Άπληστη Μέθοδος

Εισαγωγή (1/2)

Η άπληστη μέθοδος στοχεύει στη λύση που φαίνεται η βέλτιστη μέχρι στιγμής

Επιλέγει την τοπικά βέλτιστη λύση με την ελπίδα ότι αυτή η λύση θα είναι τελικά η βέλτιστη

Μπορούμε να συνθέσουμε μια ολικά βέλτιστη λύση μέσα από την επιλογή τοπικών βέλτιστων λύσεων

Οι άπληστοι αλγόριθμοι δεν δίνουν πάντα τις βέλτιστες

Πρόκειται για μια ισχυρή μέθοδο που χρησιμοποιείται σε μεγάλο εύρος προβλημάτων

Εισαγωγή (2/2)

Διαφορές με το δυναμικό προγραμματισμό

- Στην άπληστη μέθοδο επιλέγουμε την τοπικά βέλτιστη λύση χωρίς να εστιάζουμε στις λύσεις των υπο-προβλημάτων
- Οι επιλογές που κάνουμε στο δυναμικό προγραμματισμό βασίζονται στις λύσεις των υπο-προβλημάτων
- Τυπικά στο δυναμικό προγραμματισμό ακολουθούμε μια bottom-up προσέγγιση όπου από πιο απλά προβλήματα φτάνουμε στη λύση πιο σύνθετων προβλημάτων
- Οι επιλογές στην άπληστη μέθοδο μπορεί να βασίζονται σε παλιές επιλογές αλλά ποτέ δεν βασίζονται σε μελλοντικές
- Ο δυναμικός προγραμματισμός επιλύει τα υπο-προβλήματα πριν κάνει μια επιλογή, ενώ η άπληστη μέθοδος κάνει την επιλογή πριν λύσει τα υπο-προβλήματα

Κώδικες Huffman (1/19)

Οι κώδικες Huffman στοχεύουν στη συμπίεση δεδομένων

Θεωρούμε τα δεδομένα ως ακολουθίες χαρακτήρων

Η προσέγγιση με την άπληστη μέθοδο υιοθετεί ένα πίνακα που αποθηκεύει πόσο συχνά εμφανίζεται ο κάθε ένας χαρακτήρας

Απεικονίζει κάθε χαρακτήρα σαν ένα binary string

Κώδικες Huffman (2/19)

Παράδειγμα

	a	b	c	d	e	f
Frequency (in thousands)	45	13	12	16	9	5
Fixed-length codeword	000	001	010	011	100	101
Variable-length codeword	0	101	100	111	1101	1100

Με την fixed length μέθοδο χρειαζόμαστε 300.000 bits για ένα αρχείο των 100.000 χαρακτήρων

Με την variable length μέθοδο χρειαζόμαστε

$$(45 \cdot 1 + 13 \cdot 3 + 12 \cdot 3 + 16 \cdot 3 + 9 \cdot 4 + 5 \cdot 4) \cdot 1{,}000 = 224{,}000 \text{ bits}$$

Κώδικες Huffman (3/19)

Εστιάζουμε στα prefix codes, κώδικες που δεν είναι προθέματα άλλων κωδίκων

Η τελική κωδικοποίηση είναι η συγχώνευση των κωδίκων για κάθε χαρακτήρα

Αν το prefix code είναι μοναδικό για κάθε χαρακτήρα τότε η αποκωδικοποίηση είναι εύκολη

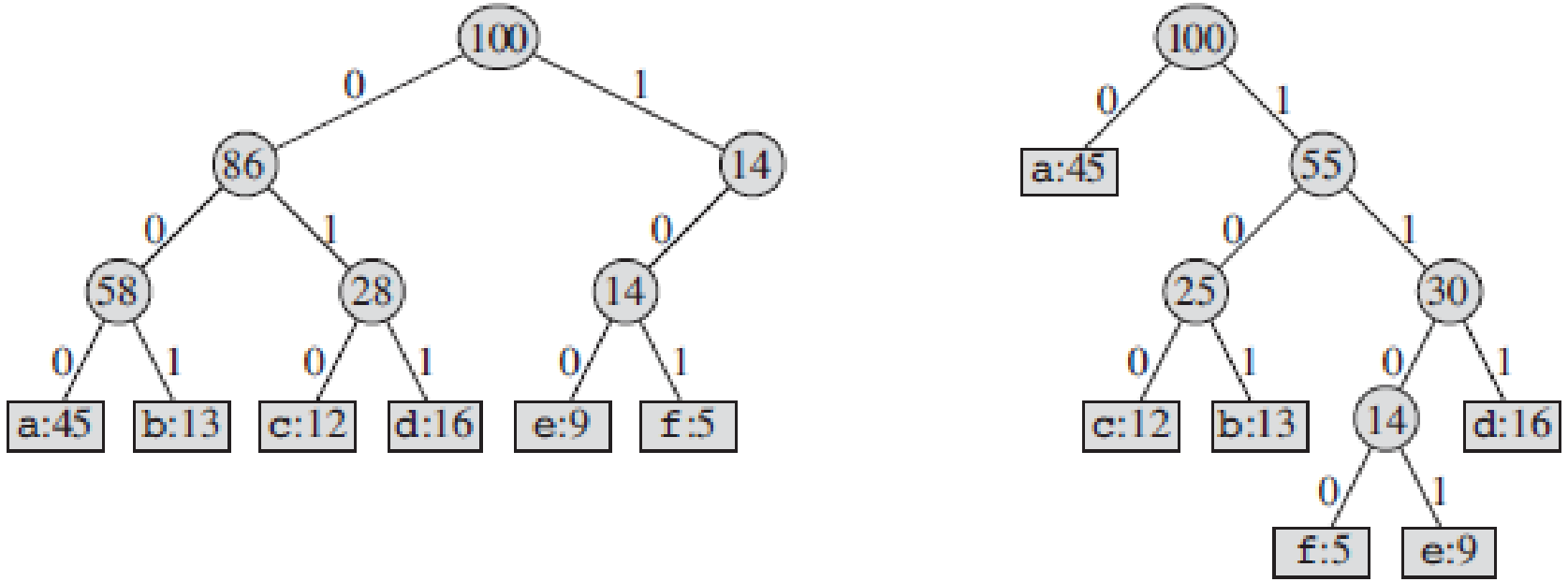
Υιοθετούμε ένα δυαδικό δένδρο στο οποίο τα φύλλα είναι οι χαρακτήρες

Ο δυαδικός κώδικας είναι η διαδρομή από τη ρίζα του δένδρου προς τα φύλλα

Κώδικες Huffman (4/19)

Το 0 αντιστοιχεί στην κίνηση προς το αριστερό παιδί, ενώ το 1 την κίνηση προς το δεξιό παιδί

Παράδειγμα



Κώδικες Huffman (5/19)

Ο βέλτιστος κώδικας εμπλέκει ένα πλήρες δυαδικό δένδρο στο οποίο ο κάθε κόμβος έχει δύο παιδιά

Η fixed length προσέγγιση δεν οδηγεί σε βέλτιστο κώδικα

Έστω C το αλφάβητο για το οποίο πρέπει να κατασκευάσουμε τον κώδικα

Το δένδρο έχει |C| φύλλα

Το δένδρο έχει ακριβώς |C|-1 εσωτερικούς κόμβους

Κώδικες Huffman (6/19)

Δοσμένου ενός δένδρου Τ που αντιστοιχεί σε ένα κώδικα, μπορούμε να υπολογίσουμε εύκολα το πλήθος των bits που απαιτούνται για να κωδικοποιήσουμε το αρχείο

Έστω c ένας χαρακτήρας του C, c.freq είναι η συχνότητα του c και $d_T(c)$ είναι το βάθος του c στο δένδρο

Το $d_T(c)$ είναι ταυτόχρονα και το μήκος του κώδικα του c

Το πλήθος των bits θα είναι

$$B(T) = \sum_{c \in C} c.freq \cdot d_T(c)$$

Κώδικες Huffman (7/19)

Υποθέτουμε ότι το C είναι ένα σύνολο n χαρακτήρων

Ο άπληστος αλγόριθμος ξεκινά από τα φύλλα και εκτελεί |C|-1 συγχωνεύσεις για να δημιουργήσει το τελικό δένδρο

Υιοθετεί μια ουρά ελάχιστης προτεραιότητας (min-priority queue)

Μέσω της ουράς αναγνωρίζει τα δύο ελάχιστα συχνά στοιχεία για να συγχωνεύσει

Όταν γίνεται η συγχώνευση το αποτέλεσμα είναι ένα νέο στοιχείο του οποίου η συχνότητα είναι το άθροισμα των συχνοτήτων των δύο στοιχείων

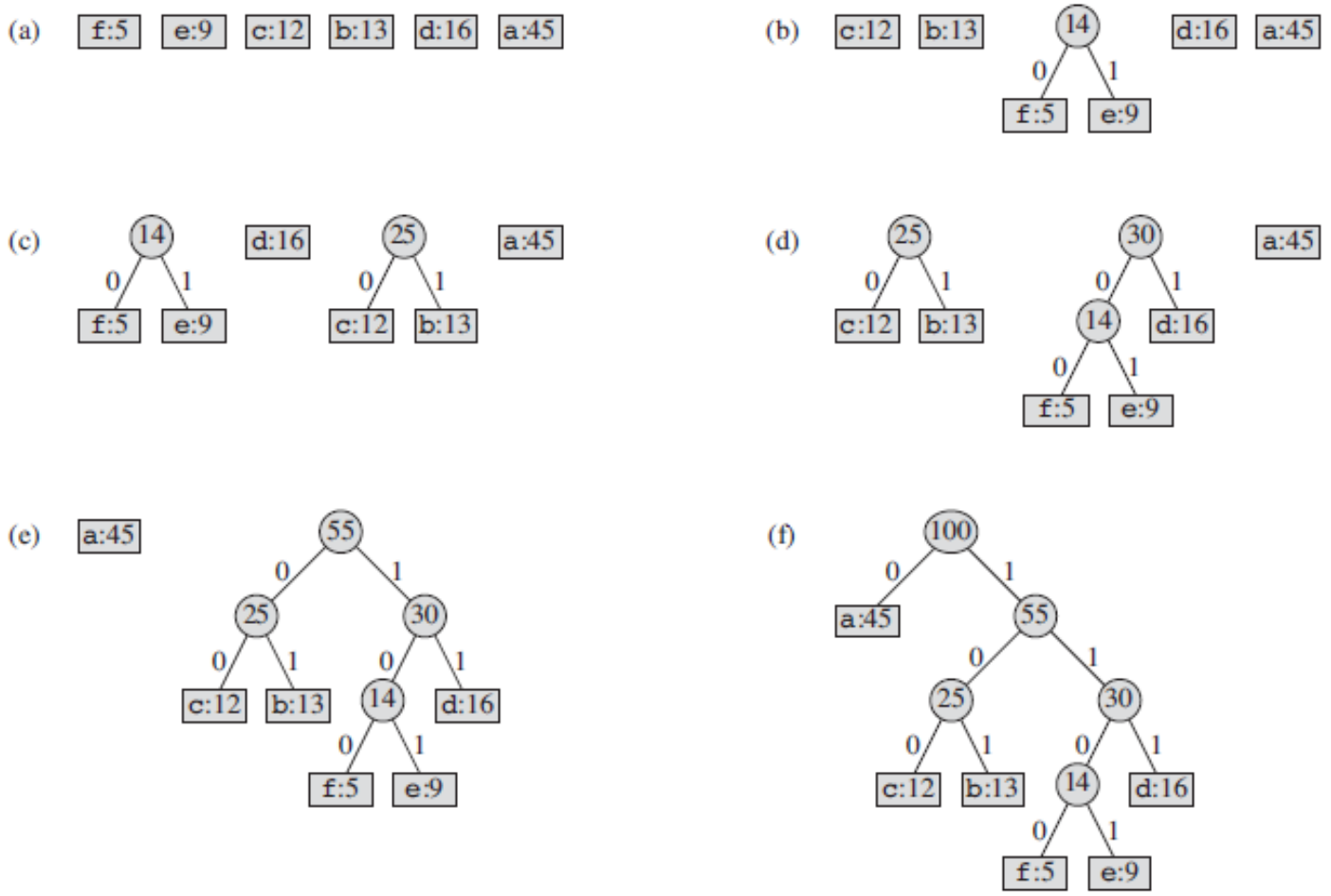
Κώδικες Huffman (8/19)

Αλγόριθμος

- Η αρχική ουρά έχει μέγεθος ίσο με το n
- Οι γραμμές 3-8 εξάγουν επαναληπτικά τους δύο κόμβους x, y με την ελάχιστη συχνότητα αντικαθιστώντας τους στην ουρά με ένα νέο κόμβο z
- Η συχνότητα του z είναι το άθροισμα των συχνοτήτων
- Ο z έχει αριστερό παιδί το x και δεξιό παιδί το y

```
HUFFMAN(C)
1  n = |C|
2  Q = C
3  for i = 1 to n − 1
4      allocate a new node z
5      z.left = x = EXTRACT-MIN(Q)
6      z.right = y = EXTRACT-MIN(Q)
7      z.freq = x.freq + y.freq
8      INSERT(Q, z)
9  return EXTRACT-MIN(Q)      // return the root of the tree
```

Κώδικες Huffman (9/19)



Κώδικες Huffman (10/19)

Lemma 1

Για ένα αλφάβητο C όπου ο κάθε χαρακτήρας c έχει συχνότητα c.freq, αν x, y είναι οι δύο χαρακτήρες με τη χαμηλότερη συχνότητα, τότε υπάρχει ένα βέλτιστο πρόθεμα (prefix) στο οποίο οι κώδικες για τα x, y έχουν το ίδιο μήκος αλλά διαφέρουν στο τελευταίο bit

Απόδειξη

Έστω a, b δύο χαρακτήρες που είναι αδέρφια σε μέγιστο βάθος στο T.

Υποθέτουμε ότι $a.freq \leq b.freq$ και $x.freq \leq y.freq$.

Κώδικες Huffman (11/19)

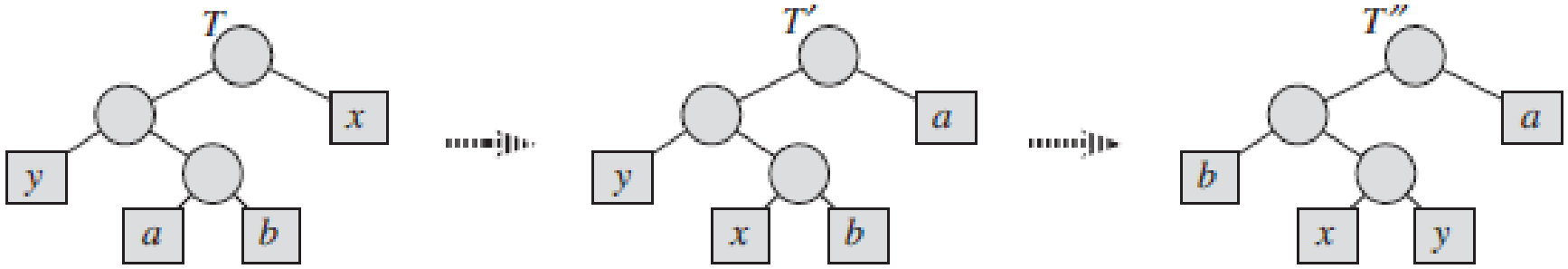
Αφού τα x.freq & y.freq είναι οι ελάχιστες συχνότητες, και a.freq & b.freq είναι δύο συχνότητες χαρακτήρων θα έχουμε x.freq<=a.freq & y.freq<=b.freq

Αν x.freq=a.freq & y.freq=b.freq τότε θα έχουμε x.freq=a.freq=y.freq=b.freq οπότε το λήμμα ισχύει

Αν υποθέσουμε ότι x.freq<>b.freq, τότε θα είναι x<>b

Προχωρούμε σε αλλαγή στο δένδρο με αντιμεταθέσεις θέσεων στα a,x & b,y ως ακολούθως:

Κώδικες Huffman (12/19)

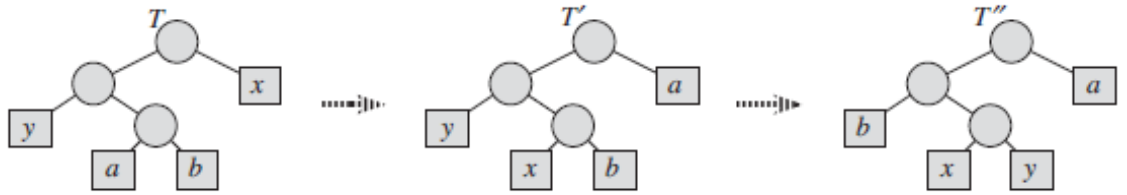


Αν x=b & y<>a το T’’ δεν έχει τα x,y σαν ‘αδέρφια’

Η διαφορά στο κόστος των δένδρων T, T’ είναι:

Κώδικες Huffman (13/19)

$$
\begin{aligned}
& B(T) - B(T') \\
&= \sum_{c \in C} c.freq \cdot d_T(c) - \sum_{c \in C} c.freq \cdot d_{T'}(c) \\
&= x.freq \cdot d_T(x) + a.freq \cdot d_T(a) - x.freq \cdot d_{T'}(x) - a.freq \cdot d_{T'}(a) \\
&= x.freq \cdot d_T(x) + a.freq \cdot d_T(a) - x.freq \cdot d_T(a) - a.freq \cdot d_T(x) \\
&= (a.freq - x.freq)(d_T(a) - d_T(x)) \\
&\geq 0
\end{aligned}
$$



διότι τα a.freq-x.freq & $d_T(a)$-$d_T(x)$ είναι μη αρνητικά

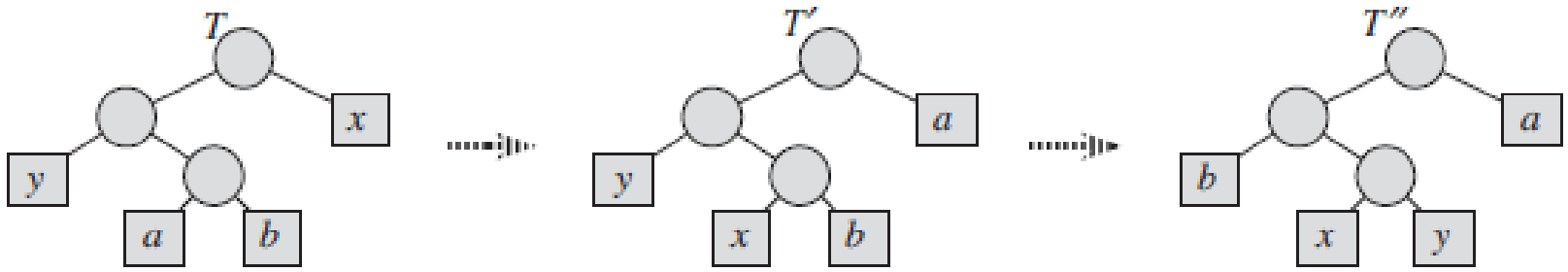
Το a.freq-x.freq είναι μη αρνητικό διότι το x έχει ελάχιστη συχνότητα και το $d_T(a)$-$d_T(x)$ είναι μη αρνητικό αφού το a είναι ένα φύλλο μέγιστου βάθους

Κώδικες Huffman (14/19)

Αν αντιμεταθέσουμε τα y, b, το κόστος δεν αυξάνει, οπότε το B(T')-B(T'') είναι μη αρνητικό

Συνεπώς, B(T'')<=B(T) και αφού το T είναι το βέλτιστο έχουμε B(T)<=B(T'') το οποίο μας δίνει ότι B(T'')=B(T)

Οπότε το T'' είναι βέλτιστο και τα x, y είναι αδέρφια



Κώδικες Huffman (15/19)

Lemma 2

Έστω C είναι ένα αλφάβητο με συχνότητα c.freq για κάθε χαρακτήρα c του αλφαβήτου. Έστω x,y δύο χαρακτήρες με την ελάχιστη συχνότητα. Έστω C’ ένα αλφάβητο χωρίς τα x,y και με ένα νέο χαρακτήρα z τέτοιο ώστε C’=C-{x,y}U{z} και z.freq=x.freq+y.freq. Έστω T’ οποιοδήποτε δένδρο που αναπαριστά τον κώδικα του βέλτιστου προθέματος για το αλφάβητο C’. Τότε το δένδρο T το οποίο εξάγεται από το T’ αντικαθιστώντας το φύλο z με ένα εσωτερικό κόμβο που έχει παιδιά τα x, y, αναπαριστά ένα βέλτιστο κώδικα για το αλφάβητο C.

Κώδικες Huffman (16/19)

Απόδειξη

Για κάθε χαρακτήρα $c \in C - \{x, y\}$.

Έχουμε $d_T(c) = d_{T'}(c)$

Και έτσι $c.freq \cdot d_T(c) = c.freq \cdot d_{T'}(c)$

Αφού $d_T(x) = d_T(y) = d_{T'}(z) + 1$

Έχουμε

$$\begin{aligned} x.freq \cdot d_T(x) + y.freq \cdot d_T(y) &= (x.freq + y.freq)(d_{T'}(z) + 1) \\ &= z.freq \cdot d_{T'}(z) + (x.freq + y.freq) \end{aligned}$$

Κώδικες Huffman (17/19)

Συμπεραίνουμε πως $B(T) = B(T') + x.freq + y.freq$

Ή $B(T') = B(T) - x.freq - y.freq$

Έστω τώρα ότι το Τ δεν αναπαριστά ένα βέλτιστο κώδικα

Υπάρχει ένα Τ'' τέτοιο ώστε $B(T'') < B(T)$

Το Τ'' έχει τα x, y ως αδέρφια

Έστω το Τ''' το δένδρο Τ'' με το κοινό πατέρα των x, y που έχουν αντικατασταθεί από το z με συχνότητα z.freq=x.freq+y.freq

Κώδικες Huffman (18/19)

Τότε

$$\begin{aligned} B(T''') &= B(T'') - x.freq - y.freq \\ &< B(T) - x.freq - y.freq \\ &= B(T') \end{aligned}$$

Το οποίο δεν ισχύει αφού το Τ’ αναπαριστά ένα βέλτιστο κώδικα του C’

Έτσι το Τ είναι ο βέλτιστος κώδικας του C

Κώδικες Huffman (19/19)

Θεώρημα

Η διαδικασία HUFFMAN παράγει ένα βέλτιστο κώδικα

Απόδειξη

Εξάγεται εύκολα από τα δύο προηγούμενα λήματα

Γράφοι

Εισαγωγή (1/2)

Τα προβλήματα που εμπλέκουν γράφους είναι πολύ σημαντικά για διάφορους τομείς της Επιστήμης των ΗΥ

Ένας γράφος ουσιαστικά απεικονίζεται με το συμβολισμό G=(V,E) όπου V είναι οι κορυφές και E είναι οι ακμές που τις συνδέουν

Όταν χαρακτηρίζουμε το χρόνο εκτέλεσης / πολυπλοκότητα αλγορίθμων πάνω σε γράφους, μετράμε το μέγεθος της εισόδου σε σχέση με το πλήθος των κορυφών |V| και το πλήθος των ακμών |E|

Άρα μελετούμε την επίδραση δύο παραμέτρων και όχι μιας

Παράδειγμα: O(V E) ή O(|V||E|)

Εισαγωγή (2/2)

Δύο τρόποι αναπαράστασης

- **Συλλογή λιστών γειτνίασης** (collection of adjacency lists)
- **Πίνακας γειτνίασης** (adjacency matrix)

Οι λίστες γειτνίασης αποτελούν πιο συμπαγή παρουσίαση ιδιαίτερα των **αραιών (sparse)** γράφων

Οι πίνακες γειτνίασης είναι προτιμότεροι όταν οι γράφοι είναι **πυκνοί (dense)**

Αραιοί γράφοι: το $|E|$ είναι κατά πολύ μικρότερο του $|V|^2$

Πυκνοί γράφοι: το $|E|$ είναι κοντά στο $|V|^2$

Λίστες Γειτνίασης (1/2)

Για ένα γράφο G=(V,E), χρησιμοποιούμε ένα πίνακα |V| λιστών, μια για κάθε κόμβο / κορυφή του G

Για κάθε u ϵ V, η u θέση του πίνακα περιέχει μια **συνδεδεμένη λίστα** με τους κόμβους v για τους οποίους υπάρχει μια ακμή τέτοια ώστε (u,v) ϵ E

Αν ο G είναι **κατευθυνόμενος**, τότε το άθροισμα του μήκους όλων των λιστών γειτνίασης είναι ίσο με |E|

Αν ο G είναι **μη κατευθυνόμενος**, τότε το άθροισμα του μήκους όλων των λιστών γειτνίασης είναι ίσο με 2|E|

Λίστες Γειτνίασης (2/2)

Για όλους τους γράφους, η διατήρηση των λιστών γειτνίασης απαιτεί ποσότητα μνήμης ίση με **Θ(V+E)**

Οι λίστες γειτνίασης μπορούν εύκολα να χρησιμοποιηθούν ώστε να αποτυπώσουν **γράφους με βάρη** (αποθηκεύουμε απλά το βάρος)

- Κάθε ακμή έχει ένα βάρος το οποίο τυπικά αποδίδεται μέσω μιας συνάρτησης $w : E \rightarrow \mathbb{R}$

Μειονεκτήματα

- Δεν είναι πολύ γρήγορες στην αποτύπωση του αν υπάρχει μια ακμή που συνδέει δύο κόμβους (οι πίνακες γειτνίασης είναι πιο καλή δομή για την επίλυση του συγκεκριμένου προβλήματος)

Πίνακες Γειτνίασης (1/2)

Υποθέτουμε πως οι κόμβοι αριθμούνται με 1,2,…,|V|

Ο πίνακας γειτνίασης έχει μέγεθος **|V|X|V|**

Έστω A=(a_{ij}) ο πίνακας γειτνίασης. Ισχύει ότι:

$$a_{ij} = \begin{cases} 1 & \text{if } (i, j) \in E \\ 0 & \text{otherwise} \end{cases}$$

Ο πίνακας γειτνίασης απαιτεί επιπλέον μνήμη της τάξης του **Θ(V^2)** (προσέξτε ότι δεν επηρεάζεται από το πλήθος των ακμών)

Πρόκειται για ένα συμμετρικό πίνακα

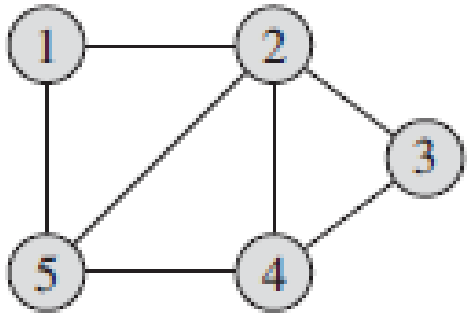
Πίνακες Γειτνίασης (2/2)

Για την αναπαράσταση ενός γράφου με βάρη, απλά αποθηκεύουμε το βάρος και όχι 0 ή 1

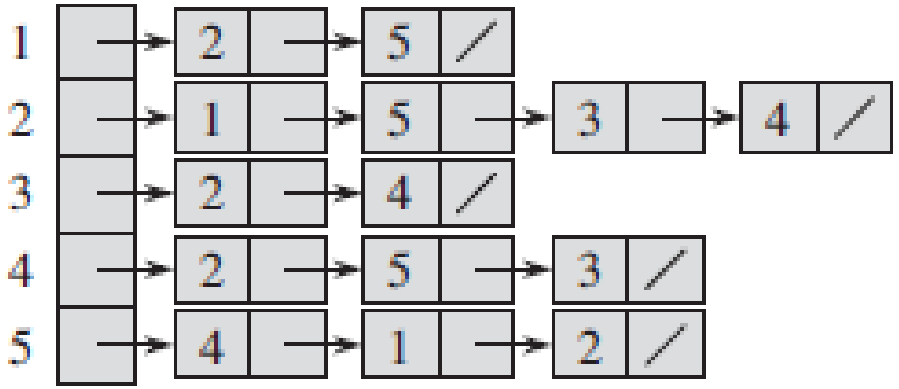
Σε μη κατευθυνόμενους γράφους για να ‘γλιτώσουμε’ χώρο επιπλέον μνήμης, αποθηκεύουμε μόνο τα στοιχεία πάνω από τη διαγώνιο (αφού είναι ίδια με τα στοιχεία κάτω από τη διαγώνιο)

Πρόκειται για πιο απλή δομή σε σχέση με τις λίστες γειτνίασης

Παράδειγμα



(a)

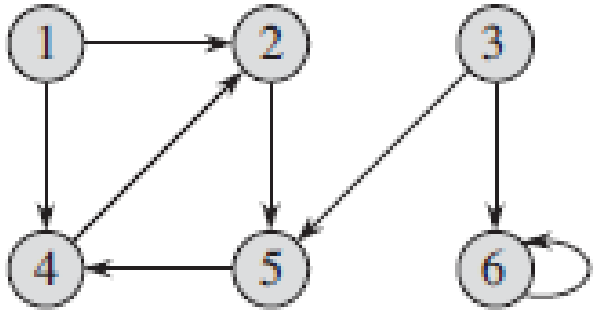


(b)

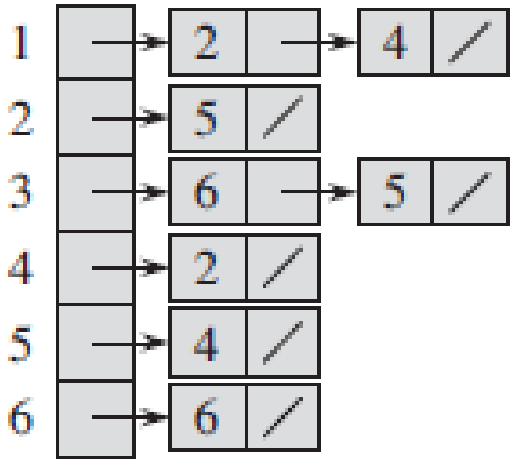
	1	2	3	4	5
1	0	1	0	0	1
2	1	0	1	1	1
3	0	1	0	1	0
4	0	1	1	0	1
5	1	1	0	1	0

(c)

Παράδειγμα



(a)



(b)

	1	2	3	4	5	6
1	0	1	0	1	0	0
2	0	0	0	0	1	0
3	0	0	0	0	1	1
4	0	1	0	0	0	0
5	0	0	0	1	0	0
6	0	0	0	0	0	1

(c)

Breadth-First Search (1/18)

Η **αναζήτηση κατά πλάτος** (**breadth first search - BFS**) είναι ο πιο εύκολος αλγόριθμος αναζήτησης σε γράφους

Οι αλγόριθμοι του **Prim** & **Dijkstra** χρησιμοποιούν ιδέες όμοιες με του BFS

Δοσμένων ενός γράφου G και ενός κόμβου έναρξης s, ο BFS προσπαθεί να ανακαλύψει κάθε κόμβο που μπορεί να προσπελαστεί από τον s

Υπολογίζει την απόσταση από τον s προς κάθε άλλο προσπελάσιμο κόμβο

Breadth-First Search (2/18)

Επίσης, παράγει ένα δένδρο (breadth-first tree) με ρίζα τον s

Για κάθε κόμβο v που μπορεί να προσπελαστεί από τον s το **απλούστερο μονοπάτι** στο δένδρο από το s στον v αντιστοιχεί στο **συντομότερο μονοπάτι** από τον s στον v πάνω στο γράφο G

Ο αλγόριθμος λειτουργεί είτε για κατευθυνόμενους ή μη κατευθυνόμενους γράφους

Ο αλγόριθμος ανακαλύπτει τους κόμβους που μπορούν να προσπελαστούν από τον s σε απόσταση k πριν ανακαλύψει τους κόμβους που βρίσκονται σε απόσταση k+1

Breadth-First Search (3/18)

Ο αλγόριθμος ‘χρωματίζει’ τους κόμβους με τα εξής χρώματα: **white, gray, black**

Όλοι οι κόμβοι ξεκινούν με λευκό χρώμα και στη συνέχεια μπορεί να γίνουν είτε γκρι είτε μαύρο

Όταν ένας κόμβος ανακαλύπτεται για πρώτη φορά τότε χάνει το λευκό χρώμα

Οι γκρι και μαύροι κόμβοι έχουν ανακαλυφθεί από τον αλγόριθμο αλλά ο διαφορετικός χρωματισμός εξασφαλίζει ότι η αναζήτηση γίνεται κατά πλάτος

Breadth-First Search (4/18)

Αν (u,v) ϵ Ε και ο u είναι ένας **μαύρος** κόμβος, τότε ο v είναι είτε **γκρι** ή **μαύρος**

Όλοι οι κόμβοι που γειτνιάζουν με μαύρους κόμβους έχουν ανακαλυφθεί από τον αλγόριθμο

Οι γκρι κόμβοι μπορεί να γειτνιάζουν με κάποιους λευκούς και αναπαριστούν το όριο ανάμεσα σε κόμβους που έχουμε ανακαλύψει και σε αυτούς που δεν έχουμε δει ακόμα

Breadth-First Search (5/18)

Το δένδρο αναζήτησης κατασκευάζεται με ρίζα τον κόμβο s

Κάθε φορά που ένας λευκός κόμβος v ανακαλύπτεται, μέσω ενός κόμβου u, ο v και η ακμή (u,v) εισάγονται στο δένδρο

Ο u είναι ο ‘πατέρας’ (predecessor or parent) του v στο δένδρο

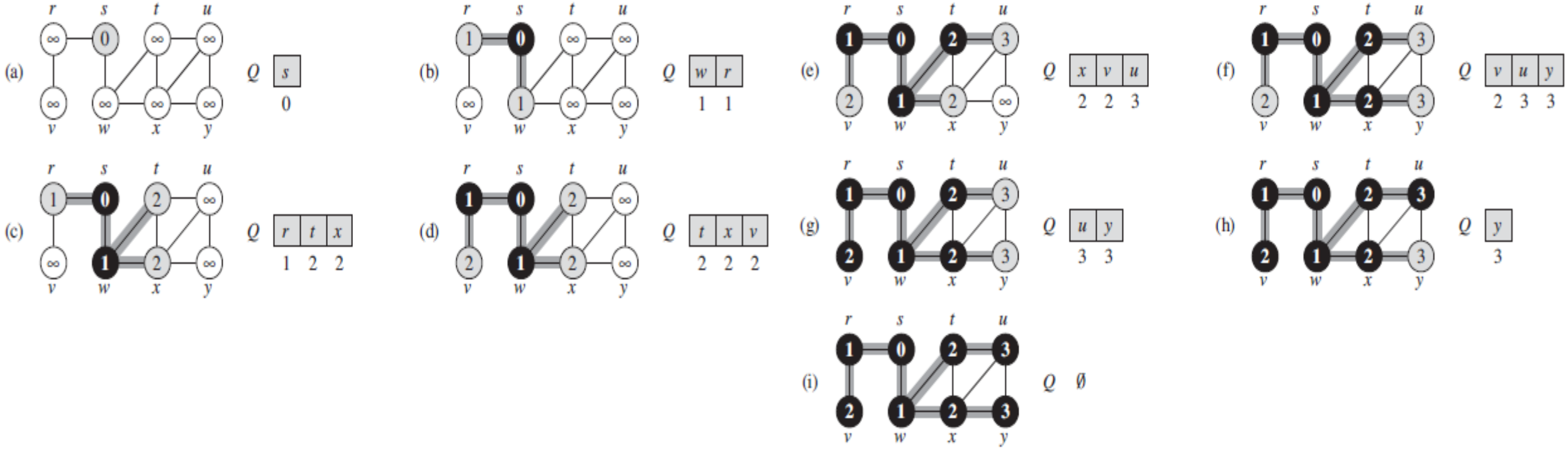
Αφού ο κάθε κόμβος ανακαλύπτεται μόνο μια φορά, θα έχει μόνο ένα πατέρα στο δένδρο

Breadth-First Search (6/18)

```
BFS(G, s)
 1  for each vertex u ∈ G.V − {s}
 2      u.color = WHITE
 3      u.d = ∞
 4      u.π = NIL
 5  s.color = GRAY
 6  s.d = 0
 7  s.π = NIL
 8  Q = ∅
 9  ENQUEUE(Q, s)
10  while Q ≠ ∅
11      u = DEQUEUE(Q)
12      for each v ∈ G.Adj[u]
13          if v.color == WHITE
14              v.color = GRAY
15              v.d = u.d + 1
16              v.π = u
17              ENQUEUE(Q, v)
18      u.color = BLACK
```

Breadth-First Search (7/18)

Παράδειγμα



Breadth-First Search (8/18)

Ανάλυση

- Κάθε κόμβος μπαίνει στην ουρά μόνο μια φορά
- Ο χρόνος εισαγωγής / εξαγωγής στην ουρά είναι O(1)
- Συνολικός χρόνος για τις πράξεις που αφορούν στην ουρά είναι O(V)
- Κάθε φορά που εξάγεται ένας κόμβος από την ουρά, τότε οι γείτονες προσπελαύνονται μόνο μια φορά
- Αφού το άθροισμα τους μήκους όλων των λιστών γειτνίασης είναι Θ(E), ο χρόνος για την προσπέλαση όλων των λιστών είναι O(E)
- Συνεπώς, ο συνολικός χρόνος του BFS είναι O(V+E)
- Γραμμικός χρόνος εκτέλεσης

```
BFS(G, s)
 1  for each vertex u ∈ G.V − {s}
 2       u.color = WHITE
 3       u.d = ∞
 4       u.π = NIL
 5  s.color = GRAY
 6  s.d = 0
 7  s.π = NIL
 8  Q = ∅
 9  ENQUEUE(Q, s)
10  while Q ≠ ∅
11       u = DEQUEUE(Q)
12       for each v ∈ G.Adj[u]
13            if v.color == WHITE
14                 v.color = GRAY
15                 v.d = u.d + 1
16                 v.π = u
17                 ENQUEUE(Q, v)
18       u.color = BLACK
```

Breadth-First Search (9/18)

Ας υποθέσουμε ότι η μικρότερη απόσταση (shortest-path distance) από τη ρίζα του δένδρου μέχρι ένα κόμβο ν είναι **δ(s,v)**

Το συντομότερο μονοπάτι (shortest path) είναι το μικρότερο πλήθος των ακμών που προσπελαύνουμε για να φτάσουμε από το s στο ν

Αν δεν υπάρχει κάποιο μονοπάτι που να συνδέει τους κόμβους τότε $\delta(s,v) = \infty$

Breadth-First Search (10/18)

Lemma 1

Αν G=(V,E) είναι ένας γράφος (κατευθυνόμενος ή μη) και s ϵ V είναι ένας κόμβος. Τότε για κάθε ακμή (u,v) ϵ E ισχύει δ(s,v) <= δ(s,u) +1

Απόδειξη

Αν ο u μπορεί να προσπελαστεί μέσω του s τότε το ίδιο ισχύει και για τον v. Τότε το συντομότερο μονοπάτι από τον s στον v δεν μπορεί να είναι μικρότερο από το μονοπάτι που συνδέει τον s με τον u (λόγω της ακμής (u,v)). Αν ο v δεν μπορεί να προσπελαστεί από τον s, τότε **δ(s,v) = ∞**. Και στις δύο περιπτώσεις η αρχική ανίσωση ισχύει.

Breadth-First Search (11/18)

Lemma 2

Αν G=(V,E) είναι ένας γράφος (κατευθυνόμενος ή μη) και ο BFS έχει εκτελεστεί με έναρξη ένα κόμβο s ϵ V. Στον τερματισμό του αλγορίθμου, για κάθε ν ϵ V, ισχύει v.d >= δ(s,v)

Απόδειξη

Υιοθετούμε επαγωγή. Εστιάζουμε στην εξαγωγή στοιχείων από την ουρά. Η ανίσωση ισχύει για την εξαγωγή στοιχείων από την ουρά αφού για το πρώτο στοιχείο s έχουμε **s.d=0=δ(s,s)** και για κάθε ν ϵ V-{s}, **v.d = ∞ >= δ(s,v)**.

Ας εστιάσουμε τώρα σε κάποιο κόμβο ν που έχει προσπελαστεί μέσω του u. Σε αυτή την περίπτωση ισχύει **ότι u.d>=δ(s,u)** οπότε **v.d=u.d+1>=δ(s,u)+1>= δ(s,v)**. Η τιμή v.d δεν αλλάζει στη συνέχεια αφού αν βγει από την ουρά, τότε δεν ξαναβγαίνει.

Breadth-First Search (12/18)

Lemma 3

Ας υποθέσουμε ότι σε κάποια στιγμή εκτέλεσης του BFS στην ουρά υπάρχουν τα στοιχεία (v_1, v_2, …, v_r) με το v_1 να είναι στην κορυφή της ουράς. Τότε, $v_r.d <= v_1.d+1$ και $v_i.d <= v_{i+1}.d$ για i=1,2,..,r-1

Breadth-First Search (13/18)

Proof of Lemma 3

Υιοθετούμε επαγωγή. Αν το v_1 βγει από την ουρά το v_2 γίνεται η νέα κορυφή της ουράς. Μέσω της επαγωγής έχουμε: $\mathbf{v_1.d <= v_2.d}$ και $\mathbf{v_r.d <= v_1.d+1 <= v_2.d}$. Στη συνέχεια το Λήμμα ισχύει για το v_2.

Όταν βάζουμε ένα στοιχείο ν τότε αυτό γίνεται το v_{r+1}. Τότε έχουμε ήδη βγάλει το u για το οποίο εξετάζουμε τη λίστα γειτνίασης. Με βάση την επαγωγή, η νέα κορυφή v_1 θα έχει $\mathbf{v_1.d >= u.d}$. Οπότε $\mathbf{v_{r+1}.d = v.d = u.d+1 <= v_1.d+1}$. Από την επαγωγική υπόθεση έχουμε πως $\mathbf{v_r.d <= u.d+1}$, συνεπώς $\mathbf{v_r.d <= u.d+1 = v_r.d = v_{r+1}.d}$. Οπότε το λήμμα ισχύει όταν το στοιχείο βγαίνει από την ουρά.

Breadth-First Search (14/18)

Πρόταση

Για δύο κόμβους v_i, v_j με τον v_i να έχει εισαχθεί στην ουρά πριν τον v_j ισχύει $v_i.d <= v_j.d$ όταν το v_j βγαίνει από την ουρά.

Θεώρημα

Έστω G=(V,E) είναι ένας γράφος (κατευθυνόμενος ή μη) και ο BFS έχει εκτελεστεί με έναρξη ένα κόμβο s ϵ V. Κατά τη διάρκεια εκτέλεσης, ο BFS ανακαλύπτει κάθε κόμβο v ϵ V που μπορεί να προσπελαστεί από τον s και στον τερματισμό ισχύει v.d=δ(s,v) για κάθε v ϵ V. Για κάθε v≠s, ένα από τα συντομότερα μονοπάτια από το s στο v είναι ένα συντομότερο μονοπάτι από το s στο v.π ακολουθούμενο από την ακμή (v.π,v)

Breadth-First Search (15/18)

Απόδειξη

Ας υποθέσουμε ότι κάποιοι κόμβοι δεν έχουν απόσταση ίση με το ελάχιστο μονοπάτι. Έστω ν ο κόμβος με το δ(s,v) που παίρνει αυτή την τιμή. Προφανώς v≠s. Από το Λήμμα 2 έχουμε **v.d>= δ(s,v)** και **v.d> δ(s,v)**. Ο ν πρέπει να προσπελαύνεται από τον s, αν όχι τότε δ(s,v)=∞>= v.d. Αν u είναι ο αμέσως προηγούμενος κόμβος στο συντομότερο μονοπάτι με **δ(s,v)= δ(s,u)+1**. Αφού **δ(s,u)<δ(s,v)**, έχουμε **u.d= δ(s,u)**.

Άρα $v.d > \delta(s, v) = \delta(s, u) + 1 = u.d + 1$

Breadth-First Search (16/18)

Απόδειξη (συνέχεια)

Ας υποθέσουμε τώρα πως ο αλγόριθμος εξάγει τον u. Σε αυτή τη στιγμή ο ν είναι λευκός ή γκρι ή μαύρος. Σε κάθε περίπτωση η εξίσωση $v.d > \delta(s, v) = \delta(s, u) + 1 = u.d + 1$ δεν ισχύει. Αν ο ν είναι **λευκός** τότε θα έχουμε ν.d=u.d+1 (αντίθετα με την εξίσωση). Αν ο ν είναι **μαύρος** έχει ήδη βγει από την ουρά και με βάση την προηγούμενη Πρόταση έχουμε **v.d<=u.d** (αντίθετα με την εξίσωση). Αν είναι **γκρι**, τότε αυτό έχει γίνει διότι έχει εξαχθεί από την ουρά ένας άλλος κόμβος w πριν από τον u οπότε **v.d=w.d+1**. Από την Πρόταση 1 έχουμε όμως πως **w.d<=u.d**, οπότε **v.d=w.d+1<=u.d+1** (αντίθετα με την εξίσωση).

Breadth-First Search (17/18)

Απόδειξη (συνέχεια)

Από τα προηγούμενα συμπεραίνουμε πως **v.d=δ(s,v)** για κάθε v ϵ V. Όλοι οι κόμβοι v που προσπελαύνονται μέσω του s μπορούν ν ανακαλυφθούν αλλιώς θα έχουν **v.d=∞> δ(s,v)**. Για να συμπεράνουμε τελικά το θεώρημα, παρατηρούμε πως αν **v.π=u** τότε **v.d=u.d+1**. Τελικά, μπορούμε να έχουμε το συντομότερο μονοπάτι από το s στο ν παίρνοντας το συντομότερο μονοπάτι από το s στο ν.π και διασχίζοντας την ακμή (ν.π,ν).

Breadth-First Search (18/18)

Η εκτύπωση του δένδρου που φτιάχνει ο BFS τυπώνεται με τον ακόλουθο αλγόριθμο.

```
PRINT-PATH(G, s, v)
1  if v == s
2      print s
3  elseif v.π == NIL
4      print "no path from" s "to" v "exists"
5  else PRINT-PATH(G, s, v.π)
6      print v
```

Demo

https://visualgo.net/en/dfsbfs

Depth-First Search (1/13)

Ο αλγόριθμος **αναζήτησης κατά βάθος (depth first search – DFS)** αναζητά τα στοιχεία στο γράφο προχωρώντας όσο πιο ‘κάτω’ στο γράφο μπορεί

Ακολουθεί ακμές που προκύπτουν από τους πιο πρόσφατα αναζητούμενους κόμβους

Όταν όλες οι ακμές ενός κόμβου έχουν ελεγχθεί τότε οπισθοχωρεί και συνεχίζει από τον επόμενο κόμβο

Αυτή η διαδικασία συνεχίζεται μέχρι να ανακαλυφθούν όλοι οι κόμβοι που μπορούν να προσπελαστούν από τον s

Depth-First Search (2/13)

Όταν ο αλγόριθμος ανακαλύπτει ένα κόμβο v μέσω του u (μέσω της λίστας γειτνίασης) καταγράφει το v.π=u

Αντίθετα με τον BFS όπου ο υπογράφος που ορίζεται με τον πατρικό κόμβο είναι ένα δένδρο, στον DFS ο υπογράφο που ορίζεται από τον πατρικό κόμβο μπορεί να αποτελείται από πολλά δένδρα

Ο υπογράφος του πατρικού κόμβου ορίζει ένα ‘δάσος’ (depth first forest) αποτελούμενο από πολλά DFS trees

Ο αλγόριθμος υιοθετεί επίσης χρώματα: **λευκό**, **γκρι** όταν έχει ανακαλυφθεί και **μαύρο** όταν η επεξεργασία του κόμβου έχει τελειώσει

Depth-First Search (3/13)

Ο αλγόριθμος χρησιμοποιεί **χρονοσφραγίδες** (timestamps) για κάθε κόμβο

Κάθε κόμβος έχει δύο χρονοσφραγίδες

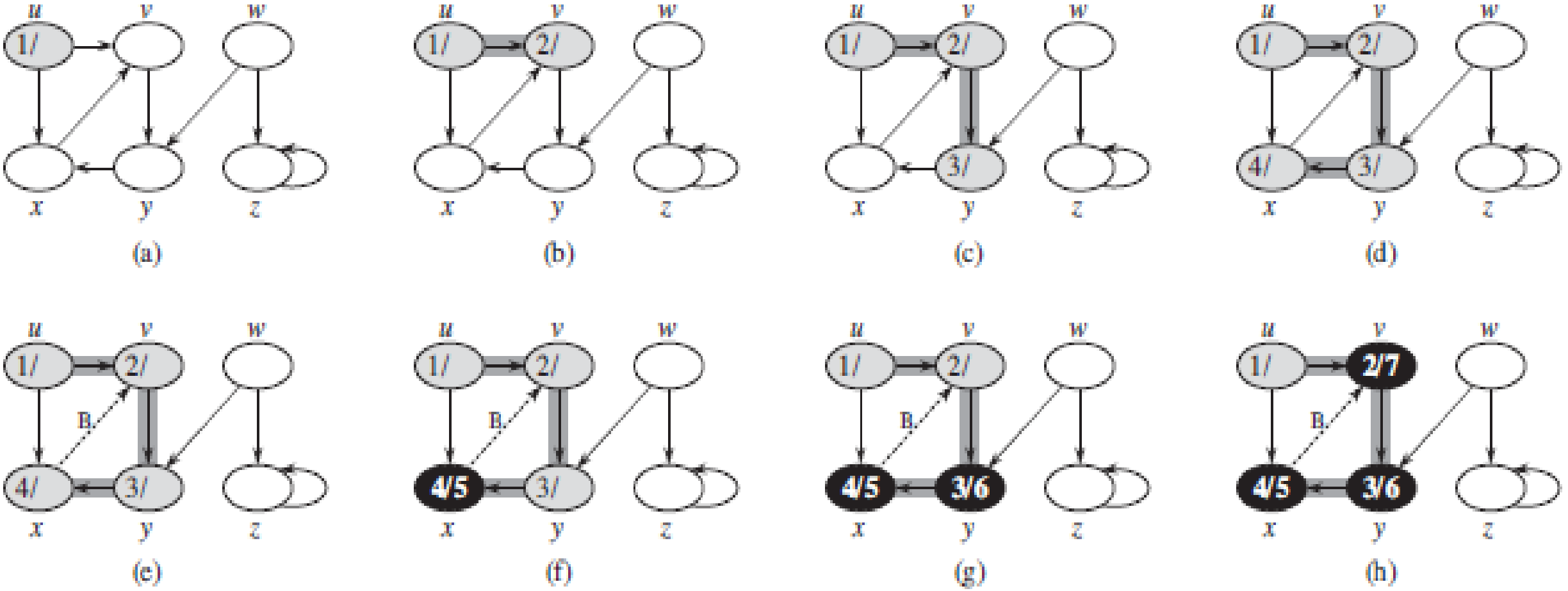
- Η v.d αποθηκεύει πότε ο ν έχει ανακαλυφθεί (έχει γίνει γκρι)
- Η v.f αποθηκεύει πότε ο ν έχει ολοκληρώσει (έχει τελειώσει η επεξεργασία των λιστών γειτνίασης και έχει γίνει μαύρος)
- Πρόκειται για ακέραιους στο διάστημα [1,2|V|]
- Επίσης, ισχύει u.d<u.f
- Ένας κόμβος u είναι λευκός πριν το u.d, γκρι ανάμεσα στα u.d, u.f και μαύρος μετά το u.f

Depth-First Search (4/13)

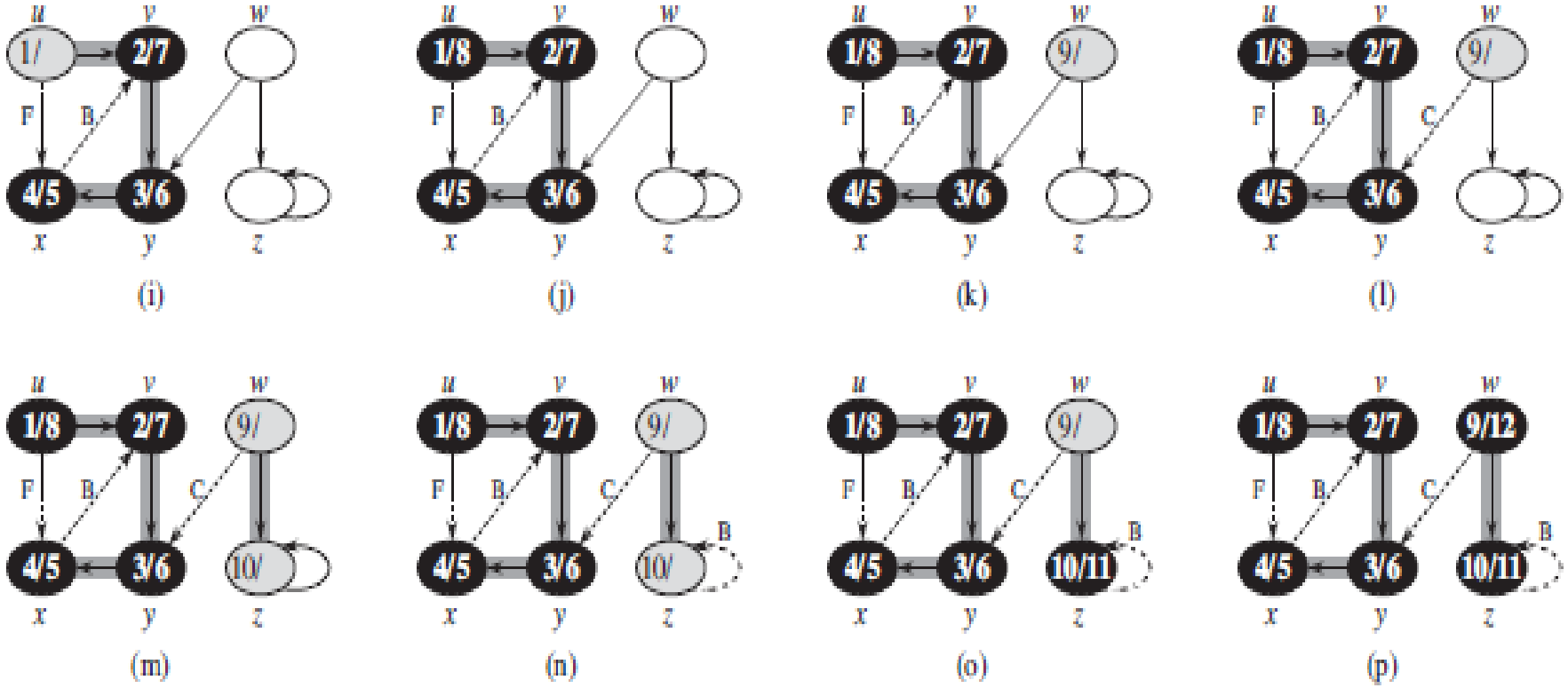
```
DFS(G)
1  for each vertex u ∈ G.V
2      u.color = WHITE
3      u.π = NIL
4  time = 0
5  for each vertex u ∈ G.V
6      if u.color == WHITE
7          DFS-VISIT(G, u)
```

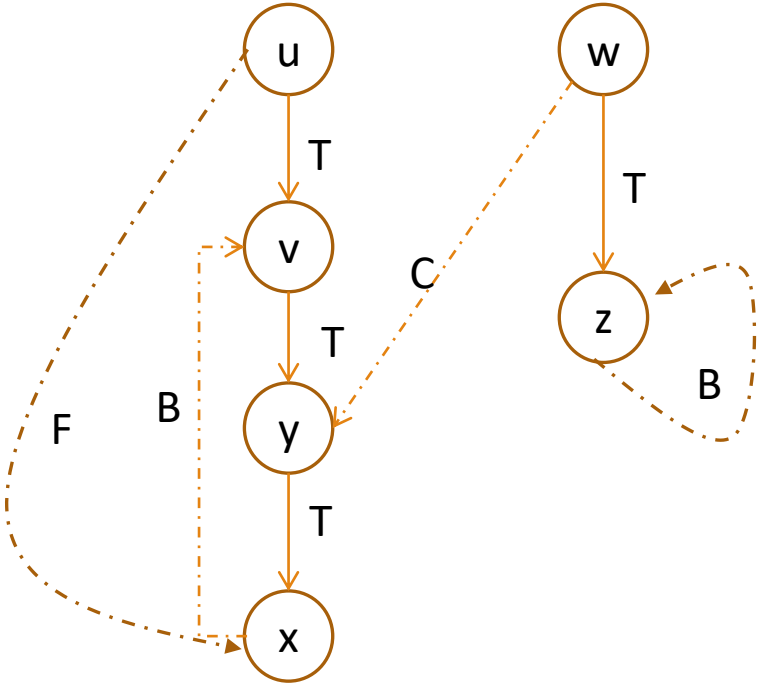
```
DFS-VISIT(G, u)
1   time = time + 1 // white vertex u has just been discovered
2   u.d = time
3   u.color = GRAY
4   for each v ∈ G.Adj[u] // explore edge (u, v)
5       if v.color == WHITE
6           v.π = u
7           DFS-VISIT(G, v)
8   u.color = BLACK // blacken u; it is finished
9   time = time + 1
10  u.f = time
```

Depth-First Search (5/13)



Depth-First Search (6/13)





Depth-First Search (7/13)

Ανάλυση

- Ο αρχικός ‘χρωματισμός’ των κόμβων απαιτεί Θ(V)
- Η visit καλείται μόνο μια φορά για κάθε κόμβο
- Η εκτέλεση της visit γίνεται |Adj[v]| φορές
- Αφού το άθροισμα όλων των |Adj[v]| για κάθε v ϵ V είναι Θ(E), το κόστος όλων των κλήσεων της visit είναι Θ(E)
- Ο συνολικός χρόνος του αλγορίθμου είναι Θ(V+E)

Depth-First Search (8/13)

Parenthesis Theorem

Στον DFS που εφαρμόζεται πάνω σε ένα γράφο G=(V,E), για οποιουσδήποτε δύο κόμβους u, v, ισχύει ακριβώς μια από τις ακόλουθες περιπτώσεις:

- ***Το διάστημα [v.d,v.f] περιλαμβάνεται στο [u.d,u.f] και ο v είναι απόγονος του u***
- ***Τα διαστήματα [u.d,u.f] & [v.d,v.f] είναι διαφορετικά και κανείς από τους u, v είναι απόγονος του άλλου***
- ***Το διάστημα [u.d,u.f] περιλαμβάνεται στο [v.d,v.f] και ο u είναι απόγονος του v***

Depth-First Search (9/13)

Απόδειξη

Ξεκινάμε από την περίπτωση όπου u.d<v.d. Υπάρχουν 2 υπο-περιπτώσεις: v.d<u.f ή όχι.

Αν **v.d<u.f**, τότε ο ν ανακαλύφθηκε όταν ο u ήταν γκρι οπότε ο ν είναι απόγονος του u. Αφού ο ν ανακαλύφθηκε πιο πρόσφατα όλες οι ακμές του έχουν προσπελαστεί και ο ν είναι μαύρος πριν από τον u. Τότε, το [v.d,v.f] περιλαμβάνεται εξ' ολοκλήρου στο [u.d,u.f].

Depth-First Search (10/13)

Απόδειξη (συνέχεια)

Στη δεύτερη υπο-περίπτωση, **u.f<v.d** και επειδή **u.d<u.f** έχουμε **u.d<u.f<v.d<v.f** οπότε τα διαστήματα **[u.d,u.f], [v.d,v.f]** είναι διαφορετικά και disjoint μεταξύ τους. Αφού είναι disjoint κανένας κόμβος από τους δύο έχει ανακαλυφθεί όταν ο άλλος ήταν γκρι, συνεπώς, κανένας κόμβος δεν είναι απόγονος του άλλου.

Η τρίτη περίπτωση είναι παραπλήσια με τα προηγούμενα, απλά **αντιστρέφουμε τους ρόλους** για τους κόμβους u, v.

Depth-First Search (11/13)

Πρόταση

Ένας κόμβος v είναι απόγονος ενός κόμβου u όταν και μόνο όταν ισχύει u.d<v.d<v.f<u.f

Θεώρημα

Στον DFS ο κόμβος v είναι ένας απόγονος του κόμβου u όταν και μόνο όταν σε χρόνο u.d όπου ανακαλύπτεται ο u υπάρχει ένα μονοπάτι από τον u στον v που αποτελείται αποκλειστικά από λευκούς κόμβους.

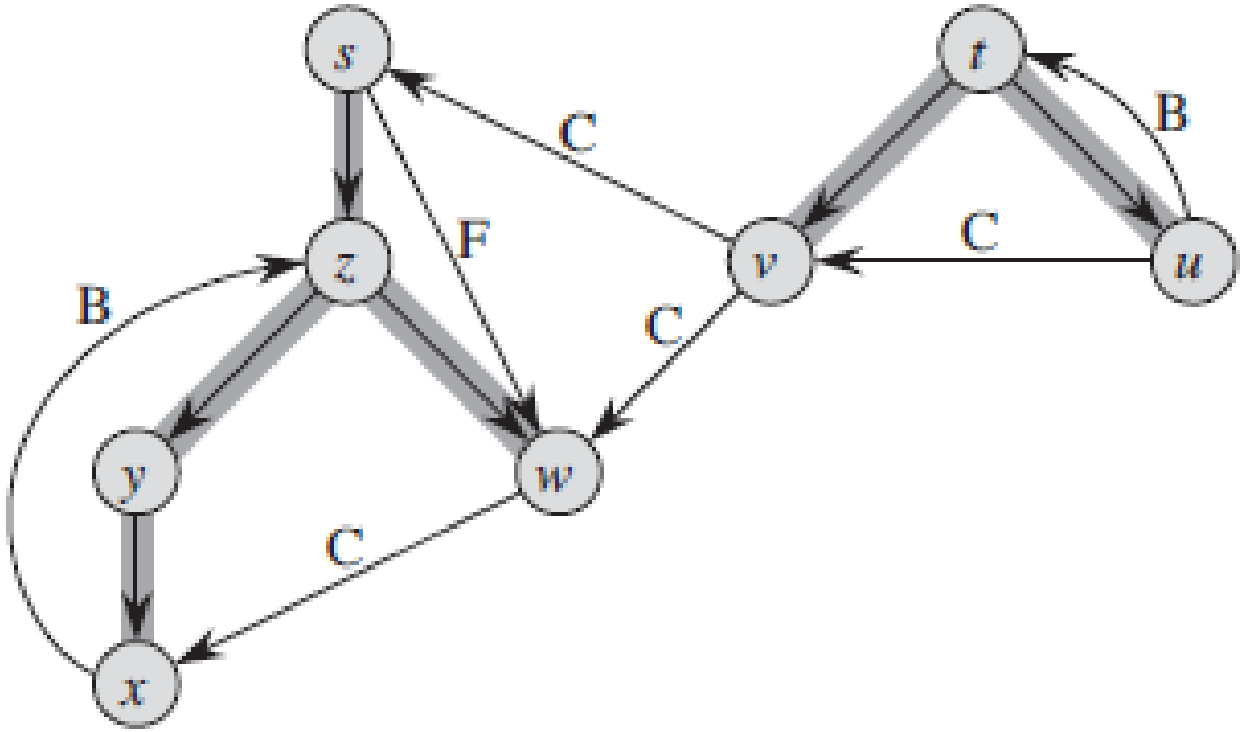
Depth-First Search (12/13)

Κατάταξη των ακμών

- Οι ακμές δίνουν σημαντικές πληροφορίες για ένα γράφο: ένας κατευθυνόμενος γράφος είναι **μη κυκλικός / ακυκλικός** όταν και μόνο όταν στον DFS δεν ανακαλύπτονται ακμές οπισθοχώρησης (back).
- Οι ακμές κατατάσσονται σε:
  - **Tree**: Η ακμή (u,v) είναι δενδρική ακμή όταν ο v έχει πρώτος ανακαλυφθεί
  - **Back**: Η (u,v) είναι ακμή οπισθοχώρησης όταν συνδέει ένα κόμβο με τον πρόγονό του
  - **Forward**: Η (u,v) είναι ακμή προώθησης όταν συνδέει ένα κόμβο με ένα απόγονό του
  - **Cross**: Η (u,v) είναι ακμή διασταύρωσης όταν συνδέει κόμβους στο ίδιο υπο-δένδρο χωρίς οι κόμβοι να έχουν σχέση πρόγονου – απόγονου ή συνδέουν κόμβους σε διαφορετικά υπο-δένδρα

Depth-First Search (13/13)

Παράδειγμα



Demo

https://visualgo.net/en/dfsbfs

Minimum Spanning Trees (1/6)

Ορισμένες φορές επιθυμούμε πάνω σε ένα γράφο να βρούμε ένα **μη κυκλικό υποσύνολο ακμών** που να συνδέουν όλους τους κόμβους

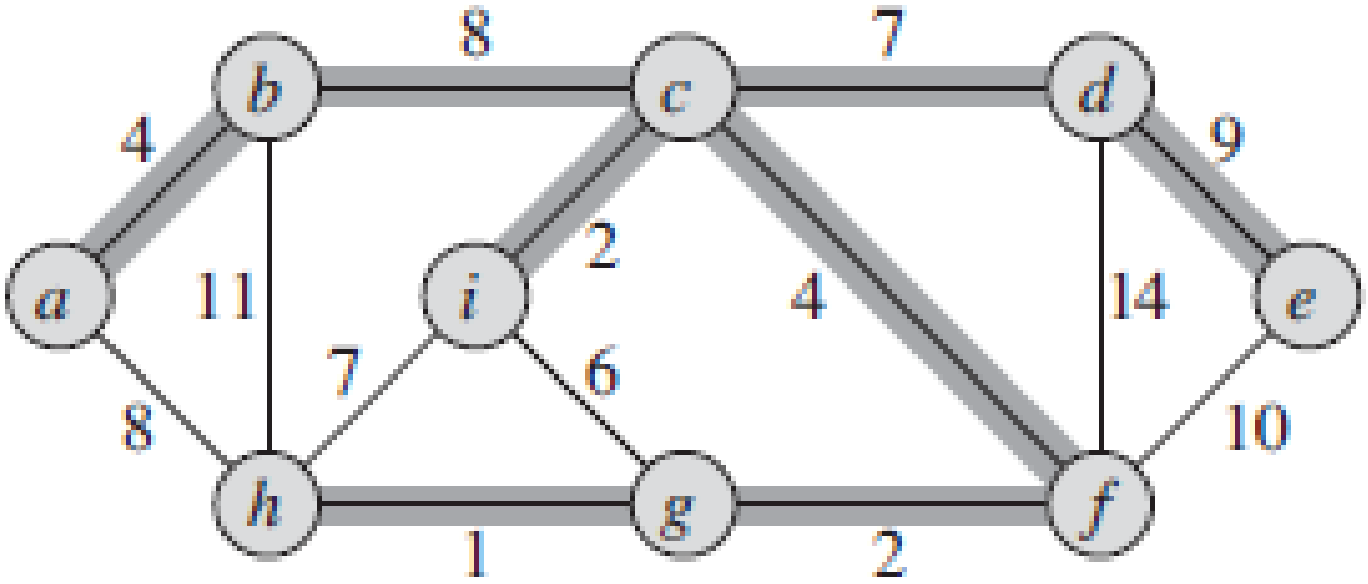
Το άθροισμα των βαρών θέλουμε να είναι το ελάχιστο

Δημιουργείται ένα δένδρο

Το δένδρο ονομάζεται το **δένδρο επικάλυψης ελάχιστου κόστους (minimum spanning tree - MST)**

Minimum Spanning Trees (2/6)

Παράδειγμα



Minimum Spanning Trees (3/6)

Έστω γράφος G=(V,E) και μια συνάρτηση βάρους w:E → R

Πρόβλημα

- Εύρεση του MST

Τεχνικές

- Άπληστη
  - Το MST μεγαλώνει κατά μια ακμή κάθε φορά
  - Η ακμή προστίθεται σε ένα σύνολο ακμών A που είναι επίσης ένα MST

Minimum Spanning Trees (4/6)

Αλγόριθμος

```
GENERIC-MST(G, w)
1  A = ∅
2  while A does not form a spanning tree
3      find an edge (u, v) that is safe for A
4      A = A ∪ {(u, v)}
5  return A
```

Πρόβλημα: πως θα βρούμε μια ακμή που είναι ασφαλές να την τοποθετήσουμε στο MST?

Minimum Spanning Trees (5/6)

Λύση

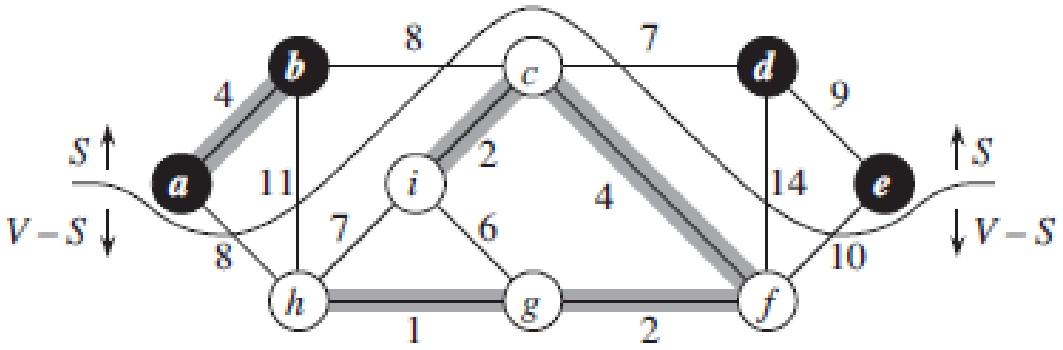
- Για ένα οποιοδήποτε διαχωρισμό των κόμβων του γράφου (S,V-S) λέμε πως μια ακμή (u,v) διασχίζει τα δύο τμήματα αν ένα από τα ακραία σημεία – κόμβοι είναι στο S και το άλλο είναι στο V-S
- Μια light ακμή, είναι η ακμή που συνδέει τα δύο τμήματα του γράφου και έχει το μικρότερο βάρος από όλες που επίσης συνδέουν τα δύο τμήματα

Θεώρημα

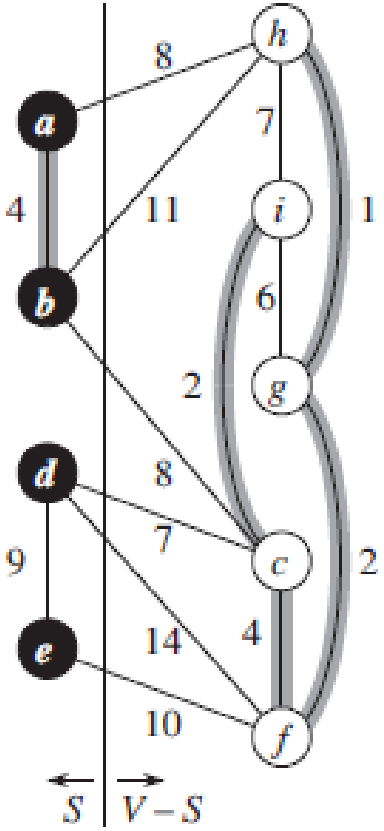
Αν A είναι το τρέχον MST για ένα γράφο G και (S,V-S) είναι ένας οποιοσδήποτε διαχωρισμός του γράφου, η light ακμή (u,v) που συνδέει τα S, V-S είναι ασφαλής για να περιληφθεί στο A.

Minimum Spanning Trees (6/6)

Παράδειγμα



(a)



(b)

Kruskal's Algorithm (1/4)

Ο αλγόριθμος βρίσκει κάθε φορά μια ασφαλή ακμή για να την τοποθετήσει στο MST

Ο αλγόριθμος αρχικά δημιουργεί |V| δένδρα που περιέχουν μόνο ένα κόμβο

Ταξινομεί τις ακμές σε αύξουσα σειρά βάρους

```
MST-KRUSKAL(G, w)
1  A = ∅
2  for each vertex v ∈ G.V
3       MAKE-SET(v)
4  sort the edges of G.E into nondecreasing order by weight w
5  for each edge (u, v) ∈ G.E, taken in nondecreasing order by weight
6       if FIND-SET(u) ≠ FIND-SET(v)
7            A = A ∪ {(u, v)}
8            UNION(u, v)
9  return A
```

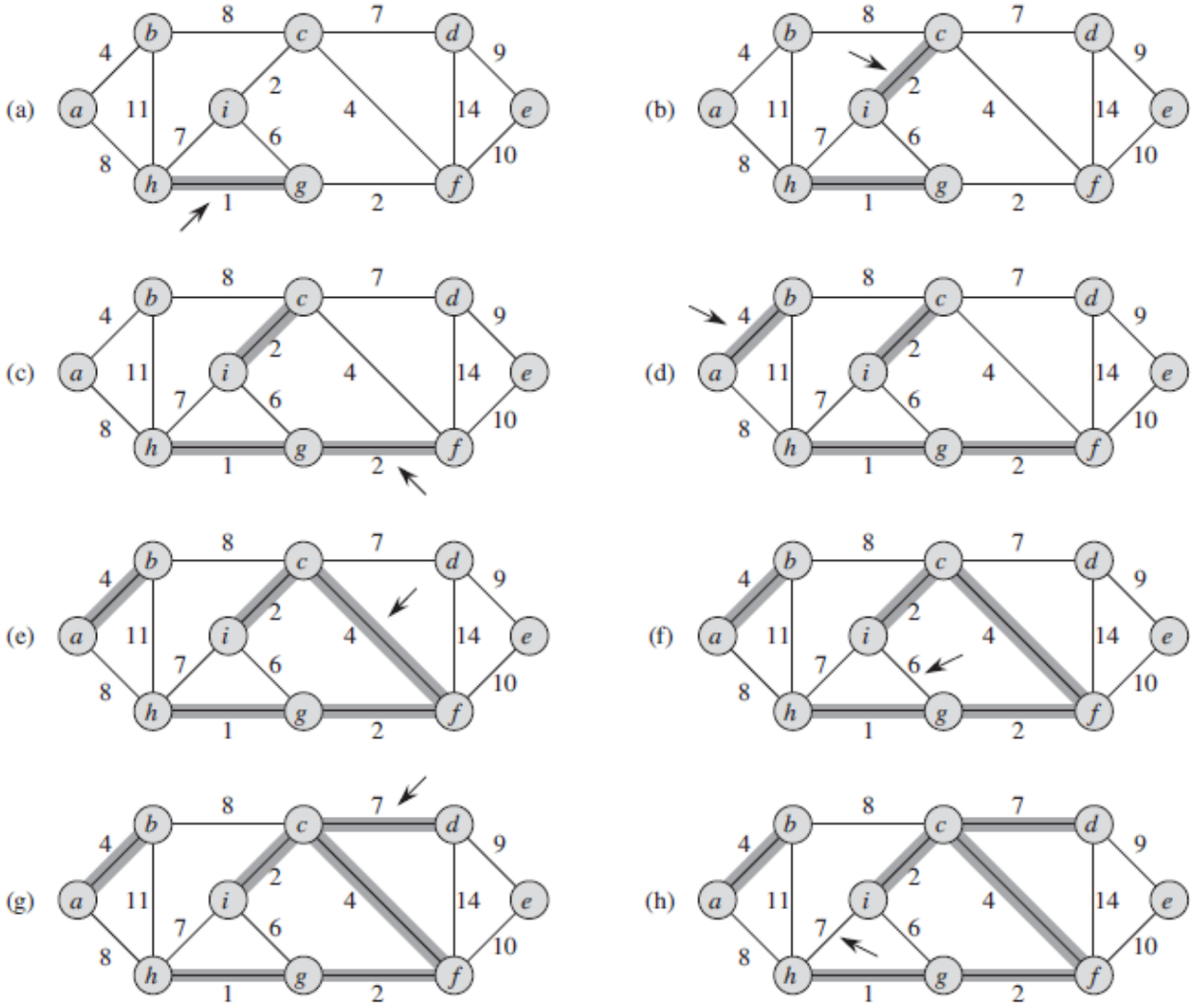
Kruskal's Algorithm (2/4)

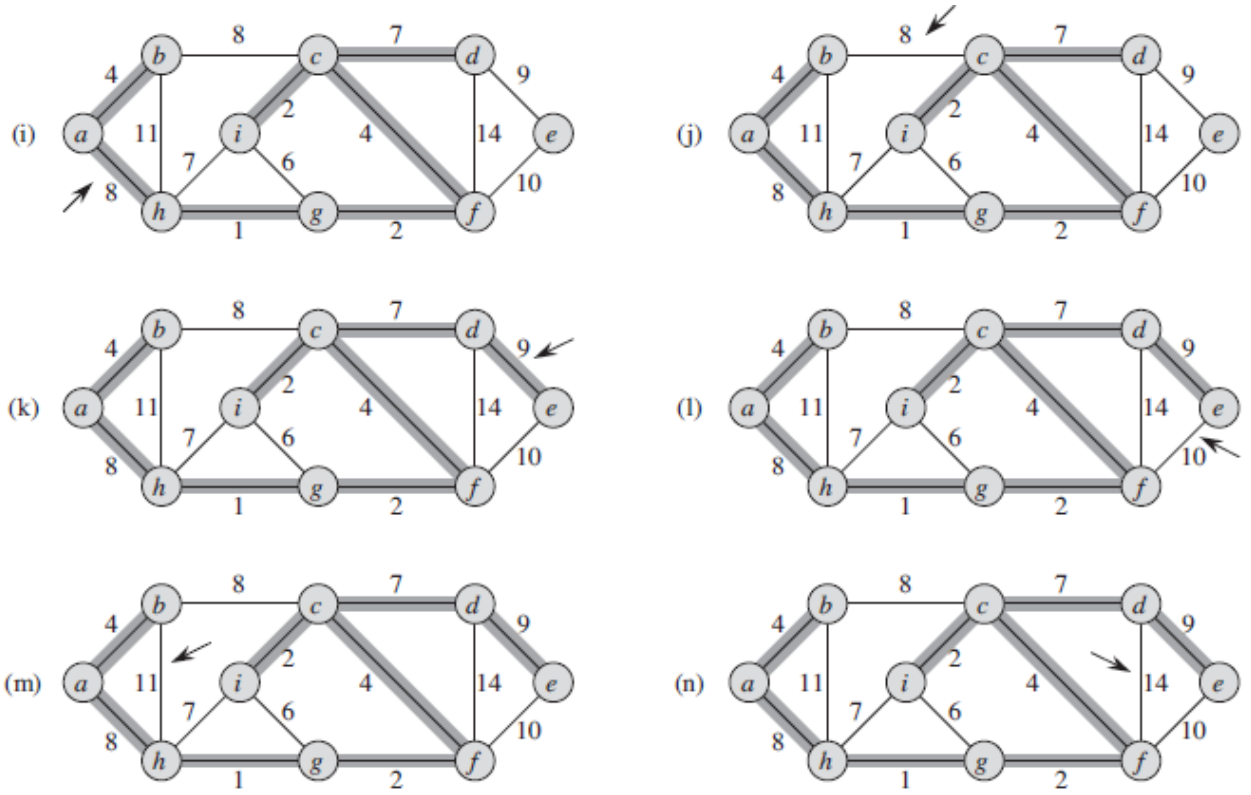
Ελέγχει για κάθε ακμή αν τα σημεία αρχής/τέλους ανήκουν σε διαφορετικά δένδρα

Αν ανήκουν σε διαφορετικά τότε η ακμή μπαίνει στο MST χωρίς να δημιουργεί κύκλο και οι δύο κόμβοι ενώνονται στο ίδιο δένδρο

```
MST-KRUSKAL(G, w)
1  A = ∅
2  for each vertex v ∈ G.V
3       MAKE-SET(v)
4  sort the edges of G.E into nondecreasing order by weight w
5  for each edge (u, v) ∈ G.E, taken in nondecreasing order by weight
6       if FIND-SET(u) ≠ FIND-SET(v)
7            A = A ∪ {(u, v)}
8            UNION(u, v)
9  return A
```

Kruskal's Algorithm (3/4)





Kruskal's Algorithm (4/4)

Ο χρόνος εκτέλεσης εξαρτάται από τη μέθοδο που υιοθετούμε για τον υπολογισμό των disjoint sets

Με τον πιο γρήγορο διαθέσιμο αλγόριθμο, ο Kruskal αλγόριθμος απαιτεί O(E logE)

Αν παρατηρήσουμε πως $|E|<|V|^2$, τότε $\log|E|= O(\log V)$

Άρα η πολυπλοκότητα είναι O(E logV)

Demo

https://www.cs.usfca.edu/~galles/visualization/Kruskal.html

Prim’s Algorithm (1/4)

Ενεργεί περίπου όπως ένας αλγόριθμος που βρίσκει το ελάχιστο μονοπάτι

Η βασική ιδιότητα του αλγορίθμου είναι ότι πάντα οι ακμές που μπαίνουν στο σύνολο Α σχηματίζουν ένα δένδρο

Σε κάθε βήμα, ο αλγόριθμος προσθέτει στο Α μια light ακμή που συνδέει το ήδη υπάρχον δένδρο του Α με ένα απομονωμένο κόμβο

Χρειαζόμαστε ένα γρήγορο τρόπο για να βρούμε το ποια ακμή θα μπει στο Α

Όλοι οι κόμβοι που δεν βρίσκονται στο MST αποθηκεύονται σε ουρά ελάχιστης προτεραιότητας

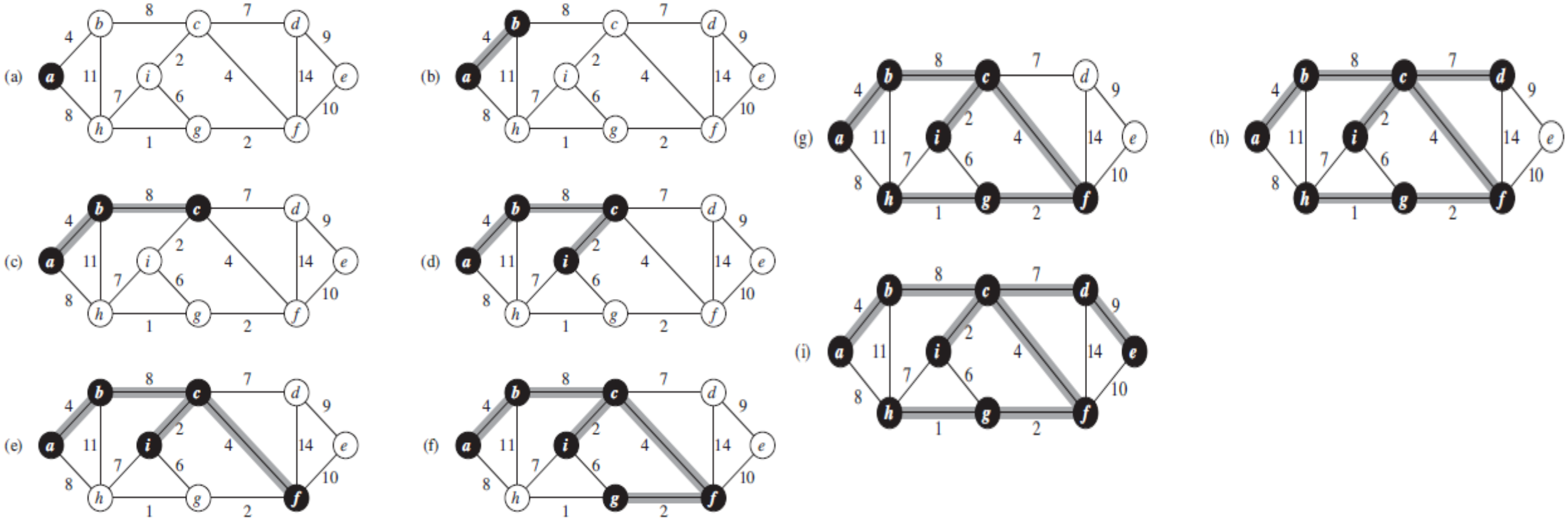
Prim's Algorithm (2/4)

Το v.key είναι το ελάχιστο βάρος οποιασδήποτε ακμής που συνδέει ένα κόμβο στο δένδρο

Το v.π υποδηλώνει τον πατρικό κόμβο του v στο δένδρο

```
MST-PRIM(G, w, r)
 1  for each u ∈ G.V
 2      u.key = ∞
 3      u.π = NIL
 4  r.key = 0
 5  Q = G.V
 6  while Q ≠ ∅
 7      u = EXTRACT-MIN(Q)
 8      for each v ∈ G.Adj[u]
 9          if v ∈ Q and w(u, v) < v.key
10              v.π = u
11              v.key = w(u, v)
```

Prim's Algorithm (3/4)



Prim’s Algorithm (4/4)

Ο χρόνος εκτέλεσης εξαρτάται από το πώς υλοποιούμε την ουρά

Μπορούμε να υιοθετήσουμε μια μέθοδο με χρόνο εκτέλεσης O(V)

Η εξαγωγή απαιτεί O(V logV)

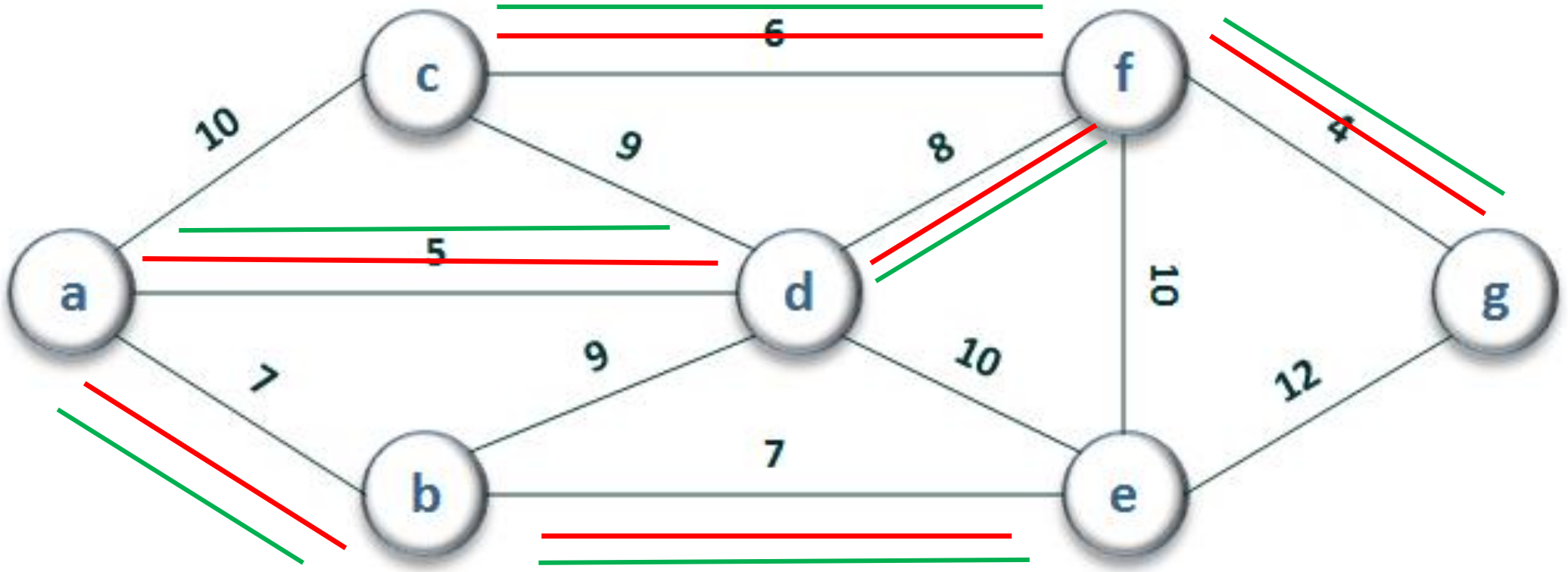
Συνολική πολυπλοκότητα O(E logV)

Demo

https://www.cs.usfca.edu/~galles/visualization/Prim.html

Άσκηση

Να κατασκευαστεί το MST επόμενο γράφο με χρήση των αλγορίθμων Kruskal (κόκκινο) & Prim (πράσινο)



Shortest Paths (1/7)

Δοσμένου ενός γράφου με βάρη αναζητούμε το **συντομότερο μονοπάτι (shortest path)** ανάμεσα σε δύο κόμβους

Γενικά ισχύει

$$w(p) = \sum_{i=1}^{k} w(v_{i-1}, v_i)$$

και

$$\delta(u, v) = \begin{cases} \min\{w(p) : u \overset{p}{\rightsquigarrow} v\} & \text{if there is a path from } u \text{ to } v \\ \infty & \text{otherwise} \end{cases}$$

Το συντομότερο μονοπάτι είναι οποιοδήποτε μονοπάτι για το οποίο ισχύει $w(p) = \delta(u, v)$

Shortest Paths (2/7)

Τα βάρη μπορεί να συμβολίζουν κόστος, απόσταση ή οποιαδήποτε άλλη μετρική ανάλογα με το application domain

Εστιάζουμε στα **single source shortest paths**

Υπάρχουν 3 εκδόσεις του προβλήματος

- **Single destination shortest path**
- **Single pair shortest path**
- **All pairs shortest path**

Shortest Paths (3/7)

Οι αλγόριθμοι εύρεσης συντομότερων μονοπατιών βασίζονται στην ιδιότητα ενός συντομότερου μονοπατιού ότι περιλαμβάνει άλλα συντομότερα μονοπάτια

Ορισμένες φορές πρέπει να υπολογίσουμε και τους κόμβους που συμμετέχουν στο συντομότερο μονοπάτι

Για κάθε κόμβο που ανήκει στο συντομότερο μονοπάτι υιοθετούμε την ιδιότητα v.π που αποθηκεύει τον προηγούμενο κόμβο

Οι αλγόριθμοι υιοθετούν την τεχνική του **relaxation**

Για κάθε κόμβο v κρατάμε την ιδιότητα v.d που αποτυπώνει ένα **πάνω όριο βάρους του συντομότερου μονοπατιού από τον κόμβο s στον v**

Shortest Paths (4/7)

Η ιδιότητα v.d ονομάζεται **shortest path estimate**

Η αρχικοποίηση των estimates γίνεται ως εξής:

```
INITIALIZE-SINGLE-SOURCE(G, s)
1  for each vertex v ∈ G.V
2      v.d = ∞
3      v.π = NIL
4  s.d = 0
```

Η τεχνική relaxation περιλαμβάνει τον έλεγχο με βάση μια ακμή (u,v) αν μπορούμε να επεκτείνουμε το συντομότερο μονοπάτι προς τον κόμβο v μέσω του u

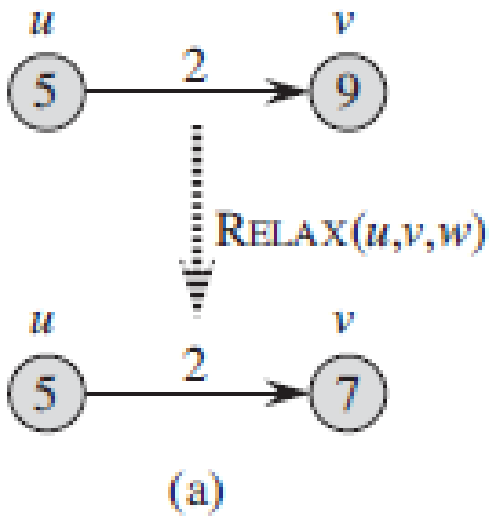
Εφόσον αυτό είναι εφικτό, τότε ενημερώνουμε τις ιδιότητες v.d και v.π

Shortest Paths (5/7)

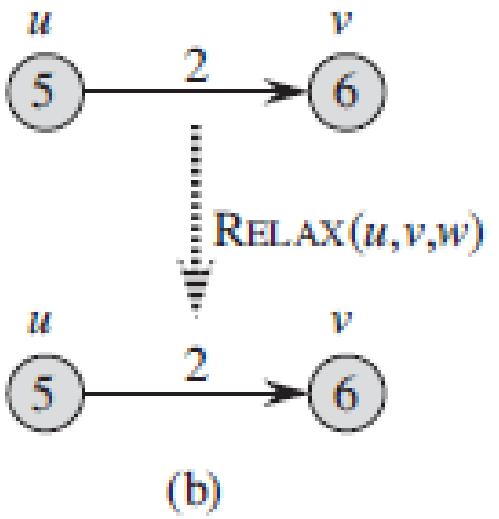
Κάποιο βήμα του relaxation μπορεί να μειώσει το estimate ενός κόμβου

Αλγόριθμος

```
RELAX(u, v, w)
1  if v.d > u.d + w(u, v)
2        v.d = u.d + w(u, v)
3        v.π = u
```



(a)



(b)

Shortest Paths (6/7)

Ιδιότητες των συντομότερων μονοπατιών

Triangle inequality

For any edge $(u, v) \in E$, we have $\delta(s, v) \leq \delta(s, u) + w(u, v)$.

Upper-bound property

We always have $v.d \geq \delta(s, v)$ for all vertices $v \in V$, and once $v.d$ achieves the value $\delta(s, v)$, it never changes.

No-path property

If there is no path from s to v, then we always have $v.d = \delta(s, v) = \infty$.

Shortest Paths (7/7)

Ιδιότητες των συντομότερων μονοπατιών (συνέχεια)

Convergence property

If $s \rightsquigarrow u \rightarrow v$ is a shortest path in G for some $u, v \in V$, and if $u.d = \delta(s, u)$ at any time prior to relaxing edge (u, v), then $v.d = \delta(s, v)$ at all times afterward.

Path-relaxation property

If $p = \langle v_0, v_1, \ldots, v_k \rangle$ is a shortest path from $s = v_0$ to v_k, and we relax the edges of p in the order $(v_0, v_1), (v_1, v_2), \ldots, (v_{k-1}, v_k)$, then $v_k.d = \delta(s, v_k)$. This property holds regardless of any other relaxation steps that occur, even if they are intermixed with relaxations of the edges of p.

Predecessor-subgraph property

Once $v.d = \delta(s, v)$ for all $v \in V$, the predecessor subgraph is a shortest-paths tree rooted at s.

Bellman-Ford Algorithm (1/10)

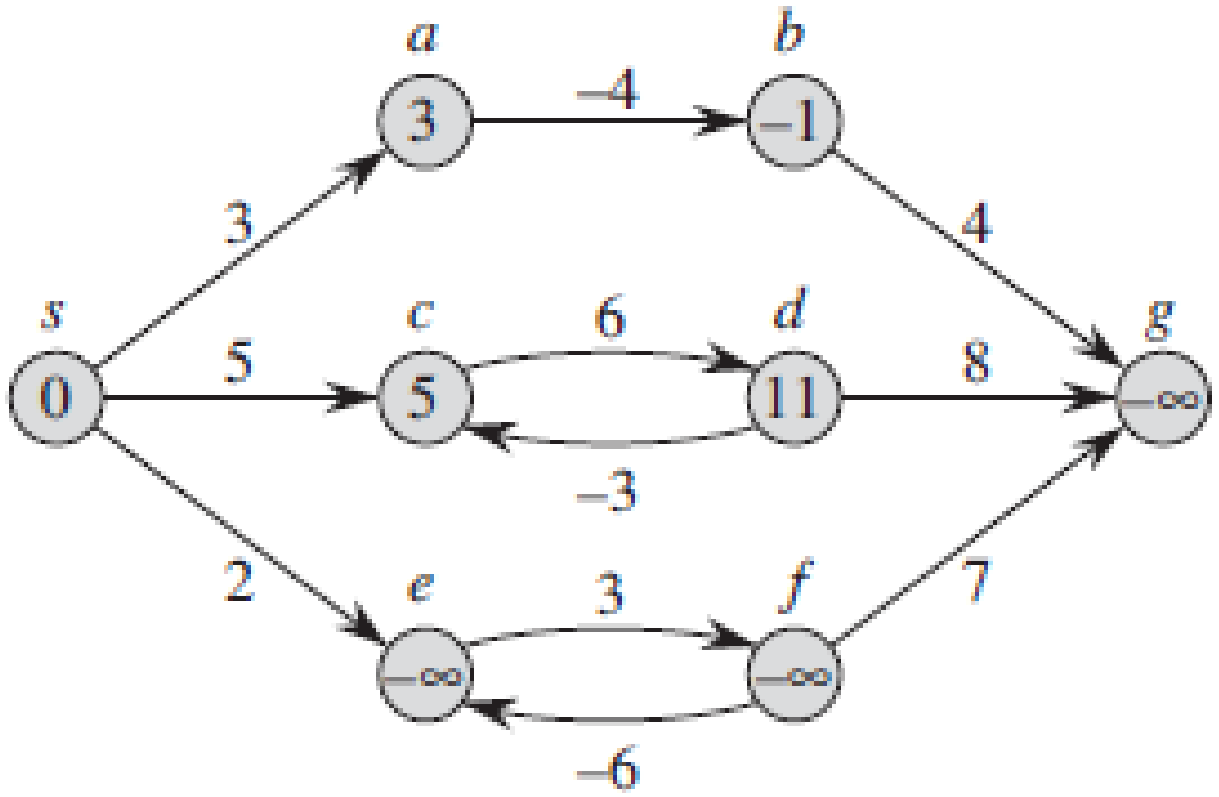
Ο αλγόριθμος επιλύει το πρόβλημα της εύρεσης του συντομότερου μονοπατιού στη γενική περίπτωση όπου τα βάρη μπορεί να είναι και αρνητικές τιμές

Δοσμένου ενός γράφου G=(E,V) με αρχικό κόμβο τον s και μια συνάρτηση βάρους w: E → R, ο αλγόριθμος επιστρέφει μια λογική τιμή που δείχνει αν υπάρχει κύκλος αρνητικών τιμών που μπορεί να προσπελαστεί από τον s

Αν υπάρχει τέτοιος κύκλος, το πρόβλημα δεν έχει λύση

Bellman-Ford Algorithm (2/10)

Παράδειγμα κύκλου με αρνητικές τιμές



Bellman-Ford Algorithm (3/10)

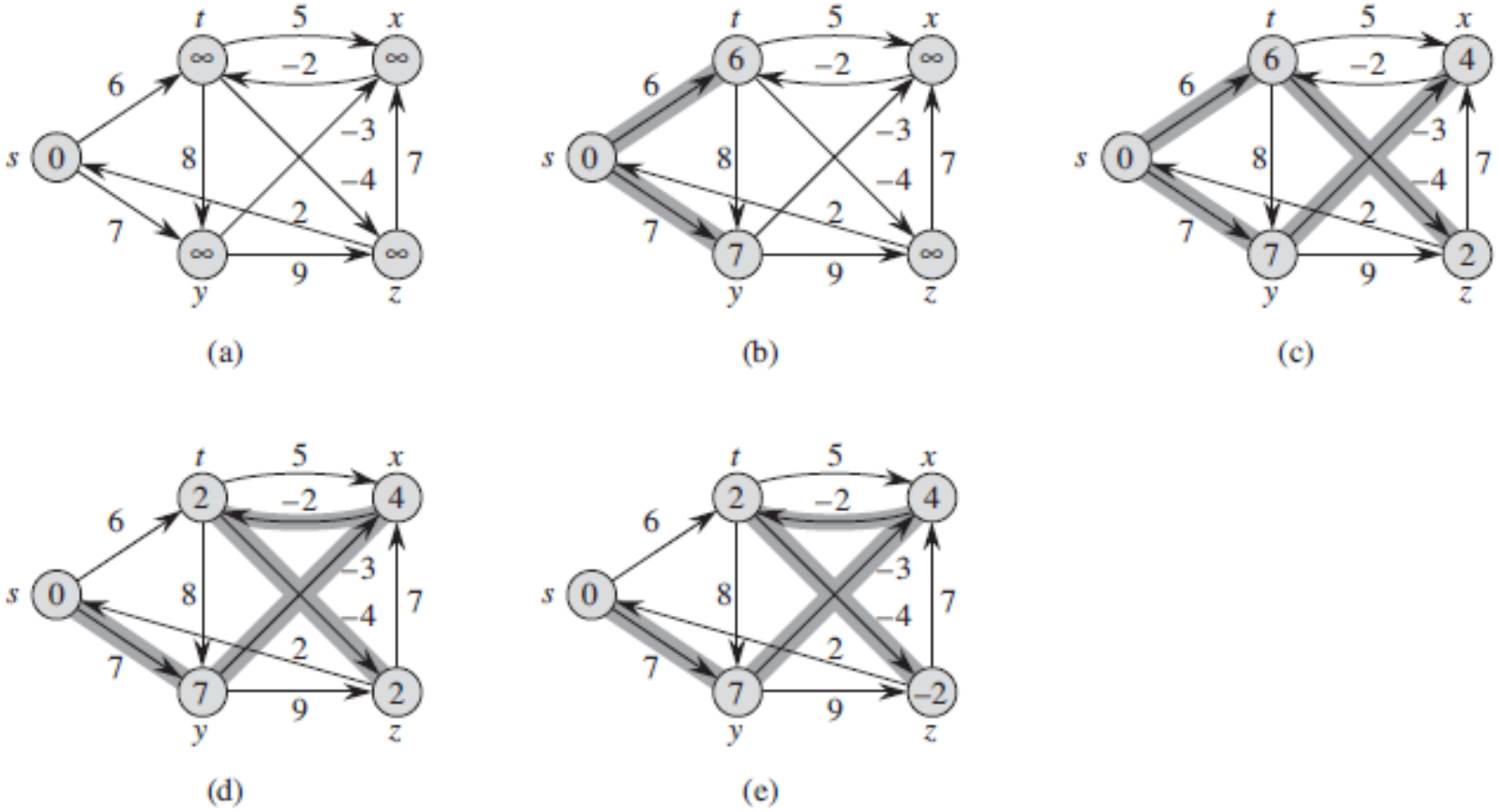
Αν υπάρχει τέτοιος κύκλος, δεν υπάρχει μονοπάτι από τον s που να είναι το συντομότερο μονοπάτι

```
BELLMAN-FORD(G, w, s)
1  INITIALIZE-SINGLE-SOURCE(G, s)
2  for i = 1 to |G.V| − 1
3       for each edge (u, v) ∈ G.E
4            RELAX(u, v, w)
5  for each edge (u, v) ∈ G.E
6       if v.d > u.d + w(u, v)
7            return FALSE
8  return TRUE
```

```
INITIALIZE-SINGLE-SOURCE(G, s)
1  for each vertex v ∈ G.V
2       v.d = ∞
3       v.π = NIL
4  s.d = 0
```

```
RELAX(u, v, w)
1  if v.d > u.d + w(u, v)
2       v.d = u.d + w(u, v)
3       v.π = u
```

Bellman-Ford Algorithm (4/10)



Bellman-Ford Algorithm (5/10)

Demo

https://www.youtube.com/watch?v=obWXjtg0L64

https://www.youtube.com/watch?v=hq3TZInZ5J4

Bellman-Ford Algorithm (6/10)

Lemma

Έπειτα από |V|-1 επαναλήψεις ο αλγόριθμος Bellman-Ford θα έχει τοποθετήσει v.d=δ(s,v) για ένα κόμβο – πηγή s για όλους τους κόμβους v που μπορεί να προσπελαστούν από τον s.

Απόδειξη

Έστω κάθε κόμβος v μπορεί να προσπελαστεί από τον s και P=(s, v_1, v_2, …, v) το οποιοδήποτε συντομότερο μονοπάτι. Το P έχει το πολύ |V|-1 ακμές αφού είναι ένα απλό μονοπάτι. Συνεπώς, k <= |V|-1. Σε κάθε επανάληψη ο αλγόριθμος αλλάζει το βάρος (relaxes) όλων των ακμών |E|. Μέσα σε αυτές τις ακμές, είναι και η (v_{i-1}, v_i). Από την ιδιότητα του path relaxation έχουμε $v.d = v_k.d = \delta(s, v_k) = \delta(s, v)$

Bellman-Ford Algorithm (7/10)

Θεώρημα

Let BELLMAN-FORD be run on a weighted, directed graph $G = (V, E)$ with source s and weight function $w : E \to \mathbb{R}$. If G contains no negative-weight cycles that are reachable from s, then the algorithm returns TRUE, we have $v.d = \delta(s, v)$ for all vertices $v \in V$, and the predecessor subgraph G_π is a shortest-paths tree rooted at s. If G does contain a negative-weight cycle reachable from s, then the algorithm returns FALSE.

Bellman-Ford Algorithm (8/10)

Απόδειξη

Αποδεικνύουμε πρώτα ότι v.d=δ(s,v) για κάθε v ϵ V. Αν ο κόμβος v μπορεί να προσπελαστεί από τον s, τότε το πρηγούμενο Λήμμα αποδεικνύει τον αρχικό ισχυρισμό. Η ιδιότητα του predecessor-subgraph μαζί με τον ισχυρισμό μας δείχνει πως ο υπο-γράφος G_π είναι ένα συντομότερο μονοπάτι. Με το πέρας εκτέλεσης του αλγορίθμου αν επιστρέψει TRUE έχουμε πως:

$$\begin{aligned} v.d &= \delta(s, v) \\ &\leq \delta(s, u) + w(u, v) \quad \text{(by the triangle inequality)} \\ &= u.d + w(u, v) \end{aligned}$$

Bellman-Ford Algorithm (9/10)

Απόδειξη (συνέχεια)

Ας υποθέσουμε τώρα πως ο γράφος περιέχει ένα κύκλο με αρνητικές τιμές που μπορεί να προσπελαστεί από τον s. Έστω ότι ο κύκλος είναι ο $c = \langle v_0, v_1, \ldots, v_k \rangle$ με $v_0 = v_k$

Τότε $\sum_{i=1}^{k} w(v_{i-1}, v_i) < 0$

Ας υποθέσουμε πως ο αλγόριθμος επιστρέφει TRUE

Τότε $v_i.d \leq v_{i-1}.d + w(v_{i-1}, v_i) \quad i = 1, 2, \ldots, k$

Bellman-Ford Algorithm (10/10)

Απόδειξη (συνέχεια)

Έχουμε $$\sum_{i=1}^{k} v_i.d \leq \sum_{i=1}^{k} (v_{i-1}.d + w(v_{i-1}, v_i))$$

$$= \sum_{i=1}^{k} v_{i-1}.d + \sum_{i=1}^{k} w(v_{i-1}, v_i)$$

Αφού $v_0 = v_k$, κάθε κόμβος στο c εμφανίζεται μια φορά στα αθροίσματα, οπότε

$$\sum_{i=1}^{k} v_i.d = \sum_{i=1}^{k} v_{i-1}.d \quad \text{και} \quad 0 \leq \sum_{i=1}^{k} w(v_{i-1}, v_i)$$ (αντίθετο με την αρχική εξίσωση)

Dijkstra’s Algorithm (1/6)

Ο αλγόριθμος Dijkstra επιλύει το πρόβλημα της εύρεσης του single source συντομότερου μονοπατιού όταν τα βάρη στις ακμές δεν είναι αρνητικά

Υποθέτουμε πως (u,v)>=0

Με μια καλή υλοποίηση, ο χρόνος εκτέλεσης του αλγορίθμου είναι καλύτερος από του Bellman-Ford

Ο αλγόριθμος διατηρεί ένα σύνολο κόμβων των οποίων το συντομότερο μονοπάτι από τον κόμβο s έχει ήδη καθοριστεί

Dijkstra’s Algorithm (2/6)

Επαναληπτικά ο αλγόριθμος:

- Επιλέγει τον κόμβο u ϵ V – S με την εκτίμηση του μικρότερου μονοπατιού
- Προσθέτει το u στο S
- Εφαρμόζει την τεχνική relaxation για όλες τις ακμές που φεύγουν από τον u

Υλοποίηση με ουρά μικρότερης προτεραιότητας

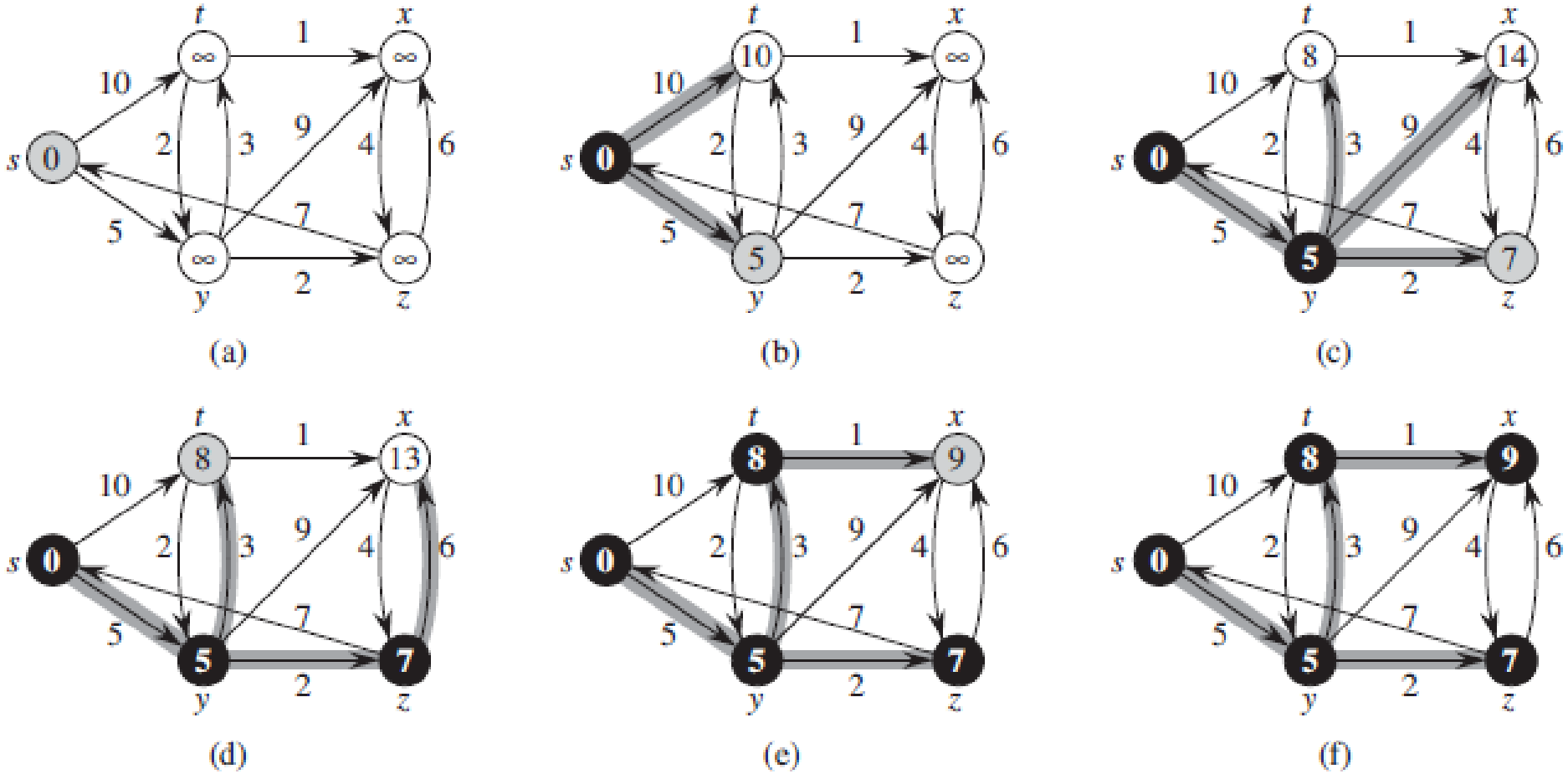
Dijkstra's Algorithm (3/6)

```
DIJKSTRA(G, w, s)
1  INITIALIZE-SINGLE-SOURCE(G, s)
2  S = ∅
3  Q = G.V
4  while Q ≠ ∅
5      u = EXTRACT-MIN(Q)
6      S = S ∪ {u}
7      for each vertex v ∈ G.Adj[u]
8          RELAX(u, v, w)
```

```
INITIALIZE-SINGLE-SOURCE(G, s)
1  for each vertex v ∈ G.V
2      v.d = ∞
3      v.π = NIL
4  s.d = 0
```

```
RELAX(u, v, w)
1  if v.d > u.d + w(u, v)
2      v.d = u.d + w(u, v)
3      v.π = u
```

Dijkstra's Algorithm (4/6)



Dijkstra’s Algorithm (5/6)

Demo

http://optlab-server.sce.carleton.ca/POAnimations2007/DijkstrasAlgo.html

Dijkstra’s Algorithm (6/6)

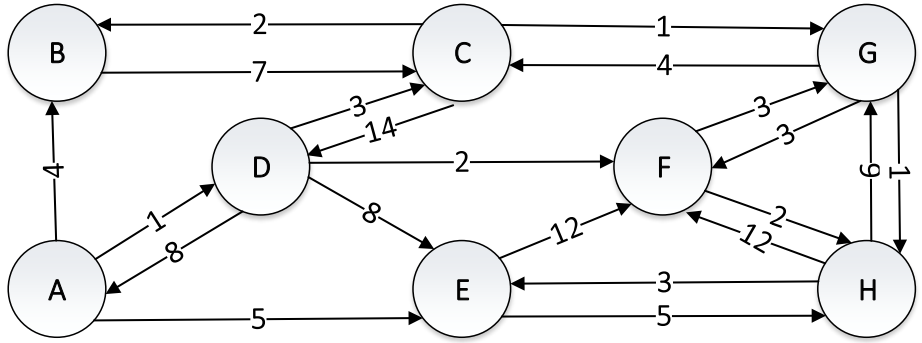
Θεώρημα

Ο αλγόριθμος Dijkstra αν εφαρμοστεί πάνω σε ένα γράφο κατευθυνόμενο με μη αρνητικά βάρη θα τερματίσει με $u.d = \delta(s,u)$ για κάθε $u \in V$.

Απόδειξη

Θα τη βρείτε στο βιβλίο

Άσκηση



S	A	B	C	D	E	F	G	H
A	0	4 (A)		1 (A)	5 (A)			
A D	0	4 (A)	4 (D)	1	5 (A)	3 (D)		
A D F	0	4 (A)	4 (D)	1	5 (A)	3	6 (F)	5 (F)
A D F B	0	4	4 (D)	1	5 (A)	3	6 (F)	5 (F)
A D F B C	0	4	4	1	5 (A)	3	5 (C)	5 (F)
A D F B C E	0	4	4	1	5	3	5 (C)	5 (F)
A D F B C E G	0	4	4	1	5	3	5	5 (F)
A D F B C E G H	0	4	4	1	5	3	5	5