6CCS3OME/7CCSMOME – Optimisation Methods

Lecture 2

Single-source shortest-paths:

Dijkstra’s algorithm, shortest-paths algorithm for DAGs

Tomasz Radzik and Kathleen Steinhöfel

Department of Informatics, King’s College London

2020/21, Second term

Topics

- Single-source shortest-paths; restricted cases

  - Only non-negative edge weights allowed:

    Dijkstra's shortest-paths algorithm

  - The input graph is acyclic (a DAG – a directed acyclic graph):

    Single-source shortest paths algorithm for DAG's

- In both cases, the Bellman-Ford shortest-paths algorithm can be used.

  In both cases, the new algorithms are faster than the Bellman-Ford algorithm.

Dijkstra's shortest-paths algorithm

- **Crucial assumption:** all edge weights are nonnegative.
- For convenience, assume that all nodes are reachable from s.

node	a	b	c	h	r	s	v	x	y	z
PARENT	nil	nil	nil	nil	nil	nil	nil	nil	nil	nil
d	∞	∞	∞	∞	∞	0	∞	∞	∞	∞

DIJKSTRA(G, w, s) $\{ G = (V, E) \}$

  INITIALIZATION(G, s) { "relaxation technique" initialization }

  $S \leftarrow \emptyset$ { nodes v for which we know that $d[v] = \delta(s, v)$ }

  $\underline{Q \leftarrow V}$ { the other nodes in <u>Priority Queue</u> with keys $d[.]$}

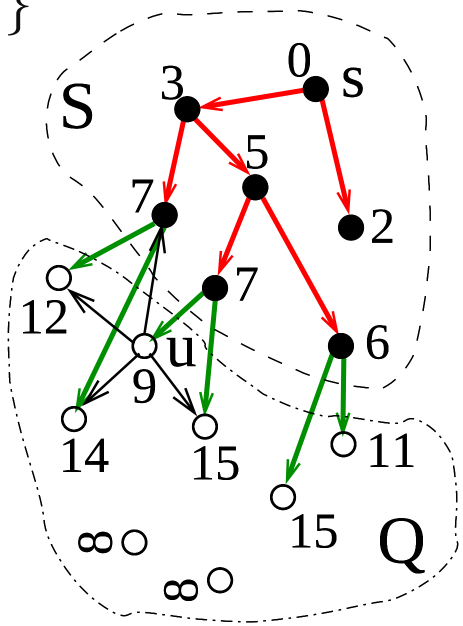
  while $\underline{Q \neq \emptyset}$ **do**

    $u \leftarrow$ EXTRACT-MIN(Q) { u has the min $d[.]$ value in Q }

    $S \leftarrow S \cup \{u\}$

    for each node $v \in Adj[u]$ **do** RELAX(u, v, w)

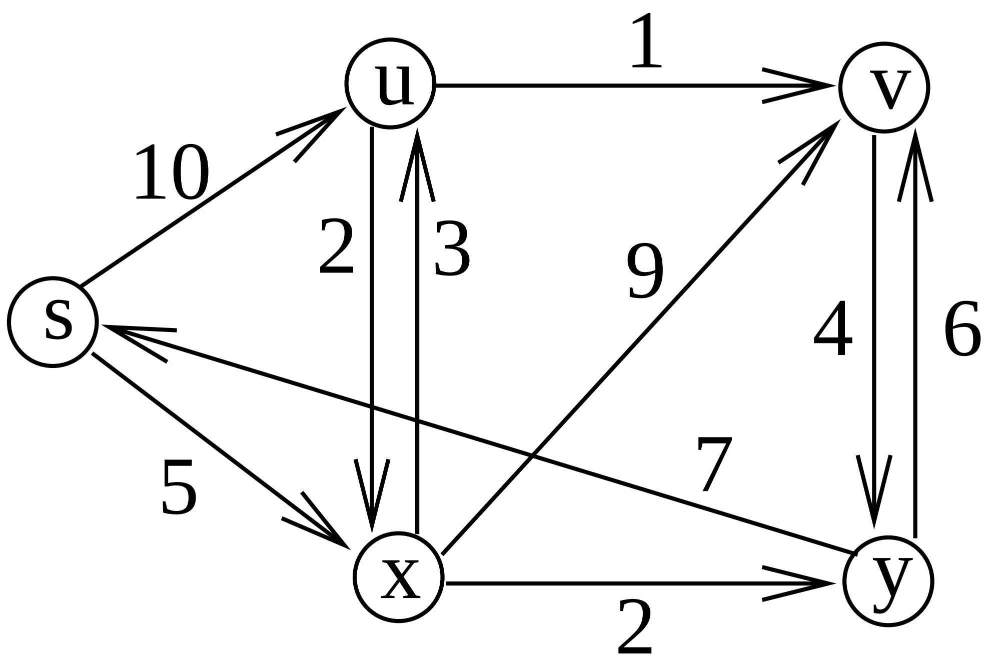
  end_while .



- Q is implemented as a
  Priority Queue data structure.
  Priority Queue maintains pairs (value, key)
  and the main operation is EXTRACT-MIN.
  In Dijkstra's algorithm: pairs (node, d[node]).

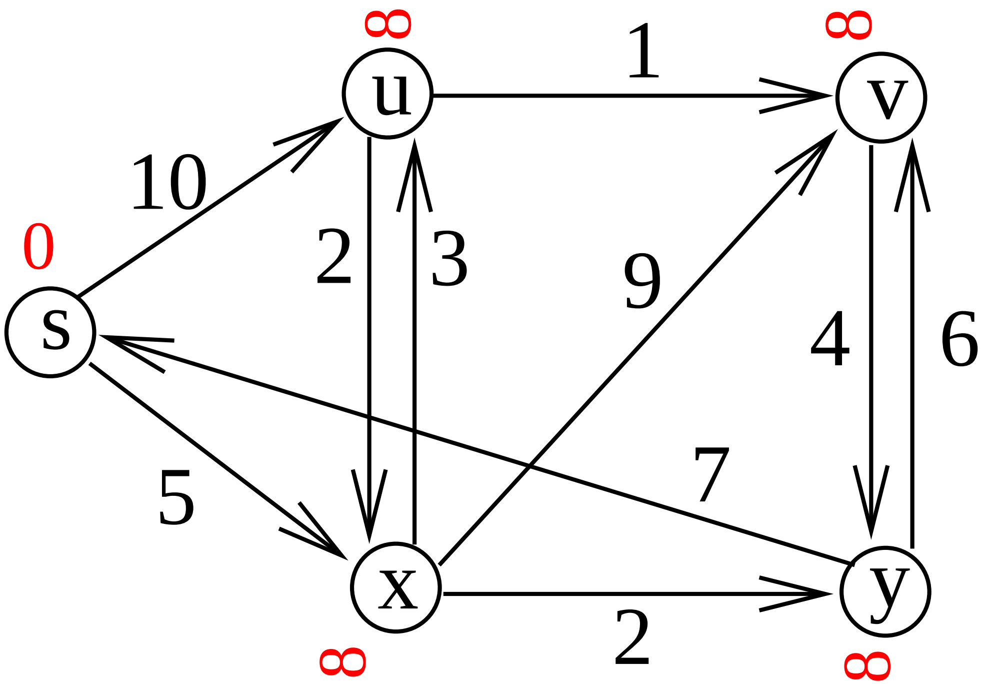
Example

From [CLRS] textbook:



Example: initialization (continued at LGT)

From [CLRS] textbook:

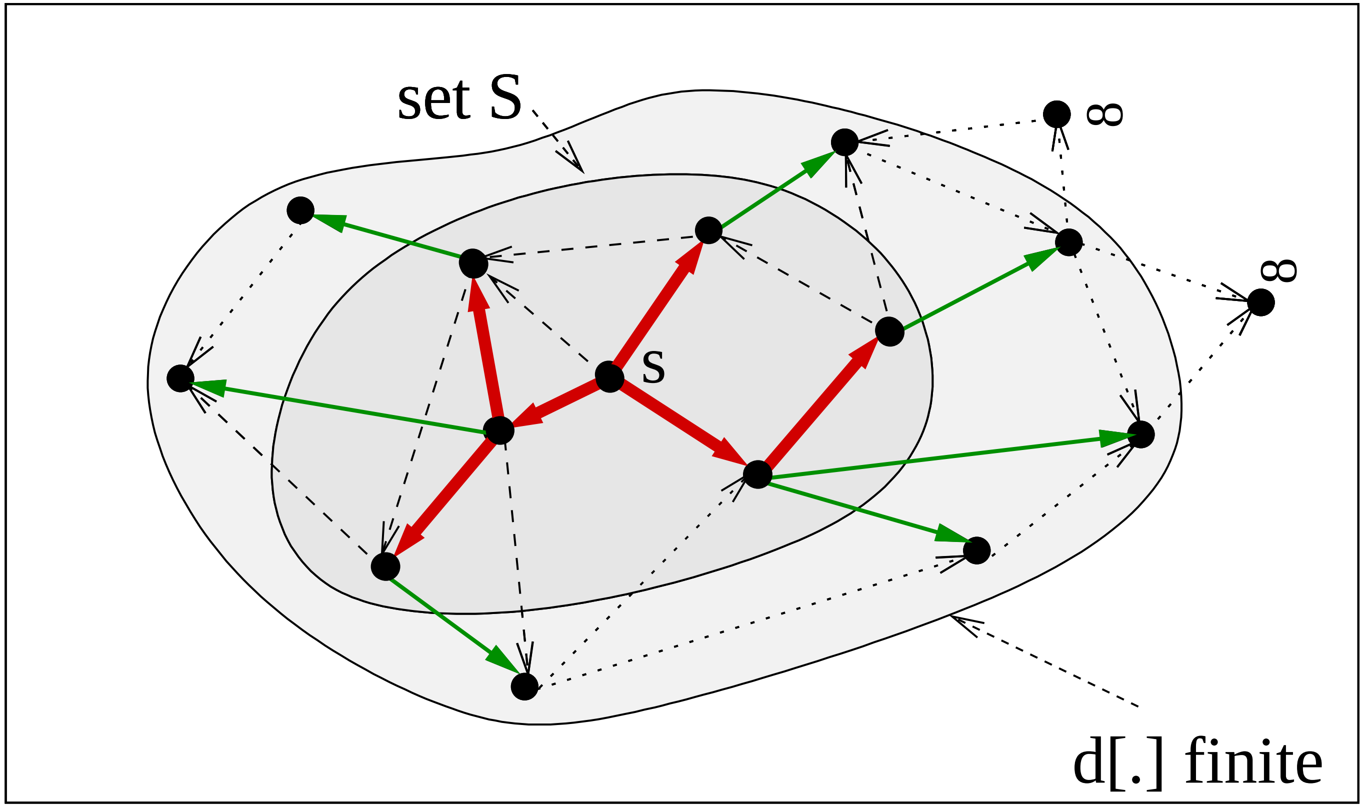


$Q = \{\ (s,0),\ (u,\infty),\ (v,\infty),\ (x,\infty),\ (y,\infty)\ \}$

An intermediate state of the computation (at the end of one iteration).

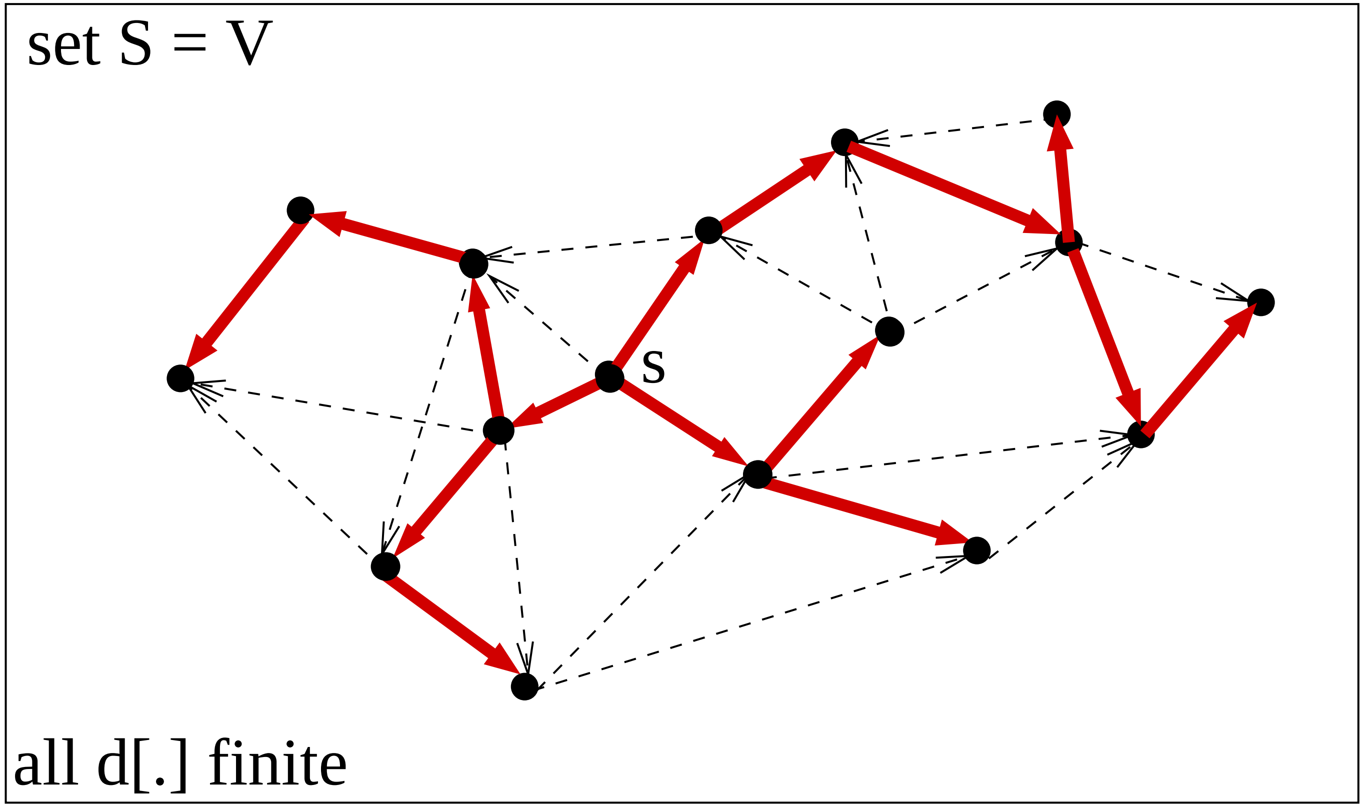
- Set S grows by one node per one iteration

  End 1-st iter.: $S = \{s\}$
  End last iter.: $S = V$



- All edges from nodes in S, and only these edges, have been relax'ed.
- All nodes in S or at the end of edges from S:
  - have finite $d[.]$ values (by induction)
  - have parents, forming a tree (no negative cycles and Prop. (8)).
- For all other nodes, $d[.] = \infty$, and not in the parent tree.

At the end of
the computation:



- Set $S = V$.
  For each $v \in V$, $d[v]$ is finite and $d[v] \geq \delta(s, v)$.
  The parent subgraph is a tree (rooted at s)
  which includes all nodes.

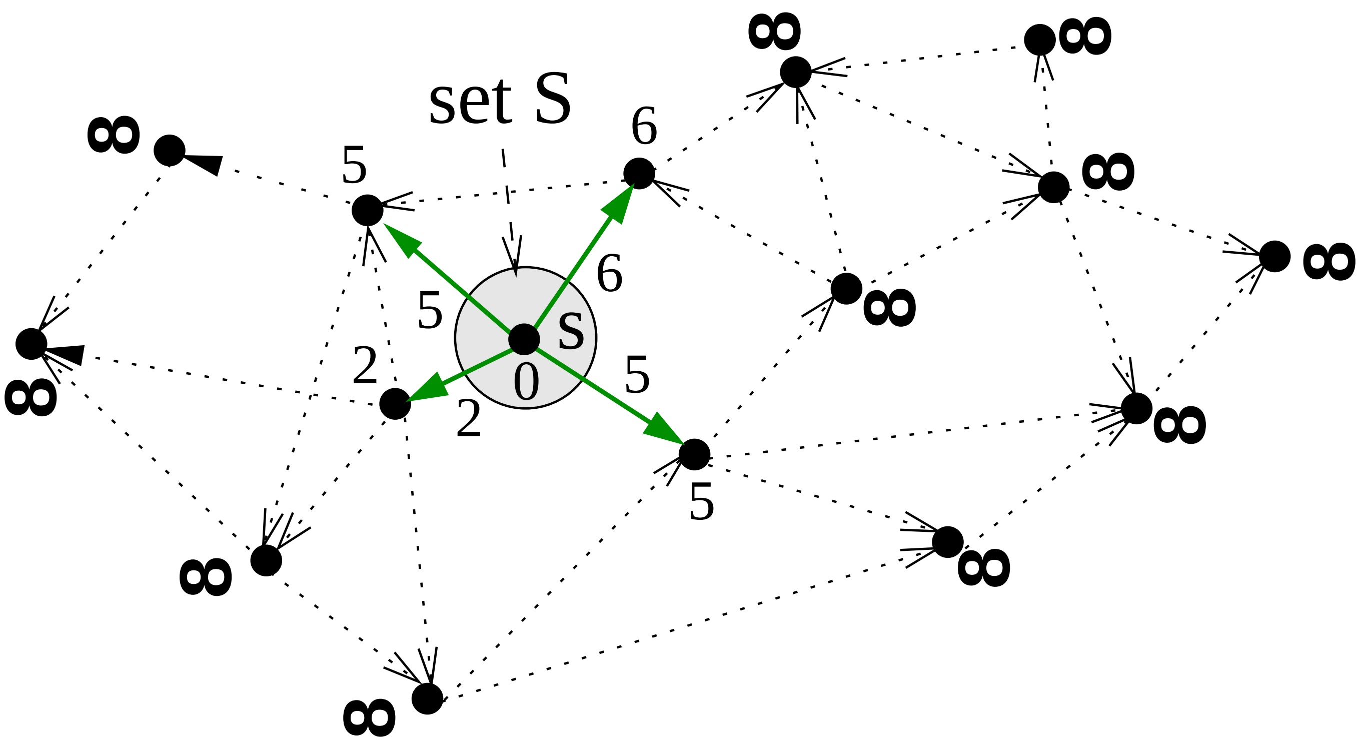
- We haven't shown yet that the computed:
  tree is a shortest paths tree and
  the $d[.]$ values are the shortest-path weights.

The correctness of Dijkstra's algorithm (2): the crucial property

The crucial property (the invariant of the computation of Dijkstra's algorithm):

At the end of each iteration (on the main loop) of Dijkstra's algorithm, for each node $v \in S$, $d[v] = \delta(s, v)$.

- This property can be shown **by induction** on the number of iterations.
- **Basis of the induction:** The property holds at the end of the first iteration: $S = \{s\}$, $d[s] = 0 = \delta(s, s)$.



Induction step:

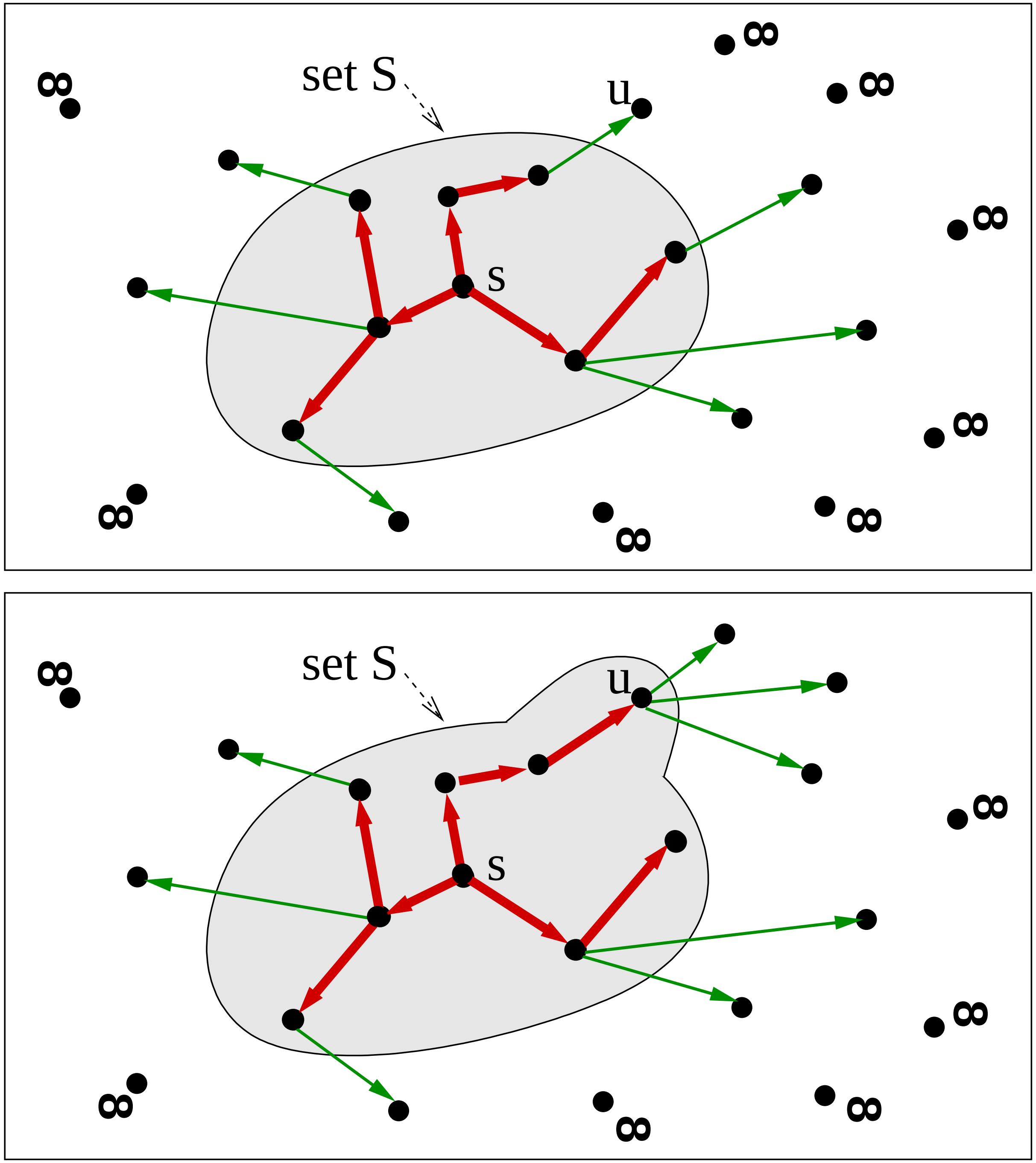
- Assume the invariant holds at the end of some iteration (not the last one):

  for each $v \in S$, $d[v] = \delta(s, v)$.

- Let u denote the node selected in the next (current) iteration.

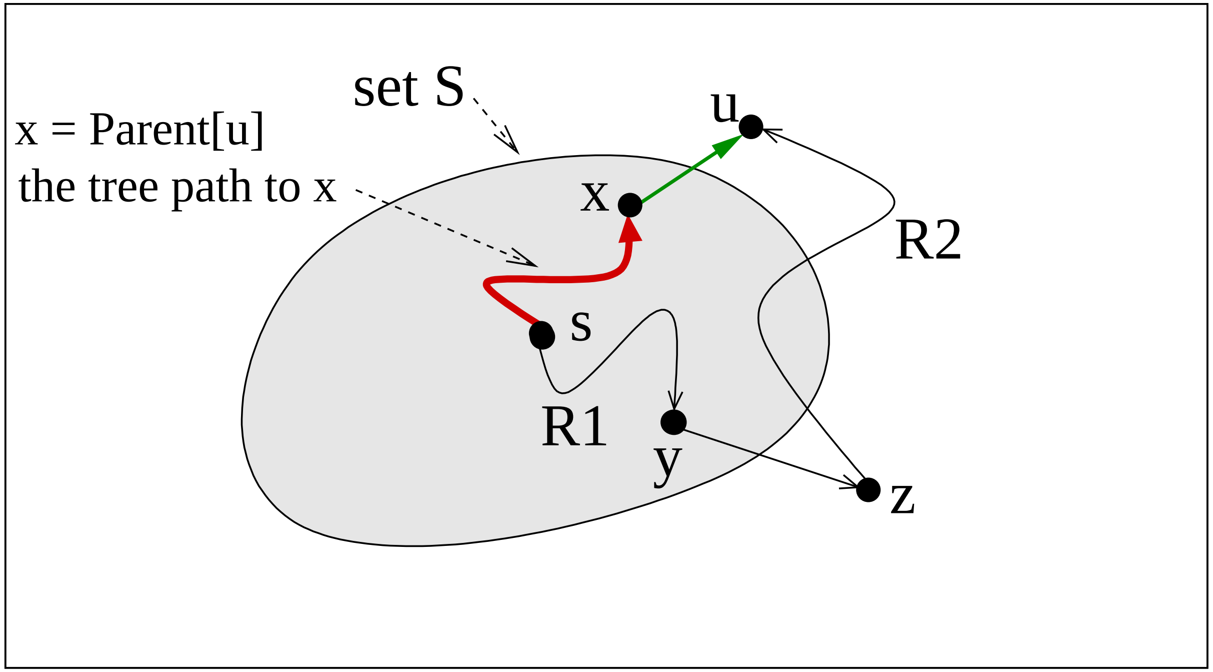
- Show that the invariant holds at the end of the current iteration, (after u added to set S).

- That is, show that $d[u] = \delta(s, u)$.



The invariant of Dijkstra's algorithm: the induction step (cont.)

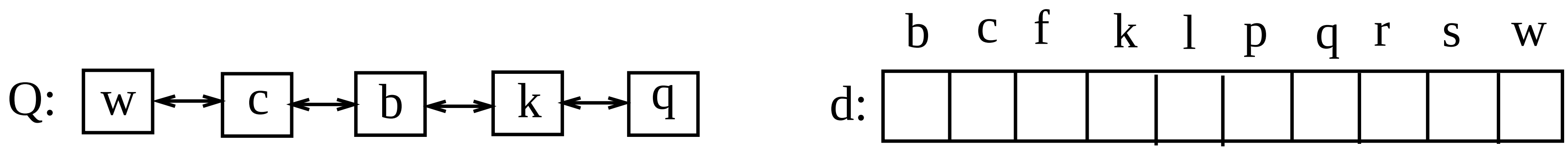
- We have to show that at the beginning of the current iteration, $d[u]$ is the shortest-path weight from s to u. That is, we have to show that $d[u] = \delta(s,u)$.
- We have $d[u] \geq \delta(s,u)$ (relaxation technique). We show $d[u] \leq \delta(s,u)$.
- Take any (simple) path R from s to u and show that its weight $w(R) \geq d[u]$.
- Let z be the first node on path R which is outside S, and let y be the predecessor of z on R (so $y \in S$).



- Let $R1$ be the initial part of path R ending at node y, and let $R2$ be the part of path R from node z to the final node u.
- $$w(R) = w(R1) + w(y,z) + w(R2) \geq d[y] + w(y,z) + w(R2) \geq d[z] + w(R2) \geq d[u] + w(R2) \geq d[u].$$
- The inequalities above follow from: the inductive assumption, RELAX(y,z) done, the rule for selecting u, and the non-negative weights of edges, respectively.
- Thus no path from s to u has smaller weight than $d[u]$, so $d[u] \leq \delta(s,u)$.

The running time of Dijkstra's algorithm

- n – number of nodes; m – number of edges.
- Initialisation (as in the relaxation technique): $\Theta(n)$.
- For every edge (u, v), RELAX(u, v) is performed exactly once.
  This happens during the iteration when node u is removed from Q.
  (Each node is removed from Q exactly once.)
- The running time depends on the implementation of the priority queue Q.
- Note that $n - 1 \leq m \leq n(n-1)$, so $m = \Omega(n)$ and $m = O(n^2)$.
- The naive implementation of Q, using an unordered list of elements:



  - one EXTRACT-MIN(Q): $O(n)$ time; all of them: $O(n^2)$;
  - one RELAX operation: $O(1)$; all of them: $O(m)$;
  - the total running time: $\Theta(n) + O(n^2) + O(m) = O(n^2)$.

- What would be the running time of Dijkstra's algorithm, if Q was maintained as an ordered list? (LGT).

Priority Queue implemented as Heap: operations and performance

- To improve the running time of Dijkstra's algorithm, use the **Heap** implementation of the Priority-Queue to maintain Q.
- The main Priority Queue operations in Heap (n denotes the size of the heap, that is, the number of elements in the heap):
  - $\textsc{Insert}(Q, v, k)$: insert the value-key pair (v, k). $O(\log n)$ time
  - $\textsc{Extract-Min}(Q)$: remove and return the pair with the smallest key. $O(\log n)$ time
  - Check if Heap is not empty. $O(1)$ (constant) time
  - Initialise Heap with given n elements. $\Theta(n)$ time.
- In Dijkstra's algorithm, we also need heap operation:

  $\textsc{HEAP-Decrease-Key}(Q, v, k)$
  – decrease the key associated with v to k.

  $O(\log n)$ time

Dijkstra's algorithm with Heap

- The set Q in Dijkstra's algorithm maintained as **Heap**:
  - Initialisation of the Heap: $\Theta(n)$ time.
  - One $\text{EXTRACT-MIN}(Q)$: $O(\log n)$ time; all: $O(n \log n)$ time.
  - one $\text{RELAX}(u, v)$: $O(\log n)$ time, since it may involve one operation HEAP-DECREASE-KEY; all: $O(m \log n)$ time.
  - The total running time of Dijkstra's algorithm with Heap is: $\Theta(n) + \Theta(n) + O(n \log n) + O(m \log n) = O(m \log n)$.

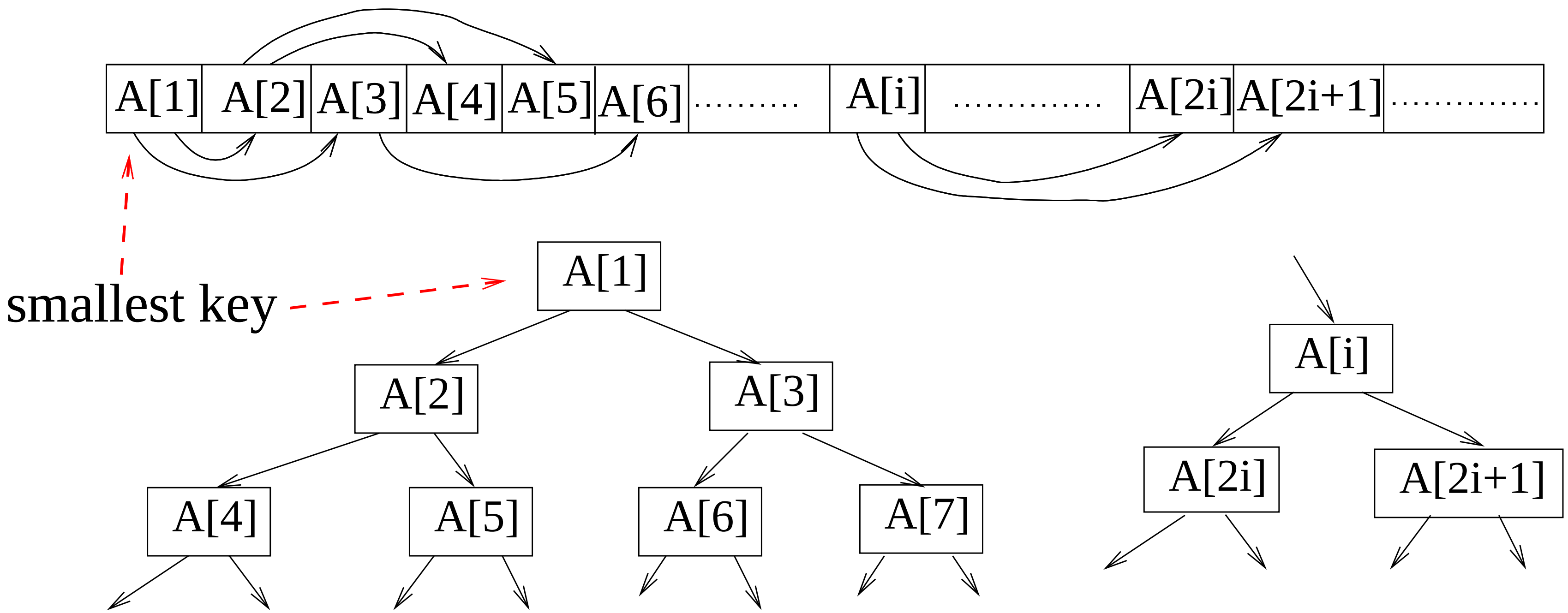
- If the input graph is **dense**, that is (here), if $m = \Omega(n^2/\log n)$, then the implementation of Dijkstra's algorithm **without Heap** (maintaining Q as an unordered list) gives the better (worst-case) running time – $O(n^2)$.
- For the other cases (when $m = O(n^2/\log n)$), the implementation of Dijkstra's algorithm **with Heap** gives the better (worst-case) running time.
- In most applications of Dijkstra's algorithm, graphs are not dense, so Dijkstra's algorithm is commonly assumed to use Heap.

Heap implementation of Priority Queue data structure

- **Heap**: An array $A[1..n]$ with a sequence of numbers (keys) and associated data (values), satisfying some specific partial order of the entries.
  This specific order is called the **heap property** (Min-Heap here):

  $$A[i] \leq A[2i] \quad \text{and} \quad A[i] \leq A[2i+1], \quad \text{for } i = 1, 2, \ldots$$

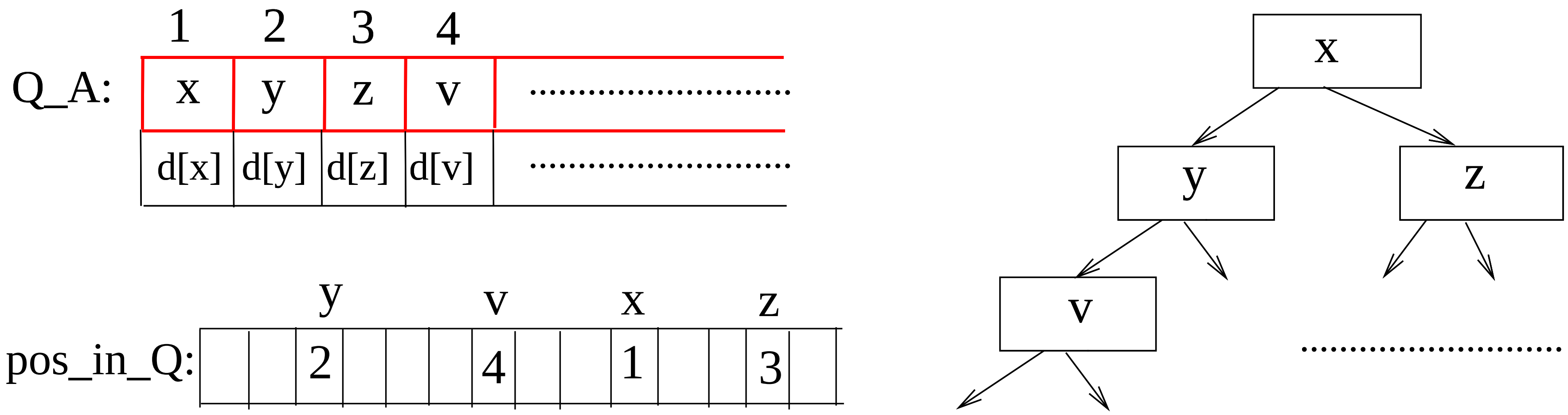
  Below, an arrow points from a smaller key to a larger key:



- The operations of extracting the minimum and inserting a new element take $O(\log n)$ time, where n is the current size of the Heap.

Heap in Dijkstra's algorithm

The entries in the Heap array are the pairs $(u, d[u])$, where u is a node in the graph and $d[u]$ is its current shortest path estimate. The number $d[u]$ is the key of this entry.



- We need a heap operation HEAP-DECREASE-KEY, which restores the heap property when the key of one entry is decreased.

RELAX(u, v) (in Dijkstra's algorithm, if Q is a Heap)

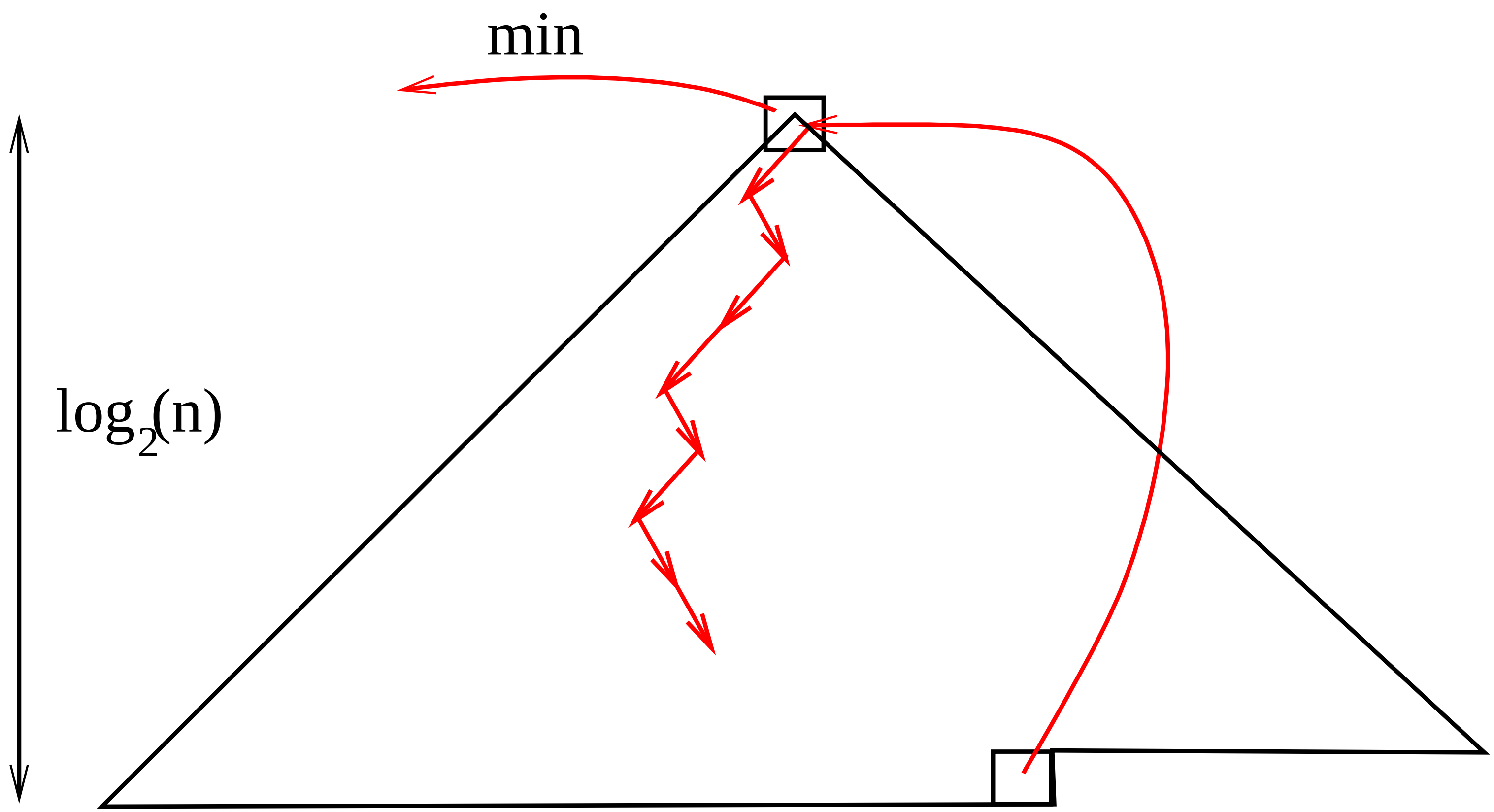
if $d[v] > d[u] + w(u, v)$ **then**

$d[v] \leftarrow d[u] + w(u, v)$; PARENT$[v] \leftarrow u$

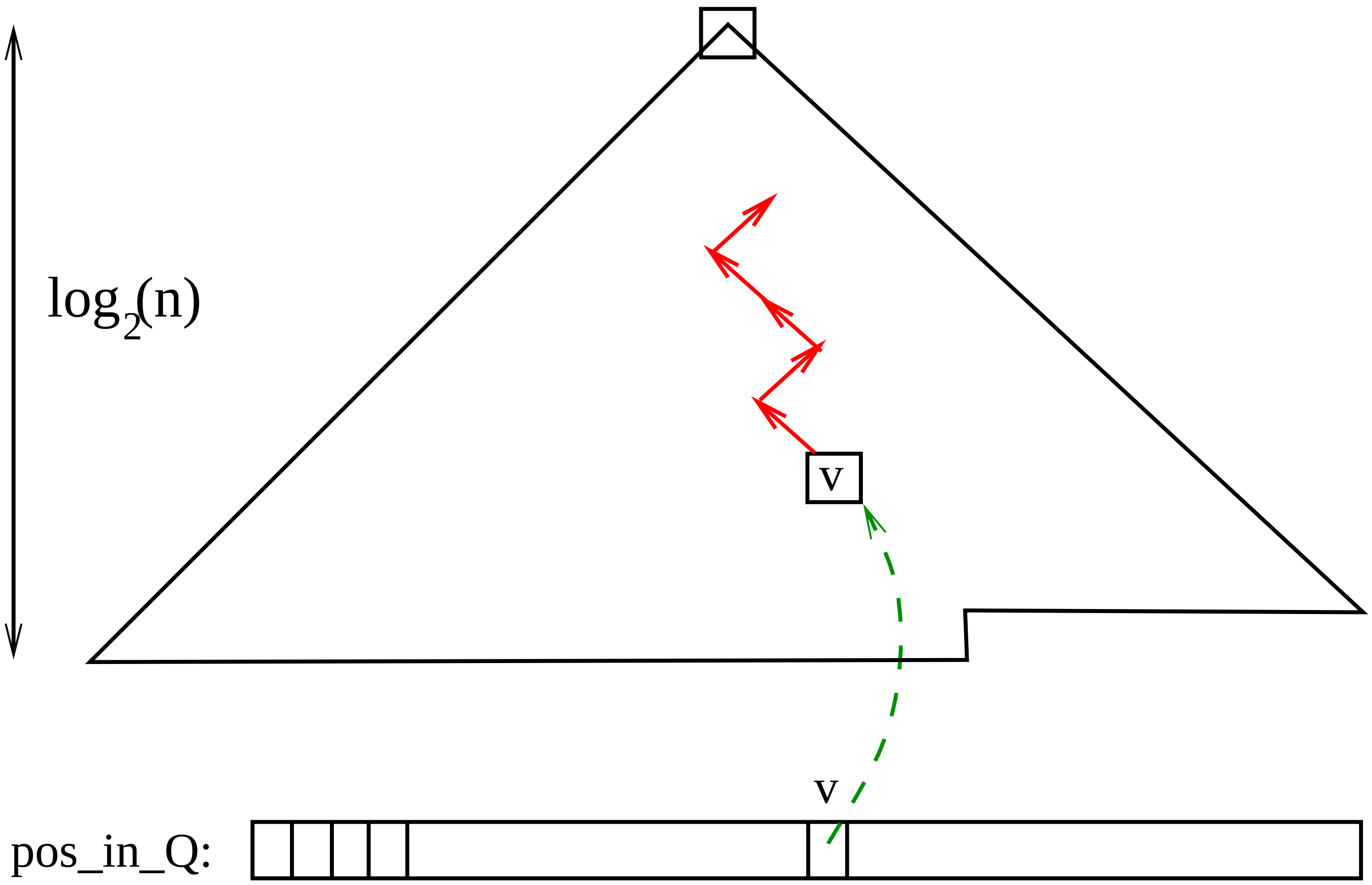
HEAP-DECREASE-KEY$(Q, v, d[v])$

Operation HEAP-DECREASE-KEY takes $O(\log n)$ time.

Heap Extract-Min operation

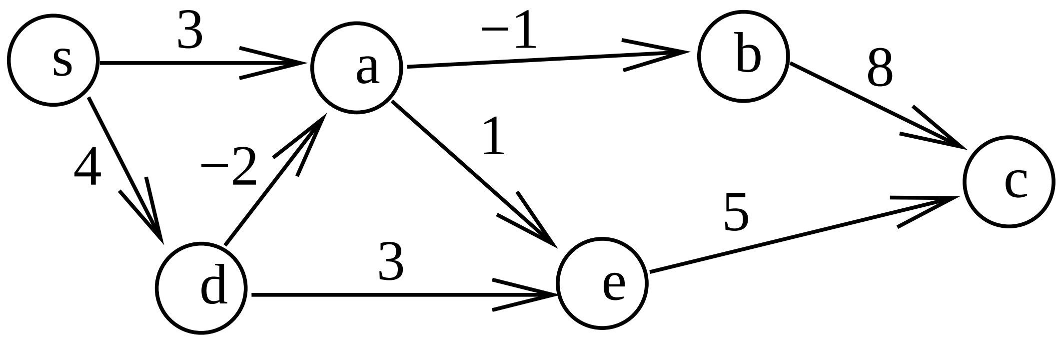


Heap-Decrease-Key operation



Input:

- $G = (V, E)$ – directed acyclic graph (DAG);
- w – weights of edges (may be negative);
- $s \in V$ – the source node.



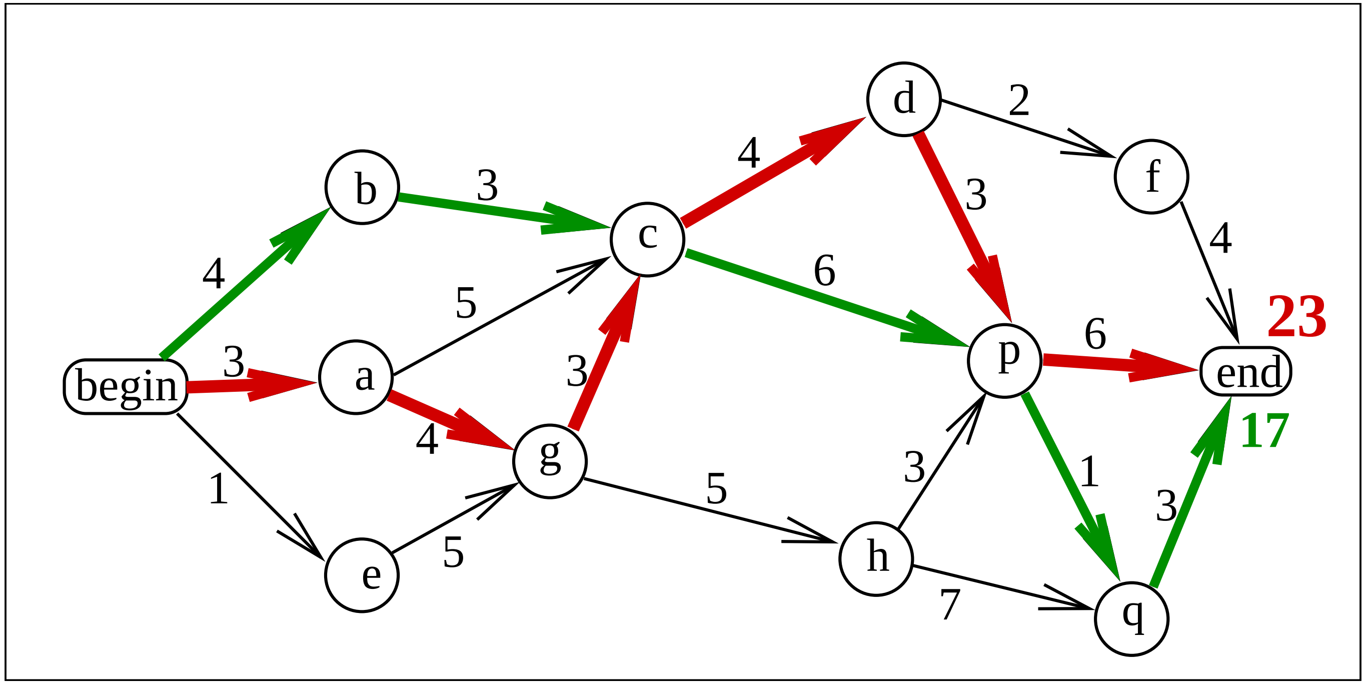
Output: the shortest-path weights from s to all other nodes and a shortest-paths tree from s.

There may be edges with negative weights,
but no problem with negative cycles, because
there are no cycles at all.

Bellman-Ford: $O(mn)$.

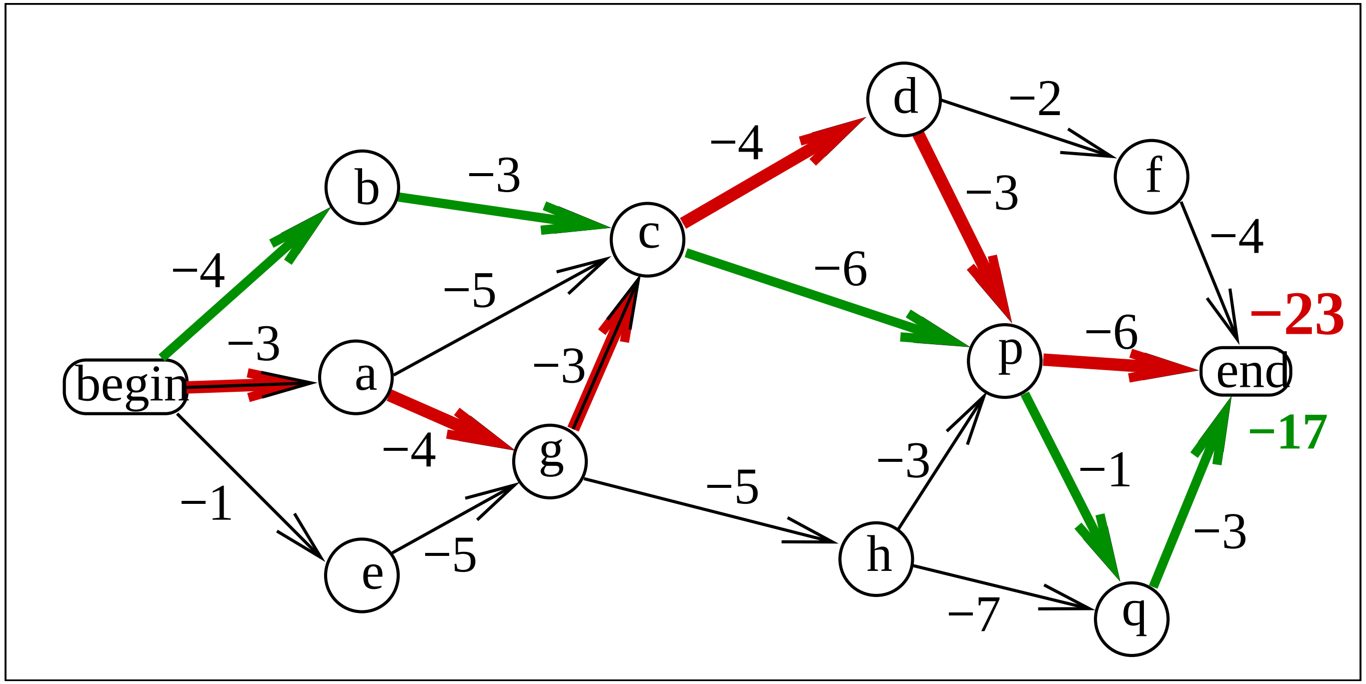
We show an algorithm which runs in $O(n + m)$ time.
(Linear, optimal time)

- Determine the critical (longest) paths in PERT chart analysis (Program Evaluation and Review Technique).
- Nodes: milestones of the project ('synchronisation points') . Edges: tasks of the project. The weight of an edge: the duration of this task.
  Find the longest path from node "begin" to node "end":
  this gives the fastest possible time for completing the whole project (that is, for completing all tasks of the project).

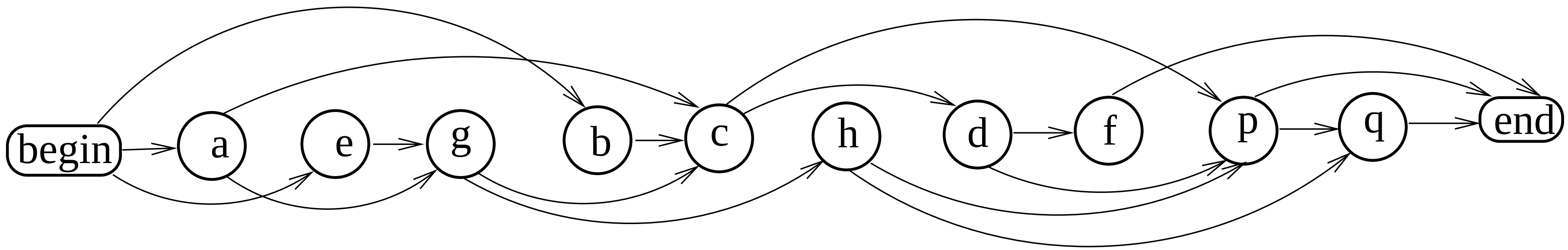


DAGs: Longest paths to shortest paths by negating all egde weights

- To compute longest paths from the start node, negate all edge weights and compute shortest paths from the start node.



- Consider the nodes in a topological order: all edges go in the same direction.



DAGSHORTESTPATHS(G, w, s)
1: topologically sort the nodes of G $\{$ put the nodes in a topological order $\}$
2: INITIALIZATION(G, s)
3: **for** each node u taken in the topological order computed in step 1 **do**
4:     **for** each node $v \in Adj[u]$ **do**
5:         RELAX(u, v, w)

- Running time: $O(n + m)$
  (topological sort takes $O(n + m)$ time).
- Correctness: show (by induction) that when node u is considered in line 3, then $d[u] = \delta(s, u)$.

Examples and Exercises – LGT

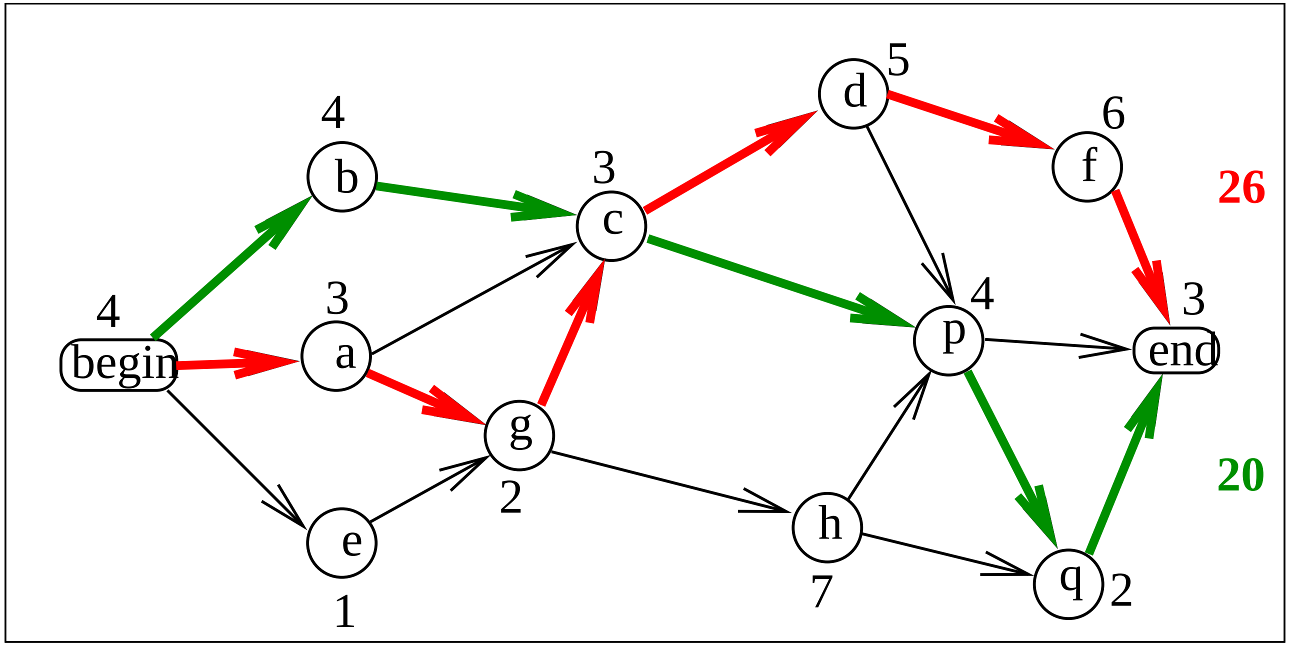
1. Example of the computation of Dijkstra’s algorithm – slides 4-5.

2. What would be the running time of Dijkstra’s algorithm, if the priority queue Q was maintained as an ordered list?

3. (Exercise 24.2-3 from [CLRS])
The PERT chart formulation given in this lecture is somewhat unnatural. More naturally, vertices would represent tasks and edges would represent order constraints; edge (u, v) would indicate that task u must be performed before task v. We would then assign weights (duration of the tasks) to vertices, not edges. Modify the DAGSHORTESTPATHS algorithm so that it finds, in linear time, a longest path in a directed acyclic graph with weighted vertices. Let $w(v)$ denote the weight of a vertex v.
Trace the computation of the algorithm on the graph below.

The green path has weight 20.

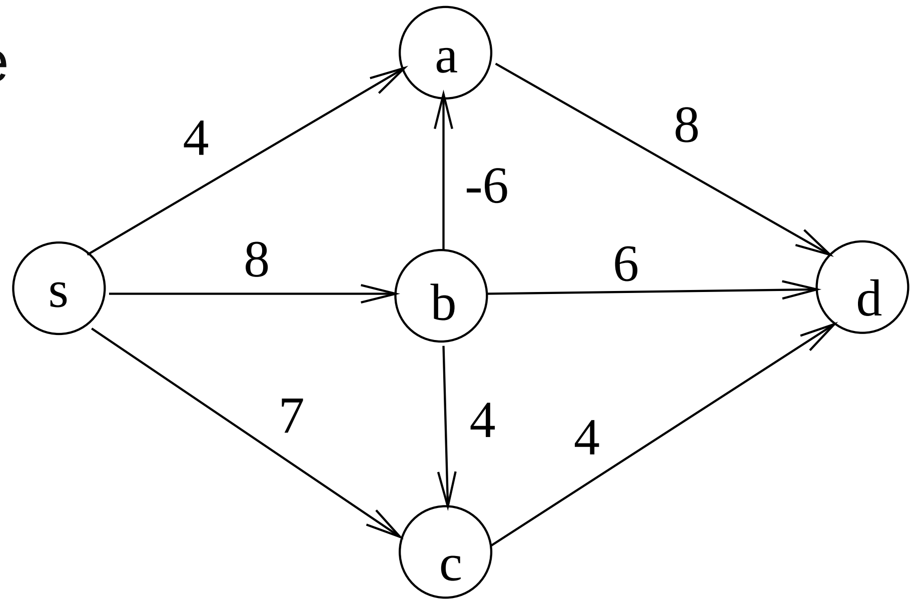
The red path, with weight 26, is the longest path.



Exercises – SGT

1. This exercise shows that Dijkstra's algorithm does not necessary compute shortest paths, if there are negative weights; even if there are no negative cycles.

   (a) Show the values $d[x]$ and the tree computed by Dijkstra's algorithm for the input graph:



   (b) Show the shortest-path weights and a shortest-paths tree in this graph.

   (c) How to modify Dijkstra's algorithm so that it runs also for graphs where edge weights may be negative?

   (d) What is the running time of the modified algorithm?

Exercises – SGT (cont.)

2. Design a linear-time ($O(n+m)$-time) algorithm which for a given directed acyclic graph (with n nodes and m edges) computes a topological order of the nodes.

   Do not use the Depth-First Search (DFS) algorithm.
   (There is a topological sort algorithm which is based on the DFS algorithm, but in this exercise you are asked to design an alternative algorithm.)

   Hint: How to identify the fist node for the topological order? How to identify subsequent nodes?

   For the running time, assume the adjacency-list representation of the graph.

   Specify all data structures which your algorithm needs to achieve the $O(n+m)$ running time.

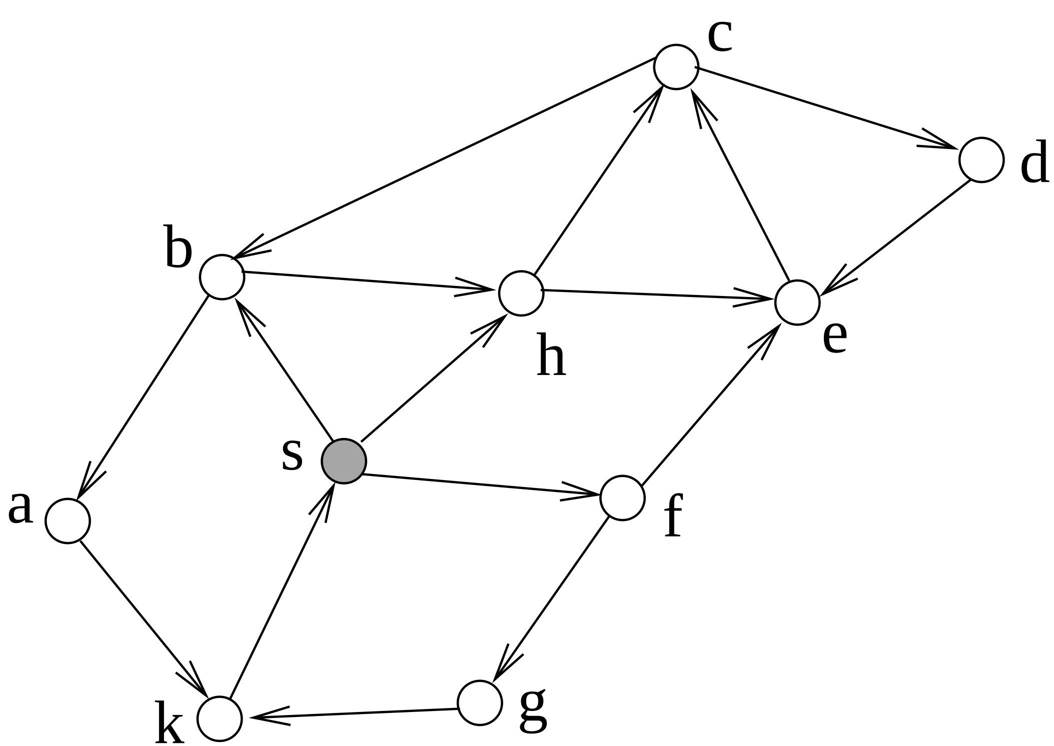
Exercises – SGT (cont.)

3. Revise the Breadth-First Search (BFS) algorithm.

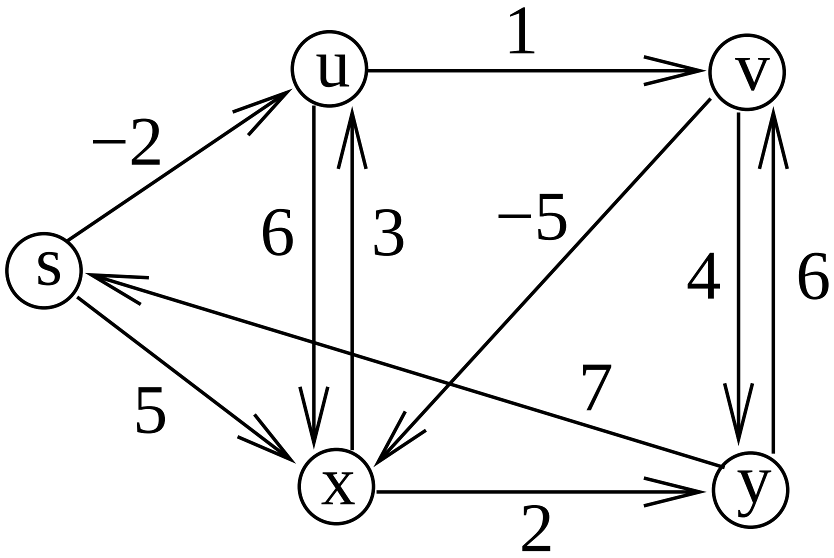
   Trace the execution of BFS on the graph shown below.

4. Revise the Depth-First Search (DFS) algorithm.

   Trace the execution of BFS on the graph shown below.



5. The graph below has a negative cycle reachable from the source. Trace the computation of the Bellman-Ford algorithm with FIFO queue on this graph until the PARENT edges form a cycle.



Assume that the edges outgoing from nodes are given in this order:
$(s, u), \quad (s, x)$;
$(u, x), \quad (u, v)$;
$(x, u), \quad (x, y)$;
$(v, x), \quad (v, y)$;
$(y, s), \quad (y, v)$.