Algorithmics

Lecture 6

Dr. Oana Andrei

School of Computing Science
University of Glasgow

oana.andrei@glasgow.ac.uk

Section 3 – Graphs and graph algorithms

Graph basics

- definitions: directed, undirected, connected, bipartite, …

Graph representations

- adjacency matrix/lists and implementation

Graph search and traversal algorithms

- depth/breadth first search

Topological ordering

Weighted graphs

- shortest path (Dijkstra's algorithm)
- minimum spanning tree (Prim–Jarnik and Dijkstra's refinement)

Directed Acyclic Graphs –Topological ordering

A Directed Acyclic Graph (DAG) is a directed graph with no cycles

A topological order on a DAG is a labelling of the vertices `1,…,n` such that `(u,v)∈E` implies `label(u)<label(v)`

- many applications, e.g. scheduling, PERT networks, deadlock detection

Scheduling and PERT networks

- PERT – program/project evaluation and review technique
- can model a project with a DAG
  - vertices are the tasks/activities and edges indicate dependencies
  - can also add timing information to edges
- if we have a topological order can find longest path (see tutorial 6) or longest weighted path
- can then determine which activities are "critical" (i.e., on longest paths)

Directed Acyclic Graphs –Topological ordering

A directed graph D has a topological order if and only if it is a DAG

- obviously impossible if D has a cycle (try to label the vertices in a cycle)

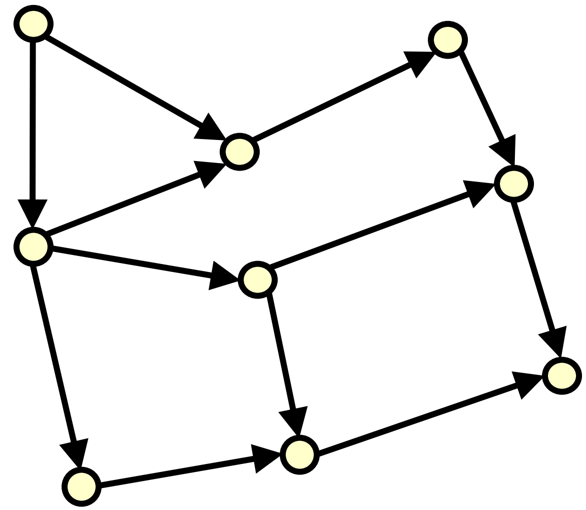
A source is a vertex of in-degree 0 and a sink has out-degree 0

Basic fact: a DAG has at least one source and at least one sink

- forms the basis of a topological ordering algorithm
- if there is no source or sink can build a cycle, and therefore not acyclic
  - if no source or sink can always keep adding vertices to the start or end of a path respectively as any vertex is neither a source nor a sink
  - eventually must add the same vertex twice to the path as there are only finitely many vertices, and therefore create a cycle

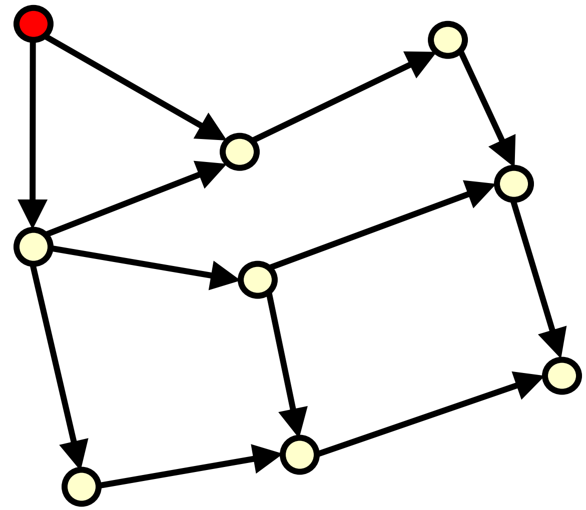
Directed Acyclic Graphs – Example

Directed acyclic graph **D**



Directed Acyclic Graphs – Example

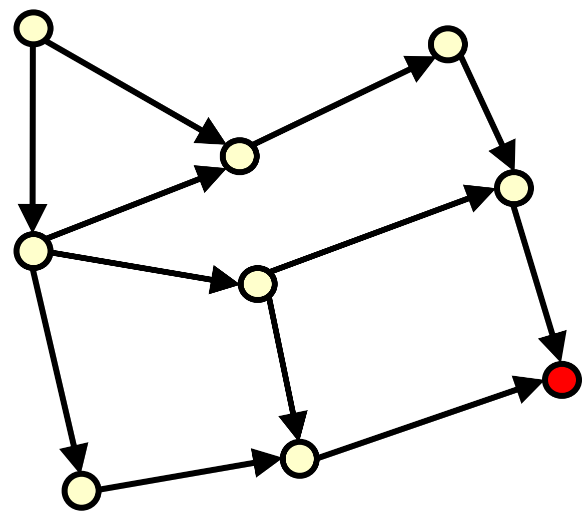
Directed acyclic graph **D**



Source vertex (in-degree equals 0)

Directed Acyclic Graphs – Example

Directed acyclic graph **D**

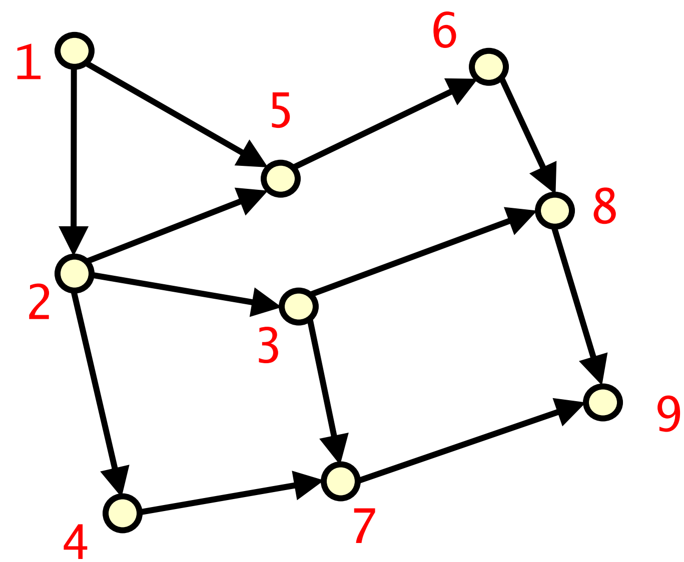


Sink vertex (out-degree equals 0)

Directed Acyclic Graphs – Example

Directed acyclic graph **D**

Topological ordering of **D**



A **topological order** on a DAG is a labelling of the vertices **1,…,n** such that **(u,v)∈E** implies **label(u)<label(v)**

Topological ordering algorithm

Add two integer attributes to every vertex in the graph

`label`

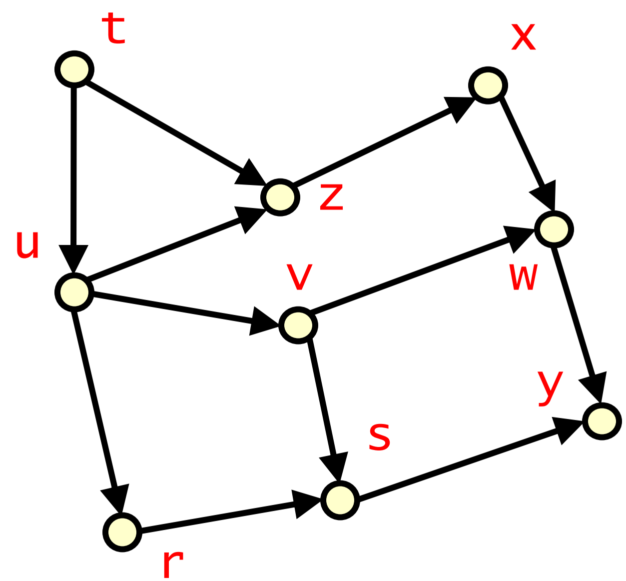
- the label in the topological order

`count`

- initially equals the number of incoming edges (in-degree) of the vertex
- updated as the algorithm labels vertices
- always equals the number of incoming edges from vertices not labelled
  - require the label of this vertex is greater than that of all incoming vertices
  - therefore if all vertices that have incoming edges have been labelled we can just label this vertex with a greater value
- when attribute becomes zero add vertex to a queue to be labelled
  - any source vertex can be added to the queue immediately

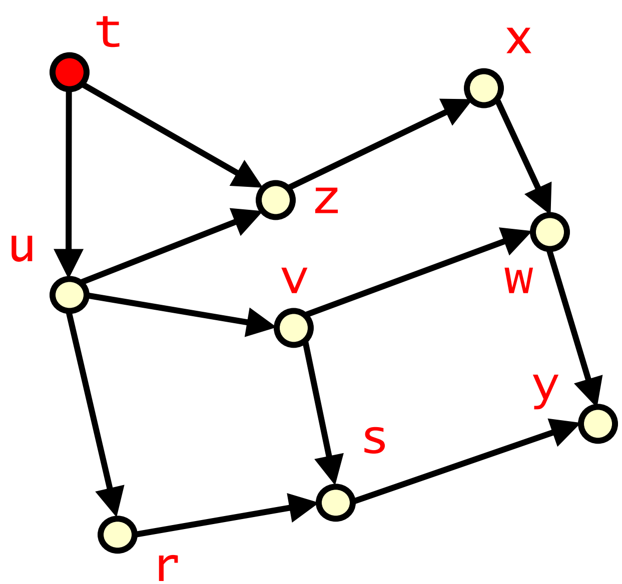
Topological ordering – Example

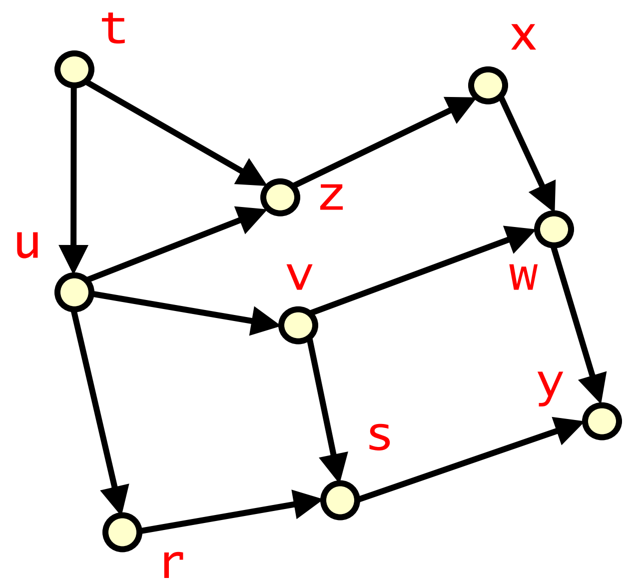
Directed acyclic graph D



Topological ordering – Example

Directed acyclic graph D



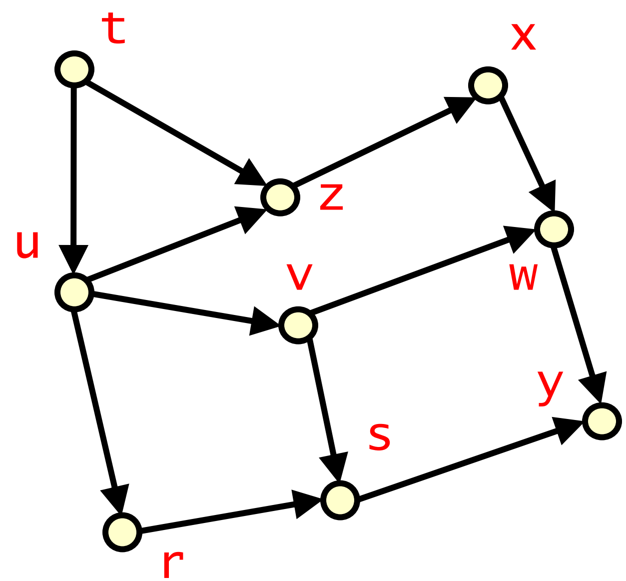


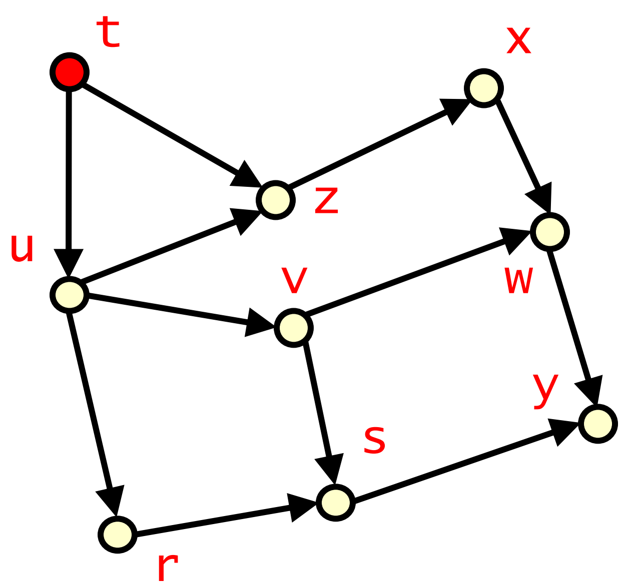
- labelled vertices
- queued vertices (count equals 0)
- vertices with count greater than 0

Topological ordering – Example

Directed acyclic graph **D**

source queue: ⟨**t**⟩





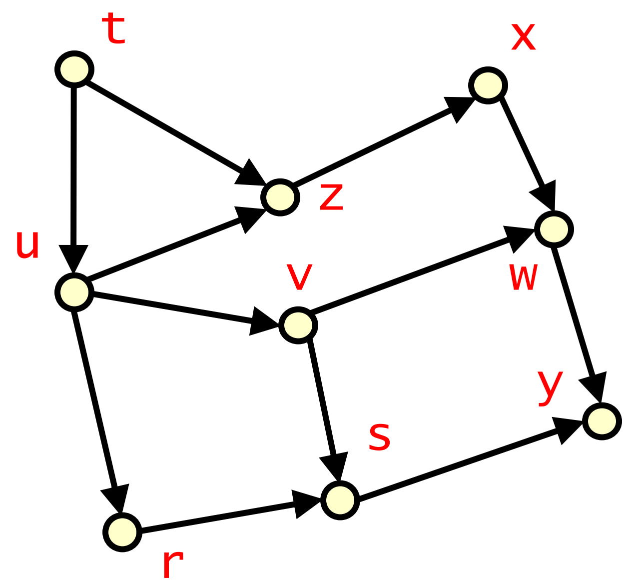
t is the only source vertex (only vertex with zero incoming edges)

add t to the source queue

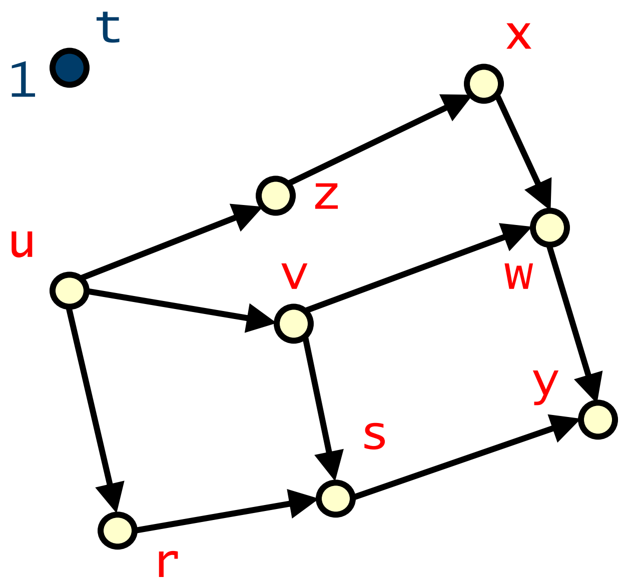
- labelled vertices
- queued vertices (count equals 0)
- vertices with count greater than 0

Topological ordering – Example

Directed acyclic graph D



source queue: ⟨⟩

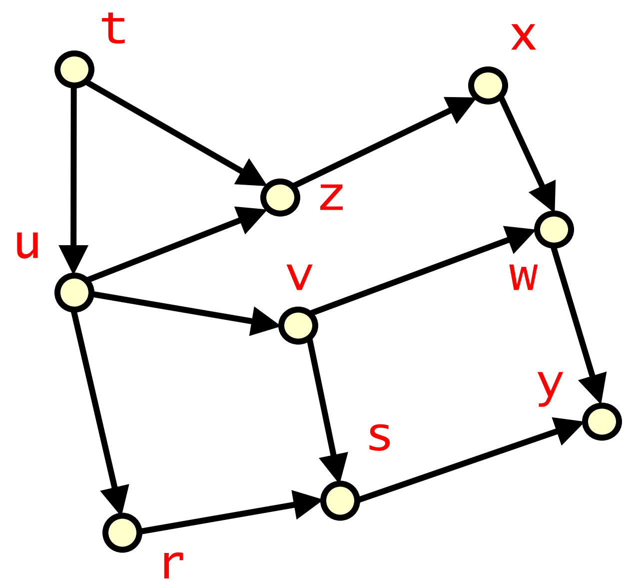


label and remove t from the graph and source queue and decrement the count for adjacent vertices

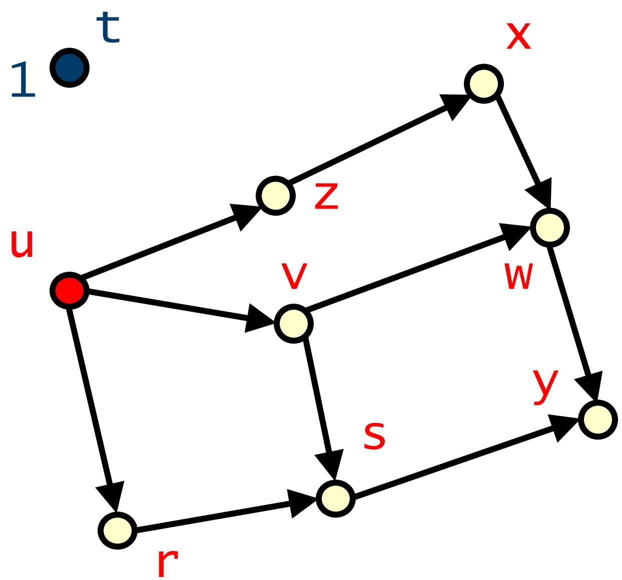
- labelled vertices
- queued vertices (count equals 0)
- vertices with count greater than 0

Topological ordering – Example

Directed acyclic graph D



source queue: ⟨u⟩



u now has no incoming edges

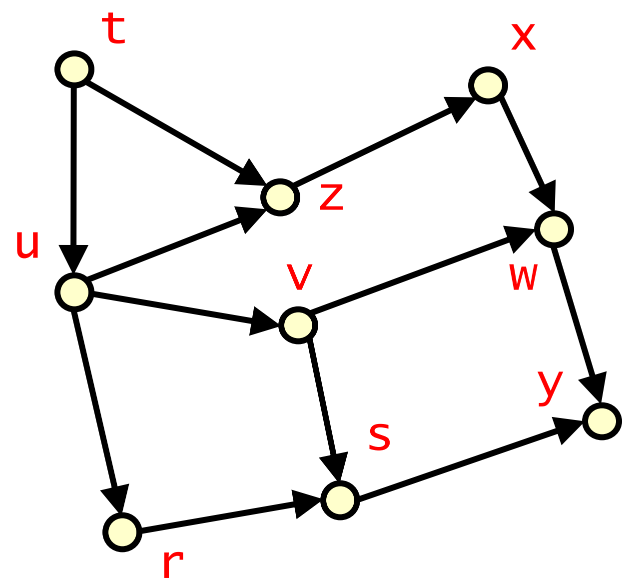
add u to the source queue

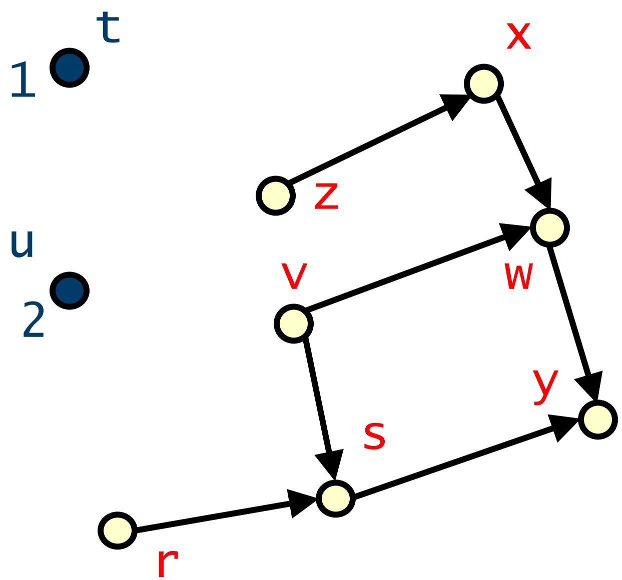
- labelled vertices
- queued vertices (count equals 0)
- vertices with count greater than 0

Topological ordering – Example

Directed acyclic graph D

source queue: ⟨⟩





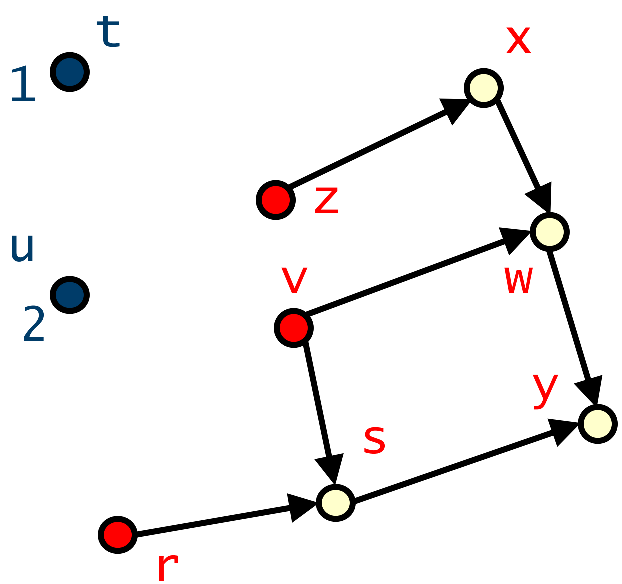
label and remove u from the graph and source queue

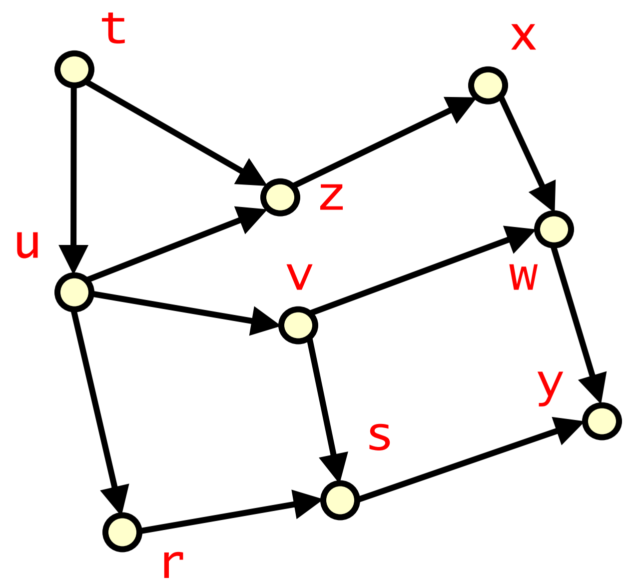
- labelled vertices
- queued vertices (count equals 0)
- vertices with count greater than 0

Topological ordering – Example

Directed acyclic graph D

source queue: ⟨**v**, **r**, **z**⟩





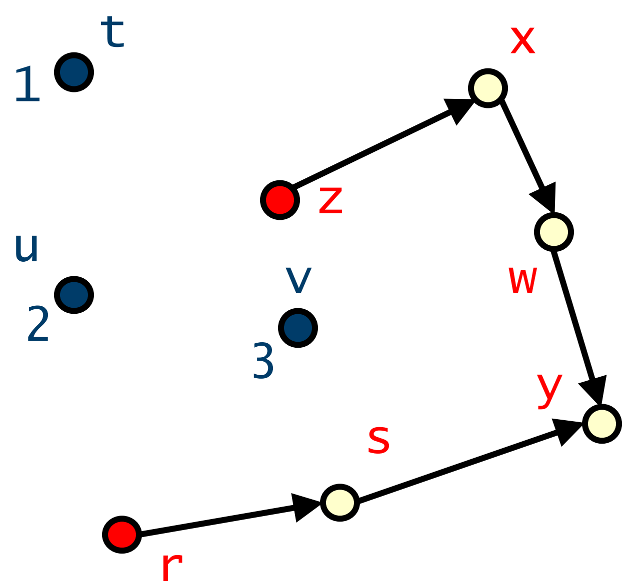
v, r and z become queued vertices (no incoming edges)

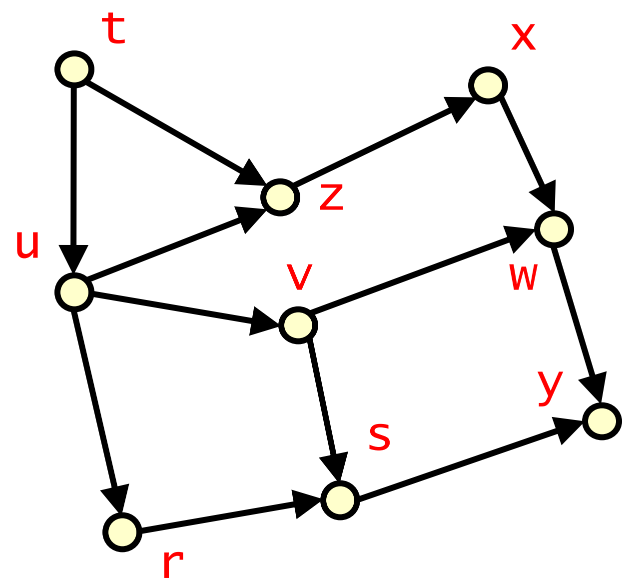
- labelled vertices
- queued vertices (count equals 0)
- vertices with count greater than 0

Topological ordering – Example

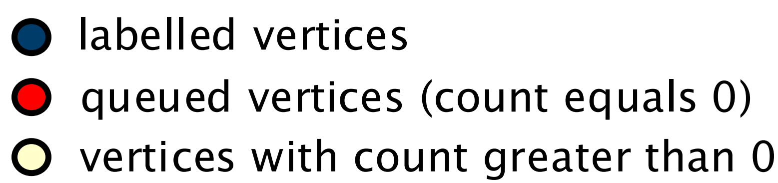
Directed acyclic graph **D**

source queue: ⟨**r**,**z**⟩





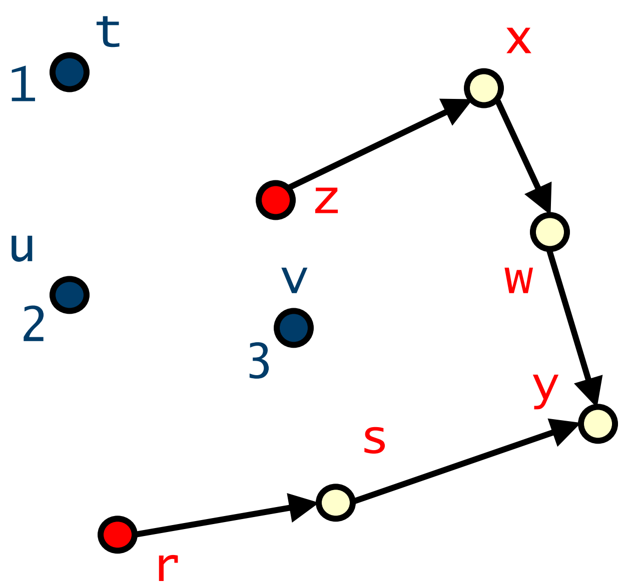
label and remove v from the graph and source queue

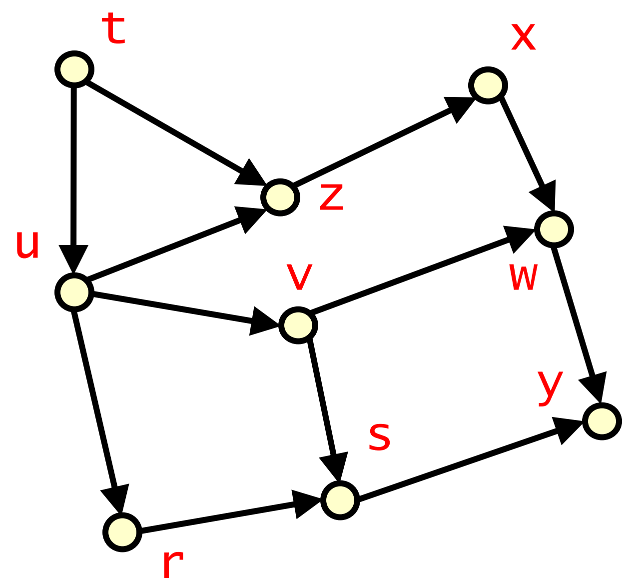


Topological ordering – Example

Directed acyclic graph **D**

source queue: **⟨r,z⟩**





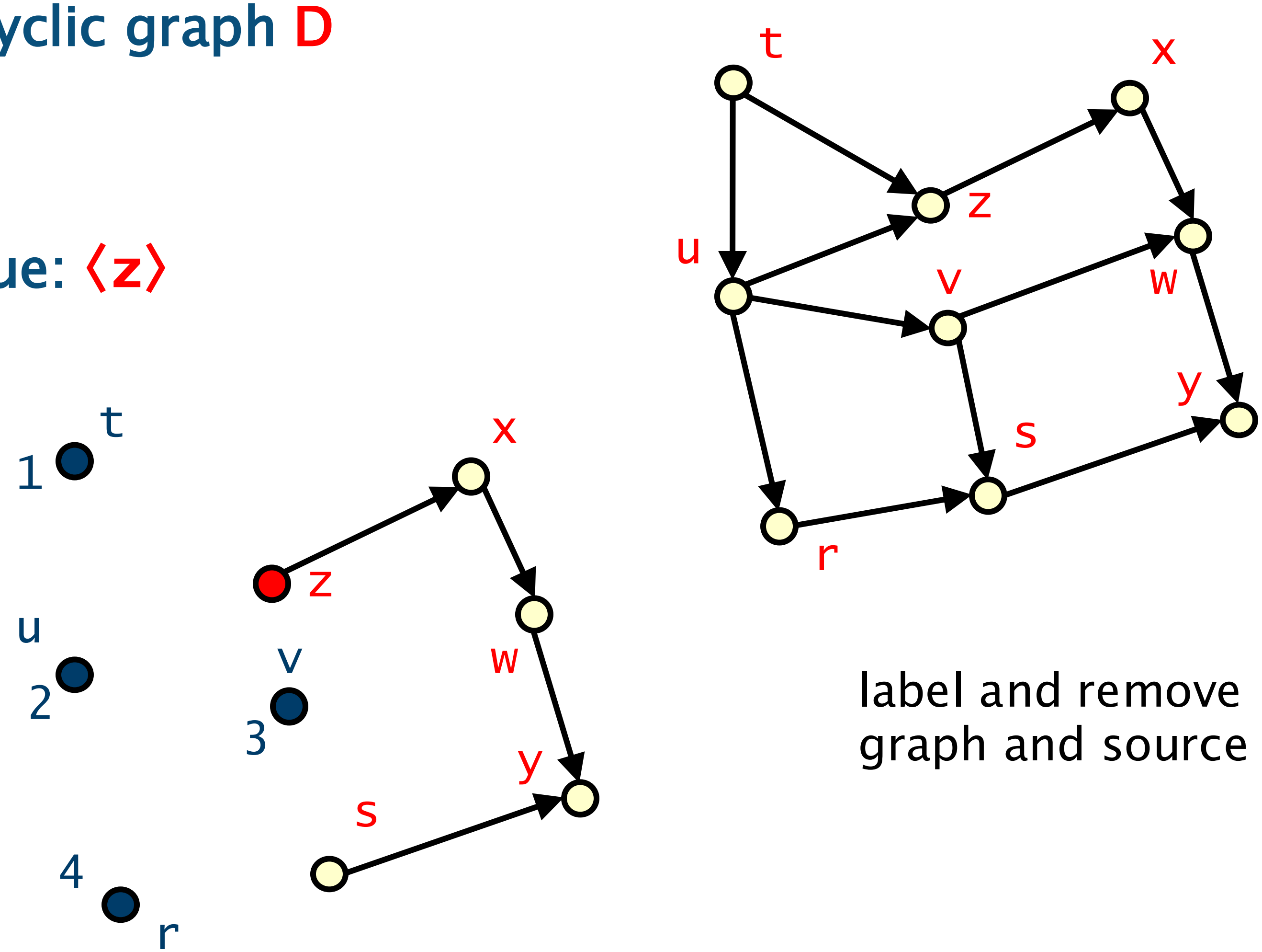
no new vertices have zero incoming edges so source queue remains unchanged

- labelled vertices
- queued vertices (count equals 0)
- vertices with count greater than 0

Topological ordering – Example

Directed acyclic graph **D**

source queue: ⟨**z**⟩



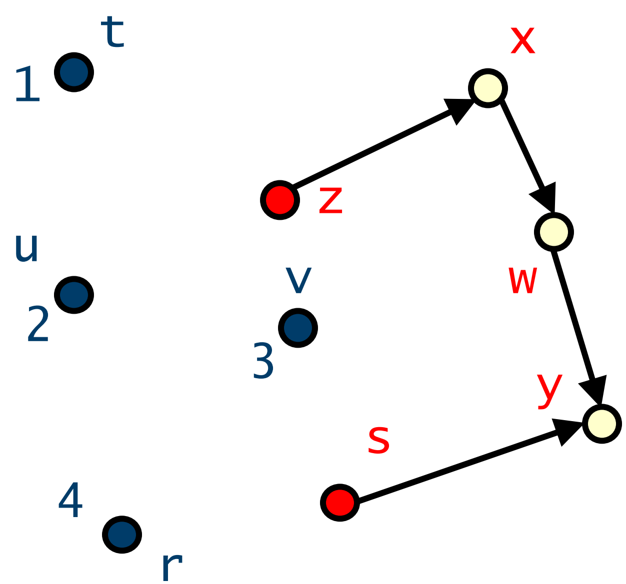
label and remove r from the graph and source queue

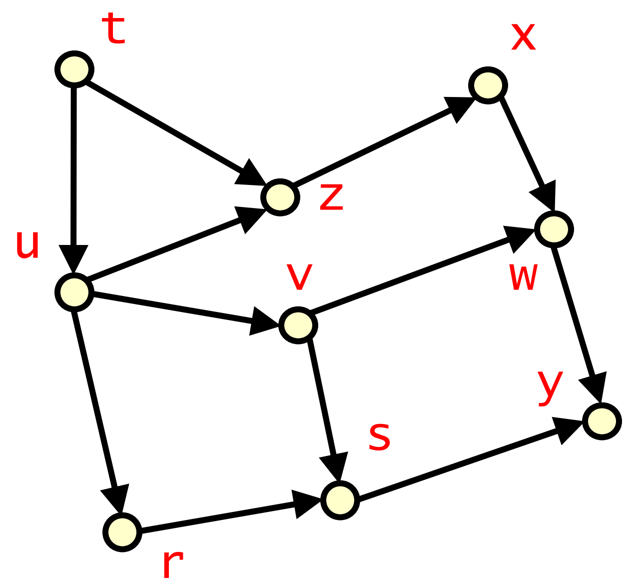
- labelled vertices
- queued vertices (count equals 0)
- vertices with count greater than 0

Topological ordering – Example

Directed acyclic graph D

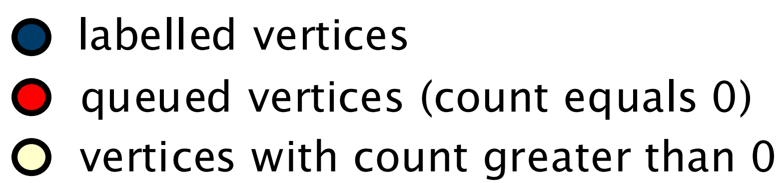
source queue: ⟨**z,s**⟩





label and remove r from the graph and source queue

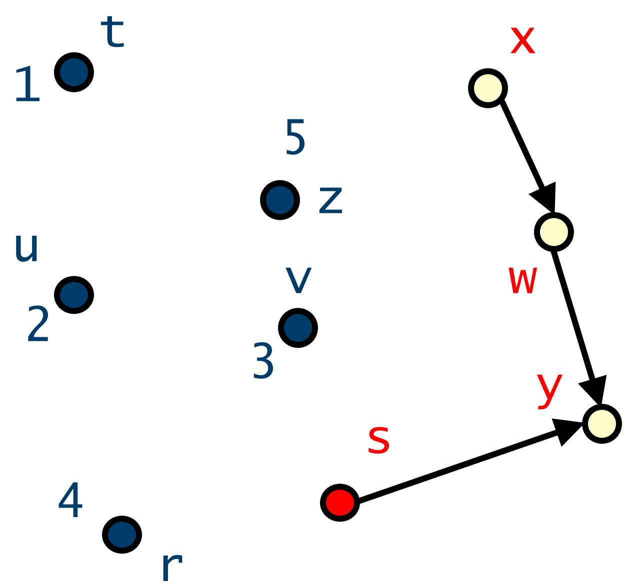
s now has no incoming edges so add to the queue

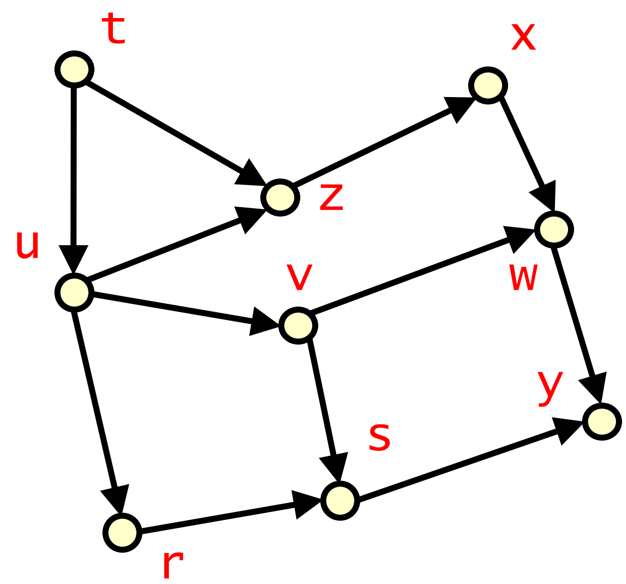


Topological ordering – Example

Directed acyclic graph **D**

source queue: ⟨**s**⟩





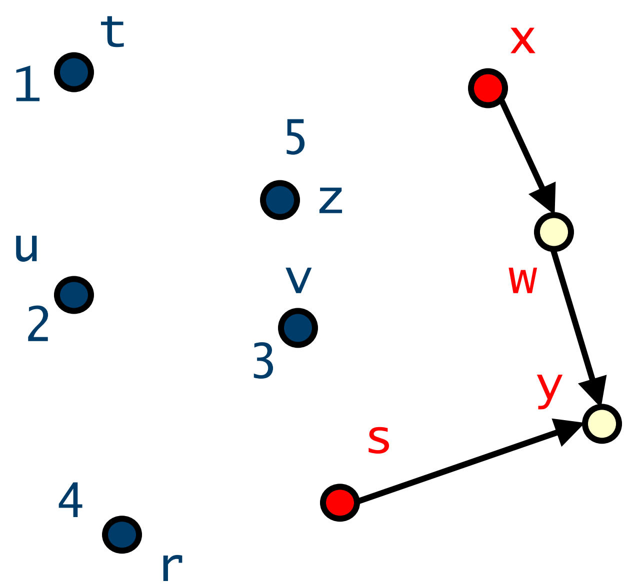
label and remove z from the graph and source queue

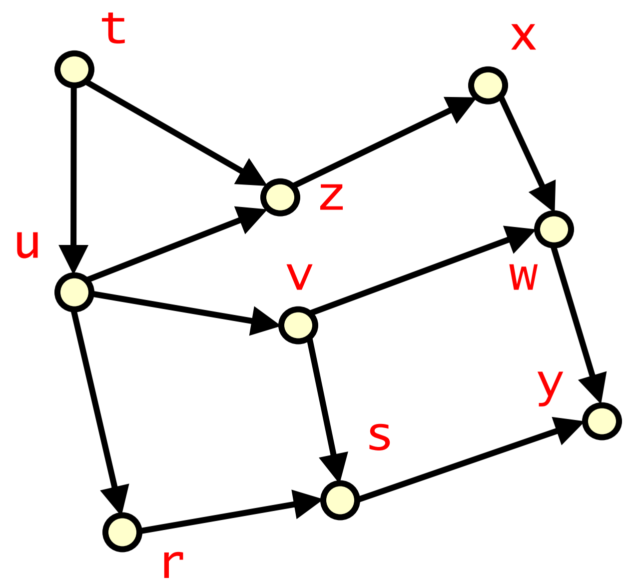
- labelled vertices
- queued vertices (count equals 0)
- vertices with count greater than 0

Topological ordering – Example

Directed acyclic graph **D**

source queue: **⟨s,x⟩**





label and remove z from the graph and source queue

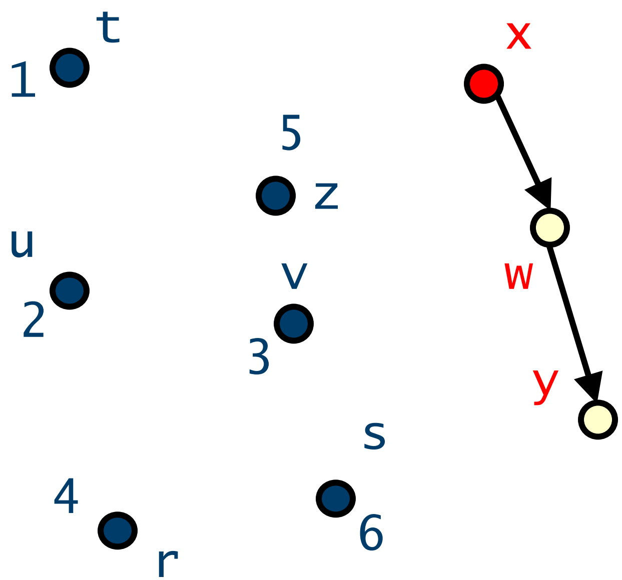
x now has no incoming edges so add to the source queue

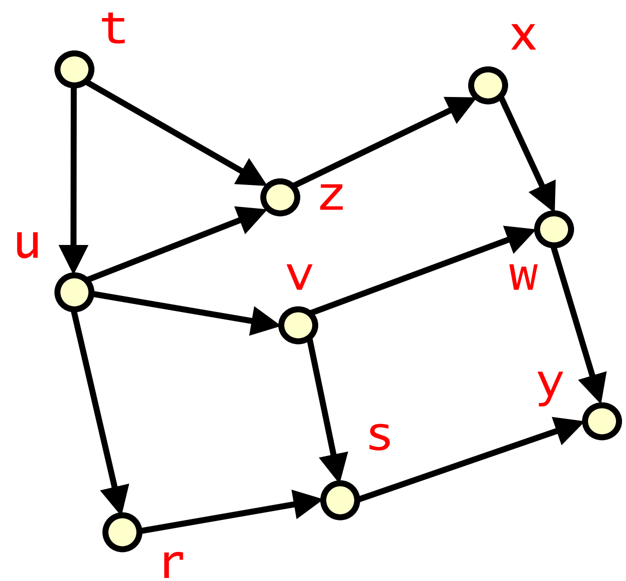
- labelled vertices
- queued vertices (count equals 0)
- vertices with count greater than 0

Topological ordering – Example

Directed acyclic graph **D**

source queue: ⟨**x**⟩





label and remove s from the graph and source queue

labelled vertices

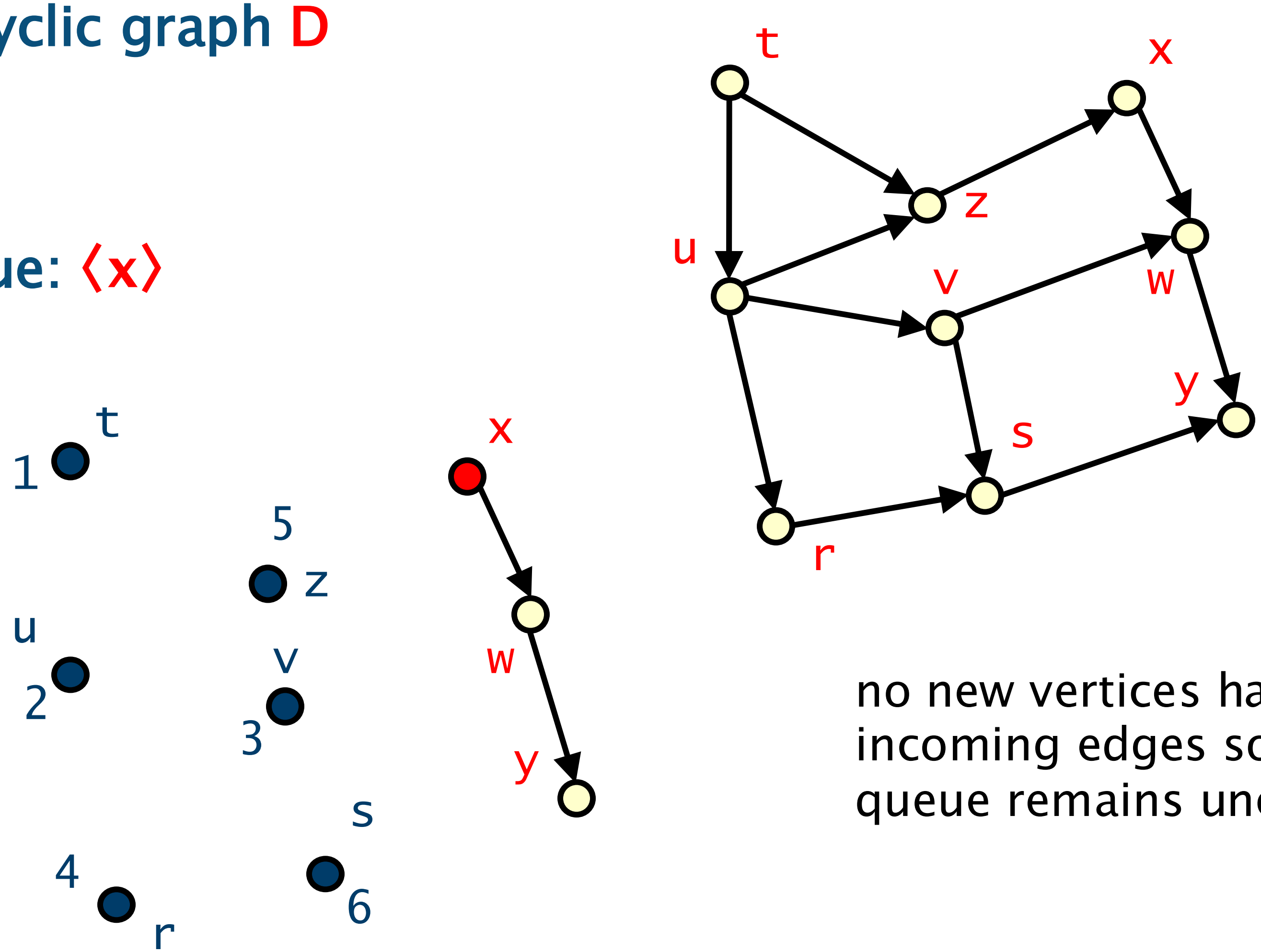
queued vertices (count equals 0)

vertices with count greater than 0

Topological ordering – Example

Directed acyclic graph **D**

source queue: **⟨x⟩**



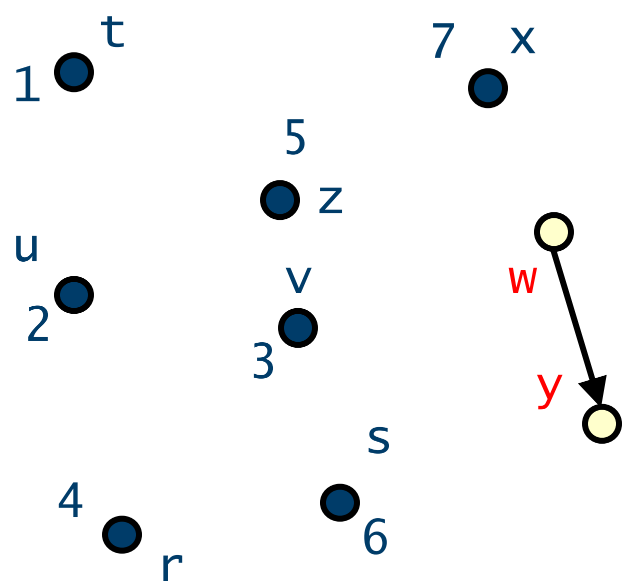
no new vertices has zero incoming edges so source queue remains unchanged

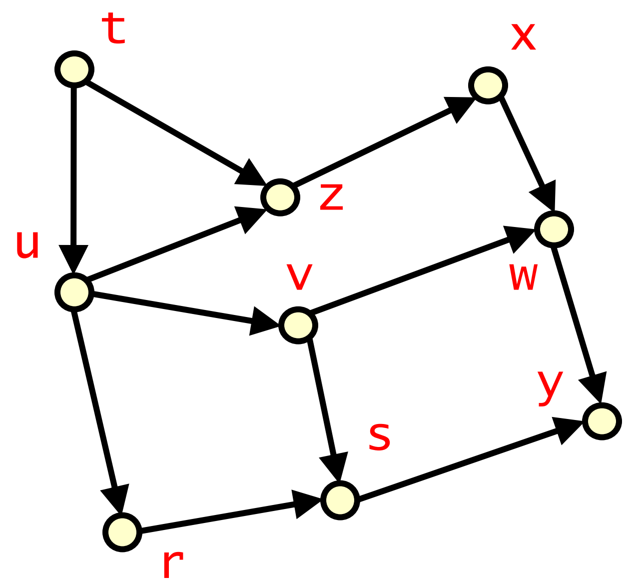
- labelled vertices
- queued vertices (count equals 0)
- vertices with count greater than 0

Topological ordering – Example

Directed acyclic graph D

source queue: ⟨⟩





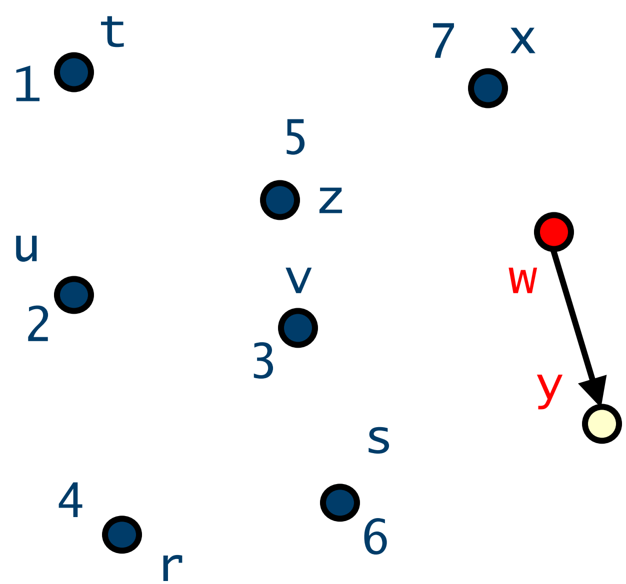
label and remove x from the graph and source queue

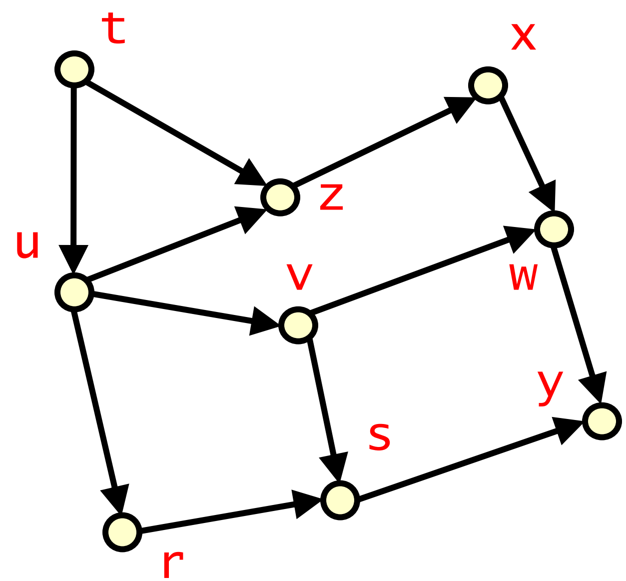
- labelled vertices
- queued vertices (count equals 0)
- vertices with count greater than 0

Topological ordering – Example

Directed acyclic graph D

source queue: ⟨w⟩



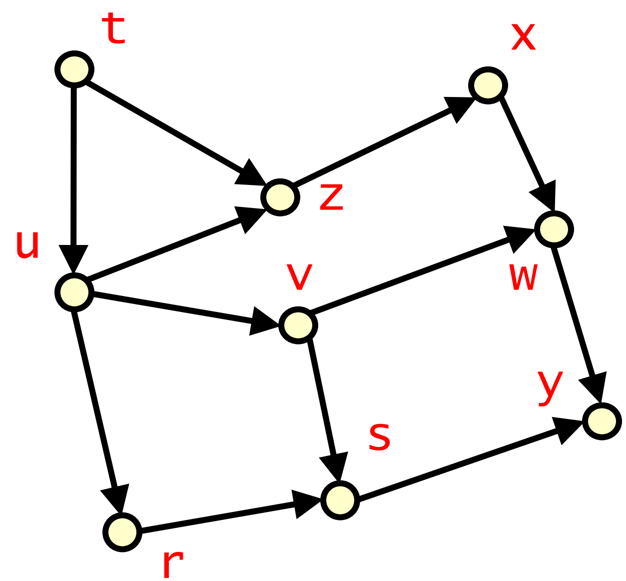


w now has no incoming edges so added to the queue

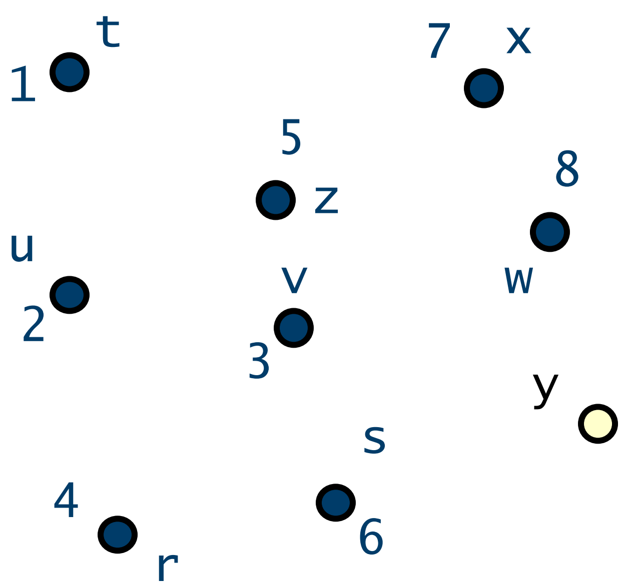
- labelled vertices
- queued vertices (count equals 0)
- vertices with count greater than 0

Topological ordering – Example

Directed acyclic graph D



source queue: ⟨⟩



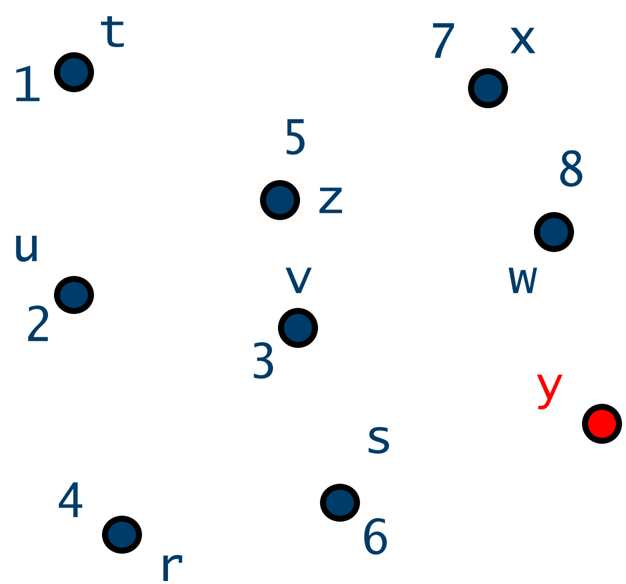
label and remove w from the graph and source queue

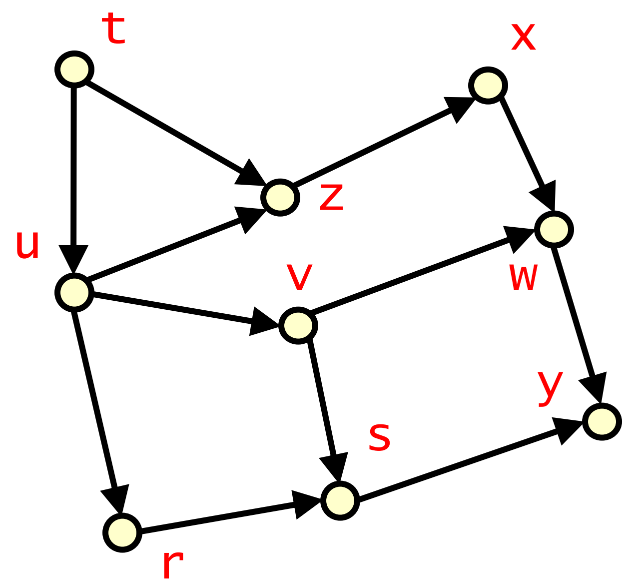
- labelled vertices
- queued vertices (count equals 0)
- vertices with count greater than 0

Topological ordering – Example

Directed acyclic graph **D**

source queue: **⟨y⟩**





y now has no incoming edges so added to the queue

labelled vertices

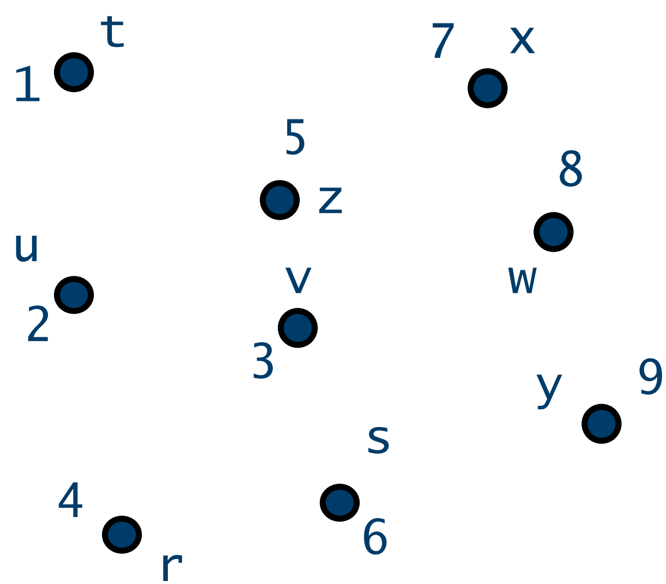
queued vertices (count equals 0)

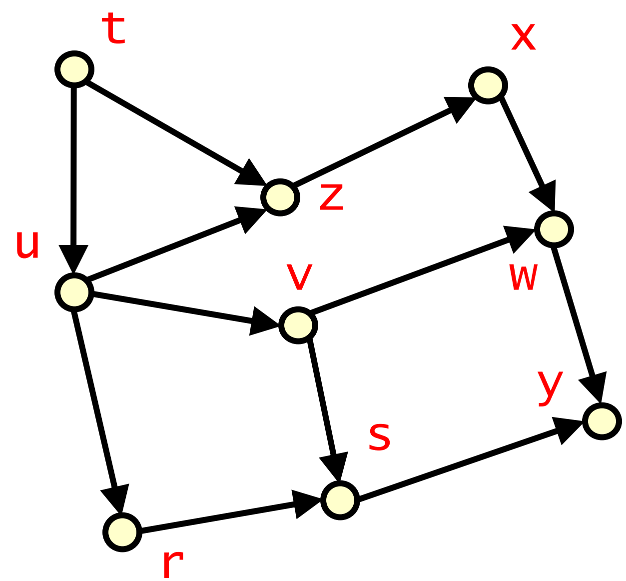
vertices with count greater than 0

Topological ordering – Example

Directed acyclic graph D

source queue: ⟨⟩



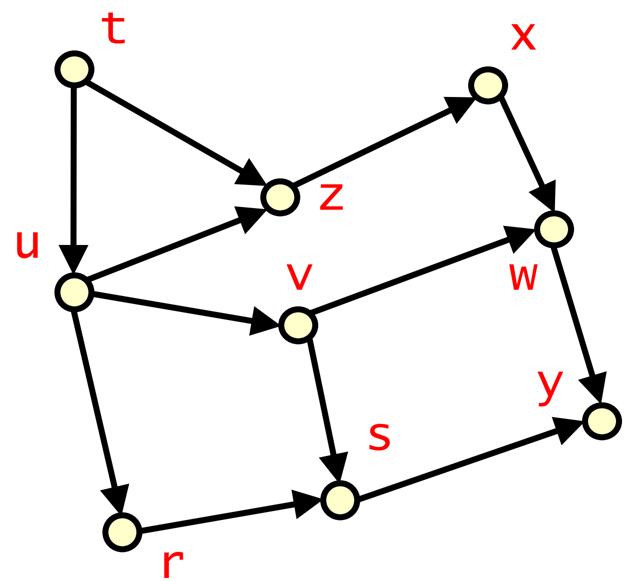


after labelling and removing y, the topological ordering is complete

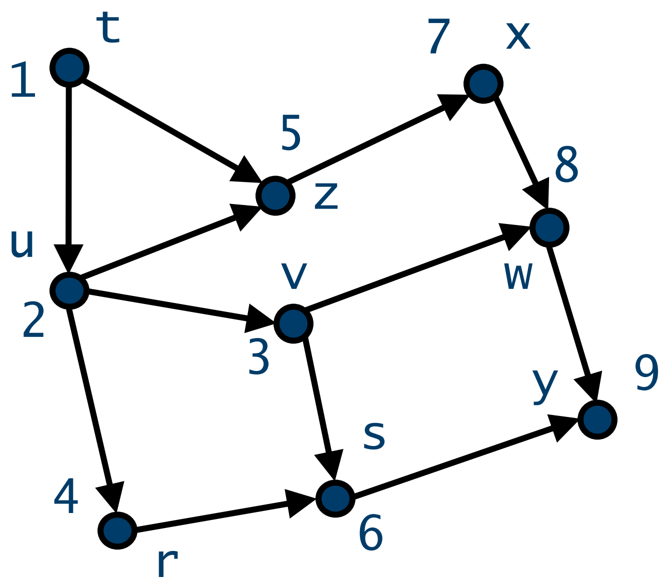
- labelled vertices
- queued vertices (count equals 0)
- vertices with count greater than 0

Topological ordering – Example

Directed acyclic graph D



source queue: ⟨⟩



a topological ordering on D

Topological ordering algorithm

```
// assume each vertex has 2 integer attributes: label and count
// count is the number of incoming edges from unlabelled vertices
// label will give the topological ordering

for (each vertex v) v.setCount(v.getInDegree()); // initial count values

Set up an empty sourceQueue

for (each vertex v) // add vertices with no incoming edges to the queue
  if (v.getCount() == 0) add v to sourceQueue; // i.e. source vertices

int nextLabel = 1; // initialise labelling (gives topological ordering)
while (sourceQueue is non-empty){
  dequeue v from sourceQueue;
  v.setLabel(nextLabel++); // label vertex (and increment nextLabel)
  for (each w adjacent from v){ // consider each vertex w adjacent from v
    w.setCount(w.getCount() - 1); // update attribute count
    // add vertex to source queue if there are no incoming vertices
    if (w.getCount() == 0) add w to sourceQueue;
  }
}
```

Topological ordering algorithm – Correctness

A vertex is given a label only when the number of incoming edges from unlabelled vertices is zero

- all predecessor vertices must already be labelled with smaller numbers
- dependent on using a queue (first-in-first-out for labelling)

Topological ordering algorithm – Analysis

Analysis (n vertices, m edges)

- **for adjacency lists representation**
  - finding in-degree of each vertex is O(n+m)
    - set the count for each vertex
    - scan the adjacency list for each vertex
  - main loop is executed n times
    - each time one adjacency list is scanned – that of the vertex being labelled
    - decrement the count of all vertices that have an incoming edge with the vertex labelled as source
    - therefore the same list is never scanned twice
  - so every list is scanned again and overall algorithm is O(n+m)

Topological ordering algorithm – Analysis

Analysis (n vertices, m edges)

- **for adjacency matrix representation**
  - finding in-degree of each vertex is $O(n^2)$
    - scan each column to find the in-degree
  - main loop is executed n times within it one row is scanned $O(n)$
    - `looking for edges going out`
  - so overall the algorithm is $O(n^2)$

Section 3 – Graphs and graph algorithms

Graph basics

- definitions: directed, undirected, connected, bipartite, …

Graph representations

- adjacency matrix/lists and implementation

Graph search and traversal algorithms

- depth/breadth first search

Topological ordering

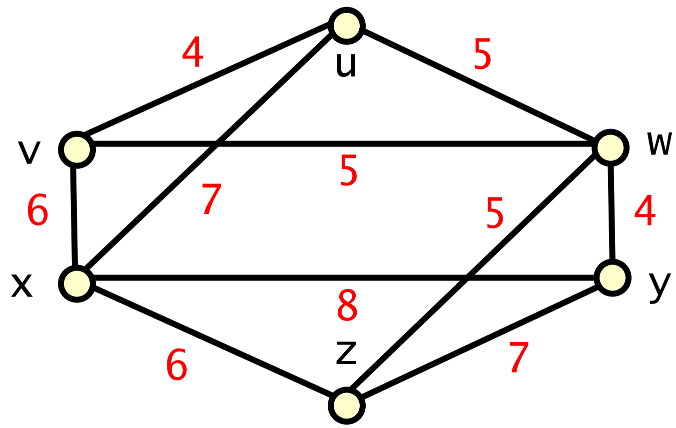
Weighted graphs

- shortest path (Dijkstra's algorithm)
- minimum spanning tree (Prim–Jarnik and Dijkstra's refinement)

Weighted graphs

Each edge **e** has an integer **weight** given by `wt(e)>0`

- graph may be undirected or directed
- weight may represent length, cost, capacity, etc
- if an edge is not part of the graph its weight is assumed to be infinity



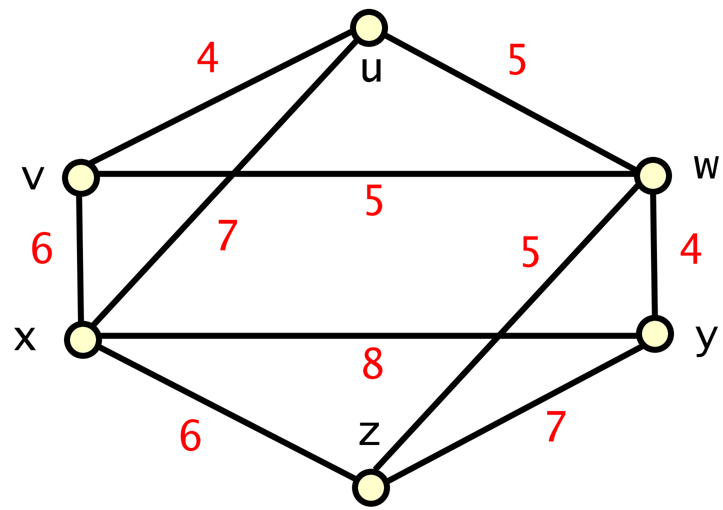
Example: cost of sending a message down a particular edge

- could be a monetary cost or some combination of time and distance
- can be used to formulate the shortest path problem for routing packets

Weighted graphs – Representation

Adjacency matrix becomes **weight matrix**

Adjacency lists include weight in node



adjacency matrix

	u	v	w	x	y	z
u	0	4	5	7	0	0
v	4	0	5	6	0	0
w	5	5	0	0	4	5
x	7	6	0	0	8	6
y	0	0	4	8	0	7
z	0	0	5	6	7	0

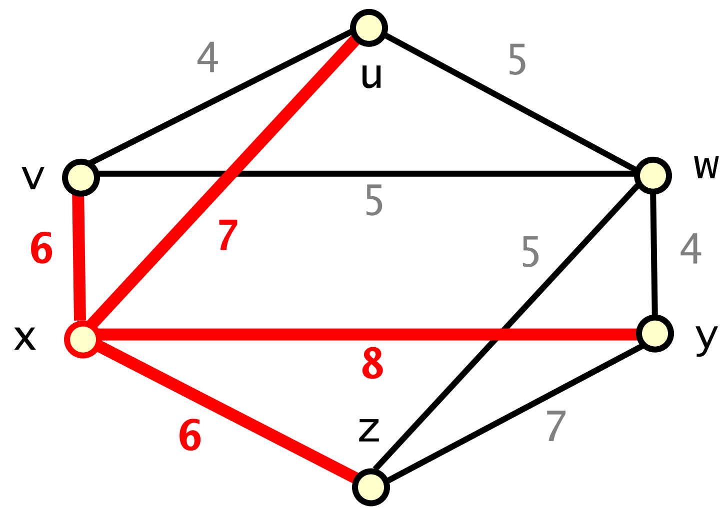
adjacency list

u:v(4)→w(5)→x(7)
v:u(4)→w(5)→x(6)
w:u(5)→v(5)→y(4)→z(5)
x:u(7)→v(6)→y(8)→z(6)
y:w(4)→x(8)→z(7)
z:w(5)→x(6)→y(7)

Weighted graphs – Representation

Adjacency matrix becomes **weight matrix**

Adjacency lists include weight in node



adjacency matrix

	u	v	w	x	y	z
u	0	4	5	7	0	0
v	4	0	5	6	0	0
w	5	5	0	0	4	5
x	**7**	**6**	**0**	**0**	**8**	**6**
y	0	0	4	8	0	7
z	0	0	5	6	7	0

adjacency list

u:v(4)→w(5)→x(7)
v:u(4)→w(5)→x(6)
w:u(5)→v(5)→y(4)→z(5)
x:u(7)→v(6)→y(8)→z(6)
y:w(4)→x(8)→z(7)
z:w(5)→x(6)→y(7)

Section 3 – Graphs and graph algorithms

Graph basics

- definitions: directed, undirected, connected, bipartite, …

Graph representations

- adjacency matrix/lists and implementation

Graph search and traversal algorithms

- breadth/depth first search

Weighted graphs

- shortest path (Dijkstra's algorithm)
- minimum spanning tree (Prim–Jarnik and Dijkstra's refinement)

Topological ordering

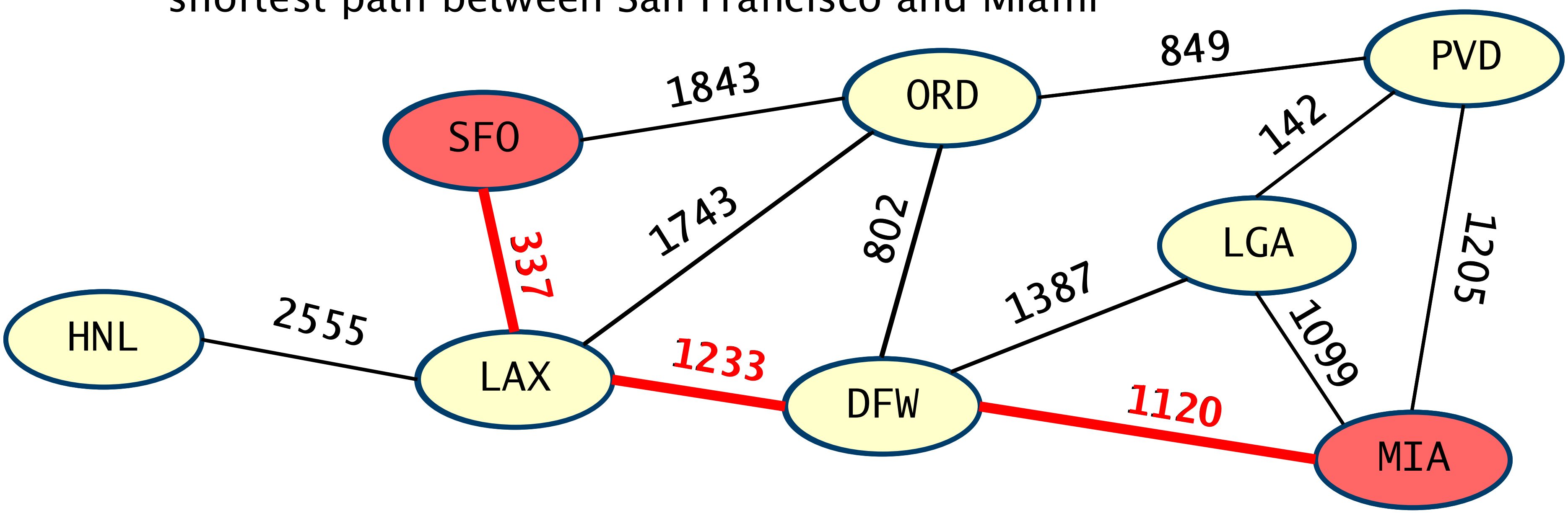
Weighted graphs – Shortest Paths

Given a weighted (un)directed graph and two vertices u and v find a shortest path between u and v (for directed from u to v)

- where the length of a path is the sum of the weights of its edges

Example: weights are distances between airports

- shortest path between San Francisco and Miami



Weighted graphs – Shortest Paths

Given a weighted (un)directed graph and two vertices u and v find a shortest path between u and v (for directed from u to v)

– where the length of a path is the sum of the weights of its edges

Example: weights are distances between airports

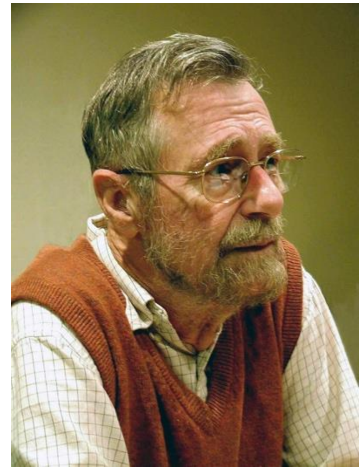
– shortest path between San Francisco and Miami

Applications include:

– flight reservations
– internet packet routing
– driving directions

Edsger Dijkstra, in an interview in 2010...

"… the algorithm for the shortest path, which I designed in about 20 minutes. One morning I was shopping in Amsterdam with my young fiancé, and tired, we sat down on the cafe terrace to drink a cup of coffee, and I was just thinking about whether I could do this, and I then designed the algorithm for the shortest path."



Dijkstra, E.W. A note on two problems in Connexion with graphs. Numerische Mathematik 1, 269–271 (1959)

Dijkstra describes the algorithm in English in 1956 (he was 26 years old)

- most people were programming in assembly language
- only one high-level language: Fortran by John Backus at IBM and not quite finished

No **big O** notation in 1959, in the paper, Dijkstra says: “my solution is preferred to another one … the amount of work to be done seems considerably less.”

Dijkstra's algorithm

Algorithm finds shortest path between one vertex u and all others

- based on maintaining a set S containing all vertices for which shortest path with u is currently known
- S initially contains only u (obviously shortest path between u and u is 0)
- eventually S contains all the vertices (so all shortest paths are known)

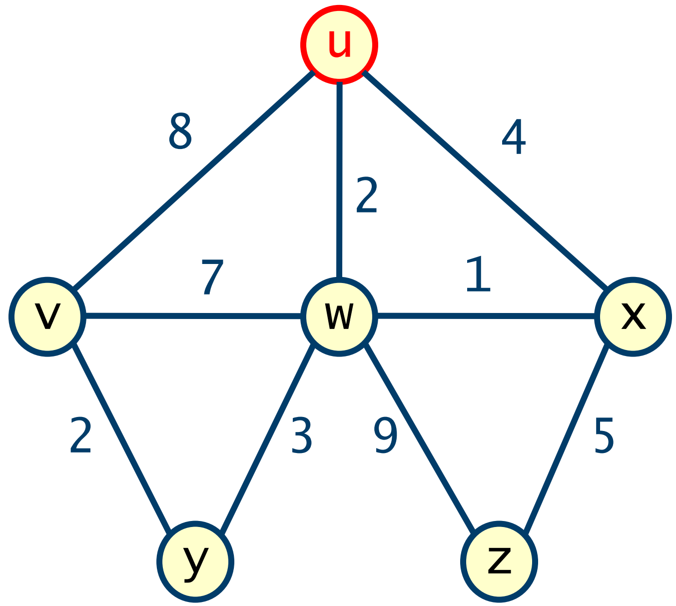
Each vertex v not in S has a label `dist(v)` indicating the length of a shortest path between u and v passing only through vertices in S

- if no path exists then we set `dist(v)` to infinity
- first step: if v is adjacent to u then dist(v)=wt({u,v}), otherwise dist(v)=∞
- let's see how the algorithm works on an example and then get back to formalising it and the pseudocode implementation

Dijkstra's algorithm – Example

Weighted graph

- compute shortest path with u

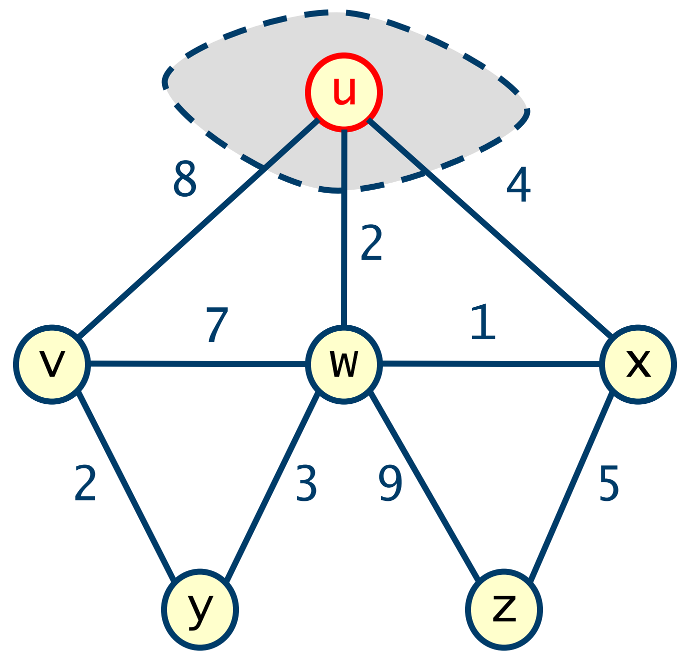


Dijkstra's algorithm – Example

Weighted graph

- compute shortest path with u

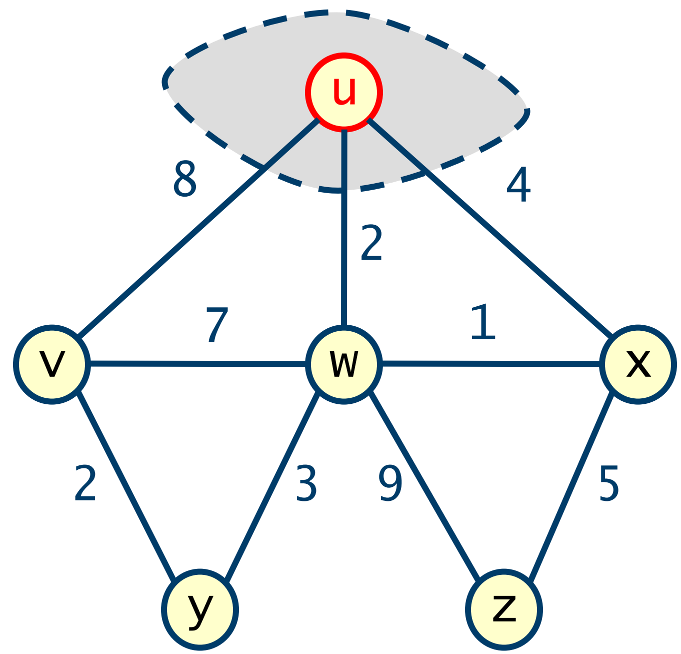
S={u}



Dijkstra's algorithm – Example

Weighted graph

- compute shortest path with u



S={u}

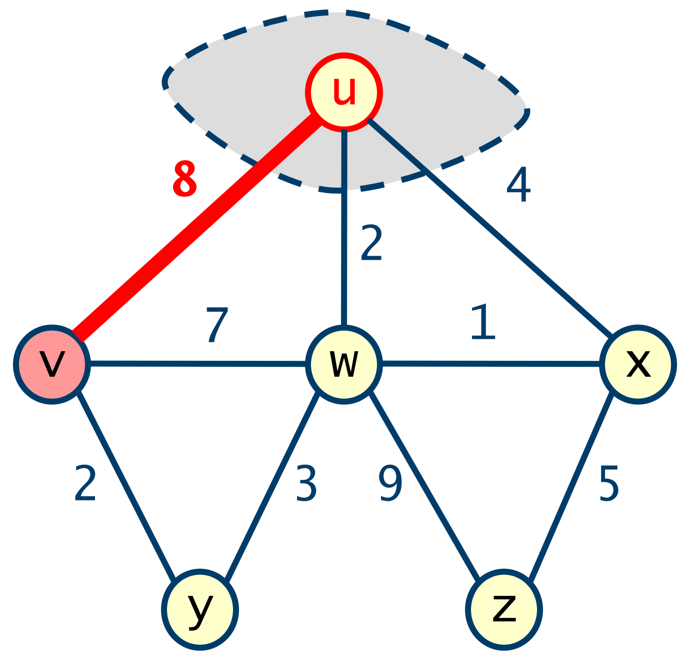
dist(u)=0
dist(v)=8
dist(w)=2
dist(x)=4
dist(y)=∞
dist(z)=∞

compute distances

Dijkstra’s algorithm – Example

Weighted graph

- compute shortest path with u



S={u}

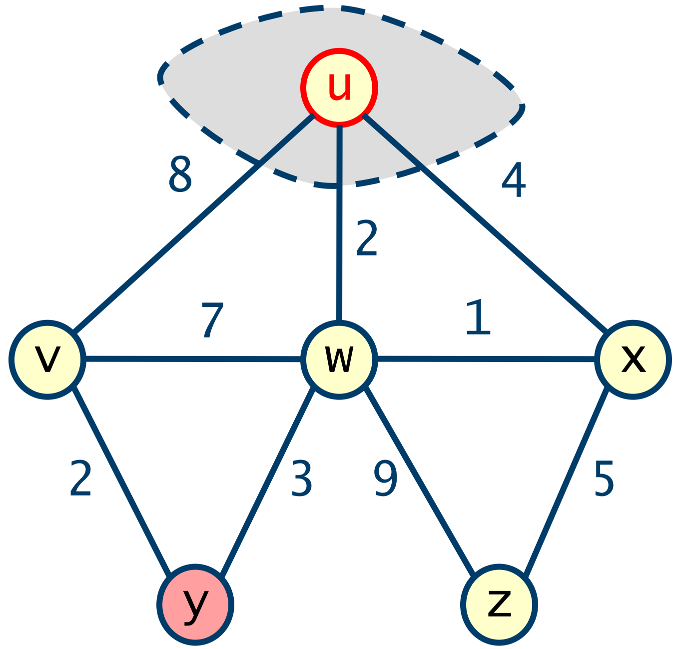
dist(u)=0
dist(v)=8
dist(w)=2
dist(x)=4
dist(y)=∞
dist(z)=∞

compute distances

Dijkstra's algorithm – Example

Weighted graph

– compute shortest path with u



S={u}

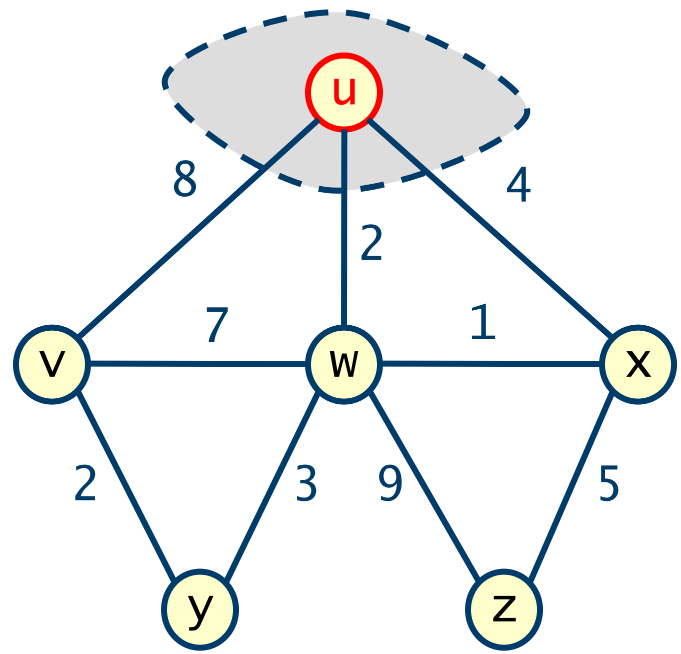
dist(u)=0
dist(v)=8
dist(w)=2
dist(x)=4
dist(y)=∞
dist(z)=∞

compute distances

Dijkstra’s algorithm – Example

Weighted graph

- compute shortest path with u



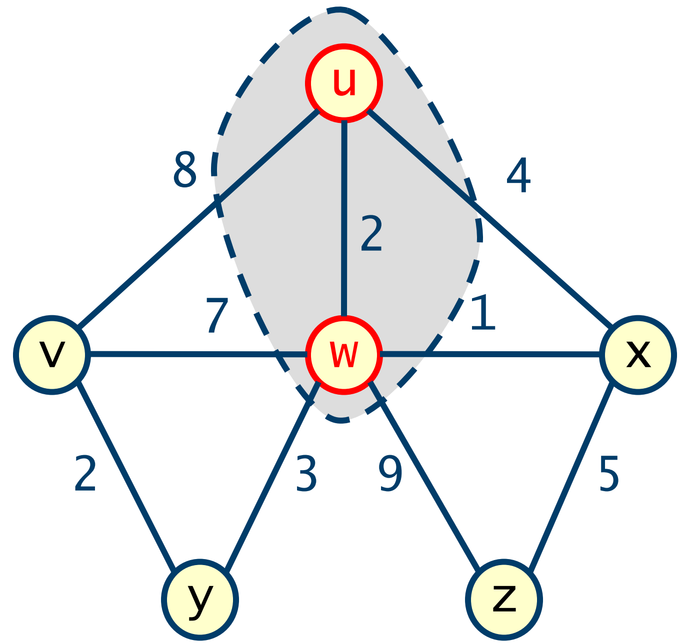
S={u}

dist(u)=0
dist(v)=8
dist(w)=2 ← minimum distance
dist(x)=4
dist(y)=∞
dist(z)=∞

Dijkstra’s algorithm – Example

Weighted graph

– compute shortest path with u



S={u,w}

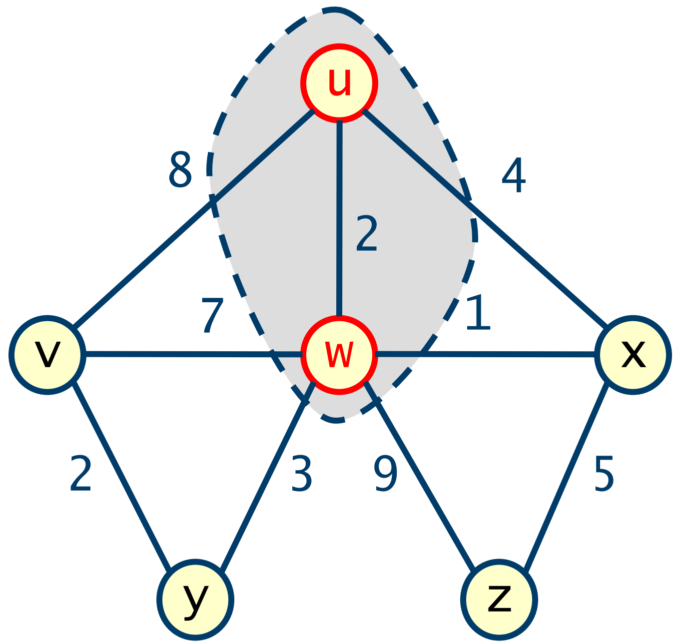
dist(u)=0
dist(v)=8
dist(w)=2
dist(x)=4
dist(y)=∞
dist(z)=∞

add w to the set S

Dijkstra’s algorithm – Example

Weighted graph

- compute shortest path with u



S={u,w}

dist(u)=0
dist(v)=8→min{8,2+7}=8
dist(w)=2
dist(x)=4→min{4,2+1}=3
dist(y)=∞→min{∞,2+3}=5
dist(z)=∞→min{∞,2+9}=11

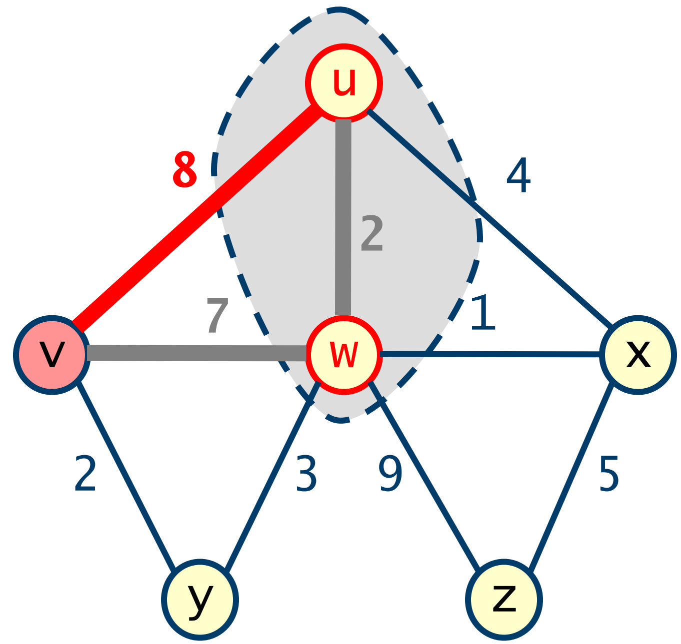
perform relaxation:

dist(vertex)=**min**{dist(vertex),dist(**w**)+wt(vertex,**w**)};

Dijkstra's algorithm – Example

Weighted graph

- compute shortest path with u



S={u,w}

dist(u)=0
dist(v)=8→min{8,2+7}=8
dist(w)=2
dist(x)=4→min{4,2+1}=3
dist(y)=∞→min{∞,2+3}=5
dist(z)=∞→min{∞,2+9}=11

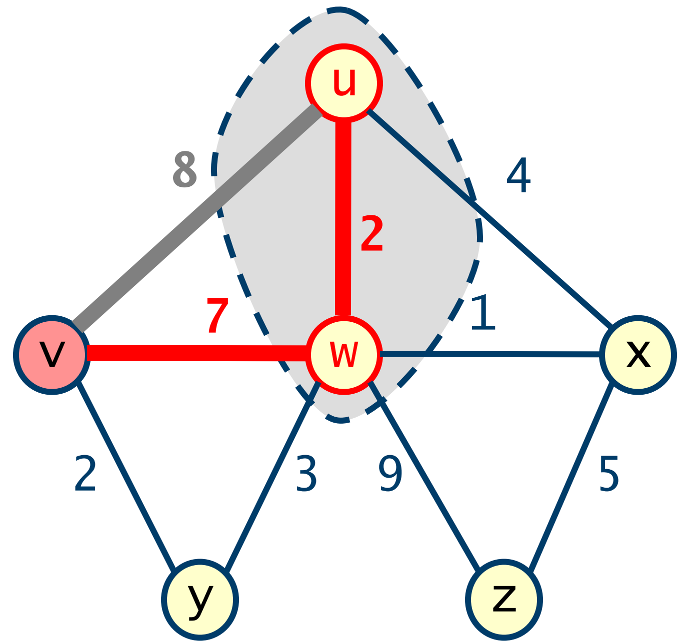
perform relaxation

dist(v)=**min**{dist(v),dist(**w**)+wt(v,**w**)} = **min**{8,2+7}

Dijkstra's algorithm – Example

Weighted graph

- compute shortest path with u



S={u,w}

dist(u)=0
dist(v)=8→min{8,2+7}=8
dist(w)=2
dist(x)=4→min{4,2+1}=3
dist(y)=∞→min{∞,2+3}=5
dist(z)=∞→min{∞,2+9}=11

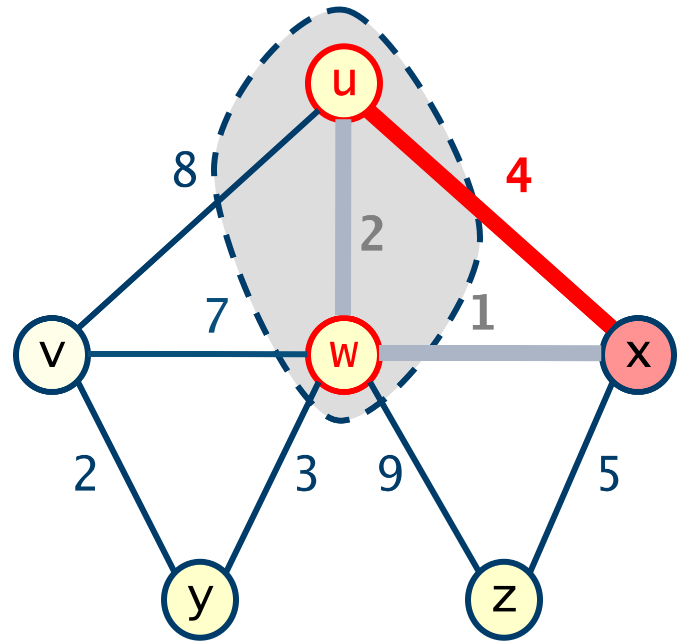
perform relaxation

dist(v)=**min**{dist(v),dist(**w**)+wt(v,**w**)} = **min**{8,2+7}

Dijkstra's algorithm – Example

Weighted graph

- compute shortest path with u



S={u,w}

dist(u)=0
dist(v)=8→min{8,2+7}=8
dist(w)=2
dist(x)=4→min{4,2+1}=3
dist(y)=∞→min{∞,2+3}=5
dist(z)=∞→min{∞,2+9}=11

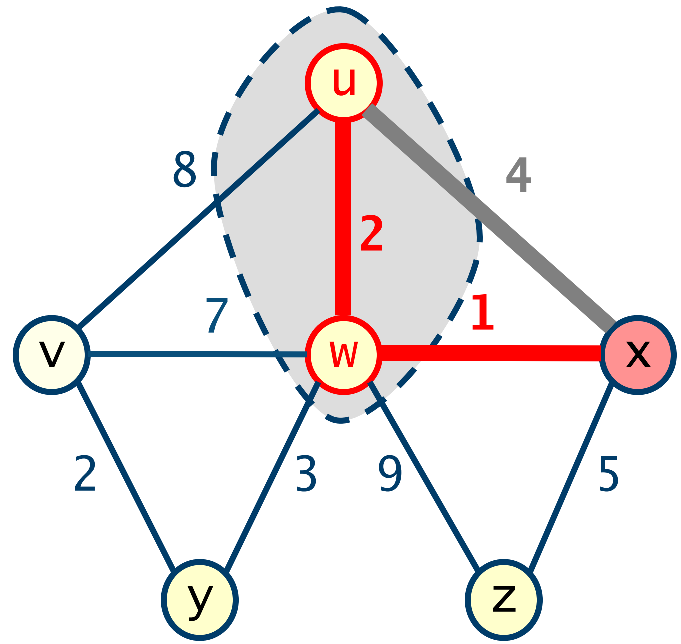
perform relaxation

dist(x)=**min**{dist(x),dist(**w**)+wt(x,**w**)} = **min**{4,2+1}

Dijkstra's algorithm – Example

Weighted graph

– compute shortest path with u



S={u,w}

dist(u)=0
dist(v)=8→min{8,2+7}=8
dist(w)=2
dist(x)=4→min{4,2+1}=3
dist(y)=∞→min{∞,2+3}=5
dist(z)=∞→min{∞,2+9}=11

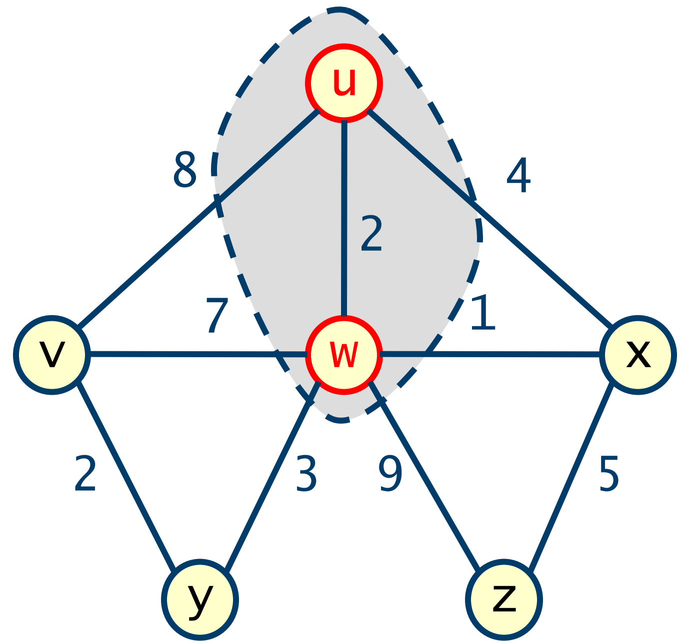
perform relaxation

dist(v)=**min**{dist(v),dist(**w**)+wt(v,**w**)} = **min**{4,2+1}

Dijkstra's algorithm – Example

Weighted graph

– compute shortest path with u



S={u,w}

dist(u)=0
dist(v)=8
dist(w)=2
dist(x)=3
dist(y)=5
dist(z)=11

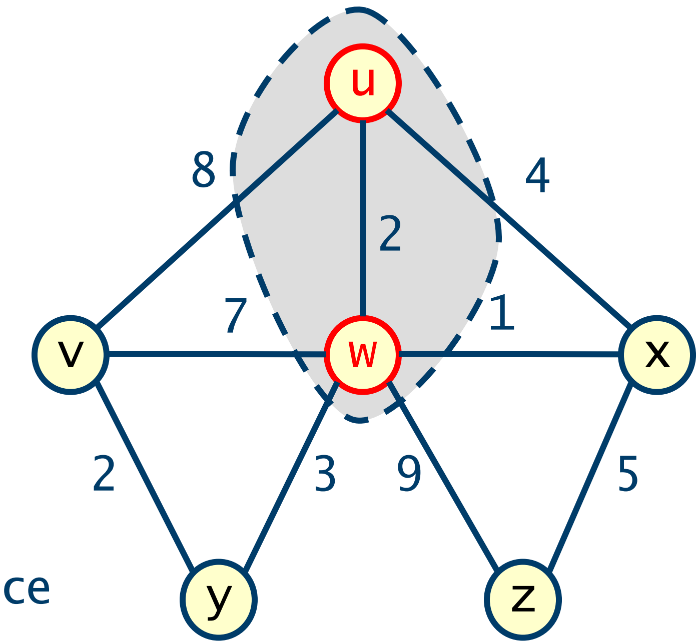
Dijkstra's algorithm – Example

Weighted graph

- compute shortest path with u

S={u,w}

dist(u)=0
dist(v)=8
dist(w)=2
dist(x)=3 ← minimum distance
dist(y)=5
dist(z)=11



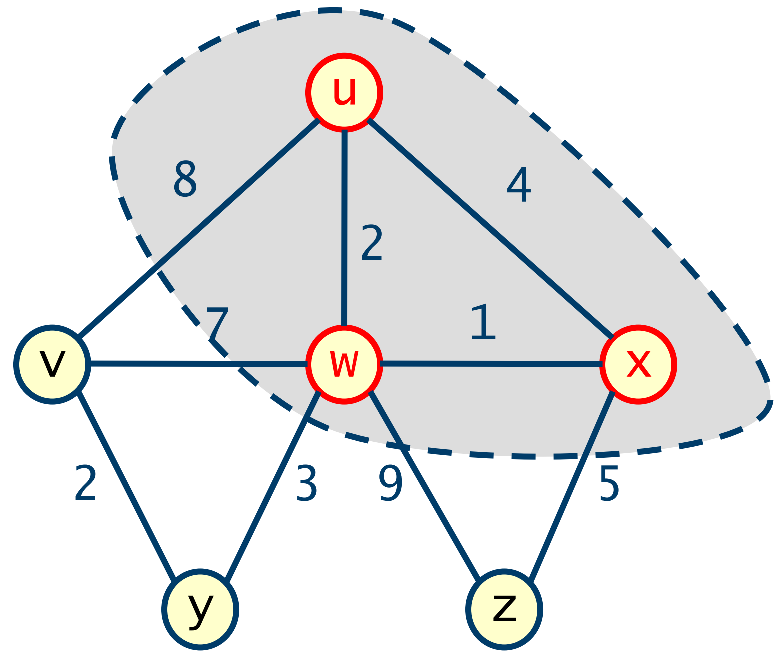
Dijkstra's algorithm – Example

Weighted graph

- compute shortest path with u

S={u,w,x}

dist(u)=0
dist(v)=8
dist(w)=2
dist(x)=3
dist(y)=5
dist(z)=11

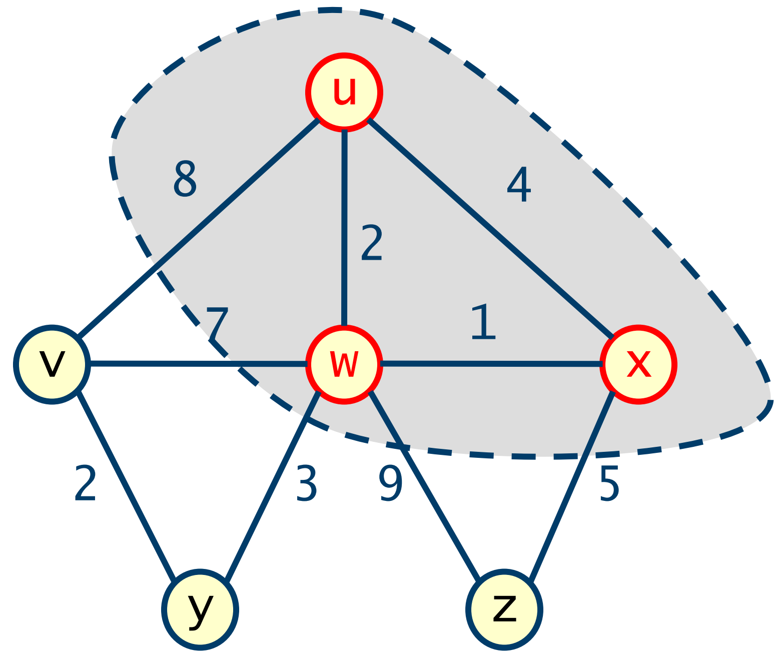


add x to the set S

Dijkstra's algorithm – Example

Weighted graph

- compute shortest path with u



S={u,w,x}

dist(u)=0
dist(v)=8→min{8,3+∞}=8
dist(w)=2
dist(x)=3
dist(y)=5→min{5,3+∞}=5
dist(z)=11→min{11,3+5}=8

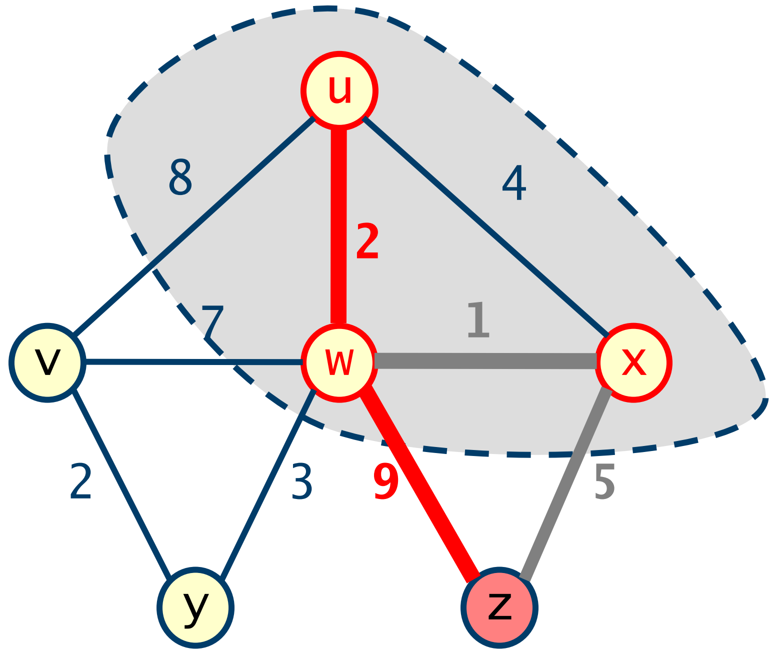
perform relaxation

dist(vertex)=**min**{dist(vertex),dist(**x**)+wt(vertex,**x**)};

Dijkstra's algorithm – Example

Weighted graph

- compute shortest path with u



S={u,w,x}

dist(u)=0
dist(v)=8→min{8,3+∞}
dist(w)=2
dist(x)=3
dist(y)=5→min{5,3+∞}
dist(z)=11→min{11,3+5}=8

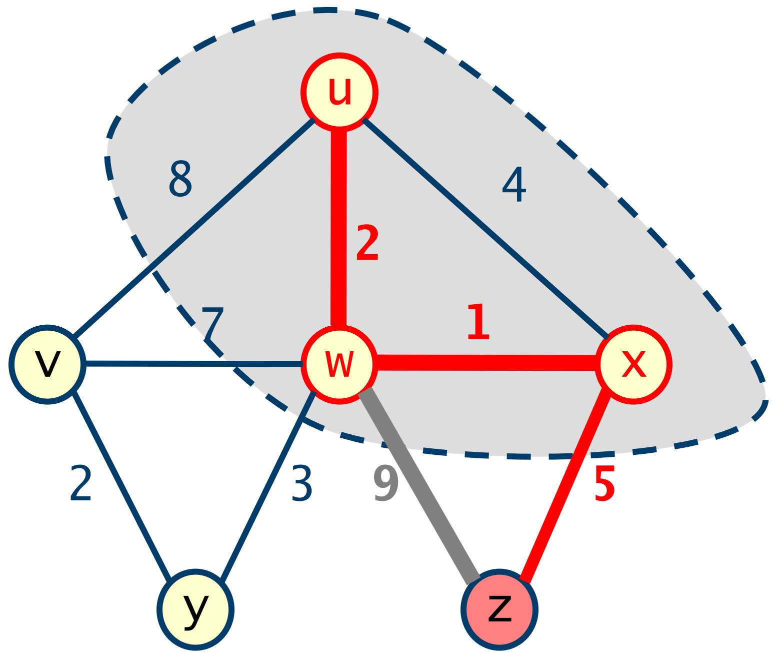
perform relaxation

dist(z)=**min**{dist(z),dist(**x**)+wt(z,**x**)}=**min**{11,3+5}

Dijkstra's algorithm – Example

Weighted graph

– compute shortest path with u



S={u,w,x}

dist(u)=0
dist(v)=8→min{8,3+∞}
dist(w)=2
dist(x)=3
dist(y)=5→min{5,3+∞}
dist(z)=13→min{11,3+5}=8

perform relaxation

dist(z)=**min**{dist(z),dist(**x**)+wt(z,**x**)}=**min**{13,3+5}

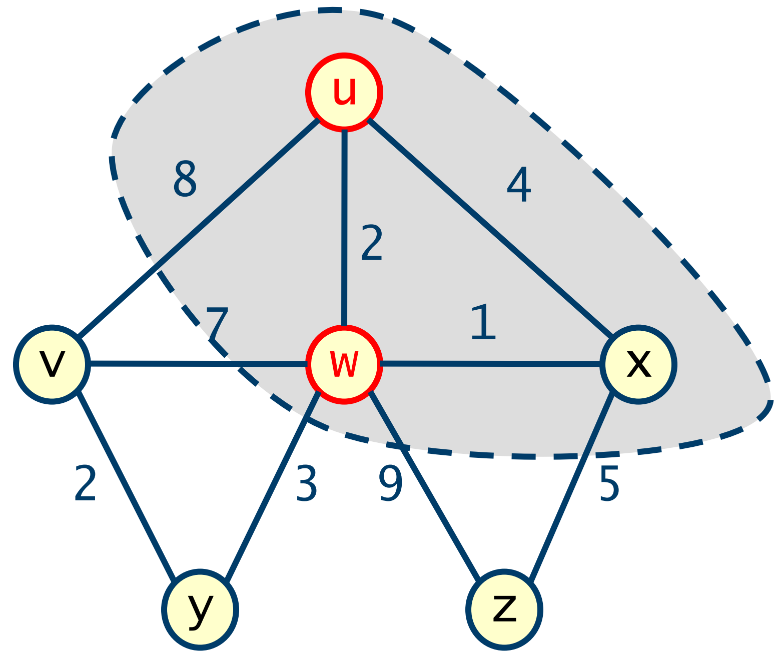
Dijkstra's algorithm – Example

Weighted graph

– compute shortest path with u

S={u,w,x}

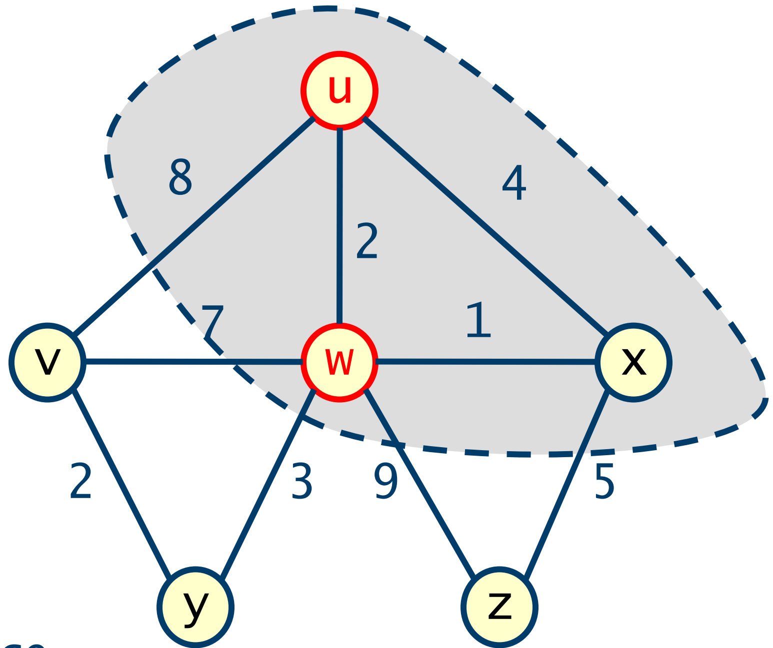
dist(u)=0
dist(v)=8
dist(w)=2
dist(x)=3
dist(y)=5
dist(z)=8



Dijkstra's algorithm – Example

Weighted graph

- compute shortest path with u




```
S={u,w,x}

dist(u)=0
dist(v)=8
dist(w)=2
dist(x)=3
dist(y)=5   ← minimum distance
dist(z)=8
```

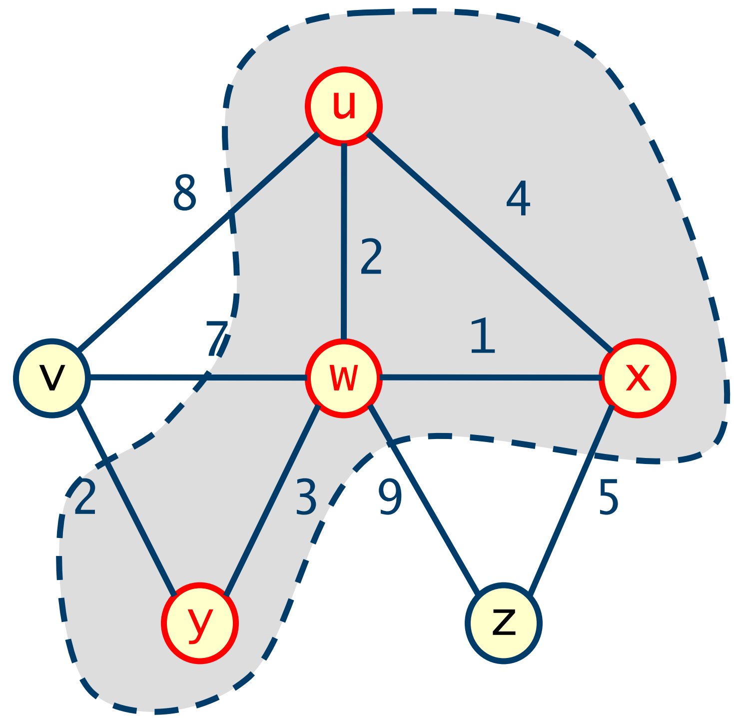
Dijkstra's algorithm – Example

Weighted graph

– compute shortest path with u

S={u,w,x,y}

dist(u)=0
dist(v)=8
dist(w)=2
dist(x)=3
dist(y)=5
dist(z)=8

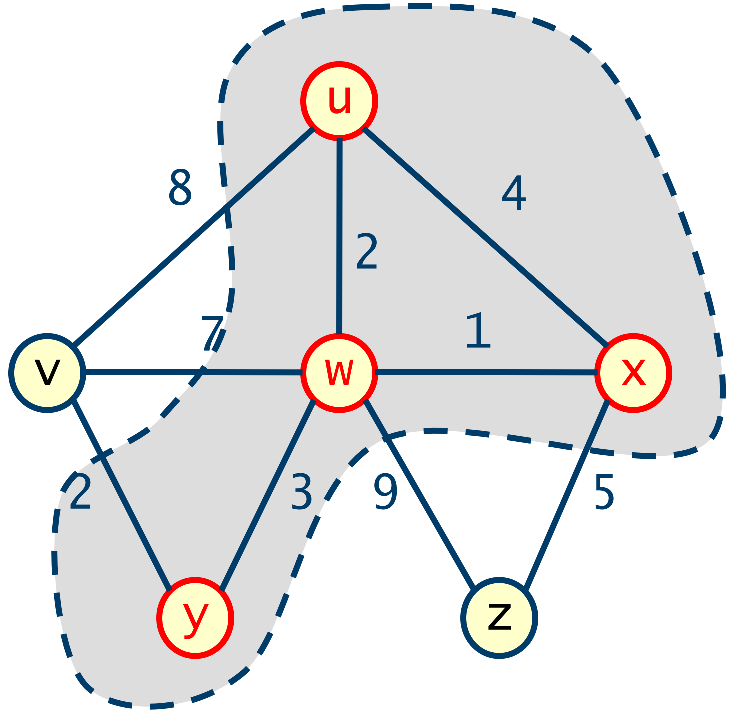


add y to the set S

Dijkstra's algorithm – Example

Weighted graph

– compute shortest path with u



S={u,w,x,y}

dist(u)=0
dist(v)=8→min{8,5+2}=7
dist(w)=2
dist(x)=3
dist(y)=5
dist(z)=8→min{8,5+∞}=8

perform relaxation

dist(vertex)=**min**{dist(vertex),dist(**y**)+wt(vertex,**y**)};

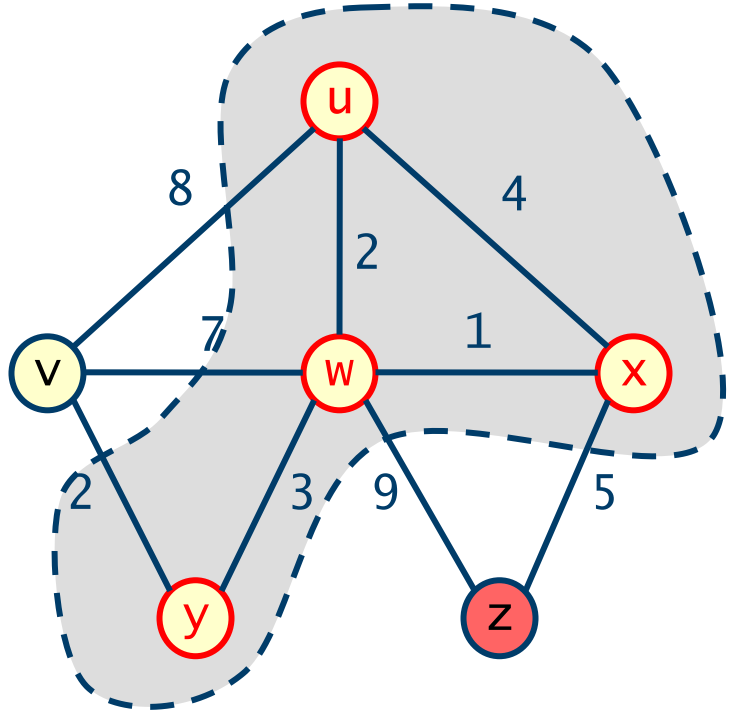
Dijkstra's algorithm – Example

Weighted graph

- compute shortest path with u

S={u,w,x,y}

dist(u)=0
dist(v)=8→min{8,5+2}
dist(w)=2
dist(x)=3
dist(y)=5
dist(z)=8→min{8,5+∞}



perform relaxation

dist(z)=**min**{dist(z),dist(y)+wt(z,y)}=**min**{8,5+∞}

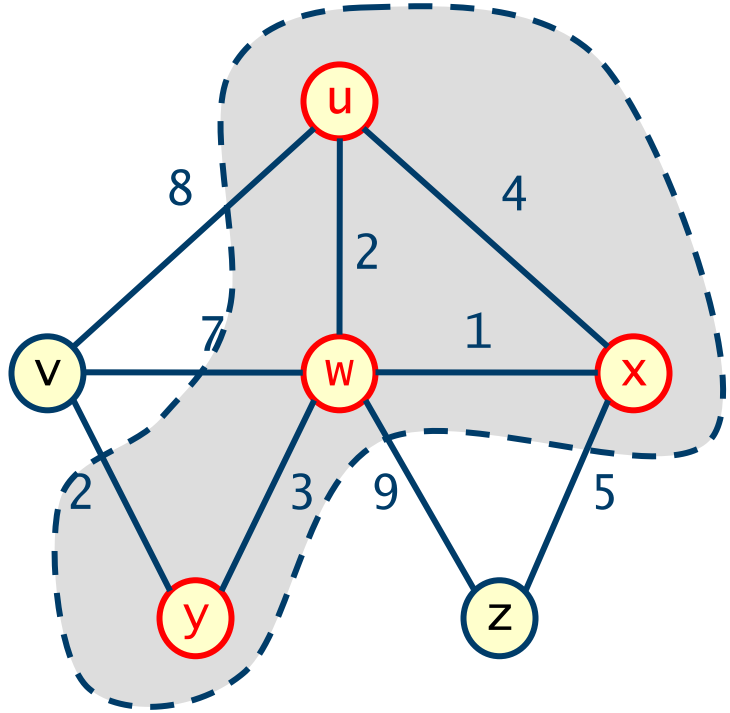
Dijkstra's algorithm – Example

Weighted graph

- compute shortest path with u

S={u,w,x,y}

dist(u)=0
dist(v)=7
dist(w)=2
dist(x)=3
dist(y)=5
dist(z)=8



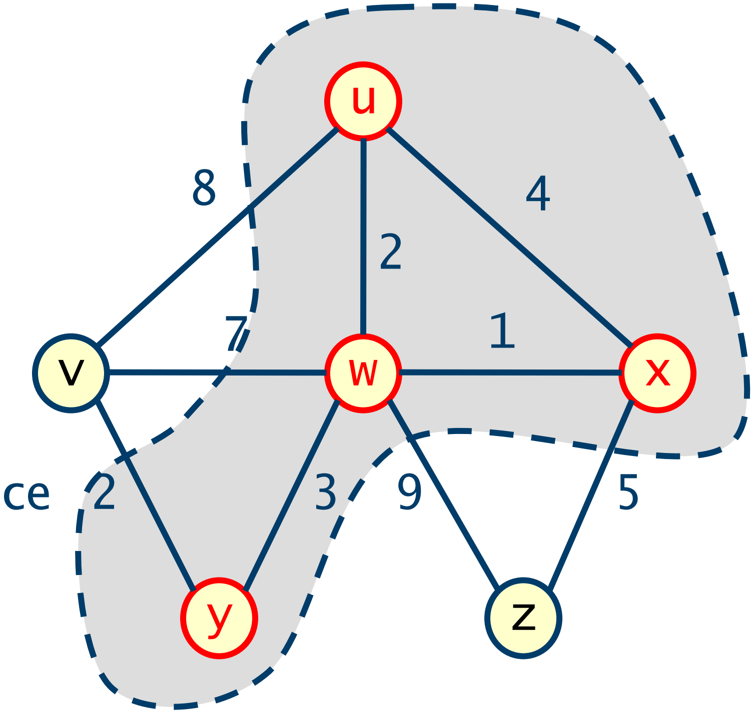
Dijkstra's algorithm – Example

Weighted graph

– compute shortest path with u

S={u,w,x,y}

dist(u)=0
dist(v)=7 ← minimum distance
dist(w)=2
dist(x)=3
dist(y)=5
dist(z)=8



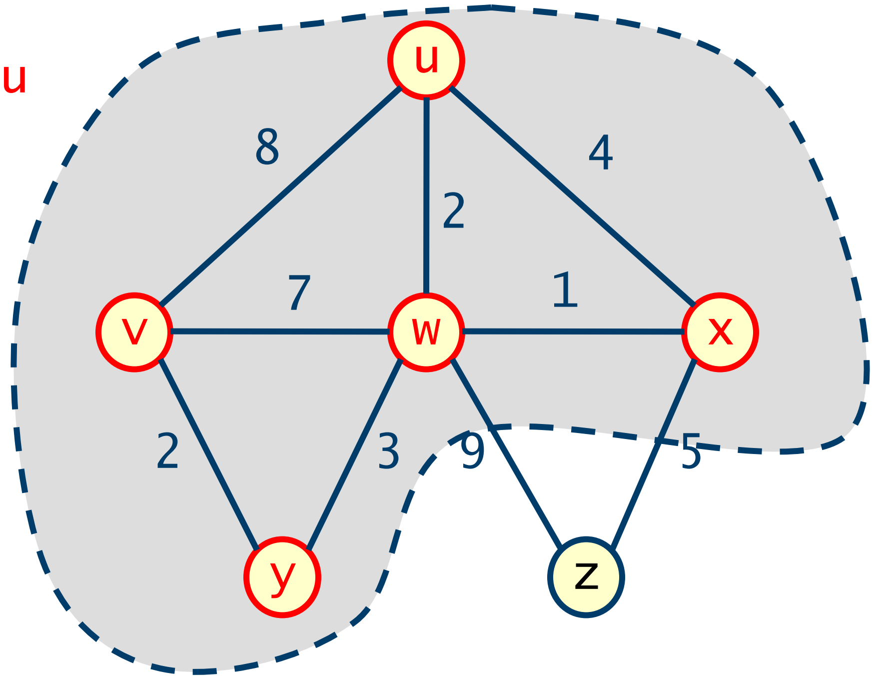
Dijkstra's algorithm – Example

Weighted graph

– compute shortest path with u

S={u,v,w,x,y}

dist(u)=0
dist(v)=7
dist(w)=2
dist(x)=3
dist(y)=5
dist(z)=8

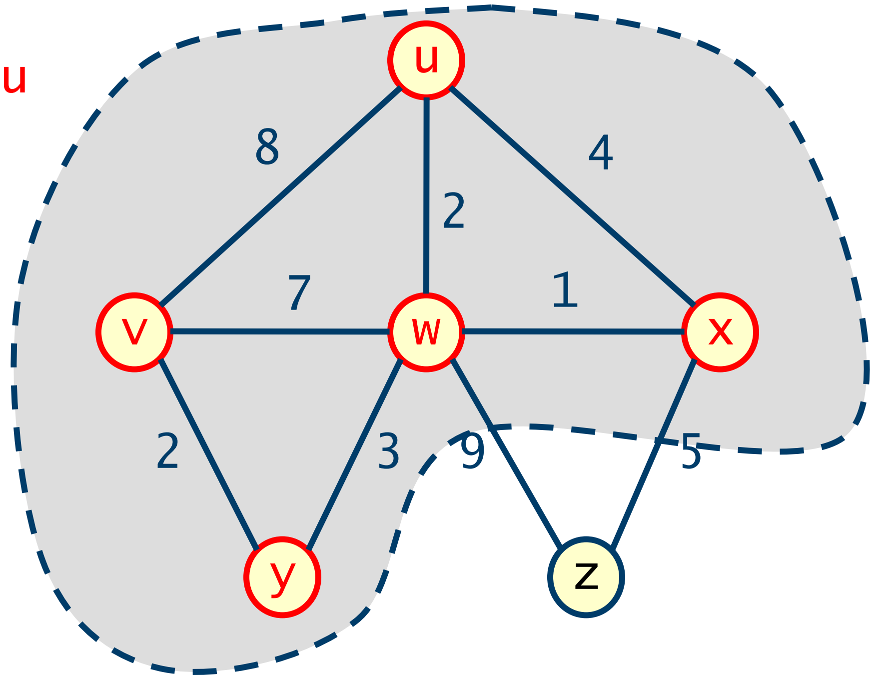


add v to the set S

Dijkstra's algorithm – Example

Weighted graph

– compute shortest path with u



S={u,v,w,x,y}

dist(u)=0
dist(v)=7
dist(w)=2
dist(x)=3
dist(y)=5
dist(z)=8→min{8,7+∞}

perform relaxation

dist(vertex)=**min**{dist(vertex),dist(**v**)+wt(vertex,**v**)};

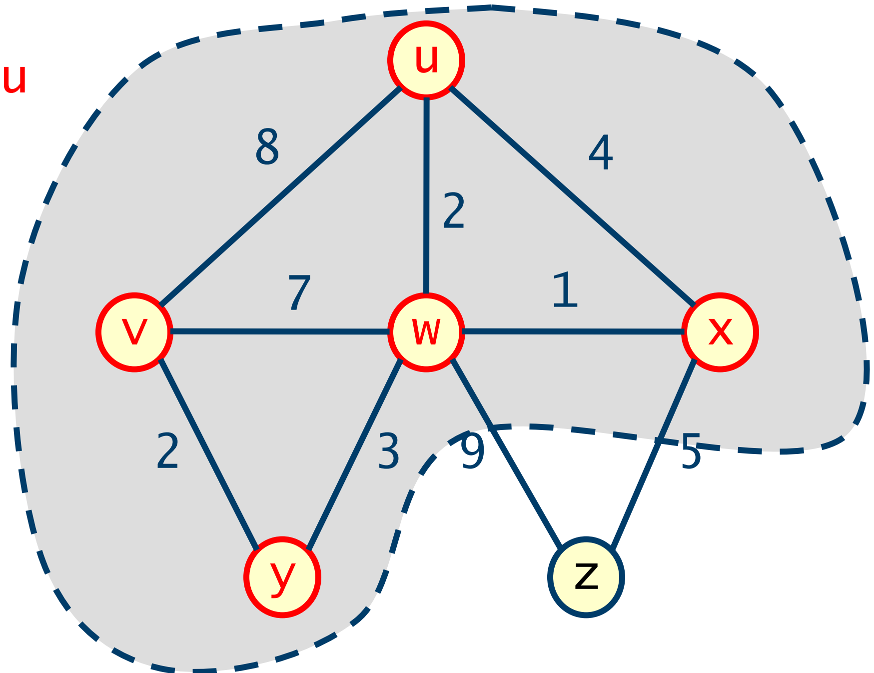
Dijkstra’s algorithm – Example

Weighted graph

– compute shortest path with u

S={u,v,w,x,y}

dist(u)=0
dist(v)=7
dist(w)=2
dist(x)=3
dist(y)=5
dist(z)=8 ← minimum distance



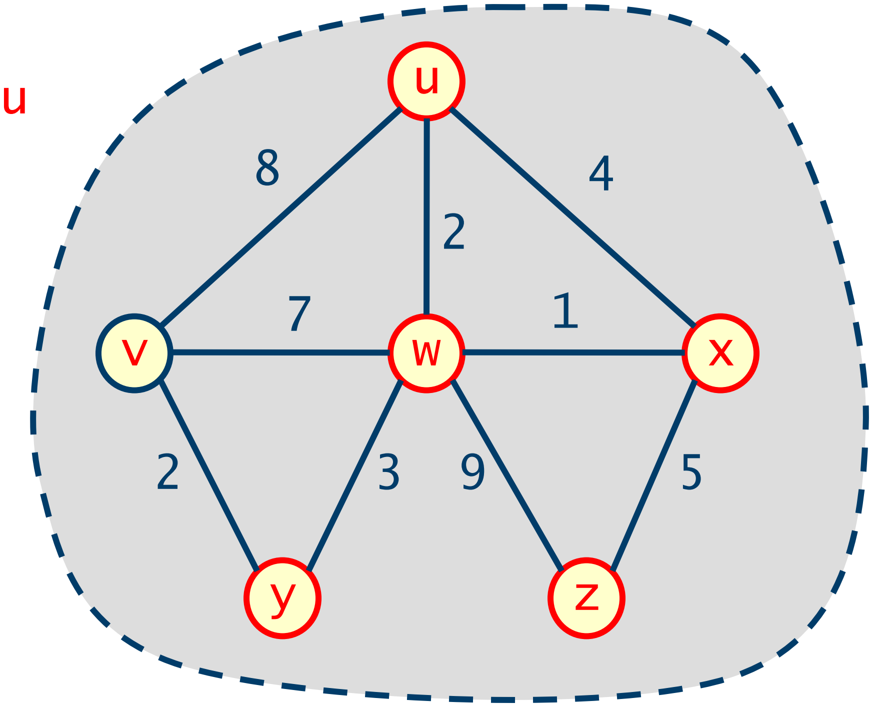
Dijkstra's algorithm – Example

Weighted graph

– compute shortest path with u

S={u,v,w,x,y,z}

dist(u)=0
dist(v)=7
dist(w)=2
dist(x)=3
dist(y)=5
dist(z)=8



add z to the set S

Dijkstra's algorithm

Algorithm finds shortest path between one vertex u and all others

- based on maintaining a set S containing all vertices for which shortest path with u is currently known
- S initially contains only u (obviously shortest path between u and u is 0)
- eventually S contains all the vertices (so all shortest paths are known)

Each vertex v not in S has a label `dist(v)` indicating the length of a shortest path between u and v passing only through vertices in S

Dijkstra’s algorithm

Each vertex **v** not in **S** has a label **`dist(v)`** indicating the length of a shortest path between **u** and **v** passing **only** through vertices in **S**

– if no path exists then we set to `dist(v)` infinity

Dijkstra’s algorithm

Each vertex v not in S has a label `dist(v)` indicating the length of a shortest path between u and v passing only through vertices in S

- at each step we add to S the vertex v not in S such that `dist(v)` is minimum
- this ensures that at any point the shortest path for vertices not in S to u have distance greater than or equal to that for all vertices in S

Dijkstra's algorithm

Each vertex v not in S has a label `dist(v)` indicating the length of a shortest path between u and v passing only through vertices in S

- at each step we add to S the vertex v not in S such that `dist(v)` is minimum
- after having added a vertex v to S, carry out edge relaxation operations i.e. we update the distance `dist(w)` for all vertices w still not in S
  - `dist(w)` is the length of a shortest path between u and v passing only through vertices in S
  - and S has changed since we have added vertex v to S

Dijkstra's algorithm

Each vertex v not in S has a label `dist(v)` indicating the length of a shortest path between u and v passing only through vertices in S

- at each step we add to S the vertex v not in S such that `dist(v)` is minimum
- after having added a vertex v to S, carry out edge relaxation operations i.e. we update the distance `dist(w)` for all vertices w still not in S

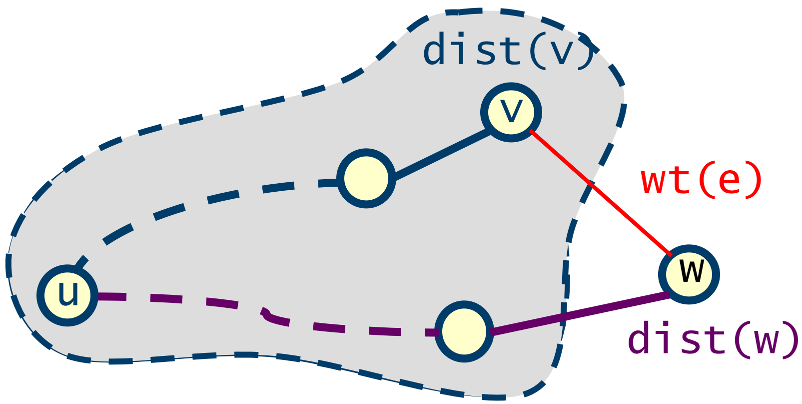
Version described finds shortest path lengths only, not the paths

- tutorial question concerns generating paths

Dijkstra's algorithm – Edge relaxation

Each vertex v not in S has a label dist(v) indicating the length of a shortest path between u and v passing only through vertices in S

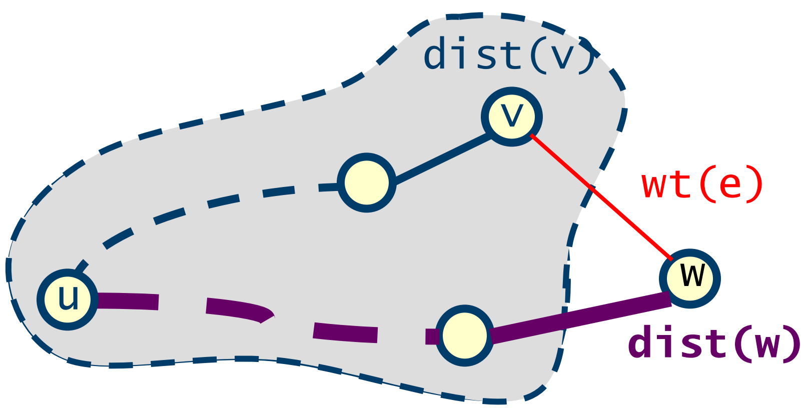
- suppose v and w are not in S then we know
- the shortest path between u and v passing only through S equals dist(v)
- the shortest path between u and w passing only through S equals dist(w)
- now suppose v is added to S and the edge e = {v,w} has weight wt(e)
- calculate the shortest path between u and w passing only through S∪{v}



Dijkstra's algorithm – Edge relaxation

Each vertex v not in S has a label `dist(v)` indicating the length of a shortest path between u and v passing only through vertices in S

- suppose v and w are not in S then we know
- the shortest path between u and v passing only through S equals `dist(v)`
- the shortest path between u and w passing only through S equals `dist(w)`
- now suppose v is added to S and the edge `e = {v,w}` has weight `wt(e)`
- calculate the shortest path between u and w passing only through S∪{v}



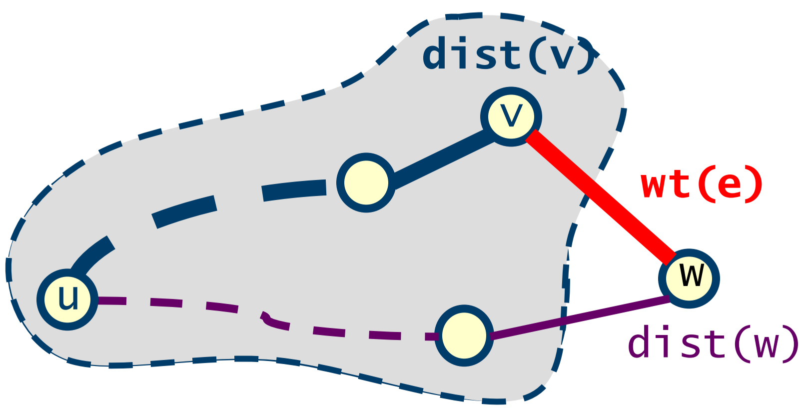
shortest path is either:

- original path through S of length `dist(w)`

Dijkstra's algorithm – Edge relaxation

Each vertex v not in S has a label dist(v) indicating the length of a shortest path between u and v passing only through vertices in S

- suppose v and w are not in S then we know
- the shortest path between u and v passing only through S equals dist(v)
- the shortest path between u and w passing only through S equals dist(w)
- now suppose v is added to S and the edge e = {v,w} has weight wt(e)
- calculate the shortest path between u and w passing only through S∪{v}



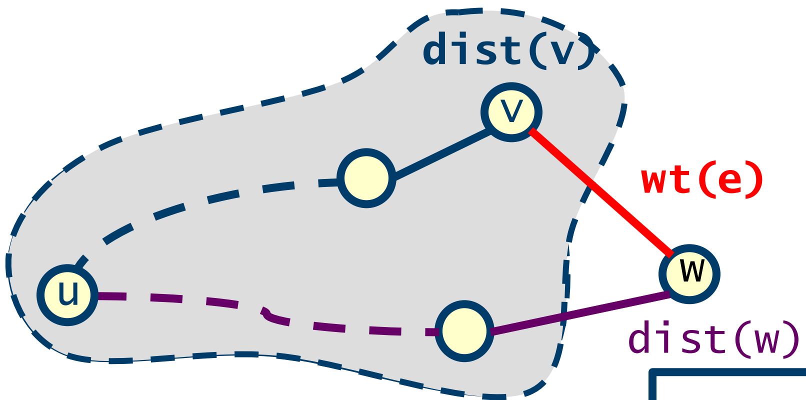
shortest path is either:

- original path through S of length dist(w)
- path combining edge e and shortest path between v and u which has length wt(e) + dist(v)

Dijkstra's algorithm – Edge relaxation

Each vertex v not in S has a label dist(v) indicating the length of a shortest path between u and v passing only through vertices in S

- suppose v and w are not in S then we know
- the shortest path between u and v passing only through S equals `dist(v)`
- the shortest path between u and w passing only through S equals `dist(w)`
- now suppose v is added to S and the edge `e = {v,w}` has weight `wt(e)`
- calculate the shortest path between u and w passing only through S∪{v}



shortest path is either:

- original path through S of length `dist(w)`
- path combining edge e and shortest path between v and u which has length `wt(e) + dist(v)`

therefore distance updated to:

`dist(w) = min{dist(w), dist(v) + wt(e)}`

Dijkstra’s algorithm – Pseudo code

```
// S is set of vertices for which shortest path with u is known
// dist(w) represents length of a shortest path between u and w
// passing only through vertices of S

S = {u}; // initialise S
for (each vertex w) dist(w) = wt(u,w); // initialise distances

while (S != V){ // still vertices to add to S
  find v not in S with dist(v) minimum;
  add v to S;
  for (each w not in S and adjacent to v) // perform relaxation
    dist(w) = min{ dist(w), dist(v)+wt(v,w) };
}
```

Dijkstra's algorithm – Complexity

```
S = {u}; // initialise S
for (each vertex w) dist(w) = wt(u,w); // initialise distances

while (S != V){ // still vertices to add to S
  find v not in S with dist(v) minimum;
  add v to S;
  for (each w not in S and adjacent to v) // perform relaxation
    dist(w) = min{ dist(w), dist(v)+wt(v,w) };
}
```

Consider two ways of implementing distances dist(v)

- unordered array
- heap

Dijkstra complexity – unordered array

```
S = {u}; // initialise S
 for (each vertex w) dist(w) = wt(u,w); // initialise distances

 while (S != V){ // still vertices to add to S
   find v not in S with dist(v) minimum;
   add v to S;
   for (each w not in S and adjacent to v) // perform relaxation
     dist(w) = min{ dist(w) , dist(v)+wt(v,w) };
 }
```

Analysis (n vertices and m edges) using unordered array for distances

- O(n) to initialise distances
- finding minimum is $O(n^2)$ overall
  - each time it takes O(n) and there are n-1 to find
- relaxation is O(m) overall
  - each edge is considered once and updating distance takes O(1)
  - note: we are not considering each iteration of the while loop but overall ops

Dijkstra complexity – unordered array

```
S = {u}; // initialise S
 for (each vertex w) dist(w) = wt(u,w); // initialise distances

 while (S != V){ // still vertices to add to S
   find v not in S with dist(v) minimum;
   add v to S;
   for (each w not in S and adjacent to v) // perform relaxation
     dist(w) = min{ dist(w) , dist(v)+wt(v,w) };
 }
```

Analysis (n vertices and m edges) using unordered array for distances

- O(n) to initialise distances
- finding minimum is $O(n^2)$ overall
  - each time it takes O(n) and there are n-1 to find
- relaxation is O(m) overall
  - each edge is considered once and updating distance takes O(1)

hence $O(n^2)$ overall (number of edges at most n(n-1))

Dijkstra complexity – heap

```
S = {u}; // initialise S
 for (each vertex w) dist(w) = wt(u,w); // initialise distances

 while (S != V){ // still vertices to add to S
   find v not in S with dist(v) minimum;
   add v to S;
   for (each w not in S and adjacent to v) // perform relaxation
     dist(w) = min{ dist(w) , dist(v)+wt(v,w) };
 }
```

Analysis (n vertices and m edges) using a heap for distances

- `O(n)` to initialise distances and create heap
- finding minimum is `O(n log n)` overall
  - each time it takes `O(log n)` and there are `n-1` to find
- relaxation is `O(m log n)` overall
  - each edge is considered once and updating distance takes `O(log n)`
  - note: this involves updating a specific value in the heap not the root
    so care must be taken (need to keep track of positions of vertices in the heap)

Dijkstra complexity – heap

```
S = {u}; // initialise S
 for (each vertex w) dist(w) = wt(u,w); // initialise distances

 while (S != V){ // still vertices to add to S
   find v not in S with dist(v) minimum;
   add v to S;
   for (each w not in S and adjacent to v) // perform relaxation
     dist(w) = min{ dist(w) , dist(v)+wt(v,w) };
 }
```

Analysis (n vertices and m edges) using a heap for distances

- O(n) to initialise distances and create heap
- finding minimum is O(n log n) overall
    - each time it takes O(log n) and there are n-1 to find
- relaxation is O(m log n) overall
    - each edge is considered once and updating distance takes O(log n)

hence O(m log n) overall (more edges than vertices)

- a graph with n vertices has $O(n^2)$ edges

Next lecture

Graph basics

- definitions: directed, undirected, connected, bipartite, ...

Graph representations

- adjacency matrix/lists and implementation

Graph search and traversal algorithms

- depth/breadth first search

Topological ordering

Weighted graphs

- shortest path (Dijkstra's algorithm)
- minimum spanning tree (Prim–Jarnik and Dijkstra's refinement)