Parallella grafalgoritmer

Grama et. al.
Introduction to Parallel Computing, kapitel 10
Mikael Rännar efter material av
Robert Granat, Isak Jonsson och Erik Elmroth
8/5 2009

Översikt

- Grafer - definitioner, egenskaper, representation
- Minimalt uppspännande träd
  - Prims algoritm
- Kortaste vägen (1-till-alla)
  - Dijkstras algoritm
- Kortaste vägen (alla par)
  - Algoritm baserad på matrismultiplikation
  - Dijkstras algoritm
    - Source partitioned
    - Source parallel
  - Floyds algoritm
- Transitiv omslutning
- Sammanhängande komponenter

Grafer - definitioner

Oriktad graf:

- G(V, E): V är mängden av hörn och E är mängden av kanter
- En kant e $\in$ E är ett oordnat par (u, v) där u,v $\in$ V

Riktad graf:

- En kant e $\in$ E är ett ordnat par (u, v) (från u till v) där u,v $\in$ V
- **En stig** (väg, path) från u till v är en sekvens (u, ..., v) av hörn där konsekutiva hörn i sekvensen motsvaras av en kant i grafen
- Enkel (simple) stig: alla hörn i stigen är distinkta
- Cykel: u = v
- Acyklisk: innehåller inga cykler

Exempel på grafer

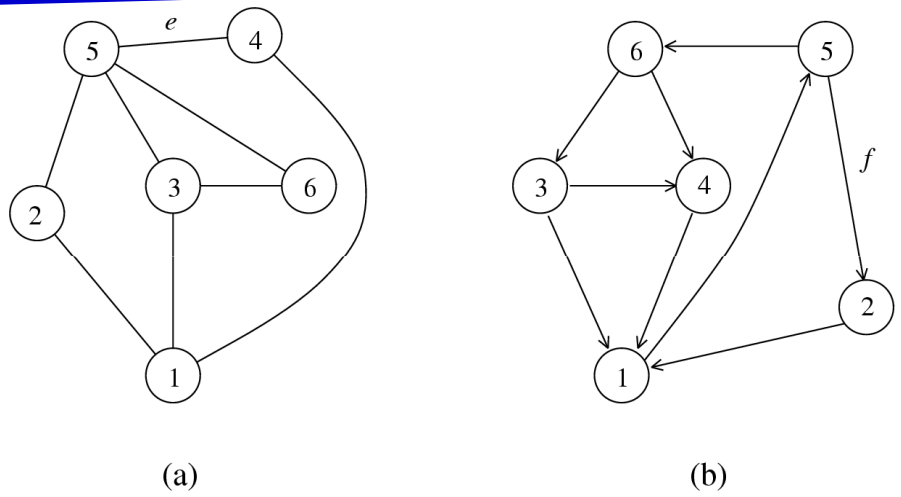


Figure 7.1 (a) An undirected graph and (b) a directed graph.

Grafer - egenskaper

- En graf är *sammanhängande* (connected) om det existerar en stig mellan varje par av hörn
- En graf är *komplett* om det existerar en kant mellan varje par av hörn
- G'(V', E') är en *delgraf* av G(V, E) om V' ∈ V och E' ∈ E
- Ett *träd* är en sammanhängande acyklisk graf
- En *skog* består av flera träd
- En graf G(V, E) är gles om |E| är mycket mindre än O(|V|²)
  - Svarar mot en gles Adj-matris (se nedan)

Viktade grafer:

- G(V, E, w), där w är en reellvärd funktion definierad på E (varje existerande kant har ett värde)
- Grafens vikt är summan av dessa kanters vikter

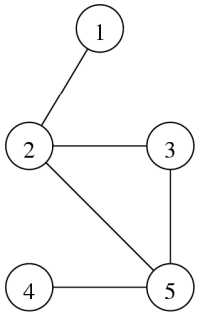
Matrisrepresentation av graf

Icke viktad graf

$$a_{i,j} = \begin{cases} 1 & \text{om } (v_i, v_j) \in E \\ 0 & \text{annars} \end{cases}$$

Viktad graf

$$a_{i,j} = \begin{cases} w(v_i, v_j) & \text{om } (v_i, v_j) \in E \\ 0 & \text{om } i = j \\ \infty & \text{annars} \end{cases}$$



$$A = \begin{bmatrix} 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 1 \\ 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 1 & 1 & 1 & 0 \end{bmatrix}$$

Lämplig för täta grafer

Figure 7.2 An undirected graph and its adjacency matrix representation.

Listrepresentation av graf

- G(V, E) representeras av listan Adj[1..|V|] av listor
- För varje v ∈V är Adj[v] en länkad lista av alla hörn som har en kant gemensam med v

Lämplig för glesa grafer

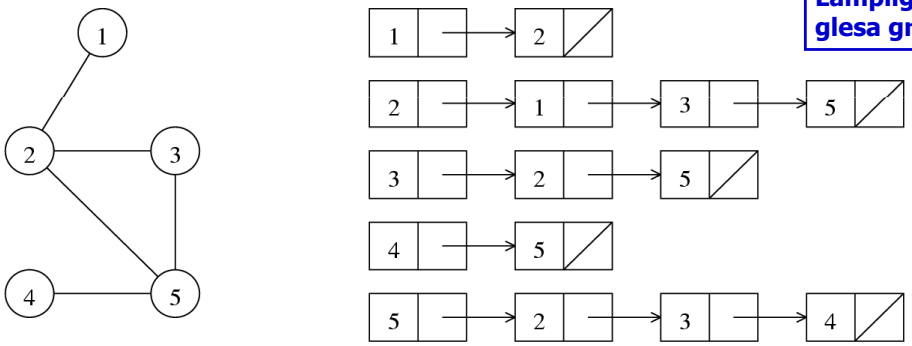


Figure 7.3 An undirected graph and its adjacency list representation.

Minimalt uppspännande träd (MST)

- Ett **uppspännande** träd innehåller grafens alla hörn
- MST för viktad graf är det uppspännande träd som har lägst vikt
- Om G ej är sammanhängande så saknar den MST (har istället minimal uppspännande skog)
- Antag fortsättningsvis att G är sammanhängande (om inte så kan vi hitta kopplade komponenter och sedan finna deras respektive MST)

Prims algoritm för minimalt uppspännande träd

- Start i godtyckligt hörn u
- Väljer hörn v så att kanten (u, v) garanterat finns med i MST
- Låt A = (a_{ij}) vara matrisrepresentationen av G = (V, E, w)
- Låt V_T vara mängd av funna hörn i MST
- Låt d[1..n] vara en vektor där d[v] för varje v ∈ (V - V_T) är vikten för den kant med minst vikt från något hörn i V_T till v
- För varje iteration väljs nytt hörn v som det vars d[v] är minimalt

Upprepas tills alla hörn är inkluderade

Kostnad:

- While-loopen exekveras n-1 gånger
- min-operationen (rad 10) och for-loopen (rad 12-13) kräver vardera O(n) steg

Totalt krävs alltså: $\Theta(n^2)$ steg

Prims algoritm

```
1.  procedure PRIM_MST(V, E, w, r)
2.  begin
3.      V_T := {r};
4.      d[r] := 0;
5.      for all v ∈ (V − V_T) do
6.          if edge (r, v) exists set d[v] := w(r, v);
7.          else set d[v] := ∞;
8.      while V_T ≠ V do
9.      begin
10.         find a vertex u such that d[u] = min{d[v]|v ∈ (V − V_T)};
11.         V_T := V_T ∪ {u};
12.         for all v ∈ (V − V_T) do
13.             d[v] = min{d[v], w(u, v)};
14.     endwhile
15. end PRIM_MST
```

Initiering: hörn med kant till r (starthörn) ges kostnad = kantens vikt. Övriga hörn ges kostnad = inf

Av alla hörn utanför V_T med kant till V_T väljs det vars kant har lägst vikt

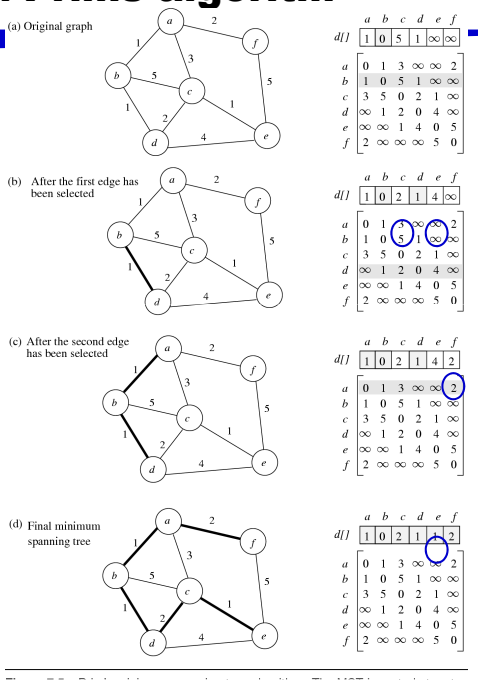
Inkludera hörnet

Uppdatera d-värden

Program 7.1 Prim's sequential minimum spanning tree algorithm.

Exempel på Prims algoritm

Startnod: b



Parallellisering av Prims algoritm

- d[v] uppdateras för alla v i varje iteration
  → kan ej välja 2 hörn samtidigt
  → kan ej parallellisera while-loopen
  Vi parallelliserar istället for-loopen!
- Varje processor håller en block-kolumn (n/p kolumner) av A och motsvarande del av d
- V_i är den delmängd hörn som hålls av P_i
- Varje processor beräknar d[u] för sina hörn
- Globalt minimum för d[u] beräknas genom alla-till-en-reduktion
- Processorn som håller globala minimat broadcastar nya hörnet u
- Processor ansvarig för u markerar att u tillhör V_T och alla uppdaterar d[v] för sina lokala hörn

Matrispartitionering för Prims algoritm

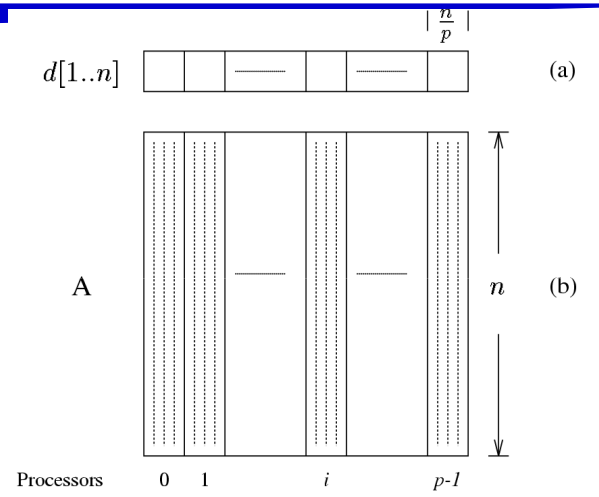


Figure 7.6 The partitioning of the distance array d and the adjacency matrix A among p processors.

Analys av parallella Prims algoritm

- Eftersom alla skall beräkna d[v] för sina hörn krävs att de har hela kolumnerna för sina hörn i Adj
- Kostnad för att uppdatera d-värden för varje processor är $\Theta(n/p)$
- Komm.kostn.: reduktion+broadcast
- Totalt Tp = $\Theta(n^2/p) + \Theta(n \log p)$

I varje iteration

Uppdatering:
...
for all v ∈ (V - V_T) that I own
  d[v] = min{d[v], w(u, v)}
end

Kortaste vägen (från 1 till alla)

Dijkstras algoritm: (från startnod s)

- I grunden som Prims algoritm men..
- Istället för att lagra d[u] lagras l[u] som är totala viktsumman från s till u

Parallell Dijkstras algoritm som parallell Prims med ovanstående modifiering. Analysen blir identisk!

Kortaste vägen (alla par)

- Finn kortaste vägen mellan alla par av hörn
- Resultatet är en n x n-matris D = d_{ij}, där d_{ij} är kortaste vägen från hörn v_i till hörn v_j

Algoritm baserad på matris-multiplikation

- Låt G = (V, E, w) representeras av matrisen A
- Låt d_{ij}^k representera den kortaste stigen från v_i till v_j som innehåller maximalt k kanter
- Låt v_m vara ett hörn i den stigen
- Då gäller att $d_{ij}^k = \min\{d_{im}^{k-1} + w(v_m, v_j)\}$ (där minimeringen sker över m från 1 till n och $d_{ij}^1 = a_{ij}$)
- Vi skapar en matris D för varje maximal stig-längd: $D^k = (d_{ij}^k)$
- Eftersom den kortaste vägen mellan två hörn alltid är maximalt n-1 så innehåller D^{n-1} alla pars kortaste vägar

Matrismultiplikationsalgoritm (forts)

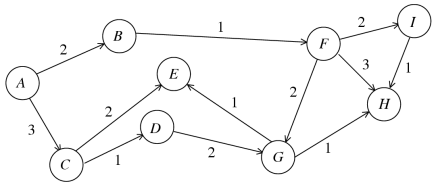
- D^k beräknas från D^{k-1} med en modifierad matrismultiplikation:

$$c_{ij} = \min_{k=1}^{k=n} a_{ik} + b_{kj}$$

(Hitta k som ger c_{ij} minimal)

- Eftersom $D^1 = A$ så är $D^k = A^k$ beräknad med den modifierade matrismultiplikationen
- Beräkna resultatet D_{n-1} genom att beräkna A^2, A^4, A^8,... A^{n-1}, med modifierad matrismultiplikation i log n steg
- Komplexitet $\Theta(n^3)$ för matrismultiplikation ger $\Theta(n^3 \log n)$ totalt
- Parallellisering: samma parallella algoritmer som i vanlig matrismultiplikation, t.ex. på $\Theta(\log n)$ tid med DNS-algoritmen

Matrismultiplikations-algoritm (exempel)



$$A^1 = \begin{pmatrix} 0 & 2 & 3 & \infty & \infty & \infty & \infty & \infty & \infty \\ \infty & 0 & \infty & \infty & \infty & 1 & \infty & \infty & \infty \\ \infty & \infty & 0 & 1 & 2 & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & 0 & \infty & \infty & 2 & \infty & \infty \\ \infty & \infty & \infty & \infty & 0 & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty & 0 & 2 & 3 & 2 \\ \infty & \infty & \infty & \infty & 1 & \infty & 0 & 1 & \infty \\ \infty & \infty & \infty & \infty & \infty & \infty & \infty & 0 & \infty \\ \infty & \infty & \infty & \infty & \infty & \infty & \infty & 1 & 0 \end{pmatrix} \quad A^2 = \begin{pmatrix} 0 & 2 & 3 & 4 & 5 & 3 & \infty & \infty & \infty \\ \infty & 0 & \infty & \infty & \infty & 1 & 3 & 4 & 3 \\ \infty & \infty & 0 & 1 & 2 & \infty & 3 & \infty & \infty \\ \infty & \infty & \infty & 0 & 3 & \infty & 2 & 3 & \infty \\ \infty & \infty & \infty & \infty & 0 & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & 3 & 0 & 2 & 3 & 2 \\ \infty & \infty & \infty & \infty & 1 & \infty & 0 & 1 & \infty \\ \infty & \infty & \infty & \infty & \infty & \infty & \infty & 0 & \infty \\ \infty & \infty & \infty & \infty & \infty & \infty & \infty & 1 & 0 \end{pmatrix}$$

$$A^4 = \begin{pmatrix} 0 & 2 & 3 & 4 & 5 & 3 & 5 & 6 & 5 \\ \infty & 0 & \infty & \infty & 4 & 1 & 3 & 4 & 3 \\ \infty & \infty & 0 & 1 & 2 & \infty & 3 & 4 & \infty \\ \infty & \infty & \infty & 0 & 3 & \infty & 2 & 3 & \infty \\ \infty & \infty & \infty & \infty & 0 & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & 3 & 0 & 2 & 3 & 2 \\ \infty & \infty & \infty & \infty & 1 & \infty & 0 & 1 & \infty \\ \infty & \infty & \infty & \infty & \infty & \infty & \infty & 0 & \infty \\ \infty & \infty & \infty & \infty & \infty & \infty & \infty & 1 & 0 \end{pmatrix} \quad A^8 = \begin{pmatrix} 0 & 2 & 3 & 4 & 5 & 3 & 5 & 6 & 5 \\ \infty & 0 & \infty & \infty & 4 & 1 & 3 & 4 & 3 \\ \infty & \infty & 0 & 1 & 2 & \infty & 3 & 4 & \infty \\ \infty & \infty & \infty & 0 & 3 & \infty & 2 & 3 & \infty \\ \infty & \infty & \infty & \infty & 0 & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & 3 & 0 & 2 & 3 & 2 \\ \infty & \infty & \infty & \infty & 1 & \infty & 0 & 1 & \infty \\ \infty & \infty & \infty & \infty & \infty & \infty & \infty & 0 & \infty \\ \infty & \infty & \infty & \infty & \infty & \infty & \infty & 1 & 0 \end{pmatrix}$$

Figure 7.7 An example of the matrix-multiplication-based all-pairs shortest paths algorithm.

Dijkstras algoritm för kortaste vägen: alla par

- Problemet löses för alla par genom att applicera 1-hörns-algoritmen på varje hörn i grafen
- 1-hörns-algoritmens komplexitet $\Theta(n^2)$ ger $\Theta(n^3)$ för alla par
- 2 parallella varianter - source partitioned & source parallel

Source-partitioned

- Varje processor tilldelas ett hörn och löser 1-hörnsproblemet sekventiellt för det hörnet
- Kräver ingen kommunikation
- Ur kommunikationssynpunkt optimal, men kan endast nyttja n processorer
- Kräver också p ggr så mycket minne!

Dijkstras, alla par (variant 2)

Source-parallel

- Processorerna delas in i n partitioner med vardera p/n processorer (p > n)
- Varje partition löser 1-hörnsproblemet för 1 hörn med den parallella 1-hörnsalgoritmen → 2 Nivåer av parallellitet:
  - Grovkorning: Varje nod i grafen behandlas oberoende av de andra
  - Finkorning: p/n processorer delar på arbetet med ett hörn

På 2-dimensionellt nät:

- $p^{1/2}$ x $p^{1/2}$ nät delas in i partitioner om vardera $(p/n)^{1/2}$ x $(p/n)^{1/2}$ processorer
- Ingen kommunikation mellan processorgrupperna
- Den parallella 1-hörnsalgoritmen för 2-dimensionellt nät inom varje grupp

Kortaste vägen (alla par): Floyds algoritm

- Låt V = {v_1, v_2, ..., v_n } (alla hörn i G)
  låt V^k = {v_1, v_2, ..., v_k}, k ≤ n vara en delmängd av V.
- För varje par v_i, v_j ∈ V, betraktar vi alla stigar vars mellanliggande hörn tillhör V^k.
- Låt p_{ij}^k vara den kortaste av dessa stigar (med vikt d_{ij}^k)
- Om hörnet v_k ej ingår i stigen är p_{ij}^k samma som p_{ij}^{k-1}
- Om v_k ingår i stigen så kan stigen delas i två stigar: En från v_1 till v_k och en från v_k till v_j där var och en av stigarna går över hörn i V^{k-1} = {v_1, v_2, ..., v_{k-1}}
- I det fallet är vikten av stigen $d_{ij}^k = d_{ij}^{k-1} + d_{kj}^{k-1}$

Floyds algoritm (forts...)

Sammanfattningsvis får vi rekursionen:

$$d_{ij}^k = \begin{cases} w(v_i, v_j) & \text{om } k = 0 \\ \min\{d_{ij}^{k-1}, d_{ik}^{k-1} + d_{kj}^{k-1}\} & \text{om } k \geq 1 \end{cases}$$

och algoritmen:

```
1.  procedure FLOYD_ALL_PAIRS_SP(A)
2.  begin
3.      D^(0) = A;
4.      for k = 1 to n do
5.          for i = 1 to n do
6.              for j = 1 to n do
7.                  d_{i,j}^(k) := min(d_{i,j}^(k-1), d_{i,k}^(k-1) + d_{k,j}^(k-1));
8.  end FLOYD_ALL_PAIRS_SP
```

Program 7.3 Floyd's all-pairs shortest paths algorithm. This program computes the all-pairs shortest paths of the graph $G = (V, E)$ with adjacency matrix A.

Parallell Floyds algoritm (block-schackbrädes...)

- D^k partitioneras i p block av storlek $(n/p^{1/2})$ x $(n/p^{1/2})$

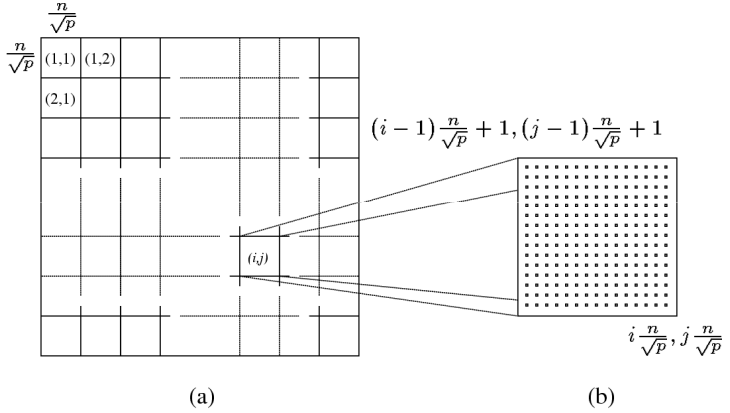


Figure 7.9 (a) Matrix $D^{(k)}$ partitioned by block checkerboarding into $\sqrt{p} \times \sqrt{p}$ subblocks, and (b) the square subblock of $D^{(k)}$ assigned to processor $P_{i,j}$.

Parallell Floyds algoritm (forts)

- I iteration k behöver processor P_{ij} data från kolumn k och rad k i D^{k-1}-matrisen

Exempel: för att beräkna d_{lr}^{k} behöver den d_{lk}^{k-1} och d_{kr}^{k-1}

- I iteration k skickar alla p processorer som håller data i rad k dessa element till övriga processorer i samma processorkolumn

- Pss skickar processorer som håller element i kolumn k dessa element till övriga processorer i samma processorrad

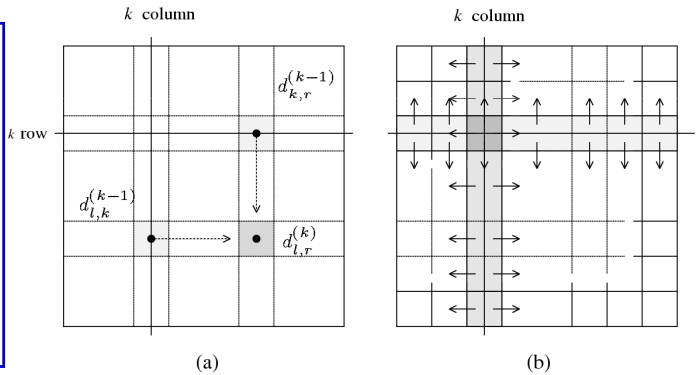


Figure 7.10 (a) Communication patterns used in the block-checkerboard partitioning. When computing $d_{i,j}^{(k)}$, information must be

Parallell Floyds algoritm på hyperkub

- 2D nätet avbildas på hyperkub så att varje processorrad och processorkolumn i nätet motsvarar en delkub

```
1.  procedure FLOYD_CHECKERBOARD(D^(0))
2.  begin
3.      for k = 1 to n do
4.      begin
5.          each processor P_{i,j} that has a segment of the k^th row of D^(k-1);
                broadcasts it to the P_{*,j} processors;
6.          each processor P_{i,j} that has a segment of the k^th column of D^(k-1);
                broadcasts it to the P_{i,*} processors;
7.          each processor waits to receive the needed segments;
8.          each processor P_{i,j} computes its part of the D^(k) matrix;
9.      end
10. end FLOYD_CHECKERBOARD
```

Program 7.4 Floyd's parallel formulation using the block-checkerboard partitioning. $P_{*,j}$

Transitiv omslutning

- Om G = (V, E) är en graf så är dess transitiva omslutning en graf $G^* = (V, E^*)$, där $E^* = \{(v_i, v_j) |$ existerar en stig från v_i till v_j i G$\}$
- Beräkna connectivity-matrisen A^* så att
  $a_{ij} = 1$ om i = j eller stig från v_i till v_j existerar och
  $a_{ij} = \infty$ annars.

Metod 1:

- Sätt vikter i G till 1 och beräkna kortaste vägen mellan alla par. Tolka resultatet D så att

  $d_{ij} = \infty \quad \rightarrow \quad a_{ij} = \infty$ och
  i = j eller $d_{ij} > 0 \quad \rightarrow \quad a_{ij} = 1 \qquad (=> A^*)$

Metod 2:

- Modifiera Floyds algoritm genom att byta ut min och + på rad 7 till logiskt *eller* respektive logiskt *och*. $d_{ij}^{(k)} = d_{ij}^{(k-1)}$ OR ($d_{ik}^{(k-1)}$ AND $d_{kj}^{(k-1)}$)

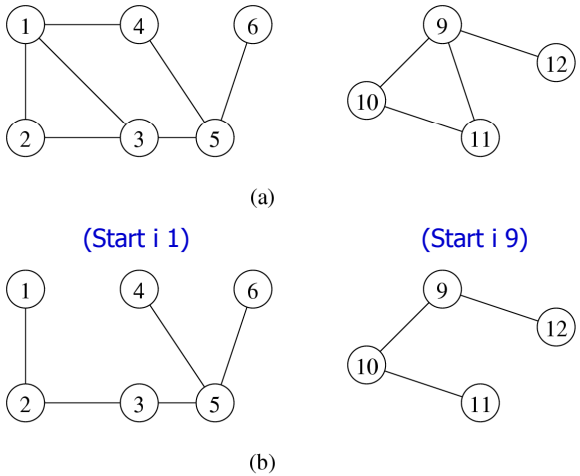
Sammanhängande komponenter

- De sammanhängande komponenterna av G = (V, E) = maximala disjunkta mängden C_1, C_2, ..., C_k sådan att V = C_1 U C_2 U ... U C_k och u, v ∈ C_i om och endast om det finns en stig mellan u och v

Djupet-först-baserad algoritm:

- Ger en skog av djupet-först-träd

- Varje träd innehåller endast komponenter som ej tillhör något annat träd



// algoritm för sammanhängande komponenter

Alla processorer har alla hörn men inte alla kanter

- Låt G = (V, E) representeras av matrisen A
- Fördela A med 1 del på varje processor
  → varje processor håller delgraf $G_i = (V, E_i)$

Steg 1:

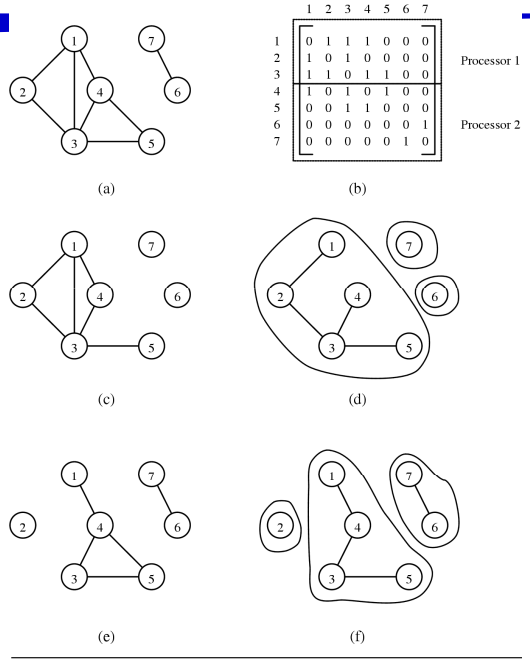
- Alla beräknar djupet-först uppspännande skog för sin delgraf

Steg 2:

- Alla uppspännande skogar slås parvis samman till en skog:

  Givet uppspännande skogar A och B görs följande för varje kant (u, v) i A:
  - Om hörnen u och v finns i samma träd i B görs inget
  - Annars slås B:s träd med u ihop med B:s träd med v

// algoritm för sammanhängande komponenter



Algoritmer för glesa grafer

- G = (V, E) är gles om $|E| << |V|^2$
- Algoritmer för täta grafer fungerar naturligtvis även för glesa men de är ofta ineffektiva
  - Exempelvis tar Prims algoritm för minimalt uppspännande träd $\Theta(n^2)$ tid, oavsett antal kanter
  - Med listrepresentation kan algoritmen modifieras till att ta O(|E| log n) (är effektivare för $|E| = O(n^2/\log n)$)
- Komplexitet för algoritm med matrislagring är normalt $W(n^2)$ medan den för listrepresentation normalt är W(n + |E|)

Arbetsfördelning

Matrisrepresentation

- Matrisen fördelas jämnt på processorerna
  - Jämn arbetsmängd
  - Lite kommunikation (varje processor håller konsekutiva rader / kolumner)

Listrepresentation

- Antalet listor fördelas jämnt på proc.
  - Kan ge ojämn arbetsmängd då listorna ofta är olika långa
- Antalet kanter fördelas jämnt på proc.
  - Kan kräva att listor delas mellan processorer ==> ökad kommunikation

→

- Svårt att göra bra algoritmer för generella glesa grafer
- Speciella algoritmer för olika typer av glesa grafer
- Glesa grafer med nätstruktur (grid graphs)

Kortaste vägen (1-till-alla) Johnsons algoritm

Repetition av Dijkstras algoritm

- hittar hörn u $\in$ (V - V_T) sådant att
  l[u] = min{l[v] | v $\in$ (V - V_T)} och sätter in i V_T
- För varje v $\in$ (V - V_T) beräknas
  l[v] = min{l[v] | l[u] + w(u,v)}

Modifiering till Johnsons algoritm

- Prioritetskö Q för alla v $\in$ (V - V_T) sorterad efter storlek på l[v] (minst först)
- Initialt är alla l[v] = ∞, förutom starthörnet s som har l[s] = 0
- För varje steg tas hörnet u med minsta värdet l[u] bort ur kön, u:s lista traverseras och för varje kant (u, v) uppdateras l[v]. (Bara hörn i samma lista behöver genomsökas.) Därefter sorteras kön.

Johnsons algoritm

```
1.  procedure JOHNSON_SINGLE_SOURCE_SP(V, E, s)
2.  begin
3.      Q := V;
4.      for all v ∈ Q do
5.          l[v] := ∞;
6.      l[s] := 0;
7.      while Q ≠ ∅ do
8.      begin
9.          u := extract_min(Q);
10.         for each v ∈ Adj[u] do
11.             if v ∈ Q and l[u] + w(u, v) < l[v] then
12.                 l[v] := l[u] + w(u, v);
13.     endwhile
14. end JOHNSON_SINGLE_SOURCE_SP
```

Program 7.5 Johnson's sequential single-source shortest paths algorithm.

Parallell Johnsons (central köhantering)

- 1 processor håller prioritetskön
- Alla andra uppdaterar l[v] för v $\in$ (V - V_T) och skickar nya värden till processorn med prioritetskön
- I varje iteration uppdateras grovt räknat
  |E|/|V| hörn ==> maximalt |E|/|V| proc. kan ha arbete
- För varje ny kant uppdateras kön på
  O(log n) tid (ta bort gammalt l[v] och sätta in nytt värde)
- För totalt |E| kanter krävs O(|E| log n) tid
- Samma storleksordning som för sekventiell algoritm

Parallell Johnsons (distribuerad köhantering)

- Fördela V på processorena i p disjunkta mängder så att P_i håller V_i
- Varje processor
  - håller prioritetskö Q_i för de egna hörnen
  - har en vektor sp, där slutligen sp[v] skall vara kostnaden för kortaste vägen från s till v
  - exekverar Johnsons algoritm på den egna delgrafen
- Initialt är l[v] = ∞ för alla hörn utom s
  (l[s] = 0)
- Varje gång ett hörn v tas bort ur kön sätts sp[v] = l[v]

Beroenden i Johnsons algoritm

- Om u ∈ V_i och v ∈ V_j då P_i tar bort u ur Q_i så skickar P_i information till P_j om att l[v] eventuellt kan få det nya värdet l[u] + w(u,v)
- P_j sätter l[v] = min{l[v], l[u] + w(u,v)}
- Eftersom P_j också exekverar Johnsons algoritm kan P_j redan ha tagit bort v ur kön Q_j. Då kan vi ha två fall:
  - Om l[u] + w(u,w) ≥ sp[v] så är den tidigare funna vägen kortare. P_j vidtar ingen åtgärd.
  - Om sp[v] > l[u] + w(u,w) så är vägen via u kortare än den hittills kortaste funna vägen.
    P_j sätter l[v] = l[u] + w(u,w), tar bort värdet på sp[v] samt sätter tillbaka v i Q_j.
- Algoritmen avslutas först när alla köer är tomma!
- Notera att pga att den distribuerade köhanteringen inte alltid tar bort värden ur kön i "rätt" ordning så utförs "onödigt" extraarbete av den parallella algoritmen.

Exempel på extraarbete

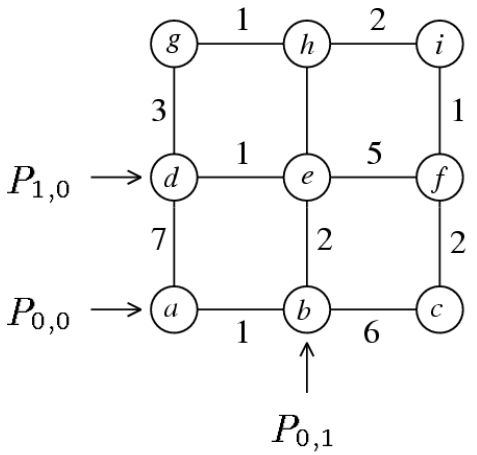


Figure 7.17 A grid graph.
Copyright (r) 1994 Ben-

Våg av aktivitet i prioritetsköerna

- Initialt har bara processorn med starthörnet en icke-tom kö
- Vågfront av aktivitet i köerna
- Processor är utan arbete innan vågfronten kommer och efter den har passerat

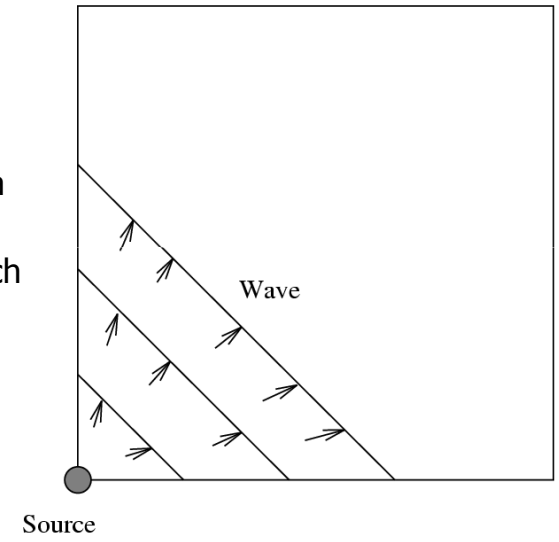


Figure 7.18 The wave of activity

Block-schackbrädes-partitionering

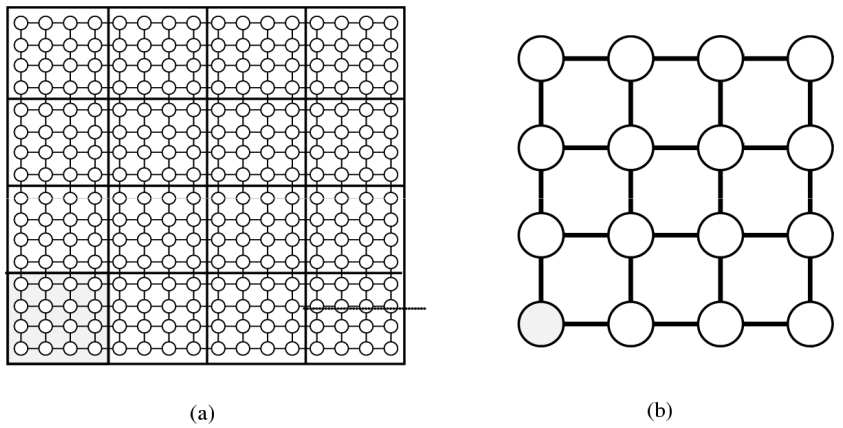


Figure 7.19 Mapping the grid graph (a) onto a mesh (b) by using the block-checkerboard mapping. In this example, $n = 16$ and $\sqrt{p} = 4$. The shaded vertices are mapped onto the shaded processor.

Cyklisk schackbrädes-partitionering

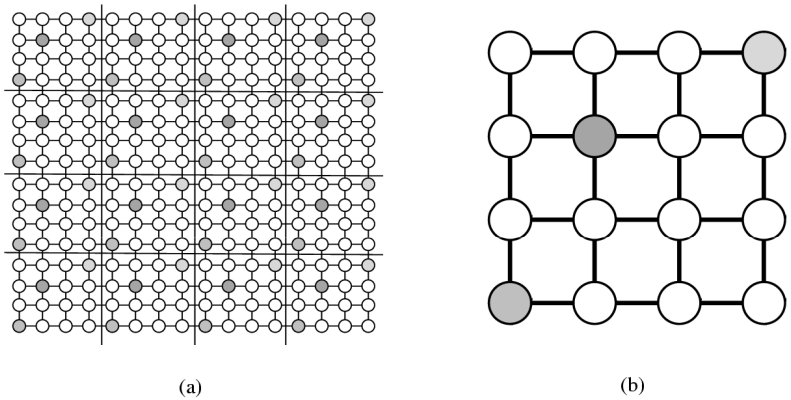


Figure 7.20 Mapping the grid graph (a) onto a mesh (b) by using the cyclic-checkerboard mapping. In this example, $n = 16$ and $\sqrt{p} = 4$. The shaded graph vertices are mapped onto the correspondingly shaded mesh processors.

Blockkolumns-partitionering

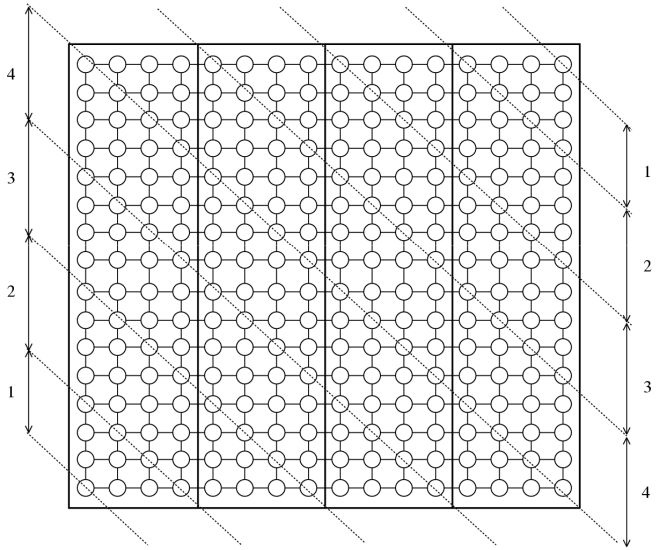


Figure 7.22 The number of busy processors as the computational wave propagates across the grid graph.