Lecture 15

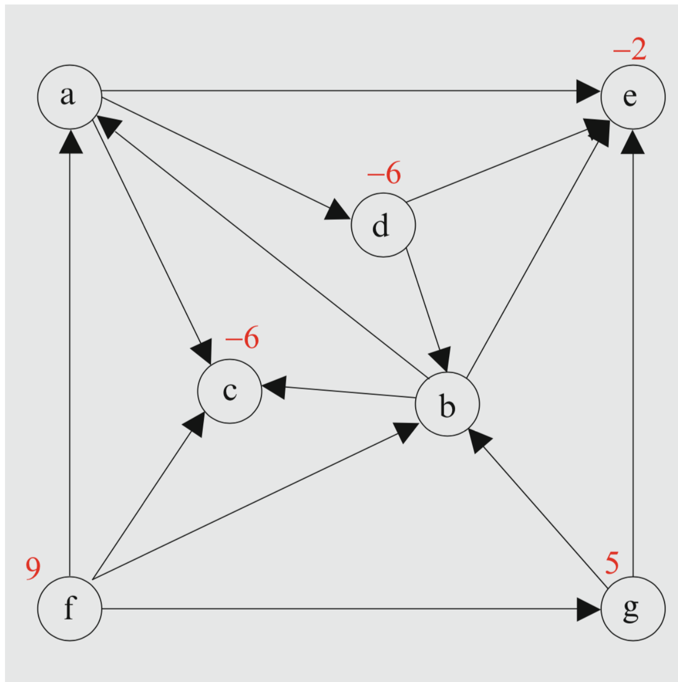
Last time: Network Flow Problems (V14 & GDs notes)

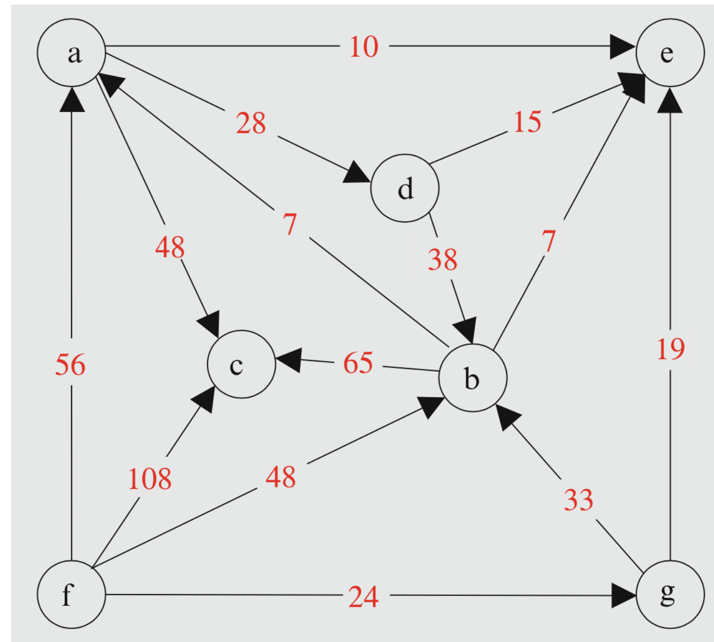
- Tree solutions correspond to basic solutions
- Primal and dual network simplex method
- (Integrality)

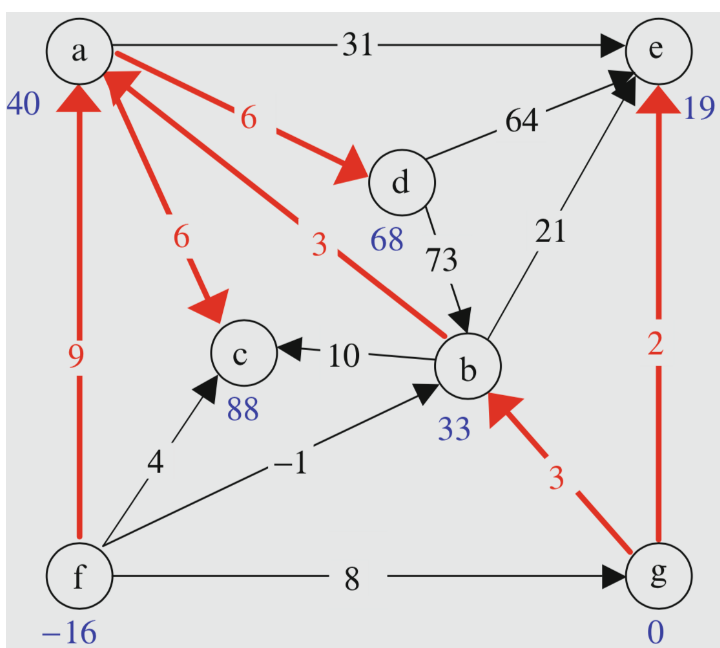
Today:

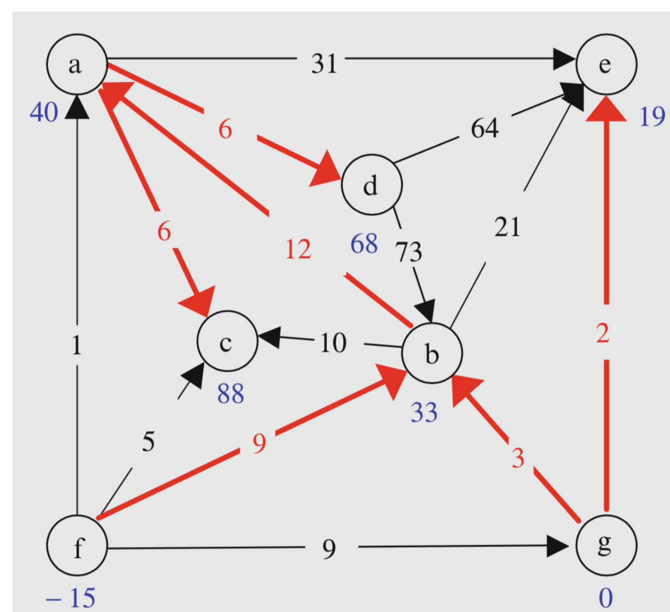
- Integrality (V14)
- Applications (V14-15)
  - > Transportation
  - > Matching/assignment
  - > Shortest paths

Network simplex method - recap









Integrality

THEOREM 14.2. *Integrality Theorem. For network flow problems with integer data, every basic feasible solution and, in particular, every basic optimal solution assigns integer flow to every arc.*

Follows for the algorithm: integer data and no divisions

primal and dual solutions x, y, z are integral

Also valid for problems with capacity

$$0 \le x_{uv} \le a_{uv} \quad (u,v) \in E$$

NB: Integer programming is much harder

THEOREM 14.3. *König's Theorem. Suppose that there are n girls and n boys, that every girl knows exactly k boys, and that every boy knows exactly k girls. Then n marriages can be arranged with everybody knowing his or her spouse.*

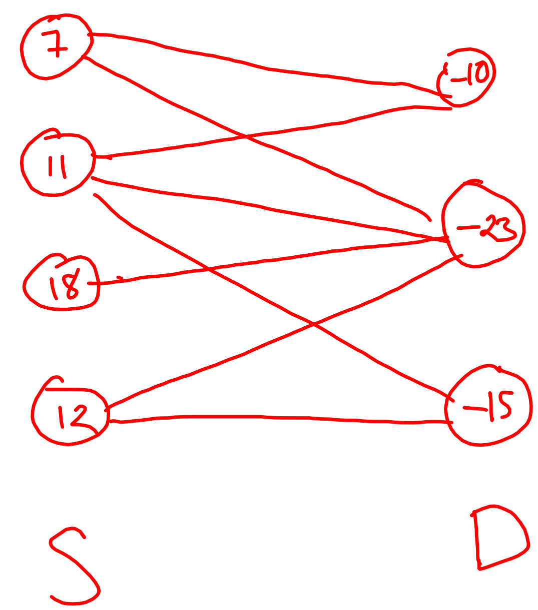
Transportation problem

$V = S \cup D \qquad S \cap D = \emptyset$

Bipartite graph

$b_i \geq 0$ for $i \in S$

$b_i \leq 0$ for $i \in D$



LP-formulation

$$\min \sum_{i \in S} \sum_{j \in D} c_{ij} x_{ij}$$

$$\text{s.t.} \quad \sum_{j \in D} x_{ij} = b \qquad i \in S$$

$$\sum_{i \in S} x_{ij} = -b \qquad j \in D$$

$$x_{ij} \geq 0 \qquad i \in S, j \in D$$

Integral solution

Tabular notation: supply (rows), demand (cols) and edge costs

	-10	-23	-15
7	5	6	*
11	8	4	3
18	*	9	*
12	*	3	6

Solution: x (boxed) and z (not boxed), * means no edge

	-10	-23	-15
7	[7]	5	*
11	[3]	[8]	-4
18	*	[18]	*
12	*	[-3]	[15]

Matching/assignment problem

Special case of transportation

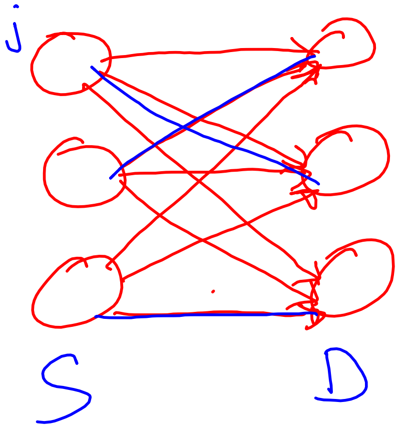
S – n persons

D – n tasks

c_{ij} – cost of assigning person i to task j

problem: assign each person to one and only one task so that all tasks are covered and cost is minimized

Set $x_{ij} = 1$ if i assigned to j



minimize $\sum_{i \in S} \sum_{j \in D} c_{ij} x_{ij}$

So that $\sum_{j \in D} x_{ij} = 1 \quad i \in S$

$\sum_{i \in S} x_{ij} = 1 \quad j \in D$

Transportation problem!

$b_i = 1$ for $i \in S$

$b_j = -1$ for $j \in D$

Can use NSM to obtain integral sol!

Not guaranteed for other versions of the simplex method.

NB: In general $n!$ different solutions, huge number even for moderate size n.

Find optimal solution efficiently using NSM

The shortest path problem (section 15.3)

We study a basic combinatorial problem. But first:

- A (directed) walk in a directed graph $D = (V, E)$ is a sequence
$$P = (v_0, e_1, v_1, e_2, \ldots, e_k, v_k)$$
where $k \geq 0$, $v_i \in V$ ($0 \leq i \leq k$) and $e_i = (v_{i-1}, v_i)$ ($i \leq k$). We say that P goes from v_0 to v_k, and call P a $v_0 v_k$-walk.
- A (directed) path is walk P where $v_0, v_1, \ldots, v_k$ are distinct; it is called a $v_0 v_k$-path.
- The difference is that a walk may contain cycles.

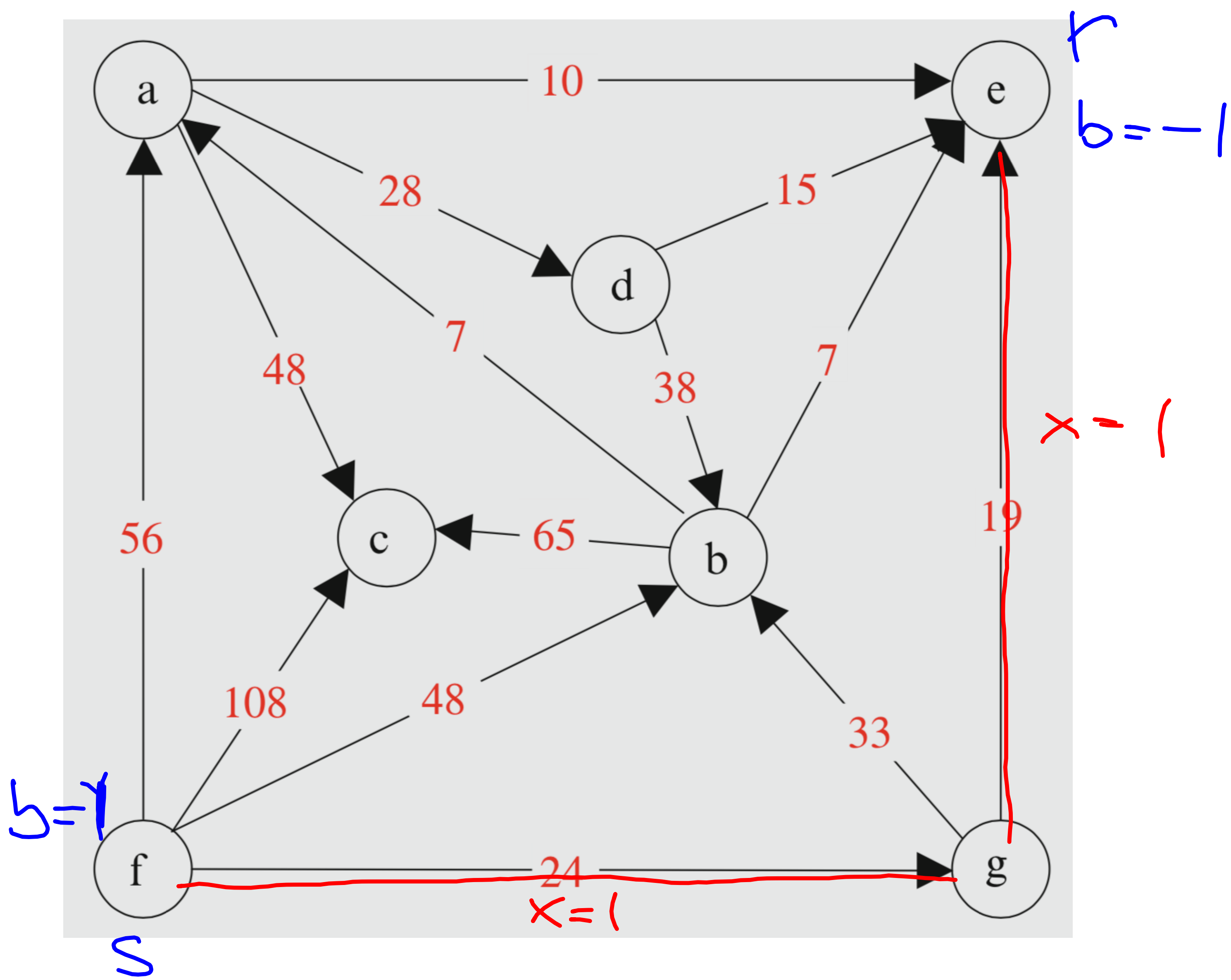
The shortest path problem: given a directed graph $D = (V, E)$ with a nonnegative number (length, weight) c_{ij} for each edge (i, j), and two nodes s and r, find a shortest path P from s to r. Here the length of a path is the sum of the c_{ij}'s for its edges.

Network flow formulation

The shortest path problem is a special case of the minimum cost network flow problem:

$$\min\{c^T x : Ax = -b,\ x \geq O\}.$$

- Here A is the node-edge incidence matrix of the graph, c is the cost vector (the edge lengths), and $b = (b_v : v \in V)$ is the vector given by $b_s = 1$, $b_r = -1$ and $b_v = 0$ otherwise.
- This approach works because there is an integer optimal solution, and the edges with positive flow must contain a path from s to r: $x_{ij} = 1$ for all edges in the path, and $x_{ij} = 0$ otherwise. (If there are edges with zero length, one may get cycles in addition to the path.)
- So one may solve the shortest path problem as a min. cost network flow problem using the network simplex algorithm.
- However, simpler and faster algorithms also exist! We shall discuss two such methods.



The distance between two vertices is the length of the shortest path.

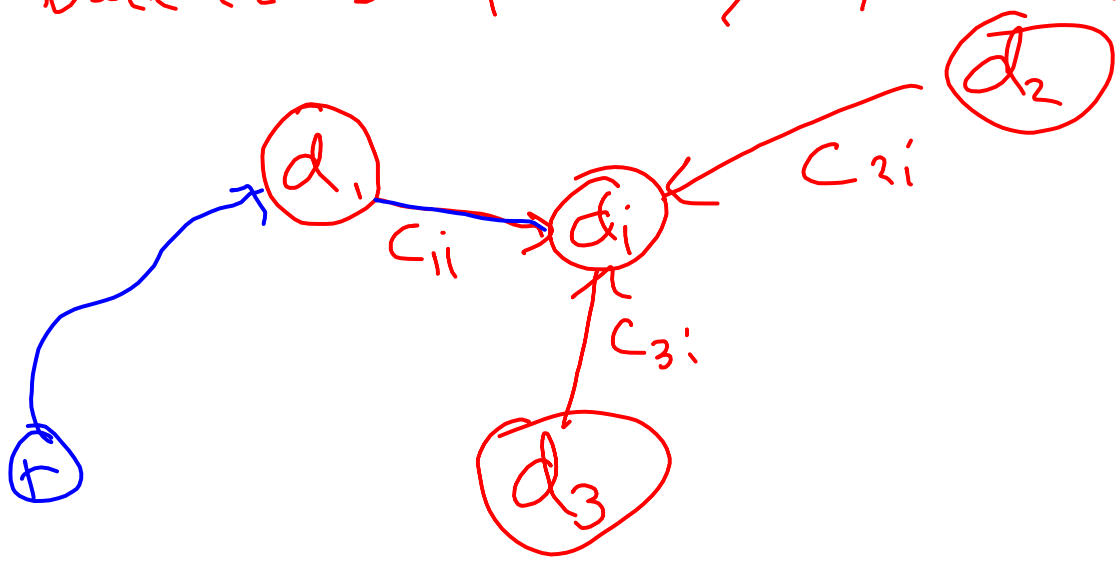
Shortest paths by dynamic programming

Define for each vertex $i \in V$ the distance d_i to the source vertex s, i.e. the length of the optimal path.

Then the distances must satisfy

$$d_i = \min_{j:(j,i)\in E} \{ d_j + c_{ji} \} \quad \text{for all } i \in V$$

Bellmans equation / dynamic programming eq.



The Bellman-Ford algorithm

- For $v \in V$ og $k \geq 0$ (integer), we define $d_k(v)$ as the minimum length of an *sv*-walk with *at most k edges*. If there is no such walk, define $d_k(v) = \infty$.

How can we compute these these distance functions?

The Bellman-Ford's algorithm: *let $d_0(s) = 0$ and $d_0(v) = \infty$ for each $v \neq s$. Compute the functions $d_1, d_2, \ldots, d_n$ by*

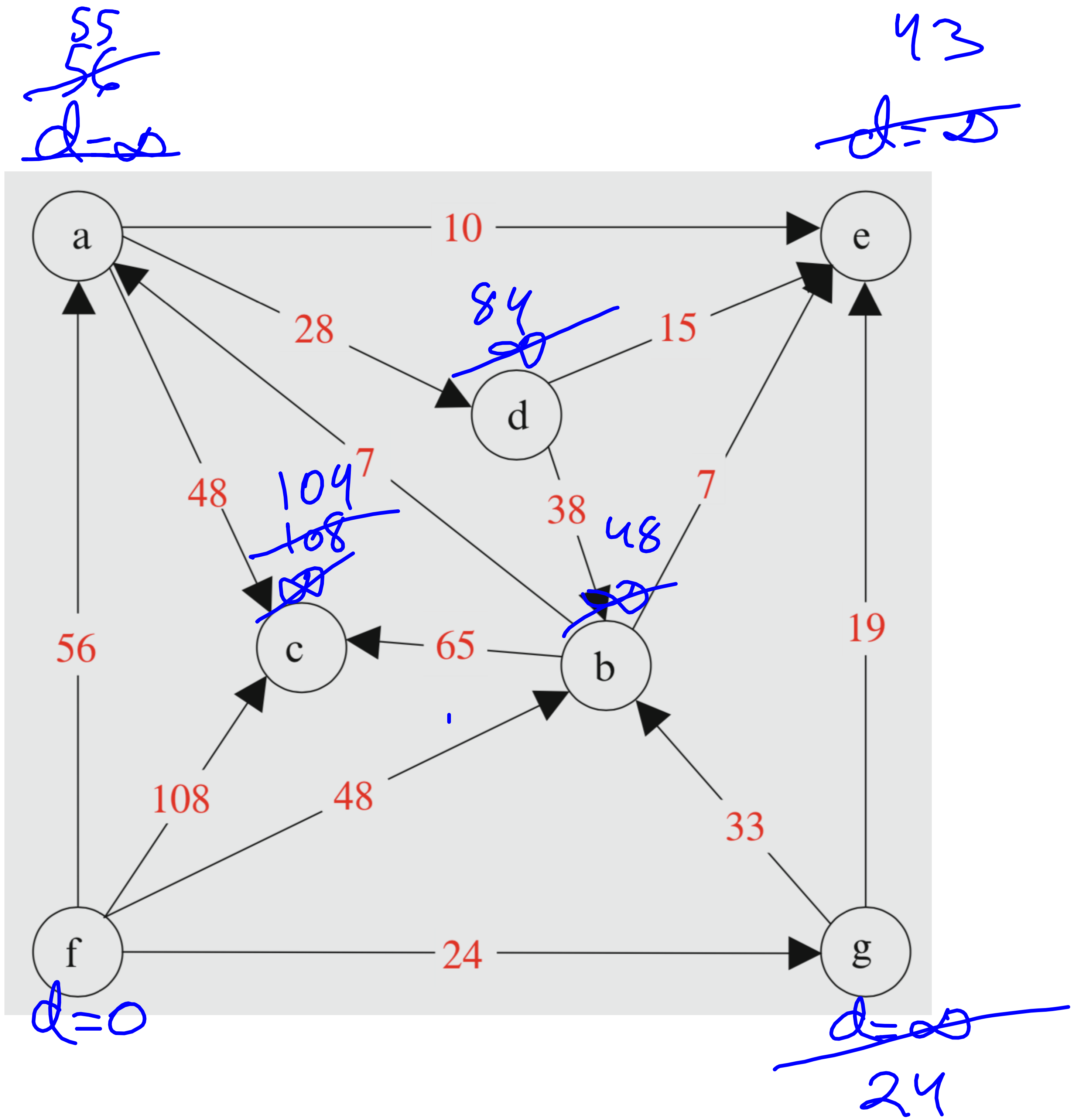
$$d_{k+1}(v) = \min\{d_k(v), \min_{u:(u,v)\in E}(d_k(u) + c_{uv})\} \tag{1}$$

for all $v \in V$.

Theorem: *The Bellman-Ford algorithm finds the correct distances, i.e., $d_k(v)$ becomes the minimum length of an sv-walk with at most k edges. In particular, $d_{n-1}(v)$ is the length of a shortest sv-path (here n is the number of nodes in the graph).*

Proof: A shortest *sv*-path with at most $k+1$ edges has either (i) at most k edges or (ii) it has $k+1$ edges and contains an edge (u, v) as its final edge. But in case (ii) the subpath to u must be a shortest *su*-path with at most k edges (for otherwise we could find another shorter *su*-path and thereby improve the *sv*-path). □

- The equation (1) for computing d_{k+1} based on d_k is called Bellman's equation. It is also used in similar problems called (discrete) *dynamic programmering* or *optimal control* (continuous version); the equation is then called the *Hamilton-Jacobi-Bellman (HJB) equation*.
- The BF-algorithm has complexity (number of arithmetic computations) $O(nm)$ where the graph has n nodes and m edges.
- The algorithm has another important property: it can also be used if there are negative lengths on the edges. The BF algorithm will then decide if there exists a cycle reachable from s with total length which is negative; then $d_n(v) < d_{n-1}(v)$. If this does not happen, the BF algorithm finds a shortest *sv*-path.


a
e
d
c
b
f
g
10
28
15
48
7
38
7
65
56
19
108
48
33
24
d=0

Dijkstra's algorithm

- This is also an algorithm for the shortest path problem.
- It only works for nonnegative edge lengths (which is most common in applications!)
- Dijkstra's algorithm is faster than the Bellman-Ford algorithm.
- Note: our description is slightly different than the one in the book: we start at s and move forward along edges while Vanderbei goes backwards!
- A usual n is the number of nodes.

- The algorithm performs n iterations, in each iteration one node is added to a certain set $\mathcal{F}$ and certain computations are done. At the start $\mathcal{F} = \emptyset$.
- One has a value (a label) d_i, for each node i: d_i is an upper bound on the (shortest) distance from s to i. Initially: $d_s = 0$, and $d_i = \infty$ otherwise. $\mathcal{F}$ consist of the nodes to which one already has found a shortest path, for these nodes d_i is *equal* to the distance from s to i.
- In each iteration:
  1. choose an $i \notin \mathcal{F}$ with d_i smallest possible ("a closest node"), and update $\mathcal{F} := \mathcal{F} \cup \{i\}$.
  2. for each edge $(i, j) \in E$ where $j \notin \mathcal{F}$, set

     $$d_j = \min\{d_j, d_i + c_{ij}\}$$

     and, if d_j was reduced, set a pointer $prev(j) = i$.

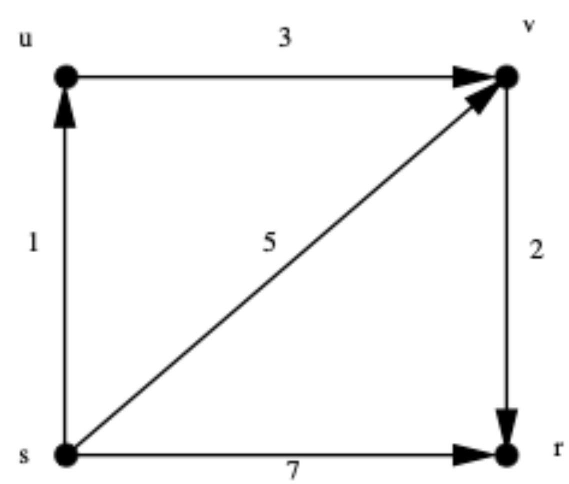
- This means that, at the start of each iteration, $d(v)$ is equal to the length of a *shortest sv-path that only uses nodes in* $\mathcal{F}$.

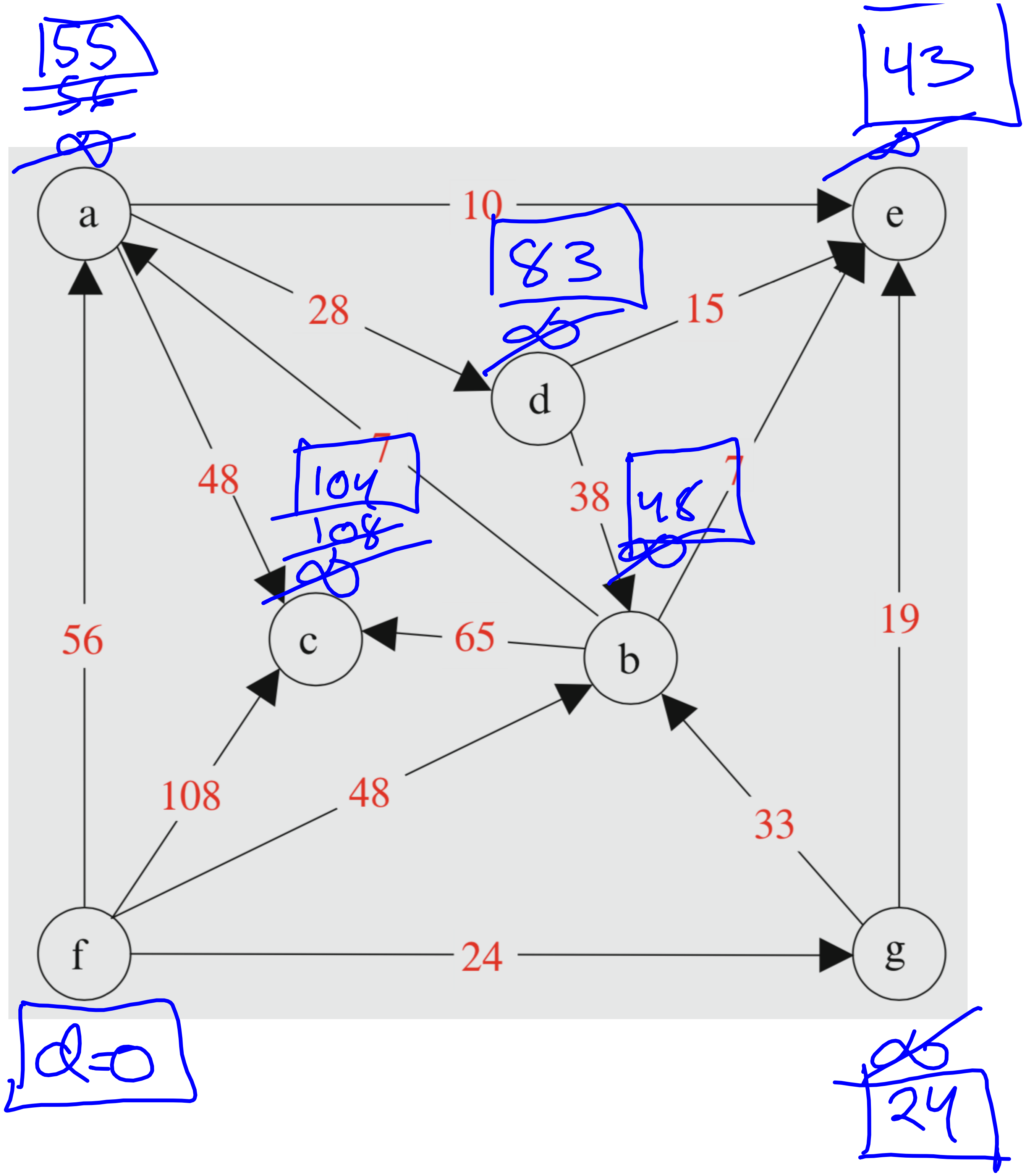
We have (without giving the proof, which is a rather simple induction proof, by the way):

Theorem: *Dijkstra's algorithm finds a shortest path, and corresponding distances* d_v, *from s to each node v. The complexity is* $O(n^2)$.

$O(n \log n)$

Example: use Dijkstra (and Bellman-Ford) here:





a
b
c
d
e
f
g
10
28
15
48
7
38
7
56
65
19
108
48
33
24
155
56
∞
43
∞
83
∞
104
108
∞
48
∞
d=0
∞
24

Shortest paths on meshes (bonus material)

Method using continuous version of Dijkstras algorithm