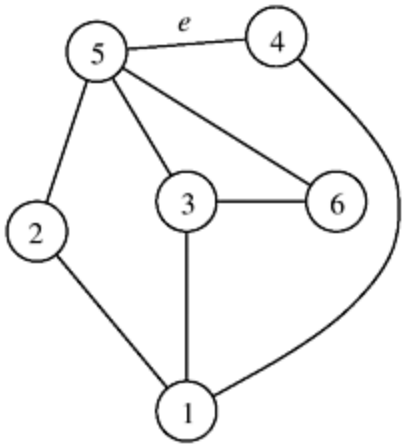
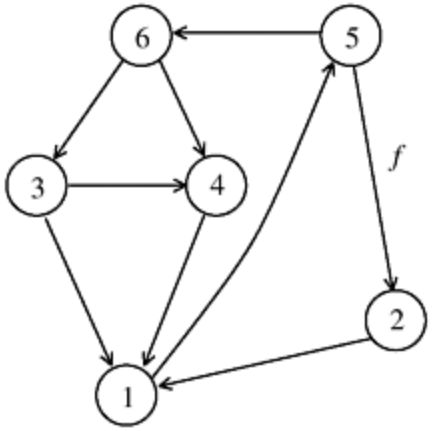
# Αλγόριθμοι Γραφημάτων

# Ορισμοί

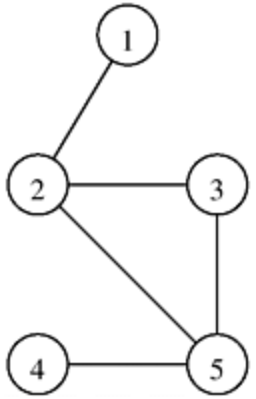



Μη κατευθυνόμενο γράφημα



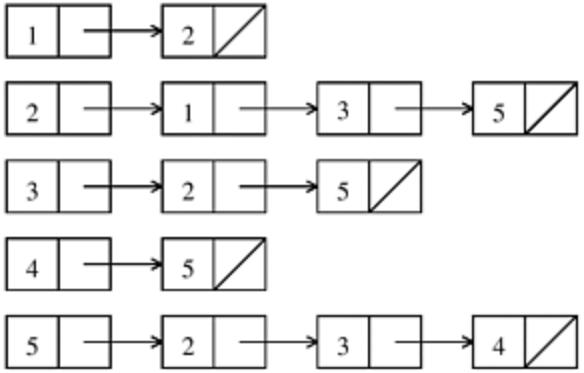

Κατευθυνόμενο γράφημα

- Ένα μη κατευθυνόμενο γράφημα *G* είναι ένα ζεύγος *(V,E)*, όπου *V* είναι ένα πεπερασμένο σύνολο σημείων τα οποία καλούνται κορυφές και Ε είναι ένα πεπερασμένο σύνολο ακμών.
- Μία ακμή $e \in E$ είναι ένα μη διατεταγμένο ζεύγος *(u,v)*, όπου $u,v \in V$.
- Σε ένα κατευθυνόμενο γράφημα, οι ακμή *e* είναι ένα διατεταγμένο ζεύγος *(u,v)*.
- Ένα μονοπάτι από μία κορυφή *v* σε μία κορυφή *u* είναι μία ακολουθία $<v_0,v_1,v_2,\ldots,v_k>$ κορυφών όπου $v_0 = v$, $v_k = u$, και $(v_i, v_{i+1}) \in E$ για $l = 0, 1,\ldots, k-1$.
- Το μήκος ενός μονοπατιού ορίζεται ως το πλήθος των ακμών στο μονοπάτι.
- Ένα μη κατευθυνόμενο γράφημα είναι συνεκτικό αν κάθε ζεύγος κορυφών συνδέεται από ένα μονοπάτι.
- Δάσος είναι ένα άκυκλο γράφημα και δέντρο ένα συνεκτικό άκυκλο γράφημα.
- Ένα γράφημα όπου κάθε ακμή συσχετίζεται με ένα βάρος λέγεται βεβαρυμένο γράφημα.




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

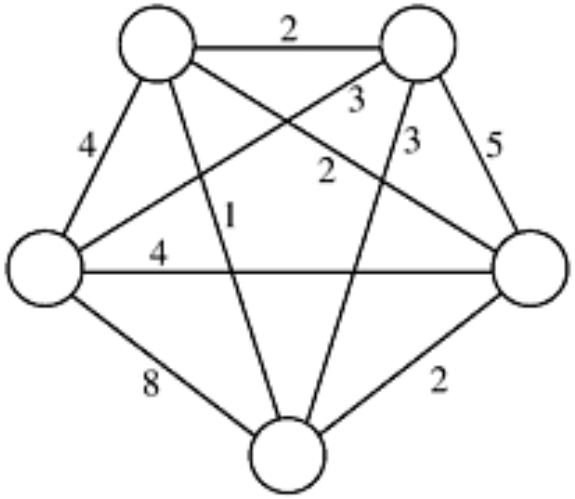
Πίνακας γειτνίασης

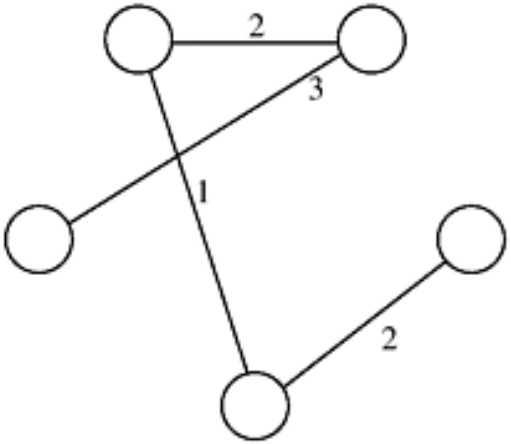



Λίστα γειτνίασης

- Τα γραφήματα μπορούν να αναπαρασταθούν με ένα πίνακα γειτνίασης ή με μία λίστα γειτνίασης.
- Ο πίνακας γειτνίασης έχει τιμή $a_{i,j} = 1$ αν οι κόμβοι $i$ και $j$ συνδέονται με μία ακμή, διαφορετικά η τιμή του στοιχείου είναι 0. Στη περίπτωση των βεβαρυμμένων γραφημάτων, $a_{i,j} = w_{i,j}$, δηλ. το βάρος της ακμής.
- Η λίστα γειτνίασης ενός γραφήματος $G = (V,E)$ είναι ένας πίνακας $Adj[1..|V|]$ λιστών. Κάθε λίστα $Adj[v]$ είναι η λίστα όλων των κορυφών που είναι γειτονικές με τη κορυφή $v$.
- Για ένα γράφημα με $n$ κορυφές, ο πίνακας γειτνίασης απαιτεί $\Theta(n^2)$ χώρο ενώ η λίστα γειτνίασης απαιτεί χώρο $\Theta(|E|)$.

# Ελάχιστο γεννητικό δέντρο



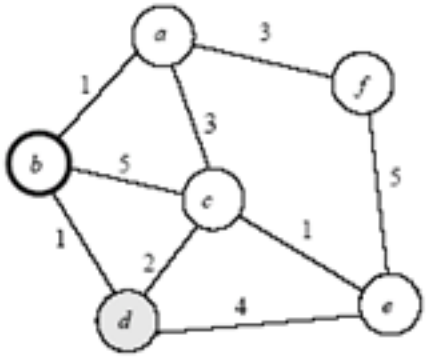

Το ελάχιστο γεννητικό δέντρο του αριστερού γραφήματος

- Γεννητικό δέντρο ενός μη κατευθυνόμενου γραφήματος *G* είναι ένα συνεκτικό άκυκλο υπογράφημα του *G* που περιέχει όλες τις κορυφές του G. Πιο απλά, είναι ένα δέντρο το οποίο περιλαμβάνει όλους τις κορυφές του G.
- Σε ένα βεβαρυμμένο γράφημα, το βάρος του υπογραφήματος είναι το άθροισμα των βαρών των ακμών του υπογραφήματος.
- Ελάχιστο Γεννητικό Δέντρο (ΕΓΔ) για ένα βεβαρυμμένο μη κατευθυνόμενο γράφημα είναι ένα γεννητικό δέντρο με ελάχιστο βάρος.

# Ελάχιστο γεννητικό δέντρο: Αλγόριθμος του Prim

- Ο αλγόριθμος του Prim ακολουθεί την τεχνική της απληστίας για την εύρεση του ελάχιστου γεννητικού δέντρου.
- Αρχικά, ο αλγόριθμος επιλέγει αυθαίρετα μία κορυφή του γραφήματος και την εισάγει στο αρχικά άδειο ΕΓΔ.
- Σε κάθε βήμα στο υπό κατασκευή ΕΓΔ εισάγεται η κορυφή εκείνη η οποία είναι πιο κοντά στις κορυφές που είναι ήδη στο ΕΓΔ.
- Στη συνέχεια ακολουθεί ένα παράδειγμα εφαρμογής τού αλγορίθμου Prim

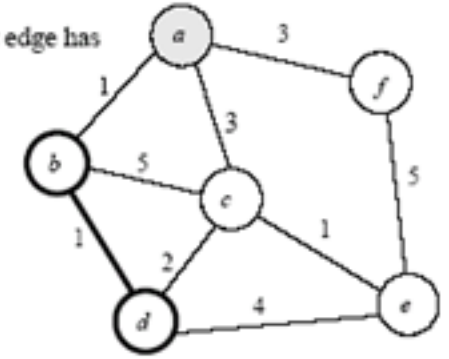
(a) Original graph




| | a | b | c | d | e | f |
|---|---|---|---|---|---|---|
| d[] | 1 | 0 | 5 | 1 | ∞ | ∞ |

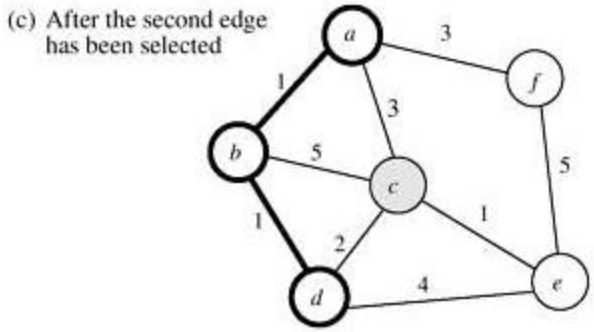
| | a | b | c | d | e | f |
|---|---|---|---|---|---|---|
| a | 0 | 1 | 3 | ∞ | ∞ | 3 |
| b | 1 | 0 | 5 | 1 | ∞ | ∞ |
| c | 3 | 5 | 0 | 2 | 1 | ∞ |
| d | ∞ | 1 | 2 | 0 | 4 | ∞ |
| e | ∞ | ∞ | 1 | 4 | 0 | 5 |
| f | 2 | ∞ | ∞ | ∞ | 5 | 0 |

(b) After the first edge has been selected



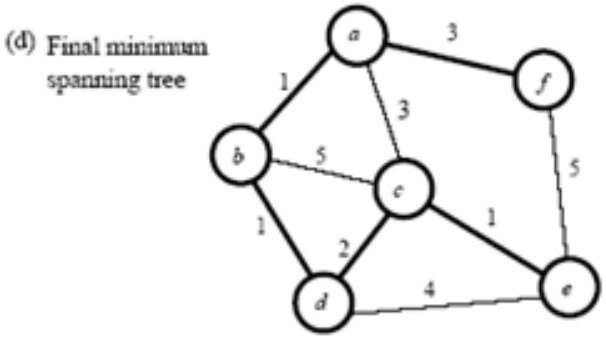

| | a | b | c | d | e | f |
|---|---|---|---|---|---|---|
| d[] | 1 | 0 | 2 | 1 | 4 | ∞ |

| | a | b | c | d | e | f |
|---|---|---|---|---|---|---|
| a | 0 | 1 | 3 | ∞ | ∞ | 3 |
| b | 1 | 0 | 5 | 1 | ∞ | ∞ |
| c | 3 | 5 | 0 | 2 | 1 | ∞ |
| d | ∞ | 1 | 2 | 0 | 4 | ∞ |
| e | ∞ | ∞ | 1 | 4 | 0 | 5 |
| f | 2 | ∞ | ∞ | ∞ | 5 | 0 |




| | a | b | c | d | e | f |
|---|---|---|---|---|---|---|
| d[] | 1 | 0 | 2 | 1 | 4 | 3 |

| | a | b | c | d | e | f |
|---|---|---|---|---|---|---|
| a | 0 | 1 | 3 | ∞ | ∞ | 3 |
| b | 1 | 0 | 5 | 1 | ∞ | ∞ |
| c | 3 | 5 | 0 | 2 | 1 | ∞ |
| d | ∞ | 1 | 2 | 0 | 4 | ∞ |
| e | ∞ | ∞ | 1 | 4 | 0 | 5 |
| f | 2 | ∞ | ∞ | ∞ | 5 | 0 |




| | a | b | c | d | e | f |
|---|---|---|---|---|---|---|
| d[] | 1 | 0 | 2 | 1 | 1 | 3 |

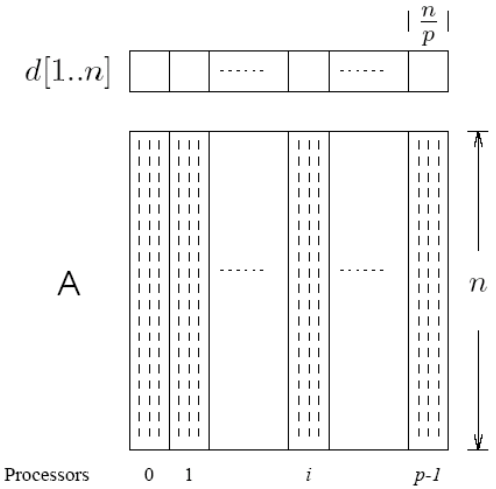
| | a | b | c | d | e | f |
|---|---|---|---|---|---|---|
| a | 0 | 1 | 3 | ∞ | ∞ | 3 |
| b | 1 | 0 | 5 | 1 | ∞ | ∞ |
| c | 3 | 5 | 0 | 2 | 1 | ∞ |
| d | ∞ | 1 | 2 | 0 | 4 | ∞ |
| e | ∞ | ∞ | 1 | 4 | 0 | 5 |
| f | 2 | ∞ | ∞ | ∞ | 5 | 0 |

Ο ψευδοκώδικας για τον ακολουθιακό αλγόριθμο του Prim δίνεται δεξιά:

```
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
```

# Παράλληλη υλοποίηση του αλγόριθμου **Prim**

- Θα περιγράψουμε την παράλληλη υλοποίηση του αλγόριθμου του Prim σε σύστημα κατανεμημένης μνήμης. Εύκολα, ο παράλληλος αλγόριθμος μπορεί να προσαρμοσθεί και σε συστήματα διαμοιραζόμενης μνήμης
- Οι επαναλήψεις του while-loop δεν μπορούν εύκολα να εκτελεστούν παράλληλα, αφού οι υπολογισμοί που εκτελούνται σε μία επανάληψη εξαρτώνται από τα αποτελέσματα των προηγουμένων επαναλήψεων.
- Αντίθετα οι επαναλήψεις του εσωτερικου for-loop είναι εύκολο να εκτελεστούν παράλληλα. Αν *p* είναι το πλήθος των διαθέσιμων διεργασιών και n το πλήθος των κορυφών του γραφήματος, οι κορυφές του γραφήματος χωρίζονται σε p υποσύνολα και κάθε διεργασία $P_i$ αναλαμβάνει τις κορυφές i*n/p, i*n/p+1 ...(i+1)*n/p-1 όπου i=0,...,p-1. Έστω $V_i$ το σύνολο αυτών των κορυφών.
- Ο διαμοιρασμός αυτός των κορυφών στις διαθέσιμες διεργασίες συνεπάγεται και το χωρισμό του πίνακα γειτνίασης σε p κατακόρυφες λωρίδες πάχους n/p στηλών. Κάθε επεξεργαστής αναλαμβάνει την λωρίδα που αντιστοιχεί στις δικές του κορυφές.
- Με ανάλογο τρόπο χωρίζεται ο πίνακας d που αποθηκεύει τη τρέχουσα ελάχιστη απόσταση μίας κορυφής από το υπό κατασκευή ΕΓΔ. Ο χωρισμός του πίνακα γειτνίασης και του πίνακα d φαίνεται στο σχήμα που ακολουθεί.



- Σε κάθε επανάληψη του while loop, κάθε διεργασία $P_i$ υπολογίζει τοπικά την ελάχιστη απόσταση που χωρίζει τις κορυφές της από το υπο-κατασκευή ΕΓΔ δηλ.

$$d_i[u] = \min\{d_i[v], v \in (V - V_T) \cap V_i\}$$

Η πολυπλοκότητα αυτού του βήματος είναι O(n/p).

- Το συνολικό ελάχιστο λαμβάνεται από όλα τα $d_i[u]$ χρησιμοποιώντας μία λειτουργία μείωσης (reduction) και αποθηκεύεται στη διεργασία $P_0$ *(κόστος O(log p))*
- Έτσι, η διεργασία $P_0$ έχει τη νέα κορυφή u που θα εισέλθει στο ΕΓΔ (σύνολο $V_T$).
- Στη συνέχεια η διεργασία $P_0$ στέλνει τη u σε όλες τις διεργασίες χρησιμοποιώντας τη λειτουργία της εκπομπής (κόστος O(log p))
- Η διεργασία $P_i$ υπεύθυνη για την κορυφή u σημειώνει τη u ότι ανήκει στο σύνολο $V_T$ (κόστος O(1)).
- Στο τέλος, κάθε διεργασία ενημερώνει τις τιμές των d[v] των τοπικών της κορυφών (κόστος O(n/p)).
- Η παραπάνω διαδικασία επαναλαμβάνεται μέχρι όλες οι κορυφές να εισέλθουν στο σύνολο $V_T$
- Η συνολική πολυπλοκότητα του αλγορίθμου θα είναι $O(n^2/p + n \log p)$.

# Εύρεση συντομότερων μονοπατιών με κοινή αφετηρία

- Για ένα βεβαρυμμένο γράφημα *G = (V,E,w)*, το πρόβλημα των συντομότερων μονοπατιών με κοινή αφετηρία είναι η εύρεση των συντομότερων μονοπατιών από μία κορυφή $s \in V$ στις άλλες κορυφές του γραφήματος.
- Το πρόβλημα αυτό επιλύεται με τον αλγόριθμο του Dijkstra ο οποίος παρουσιάζει πολλές ομοιότητες με τον αλγόριθμο του Prim. Συγκεκριμένα σε κάθε βήμα του, ο αλγόριθμος συντηρεί το σύνολο $V_T$ των κορυφών για τις οποίες είναι ήδη γνωστά τα συντομότερα μονοπάτια.
- Σε κάθε βήμα, το σύνολο $V_T$ αυξάνεται κατά ένα κόμβο. Συγκεκριμένα, ο κόμβος αυτός είναι ο πιο κοντινός στη αφετηρία από όλους τους υπόλοιπους κόμβους που δεν ανήκουν στο σύνολο $V_T$. Σημαντική λεπτομέρεια του αλγορίθμου είναι ότι το μονοπάτι του νεοεισερχόμενου κόμβου στο $V_T$ θα πρέπει απαρτίζεται μόνο από κόμβους που είναι ήδη στο $V_T$.
- Ο ακολουθιακός κώδικας για τον αλγόριθμο του Dijkstra έχει ως εξής:

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

# Παράλληλη υλοποίηση του αλγόριθμου του Dijkstra

- Η παράλληλη υλοποίηση του αλγόριθμου αυτού είναι παρόμοια με αυτή του αλγόριθμου του Prim.
- Τόσο ο πίνακας γειτνίασης όσο και ο πίνακας των αποστάσεων l μοιράζονται με τον ίδιο τρόπο στις διεργασίες.
- Κάθε διεργασία επιλέγει τοπικά την κορυφή που είναι κοντινότερη στην αφετηρία και στη συνέχεια, με μία λειτουργία μείωσης, βρίσκεται η συνολικά κοντινότερη κορυφή ως προς την αφετηρία.
- Στη συνέχεια, η κορυφή αυτή στέλνεται σε όλες τις διεργασίες με μία λειτουργία εκπομπής και οι διεργασίες ενημερώνουν το τμήμα του πίνακα l που κατέχουν.
- Είναι εύκολο να δει κανείς ότι η πολυπλοκότητα του παράλληλου αλγόριθμου Dijkstra είναι ίδια με αυτή του αλγορίθμου Prim.

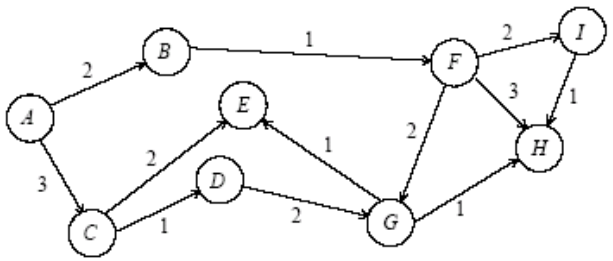
# Εύρεση συντομότερων μονοπατιών μεταξύ όλων των ζευγών κορυφών

- Δοθέντος ενός βεβαρυμμένου γραφήματος G(V,E,w), το πρόβλημα των συντομότερων μονοπατιών μεταξύ όλων των ζευγών κορυφών είναι η εύρεση των συντομότερων μονοπατιών μεταξύ όλων των ζευγών κορυφών $v_i$, $v_j \in V$.
- Ο πρώτος αλγόριθμος που επιλύει το πρόβλημα βασίζεται στον πολλαπλασιασμό πινάκων.
- Συγκεκριμένα αν υπολογίσουμε το τετράγωνο του πίνακα γειτνίασης A δηλ. A*A, όπου στη θέση του πολλαπλασιασμού εκτελείται πρόσθεση και στη θέση της πρόσθεσης τώρα εκτελείται η λειτουργία του ελάχιστου, τότε το αποτέλεσμα $A^2$ θα περιέχει όλα τα συντομότερα μονοπάτια που έχουν μήκος το πολύ 2. Συγκεκριμένα, εκτελούμε τον εξής μετασχηματισμό:

$$a_{ij} = \sum_{k=0}^{|V|-1} a_{ik} a_{kj} \quad \rightarrow \quad a_{ij} = \min(a_{ij}, a_{i0} + a_{0j}, a_{i1} + a_{1j}, \ldots, a_{i,|V|-1} + a_{|V|-1,j})$$

- Γενικεύοντας αυτή την παρατήρηση, μπορούμε να δούμε ότι ο πίνακας $A^n$ περιέχει τα κόστη των συντομότερων μονοπατιών μεταξύ όλων των ζευγών κορυφών.
- Ο πίνακας $A^n$ υπολογίζεται με διπλασιασμό δυνάμεων – δηλ., $A$, $A^2$, $A^4$, $A^8$, κοκ.
- Χρειαζόμαστε *log n* πολλαπλασιασμούς πινάκων. Ο ακολουθιακός υπολογισμός για τον πολλαπλασιασμό πινάκων απαιτεί χρόνο $O(n^3)$.
- Άρα η συνολική ακολουθιακή πολυπλοκότητα θα είναι $O(n^3 \log n)$.
- Αυτός ο αλγόριθμος δεν είναι βέλτιστος αφού ο καλύτερος ακολουθιακός αλγόριθμος έχει πολυπλοκότητα $O(n^3)$.
- Η τεχνική αυτή μπορεί να υλοποιηθεί παράλληλα, με την παράλληλη εκτέλεση καθενός από του O(log n) πολλαπλασιασμών που απαιτούνται συνολικά.

# Παράδειγμα εκτέλεσης της τεχνικής του πολλαπλασιασμού πινάκων




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

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

# Αλγόριθμος του Floyd

- Ο αλγόριθμος αυτός επιλύει το πρόβλημα των συντομότερων μονοπατιών μεταξύ όλων των ζευγών κορυφών και βασίζεται στην εξής παρατήρηση:
  - Για οποιοδήποτε ζεύγος κορυφών $v_i$, $v_j \in V$, έστω $p_{i,j}^{(k)}$ (βάρους $d_{i,j}^{(k)}$) το μονοπάτι με το ελάχιστο βάρος μεταξύ όλων των μονοπατιών από τη κορυφή $v_i$ στη κορυφή $v_j$ των οποίων οι ενδιάμεσες κορυφές ανήκουν στο σύνολο $\{v_1, v_2, \ldots, v_k\}$.
  - Αν η κορυφή$v_k$ δεν είναι στο συντομότερο μονοπάτι από τη κορυφή $v_i$ στη κορυφή $v_j$, τότε το μονοπάτι $p_{i,j}^{(k)}$ είναι το ίδιο με $p_{i,j}^{(k-1)}$.
  - Αν $v_k$ είναι στο $p_{i,j}^{(k)}$, τότε μπορούμε να σπάσουμε το μονοπάτι $p_{i,j}^{(k)}$ σε δύο μονοπάτια – ένα από τη κορυφή $v_i$ στη κορυφή $v_k$ και ένα από τη $v_k$ στη $v_j$ . Κάθε ένα από αυτά τα μονοπάτια χρησιμοποιεί κορυφές από το σύνολο $\{v_1, v_2, \ldots, v_{k-1}\}$.
- Από τις παραπάνω παρατηρήσεις, η ακόλουθη αναδρομική εξίσωση προκύπτει:

$$d_{i,j}^{(k)} = \begin{cases} w(v_i, v_j) & if \;\; k = 0 \\ \min\left\{ d_{i,j}^{(k-1)}, d_{i,k}^{(k-1)} + d_{k,j}^{(k-1)} \right\} & if \;\; k \geq 1 \end{cases}$$

- Επομένως η τιμή $d_{i,j}^{(n)}$ θα δίνει το κόστος του συντομότερου μονοπατιού μεταξύ των κορυφών $v_i$ και $v_j$.

Με βάση την προηγούμενη αναδρομική σχέση, ο ακολουθιακός αλγόριθμος FLOYD θα έχει ως εξής:

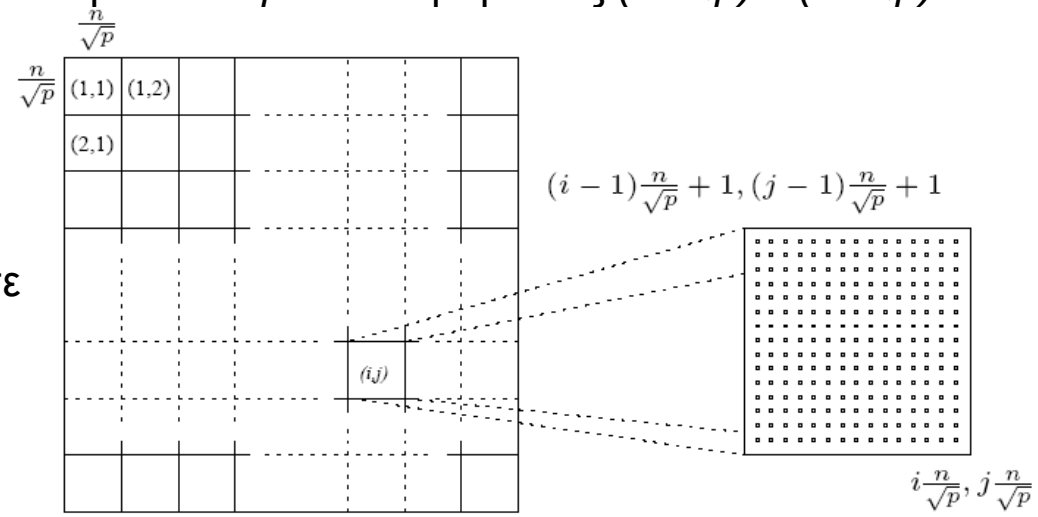
```
1.  procedure FLOYD_ALL_PAIRS_SP(A)
2.  begin
3.      D^(0) = A;
4.      for k := 1 to n do
5.          for i := 1 to n do
6.              for j := 1 to n do
7.                  d_{i,j}^(k) := min( d_{i,j}^(k-1), d_{i,k}^(k-1) + d_{k,j}^(k-1) );
8.  end FLOYD_ALL_PAIRS_SP
```

Η πολυπλοκότητα του αλγορίθμου είναι $O(n^3)$

# Παράλληλη υλοποίηση του αλγόριθμου του **Floyd**

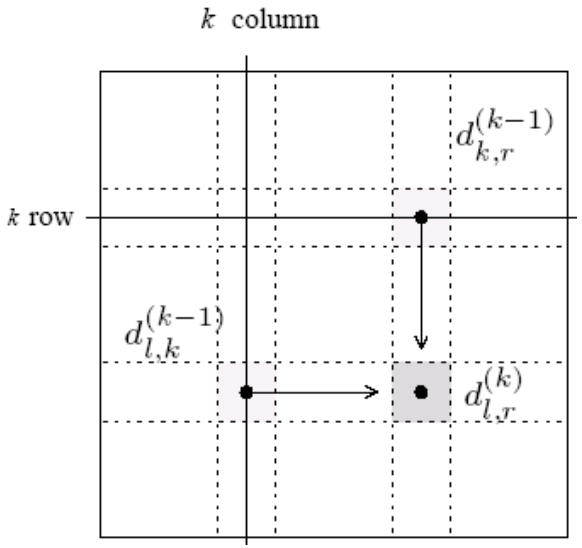
- Ο πίνακας $D^{(k)}$ διαιρείται σε $p$ blocks μεγέθους $(n / \sqrt{p}) \times (n / \sqrt{p})$.

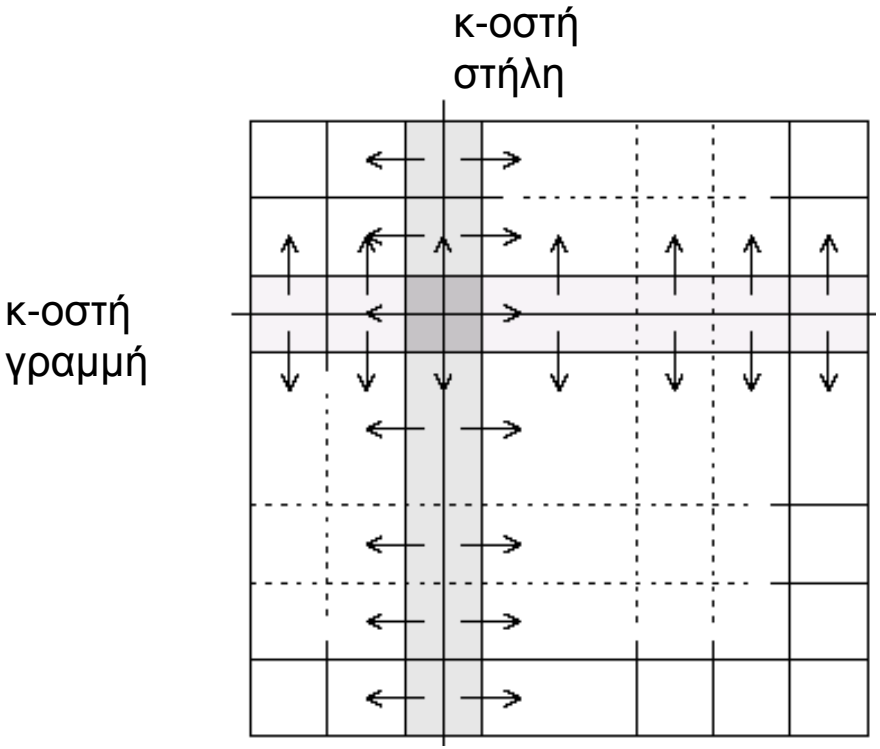
Matrix $D^{(k)}$ διαιρείται σε $\sqrt{p} \times \sqrt{p}$ blocks




Το block αυτό δίνεται στην διεργασία $P_{i,j}$

- Κάθε διεργασία αναλαμβάνει ένα από αυτά τα blocks
- Σε κάθε επανάληψη, κάθε διεργασία ενημερώνει το τμήμα του πίνακα που έχει αναλάβει. Για να υπολογίζει το $d_{l,r}^{(k-1)}$ η διεργασία $P_{i,j}$ πρέπει να λάβει τα στοιχεία $d_{l,k}^{(k-1)}$ και $d_{k,r}^{(k-1)}$.






- Γενικά, κατά την κ-οστή επανάληψη, κάθε μία από τις $\sqrt{p}$ διεργασίες οι οποίες περιέχουν τμήμα της κ-οστής γραμμής στέλνουν το τμήμα αυτό στις $\sqrt{p}$ - 1 διεργασίες της ίδιας στήλης.
- Όμοια, κάθε μία από τις $\sqrt{p}$ διεργασίες οι οποίες περιέχουν τμήμα της κ-οστής στήλης στέλνουν το τμήμα τους στις $\sqrt{p} - 1$ διεργασίες στην ίδια γραμμή.

Έτσι, ο παράλληλος αλγόριθμος για το αλγόριθμο του FLOYD έχει ως εξής:

```
1.    procedure FLOYD_2DBLOCK(D^(0))
2.    begin
3.        for k := 1 to n do
4.        begin
5.            each process P_{i,j} that has a segment of the k^th row of D^(k-1);
                  broadcasts it to the P_{*,j} processes;
6.            each process P_{i,j} that has a segment of the k^th column of D^(k-1);
                  broadcasts it to the P_{i,*} processes;
7.            each process waits to receive the needed segments;
8.            each process P_{i,j} computes its part of the D^(k) matrix;
9.        end
10.   end FLOYD_2DBLOCK
```

Στο ψευδοκώδικα, $P_{*,j}$ είναι όλες τις διεργασίες της j-οστής στήλης και $P_{i,*}$ είναι όλες οι διεργασίες στην i-οστη γραμμή. Ο πίνακας $D^{(0)}$ είναι ο αρχικός πίνακας γειτνίασης.

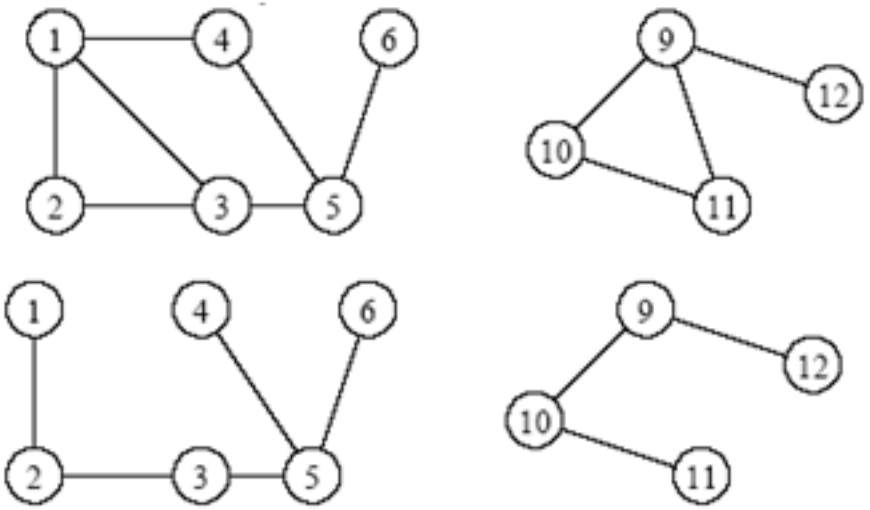
Ανάλυση πολυπλοκότητας αλγορίθμου:
Σε κάθε επανάληψη του αλγορίθμου, οι διεργασίες της κ-οστής γραμμής και της κ-οστής στήλης εκτελούν λειτουργία εκπομπής κατά μήκος των στηλών και των γραμμών τους αντίστοιχα.
Το πλήθος των δεδομένων που στέλνεται σε κάθε λειτουργία εκπομπής είναι $n/\sqrt{p}$. Σε ένα σύστημα κατανεμημένης μνήμης, η εκπομπή ενός στοιχείου μπορεί να ολοκληρωθεί σε χρόνο $\Theta(\log p)$. Άρα, συνολικά ο χρόνος για αυτό το βήμα θα είναι $\Theta((n \log p)/\sqrt{p})$.
Το βήμα συγχρονισμού μετά από κάθε επανάληψη απαιτεί χρόνο $\Theta(\log p)$.
Σε κάθε επανάληψη, ο χρόνος που απαιτείται για τους υπολογισμούς του αλγορίθμου είναι $\Theta(n^2/p)$.
Συνολικά, ο χρόνος εκτέλεσης του αλγορίθμου θα είναι: $T_P = \Theta\left(\frac{n^3}{p}\right) + \Theta\left(\frac{n^2}{\sqrt{p}} \log p\right)$

# Μεταβατική κλειστότητα

- Αν $G = (V,E)$ είναι ένα γράφημα, τότε η μεταβατική κλειστότητα του $G$ ορίζεται ως το γράφημα $G^* = (V,E^*)$, όπου $E^* = \{(v_i,v_j) \mid$ *υπάρχει μονοπάτι από τη κορυφή* $v_i$ *στη κορυφή* $v_j$ *στο* $G\}$
- Ο πίνακας συνεκτικότητας του $G$ είναι ένας πίνακας $A^* = (a^*_{i,j})$ όπου το στοιχείο $a^*_{i,j} = 1$ αν και μόνο αν υπάρχει μονοπάτι από την κορυφή $v_i$ στη κορυφή $v_j$ ή όταν $i = j$. Σε διαφορετική περίπτωση, $a^*_{i,j} = \infty$.
- Για να υπολογίσουμε τον $A^*$, αρχικά θεωρούμε το βάρος κάθε ακμής ίσο με 1 και στη συνέχεια μπορούμε να εφαρμόσουμε σε αυτό το βεβαρυμμένο γράφημα οποιοδήποτε αλγόριθμο που επιλύει το πρόβλημα συντομότερων διαδρομών μεταξύ όλων των ζευγών κορυφών.
- Οι συνεκτικές συνιστώσες ενός μη κατευθυνόμενου γραφήματος είναι κλάσεις ισοδυναμίας της σχέσης «έχει πρόσβαση στη» στο σύνολο των κορυφών του γραφήματος.

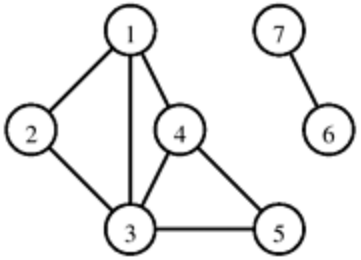
# Εύρεση συνεκτικών συνιστωσών : Αναζήτηση πρώτα κατά βάθος

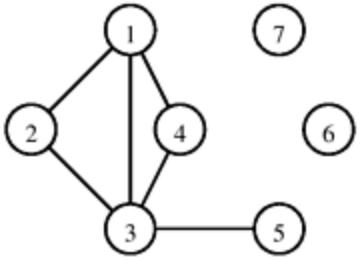
- Η εκτέλεση της αναζήτησης πρώτα κατά βάθος σε ένα γράφημα δημιουργεί ένα δάσος – κάθε δέντρο στο δάσος αντιστοιχεί σε μία διαφορετική συνεκτική συνιστώσα.
- Π.χ. με εφαρμογή της αναζήτησης πρώτα κατά βάθος στο γράφημα που ακολουθεί, προκύπτουν δύο διαφορετικά δέντρα τα οποία αντιστοιχούν στις δύο συνεκτικές συνιστώσες του γραφήματος



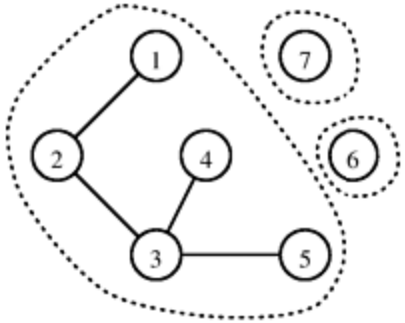

Για τη παράλληλη υλοποίηση, το αρχικό γράφημα μοιράζεται στις διαθέσιμες διεργασίες και κάθε διεργασία υπολογίζει τις συνεκτικές συνιστώσες αποκλειστικά στο τμήμα του γραφήματος που του αντιστοιχεί. Σε αυτό το σημείο, έχουν δημιουργηθεί *p* γεννητικά δάση.
Στο δεύτερο βήμα, τα γεννητικά δάση συγχωνεύονται σε ζευγάρια μέχρι ένα γεννητικό δάσος να απομείνει.

Παράδειγμα:



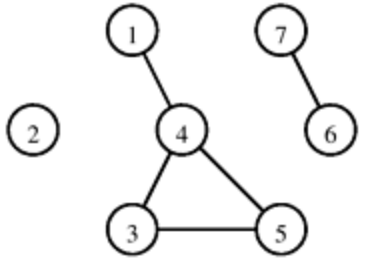

Η διεργασία 1 βρίσκει τις συνεκτικές συνιστώσες του παραπάνω υπογραφήματος.



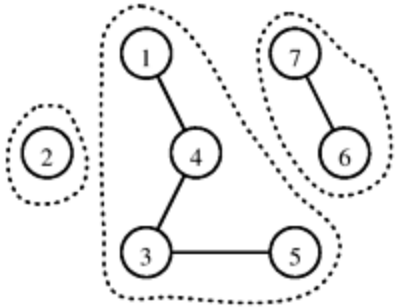

Το αποτέλεσμα της επεξεργασίας της διεργασίας 1.

| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | |
|---|---|---|---|---|---|---|---|---|
| 1 | 0 | 1 | 1 | 1 | 0 | 0 | 0 | |
| 2 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | Processor 1 |
| 3 | 1 | 1 | 0 | 1 | 1 | 0 | 0 | |
| 4 | 1 | 0 | 1 | 0 | 1 | 0 | 0 | |
| 5 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | |
| 6 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | Processor 2 |
| 7 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | |

Ο πίνακας γειτνίασης του αριστερού γραφήματος διαιρείται στις δύο διαθέσιμες διεργασίες




Η διεργασία 2 βρίσκει τις συνεκτικές συνιστώσες του παραπάνω υπογραφήματος.




Το αποτέλεσμα της επεξεργασίας της διεργασίας 2.

Στο τέλος, τα γεννητικά δέντρα των δύο διεργασιών συνενώνονται σε ένα δάσος.

- Για τη συγχώνευση των γεννητικών δασών, ο αλγόριθμος χρησιμοποιεί τα σύνολα ακμών των δασών Α και Β τα οποία είναι ξένα μεταξύ τους.

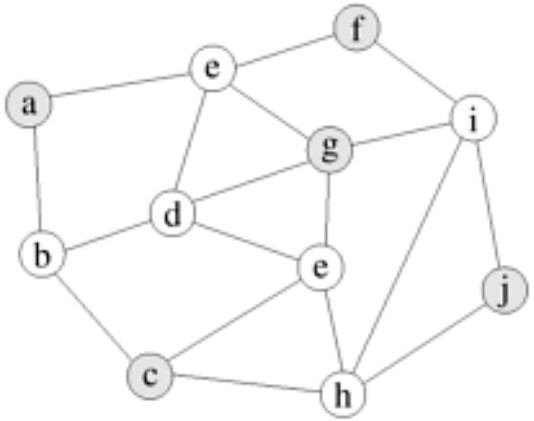
- Ορίζουμε τις ακόλουθες λειτουργίες στα ξένα σύνολα:
- ***find*(*x*)**
  - επιστρέφει ένα δείκτη στο στοιχείο αντιπρόσωπο του συνόλου που περιέχει το *x*. Κάθε σύνολο έχει το δικό του μοναδικό αντιπρόσωπο.
- ***union*(*x, y*)**
  - ενώνει δύο σύνολα τα οποία περιέχουν τα στοιχεία *x* και *y*. Γίνεται η υπόθεση ότι τα δύο σύνολα είναι ξένα πριν την εφαρμογή της λειτουργίας.
- Για την συγχώνευση του δάσους Α στο δάσος *B*, για κάθε ακμή *(u,v)* του *A*, εκτελείται μία *find* λειτουργία για να προσδιορισθεί αν οι κορυφές της ακμής είναι και οι δύο στο ίδιο δέντρο του *B*.
- Αν αυτό ισχύει, δεν απαιτείται καμία λειτουργία union
- Στην αντίθετη περίπτωση, τα δύο δέντρα του *B* που περιέχουν τις κορυφές *u* και *v* ενώνονται με μία λειτουργία *union*.
- Συνολικά, η συγχώνευση των δασών *A* και *B* απαιτεί το πολύ *2(n-1)* λειτουργίες *find* και *(n-1)* λειτουργίες *union*.

- Στη γενική περίπτωση όταν υπάρχουν p διαθέσιμες διεργασίες, ο πίνακας γειτνίασης nxn διαιρείται σε p blocks.
- Κάθε επεξεργαστής μπορεί να υπολογίσει το τοπικό του γεννητικό δάσος σε χρόνο $\Theta(n^2/p)$.
- Η συγχώνευση των δασών μπορεί να γίνει οργανώνοντας τις διαθέσιμες διεργασίες σε ένα λογικό δέντρο. Υπάρχουν log p στάδια συγχώνευσης καθένα από τα οποία απαιτεί χρόνο Θ(n). Έτσι, το συνολικό κόστος λόγω συγχώνευσης είναι Θ(n log p).
- Κατά τη διάρκεια κάθε βήματος συγχώνευσης, τα γεννητικά δάση στέλνονται στους πιο κοντινούς γείτονες. Συνολικά Θ(n) ακμές γεννητικών ακμών μεταδίδονται. Έτσι, το συνολικό κόστος επικοινωνίας θα είναι Θ(n log p).
- Έτσι το συνολικό κόστος του αλγορίθμου θα είναι:

$$T_P = \Theta\left(\frac{n^2}{p}\right) + \Theta\left(n \log p\right)$$

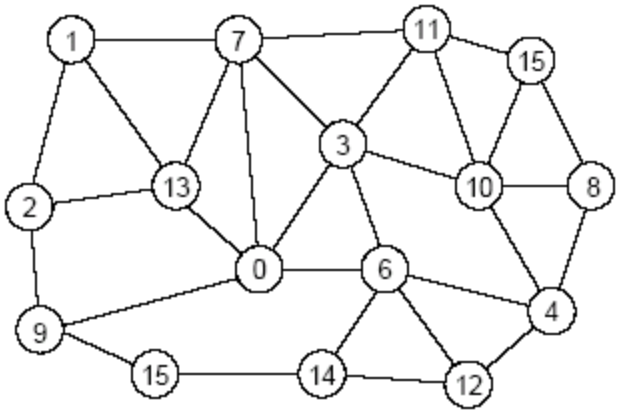
# Εύρεση ενός Μέγιστου (Maximal) Ανεξάρτητου Συνόλου

- Ένα σύνολο κορυφών $I \subset V$ ονομάζεται ανεξάρτητο αν κανένα ζεύγος κορυφών στο *I* δεν συνδέεται με μία ακμή στο *G*. Ένα ανεξάρτητο σύνολο λέγεται μέγιστο (*maximal)* αν με τη συμπερίληψη οποιασδήποτε άλλης κορυφής που δεν ανήκει στο σύνολο I, η ιδιότητα της ανεξαρτησίας παραβιάζεται.

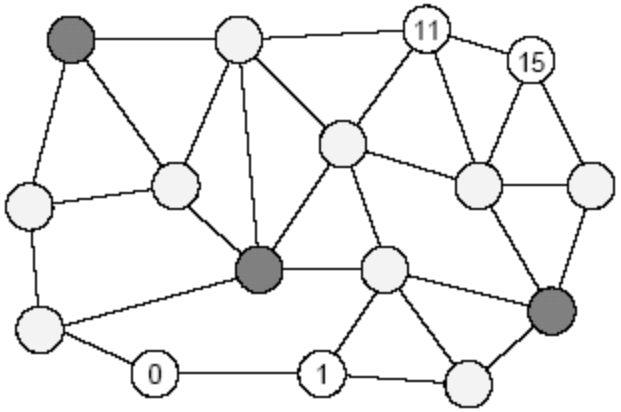



  - Στο συγκεκριμένο παράδειγμα, το σύνολο των κορυφών {a, d, i, h} είναι ανεξάρτητο. Τα σύνολα {a, c, j, f, g} και {a, d, h, f} είναι μέγιστα ανεξάρτητα σύνολα.
  - Ένας απλός αλγόριθμος για την εύρεση του μέγιστου ανεξάρτητου συνόλου αρχίζει με το ανεξάρτητο σύνολο I αρχικά κενό. Επίσης, υπάρχει ένα σύνολο C που περιέχει τις υποψήφιες κορυφές προς εισαγωγή στο ανεξάρτητο σύνολο I και αρχικά περιλαμβάνει όλες τις κορυφές του γραφήματος.
  - Σε κάθε βήμα, μία κορυφή *v* επιλέγεται από το *C* η οποία στη συνέχεια μετακινείται στο σύνολο I ενώ όλες οι κορυφές που είναι γειτονικές στην v αφαιρούνται από το σύνολο C.
  - Αυτή η διαδικασία επαναλαμβάνεται μέχρι το σύνολο C να αδειάσει.
  - Το πρόβλημα με την παραπάνω διαδικασία είναι ότι είναι εγγενώς ακολουθιακή.

- Ο παράλληλος αλγόριθμος για την εύρεση του μέγιστου ανεξάρτητου συνόλου εισάγει τυχαιότητα στην επιλογή της επόμενης κορυφής που θα συμπεριληφθεί στο ανεξάρτητο σύνολο προκειμένου να είναι δυνατή η παράλληλη εκτέλεση κάποιων ενεργειών
- Ο αλγόριθμος βασίζεται στον αλγόριθμο του Luby για το χρωματισμό γραφήματος.
- Αρχικά, κάθε κόμβος είναι στο σύνολο C των υποψήφιων προς εισαγωγή κόμβων. Κάθε κόμβος παράγει ένα μοναδικό τυχαίο αριθμό και το στέλνει στους γείτονες του.
- Αν ο αριθμός ενός κόμβου είναι μεγαλύτερος από αυτούς των γειτόνων του, τότε ο κόμβος αυτός συμπεριλαμβάνεται στο σύνολο I και αφαιρείται από το σύνολο *C*.
- Η διαδικασία συνεχίζεται μέχρι το σύνολο *C* να γίνει κενό.
- Κατά μέσο όρο, αυτός ο αλγόριθμος συγκλίνει μετά απόr $O(\log|V|)$ τέτοια βήματα.
- Ακολουθεί ένα παράδειγμα εφαρμογής του αλγορίθμου. Με γκρι χρωματίζονται οι κόμβοι που ανήκουν στο ανεξάρτητο σύνολο, ενώ οι κενοί κόμβοι είναι γείτονες αυτών των κόμβων

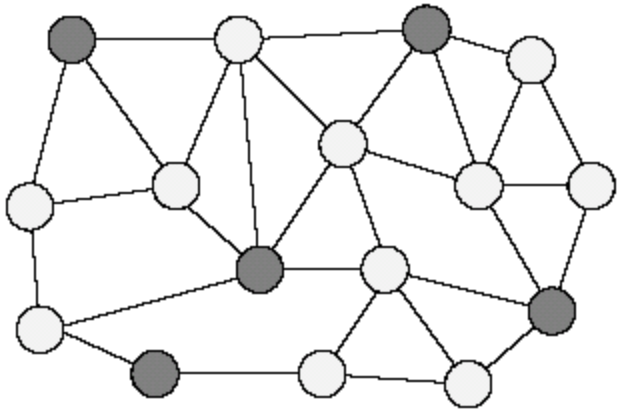



Ανάθεση αρχικών τυχαίων αριθμών στους κόμβους του γραφήματος




Δεύτερη ανάθεση τυχαίων αριθμών στους κόμβους του γραφήματος

Τελικό μέγιστο ανεξάρτητο σύνολο



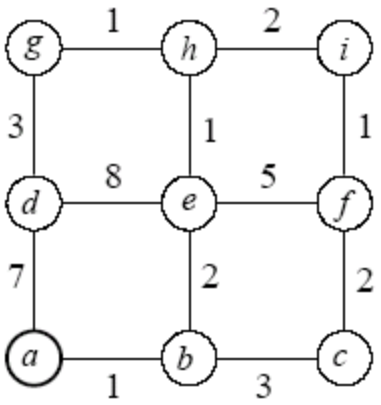
- Για την παράλληλη εκτέλεση του αλγορίθμου, απαιτούνται τρεις πίνακες n στοιχείων. Συγκεκριμένα, ο πίνακας I που αποθηκεύει κόμβους του μέγιστου ανεξάρτητου συνόλου, ο C που αποθηκεύει το σύνολο των υποψήφιων κορυφών και το σύνολο R που αποθηκεύει τους τυχαίους αριθμούς. Τα στοιχεία των πινάκων I και C παίρνουν δυαδικές τιμές, με το 1 να δείχνει ότι ο αντίστοιχος κόμβος ανήκει στο σύνολο και το 0 να δείχνει το αντίθετο.
- Στη συνέχεια, μοιράζουμε τον πίνακα *C* στις *p* διεργασίες. Κάθε διεργασία παράγει τις τυχαίες τιμές των κόμβων που έχει αναλάβει και τις αποθηκεύει στις αντίστοιχες θέσεις του πίνακα *R.*
- Με βάση τις τιμές του πίνακα R κάθε διεργασία υπολογίζει ποιες είναι οι επόμενες κορυφές μεταξύ των κορυφών που έχει αναλάβει που θα μπουν στο μέγιστο ανεξάρτητο σύνολο.
- Ο πίνακας C ενημερώνεται θέτοντας 0 στις θέσεις των κόμβων που εισήλθαν στο μέγιστο ανεξάρτητο σύνολο ή είναι γείτονες των κόμβων που μόλις εισήλθαν στο ανεξάρτητο σύνολο. Επίσης θέτουμε 1 στις θέσεις του πίνακα I που αντιστοιχούν στους νεοεισερχόμενους κόμβους.
- Η συνολική πολυπλοκότητα του αλγορίθμου εξαρτάται από τη συγκεκριμένη δομή του γραφήματος.

# Συντομότερα μονοπάτια με κοινή αφετηρία

- Ο αλγόριθμος του Johnson αποτελεί τροποποίηση του αλγόριθμου του Dijkstra, ώστε να εκτελείται αποδοτικά στη περίπτωση των αραιών γραφημάτων.
- Η τροποποίηση αυτή λαμβάνει υπόψη το γεγονός ότι το βήμα ελαχιστοποίησης του αλγόριθμου Dijkstra χρειάζεται, στη πράξη, να εκτελείται μόνο στους κόμβους οι οποίοι είναι γειτονικοί ως προς κόμβο του οποίου μόλις προσδιορίσθηκε η συντομότερη διαδρομή από το αλγόριθμο Dijkstra.
- Ο αλγόριθμος του Johnson χρησιμοποιεί μία ουρά προτεραιότητας *Q* για να αποθηκεύσει την τιμή *l[v]* για κάθε κορυφή $v \in (V - V_T)$.
- Ο ακολουθιακός αλγόριθμος έχει ως εξής:

```
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
```

- Πιστή εκτέλεση του αλγόριθμου του Johnson δεν επιτρέπει την εκτέλεση πολλών λειτουργιών παράλληλα. Αυτό οφείλεται στο γεγονός ότι ο ακολουθιακός αλγόριθμος επιβάλλει μία συγκεκριμένη σειρά επίσκεψης των κορυφών του γραφήματος.
- Για να αυξηθεί ο παραλληλισμός στο συγκεκριμένο υπολογισμό, χρειάζεται να εξεταστούν πολλοί κόμβοι ταυτόχρονα. Για το σκοπό αυτό, εξάγονται p κορυφές (όσες και οι διαθέσιμες διεργασίες) από την ουρά προτεραιότητας και επεκτείνονται ανάλογα τα συντομότερα μονοπάτια. Στη συνέχεια, ενημερώνονται τα κόστη όλων των γειτόνων αυτών των κορυφών.
- Αν στη συνέχεια διαπιστωθεί ότι υπάρχει συντομότερο μονοπάτι για ένα κόμβο που βγήκε από την ουρά προτεραιότητας, ο κόμβος αυτός εισάγεται ξανά στην ουρά προτεραιότητας.
- Στο παράδειγμα που ακολουθεί, υπάρχουν τρεις διεργασίες και το ζητούμενο είναι η εύρεση των συντομότερων διαδρομών με αρχή την κορυφή a.
- Αρχικά, οι διεργασίες $P_0$ και $P_1$ εξάγουν τις κορυφές b και d και στη συνέχεια ενημερώνουν τις τιμές l των κορυφών που είναι γειτονικές στη b και d.
- Στο δεύτερο βήμα, οι διεργασίες $P_0$,$P_1$ και $P_2$ εξάγουν τις κορυφές e, c και g και στη συνέχεια ενημερώνουν τις τιμές l των κορυφών που είναι γειτονικές σε αυτές.
- Παρατηρείστε ότι όταν η διεργασία $P_0$ εξάγει την κορυφή h, διαπιστώνει ότι l[h]+w(h,g)=5 σε σύγκριση με την τιμή l[g]=10 η οποία εξήχθη από την ουρά προτεραιότητας στη προηγούμενη επανάληψη.
- Αυτό έχει ως αποτέλεσμα, η κορυφή g να επιστρέψει στην ουρά προτεραιότητας με ενημερωμένη την τιμή l[g].




(1) *b:1, d:7, c:inf, e:inf, f:inf, g:inf, h:inf, i:inf*

(2) *e:3, c:4, g:10, f:inf, h:inf, i:inf*

(3) *h:4, f:6, i:inf*

(4) *g:5, i:6*

Array l[]

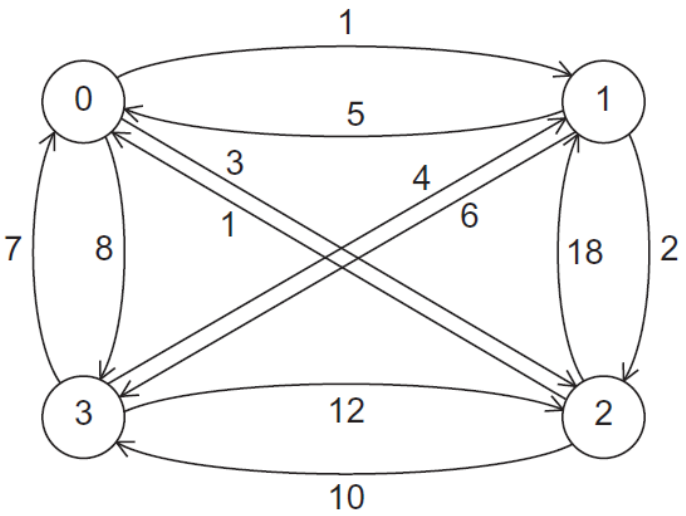
| *a* | *b* | *c* | *d* | *e* | *f* | *g* | *h* | *i* |
|---|---|---|---|---|---|---|---|---|
| 0 | 1 | ∞ | 7 | ∞ | ∞ | ∞ | ∞ | ∞ |
| 0 | 1 | 4 | 7 | 3 | ∞ | 10 | ∞ | ∞ |
| 0 | 1 | 4 | 7 | 3 | 6 | 10 | 4 | ∞ |
| 0 | 1 | 4 | 7 | 3 | 6 | 5 | 4 | 6 |

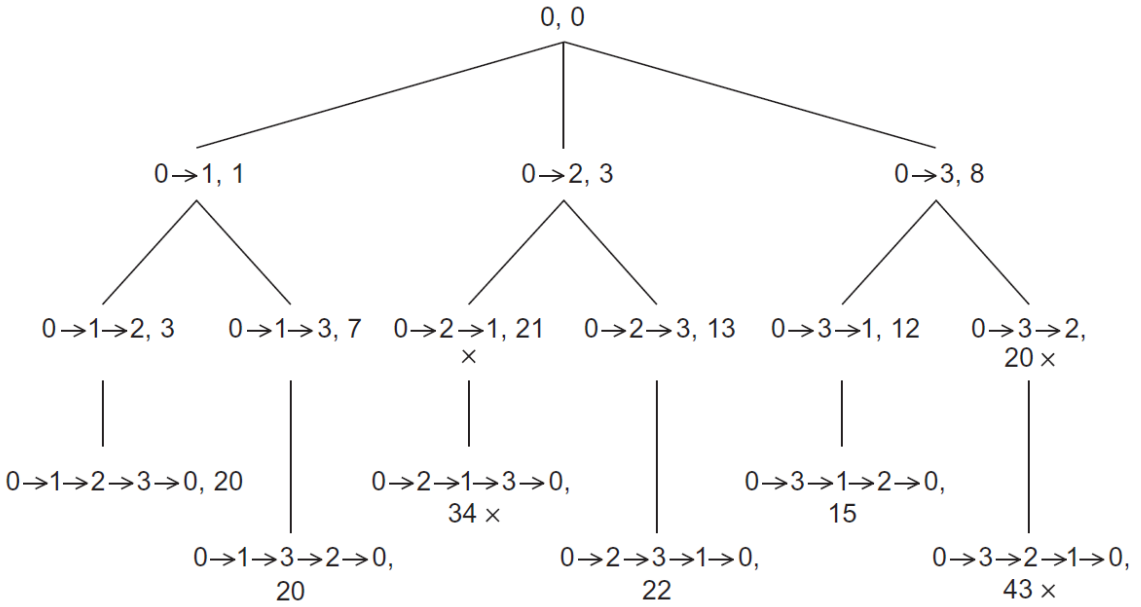
- Αν υλοποιήσουμε την παραπάνω τεχνική σε σύστημα κατανεμημένης μνήμης, η διαχείριση της κοινής ουράς αποτελεί αιτία καθυστέρησης του συνολικού υπολογισμού.
- Για αυτό το λόγο, πολλαπλές ανεξάρτητες ουρές χρησιμοποιούνται, μία για κάθε διεργασία.
- Συγκεκριμένα, κάθε διεργασία χτίζει τη δική της ουρά προτεραιότητας η οποία περιέχει τις κορυφές που χειρίζεται η διεργασία.
- Όταν μία διεργασία $P_i$ εξάγει την κορυφή $u \in V_i$, στέλνει μήνυμα με την τιμή l[u] σε όλες τις διεργασίες που χειρίζονται κορυφές γειτονικές στην κορυφή $u$.
- Κάθε διεργασία $P_j$, αφού λάβει το μήνυμα από την $P_i$, θέτει την τιμή $l[v]$ στην τιμή $min\{l[v],l[u] + w(u,v)\}$
- Αν συντομότερο μονοπάτι ανακαλυφθεί για τον κόμβο $v$, η τιμή l[v] επανεισάγεται στην τοπική ουρά προτεραιότητας.
- Ο αλγόριθμος τερματίζει μόνο όταν όλες οι ουρές αδειάσουν.
- Μπορούμε να χρησιμοποιήσουμε διαφορετικούς τρόπους διαμοίρασης των κορυφών του γραφήματος στις διάφορες διεργασίες ανάλογα με τη συγκεκριμένη δομή του γραφήματος που επεξεργαζόμαστε.

# Το Πρόβλημα του Περιοδεύοντος Πωλητή (ΠΠΠ)

- Το πρόβλημα αυτό είναι ένα NP-complete πρόβλημα.
- Δεν υπάρχει γνωστή λύση για το συγκεκριμένο πρόβλημα η οποία να είναι καλύτερη σε όλες τις περιπτώσεις σε σχέση με την εξαντλητική αναζήτηση.

Ένα παράδειγμα
με 4 πόλεις



# Ψεδοκώδικας για την αναδρομική λύση του ΠΠΠ χρησιμοποιώντας την αναζήτηση πρώτα κατά βάθος

```
void Depth_first_search(tour_t tour) {
   city_t city;

   if (City_count(tour) == n) {
      if (Best_tour(tour))
         Update_best_tour(tour);
   } else {
      for each neighboring city
         if (Feasible(tour, city)) {
            Add_city(tour, city);
            Depth_first_search(tour);
            Remove_last_city(tour);
         }
   }
}  /* Depth_first_search */
```

# Ψευδοκώδικας για την υλοποίηση της αναζήτησης πρώτα κατά βάθος για το ΠΠΠ χωρίς αναδρομή

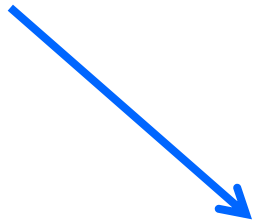
```
for (city = n-1; city >= 1; city--)
   Push(stack, city);
while (!Empty(stack)) {
   city = Pop(stack);
   if (city == NO_CITY) // End of child list, back up
      Remove_last_city(curr_tour);
   else {
      Add_city(curr_tour, city);
      if (City_count(curr_tour) == n) {
         if (Best_tour(curr_tour))
            Update_best_tour(curr_tour);
         Remove_last_city(curr_tour);
      } else {
         Push(stack, NO_CITY);
         for (nbr = n-1; nbr >= 1; nbr--)
            if (Feasible(curr_tour, nbr))
               Push(stack, nbr);
      }
   } /* if Feasible */
} /* while !Empty */
```

# Ψευδοκώδικας για τη δεύτερη λύση στο ΠΠΠ η οποία δεν χρησιμοποιεί αναδρομή

```
Push_copy(stack, tour);  // Tour that visits only the hometown
while (!Empty(stack)) {
   curr_tour = Pop(stack);
   if (City_count(curr_tour) == n) {
      if (Best_tour(curr_tour))
         Update_best_tour(curr_tour);
   } else {
      for (nbr = n-1; nbr >= 1; nbr--)
         if (Feasible(curr_tour, nbr)) {
            Add_city(curr_tour, nbr);
            Push_copy(stack, curr_tour);
            Remove_last_city(curr_tour);
         }
   }
   Free_tour(curr_tour);
}
```

# Χρησιμοποίηση macros προεπεξεργασίας

```
/* Find the ith city on the partial tour */
int Tour_city(tour_t tour, int i) {
   return tour->cities[i];
}  /* Tour_city */
```



```
/* Find the ith city on the partial tour */
#define Tour_city(tour, i) (tour->cities[i])
```

# Χρόνοι εκτέλεσης των τριών ακολουθιακών υλοποιήσεων της τεχνικής της δενδρικής αναζήτησης

| Recursive | First Iterative | Second Iterative |
|---|---|---|
| 30.5 | 29.2 | 32.9 |

(σε δευτερόλεπτα)

Το γράφημα περιέχει 15 πόλεις.

Και οι τρεις υλοποιήσεις επισκέπτονται περίπου 95.000.000 κόμβους δέντρου.

# Υπολογισμός του κόστους του τρέχοντος καλύτερου γύρου πόλεων

- Όταν μία διεργασία ολοκληρώσει ένα γύρο, χρειάζεται να ελέγξει αν έχει την καλύτερη λύση που έχει υπολογισθεί μέχρι εκείνη τη στιγμή.
- Η συνάρτηση global Best_tour διαβάζει μόνο το συνολικά καλύτερο κόστος, και έτσι δεν χρειάζεται να χρησιμοποιήσουμε μηχανισμό κλειδώματος για να εξασφαλίζουμε την ασφαλή πρόσβαση στη μεταβλητή αφού δεν υπάρχει ανταγωνισμός μεταξύ των αναγνωστών.
- Αν η διεργασία δεν έχει καλύτερη λύση, τότε δεν ενημερώνει την μεταβλητή Best_tour.
- Σε περίπτωση που μία άλλη διεργασία ενημερώνει τη μεταβλητή την ίδια χρονική στιγμή, η διεργασία αναγνώστης μπορεί να δει τη παλαιά ή τη νέα τιμή
- Θα ήταν προτιμότερο μία διεργασία να διαβάζει πάντα τη πιο πρόσφατη τιμή άλλα αυτή η εγγύηση έχει σχετικά υψηλό κόστος.
- Στην περίπτωση που μία διεργασία ελέγξει και αποφασίσει ότι έχει καλύτερη συνολικά λύση, χρειάζεται να εξασφαλίσουμε δύο πράγματα:

  1) ότι η διαδικασία κλειδώνει την τιμή Best_tour με μία μεταβλητή mutex, αποτρέποντας τον ανταγωνισμό από τις άλλες διεργασίες.

  2) Στην περίπτωση που ο πρώτος έλεγχος ήταν ως προς μία παλαιά τιμή ενώ εν τω μεταξύ κάποια άλλη διεργασία έχει ενημερώσει τη μεταβλητή, δεν θέλουμε να θέσουμε στη μεταβλητή μία χειρότερη τιμή σε σχέση με τη νεότερη που έχει γραφτεί.

  Χειριζόμαστε αυτές τις περιπτώσεις με μηχανισμό κλειδώματος και επανέλεγχο της μεταβλητής.

# Ψευδοκώδικας για την στατικά παράλληλη λύση του ΠΠΠ με Pthreads

```
Partition_tree(my_rank, my_stack);

while (!Empty(my_stack)) {
   curr_tour = Pop(my_stack);
   if (City_count(curr_tour) == n) {
      if (Best_tour(curr_tour)) Update_best_tour(curr_tour);
   } else {
      for (city = n-1; city >= 1; city--)
         if (Feasible(curr_tour, city)) {
            Add_city(curr_tour, city);
            Push_copy(my_stack, curr_tour);
            Remove_last_city(curr_tour)
         }
   }
   Free_tour(curr_tour);
}
```

# Δυναμική παραλληλοποίηση της αναζήτησης δέντρου χρησιμοποιώντας Pthreads

- Υπάρχουν θέματα σωστού τερματισμού των νημάτων.
- Ο κώδικας που εκτελείται από ένα νήμα πριν μοιράσει την εργασία του έχει ως εξής:
  - Ελέγχει ότι έχει τουλάχιστον δύο γύρους πόλεων στη στοίβα του.
  - Ελέγχει ότι υπάρχουν νήματα που περιμένουν να λάβουν νέα εργασία.
  - Ελέγχει αν η μεταβλητή new_stack έχει τιμή NULL.

# Ψευδοκώδικας για συνάρτηση Terminated - Υλοποίηση με Pthreads

```
if (my_stack_size >= 2 && threads_in_cond_wait > 0 &&
      new_stack == NULL) {
   lock term_mutex;
   if (threads_in_cond_wait > 0 && new_stack == NULL) {
      Split my_stack creating new_stack;
      pthread_cond_signal(&term_cond_var);
   }
   unlock term_mutex;
   return 0;  /* Terminated = False; don't quit */
} else if (!Empty(my_stack)) { /* Stack not empty, keep working */
   return 0;  /* Terminated = false; don't quit */
} else { /* My stack is empty */
   lock term_mutex;
   if (threads_in_cond_wait == thread_count-1) { /* Last thread  */
                                                 /* running      */
      threads_in_cond_wait++;
      pthread_cond_broadcast(&term_cond_var);
      unlock term_mutex;
      return 1;  /* Terminated = true; quit */
```

# Ψευδοκώδικας για συνάρτηση Terminated

```
} else { /* Other threads still working, wait for work */
    threads_in_cond_wait++;
    while (pthread_cond_wait(&term_cond_var, &term_mutex) != 0);
    /* We've been awakened */
    if (threads_in_cond_wait < thread_count) { /* We got work */
        my_stack = new_stack;
        new_stack = NULL;
        threads_in_cond_wait--;
        unlock term_mutex;
        return 0;  /* Terminated = false */
    } else {  /* All threads done */
        unlock term_mutex;
        return 1;  /* Terminated = true; quit */
    }
} /* else wait for work */
} /* else my_stack is empty */
```

Ομαδοποίηση των μεταβλητών τερματισμού:

```
typedef struct {
    my_stack_t new_stack;
    int threads_in_cond_wait;
    pthread_cond_t term_cond_var;
    pthread_mutex_t term_mutex;
} term_struct;
typedef term_struct* term_t;

term_t term;  // global variable
```

Χρόνοι εκτέλεσης των προγραμμάτων με βάση το πρότυπο Pthreads:

Προβλήματα με 15 πόλεις

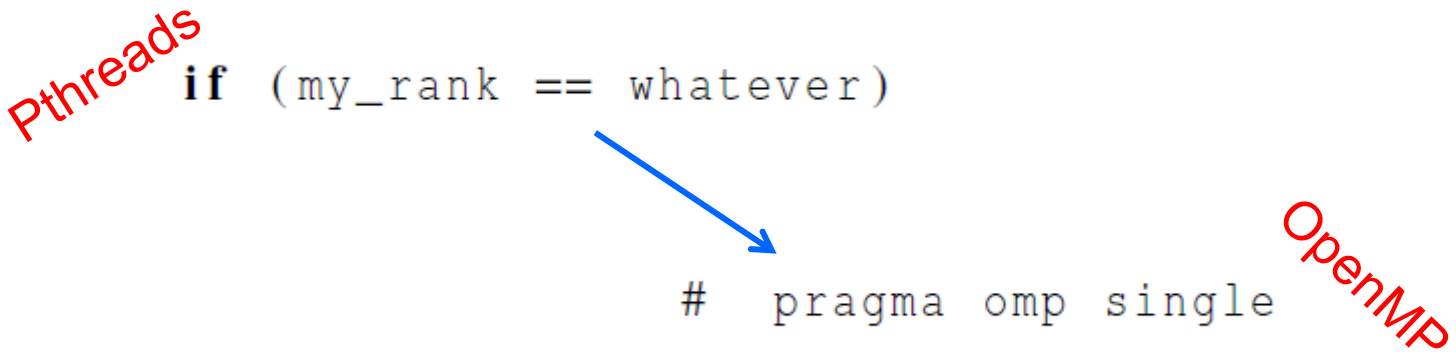
| Threads | First Problem | | | Second Problem | | |
|---|---|---|---|---|---|---|
| | Serial | Static | Dynamic | Serial | Static | Dynamic |
| 1 | 32.9 | 32.7 | 34.7 (0) | 26.0 | 25.8 | 27.5 (0) |
| 2 | | 27.9 | 28.9 (7) | | 25.8 | 19.2 (6) |
| 4 | | 25.7 | 25.9 (47) | | 25.8 | 9.3 (49) |
| 8 | | 23.8 | 22.4 (180) | | 24.0 | 5.7 (256) |

(σε δευτερόλεπτα)

πλήθος χωρισμών
των στοιβών

# Παράλληλη υλοποίηση της αναζήτησης δέντρου με OpenMP

- Τα ίδια θέματα προκύπτουν κατά τη στατική ή δυναμική παράλληλη υλοποίηση της αναζήτησης δέντρου με OpenMP.
- Χρειάζονται μόνο κάποιες μικρές αλλαγές.



Προσομοίωση του condition wait στην OpenMP:

```
/* Global vars */
int awakened_thread = -1;
work_remains = 1;  /* true */
. . .
omp_unset_lock(&term_lock);
while (awakened_thread != my_rank && work_remains);
omp_set_lock(&term_lock);
```

Η απόδοση των υλοποιήσεων αναζήτησης δέντρου με OpenMP και Pthreads:

| Th | First Problem | | | | Second Problem | | | |
|---|---|---|---|---|---|---|---|---|
| | Static | | Dynamic | | Static | | Dynamic | |
| | OMP | Pth | OMP | Pth | OMP | Pth | OMP | Pth |
| 1 | 32.5 | 32.7 | 33.7 (0) | 34.7 (0) | 25.6 | 25.8 | 26.6 (0) | 27.5 (0) |
| 2 | 27.7 | 27.9 | 28.0 (6) | 28.9 (7) | 25.6 | 25.8 | 18.8 (9) | 19.2 (6) |
| 4 | 25.4 | 25.7 | 33.1 (75) | 25.9 (47) | 25.6 | 25.8 | 9.8 (52) | 9.3 (49) |
| 8 | 28.0 | 23.8 | 19.2 (134) | 22.4 (180) | 23.8 | 24.0 | 6.3 (163) | 5.7 (256) |

(σε δευτερόλεπτα)

# Υλοποίηση της αναζήτησης δέντρου χρησιμοποιώντας MPI και στατική διαμέριση

Αποστολή διαφορετικού πλήθους αντικειμένων σε κάθε διαδικασία σε ένα communicator:

```
int MPI_Scatterv(
        void*        sendbuf        /* in  */,
        int*         sendcounts     /* in  */,
        int*         displacements  /* in  */,
        MPI_Datatype sendtype       /* in  */,
        void*        recvbuf        /* out */,
        int          recvcount      /* in  */,
        MPI_Datatype recvtype       /* in  */,
        int          root           /* in  */,
        MPI_Comm     comm           /* in  */)
```

# Συγκέντρωση ενός διαφορετικού πλήθους αντικειμένων από κάθε διεργασία στον communicator

```
int MPI_Gatherv(
      void*         sendbuf        /* in  */,
      int           sendcount      /* in  */,
      MPI_Datatype  sendtype       /* in  */,
      void*         recvbuf        /* out */,
      int*          recvcounts     /* in  */,
      int*          displacements  /* in  */,
      MPI_Datatype  recvtype       /* in  */,
      int           root           /* in  */,
      MPI_Comm      comm           /* in  */)
```

Έλεγχος αν ένα μήνυμα είναι διαθέσιμο:

```
int MPI_Iprobe(
            int          source          /* in  */,
            int          tag             /* in  */,
            MPI_Comm     comm            /* in  */,
            int*         msg_avail_p     /* out */,
            MPI_Status*  status_p        /* out */);
```

```
if (My_avail_tour_count(my_stack) >= 2) {
   Fulfill_request(my_stack);
   return false;  /* Still more work */
} else { /* At most 1 available tour */
   Send_rejects();  /* Tell everyone who's requested */
                    /* work that I have none         */
   if (!Empty_stack(my_stack)) {
      return false;  /* Still more work */
   } else {  /* Empty stack */
      if (comm_sz == 1) return true;
      Out_of_work();
      work_request_sent = false;
      while (1) {
         Clear_msgs();  /* Messages unrelated to work, termination */
         if (No_work_left()) {
            return true;  /* No work left.  Quit */
         } else if (!work_request_sent) {
            Send_work_request();  /* Request work from someone */
            work_request_sent = true;
         } else {
            Check_for_work(&work_request_sent, &work_avail);
            if (work_avail) {
               Receive_work(my_stack);
               return false;
            }
         }
      }  /* while */
   } /* Empty stack */
} /* At most 1 available tour */
```

Η συνάρτηση Terminated για τη δυναμική παραλληλοποίηση της αναζήτησης δέντρου με MPI

- Το MPI παρέχει τέσσερις τύπους για την εντολή αποστολής μηνύματος:
  - Standard
  - Synchronous
  - Ready
  - Buffered

# Εκτύπωση του καλύτερου γύρου

```
struct {
   int cost;
   int rank;
} loc_data, global_data;

loc_data.cost = Tour_cost(loc_best_tour);
loc_data.rank = my_rank;

MPI_Allreduce(&loc_data, &global_data, 1, MPI_2INT, MPI_MINLOC, comm);
if (global_data.rank == 0) return;  /* 0 already has the best tour */
if (my_rank == 0)
   Receive best tour from process global_data.rank;
else if (my_rank == global_data.rank)
   Send best tour to process 0;
```

# Η συνάρτηση Terminated για τη δυναμική παραλληλοποίηση της αναζήτησης δέντρου

```
if (My_avail_tour_count(my_stack) >= 2) {
   Fulfill_request(my_stack);
   return false;  /* Still more work */
} else { /* At most 1 available tour */
   Send_rejects();  /* Tell everyone who's requested */
                    /* work that I have none         */
   if (!Empty_stack(my_stack)) {
      return false;  /* Still more work */
   } else {  /* Empty stack */
      if (comm_sz == 1) return true;
      Out_of_work();
      work_request_sent = false;
      while (1) {
         Clear_msgs();  /* Messages unrelated to work, termination */
         if (No_work_left()) {
            return true;  /* No work left.  Quit */
```

```
        } else if (!work_request_sent) {
            Send_work_request();  /* Request work from someone */
            work_request_sent = true;
        } else {
            Check_for_work(&work_request_sent, &work_avail);
            if (work_avail) {
                Receive_work(my_stack);
                return false;
            }
        }
    }  /* while */
  } /* Empty stack */
} /* At most 1 available tour */
```

# Πακετάρισμα δεδομένων σε buffer συνεχόμενης μνήμης

```
int MPI_Pack(
      void*          data_to_be_packed    /* in      */,
      int            to_be_packed_count   /* in      */,
      MPI_Datatype   datatype             /* in      */,
      void*          contig_buf           /* out     */,
      int            contig_buf_size      /* in      */,
      int*           position_p           /* in/out  */,
      MPI_Comm       comm                 /* in      */)
```

# «Ξεπακετάρισμα» δεδομένων από ένα buffer συνεχόμενης μνήμης

```
int MPI_Unpack(
      void*          contig_buf        /* in     */,
      int            contig_buf_size   /* in     */,
      int*           position_p        /* in/out */,
      void*          unpacked_data     /* out    */,
      int            unpack_count      /* in     */,
      MPI_Datatype   datatype          /* in     */,
      MPI_Comm       comm              /* in     */)
```

# Γεγονότα τερματισμού που οδηγούν σε λάθος

| Time | Process 0 | Process 1 | Process 2 |
|---|---|---|---|
| 0 | Out of Work<br>Notify 1, 2<br>oow = 1 | Out of Work<br>Notify 0, 2<br>oow = 1 | Working<br>oow = 0 |
| 1 | Send request to 1<br>oow = 1 | Send Request to 2<br>oow = 1 | Recv notify fr 1<br>oow = 1 |
| 2 | oow = 1 | Recv notify fr 0<br>oow = 2 | Recv request fr 1<br>oow = 1 |
| 3 | oow = 1 | oow = 2 | Send work to 1<br>oow = 0 |
| 4 | oow = 1 | Recv work fr 2<br>oow = 1 | Recv notify fr 0<br>oow = 1 |
| 5 | oow = 1 | Notify 0<br>oow = 1 | Working<br>oow = 1 |
| 6 | oow = 1 | Recv request fr 0<br>oow = 1 | Out of work<br>Notify 0, 1<br>oow = 2 |
| 7 | Recv notify fr 2<br>oow = 2 | Send work to 0<br>oow = 0 | Send request to 1<br>oow = 2 |
| 8 | Recv 1st notify fr 1<br>oow = 3 | Recv notify fr 2<br>oow = 1 | oow = 2 |
| 9 | Quit | Recv request fr 2<br>oow = 1 | oow = 2 |

# Απόδοση υλοποιήσεων MPI και Pthreads

| Th/Pr | First Problem | | | | Second Problem | | | |
|---|---|---|---|---|---|---|---|---|
| | Static | | Dynamic | | Static | | Dynamic | |
| | Pth | MPI | Pth | MPI | Pth | MPI | Pth | MPI |
| 1 | 35.8 | 40.9 | 41.9 (0) | 56.5 (0) | 27.4 | 31.5 | 32.3 (0) | 43.8 (0) |
| 2 | 29.9 | 34.9 | 34.3 (9) | 55.6 (5) | 27.4 | 31.5 | 22.0 (8) | 37.4 (9) |
| 4 | 27.2 | 31.7 | 30.2 (55) | 52.6 (85) | 27.4 | 31.5 | 10.7 (44) | 21.8 (76) |
| 8 | | 35.7 | | 45.5 (165) | | 35.7 | | 16.5 (161) |
| 16 | | 20.1 | | 10.5 (441) | | 17.8 | | 0.1 (173) |

(σε δευτερόλεπτα)