Lecture 23
Directed and weighted graphs
TDDD86: DALP

Print version of the lecture *Data structures, algorithms and programming paradigms*
2 december 2016

Jalil Boudjadar, Tommy Färnqvist. IDA, Linköping University

23.1

Content

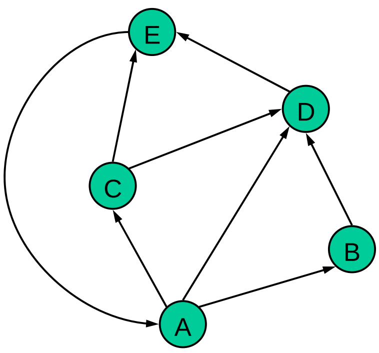
Innehåll

23.2

1 Directed graphs

Introduction

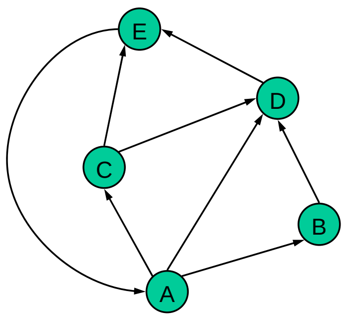
- In a directed graph, all edges are directed



23.3

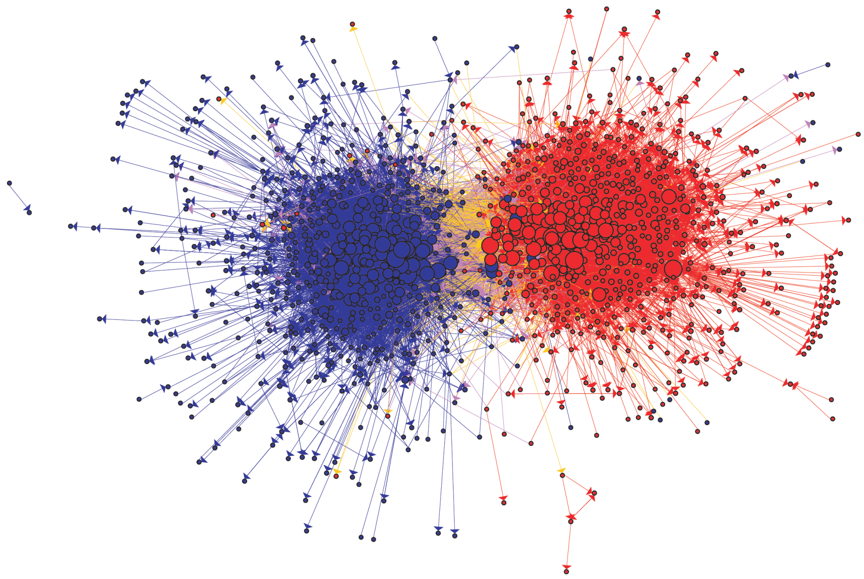
Characteristics

- A graph $G = (V,E)$ where each edge has one direction:
  - Edge (a,b) travels from a to b but not from b to a.
- If G is simple (no parallel edges or loops), then $m \leq n \cdot (n-1)$, i.e. $m \in O(n^2)$, where n is the number of nodes and m is the number of edges.



23.4

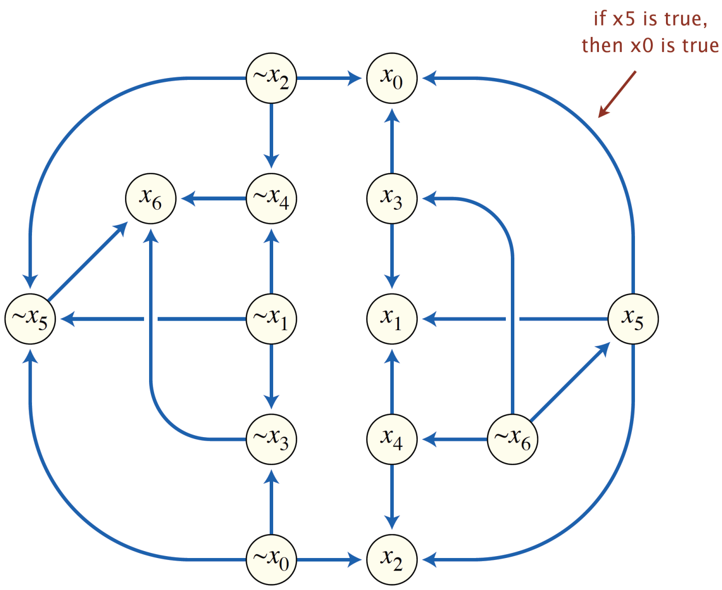
Political Blogosphere-graph



The Political Blogosphere and the 2004 US Election: Divided They Blog, Adamic och Glance, 2005

23.5

Implication graph



23.6

Applications

directed graph	node	directed edge
transport	intersection	one-way street
www	website	hyperlink
food chain	species	predator-prey ratio
financial	bank	transaction
mobile phone	personal	dialed calls
...	...	...

23.7

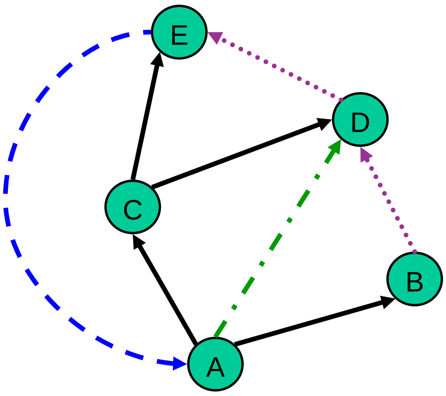
Some algorithmic graph problems

- Path. Is there a directed path from s to t?
- Shortest path. What is the shortest directed path from s to t?
- Strong connectivity. Is there a directed path between all pairs of nodes?
- Topological sorting. Is it possible to draw the directed graph so that all edges pointing upwards?
- Transitive cover. For each nodes v and w, there is a path from v to w?
- Page Rank. How important is a website?

23.8

Directed DFS

- We can adapt traversal algorithms (DFS and BFS) to directed graphs
- In the directed DFS algorithm, we get four types of edges
  - ”discovery”-edges
  - backward-edges
  - forward-edges
  - intersecting edges
- A directed DFS starting in node p determines which nodes are reachable from the s

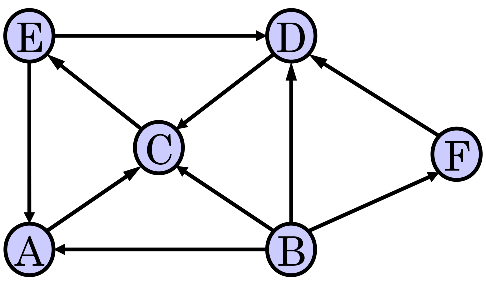


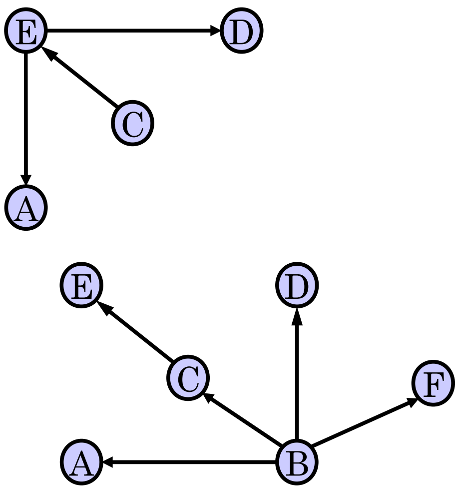
23.9

2 Connectivity

Reachability

DFS tree rooted at v: nodes reachable from v via directed paths

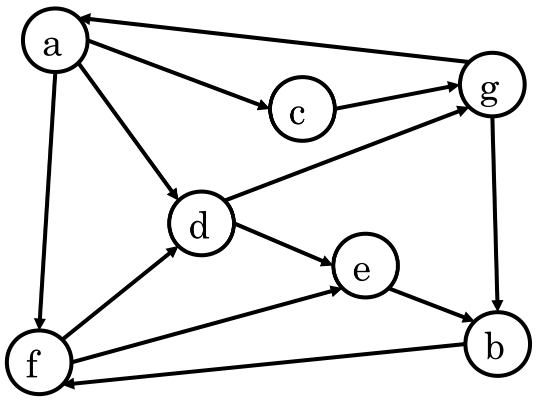




23.10

Strong connection

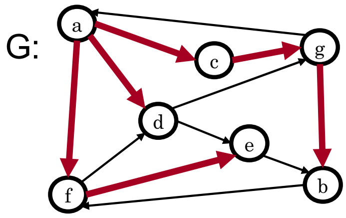
Each node is reachable from all other nodes

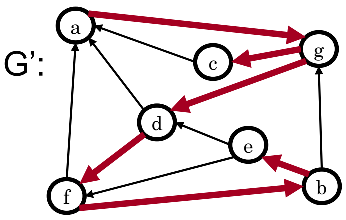


23.11

Algorithm to determine strong connections

- Choose a node v in G
- *// Can all nodes be reached from v?* Perform DFS from v in G
  - If there is w which is not frequented, answer "no"
- Let G' be G with the direction of each arc reversed
- *// Can v be reached from all nodes?* Run DFS from v in G'
  - If there is w which is not frequented, answer "no"
  - Otherwise, answer "yes"
- Execution time: $O(n+m)$

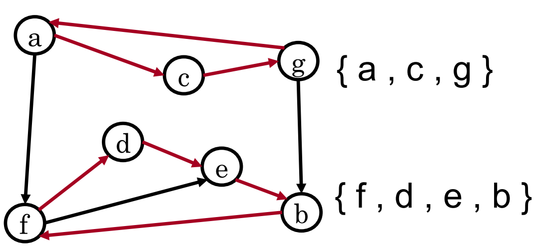




23.12

Strongly connected components

- Maximum subgraph such that each node can reach all the other nodes in the subgraph
- Can also be performed in $O(n+m)$ time by using DFS in several stages

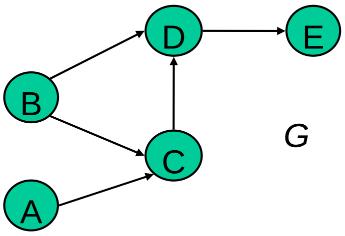


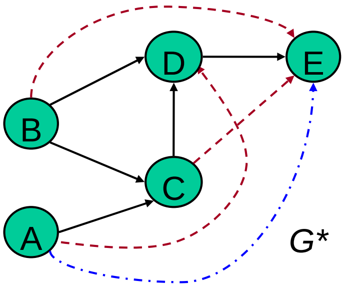
23.13

3 Transitive coverage

Transitive coverage

- Given a directed graph G, let the transitive coverage of G be a directed graph G^* such that
  - G^* has the same nodes as G
  - if G has a directed path u to v ($u \neq v$), so G^* has a directed edge from u to v
- The transitive coverage gives information about the reachability in a directed graph.





23.14

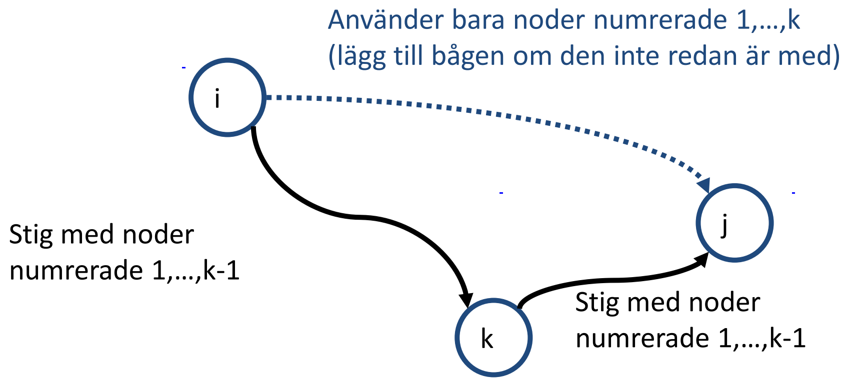
Calculation of transitive coverage

- We can run DFS with a start from each node $v_1, \ldots, v_n$, thus $O(n \cdot (n+m))$
- Alternatively, through the use of dynamic programming: Floyd-Warshall's algorithm

23.15

Transitive coverage with Floyd-Warshall

- Number the nodes $1, 2, \ldots, n$.
- In phase k, consider only paths that use the nodes with numbers $1, 2, \ldots, k$ as internediate nodes:



23.16

Floyd-Warshall algorithm

- Floyd-Warshall algorithm numbers nodes in G as $v_1, \ldots, v_n$ and calculates a serie 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 intermediate nodes from the set $\{v_1, \ldots, v_k\}$
- We see that $G_n = G^*$
- In phase k, the calculated graph G_k is outgoing from G_{k-1}
- Run time: $O(n^3)$ if areAdjacent becomes $O(1)$

23.17

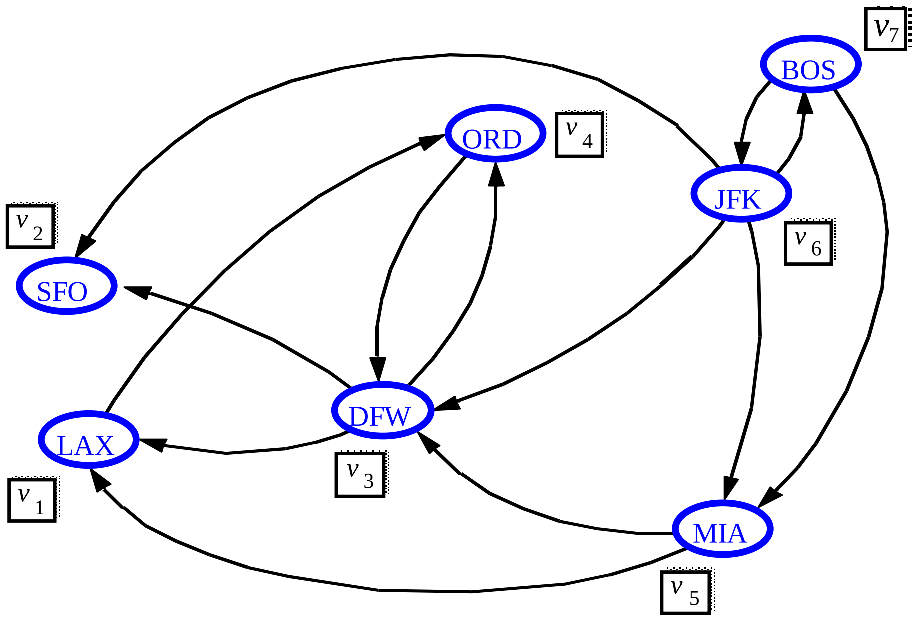
Floyd-Warshall algorithm

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

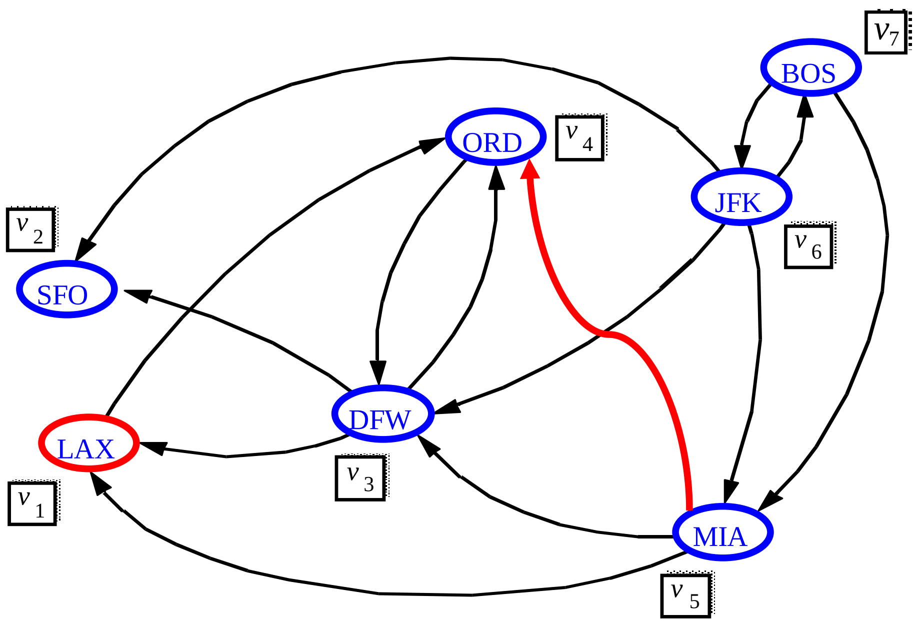
23.18

Example: Floyd-Warshall



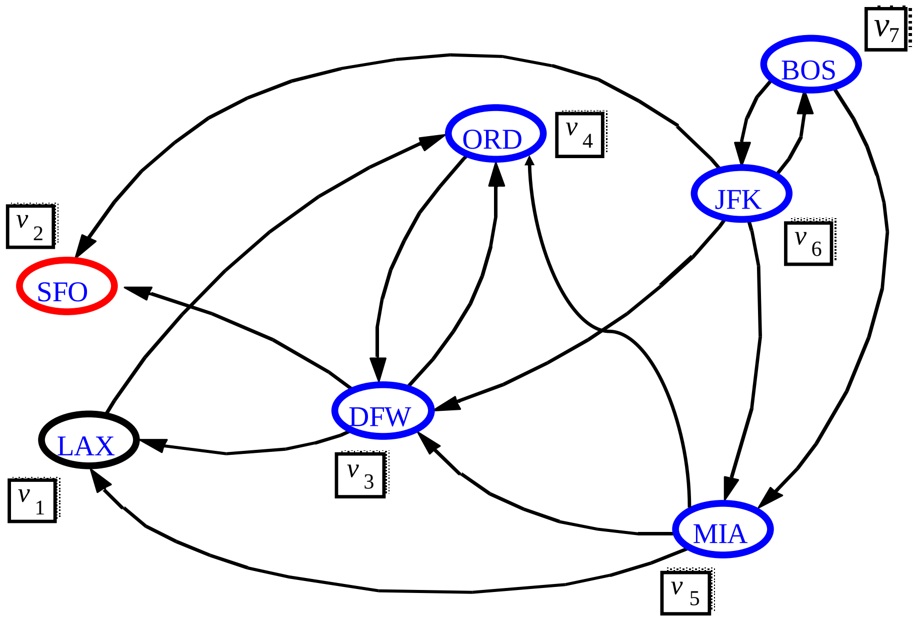
23.19

Floyd-Warshall, iteration 1



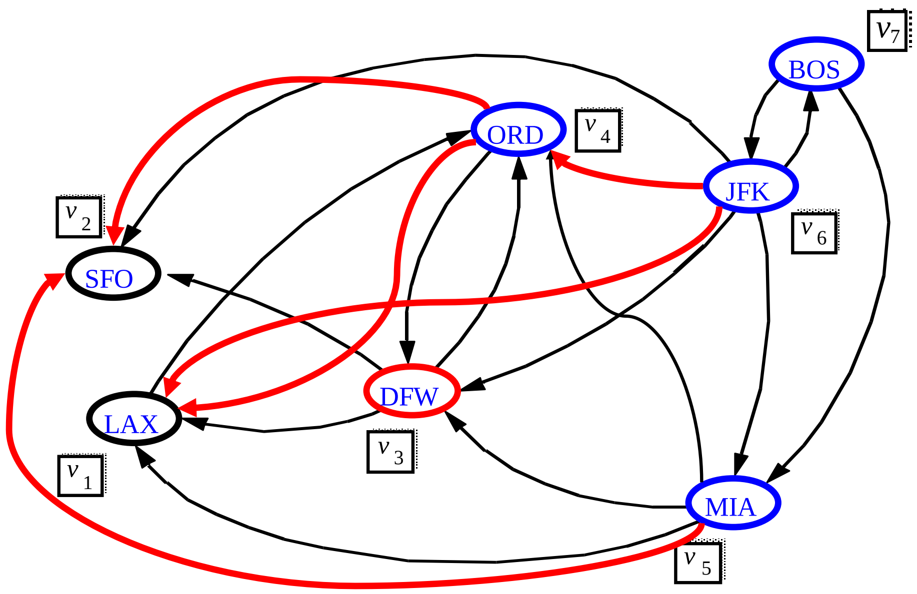
23.20

Floyd-Warshall, iteration 2



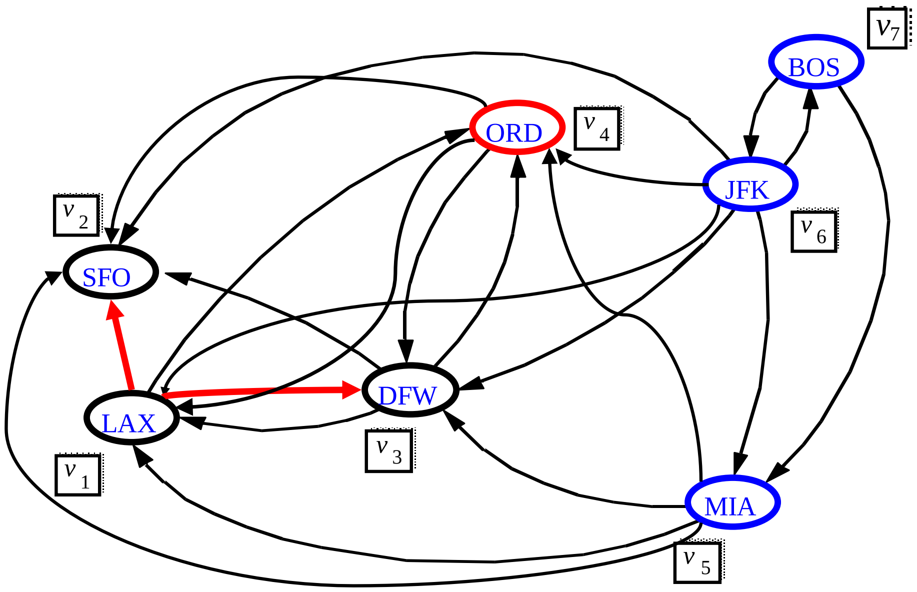
23.21

Floyd-Warshall, iteration 3

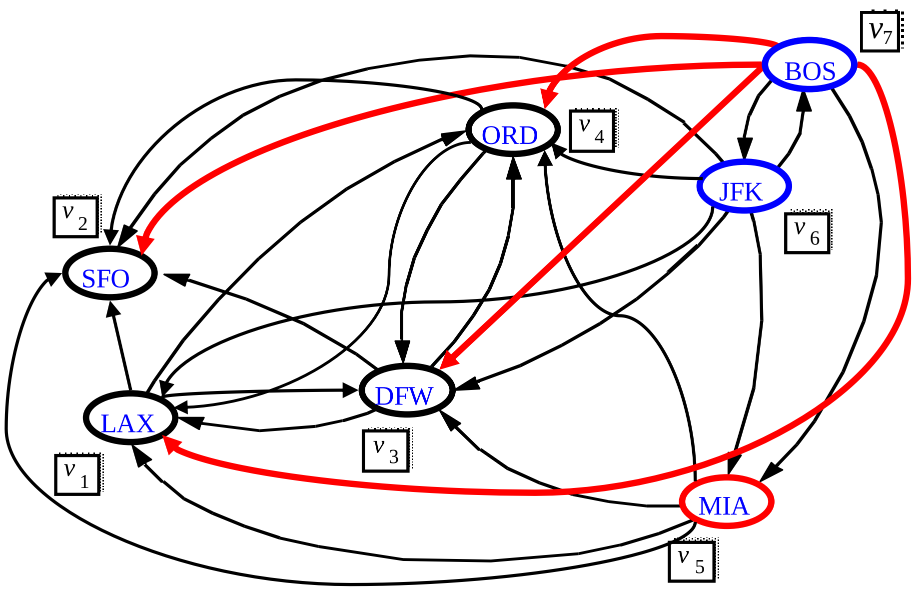


23.22

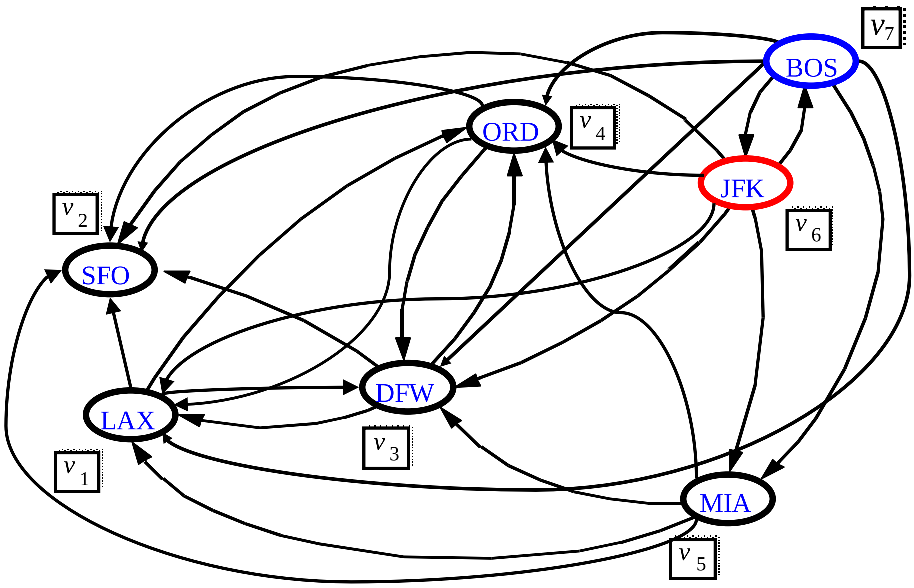
Floyd-Warshall, iteration 4



Floyd-Warshall, iteration 5

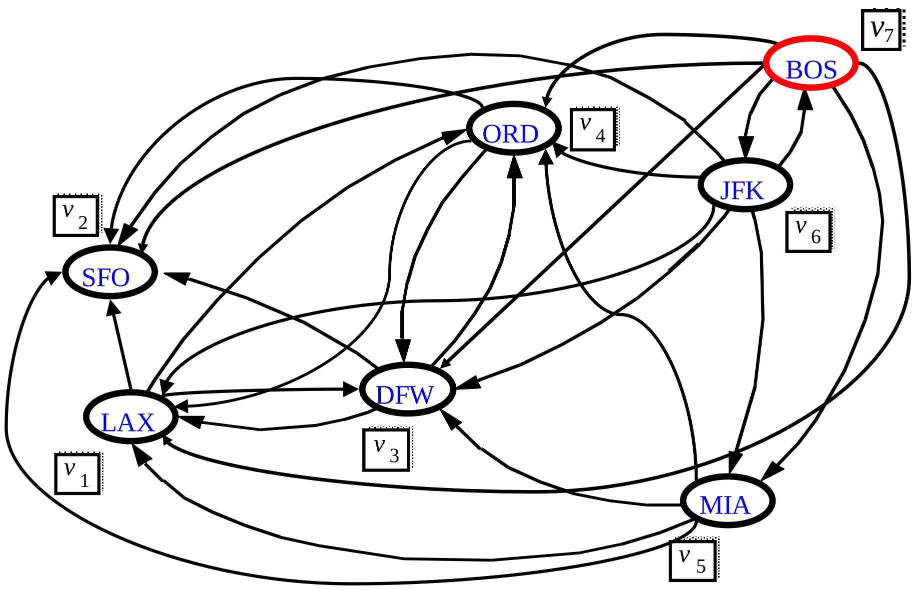


Floyd-Warshall, iteration 6



23.25

Floyd-Warshall, termination



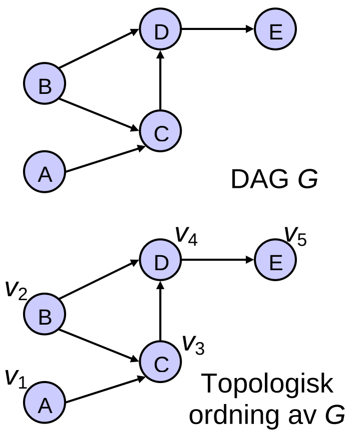
23.26

4 Topological sorting

Directed acyclic graphs and topological order

- A directed acyclic graph (DAG) is a directed graph that has no directed cycles
- A topological order of a graph is a total order $v_1, \ldots, v_n$ of nodes such that each edge (v_i, v_j) fulfills $i < j$
- Example: In a directed graph that corresponds to an instance of task scheduling, a topological order is a sequence of data that fulfill the requirements of the order between data

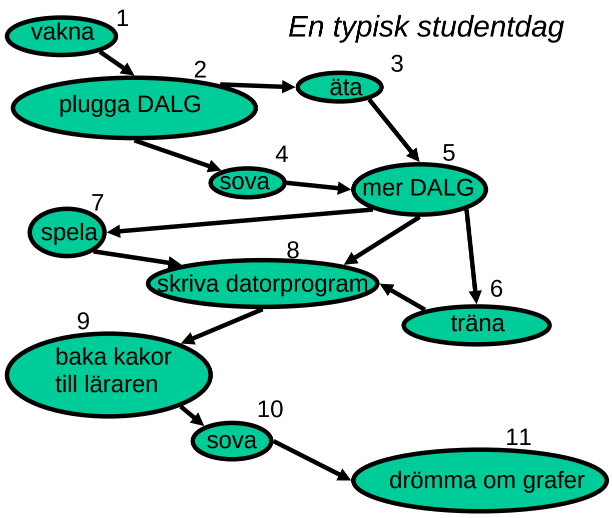
Proposition 1. A directed graph can be arranged using topological order if it is a DAG



23.27

Topological sorting

Number the nodes, so that $(u,v) \in E \Rightarrow u < v$



23.28

Algorithms for topological sort

```
procedure TopologicalSort(G)
    S ← new empty stack
    for all u ∈ G.vertices() do
        let incounter(u) be the in-degree of u
        if incounter(u) = 0 then
            S.push(u)
    i ← 1
    while ¬S.isEmpty() do
        u ← S.pop()
        let u gets number i in the topological order
        i ← i + 1
        for all outgoing edge (u, w) from u do
            incounter(w) ← incounter(w) − 1
            if incounter(w) = 0 then
                S.push(w)
```

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

23.29

Alternative algorithms for topological sort

```
procedure TopologicalSort(G)
    H ← G                                  ▷ temporary copy of G
    n ← G.numVertices
    while H is not empty do
        let v be node without outgoing edges
        mark v with n
```

$n \leftarrow n-1$
remove v from H

Execution time: $O(n+m)$. 23.30

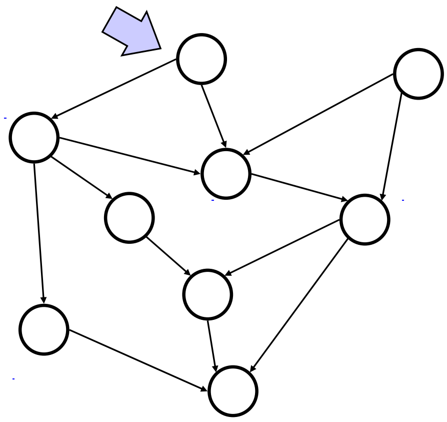
Algorithms for topological sort via DFS

Simulating the algorithm using a depth first search

procedure TOPOLOGICALDFS(G)
  $n \leftarrow G$.NUMVERTICES
  set all nodes and edges $UNEXPLORED$ as in DFS
  for all $v \in G$.VERTICES() **do**
    if GETLABEL(v) $= UNEXPLORED$ **then**
      TOPOLOGICALDFS(G, v)

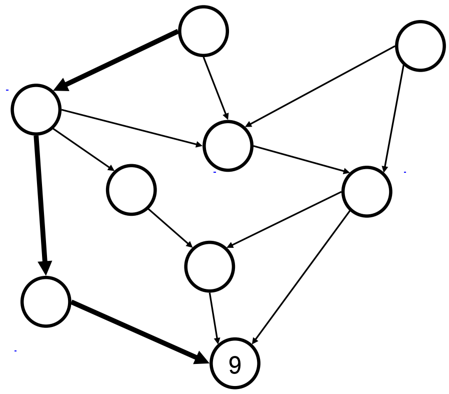
procedure TOPOLOGICALDFS(G, v)
  SETLABEL($v, VISITED$)
  for all $e \in G$.INCIDENTEDGES(v) **do**
    if GETLABEL(e) $= UNEXPLORED$ **then**
      $w \leftarrow$OPPOSITE(v, e)
      if GETLABEL(w) $= UNEXPLORED$ **then**
        SETLABEL($e, DISCOVERY$)
        TOPOLOGICALDFS(G, w)
      else
        e is a cross edge or forward edge
  mark v with a topological number n
  $n \leftarrow n-1$ 23.31

Example: Topological sort



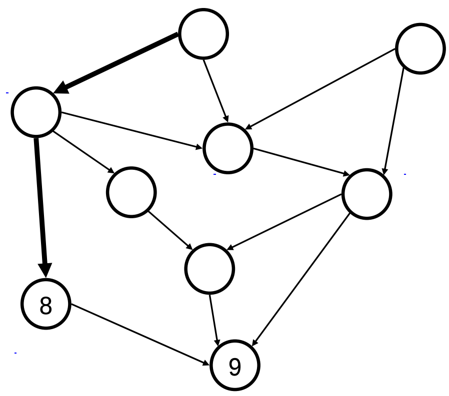
23.32

Example: Topological sort



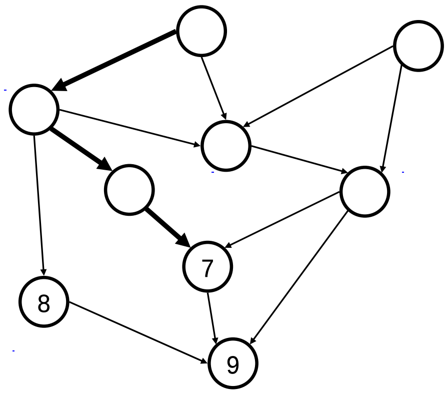
23.33

Example: Topological sort



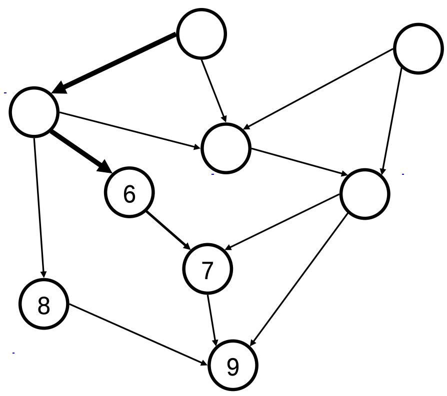
23.34

Example: Topological sort



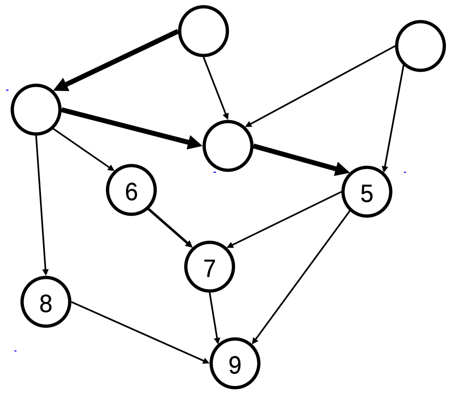
23.35

Example: Topological sort



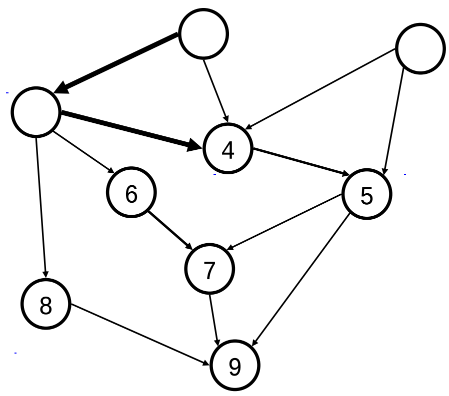
23.36

Example: Topological sort



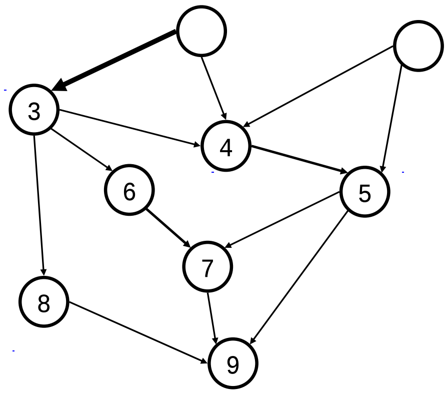
23.37

Example: Topological sort

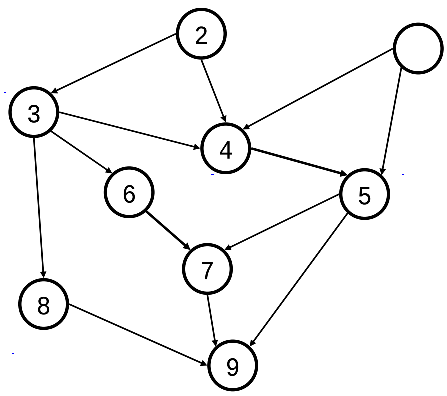


23.38

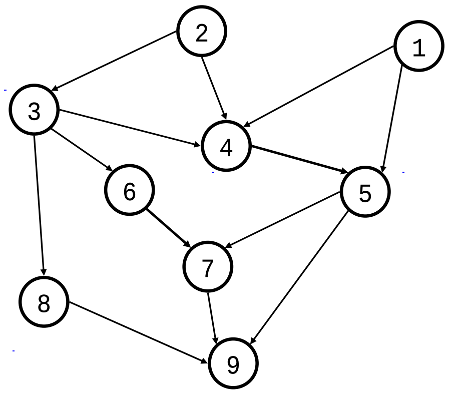
Example: Topological sort



Example: Topological sort



Example: Topological sort



23.41

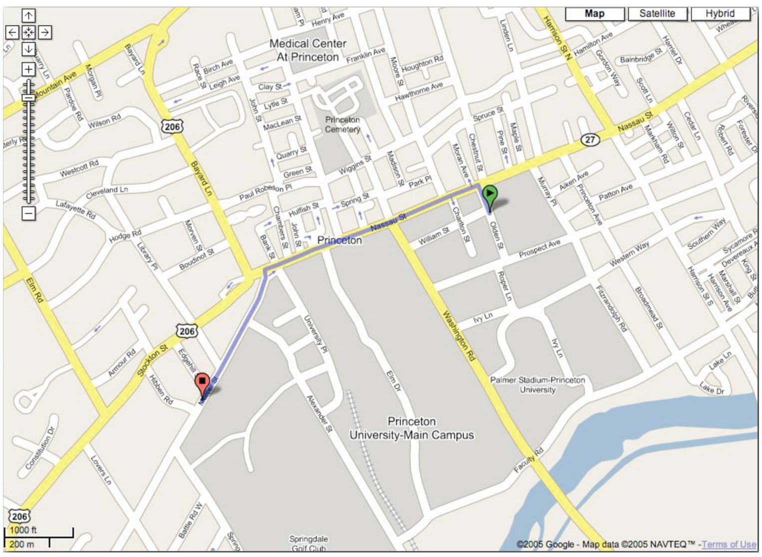
5 Weighted graphs

Weighted graphs

- In a weighted graph, each arc is associated with a numerical value called the edge *weight*.
- Edge weights can represent distances, costs, etc.

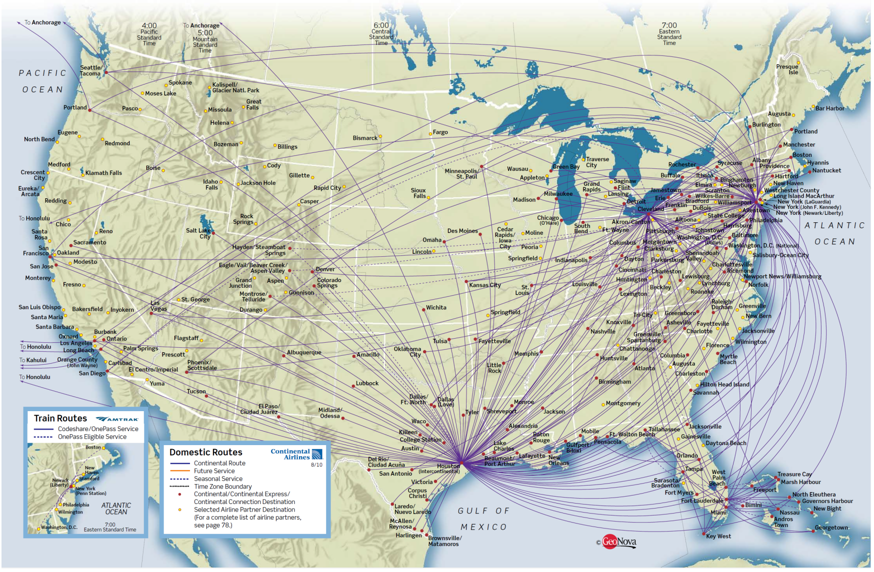
23.42

Google maps



23.43

The flight routes of the Continental company in USA (august 2010)



23.44

Apllications

- map applications
- Seam carving
- Robot navigation
- Texture mapping
- Typesetting in TeX
- Traffic in urban environments
- Routing of messages in telecom.
- Routing protocols for networks (OSPF, BGP, RIP)

http://en.wikipedia.org/wiki/Seam_carving

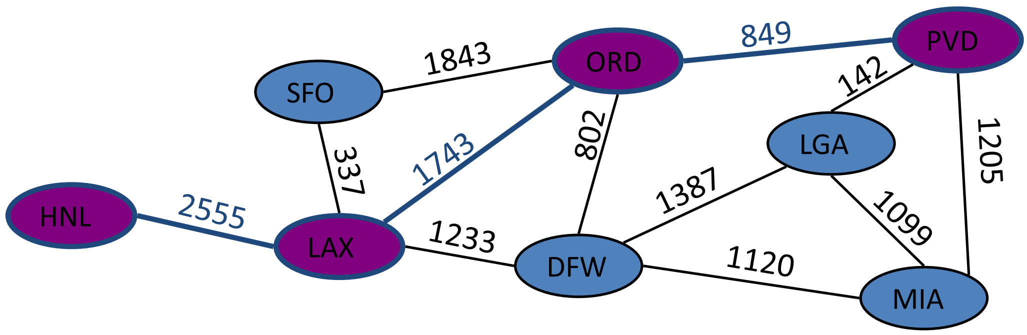
23.45

6 Shortest paths

The problem of shortest paths

- Given a weighted graph and 2 nodes u and v we will find a path between u and v with minimal total weight.
  - The length of a path is the sum of the weights of the path edges

Example
Shortest road between Providence and Honolulu

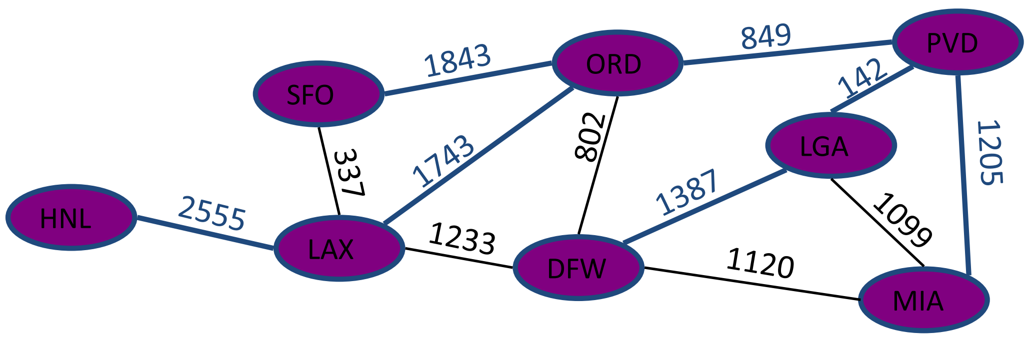


23.46

Properties of shortest paths

- A subpath 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 roads from Providence



23.47

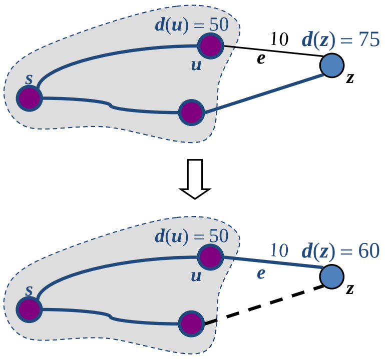
Dijkstra algorithm

- The distance from one node v to a node s is the length of the shortest route between s and v
- Dijkstra's algorithm calculates the distances from a given start node p to all nodes V in the graph
- Assumptions:
  - the graph is connected
  - edges are undirected
  - the graph has no loops and parallel edges
  - the edge weights are *not negative*
- We build a "cloud" of nodes starting at s, which ultimately cover all nodes
- We mark each node v with $d(v)$, which represents the distance between v and s in the subgraph consisting of the cloud and the nodes that are neighbors to the cloud
- In each step
  - we add the node u outside the cloud having the least distance marking $d(u)$
  - we update the labeling of nodes that are neighbors to u

23.48

Extension step

- Consider an edge $e = (u, z)$ such that
  - u is the node we recently added to the cloud
  - z not in the cloud
- The relaxation of edge e updates $d(z)$ as follows:
  - $d(z) \leftarrow \min\{d(z), d(u) + weight(e)\}$



23.49

Dijkstra pseudo-code

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.

23.50

Example

- dijkstra(A, F);

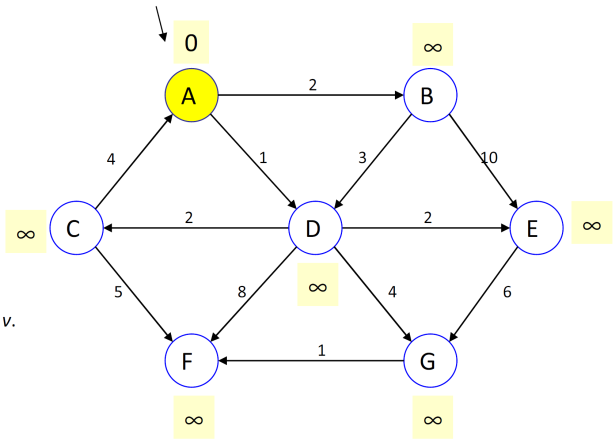
function **dijkstra**(v_1, v_2):
- v_1's cost := 0.
- *pqueue* := {v_1}. // ordered by cost

- 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 := ∞.



- 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

pqueue = {**A:0**}

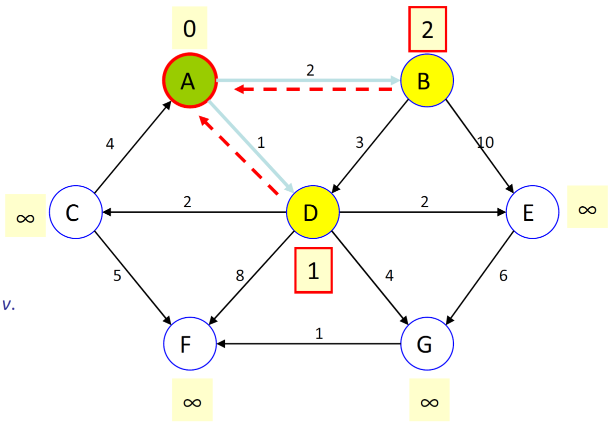
23.51

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*. // A
    mark *v* as visited.
    if *v* is v_2, 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 v_2 back to v_1,
   following previous pointers.



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

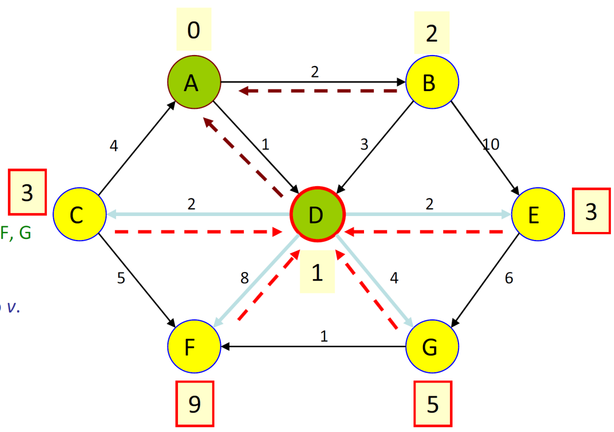
23.52

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*. // D
    mark *v* as visited.
    if *v* is v_2, 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 v_2 back to v_1,
   following previous pointers.



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

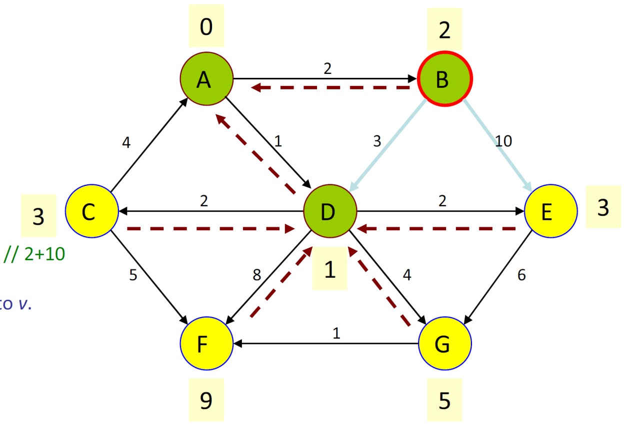
23.53

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.



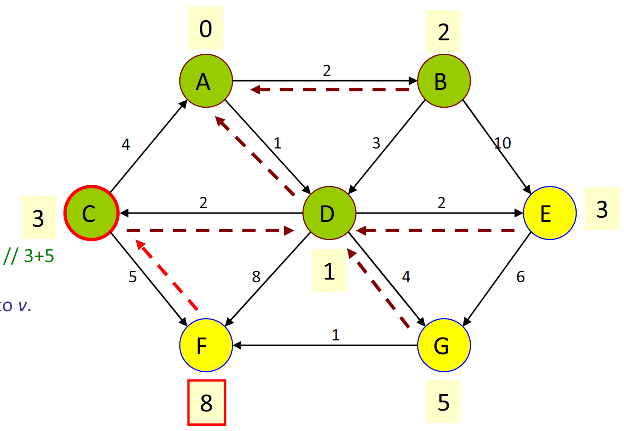
pqueue = {C:3, E:3, G:5, F:9}

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.



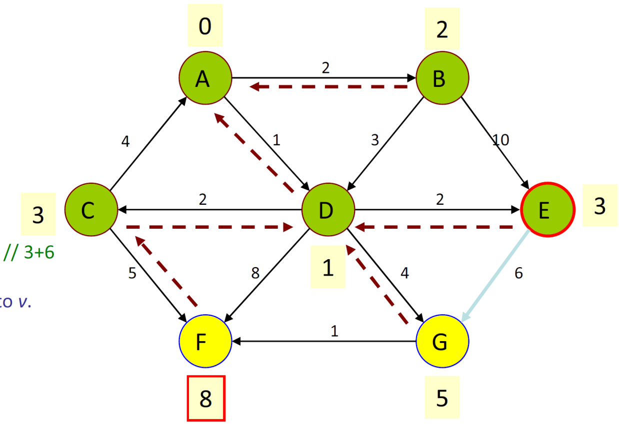
pqueue = {E:3, G:5, **F:8**}

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.



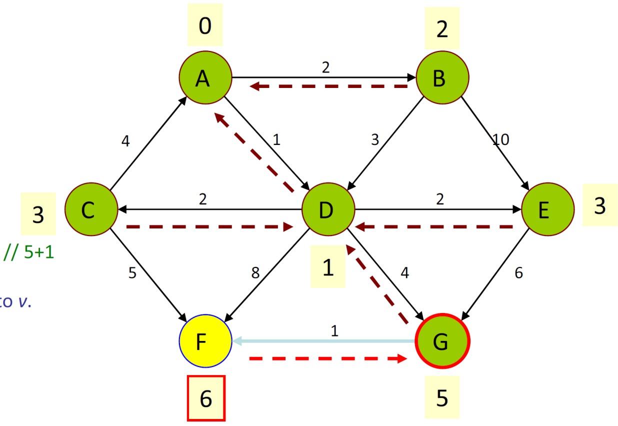
pqueue = {G:5, F:8}

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**}

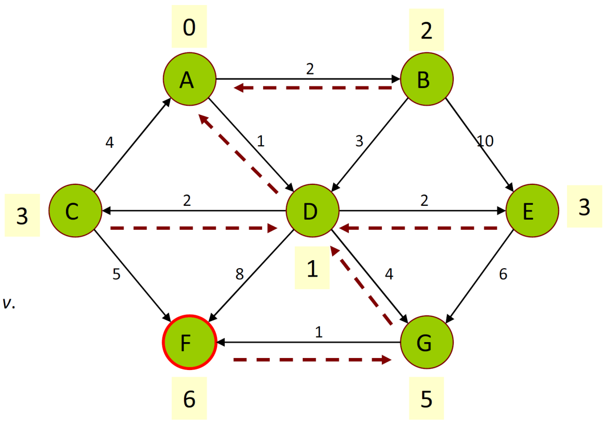
Example

- 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 = {}

23.58

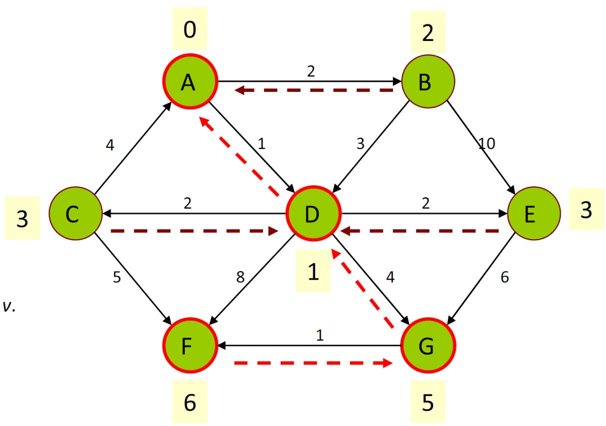
Example

- 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}
```



23.59

Analysis of Dijkstra algorithm

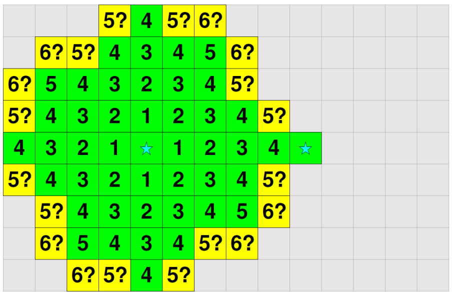
- Graph operations
  - We call incidentEdges one time for each node
- Marking operations
  - We retrieve/set the distance and locator for node z $O(deg(z))$ times
  - Setting/retrieving a marking takes $O(1)$ time
- Operations on priority queues
  - Each node is inserted once and removed once from the priority queue, where each insertion and removal takes $O(\log n)$ time
  - A node key in the priority queue changes at most $deg(w)$ times, where each key change takes $O(\log n)$ time
- Dijkstra algorithm has execution time $O((n+m)\log n)$ given that the graph is represented with an adjacency list
  - Remember $\sum_v deg(v) = 2m$

- The execution time can also be expressed as $O(m \log n)$ because we assumed that the graph is connected

23.60

Observations

- Dijkstra's algorithm works by incrementally calculating the shortest route to intermediate nodes which may be useful.
  - Most of these paths are in the wrong direction.



- The algorithm does not have a general idea of the objective to be achieved; it explores outward in all directions.
  - Can we explore in smarter order?

23.61

Heuristics

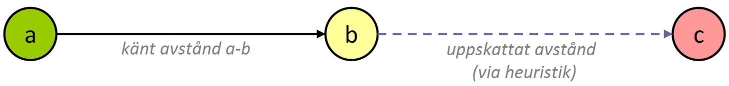
- heuristics: Speculation, estimation or guess that determines how the search for a solution to a problem goes.
  - Example: Estimate the distance between two points in a Google Maps graph to the length of a straight line between the points.

- valid heuristics: One that does not overestimate distance.
  - Ok if heuristics sometimes underestimate the distance (for example Google Maps)

23.62

A*-algorithm

- A*("A-star): A modified version of Dijkstra's algorithm uses a heuristic function to guide the exploration of the search space.



- Suppose we are looking for routes from start node a to c
  - Each intermediate node b has two costs:
  - The name (exact) cost from the start node a to b
  - The heuristic (estimated) cost from B to the end node c.

- Idea: Run Dijkstra's algorithm, but use the following priority in the priority queue:
  - priority(b) = cost(a, b) + Heuristic(b, c)
  - choose to explore ways with lower estimated cost

23.63

Example: Labyrinth heuristics

- A possible heuristics to search for paths in a labyrinth::
  - $\mathrm{H}(p_1, p_2) = \mathrm{abs}(p_1.x - p_2.x) + \mathrm{abs}(p_1.y - p_2.y)$ // dx + dy
  - Idea: Explore the neighbors with low-value (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

23.64

Pseudocode of 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.
```

Notice that the nodes *priorities* are influenced by heuristics, but not their *costs*. 23.65