Föreläsning 23

Riktade och viktade grafer

TDDD86: DALP

Utskriftsversion av föreläsning i *Datastrukturer, algoritmer och programmeringsparadigm*
4 december 2015

Tommy Färnqvist, IDA, Linköpings universitet

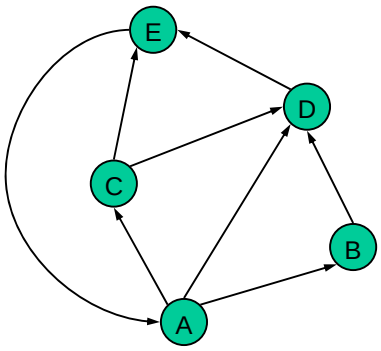
Innehåll

Innehåll

1 Riktade grafer

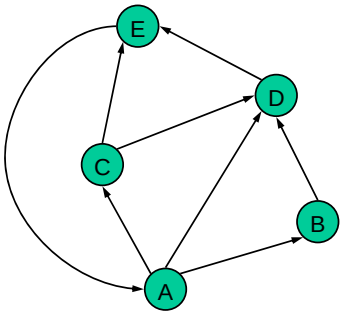
Introduktion

- I en riktad graf är alla bågar riktade



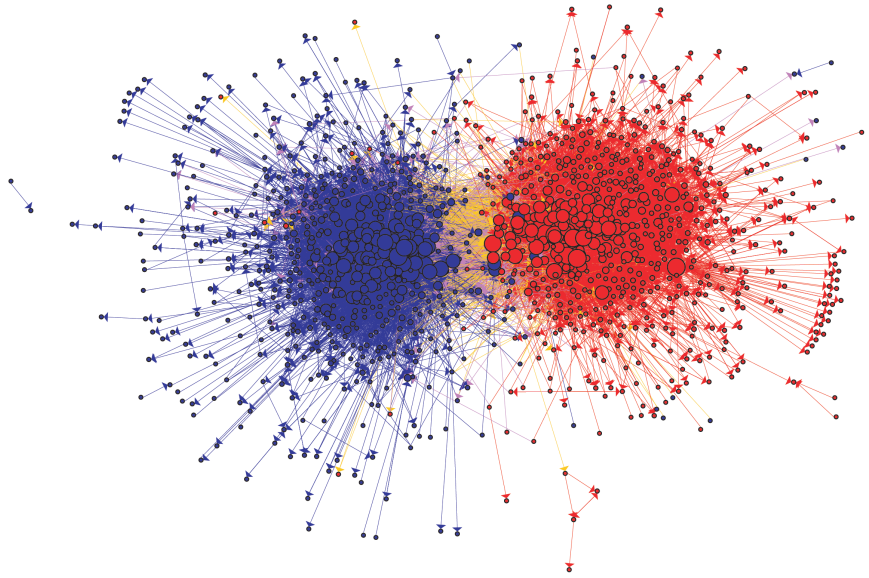
Egenskaper

- En graf $G = (V, E)$ sådan att varje båge går i en riktning:
  - Båge (a, b) går från a till b men inte från b till a.
- Om G är enkel (inga parallella bågar och inga öglor) gäller $m \leq n \cdot (n-1)$, d.v.s. $m \in O(n^2)$.



23.4

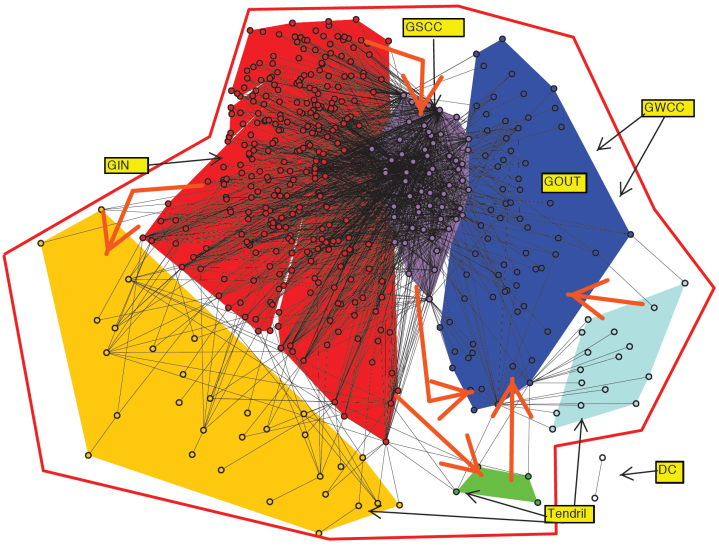
Politisk bloggosfär-graf



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

23.5

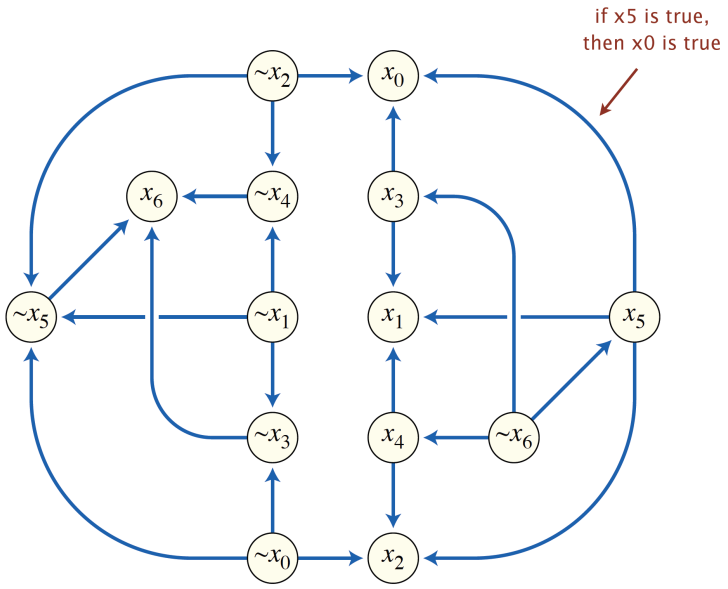
Nattliga lån mellan banker



The Topology of the Federal Funds Market, Bech och Atalay, 2008

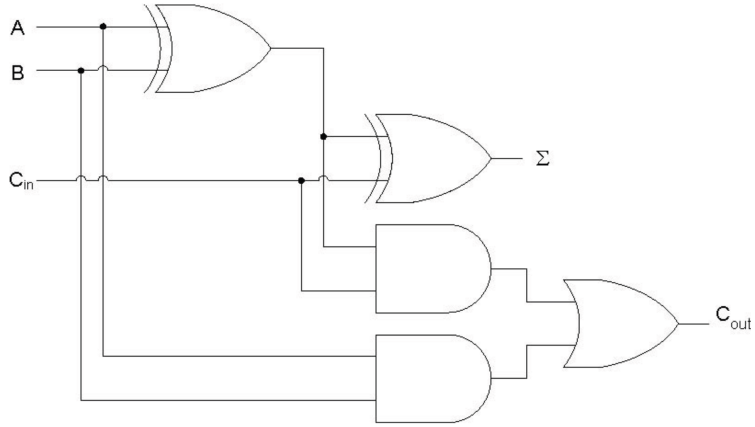
23.6

Implikationsgraf



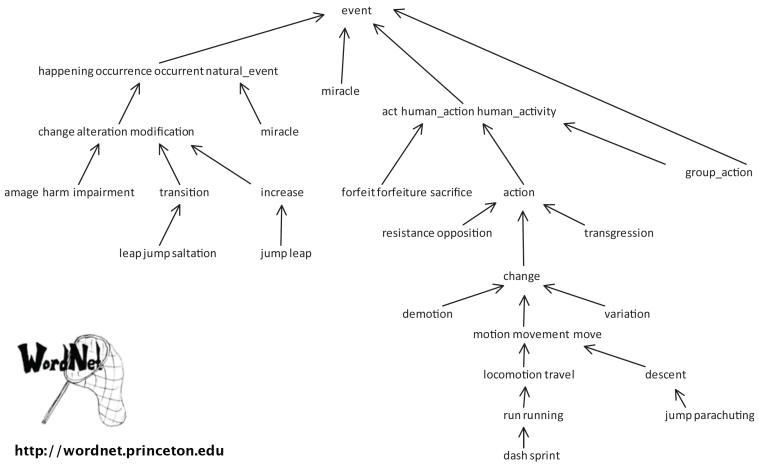
23.7

Kombinatorisk krets

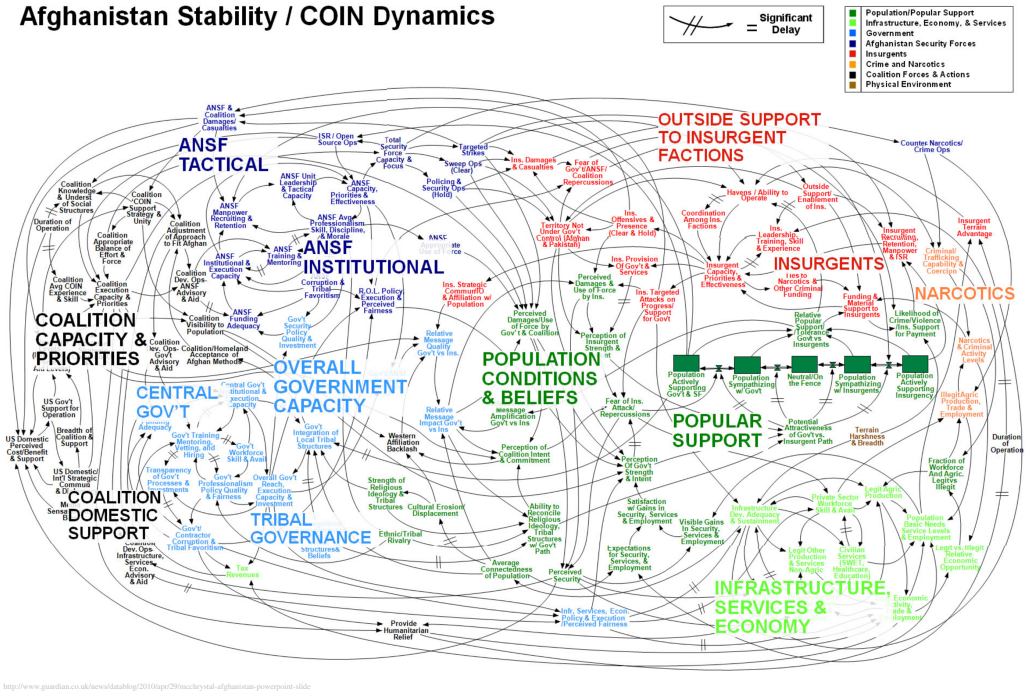


23.8

WordNet-graf



23.9



http://www.guardian.co.uk/news/datablog/2010/apr/29/mcchrystal-afghanistan-powerpoint-slide

23.10

Tillämpningar

riktad graf	nod	rikatd båge
transport	gatukorsning	enkelriktad gata
www	hemsida	hyperlänk
näringskedja	art	rovdjur-byte-förhållande
schemaläggning	uppgift	föregångarvillkor
finansiell	bank	transaktion
mobiltelefon	person	ringt samtal
smittsam sjukdom	person	infektion
citeringar	artikel	citering
objektgraf	objekt	pekare
arvshierarki	klass	ärver från
kontrollflöde	kodblock	hopp

23.11

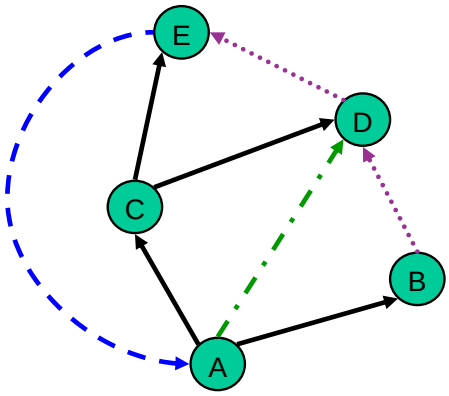
Några algoritmiska grafproblem

- Stig. Finns det en riktad stig från s till t?
- Kortaste väg. Vilken är den kortaste riktade stigen från s till t?
- Stark konnektivitet. Finns det en riktad stig mellan alla par av noder?
- Topologisk sorting. Går det att rita den riktade grafen så att alla kanter pekar uppåt?
- Transitivt hölje. För vilka noder v och w finns det en stig från v till w?
- Page Rank. Hur betydelsefull är en webbsida?

23.12

Riktad DFS

- Vi kan specialisera traverseringsalgoritmerna (DFS och BFS) till riktade grafer
- I den riktade DFS-algoritmen får vi fyra typer av bågar
  - ”discovery”-bågar
  - bakåt-bågar
  - framåt-bågar
  - korsande bågar
- En riktad DFS med start i nod s bestämmer vilka noder som är nåbara från s

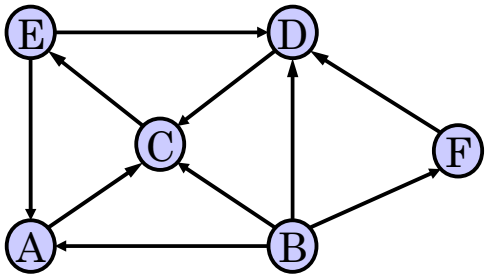


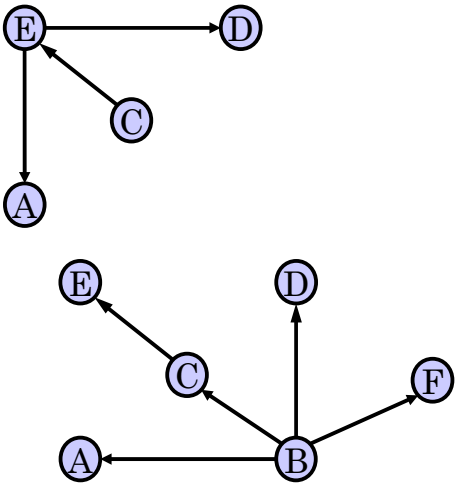
23.13

2 Konnektivitet

Nåbarhet

DFS-träd rotat i v: noder nåbara från v via riktade stigar

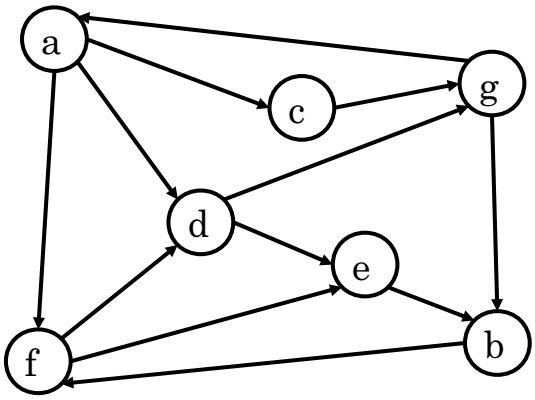




23.14

Starkt sammanhängande

Varje nod är nåbar från alla andra noder

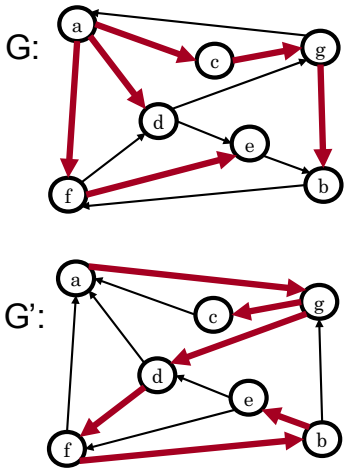


23.15

Algoritm för att avgöra starkt sammanhängande

- Välj en nod v i G
- *// Kan alla noder nås från v?*Utför DFS från v i G
  - Om det finns w som inte besöks svara ”nej”
- Låt G' vara G med riktningen på varje båge omkastad
- *// Kan v nås från alla noder?*Utför DFS från v i G'
  - Om det finns w som inte besöks svara ”nej”
  - Annars, svara ”ja”

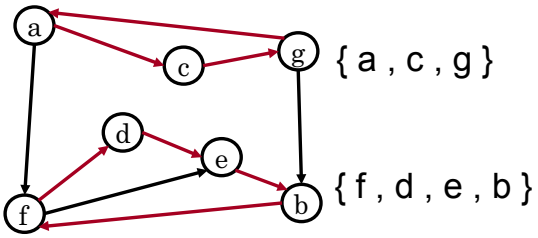
- Körtid: $O(n+m)$



23.16

Starkt sammanhängande komponenter

- Maximal delgraf sådan att varje nod kan nå alla andra noder i delgrafen
- Kan också göras i $O(n+m)$ tid genom att använda DFS i flera steg

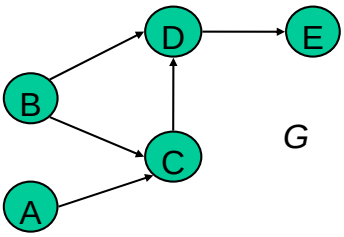


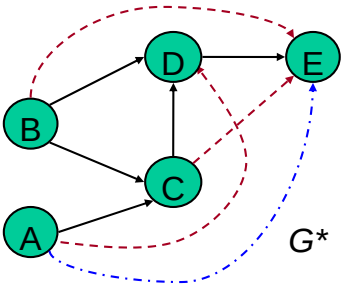
23.17

3 Transitivt hölje

Transitivt hölje

- Givet en riktad graf G, låt det transitiva höljet av G vara den riktade grafen G^* sådan att
  - G^* har samma noder som G
  - om G har en riktad stig från u till v ($u \neq v$) så har G^* en riktad båge från u till v
- Det transitiva höljet ger information om nåbarheten i en riktad graf





23.18

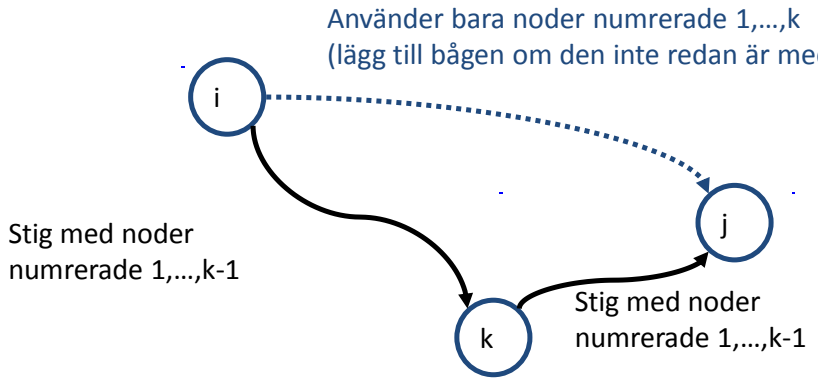
Beräkning av transitiva höljet

- Vi kan köra DFS med start i varje nod $v_1, \ldots, v_n$, alltså $O(n \cdot (n+m))$
- Alternativt använda dynamisk programmering: Floyd-Warshalls algoritm

23.19

Transitiva höljet med Floyd-Warshall

- Numrera noderna $1, 2, \ldots, n$.
- I fas k, betrakta bara stigar som använder noder med nummer $1, 2, \ldots, k$ som mellanliggande noder:



23.20

Floyd-Warshalls algoritm

- Floyd-Warshalls algoritm numrerar noderna i G som $v_1, \ldots, v_n$ och beräknar en serie riktade grafer $G_0, \ldots, G_n$
  - $G_0 = G$
  - G_k har en riktad båge (v_i, v_j) om G har en riktad stig från v_i till v_j med mellanliggande noder i mängden $\{v_1, \ldots, v_k\}$
- Vi ser att $G_n = G^*$
- I fas k beräknas grafen G_k utgående från G_{k-1}
- Körtid: $O(n^3)$ om areAdjacent är $O(1)$

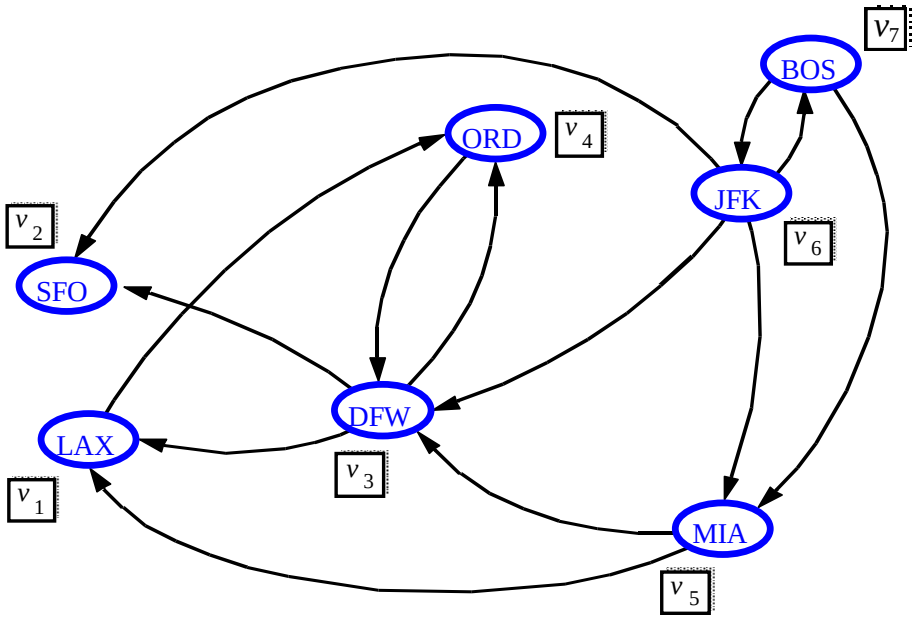
23.21

Floyd-Warshalls algoritm

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

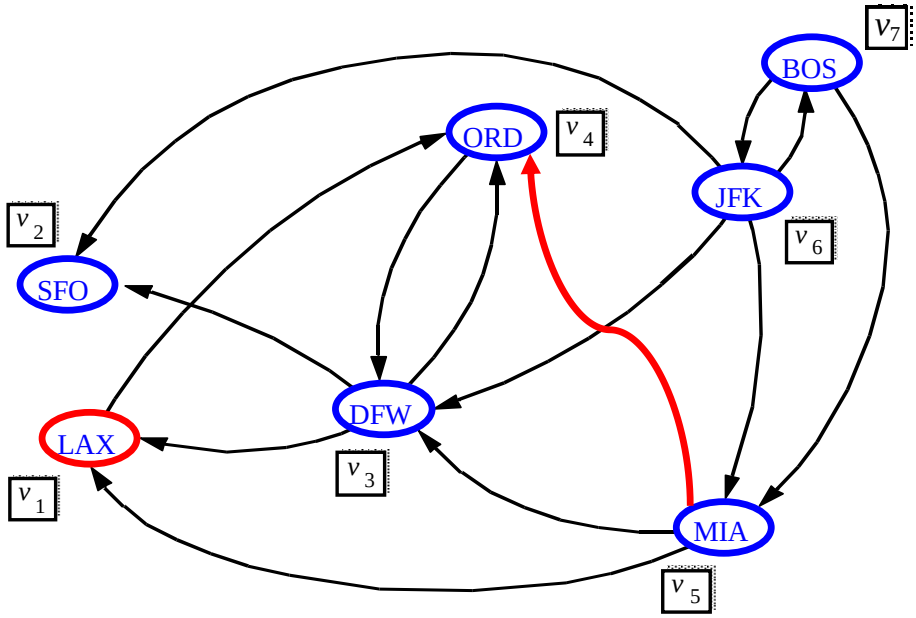
23.22

Exempel: Floyd-Warshall



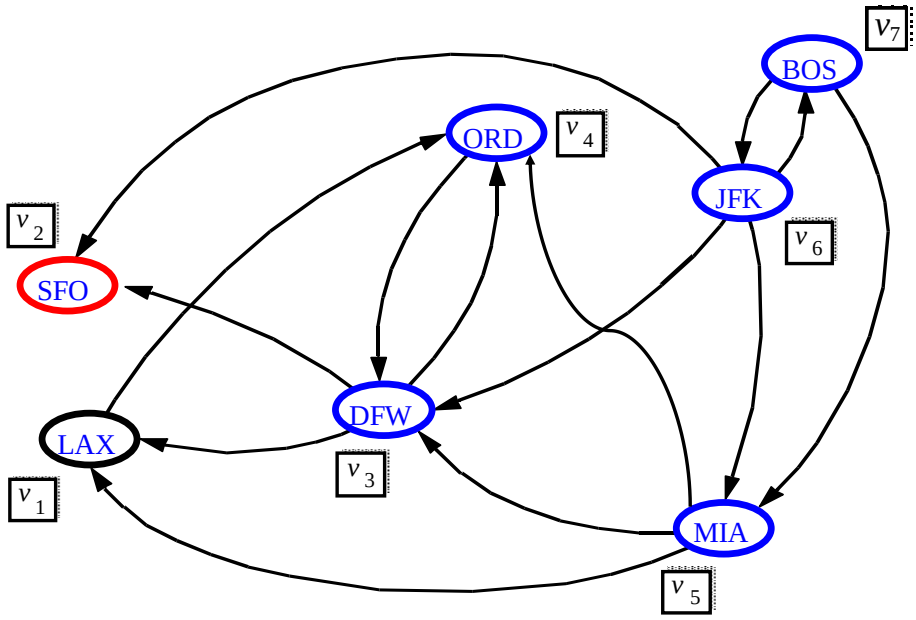
23.23

Floyd-Warshall, iteration 1



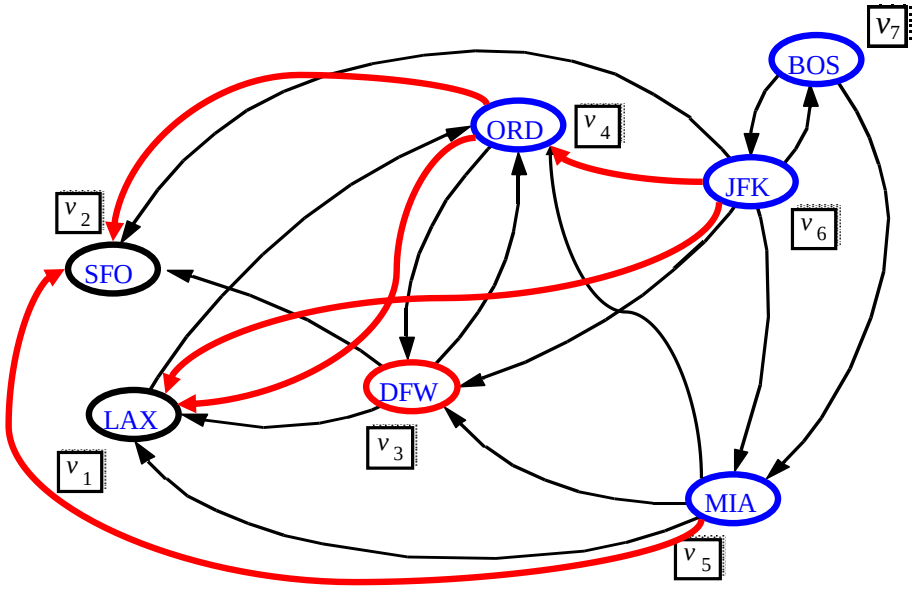
23.24

Floyd-Warshall, iteration 2



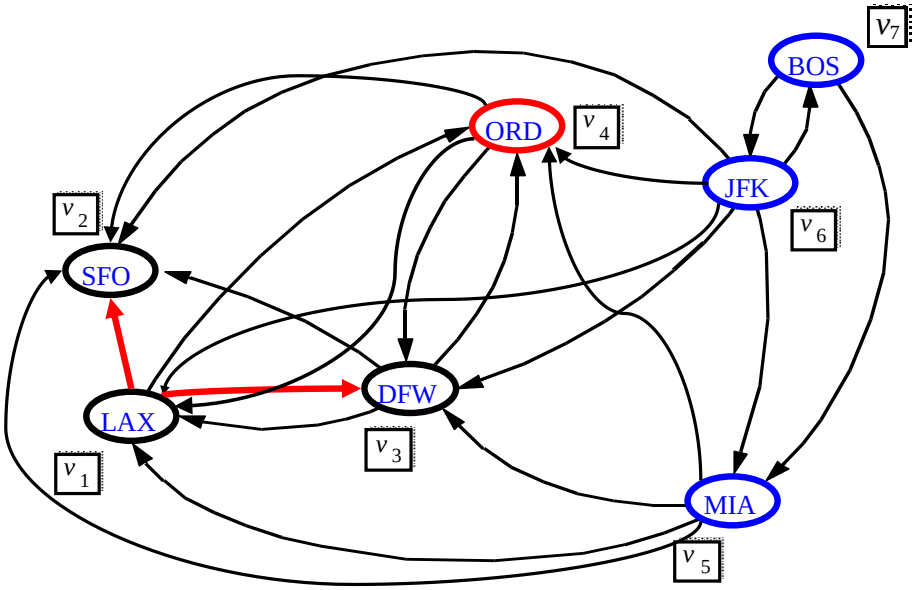
23.25

Floyd-Warshall, iteration 3



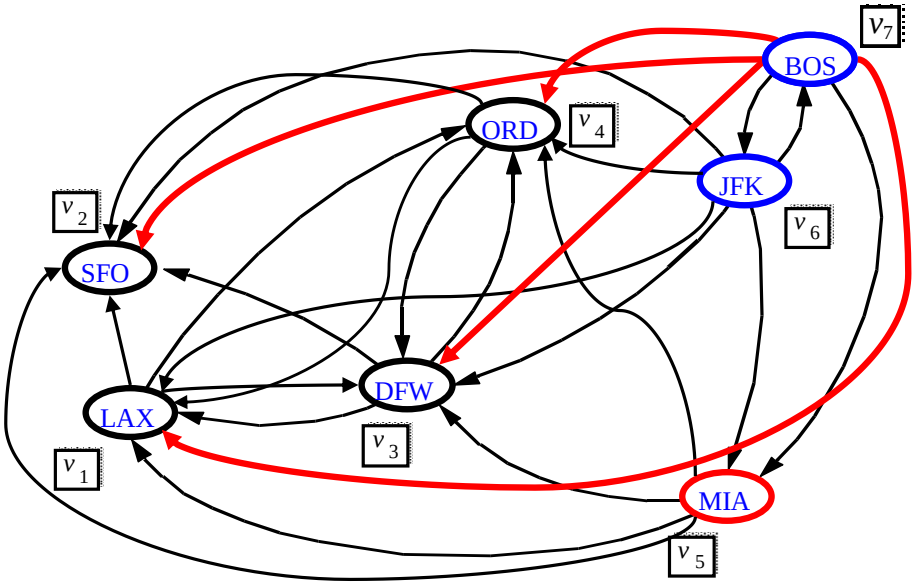
23.26

Floyd-Warshall, iteration 4



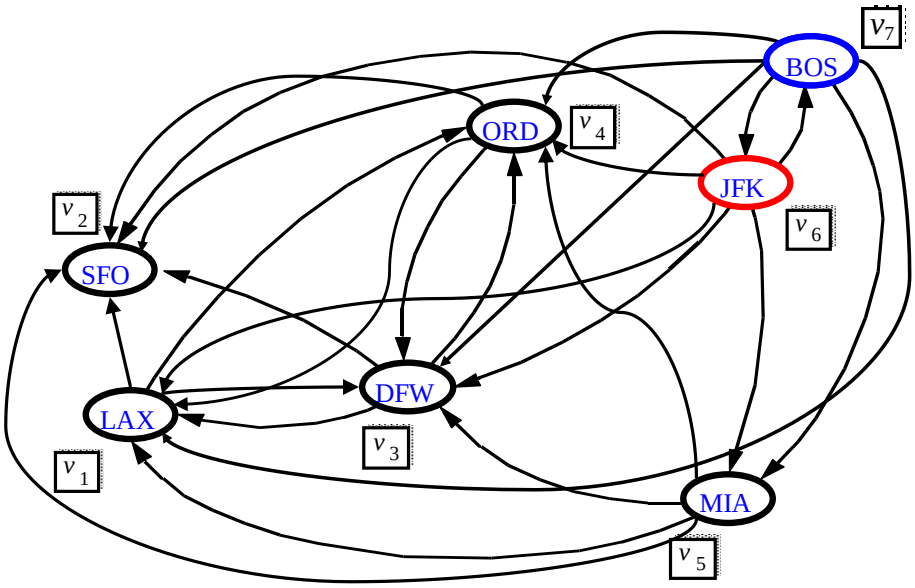
23.27

Floyd-Warshall, iteration 5



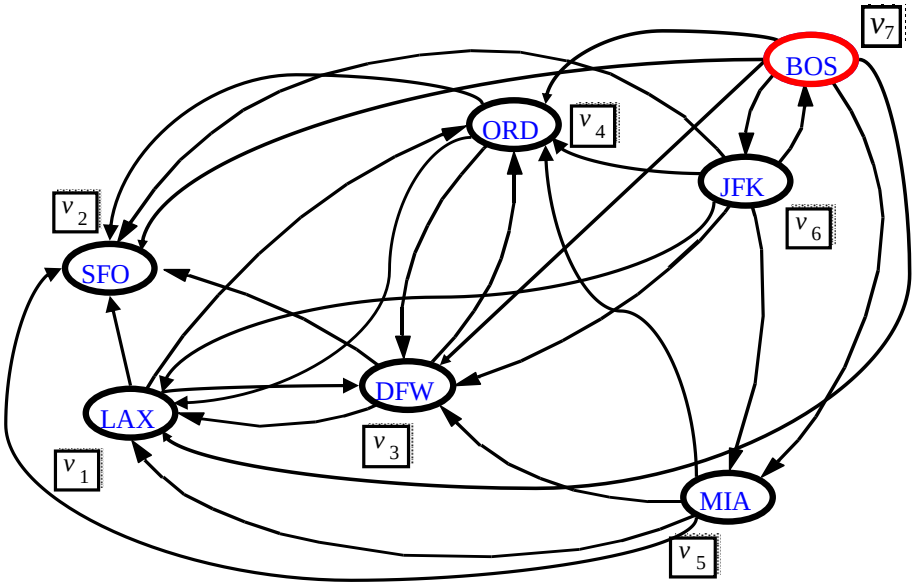
23.28

Floyd-Warshall, iteration 6



23.29

Floyd-Warshall, slutet



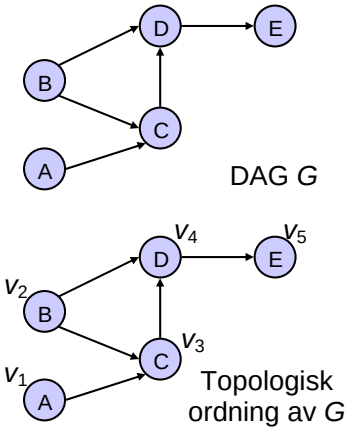
23.30

4 Topologisk sortering

Riktade acykliska grafer och topologisk ordning

- En riktad acyklisk graf (DAG) är en riktad graf som inte har några riktade cykler
- En topologisk ordning av en graf är en totalordning $v_1, \ldots, v_n$ av noderna sådan att varje båge (v_i, v_j) uppfyller $i < j$
- Exempel: I en riktad graf som svarar mot en instans av uppgiftsschemaläggning är en topologisk ordning en sekvens av uppgifter som uppfyller kraven på inbördes ordning mellan uppgifterna

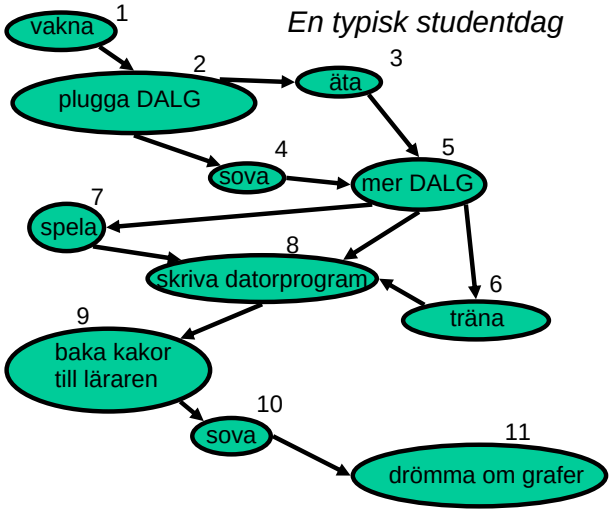
Proposition 1. En riktad graf går att ordna topologiskt omm grafen är en DAG



23.31

Topologisk sortering

Numrera noderna, så att $(u, v) \in E \Rightarrow u < v$



23.32

Algoritm för topologisk sortering

procedure TOPOLOGICALSORT(G)
  $S \leftarrow$ ny tom stack
  for all $u \in G$.VERTICES() **do**
    låt INCOUNTER(u) vara ingraden för u
    if INCOUNTER(u) $= 0$ **then**
      S.PUSH(u)
  $i \leftarrow 1$
  while $\neg S$.ISEMPTY() **do**
    $u \leftarrow S$.POP()
    låt u få nummer i i den topologiska ordningen
    $i \leftarrow i+1$
    for all utgående kanter (u,w) från u **do**
      INCOUNTER(w) $\leftarrow$ INCOUNTER(w) -1
      if INCOUNTER(w) $= 0$ **then**
        S.PUSH(w)

Körtid: $O(n+m)$.

23.33

Alternativ algoritm för topologisk sortering

procedure TOPOLOGICALSORT(G)
  $H \leftarrow G$ ▷ temporär kopia av G
  $n \leftarrow G$.NUMVERTICES
  while H är icke-tom **do**
    låt v vara en nod utan utgående bågar
    märk v med n
    $n \leftarrow n-1$
    ta bort v från H

Körtid: $O(n+m)$. Hur då...?

23.34

Algoritm för topologisk sortering via DFS

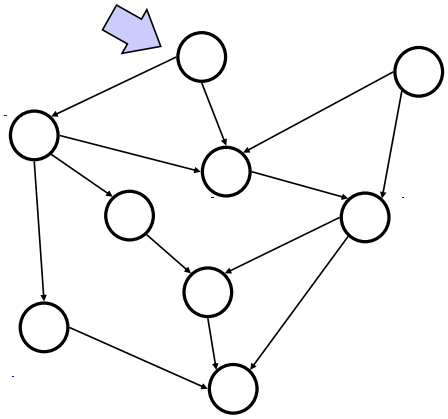
Simulera algoritmen genom att använda en djupetförstsökning

procedure TOPOLOGICALDFS(G)
  $n \leftarrow G$.NUMVERTICES
  sätt alla noder och bågar till *UNEXPLORED* som i DFS
  for all $v \in G$.VERTICES() **do**
    if GETLABEL(v) $=$ *UNEXPLORED* **then**
      TOPOLOGICALDFS(G,v)

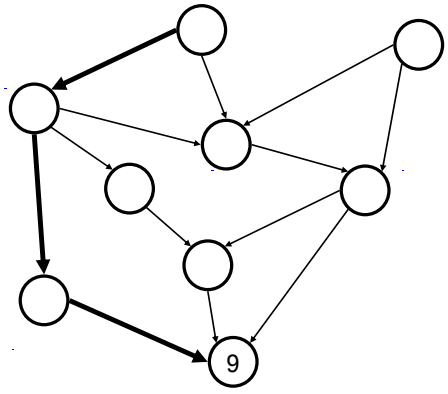
procedure TOPOLOGICALDFS(G,v)
  SETLABEL(v,*VISITED*)
  for all $e \in G$.INCIDENTEDGES(v) **do**
    if GETLABEL(e) $=$ *UNEXPLORED* **then**
      $w \leftarrow$OPPOSITE(v,e)

if GETLABEL$(w) = UNEXPLORED$ **then**
SETLABEL$(e, DISCOVERY)$
TOPOLOGICALDFS(G, w)
else
e är korsande båge eller framåt-båge
märk v med topologiskt nummer n
$n \leftarrow n - 1$

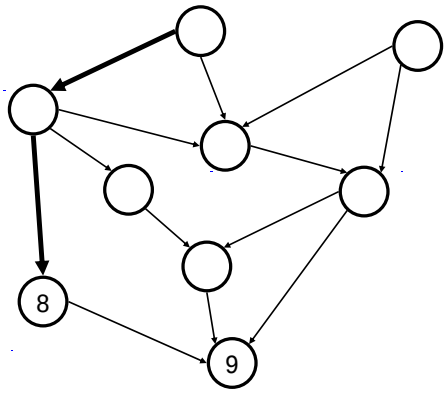
Exempel: Topologisk sortering



Exempel: Topologisk sortering

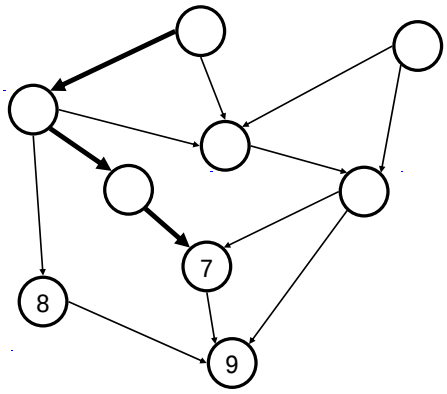


Exempel: Topologisk sortering



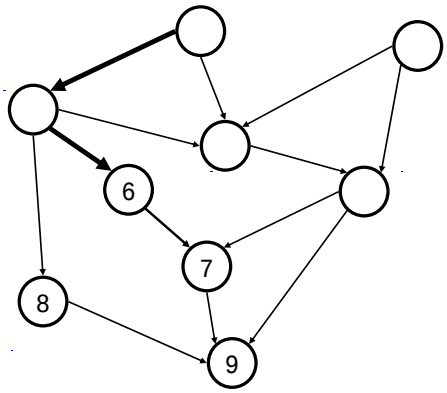
23.38

Exempel: Topologisk sortering



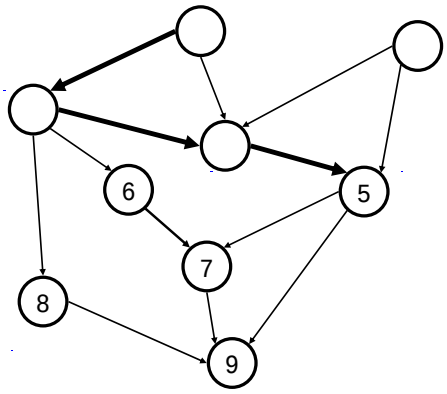
23.39

Exempel: Topologisk sortering



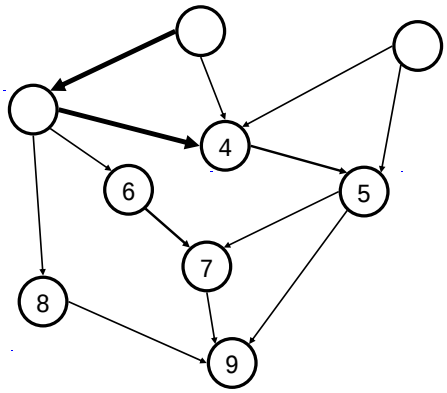
23.40

Exempel: Topologisk sortering



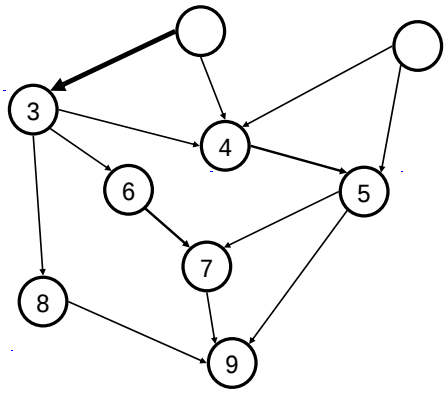
23.41

Exempel: Topologisk sortering



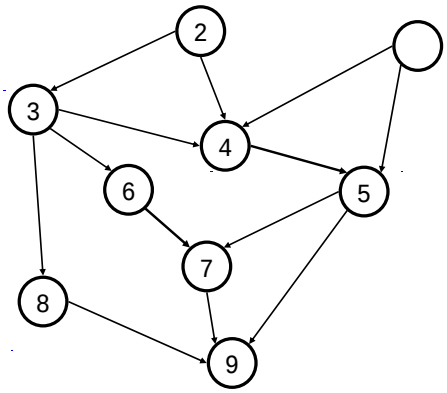
23.42

Exempel: Topologisk sortering



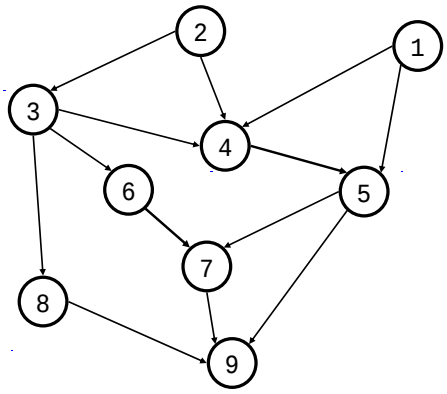
23.43

Exempel: Topologisk sortering



23.44

Exempel: Topologisk sortering



23.45

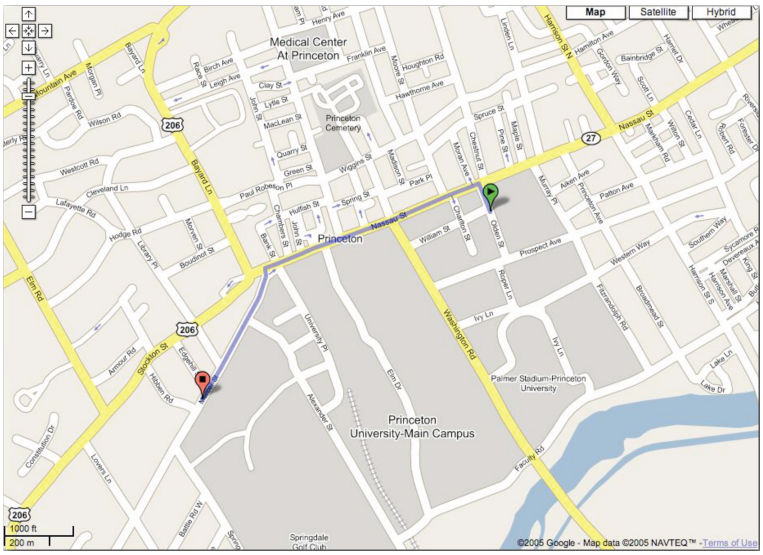
5 Viktade grafer

Viktade grafer

- I en viktad graf är varje båge associerad med ett numeriskt värde kallat bågens *vikt*.
- Bågvikter kan representera avstånd, kostnader, etc.

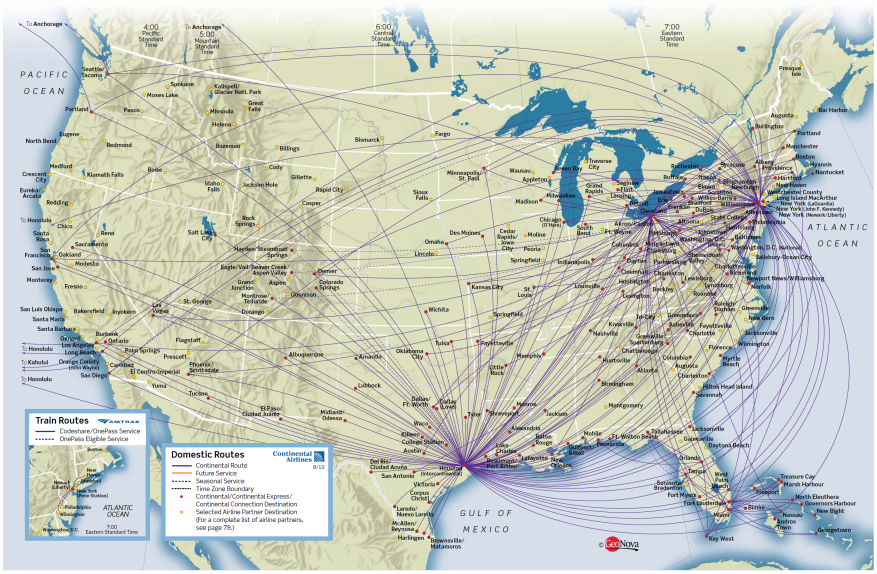
23.46

Google maps



23.47

Bolaget Continentals flygrutter i USA (augusti 2010)



23.48

Tillämpningar

- PERT/CPM
- Kartapplikationer
- Seam carving
- Robotnavigering
- Texture mapping
- Typsättning i TeX
- Trafikplanering i stadsmiljö
- Optimal pipelining för VLSI-chip
- Schemaläggning av telemarketingförsäljare
- Routing av meddelanden inom telekom.
- Routingprotokoll för nätverk (OSPF, BGP, RIP)
- Utnyttja gynsamma situationer vid valutahandel
- Optimal ruttplanering för lastbilar givet trafikstockningsmönster

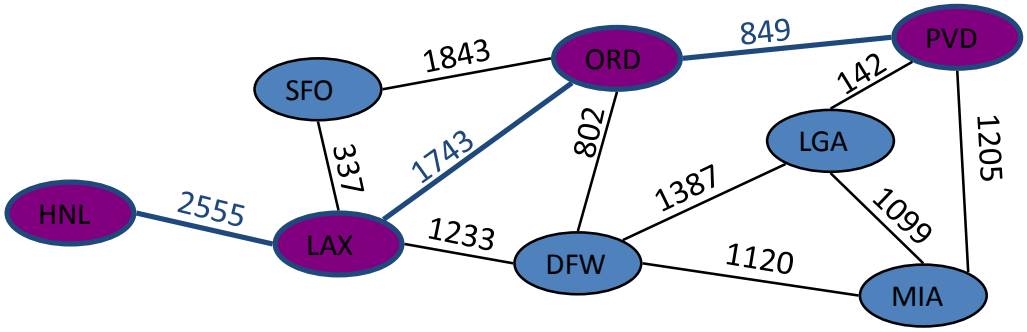
Eng. *endpoints, incident, adjacent, degree, parallel, loops* 23.49

6 Kortaste vägar

Problemet kortaste väg

- Givet en viktad graf och två noder u och v vill vi hitta en stig mellan u och v med minimal total vikt.
  - Längden av en stig är summan av vikterna på stigens bågar

Exempel
Kortaste vägen mellan Providence och Honolulu

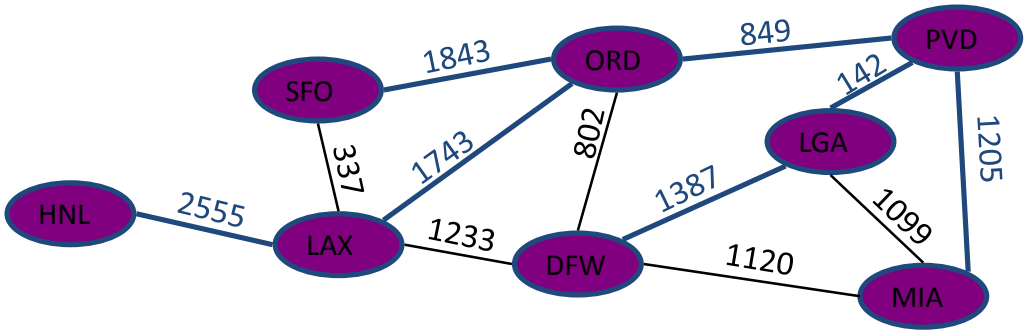


23.50

Egenskaper hos kortaste vägar

- En delväg av en kortaste väg är också en kortaste väg
- Det finns ett träd av kortaste vägar från en startnod till alla andra noder

Exempel
Ett träd av kortaste vägar från Providence



23.51

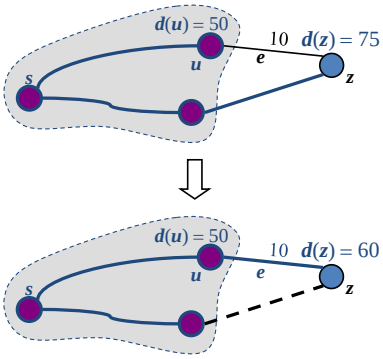
Dijkstras algoritm

- Avståndet från en nod v till en nod s är längden av kortaste vägen mellan s och v
- Dijkstras algoritm beräknar avstånden från en given startnod s till alla noder v i grafen
- Antaganden:
  - grafen är sammanhängande
  - bågarna är oriktade
  - grafen har inga öglor eller parallella bågar
  - bågvikterna är *ickenegativa*
- Vi bygger ett "moln" av noder med start i s, som till slut täcker alla noder
- Vi märker varje nod v med $d(v)$, vilket betecknar avståndet mellan v och s i delgrafen bestående av molnet och noderna som är grannar till molnet
- I varje steg
  - lägger vi till den nod u utanför molnet som har minst avståndsmärkning $d(u)$
  - uppdaterar vi märkningen av noderna som är grannar till u

23.52

Utökningssteget

- Betrakta en båge $e = (u,z)$ sådan att
  - u är noden vi nyligen lagt till i molnet
  - z inte är med i molnet
- Relaxeringen av bågen e uppdaterar $d(z)$ enligt:
  - $d(z) \leftarrow \min\{d(z), d(u) + weight(e)\}$



23.53

Dijkstra pseudokod

```
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.
```

23.54

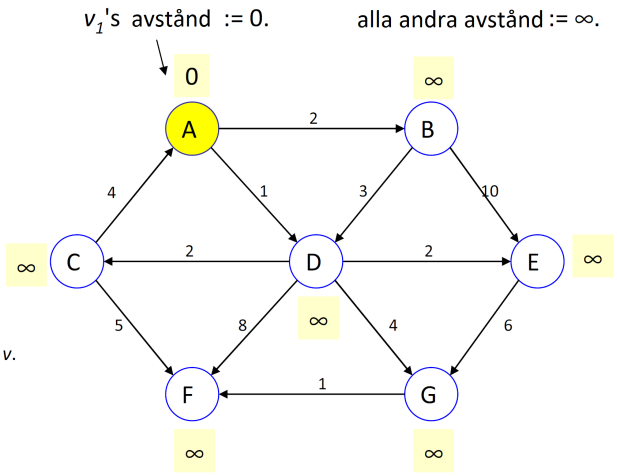
Exempel

- 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.



pqueue = {**A:0**}

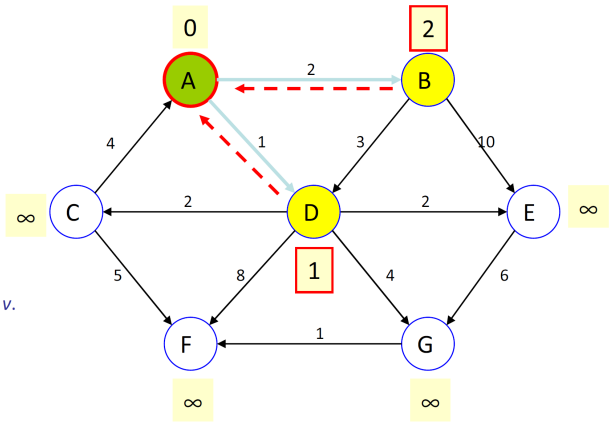
- I våra diagram färglägger vi en nod:
  - **vit** om den är outforskad
  - **gul** om den köats för senare behandling
  - **grön** om den besökts (plockats ut ur kön) och behandlats

Exempel

- 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.



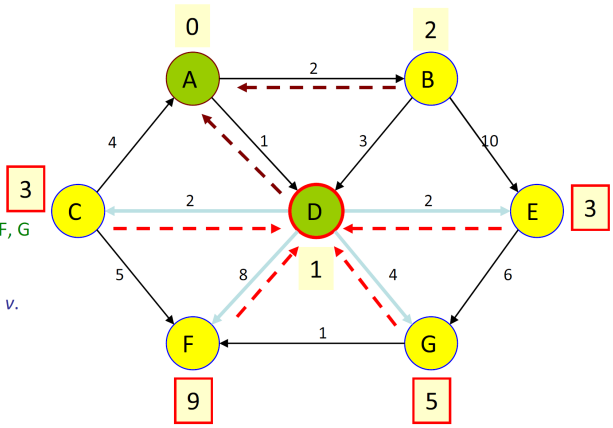
pqueue = {**D:1**, **B:2**}

Exempel

- 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.
```



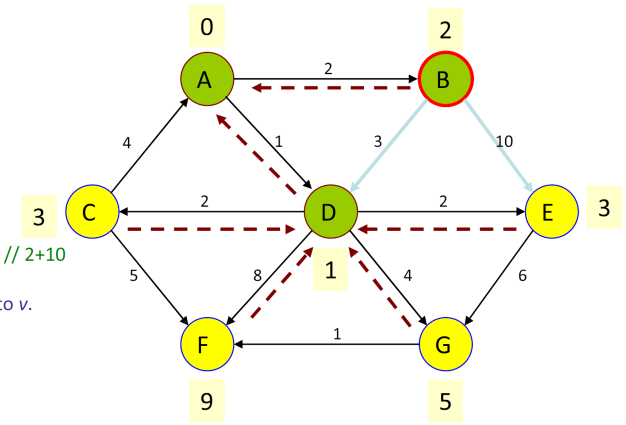
pqueue = {B:2, **C:3**, **E:3**, **G:5**, **F:9**}

Exempel

- 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.  // B
    mark v as visited.
    if v is v2, 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 v2 back to v1,
   following previous pointers.
```



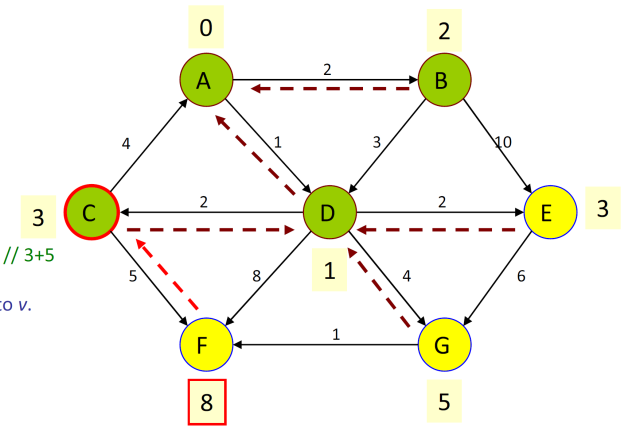
pqueue = {C:3, E:3, G:5, F:9}

Exempel

- 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.  // C
    mark v as visited.
    if v is v2, 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 v2 back to v1,
   following previous pointers.
```



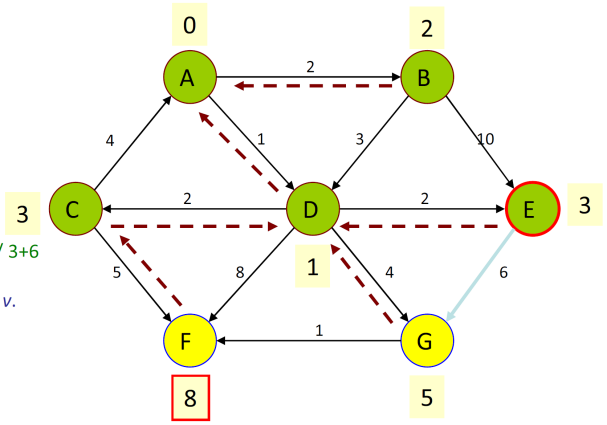
pqueue = {E:3, G:5, **F:8**}

Exempel

- 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.  // E
    mark v as visited.
    if v is v2, 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 v2 back to v1,
   following previous pointers.
```



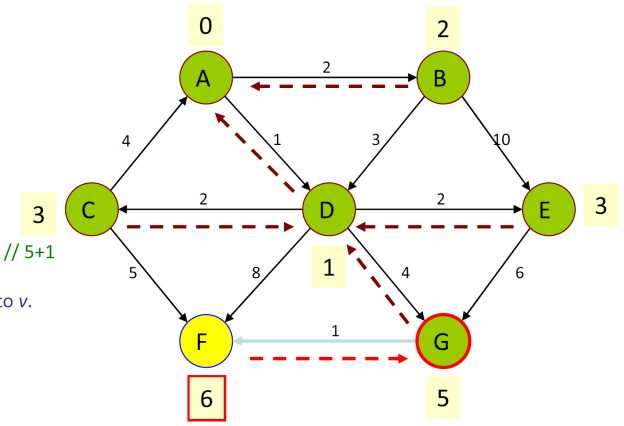
pqueue = {G:5, F:8}

Exempel

- 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.  // G
    mark v as visited.
    if v is v2, 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 v2 back to v1,
   following previous pointers.
```



pqueue = {**F:6**}

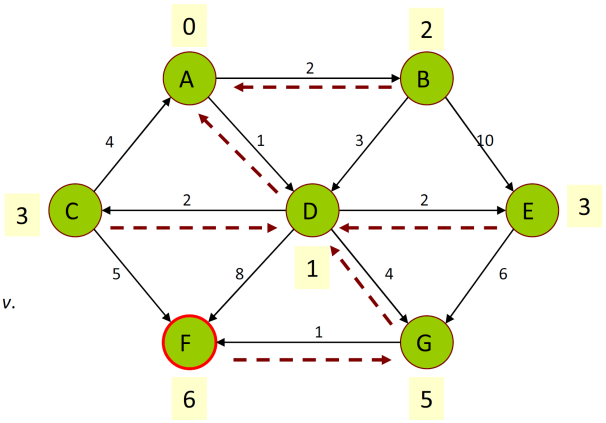
Exempel

- 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 = {}

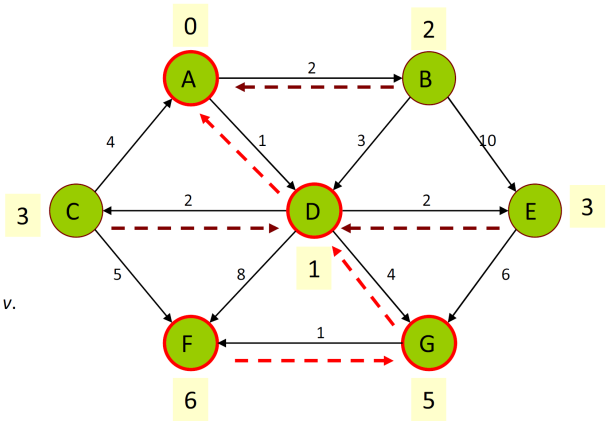
Exempel

- 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}
```



23.63

Analys av Dijkstras algoritm

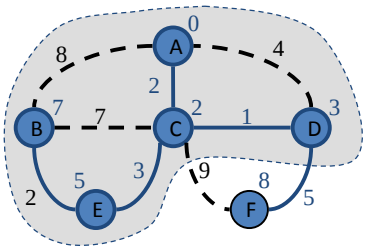
- Grafoperationer
  - Vi anropar incidentEdges en gång för varje nod
- Märkningsoperationer
  - Vi hämtar/sätter avstånd och lokator för nod z $O(deg(z))$ gånger
  - Att sätta/hämta en märkning tar tid $O(1)$
- Prio-köoperationer
  - Varje nod sätts in en gång och tas ut en gång från prio-kön, där varje insättning och borttagning tar tid $O(\log n)$
  - En nods nyckel i prio-kön ändras som mest $deg(w)$ gånger, där varje nyckeländring tar tid $O(\log n)$
- Dijkstras algoritm har exekveringstid $O((n+m)\log n)$ givet att grafen representeras med en grannlista
  - Kom ihåg att $\sum_v deg(v) = 2m$
- Exekveringstiden kan också uttryckas som $O(m \log n)$ eftersom vi antagit att grafen är sammanhängande

23.64

Varför Dijkstras algoritm fungerar

Dijkstras algoritm är baserad på den giriga metoden. Algoritmen lägger till noderna efter ökande avstånd.

- Antag att algoritmen inte hittade alla kortaste avstånd. Låt F vara den första felaktiga noden som behandlas.
- När den föregående noden, D, längs den sanna kortaste vägen behandlades var dess avstånd korrekt.
- Men bågen (D,F) *relaxerades* i det steget!
- Mao, så länge $d(F) \geq d(D)$ kan inte avståndet till F bli fel. Dvs, ingen nod får fel avstånd.

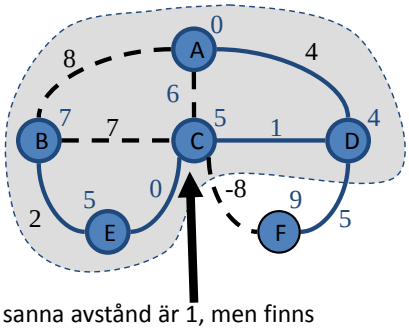


23.65

Varför Dijkstras algoritm inte fungerar med negativa bågvikter

Dijkstras algoritm är baserad på den giriga metoden. Algoritmen lägger till noderna efter ökande avstånd.

- Om en nod med en negativ incident båge skulle läggas till sent i molnet skulle den förstöra avståndet till noder som redan finns i molnet.

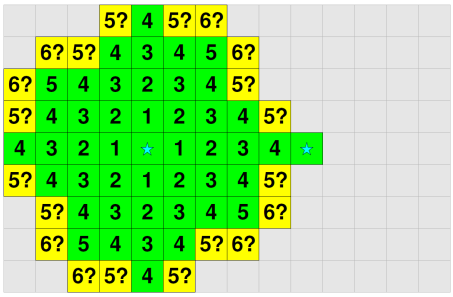


C's sanna avstånd är 1, men finns redan i molnet med d(C)=5!

23.66

Observationer

- Dijkstras algoritm fungerar genom att inkrementellt beräkna kortaste vägen till mellanliggande noder som eventuellt kan vara användbara.
  - De flesta av dessa stigar är i fel riktning.



- Algoritmen har ingen övergripande uppfattning om hur målet ska nås; algoritmen utforskar utåt i alla riktningar.
  - Kan vi tipsa algoritmen? Utforska i smartare ordning?

23.67

Heuristiker

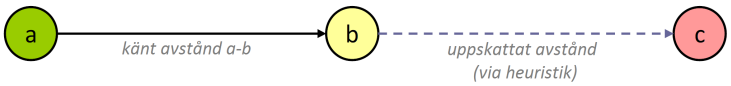
- heuristik: Spekulation, estimering eller kvalificerad gissning som avgör hur sökningen efter en lösning till ett problem ska gå till.
  - Exempel: Uppskatta avståndet mellan två platser i en Google Maps-graf till längden av en rät linje mellan punkterna.

- giltig heuristik: En som inte överskattar avståndet.
  - Ok om heuristiken ibland underskattar avståndet (till exempel Google Maps)

23.68

A*-algoritmen

- A*("A-stjärna): En modifierad version av Dijkstras algoritm som använder en heuristikfunktion för att vägleda utforskandet av sökrymden.



- Antag att vi letar efter vägar från startnod a till c
  - Varje mellanliggande nod b har två kostnader:
  - Den kända (exakta) kostnaden från startnoden a till b
  - Den heuristiska (uppskattade) kostnaden från b till slutnoden c.

- Idé: Kör Dijkstras algoritm, men använd följande prioritet i prioritetskön:
  - priority(b) = cost(a, b) + Heuristic(b, c)
  - Väljer att utforska vägar med lägre uppskattad kostnad

23.69

Exempel: Labyrintheuristik

- En möjlig heuristik för att söka efter vägar i en labyrint:
  - H(p_1, p_2) = abs(p_1.x - p_2.x) + abs(p_1.y - p_2.y) // dx + dy
  - Idén: Utforska grannar med lågt värde på (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

23.70

Kom ihåg: pseudokod för Dijkstras algoritm

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.

23.71

Pseudokod för A*-algoritmen

function **astar**(v_1, v_2):
  initialize every vertex to have a cost of infinity.
  set v_1's cost to 0.
  pqueue := {v_1, at priority **H(v_1, v_2)**}.

  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* + **H(*n*, v_2)**),
        or update its priority to be (*cost* + **H(*n*, v_2)**) if it was already in the *pqueue*.

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

Lägg märke till att nodernas *prioriteter* påverkas av heuristiken, men inte deras *kostnader*.

23.72