Föreläsning 21
Directed and weighted graphs
TDDD86: DALP

Utskriftsversion av Föreläsing i *Datastrukturer, algoritmer och programmeringsparadigm*
03 December 2024

IDA, Linköpings universitet

21.1

Content

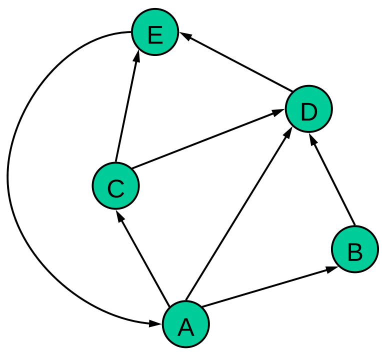
Contents

21.2

1 Directed graphs

Introduction

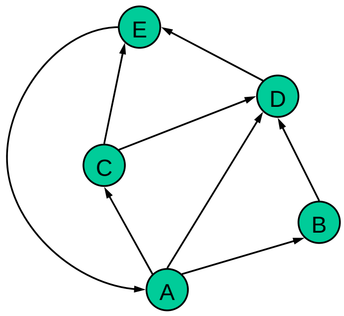
- In a directed graph, all edges are directed



21.3

Properties

- A graph $G = (V, E)$ s.t. each edge as a direction:
  - With edge (a, b) you can go from a to b but not from b to a.
- If G is simple (no parallel edges or loops) then $m \leq n \cdot (n-1)$, hence $m \in O(n^2)$.



21.4

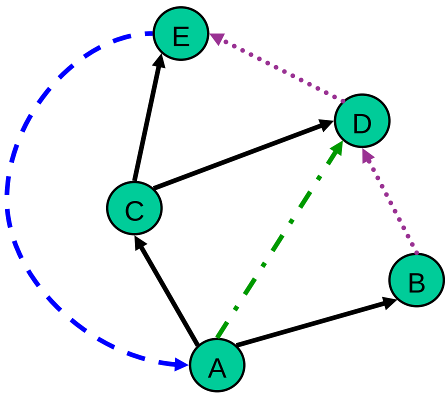
Some algorithmic graph problems

- Path. Is their a directed path from s to t?
- Shorted path. what is the shorted directed path from s to t?
- Strong connectivity. Is there a directed path between all pairs of nodes?
- Topological sort. Is it possible to draw the graph such that all the edges point in the same direction?
- Transitive closure. For which nodes v and w there is at least one path from v to w?
- Page Rank. How important is a web page?

21.5

Directed DFS

- We can specialize the DFS and BFS graph traversing algorithms to directed graphs
- In the directed DFS algorithm there are 4 kinds of edges
  - ”discovery”-edges
  - back-edges
  - forward-edges
  - crossing edges
- A directed DFS from node s lists the nodes that are reachable from s

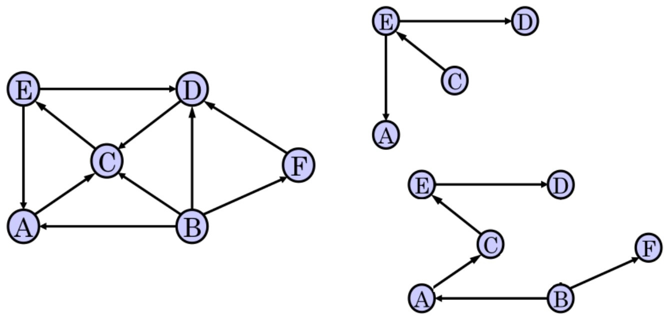


21.6

2 Connectivity

Reachability

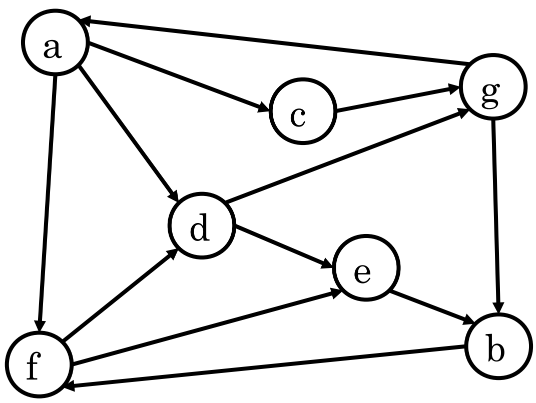
- DFS-tree rooted in v: nodes reachable from v via directed paths
- Not all nodes are reachable from node C, but all nodes are reachable from node B



21.7

Strongly connected graphs

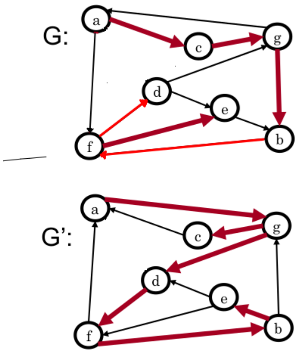
Each node is reachable from each other node



21.8

Algorithm to decide whether a graph is strongly connected

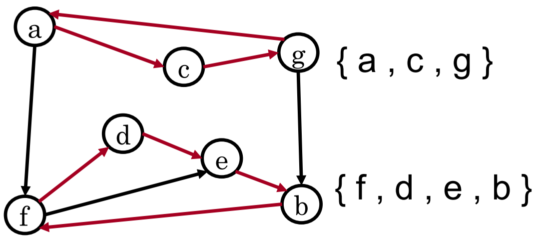
- Choose a node v in G
- *// Can we reach all nodes from v?* Perform DFS from v in G
  - If a node w remains unvisited, answer "no"
- Obtain G' from G by reversing all edges
- *// Can we reach all nodes from v in G'?* Perform DFS v in G'
  - If a node w remains unvisited, answer "no"
  - Otherwise answer "yes"
- Execution time: $O(n+m)$



21.9

Strongly connected components

- A maximal sub-graph where each node is reachable from each other node in the sub-graph
- Can also be obtained in $O(n+m)$ time complexity by using DFS in several steps
- The DFS based Kosaraju's algorithm:
  - Call DFS and enumerate vertices in post-order
  - Call DFS on transposed graph (i.e., reversed edges)

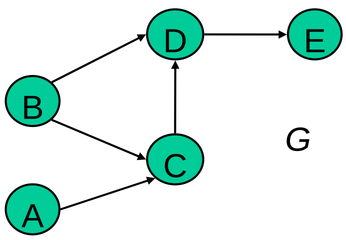


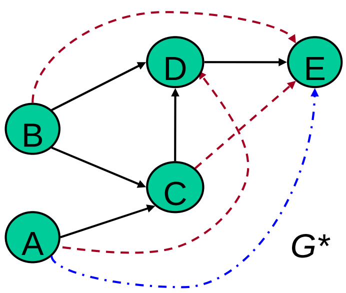
21.10

3 Transitive closure

Transitive closure

- Given a directed graph G, the transitive closure of G is the directed graph G^* where
  - G^* has the same nodes as G
  - if G has a directed path from u to v ($u \neq v$) then G^* has a directed edge fro mu to v
- The transitive closure make explicit reachability in a directed graph





21.11

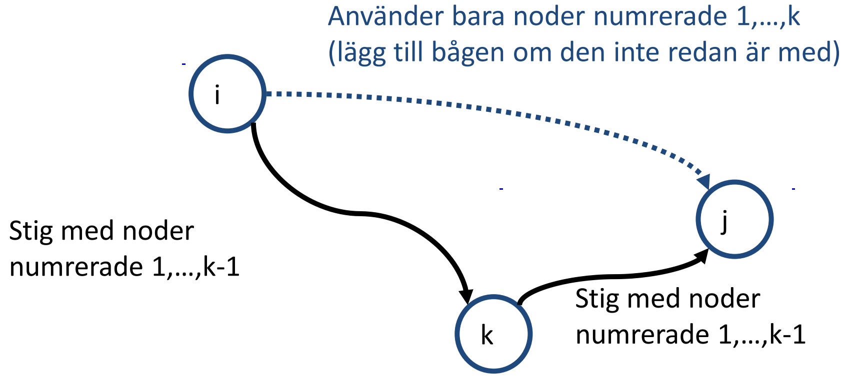
Computing the transitive closure

- We could execute DFS from each node $v_1, \ldots, v_n$, hence $O(n \cdot (n+m))$
- A **dynamic programming** alternative: Floyd-Warshalls algorithm

21.12

Transitive closure with Floyd-Warshall

- Identify the nodes with $1, 2, \ldots, n$.
- In phase k, only consider paths that use nodes in $1, 2, \ldots, k$ as intermediary nodes:



21.13

Floyd-Warshall algorithm

- The Floyd-Warshall algorithm enumerates the nodes in G as $v_1, \ldots, v_n$ and computes the series of directed graphs $G_0, \ldots, G_n$
  - $G_0 = G$
  - G_k has a directed edge (v_i, v_j) if G has a directed path from v_i to v_j with intermediary nodes in the set $\{v_1, \ldots, v_k\}$
- We get $G_n = G^*$
- At iteration k the graph G_k is computed from G_{k-1}
- Execution time: $O(n^3)$ if areAdjacent is $O(1)$

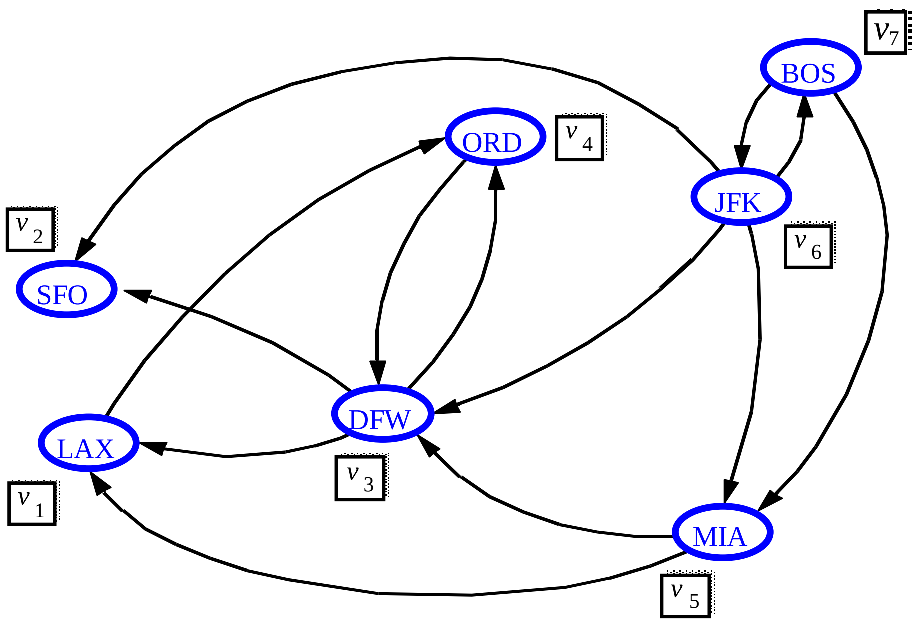
21.14

The Floyd-Warshall algorithm

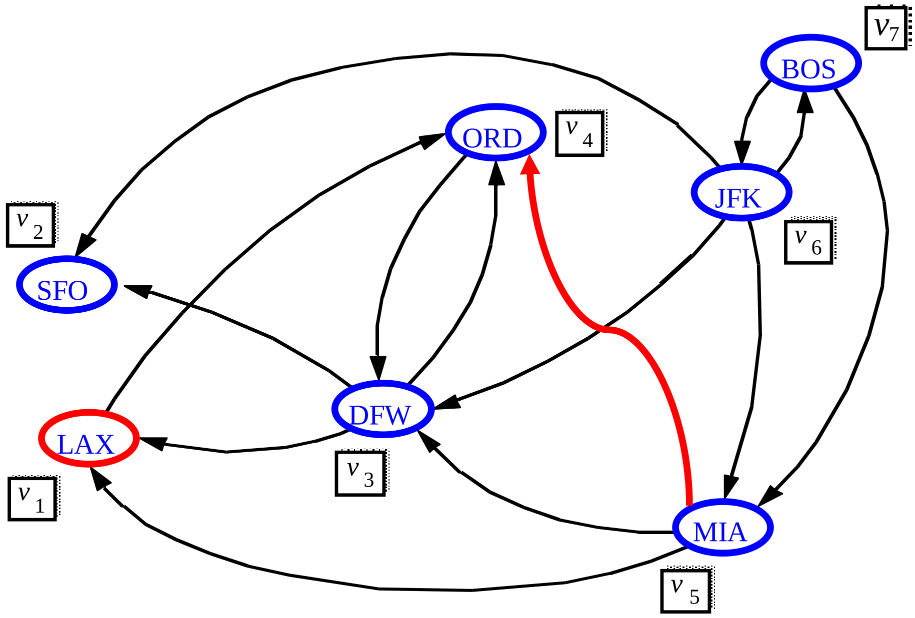
function FLOYDWARSHALL(G)
    $G_0 \leftarrow G$
    for $k \leftarrow 1$ **to** n **do**
        $G_k \leftarrow G_{k-1}$
        for $i \leftarrow 1$ **to** n $(i \neq k)$ **do**
            for $j \leftarrow 1$ **to** n $(j \neq i, k)$ **do**
                if G_{k-1}.AREADJACENT(v_i, v_k) **then**
                    if G_{k-1}.AREADJACENT(v_k, v_j) **then**
                        if $\neg G_k$.AREADJACENT(v_i, v_j) **then**
                            G_k.INSERTDIRECTEDEDGE(v_i, v_j, k)
    return G_n

21.15

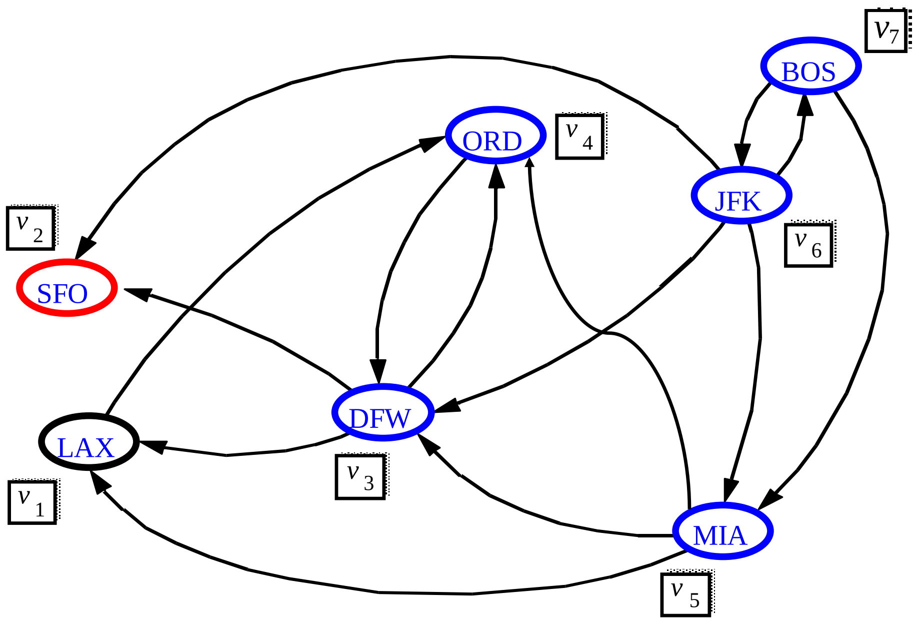
Example: Floyd-Warshall



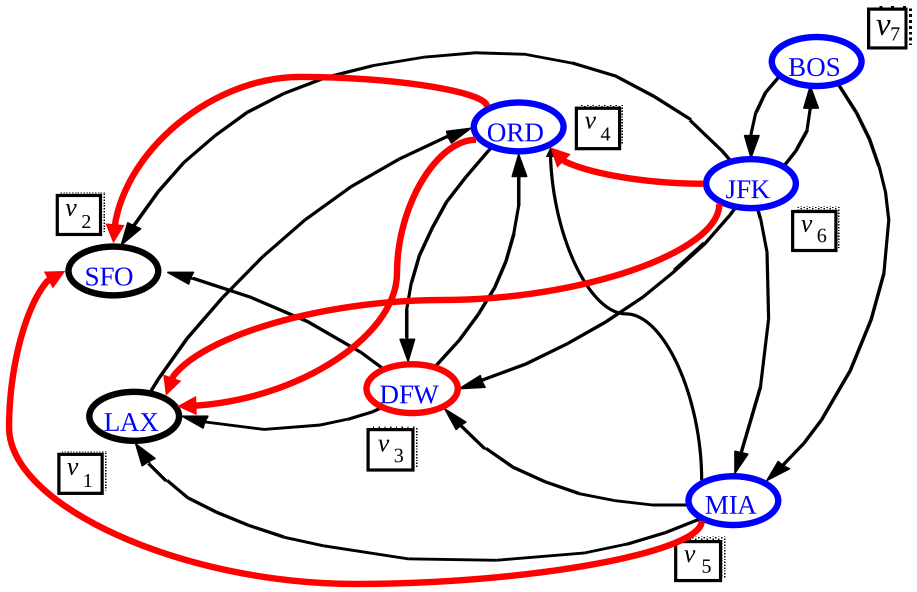
Floyd-Warshall, iteration 1



Floyd-Warshall, iteration 2

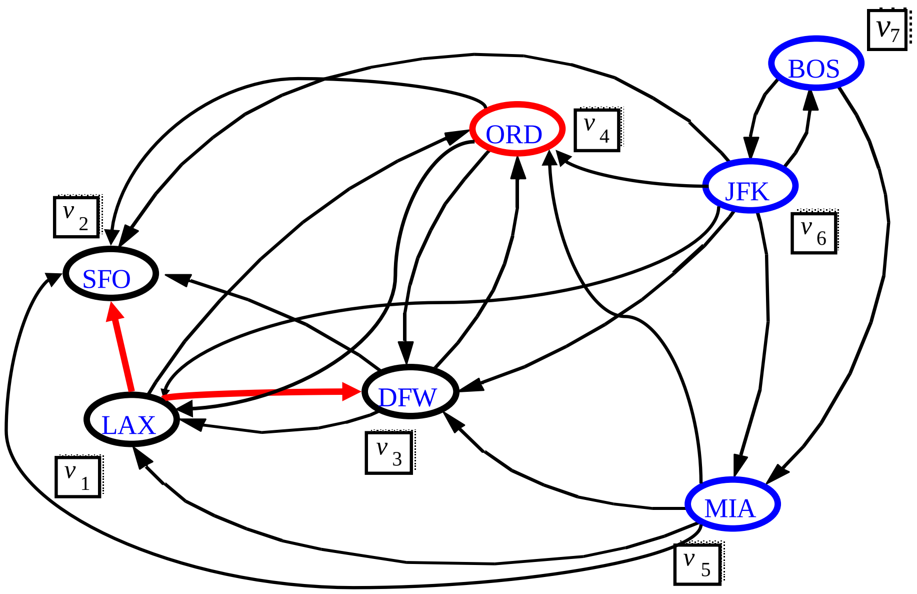


Floyd-Warshall, iteration 3



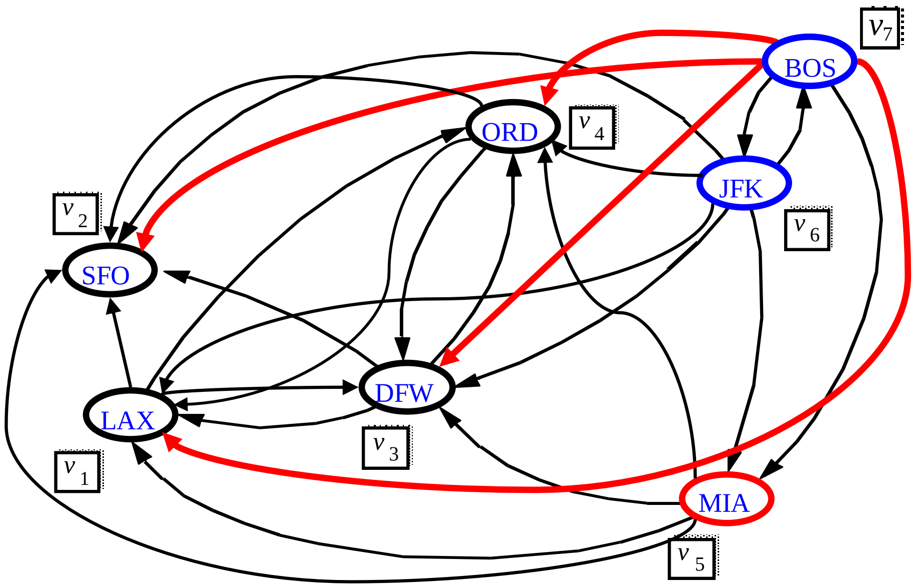
21.19

Floyd-Warshall, iteration 4



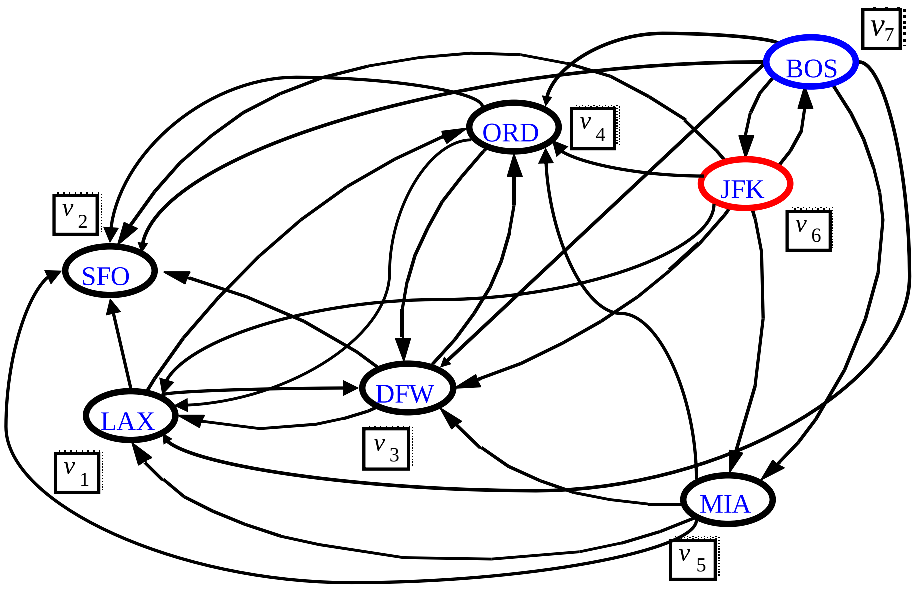
21.20

Floyd-Warshall, iteration 5



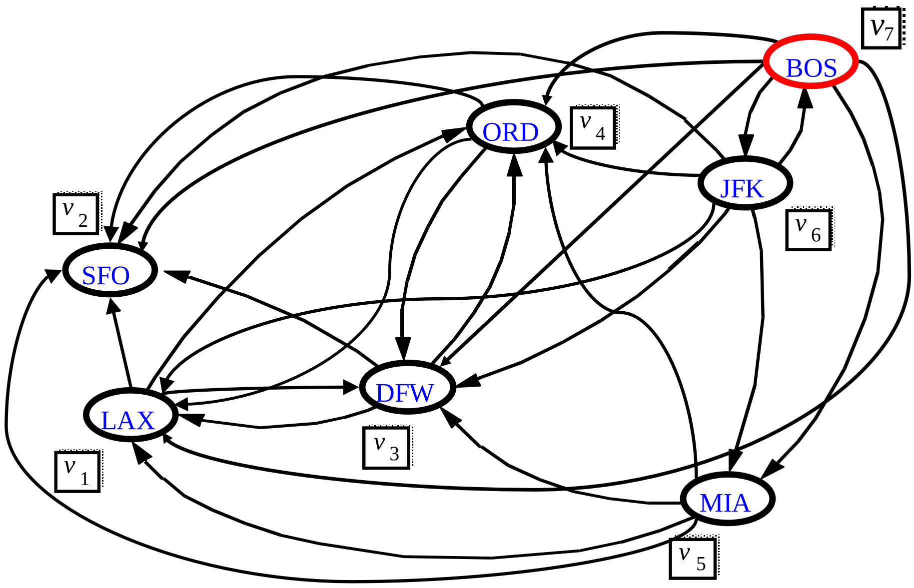
21.21

Floyd-Warshall, iteration 6



21.22

Floyd-Warshall, end



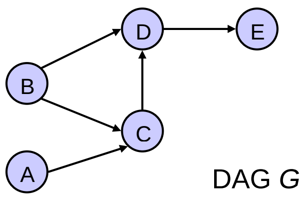
21.23

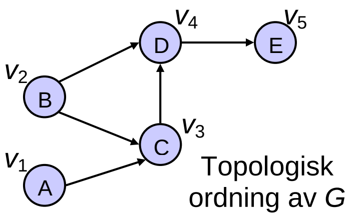
4 Topological sorting

Directed acyclic graphs and topological sorting

- A directed acyclic graph (DAG) is a directed graph that does not have any directed cycle
- A topological sorting of a graph is a total ordering $v_1, \ldots, v_n$ of the nodes such that each edge (v_i, v_j) satisfies $i < j$
- Example: Existence of a plan for tasks that depend on each other.

Proposition 1. A graph can be topologically sorted iff it is a DAG





21.24

An algorithm for topological sorting

procedure TOPOLOGICALSORT(G)
    $H \leftarrow G$ ▷ temporary copy of G
    $n \leftarrow G$.NUMVERTICES
    while H is non-empty **do**
        let v be a node without outgoing edges
        mark v with n
        $n \leftarrow n - 1$
        remove v from H

Execution time: $O(n+m)$. How...?

21.25

Algorithm for topological sorting via DFS

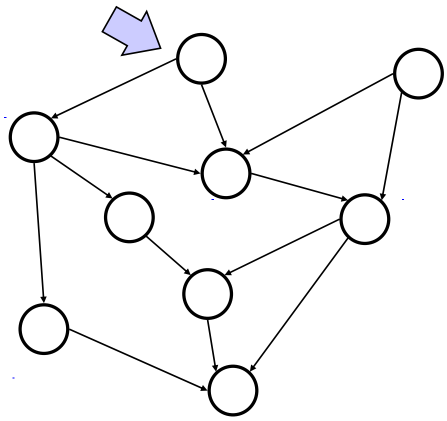
procedure TOPOLOGICALDFS(G)
    $n \leftarrow G$.NUMVERTICES
    mark all nodes and edges as *UNEXPLORED* like in DFS
    for all $v \in G$.VERTICES() **do**

if $\text{GETLABEL}(v) = UNEXPLORED$ **then**
    $\text{TOPOLOGICALDFS}(G, v)$

procedure $\text{TOPOLOGICALDFS}(G, v)$
  $\text{SETLABEL}(v, VISITED)$
  for all $e \in G.\text{INCIDENTEDGES}(v)$ **do**
    if $\text{GETLABEL}(e) = UNEXPLORED$ **then**
      $w \leftarrow \text{OPPOSITE}(v, e)$
      if $\text{GETLABEL}(w) = UNEXPLORED$ **then**
        $\text{SETLABEL}(e, DISCOVERY)$
        $\text{TOPOLOGICALDFS}(G, w)$
      else
        e is a crossing edge or a forward edge
  mark v with the topological number n
  $n \leftarrow n - 1$

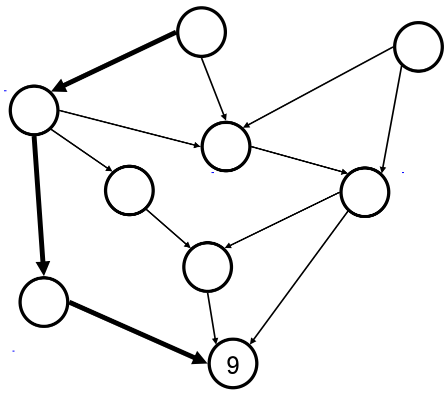
21.26

Example: Topological sorting



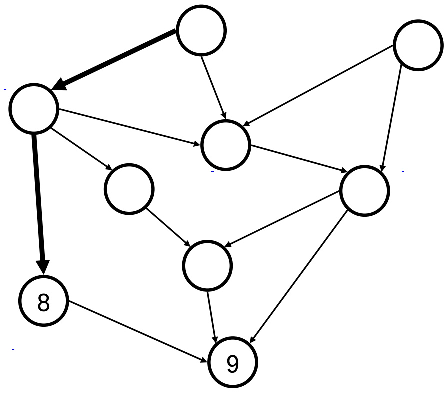
21.27

Example: Topological sorting

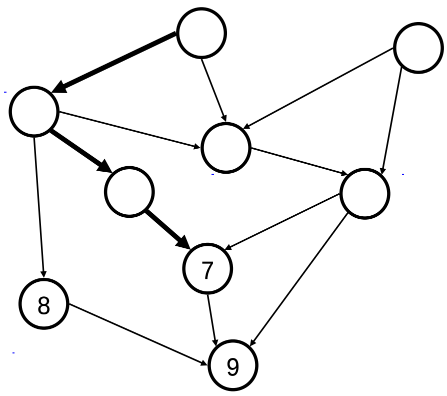


21.28

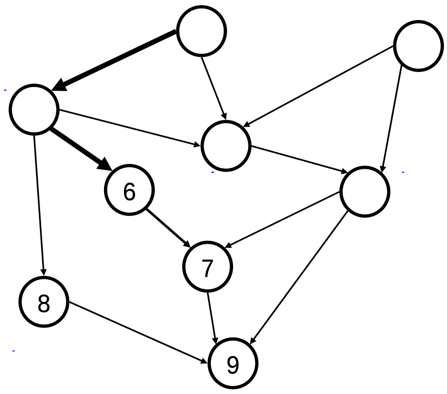
Example: Topological sorting



Example: Topological sorting

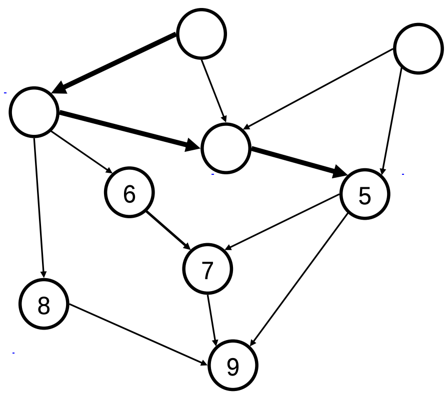


Example: Topological sorting



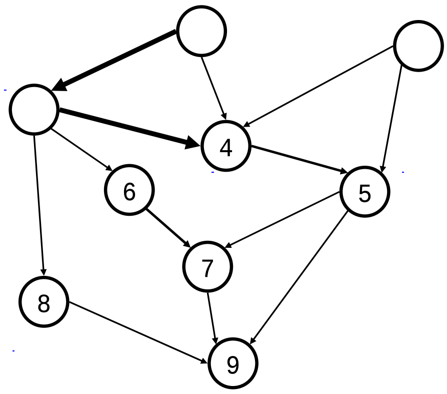
21.31

Example: Topological sorting



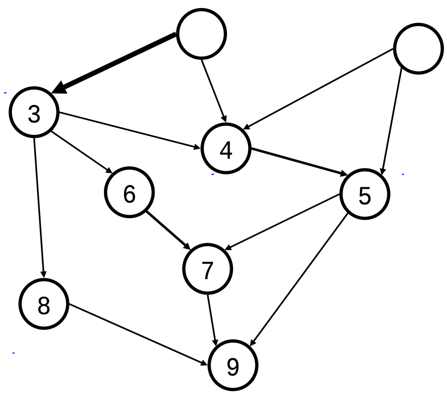
21.32

Example: Topological sorting



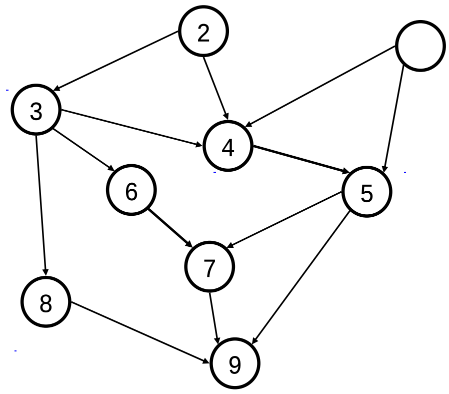
21.33

Example: Topological sorting



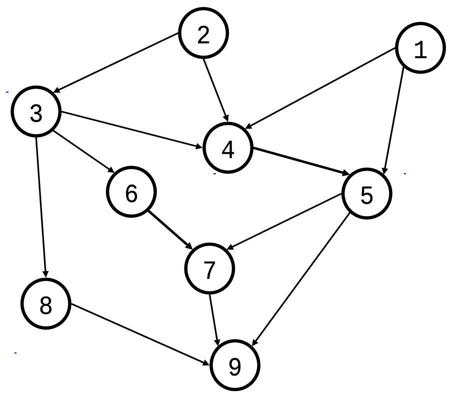
21.34

Example: Topological sorting



21.35

Example: Topological sorting



21.36

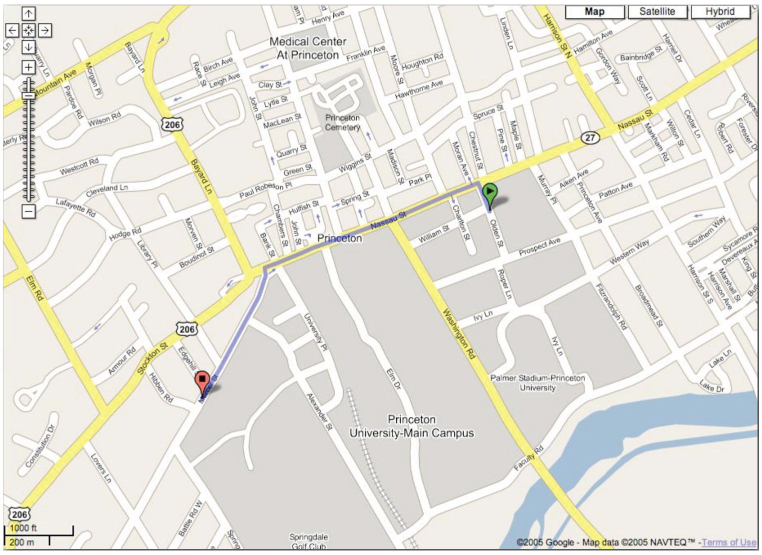
5 Weighted graphs

Weighted graphs

- In a weighted graph, each edge is associated a numerical *weight*.
- Weights can represent distances, costs, etc.

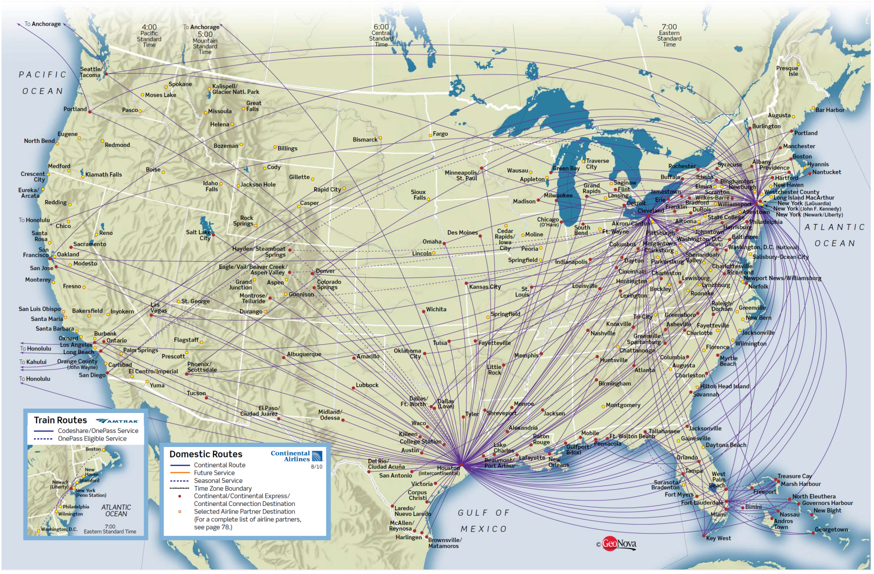
21.37

Google maps



21.38

Continentals fly routes in USA (august 2010)



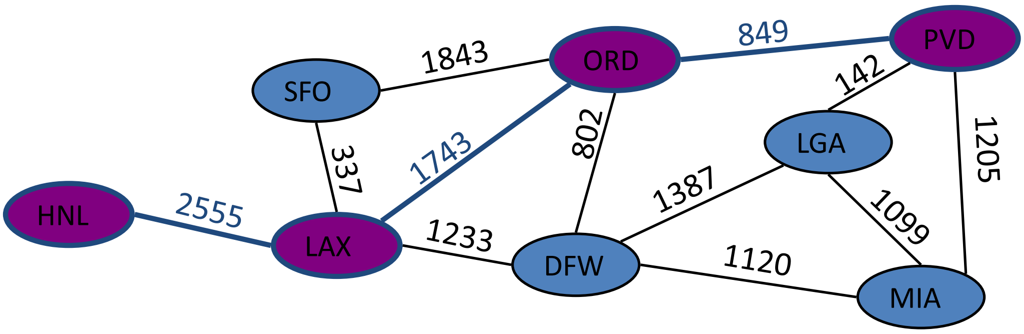
21.39

6 Shortest paths

The shortest path problem

- Given a weighted graph and two nodes u and v, find a path between u and v with minimal total weight.
  - Length of a path is the sum of the weights of its edges

Example
Shortest path between Providence and Honolulu

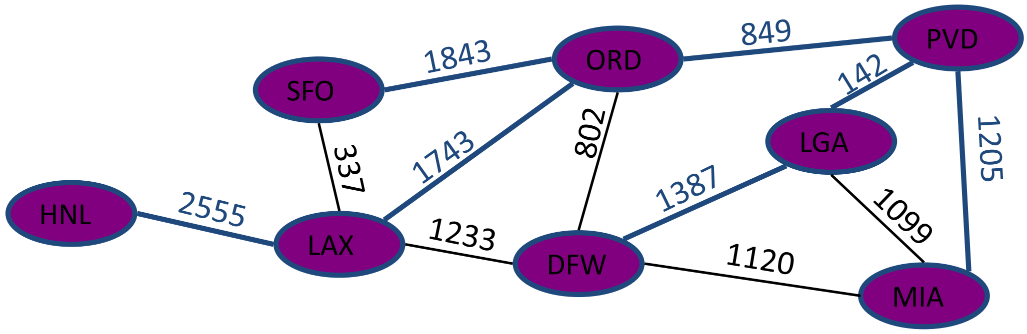


21.40

Properties of shortest paths

- A sub-path of a shortest path is also a shortest path
- There is a tree of shortest paths from a start node to all other nodes

Example
A tree of shortest paths from Providence



21.41

Weighted Floyd-Warshalls algorithm

function WEIGHTEDFLOYDWARSHALL($G(N,E,w)$)
  for $i \leftarrow 1$ **to** N **do**
    for $j \leftarrow 1$ **to** N **do**
      $dist(i,j) \leftarrow w(i,j)$
  for $i \leftarrow 1$ **to** N **do**
    $dist(i,i) \leftarrow 0$
  for $k \leftarrow 1$ **to** N **do**
    for $i \leftarrow 1$ **to** N **do**
      for $j \leftarrow 1$ **to** N **do**
        if $\left(\texttt{dist}(v_i,v_j) > \texttt{dist}(v_i,v_k)+\texttt{dist}(v_k,v_j)\right)$ **then**
          $\texttt{dist}(v_i,v_j) \leftarrow \texttt{dist}(v_i,v_k)+\texttt{dist}(v_k,v_j)$
  return `dist`

21.42

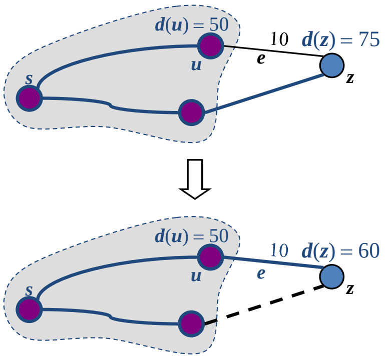
Dijkstra's algorithm

- Distance from a node v to a node s is the length of a shortest path between s and v
- Dijkstra's algorithm computes the distance from a given start node s to all other nodes v in the graph
- Assumptions:
  - the graph is connected
  - graph has no loops or parallel edges
  - weights are *non-negative*
- We build a "cloud" of nodes, starting from s, that will cover all nodes
- We mark each node v in the cloud or neighbor to it with $d(v)$, which represents the distance between v and s
- At each step:
  - Extend the cloud to the node u that was outside the cloud and which has the minimal distance $d(u)$
  - update distances of nodes that are neighbor to u

21.43

Extension step

- Consider an edge $e = (u,z)$ s.t.:
  - u has just been added to the cloud
  - z is not part of the cloud yet
- Edge e updates $d(z)$ with :
  - $d(z) \leftarrow \min\{d(z), d(u) + weight(e)\}$



21.44

Dijkstra pseudocode

function **dijkstra**(v_1, v_2):
    initialize every vertex to have a cost of infinity.
    set v_1's cost to 0.
    pqueue := {v_1, with priority 0}. // ordered by cost

    while *pqueue* is not empty:
        v := dequeue vertex from *pqueue* with minimum priority.
        mark *v* as visited.
        if *v* is v_2, we can stop.
        for each unvisited neighbor *n* of *v*:
            cost := *v*'s cost + weight of edge (*v*, *n*).
            if *cost* < *n*'s cost:
                set *n*'s cost to *cost*, and *n*'s previous to *v*.
                enqueue *n* in the *pqueue* with priority of *cost*,
                or update its priority if it was already in the *pqueue*.

    reconstruct path from v_2 back to v_1, following previous pointers.

21.45

Example

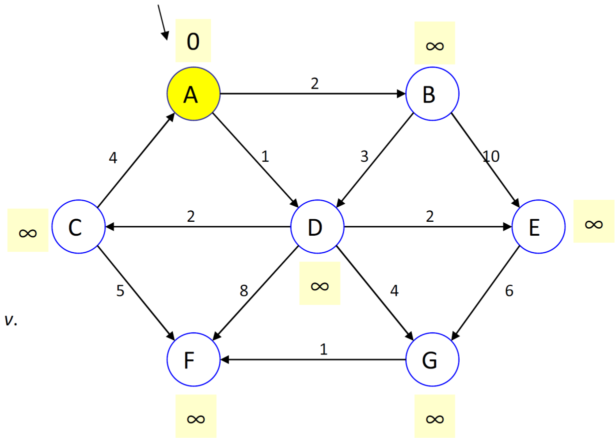
- dijkstra(A, F);

function **dijkstra**(v_1, v_2):
    v_1's cost := 0.
    pqueue := {v_1}.   // ordered by cost

    while *pqueue* is not empty:
        v := dequeue min cost from *pqueue*.
        mark *v* as visited.
        if *v* is v_2, we can stop.
        for each unvisited neighbor *n* of *v*:
            cost := *v*'s cost + weight of edge (*v*, *n*).
            if *cost* < *n*'s cost:
                set *n*'s cost to *cost* and *n*'s previous to *v*.
                enqueue or update *n* in the *pqueue*.

    reconstruct path from v_2 back to v_1,
    following previous pointers.

v_1's avstånd := 0.   alla andra avstånd := ∞.



pqueue = {**A:0**}

- I våra diagram färglägger vi en nod:
  - **vit** om den är outforskad
  - gul om den köats för senare behandling
  - **grön** om den besökts (plockats ut ur kön) och behandlats

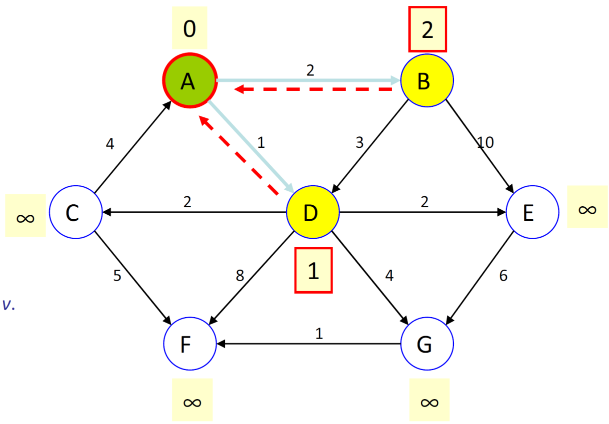
21.46

Example

- dijkstra(A, F);

```
function dijkstra(v1, v2):
   v1's cost := 0.
   pqueue := {v1}.   // ordered by cost

   while pqueue is not empty:
      v := dequeue min cost from pqueue.  // A
      mark v as visited.
      if v is v2, we can stop.
      for each unvisited neighbor n of v:  // B, D
         cost := v's cost + weight of edge (v, n).
         if cost < n's cost:
            set n's cost to cost and n's previous to v.
            enqueue or update n in the pqueue.
            // B's cost = 0+2,  D's cost = 0+1
   reconstruct path from v2 back to v1,
    following previous pointers.
```



pqueue = {**D:1**, **B:2**}

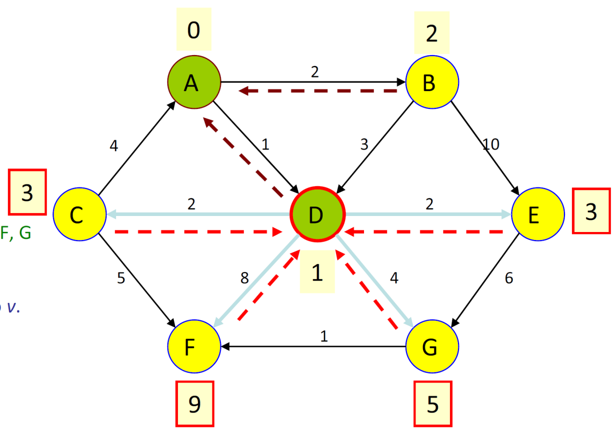
21.47

Example

- dijkstra(A, F);

```
function dijkstra(v1, v2):
   v1's cost := 0.
   pqueue := {v1}.   // ordered by cost

   while pqueue is not empty:
      v := dequeue min cost from pqueue.  // D
      mark v as visited.
      if v is v2, we can stop.
      for each unvisited neighbor n of v:  // C, E, F, G
         cost := v's cost + weight of edge (v, n).
         if cost < n's cost:
            set n's cost to cost and n's previous to v.
            enqueue or update n in the pqueue.
            // C=1+2, E=1+2, F=1+8, G=1+4
   reconstruct path from v2 back to v1,
    following previous pointers.
```



pqueue = {B:2, **C:3**, **E:3**, **G:5**, **F:9**}

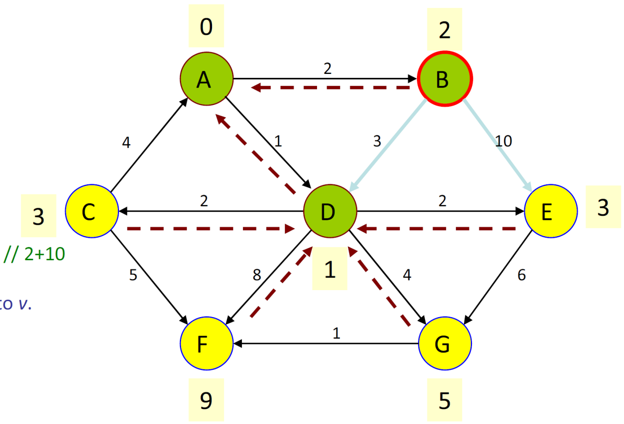
21.48

Example

- dijkstra(A, F);

function **dijkstra**(v_1, v_2):
  v_1's cost := 0.
  pqueue := {v_1}.   // ordered by cost

  while *pqueue* is not empty:
    v := dequeue min cost from *pqueue*. // B
    mark *v* as visited.
    if *v* is v_2, we can stop.
    for each unvisited neighbor *n* of *v*: // E
      cost := *v*'s cost + weight of edge (*v*, *n*). // 2+10
      if *cost* < *n*'s cost:
        set *n*'s cost to *cost* and *n*'s previous to *v*.
        enqueue or update *n* in the *pqueue*.
        // no change
  reconstruct path from v_2 back to v_1,
   following previous pointers.



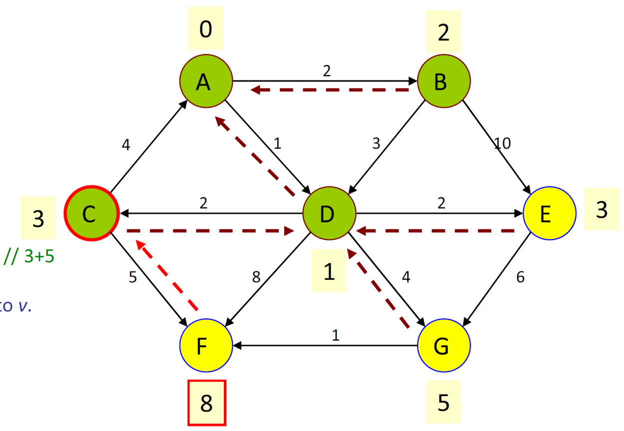
pqueue = {C:3, E:3, G:5, F:9}

Example

- dijkstra(A, F);

function **dijkstra**(v_1, v_2):
  v_1's cost := 0.
  pqueue := {v_1}.   // ordered by cost

  while *pqueue* is not empty:
    v := dequeue min cost from *pqueue*. // C
    mark *v* as visited.
    if *v* is v_2, we can stop.
    for each unvisited neighbor *n* of *v*: // F
      cost := *v*'s cost + weight of edge (*v*, *n*). // 3+5
      if *cost* < *n*'s cost: // 8 < 9
        set *n*'s cost to *cost* and *n*'s previous to *v*.
        enqueue or update *n* in the *pqueue*.
        // F = 8
  reconstruct path from v_2 back to v_1,
   following previous pointers.



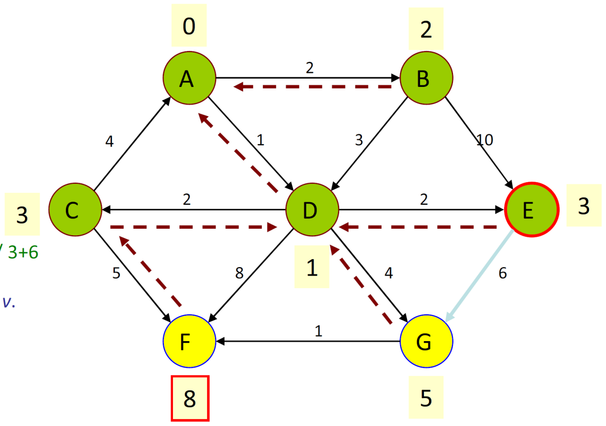
pqueue = {E:3, G:5, **F:8**}

Example

- dijkstra(A, F);

function **dijkstra**(v_1, v_2):
  v_1's cost := 0.
  pqueue := {v_1}. // ordered by cost

  while *pqueue* is not empty:
    v := dequeue min cost from *pqueue*. // E
    mark *v* as visited.
    if *v* is v_2, we can stop.
    for each unvisited neighbor *n* of *v*: // G
      cost := *v*'s cost + weight of edge (*v*, *n*). // 3+6
      if *cost* < *n*'s cost: // 9 > 5
        set *n*'s cost to *cost* and *n*'s previous to *v*.
        enqueue or update *n* in the *pqueue*.
        // no change
  reconstruct path from v_2 back to v_1,
   following previous pointers.



pqueue = {G:5, F:8}

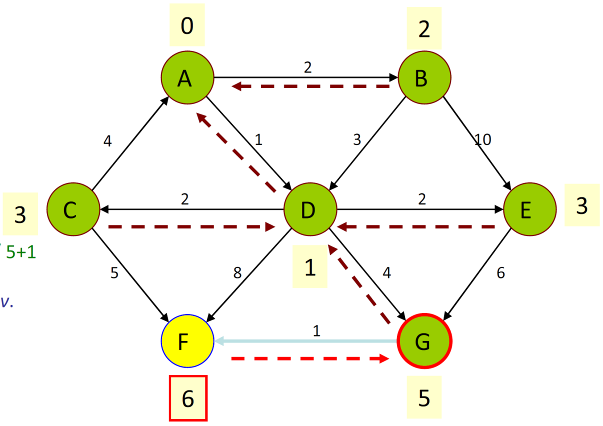
21.51

Example

- dijkstra(A, F);

function **dijkstra**(v_1, v_2):
  v_1's cost := 0.
  pqueue := {v_1}. // ordered by cost

  while *pqueue* is not empty:
    v := dequeue min cost from *pqueue*. // G
    mark *v* as visited.
    if *v* is v_2, we can stop.
    for each unvisited neighbor *n* of *v*: // F
      cost := *v*'s cost + weight of edge (*v*, *n*). // 5+1
      if *cost* < *n*'s cost: // 6 < 8
        set *n*'s cost to *cost* and *n*'s previous to *v*.
        enqueue or update *n* in the *pqueue*.
        // F = 6
  reconstruct path from v_2 back to v_1,
   following previous pointers.



pqueue = {**F:6**}

21.52

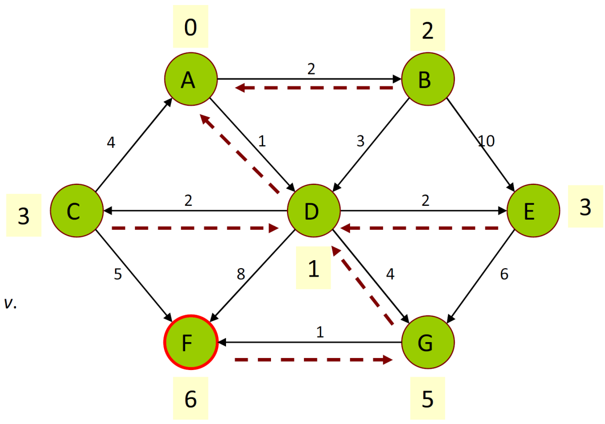
Example

- dijkstra(A, F);

```
function dijkstra(v1, v2):
  v1's cost := 0.
  pqueue := {v1}.   // ordered by cost

  while pqueue is not empty:
    v := dequeue min cost from pqueue.  // F
    mark v as visited.
    if v is v2, we can stop.
    for each unvisited neighbor n of v:
      cost := v's cost + weight of edge (v, n).
      if cost < n's cost:
        set n's cost to cost and n's previous to v.
        enqueue or update n in the pqueue.

  reconstruct path from v2 back to v1,
   following previous pointers.
```



pqueue = {}

21.53

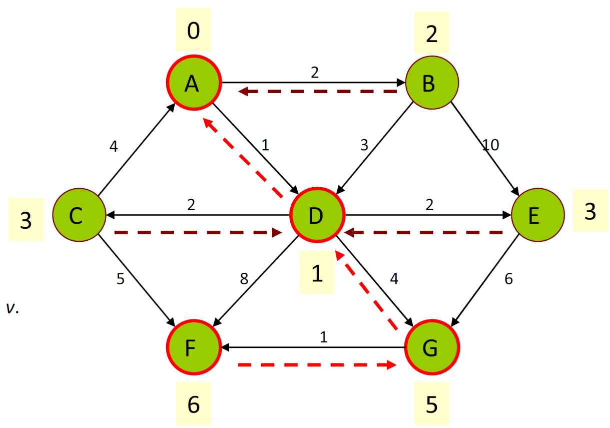
Example

- dijkstra(A, F);

```
function dijkstra(v1, v2):
  v1's cost := 0.
  pqueue := {v1}.   // ordered by cost

  while pqueue is not empty:
    v := dequeue min cost from pqueue.
    mark v as visited.
    if v is v2, we can stop.
    for each unvisited neighbor n of v:
      cost := v's cost + weight of edge (v, n).
      if cost < n's cost:
        set n's cost to cost and n's previous to v.
        enqueue or update n in the pqueue.

  reconstruct path from v2 back to v1,
   following previous pointers.
   // path = {A, D, G, F}
```



21.54

Analysis of Dijkstra algorithm

- incidentEdges is called once for each node
- markings are fetched/updated for node z $O(deg(z))$ times
- to fetch/update a marking takes $O(1)$ time
- Each node is inserted once and removed once from the priority queue, where each insertion and removal takes $O(\log n)$ time
- The key of a node in the priority queue is updated at most $deg(w)$ times, where each update may take at most $O(\log n)$ time

```
function dijkstra(v1, v2):
  initialize every vertex to have a cost of infinity.
  set v1's cost to 0.
  pqueue := {v1, with priority 0}.  // ordered by cost

  while pqueue is not empty:
    v := dequeue vertex from pqueue with minimum priority.
    mark v as visited.
    if v is v2, we can stop.
    for each unvisited neighbor n of v:
      cost := v's cost + weight of edge (v, n).
      if cost < n's cost:
        set n's cost to cost, and n's previous to v.
        enqueue n in the pqueue with priority of cost,
        or update its priority if it was already in the pqueue.

  reconstruct path from v2 back to v1, following previous pointers.
```

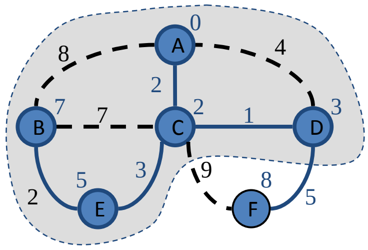
- Dijkstra's algorithm has execution time of $O((n+m)\log n)$ given the graph is represented using adjacent lists
- Execution time can also be expressed as $O(m \log n)$ since we assume it is connected

21.55

Why does it work

Dijkstra's algorithm is a greedy algorithm. It greedily adds nodes in increasing distances to the source.

- Suppose the algorithm does not find all shortest distances. Let F be the first node that got a wrong shortest distance.
- Any node D preceding F along a shortest path must have obtained a correct shortest distance and added to the cloud at some point.
- But then the edge (D,F) must have been *updated* when such a D was added!
- In other words, since $d(F) \geq d(D)$, then the distance to F should have been correct.

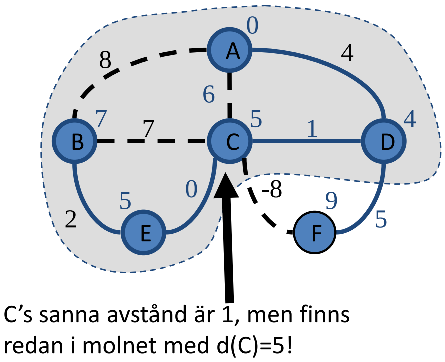


21.56

Why does it require non-negative weights?

Dijkstra's algorithm is a greedy algorithm. It greedily adds nodes in increasing distances to the source.

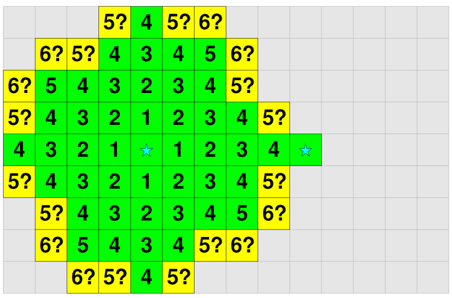
- If a node with a negative incident edge is added later to the cloud, it would jeopardize the distances to nodes that were earlier added.



21.57

Observations

- Dijkstra's algorithm works by incrementally computing shortest path to potential intermediary nodes.
  - Most such paths are in the wrong direction.



- The algorithm explores in all directions;
  - Could we tips the algorithm to first explore more promising directions?

21.58

Heuristics

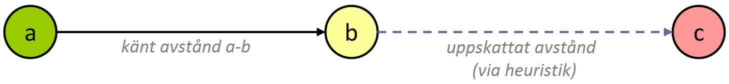
- heuristic: Speculation, estimation or a qualified guess on how the search for a solution should proceed.
  - Example: Estimate the distance between two locations in a map using a direct line.

- for the following algorithm, an admissible heuristic is one that does not over-estimate the distance.
  - Ok if the heuristic under-estimates the distance (as above with the maps).

21.59

A*-algorithm

- A*("A-star): a modified version of Dijkstra's algorithm that uses a heuristic to direct the search.



- Suppose we are looking for a path from source node *a* to target node *c*
  - Each intermediary node *b* has two costs:
  - The known exact cost from *a* to *b*
  - A heuristic based estimation of the cost from *b* to the target node *c*.

- Idea: Execute Dijkstra's algorithm but adopt the following priority in the priority queue:
  - priority(b) = cost(a, b) + Heuristic(b, c)
  - Explore based on the smallest estimated cost

21.60

Example: Labyrinth heuristic

- A possible heuristic to find paths in a labyrinth:
  - $H(p_1, p_2) = abs(p_1.x - p_2.x) + abs(p_1.y - p_2.y)$ // dx + dy
  - Idea: Explore neighbors with low value of (cost + heuristic)

6	5	4	3	4
5	4	3	2	3
4	3	2	1	2
a	2	1	c	1
4	3	2	1	2
5	4	3	2	3

21.61

Recall: pseudo-code for Dijkstra's algorithm

```
function dijkstra(v1, v2):
   initialize every vertex to have a cost of infinity.
   set v1's cost to 0.
   pqueue := {v1, with priority 0}.  // ordered by cost

   while pqueue is not empty:
      v := dequeue vertex from pqueue with minimum priority.
      mark v as visited.
      if v is v2, we can stop.
      for each unvisited neighbor n of v:
         cost := v's cost + weight of edge (v, n).
         if cost < n's cost:
            set n's cost to cost, and n's previous to v.
            enqueue n in the pqueue with priority of cost,
            or update its priority if it was already in the pqueue.

   reconstruct path from v2 back to v1, following previous pointers.
```

21.62

Pseudo-code for the A*-algorithm

```
function astar(v1, v2):
  initialize every vertex to have a cost of infinity.
  set v1's cost to 0.
  pqueue := {v1, at priority H(v1, v2)}.

  while pqueue is not empty:
    v := dequeue vertex from pqueue with minimum priority.
    mark v as visited.
    if v is v2, we can stop.
    for each unvisited neighbor n of v:
      cost := v's cost + weight of edge (v, n).
      if cost < n's cost:
        set n's cost to cost, and n's previous to v.
        enqueue n in the pqueue with priority of (cost + H(n, v2)),
        or update its priority to be (cost + H(n, v2)) if it was already in the pqueue.

  reconstruct path from v2 back to v1, following previous pointers.
```

Observe that only nodes' *priorities* are influenced by the heuristic, not their *costs*. 21.63