \in V_t$ that contain all information impacting our decision in that stage;
- once a decision is made, the transition into the next stage is known $(u, v) \in A_t$;
- the cumulative cost f_t starting from stage t can be described by recursions

$$f_t(u) = \min_{v \in V_{t+1}} \{c_{uv} + f_{t+1}(v)\}.$$

The Bellman equation allows us to solve many discrete optimization problems recursively.

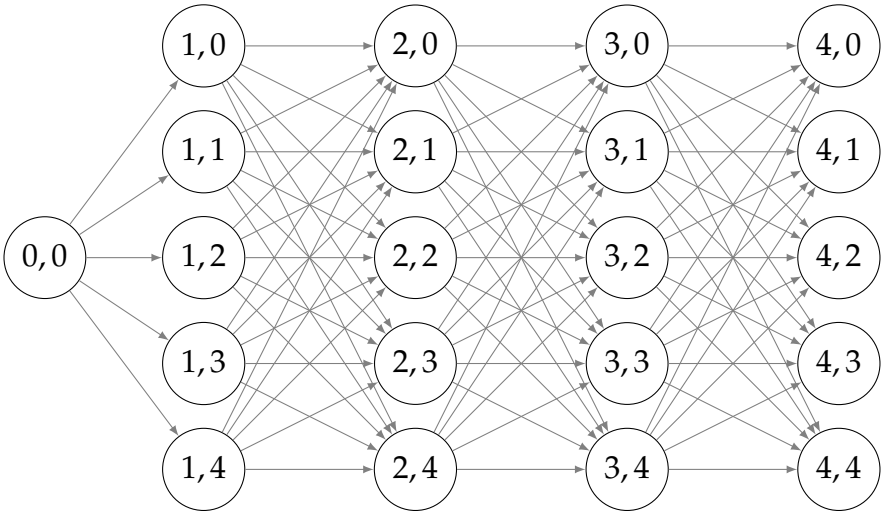
Example 10. *A company knows that the demand for its product during each of the next four months will be as follows:*

Month	*Demand*
1	*1*
2	*3*
3	*2*
4	*4*

At the beginning of each month, the company must determine how many units should be produced during the current month. During a month in which any units are produced, a setup cost of \$3 is incurred. In addition, there is a variable cost of \$1 for every unit produced. At the end of each month, a holding cost of \$0.5 per unit on hand is incurred. Capacity limitations allow a maximum of 5 units to be produced during each month. The size of the companys warehouse restricts the ending inventory for each month to 4 units at most. The company wants to determine a production schedule that will meet all demands on time and will minimize the sum of production and holding costs during the four months. Assume that 0 units are on hand at the beginning of the first month.

Here, we can index the stages by 1, 2, 3, 4 (one for each month), and the state by 0, 1, 2, 3, 4, which represents the inventory at the end of the month. The transition is given by the fact that the inventory at the end of month t equals the inventory at the end of month $t-1$ plus

the production in month t minus the demand in month t. We may use a DAG to represent the problem, where every vertex is denoted by a stage-state pair, and an additional vertex $(0,0)$ is added to denote the initial inventory at hand.



To solve the problem through DP recursion (Bellman equation), note that

$$f(4,0) = \cdots = f(4,4) = 0,$$

and

$$f(3,1) = \min_{p=3,4,5}\{f(4,1+p-4) + 3\cdot\mathbb{1}(p>0) + p + 0.5\},$$

because the demand in month 4 is 4. Similarly, this gives us

$$\begin{aligned}
f(3,0) &= \min\{3+4, 3+5\} = 7, \text{ with } p=4,\\
f(3,1) &= \min\{3+3+0.5, 3+4+0.5, 3+5+0.5\} = 6.5, \text{ with } p=3,\\
f(3,2) &= \min\{3+2+1, \ldots, 3+5+1\} = 6, \text{ with } p=2,\\
f(3,3) &= \min\{3+1+1.5, \ldots, 3+5+1.5\} = 5.5, \text{ with } p=1,\\
f(3,4) &= \min\{2, 3+1+2, \ldots, 3+4+2\} = 2, \text{ with } p=0.
\end{aligned}$$

We can now calculate $f(2,i)$ for $i = 0,1,\ldots,4$ by

$$f(2,i) = \min_{p\le 5}\{f(3,i+p-2) + 3\cdot\mathbb{1}(p>0) + p + 0.5i : 0 \le i+p-2 \le 4\}.$$

Plugging in the values, we get

$$
\begin{aligned}
&f(2,0) = \min\{7+3+2, 6.5+3+3, \ldots, 5.5+3+5\} = 12, \text{ with } p = 2,\\
&f(2,1) = \min\{7+3+1+0.5, \ldots, 5.5+3+4+0.5, 2+3+5+0.5\} = 10.5, \text{ with } p = 5,\\
&f(2,2) = \min\{7+1, \ldots, 5.5+3+3+1, 2+3+4+1\} = 8, \text{ with } p = 0,\\
&f(2,3) = \min\{6.5+1.5, 6+3+1+1.5, \ldots, 2+3+3+1.5\} = 8, \text{ with } p = 0,\\
&f(2,4) = \min\{6+2, 5.5+3+1+2, 2+3+2+2\} = 8, \text{ with } p = 0.
\end{aligned}
$$

Then, we can now calculate $f(1,i)$ for $i = 0, 1, \ldots, 4$ by

$$f(1,i) = \min_{p \le 5}\{f(2, i+p-3) + 3 \cdot \mathbb{1}(p > 0) + p + 0.5i : 0 \le i + p - 3 \le 4\}.$$

Plugging in the values, we get

$$
\begin{aligned}
&f(1,0) = \min\{18, 17.5, 16\} = 16, \text{ with } p = 5,\\
&f(1,1) = \min\{17.5, 17, 15.5, 16.5\} = 16.5, \text{ with } p = 4,\\
&f(1,2) = \min\{17, 16.5, 15, 16, 17\} = 15, \text{ with } p = 3,\\
&f(1,3) = \min\{13.5, 16, 14.5, 15.5, 16.5\} = 13.5, \text{ with } p = 0,\\
&f(1,4) = \min\{12.5, 14, 15, 16\} = 12.5, \text{ with } p = 0,
\end{aligned}
$$

We can finally calculate $f(0,0)$ by

$$f(0,0) = \min_{p \le 5}\{f(1, p-1) + 3 \cdot \mathbb{1}(p > 0) + p : 0 \le p - 1 \le 4\}.$$

This gives us

$$f(0,0) = \min\{20, 21.5, 21, 20.5, 20.5\} = 20, \text{ with } p = 1.$$

Retrieving the solutions, we should produce 1 unit in month 1, 5 units in month 2, 0 units in month 3, and 4 units in month 4. The total cost is \$20.

Sometimes DP can be applied to problems that do not have a clear stage structure. We illustrate this by the following example of a *knapsack problem* that has been introduced in MILO model and the cutting stock problem in the column generation step.

Example 11. *Suppose we would like to fill a knapsack with capacity of* 10 *kilograms. The items of each type are listed below with their values.*

Type	*Weight (kg)*	*Value ($)*
1	*4*	*11*
2	*3*	*7*
3	*5*	*12*

The goal is to maximize the total value of the items in the knapsack.

Here, we can set the stages to be $t = 1, 2, 3$*, representing the items of types* $t, t+1, \ldots, 3$ *to be filled. Then the state in each stage is denoted by* x*, the remaining capacity in the knapsack. Since we are maximizing the total value, we define our value function in stage* t *as* $f_t(x)$*, which is the maximum value that can be filled in the knapsack with items* $t, t+1, \ldots, 3$*. We have the recursion*

$$f_t(x) = \max_y \{v_t \cdot y + f_{t+1}(x - w_t \cdot y) : x - w_t \cdot y \geq 0\},$$

where v_t, w_t *are the value and the weight of item* t*.*

In stage 3, clearly we have

$$\begin{aligned} &f_3(10) = 24, \text{ with } y_3^* = 2, \\ &f_3(x) = 12, \text{ for } 5 \leq x \leq 9, \text{ with } y_3^* = 1, \\ &f_3(x) = 0, \text{ for } 0 \leq x \leq 4, \text{ with } y_3^* = 0. \end{aligned}$$

In stage 2, $f_2(x) = \max\{7y + f_3(x - 3y) : x - 3y \geq 0\}$*, which gives us*

$$\begin{aligned} f_2(10) &= \max\{24, 19, 14, 21\} = 24, \text{ with } y_2^* = 0, \\ f_2(9) &= \max\{12, 19, 14, 21\} = 21, \text{ with } y_2^* = 3, \\ f_2(8) &= \max\{12, 19, 14\} = 19, \text{ with } y_2^* = 1, \\ f_2(7) &= \max\{12, 7, 14\} = 14, \text{ with } y_2^* = 2, \\ f_2(6) &= \max\{12, 7, 14\} = 14, \text{ with } y_2^* = 2, \\ f_2(5) &= \max\{12, 7\} = 12, \text{ with } y_2^* = 0, \\ f_2(4) &= \max\{0, 7\} = 7, \text{ with } y_2^* = 1, \\ f_2(3) &= \max\{0, 7\} = 7, \text{ with } y_2^* = 1, \\ f_2(x) &= 0, \text{ for } x = 2, 1, 0, \text{ with } y_2^* = 0. \end{aligned}$$

Finally for stage 1, we only need to calculate

$$f_1(10) = \max\{24, 25, 22\} = 25, \text{ with } y_1^* = 1.$$

The optimal knapsack consists of 1 type 1 item and then 2 type 2 items, with the total value being 25.