Julius-Maximilians-
UNIVERSITÄT
WÜRZBURG

CAIDAS

WüNLP

ALGORITHMS IN AI & DATA SCIENCE 1 (AKIDS 1)

Graph Algorithms

Prof. Dr. Goran Glavaš

27.11.2023

Content

- Strongly Connected Components
- Single-Source Shortest Path

Graph: definition, types

Graph: formal definition

A **graph** G = (V, E) is a pair of sets, with V as a set of vertices, and E a set of edges between the vertices $E \subseteq \{ (u,v) \mid u, v \in V\}$. If the graph is **undirected**, the relation defined by an edges is symmetric, or $E \subseteq \{\{u,v\} \mid u, v \in V\}$, that is, edges are sets of two vertices rather than ordered pairs.

- **Directed (gerichteter) graph** – edges have directions: $(u, v) \neq (v, u)$

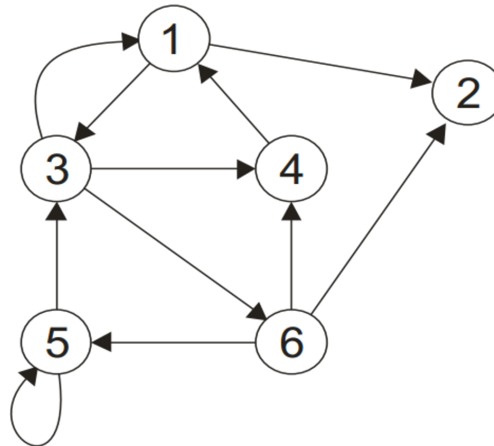


Image from ADS course of Andreas Hotho

Graphs: connectivity

- **Undirected graphs**
  - Vertices u and v **connected** is there exist a **path** (i.e., a sequence of edges) in G from u to v
  - Graph G is **connected** if any two vertices from V are connected
- **Directed graphs**
  - **Strongly connected**: if for every two nodes u, v both path from u to v and path from v to u exist
  - **Weakly connected**: if the corresponding undirected graph (make directed edges with undirected) is connected

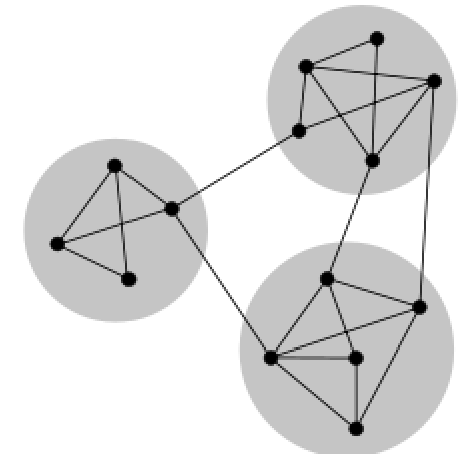


Image from Wikipedia

Strongly Connected Components

Strongly connected component

A **strongly connected component** of a directed graph G=(**V**, **E**) is a maximal set of vertices **C** ⊆ **V** such that for every pair of vertices u, v ∈ **C** such that there exists both a path from u to v and a path from v to u (i.e., u and v are reachable from each other)

- Directed graph can have **one or more SCCs**
- A node can be a part of **more than one SCC**

- Many algorithms for directed graphs

  (1) decompose the graph into SCCs

  (2) run separately on each SCC

  (3) combine solutions based on the structure of connections between SCCs

Strongly Connected Components: Recursive DFS

- Algorithm for identifying SCCs – the **Kosaraju's algorithm** – leverages DFS on G and its transpose G^T
- **(Recursive) DFS** variant for the whole graph
  - Nodes must **stay „visited"** once they have been visited, regardless from which source node we start
  - Three states for a vertex: **unvisited** (0), **visited** (1) and **finished** (2)
    - The **„finished" state** is not strictly necessary, but it facilitates the following of the algorithm visually
  - Global variable **„time"**
    - For each vertex v records the time steps of „visiting" (**vt**, when state change 0→1) and „finishing" (**ft**, when state change 1 → 2)

```
dfs(G)
  for each vertex u in G.V
    u.state = 0
  time = 0
  for each vertex u in G.V
    if u.state == 0
      dfs_visit(G, u)


dfs_visit(G, u)
  time = time + 1
  u.state = 1 # visited
  u.vt = time
  for each vertex v in G.Adj[u]
    if v.state == 0 # if v unvisited
      dfs_visit(G, v)
  u.state = 2 # finished
  time = time + 1
  u.ft = time
```

Recursive DFS

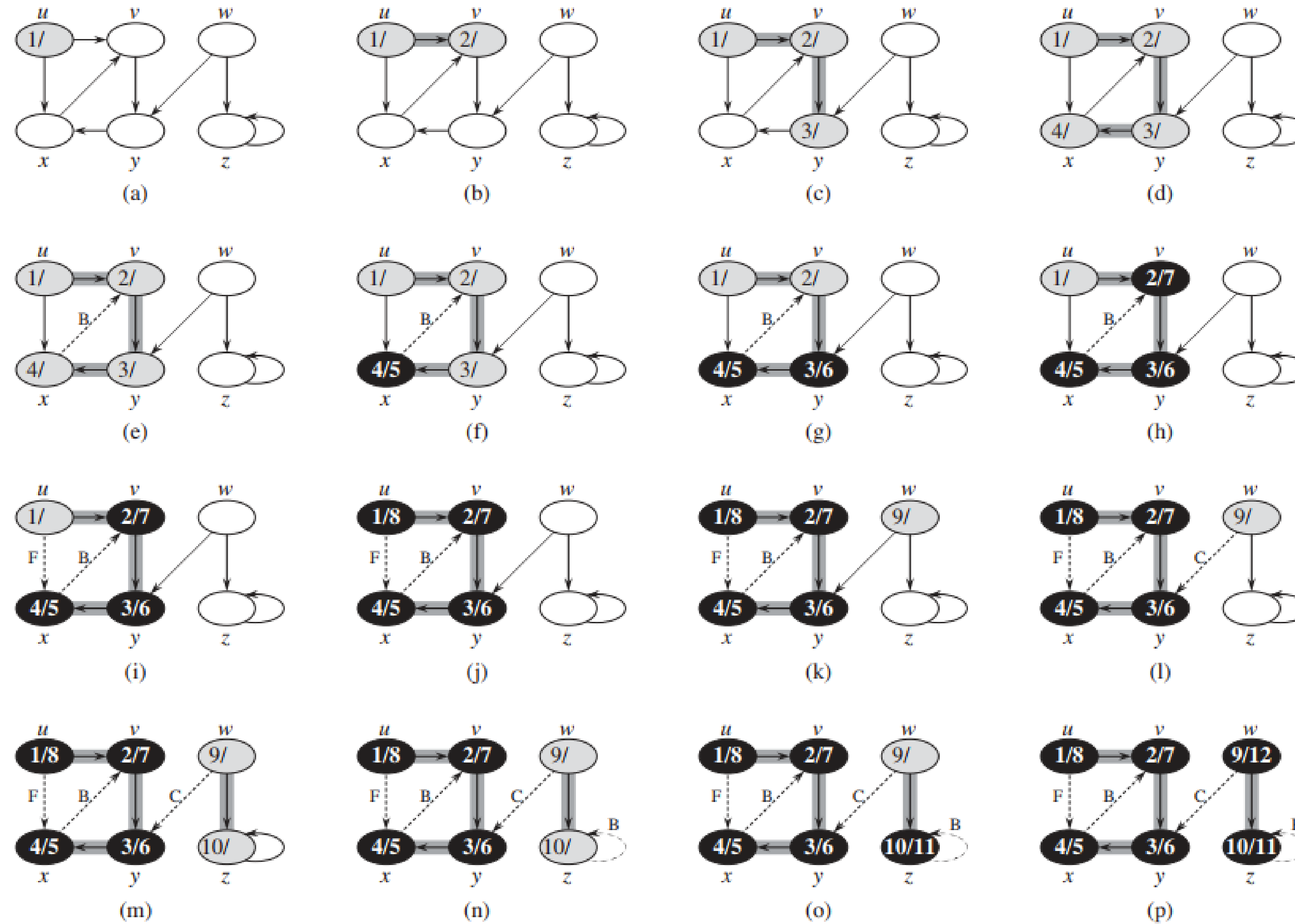
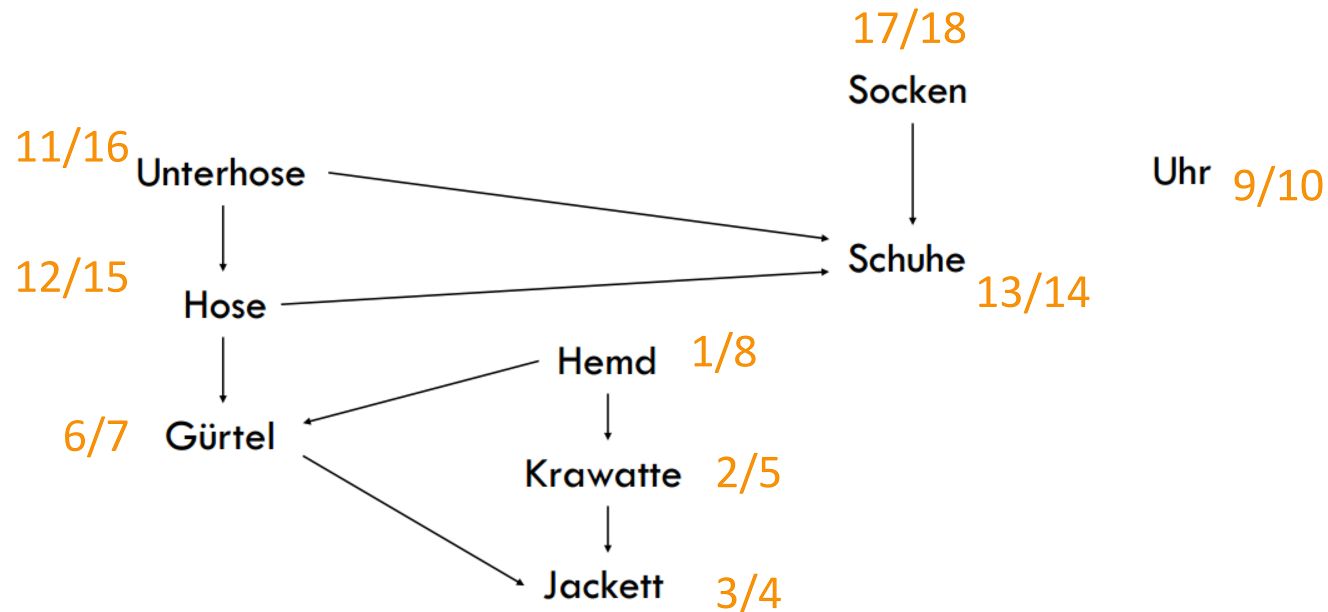


Image from Cormen et al.

Topological sort with Recursive DFS

- **Q:** How can we leverage the times u.ft for **topological sort**?
  - The exact u.vt and u.ft depend on the **order of processing** nodes without incoming edges (below, we assume: 1. Hemd, 2. Uhr, 3. Unterhose, 4. Socken)



Strongly connected components

- We will need the transpose of the graph G

Transpose of a directed graph

A **transpose** of a directed graph G=(**V**, **E**) is a graph G = (**V**, **E**^{T}) where **E**^{T} = {(v, u) : (u, v) ∈ **V**}. In other words, G^T is what you get if you invert the direction of all the edges in G.

- G and G^T have exactly the same strongly connected components
  - **Q:** Why?

Detecting SCCs: Kosaraju's algorithm

```
strongly_connected_components(G):
    dfs(G) # each vertex u gets u.ft
    GT = transpose(G)
    sccs = dfs_decrease(GT)
    return sccs
```

- **Q:** Why does this work, that is, produces the SCCs?

```
dfs(G)
  for each vertex u in G.V
    u.state = 0
  time = 0
  for each vertex u in G.V
    if u.state == 0
      dfs_visit(G, u)

dfs_decrease(G)
  for each vertex u in G.V
    u.state = 0
  time = 0
  sccs = []
  for u in G.V decreasing by u.ft
    if u.state == 0
      tree = dfs_visit(G, u)
      sccs.add(tree)
  return sccs
```

Kosaraju's SCC algorithm: analysis

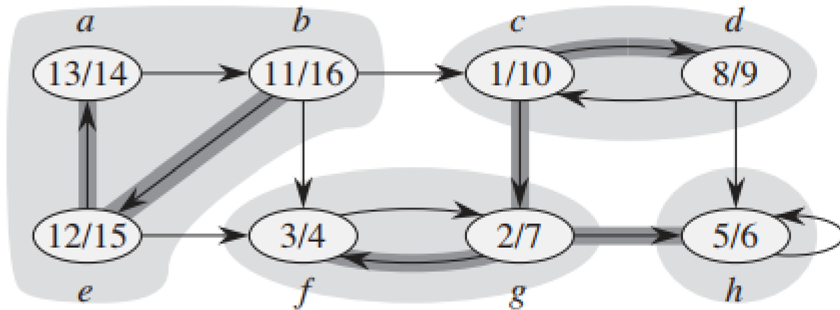
- To explain why the SCC algorithm works, we introduce the concept of a **component graph**

Component graph

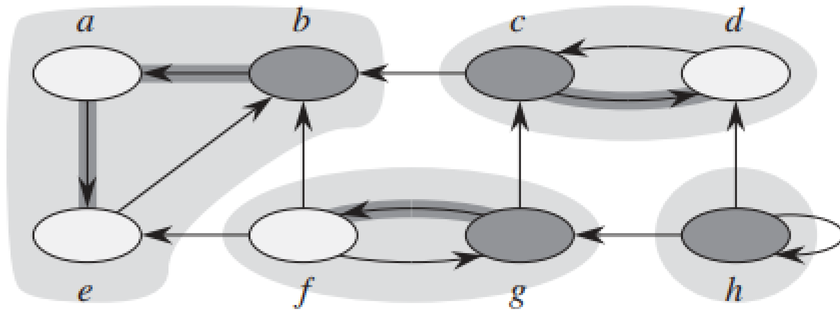
A **component graph** of a graph G is a „meta" graph $G^{SCC} = (V^{SCC}, E^{SCC})$ where each node represents one strongly connected component of G. Let G have K SCCs, $\{C_1, C_2, ..., C_K\}$. The vertex set V^{SCC} is $\{v_1, v_2, ..., v_k\}$ with each v_i representing one component C_i. An edge $(v_i, v_j) \in E^{SCC}$ if G contains an edge (x, y) where $x \in C_i$ and $y \in C_j$

- Component graph G^{SCC} of any directed graph G is a **directed acyclic graph (DAG)**. **Q:** Can you prove this?

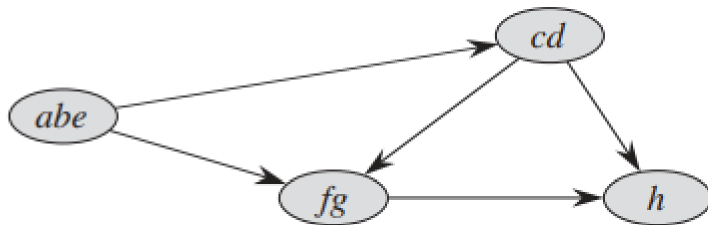
Kosaraju’s SCC algorithm: analysis



- Original directed graph G, after running **DFS** on it, with strongly connected components shaded
  - **Q:** How many root calls (i.e., non-recursive) to `dfs_visit` did we have?
  - **Q:** Which vertices were the „roots” of the DFS searches?



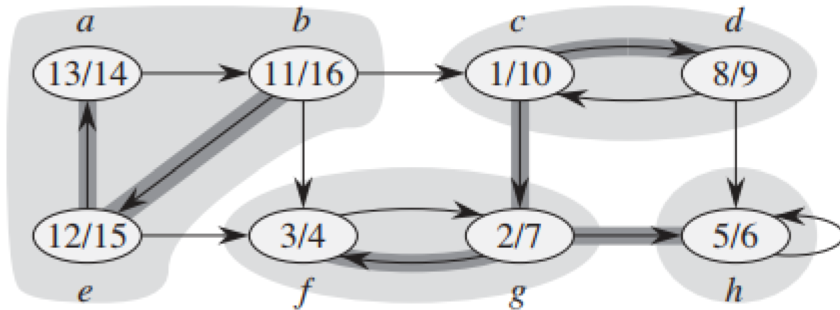
- Transposed graph G^T, dark nodes indicate the „roots” of DFS on G^T
  - In each component it is the node with largest u.ft

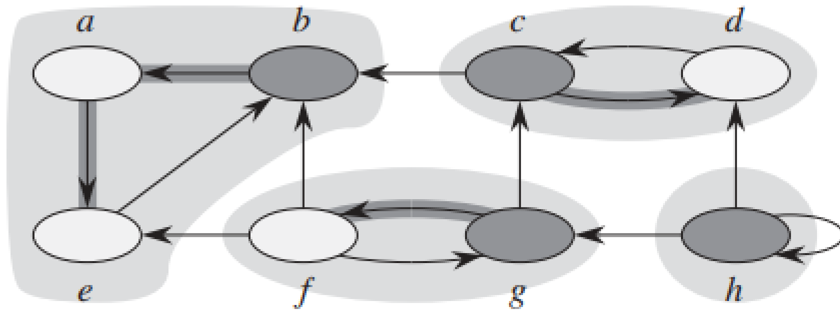


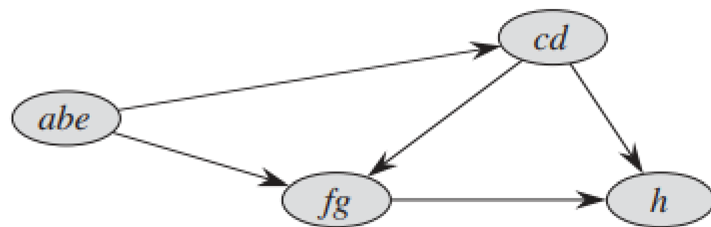
- Component graph G^{SCC} of G

Images from Cormen et al.

Kosaraju's SCC algorithm: analysis



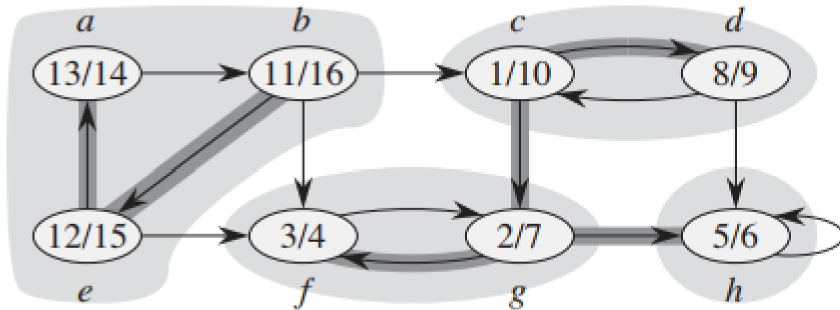


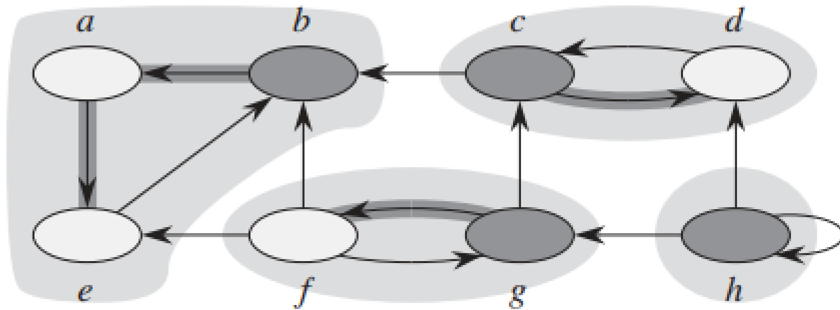


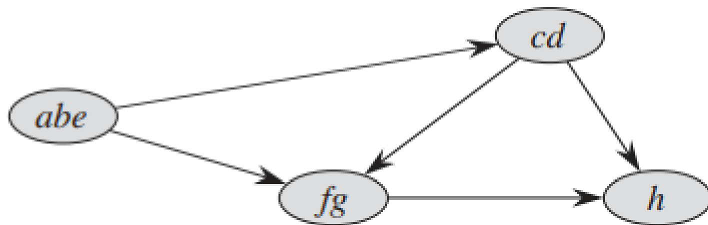
Images from Cormen et al.

- For a strongly connected component **C**, let *f*(**C**) be the maximal u.ft of its nodes
- If (u, v) in **E** such that u in $\mathbf{C}_i$ and v in $\mathbf{C}_j$, then $f(\mathbf{C}_i) > f(\mathbf{C}_j)$ (in G^T it's the opposite, $f(\mathbf{C}_i) < f(\mathbf{C}_j)$)
- DF-Trees from DFS on G^T **generate SCCs** (if carried out in decreasing order of u.ft)
- **Proof**: inductive
  - DFS in G^T on a vertex u (root of the DFS tree) that belongs to component $\mathbf{C}_i$ will collect all nodes reachable from u – will not miss any node from $\mathbf{C}_i$
  - **Q:** But can it collect a node from another component?
  - No! Because any edge exiting C_i in G^T can only be to a component for which $f(\mathbf{C}_i) < f(\mathbf{C}_j)$, i.e., the component that's already been identified

Kosaraju's SCC algorithm: analysis







Images from Cormen et al.

```
strongly_connected_components(G):
    dfs(G) # each vertex u gets u.ft
    G^T = transpose(G)
    sccs = dfs_decrease(G)
    return sccs
```

- **Q:** Runtime complexity of SCCs algorithm?
  - **First DFS** (on G): **O(V + E)**
  - **Graph transposition** – assuming adjacency list representation of G: **O(V + E)**
  - Second DFS (on G^T): **O(V + E)**

Content

- Strongly Connected Components
- Single-Source Shortest Path

Shortest paths on weighted graphs

Single-pair shortest-path problem

We are given a weighted directed graph G(**V**, **E**) with the weights w: **E** → $\mathbb{R}$. The path p passing through nodes <v_0, v_1, ..., v_k> then has the weight $w(p) = \sum_{i=1}^{k} w(v_{i-1}, v_i)$. The **shortest path problem** for a pair of vertices (u, v) amounts to finding the path from u to v (from all the possible paths that exist) with the lowest w(p), if such a path exists at all.

- There can be **multiple paths** from u to v with the same weight

Shortest paths problems

- Types
  - **Single-pair shortest-path**: find the shortest paths from u to v
  - **Single source shortest paths**: find the shortest paths between some specified **source vertex** u to all other vertices in the graph
  - **Single destination shortest-paths**: find the shortest paths from all other vertices in the graph to some specified destination vertex v
    - We can easily cast this to **single source shortest paths problem**. **Q:** How?
  - **All-pairs shortest-paths**: find sh. path from u to v for every pair of vertices u and v
    - **Q:** Just run **single source shortest paths** V **times** (once with each vertex as a source)?

Optimal substructure of shortest paths

- Shortest paths algorithms rely on the property that a shortest path between two vertices contains other shortest paths within it
  - **Dijkstra's algorithm** (single-source): uses this in a greedy manner
  - **Floyd-Warshall algorithm** (all pairs): uses this for dynamic programming
- Prove that subpaths of shortest paths are shortest paths
  - Path $p = \langle v0, v1, \ldots, vk \rangle$, subpath $p_{ij} = \langle v_i, v_{i+1} \ldots, v_j \rangle$
  - We can decompose p into p_{0i}, p_{ij} and p_{jk}
  - Then $w(p) = w(p_{0i}) + w(p_{ij}) + w(p_{jk})$
  - Assume a shorter path p'_{ij} between v_i and v_j, $w(p'_{ij}) < w(p'_{ij})$
    - Then there would be a shorter path p' between v_0 and v_k : $w(p') = w(p_{0i}) + w(p'_{ij}) + w(p_{jk})$

Shortest paths and negative weights

- The **Dijkstra algorithm** we'll examine assumes that there are no negative weights in the graph

- **Negative weights**
  - One or more **edges** in the graph have negative weights
  - **Q:** Are shortest path problems still well-defined with negative weights?
  - Depends on whether there are **negative weights cycles**
    - If yes, no longer well-defined problem
    - **Q:** Why?

- Even without negative weights, a shortest „walk" **never has cycles**
  - **Q:** why?

Single-source shortest paths: Bellman-Ford

- **Bellman-Ford** algorithm: general directed graph, with negative edges
- Two helper functions
  - `Initialize` (s gets distance 0, other vertices inf)
  - `relax`: changes the distance if better is found through some vertex

```
initialize(G, s)
   for each v in G.V
     v.dist = inf
     v.prev = null
   s.dist = 0

relax(u, v, w)
  if v.dist > u.dist + w(u, v)
    v.dist = u.dist + w(u, v)
    v.prec = u
```

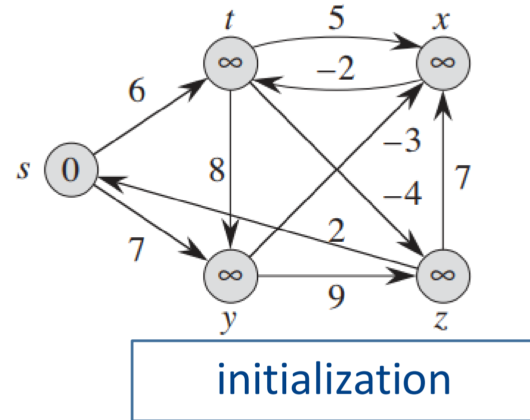
- „Relax" all edges

  |V|-1 times

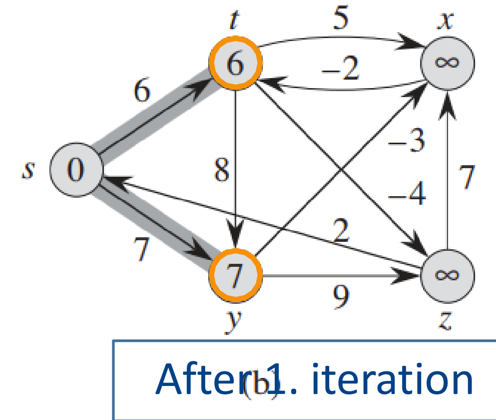
```
bellman-ford(G, w, s)
  initialize(G)
  for i in 1 to |G.V| - 1
    for each edge (u, v) in G.E
      relax(u, v, w)

  for each edge (u, v) in G.E
    if v.dist > u.dist + w(u, v) # negative weight cycle
       return False

  return True
```

Bellman-Ford algorithm

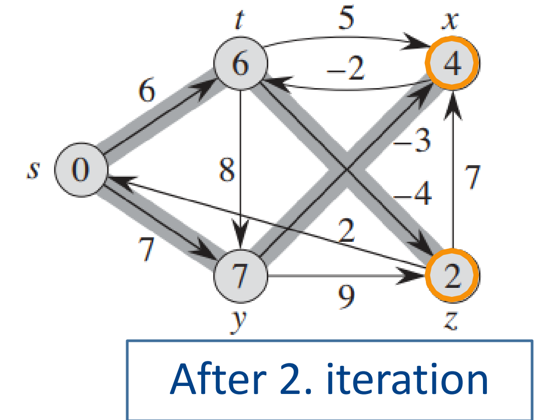
```
bellman-ford(G, w, s)
  initialize(G)
  for i in 1 to |G.V| - 1
    for each edge (u, v) in G.E
      relax(u, v, w)
  for each edge (u, v) in G.E
    if v.dist > u.dist + w(u, v)
      return False
  return True
```



initialization

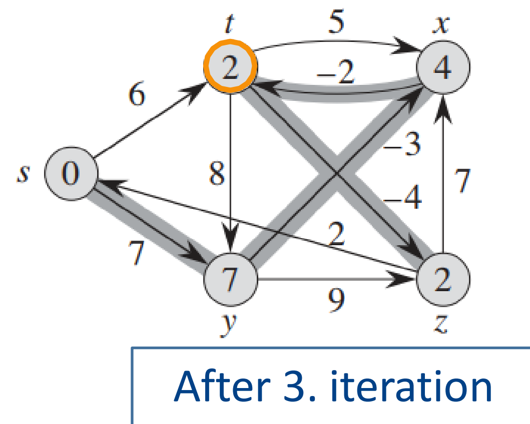


After 1. iteration

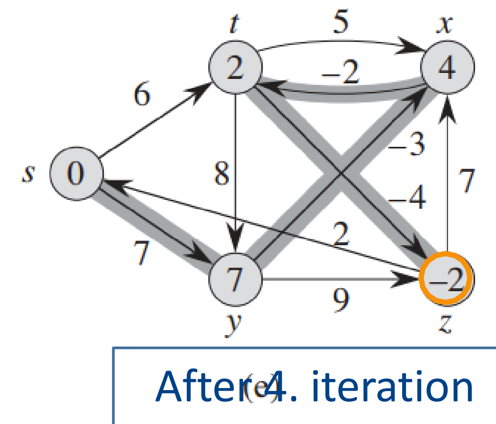


After 2. iteration

- **Q:** Why does the for loop run |G.V| - 1 times?
- **Q:** Runtime of Bellman-Ford?
- **Q:** What if we knew we had no negative weights?



After 3. iteration



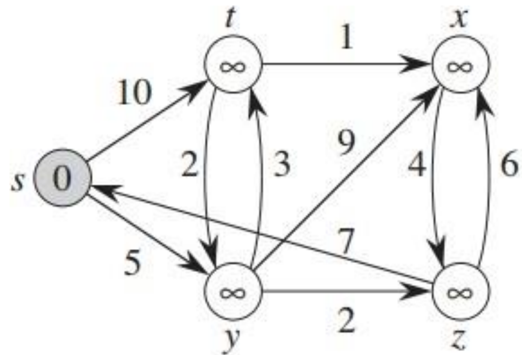
After 4. iteration

Single-source shortest paths: Dijkstra

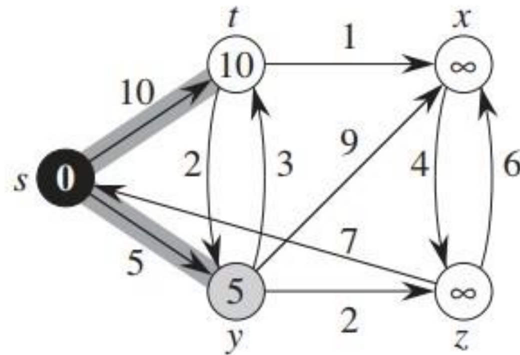
- **Dijkstra** algorithm: weighted directed graph, with **non-negative weights**
- Maintains a set of **S** vertices whose final shortest-path distance from source s has been determined
  - Since there are no negative edges, onde determined, **it cannot be changed**
- From the remaining edges **V-S**, in each iteration, we select a vertex **greedily**
  - One that has the smallest estimate of the distance from s

```
initialize(G, s)
   for each v in G.V
     v.dist = inf
     v.prev = null
   s.dist = 0

relax(u, v, w)
  if v.dist > u.dist + w(u, v)
    v.dist = u.dist + w(u, v)
    v.prec = u

dijkstra(G, w, s)
  initialize(G)
  S = [] # empty set
  Q = G.V # set of nodes to be „finished"

  while len(Q) > 0 # while Q not empty
    u = extract_min(Q) # node with smallest u.dist
    S = S U {u}
    for each v in G.Adj[u]
      relax(u, v, w)
```

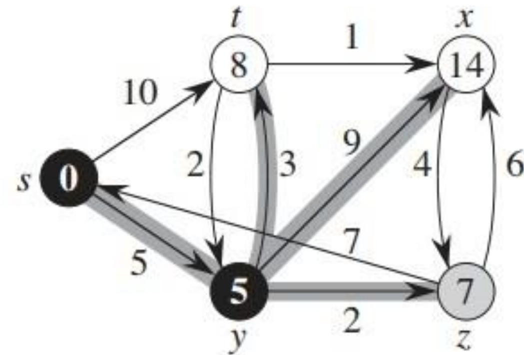
Dijkstra algorithm



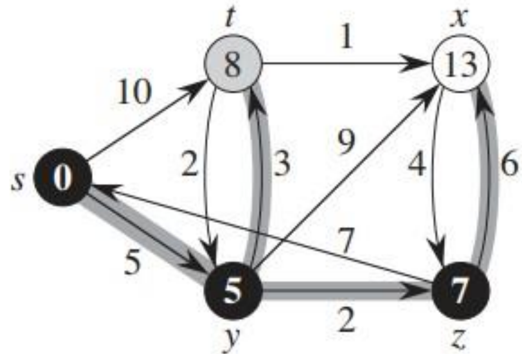
initialization



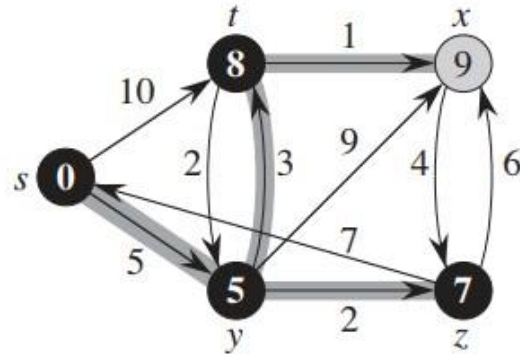
After 1. iteration



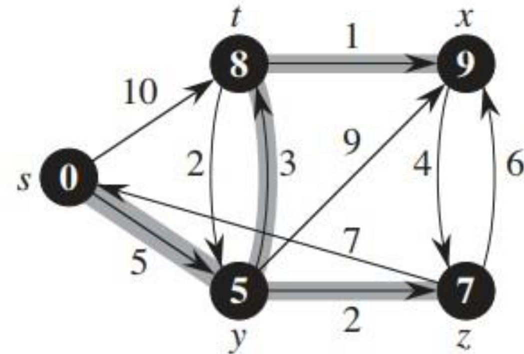
After 2. iteration



After 3. iteration



After 4. iteration



After 5. iteration

Dijkstra algorithm: runtime analysis

- **Q:** runtime of Dijkstra?
- How fast can we extract the min value from **Q**?
- **Q:** Data structure that extracts the minimum of a dynamic set the fastest?

```
initialize(G, s)
   for each v in G.V
     v.dist = inf
     v.prev = null
   s.dist = 0

relax(u, v, w)
  if v.dist > u.dist + w(u, v)
    v.dist = u.dist + w(u, v)
    v.prec = u

dijkstra(G, w, s)
  initialize(G)
  S = [] # empty set
  Q = G.V # set of nodes to be „finished”

  while len(Q) > 0 # while Q not empty
    u = extract_min(Q) # node with smallest u.dist
    S = S U {u}
    for each v in G.Adj[u]
      relax(u, v, w)
```

Recap: Priority Queue

- We've used heap as a data structure that supports heapsort
  - In most practical sorting applications, **quicksort** faster than **heapsort**
- But heap is useful for more than just sorting, as an actual implementation of an ADS called **priority queue**

Priority queuing

A set of elements S, each $s \in S$ has a corresponding **priority number (key)** assigned to it.
Elements with higher priority should be processed before elements of lower priority.
Elements with the same priority should be processed in the order of insertion (queue).

- **Min**-Priority queue has:
  - `Insert`, `Minimum`, `Extract-Min`, `Decrease-Prio`

Dijkstra algorithm: runtime analysis

- **Q:** runtime of Dijkstra?
- Data structure: **min-heap**

- What operations on min-heap do we need?
  - build_heap
  - extract_min (minimum)
  - decrease_prio

```
initialize(G, s)
   for each v in G.V
     v.dist = inf
     v.prev = null
   s.dist = 0

relax(u, v, w)
  if v.dist > u.dist + w(u, v)
    v.dist = u.dist + w(u, v)
    v.prec = u

dijkstra(G, w, s)
  initialize(G)
  S = [] # empty set
  Q = G.V # set of nodes to be „finished"

  while len(Q) > 0 # while Q not empty
    u = extract_min(Q) # node with smallest u.dist
    S = S U {u}
    for each v in G.Adj[u]
      relax(u, v, w)
```

Recap: Build heap

- **How many times** and **for which indices** (nodes) of the array do we need to call heapify in order to transform an array into a **heap**?
- `heapify` propagates the „smaller values down”
  - We actually want to propagate the „larger values up”
- To convert an array into a heap, we will call heapify in a **bottom-up manner**, **for each non-leaf node**
  - binary tree has n elements: how many non-leaf nodes (nln) does it have?

```
build_heap(A)
   A.HeapSize = A.length
   nln = n//2

   for i in nln - 1 downto 0
      heapify(A, i)
```

Recap: Build heap – runtime

- Let H be the height of the tree, H = $\lfloor \log_2 n \rfloor$
  - Let h be the **height** of a node/index
  - Let d be the depth of a node/index, d = H – h
- **T(n)** $= \sum_{h=0}^{H} 2^d * O(h)$

$$= \sum_{h=0}^{H} 2^{(H-h)} * O(h)$$
$$= \sum_{h=0}^{H} 2^H/2^h * O(h)$$
$$\leq \sum_{h=0}^{H} n/2^h * O(h)$$
$$= O\left(n \sum_{h=0}^{H} \frac{O(h)}{2^h}\right)$$
$$= \boldsymbol{O(n)}$$

H = $\lfloor \log_2 n \rfloor$ means that

$2^H \leq n < 2^{H+1}$

O(h) means T(h) = c*h

When H is large (approx. infinity)

$$\sum_{h=0}^{\infty} \frac{c*h}{2^h} = c * \mathbf{2}$$

Recap: min-priority queue

```
Extract-Min(A)
  if A.HeapSize < 1
    error „underflow”
  min = A[0]
  A[0] = A[A.HeapSize - 1]
  A.HeapSize = A.HeapSize - 1
  heapify(A, 0)
  return min
```

O(log n)

```
Decrease-Prio(A, i, key)
  if key > A[i]
    error „new key larger than current”
  A[i].key = key

  # restore heap property
  while i > 0 and A[i].key < A[parent(i)].key
    exchange(A[i], A[parent(i)])
    i = parent(i)
```

O(log n)

Dijkstra algorithm: runtime analysis

- **Q:** runtime of Dijkstra?
- Data structure: **min-heap**

- What operations on min-heap do we need?
  - build_heap: **O(V)**
  - extract_min (minimum): **O(V log V)**
  - decrease_prio: O(**E log V**)

- Total: **O((V+E) log V)**
  - If the graph is very dense, so that **E** approaches V^2, then $O(V^2 \log V)$

```
initialize(G, s)
    for each v in G.V
      v.dist = inf
      v.prev = null
    s.dist = 0

relax(u, v, w)
   if v.dist > u.dist + w(u, v)
     v.dist = u.dist + w(u, v)
     v.prec = u

dijkstra(G, w, s)
   initialize(G)
   S = [] # empty set
   Q = G.V # set of nodes to be „finished"

   while len(Q) > 0 # while Q not empty
     u = extract_min(Q) # node with smallest u.dist
     S = S U {u}
     for each v in G.Adj[u] # total E times
       relax(u, v, w)
```

Dijkstra algorithm: runtime analysis

- **Q:** runtime of Dijkstra?
- Data structure: **array** (we know the index of each vertex in the array)
- What operations do we need?
  - „build_heap”: **O(V)** (**V** times **O(1)**)
  - extract_min (minimum): **O(V^2)** (**V** times **O(V)**)
  - decrease_prio: **O(E)** (**E** times **O(1)**)
- Total: **O(V^2+E) = O(V^2)**
  - **Faster** than min-heap if the graph is **dense**

```
initialize(G, s)
    for each v in G.V
      v.dist = inf
      v.prev = null
    s.dist = 0

relax(u, v, w)
   if v.dist > u.dist + w(u, v)
     v.dist = u.dist + w(u, v)
     v.prec = u


dijkstra(G, w, s)
   initialize(G)
   S = [] # empty set
   Q = G.V # set of nodes to be „finished”

   while len(Q) > 0 # while Q not empty
     u = extract_min(Q) # node with smallest u.dist
     S = S U {u}
     for each v in G.Adj[u] # total E times
       relax(u, v, w)
```

Questions?

Awọn ibeere?
ਸਵਾਲ?
Küsimusi?
Turite klausimų?
Pitanja?
Sorusu olan?
Dúvidas?
質問は？
有问题吗？
Fragen?
¿Preguntas?
Frågor?
Questions?
Pytania?
Domande?
Питання?
Vragen?
Ερωτήσεις;
Porandukuéra?
أسئلة؟