All Pairs Shortest Paths (1/10)

Το πρόβλημα είναι να βρούμε τα συντομότερα μονοπάτια ανάμεσα σε όλους τους κόμβους

Για κάθε ζεύγος u, v θέλουμε να εξάγουμε το συντομότερο μονοπάτι

Τυπικά, θέλουμε το αποτέλεσμα σε μια μορφή πίνακα

Μπορούμε να λύσουμε το πρόβλημα αν τρέξουμε |V| φορές ένα αλγόριθμο εύρεσης του συντομότερου μονοπατιού

Αν ο γράφος έχει αρνητικά βάρη μπορούμε να υιοθετήσουμε τον Bellman-Ford αν όχι τον Dijkstra

Πολυπλοκότητα: $O(V^2 E)$ ή σε πυκνούς γράφους $O(V^4)$

All Pairs Shortest Paths (2/10)

Θα υιοθετήσουμε λύση με βάση πίνακες γειτνίασης

Ένας πίνακας γειτνίασης περιλαμβάνει στοιχεία ως εξής:

$$w_{ij} = \begin{cases} 0 & \text{if } i = j \ , \\ \text{the weight of directed edge } (i, j) & \text{if } i \neq j \text{ and } (i, j) \in E \ , \\ \infty & \text{if } i \neq j \text{ and } (i, j) \notin E \ . \end{cases}$$

Ο πίνακας με τα στοιχεία των συντομότερων μονοπατιών περιέχει στοιχεία d_{ij} που απεικονίζουν το βάρος του συντομότερου μονοπατιού που συνδέει τους αντίστοιχους κόμβους

All Pairs Shortest Paths (3/10)

Πρέπει να υπολογίσουμε όχι μόνο τα βάρη των συντομότερων μονοπατιών αλλά και τον πίνακα των προηγούμενων κόμβων

Τα στοιχεία π_{ij} δείχνουν αν δεν υπάρχει μονοπάτι ανάμεσα σε δύο κόμβους (NIL) ή τον προηγούμενο κόμβο σε κάποιο συντομότερο μονοπάτι

```
PRINT-ALL-PAIRS-SHORTEST-PATH(Π, i, j)
1  if i == j
2       print i
3  elseif π_ij == NIL
4       print "no path from" i "to" j "exists"
5  else PRINT-ALL-PAIRS-SHORTEST-PATH(Π, i, π_ij)
6       print j
```

All Pairs Shortest Paths (4/10)

Έστω $l_{ij}^{(m)}$ το ελάχιστο βάρος οποιουδήποτε μονοπατιού από τον κόμβο i στον κόμβο j που περιέχει το πολύ m ακμές

Παίρνουμε σαν είσοδο τον πίνακα W=(w_{ij}) και υπολογίζουμε μια σειρά από πίνακες $L^{(1)}, L^{(2)}, \ldots, L^{(n-1)}$ όπου για m=1,2,…,n-1 έχουμε $L^{(m)} = (l_{ij}^{(m)})$

Ο τελικός πίνακας $L^{(n-1)}$ περιέχει τα τελικά βάρη των συντομότερων μονοπατιών

All Pairs Shortest Paths (5/10)

EXTEND-SHORTEST-PATHS(L, W)

```
1  n = L.rows
2  let L' = (l'_ij) be a new n × n matrix
3  for i = 1 to n
4      for j = 1 to n
5          l'_ij = ∞
6          for k = 1 to n
7              l'_ij = min(l'_ij, l_ik + w_kj)
8  return L'
```

$$l_{ij}^{(m)} = \min\left(l_{ij}^{(m-1)}, \min_{1 \le k \le n}\left\{l_{ik}^{(m-1)} + w_{kj}\right\}\right)$$

$$= \min_{1 \le k \le n}\left\{l_{ik}^{(m-1)} + w_{kj}\right\}$$

All Pairs Shortest Paths (6/10)

Με πολ/μο πινάκων

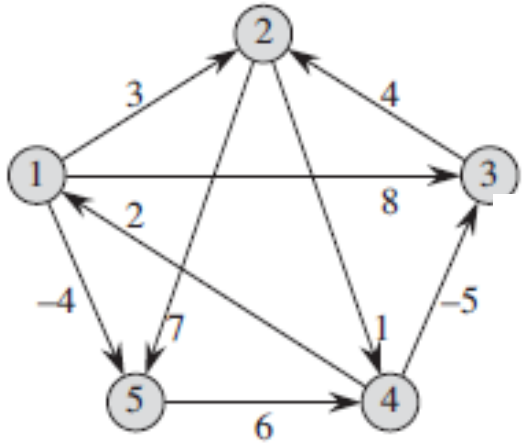
$$
\begin{aligned}
L^{(1)} &= L^{(0)} \cdot W &&= W \\
L^{(2)} &= L^{(1)} \cdot W &&= W^2 \\
L^{(3)} &= L^{(2)} \cdot W &&= W^3 \\
&\vdots \\
L^{(n-1)} &= L^{(n-2)} \cdot W &&= W^{n-1}
\end{aligned}
$$

All Pairs Shortest Paths (7/10)

```
SLOW-ALL-PAIRS-SHORTEST-PATHS(W)
1  n = W.rows
2  L^(1) = W
3  for m = 2 to n − 1
4       let L^(m) be a new n × n matrix
5       L^(m) = EXTEND-SHORTEST-PATHS(L^(m−1), W)
6  return L^(n−1)
```

All Pairs Shortest Paths (8/10)

Παράδειγμα



$$L^{(1)} = \begin{pmatrix} 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \infty & \infty \\ 2 & \infty & -5 & 0 & \infty \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \quad L^{(2)} = \begin{pmatrix} 0 & 3 & 8 & 2 & -4 \\ 3 & 0 & -4 & 1 & 7 \\ \infty & 4 & 0 & 5 & 11 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & \infty & 1 & 6 & 0 \end{pmatrix}$$

$$L^{(3)} = \begin{pmatrix} 0 & 3 & -3 & 2 & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & 11 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{pmatrix} \quad L^{(4)} = \begin{pmatrix} 0 & 1 & -3 & 2 & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & 3 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{pmatrix}$$

EXTEND-SHORTEST-PATHS(L, W)

1 $n = L.rows$
2 let $L' = (l'_{ij})$ be a new $n \times n$ matrix
3 **for** $i = 1$ **to** n
4 **for** $j = 1$ **to** n
5 $l'_{ij} = \infty$
6 **for** $k = 1$ **to** n
7 $l'_{ij} = \min(l'_{ij}, l_{ik} + w_{kj})$
8 **return** L'

All Pairs Shortest Paths (9/10)

Βελτίωση του χρόνου εκτέλεσης

- Πράξεις

$$
\begin{aligned}
L^{(1)} &= W\,, \\
L^{(2)} &= W^2 &&= W \cdot W \\
L^{(4)} &= W^4 &&= W^2 \cdot W^2 \\
L^{(8)} &= W^8 &&= W^4 \cdot W^4 \\
&\vdots \\
L^{(2^{\lceil \lg(n-1) \rceil})} &= W^{2^{\lceil \lg(n-1) \rceil}} &&= W^{2^{\lceil \lg(n-1) \rceil - 1}} \cdot W^{2^{\lceil \lg(n-1) \rceil - 1}}
\end{aligned}
$$

- Χρόνος υπολογισμού ceiling(log(n-1))

All Pairs Shortest Paths (10/10)

```
FASTER-ALL-PAIRS-SHORTEST-PATHS(W)
1  n = W.rows
2  L^(1) = W
3  m = 1
4  while m < n − 1
5      let L^(2m) be a new n × n matrix
6      L^(2m) = EXTEND-SHORTEST-PATHS(L^(m), L^(m))
7      m = 2m
8  return L^(m)
```

Floyd-Warshall Algorithm (1/12)

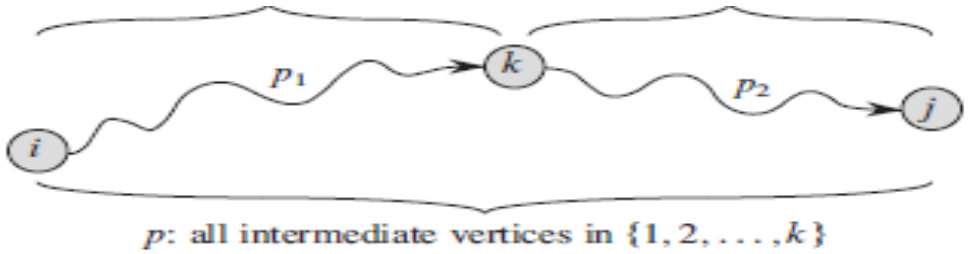
Ο αλγόριθμος βασίζεται στην ακόλουθη παρατήρηση:

- Ας θεωρήσουμε ένα υποσύνολο κόμβων {1,2,…, k}
- Για κάθε ζεύγος κόμβων i, j θεωρούμε όλα τα μονοπάτια που τους συνδέουν και έχουν ενδιάμεσες ακμές από το {1,2,…, k}
- Έστω p ένα μονοπάτι ελάχιστου βάρους ανάμεσά τους
- Ο αλγόριθμος αναζητά τη σχέση ανάμεσα στο p και στα συντομότερα μονοπάτια από το i στο j με ενδιάμεσους κόμβους στο σύνολο {1,2,…, k-1}
- Η σχέση εξαρτάται από το αν ο k κόμβος είναι ένας ενδιάμεσος κόμβος του p

Floyd-Warshall Algorithm (2/12)

Αν ο k **δεν είναι ένας ενδιάμεσος κόμβος** του p, τότε όλοι οι ενδιάμεσοι κόμβοι του p είναι στο σύνολο {1,2,…, k-1}. Έτσι ένα συντομότερο μονοπάτι από το i στο j με ενδιάμεσους κόμβους στο {1,2,…, k-1} είναι επίσης ένα συντομότερο μονοπάτι με ενδιάμεσους κόμβους στο {1,2,…, k}

Αν ο k **είναι ένας ενδιάμεσος κόμβος** του p, τότε χωρίζουμε το p σε δύο τμήματα: i-(p_1)-k-(p_2)-j. Το p_1 είναι ένα συντομότερο μονοπάτι από το i στο k με ενδιάμεσους κόμβους στο {1,2,…, k}. Όλοι οι ενδιάμεσοι κόμβοι του p_1 είναι στο {1,2,…, k-1}. Έτσι, το p_1 είναι ένα συντομότερο μονοπάτι. Ομοίως για το p_2.



Floyd-Warshall Algorithm (3/12)

Έστω $d_{ij}^{(k)}$ το βάρος του συντομότερου μονοπατιού από τον κόμβο i στον j για το οποίο όλες οι ενδιάμεσοι κόμβοι είναι στο {1,2,…,k}.

Όταν k=0, ένα μονοπάτι από το i στο j με μεγαλύτερο βάρος από 0 δεν έχει ενδιάμεσους κόμβους

Ένα τέτοιο μονοπάτι έχει το πολύ μια ακμή και έτσι $d_{ij}^{(0)} = w_{ij}$

Η αναδρομική σχέση έχει ως εξής:

$$d_{ij}^{(k)} = \begin{cases} w_{ij} & \text{if } k = 0 \\ \min\left(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}\right) & \text{if } k \geq 1 \end{cases}$$

Floyd-Warshall Algorithm (4/12)

Βασιζόμενοι στην αναδρομική σχέση, μπορούμε χρησιμοποιώντας μια bottom-up προσέγγιση να υπολογίσουμε τις τιμές του $d_{ij}^{(k)}$

Ο αλγόριθμος θα μας επιστρέψει ένα πίνακα με τα βάρη των συντομότερων μονοπατιών

```
FLOYD-WARSHALL(W)
1  n = W.rows
2  D^(0) = W
3  for k = 1 to n
4      let D^(k) = (d_ij^(k)) be a new n × n matrix
5      for i = 1 to n
6          for j = 1 to n
7              d_ij^(k) = min(d_ij^(k-1), d_ik^(k-1) + d_kj^(k-1))
8  return D^(n)
```

Floyd-Warshall Algorithm (5/12)

Παράδειγμα

$$D^{(0)} = \begin{pmatrix} 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \infty & \infty \\ 2 & \infty & -5 & 0 & \infty \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \quad \Pi^{(0)} = \begin{pmatrix} \text{NIL} & 1 & 1 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 2 & 2 \\ \text{NIL} & 3 & \text{NIL} & \text{NIL} & \text{NIL} \\ 4 & \text{NIL} & 4 & \text{NIL} & \text{NIL} \\ \text{NIL} & \text{NIL} & \text{NIL} & 5 & \text{NIL} \end{pmatrix}$$

$$D^{(1)} = \begin{pmatrix} 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \infty & \infty \\ 2 & 5 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \quad \Pi^{(1)} = \begin{pmatrix} \text{NIL} & 1 & 1 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 2 & 2 \\ \text{NIL} & 3 & \text{NIL} & \text{NIL} & \text{NIL} \\ 4 & 1 & 4 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 5 & \text{NIL} \end{pmatrix}$$

$$D^{(2)} = \begin{pmatrix} 0 & 3 & 8 & 4 & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & 5 & 11 \\ 2 & 5 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \quad \Pi^{(2)} = \begin{pmatrix} \text{NIL} & 1 & 1 & 2 & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 2 & 2 \\ \text{NIL} & 3 & \text{NIL} & 2 & 2 \\ 4 & 1 & 4 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 5 & \text{NIL} \end{pmatrix}$$

```
FLOYD-WARSHALL(W)
1  n = W.rows
2  D^(0) = W
3  for k = 1 to n
4       let D^(k) = (d_ij^(k)) be a new n × n matrix
5       for i = 1 to n
6            for j = 1 to n
7                 d_ij^(k) = min(d_ij^(k-1), d_ik^(k-1) + d_kj^(k-1))
8  return D^(n)
```

Floyd-Warshall Algorithm (6/12)

Παράδειγμα

$$D^{(3)} = \begin{pmatrix} 0 & 3 & 8 & 4 & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & 5 & 11 \\ 2 & -1 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \quad \Pi^{(3)} = \begin{pmatrix} \text{NIL} & 1 & 1 & 2 & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 2 & 2 \\ \text{NIL} & 3 & \text{NIL} & 2 & 2 \\ 4 & 3 & 4 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 5 & \text{NIL} \end{pmatrix}$$

$$D^{(4)} = \begin{pmatrix} 0 & 3 & -1 & 4 & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & 3 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{pmatrix} \quad \Pi^{(4)} = \begin{pmatrix} \text{NIL} & 1 & 4 & 2 & 1 \\ 4 & \text{NIL} & 4 & 2 & 1 \\ 4 & 3 & \text{NIL} & 2 & 1 \\ 4 & 3 & 4 & \text{NIL} & 1 \\ 4 & 3 & 4 & 5 & \text{NIL} \end{pmatrix}$$

$$D^{(5)} = \begin{pmatrix} 0 & 1 & -3 & 2 & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & 3 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{pmatrix} \quad \Pi^{(5)} = \begin{pmatrix} \text{NIL} & 3 & 4 & 5 & 1 \\ 4 & \text{NIL} & 4 & 2 & 1 \\ 4 & 3 & \text{NIL} & 2 & 1 \\ 4 & 3 & 4 & \text{NIL} & 1 \\ 4 & 3 & 4 & 5 & \text{NIL} \end{pmatrix}$$

```
FLOYD-WARSHALL(W)
1  n = W.rows
2  D^(0) = W
3  for k = 1 to n
4      let D^(k) = (d_ij^(k)) be a new n × n matrix
5      for i = 1 to n
6          for j = 1 to n
7              d_ij^(k) = min(d_ij^(k-1), d_ik^(k-1) + d_kj^(k-1))
8  return D^(n)
```

Floyd-Warshall Algorithm (7/12)

Η δημιουργία των συντομότερων μονοπατιών περιλαμβάνει την κατασκευή ενός πίνακα Π που περιέχει τους προηγούμενους κόμβους στα συντομότερα μονοπάτια

Μπορούμε να υπολογίσουμε μια αλληλουχία πινάκων $\Pi^{(0)}$, $\Pi^{(1)}$, …, $\Pi^{(n)}$ καθώς κατασκευάζουμε τους πίνακες $D^{(i)}$

Ως $\pi_{ij}^{(k)}$ ορίσουμε τον προηγούμενο κόμβο j σε ένα συντομότερο μονοπάτι από το i με όλους τους ενδιάμεσους κόμβους στο {1,2,…,k}

Η εξίσωση που ισχύει έχει ως εξής:

$$\pi_{ij}^{(0)} = \begin{cases} \text{NIL} & \text{if } i = j \text{ or } w_{ij} = \infty \\ i & \text{if } i \neq j \text{ and } w_{ij} < \infty \end{cases}$$

Floyd-Warshall Algorithm (8/12)

Για k>=1, αν πάρουμε το μονοπάτι i-k-j με k διαφορετικό του j, τότε ο προηγούμενος κόμβος του j είναι ο ίδιος του j που επιλέξαμε στο συντομότερο μονοπάτι από το k με όλους τους ενδιάμεσους στο {1,2,…,k-1}

Διαφορετικά, παίρνουμε τον προηγούμενο κόμβο του j που επιλέξαμε στο συντομότερο μονοπάτι από το i με όλους τους ενδιάμεσους στο {1,2,…,k-1}

$$\pi_{ij}^{(k)} = \begin{cases} \pi_{ij}^{(k-1)} & \text{if } d_{ij}^{(k-1)} \leq d_{ik}^{(k-1)} + d_{kj}^{(k-1)} \\ \pi_{kj}^{(k-1)} & \text{if } d_{ij}^{(k-1)} > d_{ik}^{(k-1)} + d_{kj}^{(k-1)} \end{cases}$$

Floyd-Warshall Algorithm (9/12)

Ορισμένες φορές θέλουμε να ορίσουμε αν ένας γράφος περιλαμβάνει ένα μονοπάτι από το i στο j για κάθε ζευγάρι κόμβων

Η ιδιότητα αυτή ορίζεται ως η **μεταβατική κλειστότητα** (transitive closure) του γράφου

Για τον υπολογισμό της αντικαθιστούμε με τη λογική πράξη του OR για το min και την πράξη AND για το + στον αλγόριθμο Floyd-Warshall

Για κάθε i,j,k=1,2,…,n ορίζουμε το $t_{ij}^{(k)}$ που παίρνει την τιμή 1 αν υπάρχει μονοπάτι από τον κόμβο i στον κόμβο j με όλους τους ενδιάμεσους κόμβους στο {1,2,…,k} και 0 διαφορετικά

Floyd-Warshall Algorithm (10/12)

Η αναδρομική σχέση έχει ως εξής:

$$t_{ij}^{(0)} = \begin{cases} 0 & \text{if } i \neq j \text{ and } (i,j) \notin E \\ 1 & \text{if } i = j \text{ or } (i,j) \in E \end{cases}$$

$$k \geq 1,$$

$$t_{ij}^{(k)} = t_{ij}^{(k-1)} \vee \left(t_{ik}^{(k-1)} \wedge t_{kj}^{(k-1)}\right)$$

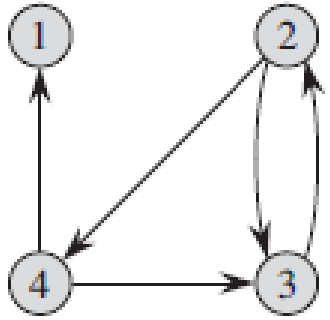
Floyd-Warshall Algorithm (11/12)

TRANSITIVE-CLOSURE(G)

1 $n = |G.V|$
2 let $T^{(0)} = (t_{ij}^{(0)})$ be a new $n \times n$ matrix
3 **for** $i = 1$ **to** n
4 **for** $j = 1$ **to** n
5 **if** $i == j$ or $(i, j) \in G.E$
6 $t_{ij}^{(0)} = 1$
7 **else** $t_{ij}^{(0)} = 0$
8 **for** $k = 1$ **to** n
9 let $T^{(k)} = (t_{ij}^{(k)})$ be a new $n \times n$ matrix
10 **for** $i = 1$ **to** n
11 **for** $j = 1$ **to** n
12 $t_{ij}^{(k)} = t_{ij}^{(k-1)} \vee (t_{ik}^{(k-1)} \wedge t_{kj}^{(k-1)})$
13 **return** $T^{(n)}$

Floyd-Warshall Algorithm (12/12)

```
TRANSITIVE-CLOSURE(G)
1   n = |G.V|
2   let T^(0) = (t_ij^(0)) be a new n × n matrix
3   for i = 1 to n
4       for j = 1 to n
5           if i == j or (i, j) ∈ G.E
6               t_ij^(0) = 1
7           else t_ij^(0) = 0
8   for k = 1 to n
9       let T^(k) = (t_ij^(k)) be a new n × n matrix
10      for i = 1 to n
11          for j = 1 to n
12              t_ij^(k) = t_ij^(k-1) ∨ (t_ik^(k-1) ∧ t_kj^(k-1))
13  return T^(n)
```



$$T^{(0)} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 \\ 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 1 \end{pmatrix} \quad T^{(1)} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 \\ 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 1 \end{pmatrix} \quad T^{(2)} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 \end{pmatrix}$$

$$T^{(3)} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \end{pmatrix} \quad T^{(4)} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \end{pmatrix}$$

Johnson's Algorithm (1/4)

Ο αλγόριθμος είτε επιστρέφει ένα πίνακα με τα βάρη των συντομότερων μονοπατιών ή επίστρέφει πως ο γράφος έχει ένα κύκλο αρνητικών τιμών

Υιοθετεί τους αλγορίθμους των Bellman-Ford & Dijkstra

Υιοθετεί την τεχνική του επαναυπολογισμού βαρών (reweighting)

Αν όλα τα βάρη είναι μη αρνητικά, μπορούμε να βρούμε τα συντομότερα μονοπάτια με τη χρήση του Dijkstra υιοθετώντας τον για κάθε κόμβο

Αν υιοθετήσουμε σωρούς Fibonacci η πολυπλοκότητα είναι $O(V^2 \log V + V E)$

Johnson's Algorithm (2/4)

Αν ο γράφος έχει αρνητικά βάρη αλλά όχι κύκλους αρνητικών τιμών, μπορούμε να υπολογίσουμε ένα νέο σύνολο βαρών που μας επιτρέπει να υιοθετήσουμε την προηγούμενη μέθοδο

Το νέο σύνολο πρέπει να ικανοποιεί τα ακόλουθα:

- Για κάθε ζεύγος κόμβων, ένα μονοπάτι είναι το συντομότερο για μια συνάρτηση βαρών w αν είναι συντομότερο για μια συνάρτηση g – πρέπει να ορίσουμε την $\hat{w}$
- Για κάθε ακμή του γράφου, το νέο βάρος να είναι μη αρνητικό

Johnson’s Algorithm (3/4)

Η συνάρτηση που χρησιμοποιούμε είναι η ακόλουθη:

$$\hat{w}(u, v) = w(u, v) + h(u) - h(v)$$

Όπου h είναι μια συνάρτηση που απεικονίζει ένα κόμβο σε πραγματικούς αριθμούς

Johnson's Algorithm (4/4)

```
JOHNSON(G, w)
 1  compute G', where G'.V = G.V ∪ {s},
        G'.E = G.E ∪ {(s, v) : v ∈ G.V}, and
        w(s, v) = 0 for all v ∈ G.V
 2  if BELLMAN-FORD(G', w, s) == FALSE
 3      print "the input graph contains a negative-weight cycle"
 4  else for each vertex v ∈ G'.V
 5          set h(v) to the value of δ(s, v)
                computed by the Bellman-Ford algorithm
 6      for each edge (u, v) ∈ G'.E
 7          ŵ(u, v) = w(u, v) + h(u) − h(v)
 8      let D = (d_uv) be a new n × n matrix
 9      for each vertex u ∈ G.V
10          run DIJKSTRA(G, ŵ, u) to compute δ̂(u, v) for all v ∈ G.V
11          for each vertex v ∈ G.V
12              d_uv = δ̂(u, v) + h(v) − h(u)
13      return D
```

Maximum Flow (1/7)

Μπορούμε να θεωρήσουμε ένα γράφο ως ένα δίκτυο ροής

Παράδειγμα: σε ένα κόμβο κατασκευάζεται ένα υλικό ενώ σε κάποιον άλλο καταναλώνεται

Η ροή σε κάθε σημείο του δικτύου είναι ο ρυθμός με τον οποίο το υλικό μετακινείται

Στο πρόβλημα της μέγιστης ροής (maximum flow) θέλουμε να βρούμε το μεγαλύτερο ρυθμό με το οποίο μπορούμε να μετακινήσουμε το υλικό μέσα στο δίκτυο

Maximum Flow (2/7)

Ένα δίκτυο ροής (flow network) G=(V,E) είναι ένας κατευθυνόμενος γράφος στο οποίο κάθε ακμή έχει μη αρνητική τιμή (χωρητικότητα - capacity)

Αν ο γράφος περιλαμβάνει μια ακμή προς μια κατεύθυνση τότε δεν υπάρχει ακμή προς την αντίθετη κατεύθυνση

Διακρίνουμε δύο ειδών κόμβους: τους κόμβους – πηγές (source) και τους κόμβους καταβόθρες (sink)

Maximum Flow (3/7)

Μια ροή (flow) είναι μια συνάρτηση f: V X V → R που ικανοποιεί τα ακόλουθα κριτήρια:

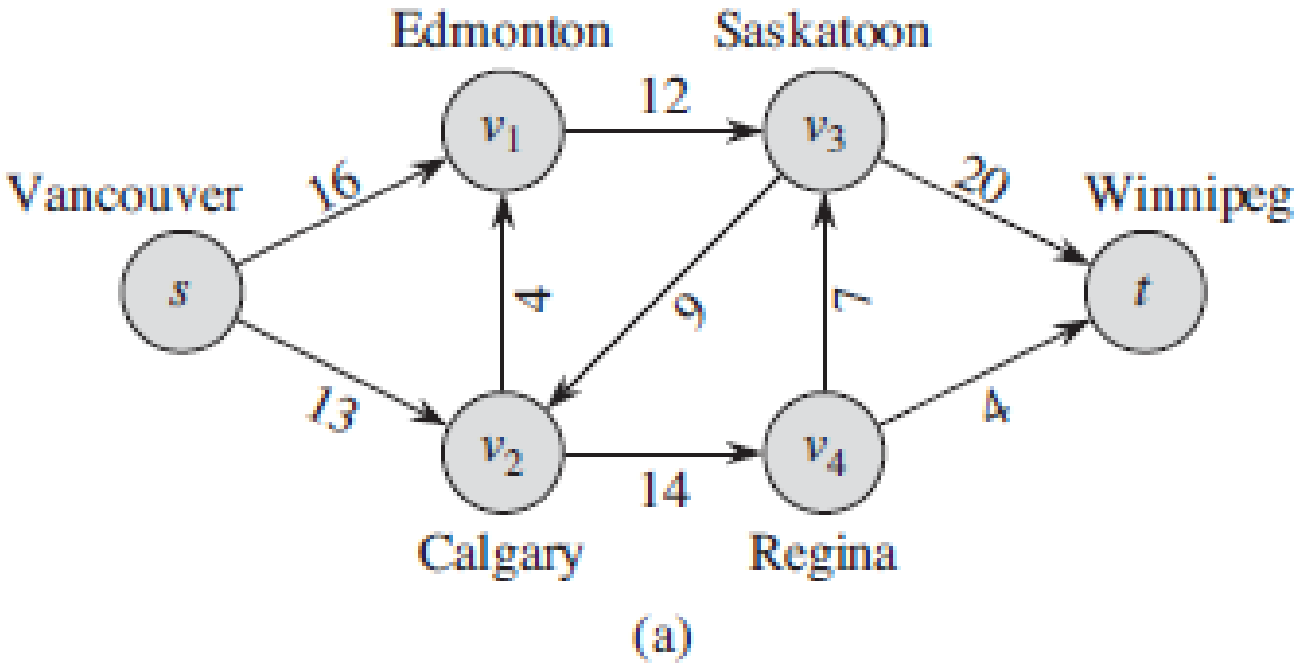
- Για κάθε u,v ϵ V, απαιτείται 0<= f(u,v) <= c(u,v), όπου c(u,v) είναι η χωρητικότητα της διαδρομής (**capacity constraint**)
- Για κάθε v ϵ V – {s, t} απαιτείται (**flow conservation**)

$$|f| = \sum_{v \in V} f(s, v) - \sum_{v \in V} f(v, s)$$

- Η ποσότητα |f| (**flow value**) απεικονίζει τη συνολική ροή από το s μείον τη συνολική ροή προς το s
- Όταν μια ακμή δεν ανήκει στο E τότε έχουμε f(u,v)=0

Maximum Flow (4/7)

Παράδειγμα



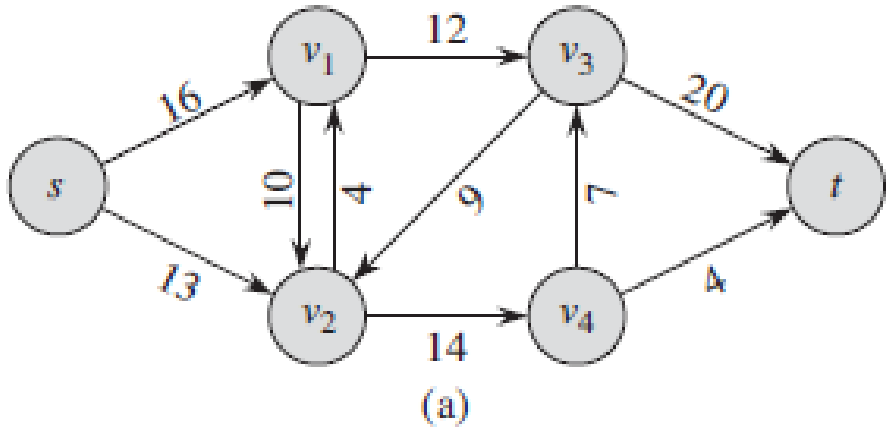
Maximum Flow (5/7)

Ας υποθέσουμε πως σε μια ακμή δημιουργούμε μια ‘αντίθετή’ της

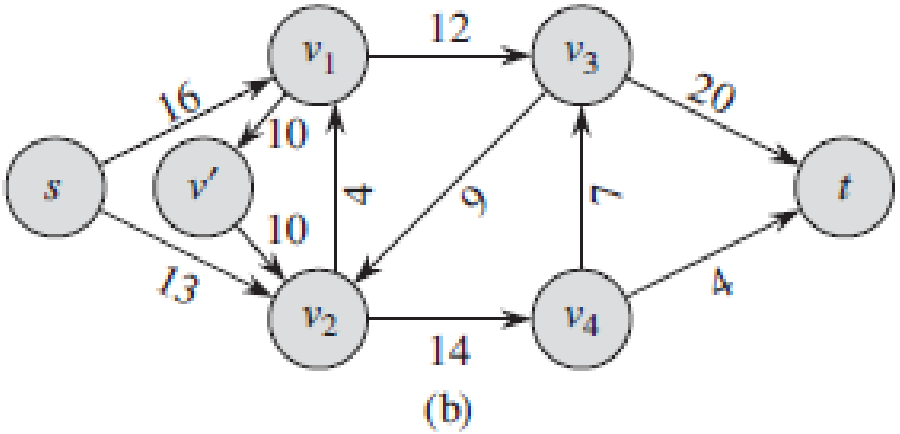
Προφανώς το δίκτυο παραβιάζει την αρχική υπόθεση της μη ύπαρξης αντίθετων ακμών

Ονομάζουμε τις δύο ακμές **αντιπαράλληλες (antiparallel)**

Για είμαστε σύμφωνοι με την αρχική υπόθεση δημιουργούμε ένα ενδιάμεσο κόμβο



(a)



(b)

Maximum Flow (6/7)

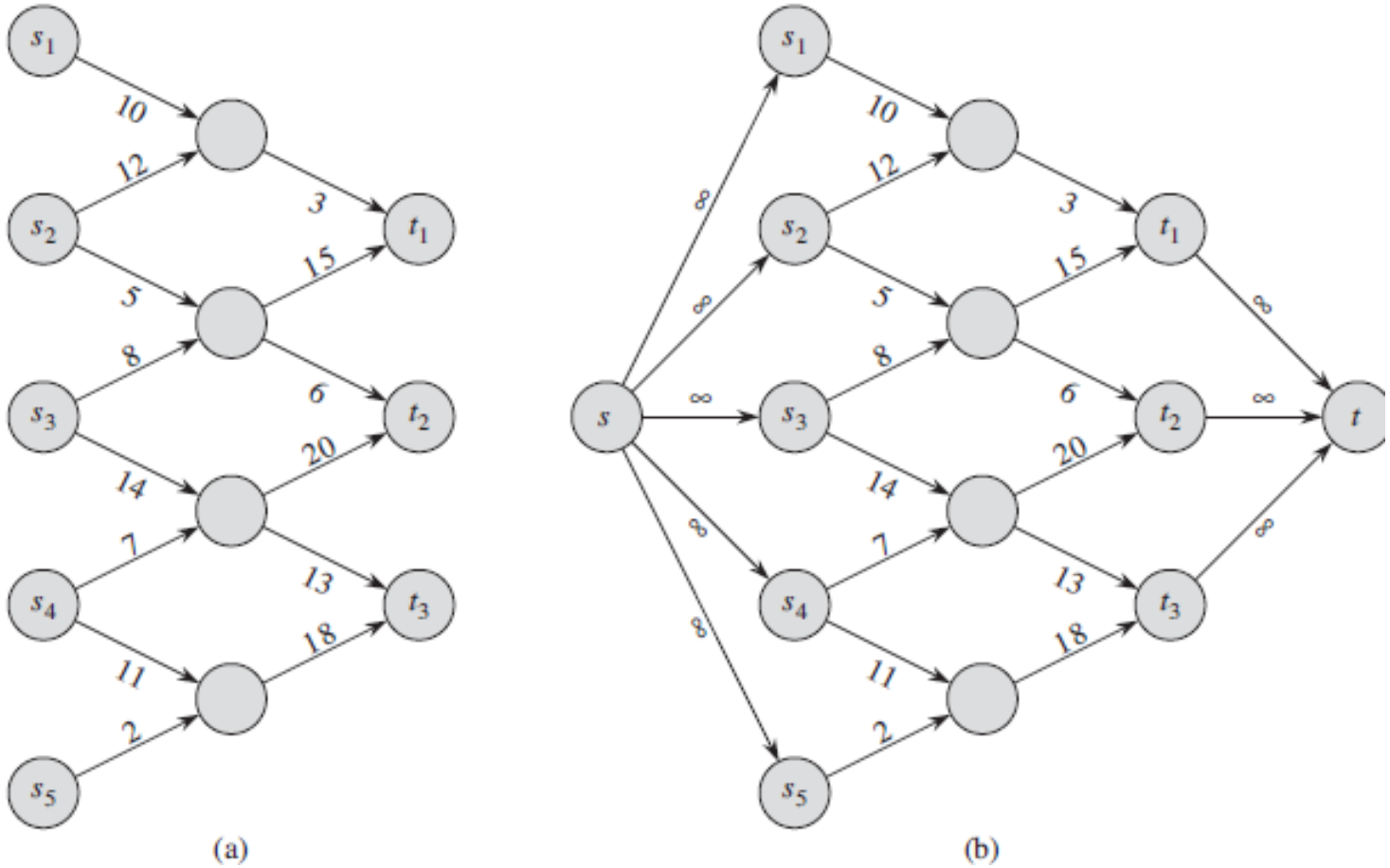
Το πρόβλημα της μέγιστης ροής μπορεί να εμπλέκει πολλαπλές πηγές και πολλαπλά sinks

Μπορούμε όμως να το αντιστοιχίσουμε στο αρχικό μας πρόβλημα εύκολα

Προσθέτουμε μια υπερ-πηγή (super-source) και ένα super-sink θέτοντας τη χωρητικότητά τους στο άπειρο

Maximum Flow (7/7)

Παράδειγμα



Ford-Fulkerson Method (1/17)

Υπάρχουν πολλές υλοποιήσεις με διαφορετικούς χρόνους εκτέλεσης

Βασίζεται στις ακόλουθες ιδέες:

- Residual networks
- Augmenting paths
- Cuts

Η μέθοδος αυξάνει επαναληπτικά την τιμή μιας ροής

Ξεκινάμε με f(u,v) = 0 για κάθε ζεύγος κόμβων

Σε κάθε επανάληψη, αυξάνουμε την τιμή της ροής στο G βρίσκοντας το augmenting path σε ένα residual network G_f

Ford-Fulkerson Method (2/17)

Όταν γνωρίζουμε τις ακμές του G_f είναι εύκολο στη συνέχεια να βρούμε ακμές του G για τις οποίες μπορούμε να αλλάξουμε τη ροή και να αυξήσουμε την τιμή της

Κατά τη διάρκεια εκτέλεσης του αλγορίθμου, η ροή σε οποιαδήποτε ακμή μπορεί να αυξηθεί ή να μειωθεί

Η μείωση μπορεί να είναι απαραίτητη ώστε να ‘αναγκαστεί’ ο αλγόριθμος να στείλει μεγαλύτερη ροή προς το sink

Ο αλγόριθμος θα σταματήσει όταν δεν μπορεί να αναγνωριστούν επιπλέον augmenting paths

Ford-Fulkerson Method (3/17)

FORD-FULKERSON-METHOD(G, s, t)

1 initialize flow f to 0
2 **while** there exists an augmenting path p in the residual network G_f
3 augment flow f along p
4 **return** f

Ford-Fulkerson Method (4/17)

Residual networks

- Δοσμένου ενός γράφου G και μιας ροής f, το residual network G_f αποτελείται από τις ακμές με χωρητικότητα που αναπαριστά το πώς μπορούμε να αλλάξουμε τη ροή στις ακμές του G
- Μια ακμή μπορεί να δεχτεί μια ποσότητα επιπρόσθετης ροής ίση με τη χωρητικότητα της ακμής μείον τη ροή σε αυτή την ακμή
- Αν η τιμή αυτή είναι θετική, βάζουμε την ακμή στο G_f με μια εναπομείνουσα χωρητικότητα $c_f(u, v) = c(u, v) - f(u, v)$
- Οι μόνες ακμές του G που μπορούν να μπουν στο G_f είναι αυτές που μπορούν να 'δεχτούν' μεγαλύτερη ροή
- Οι ακμές των οποίων η ροή είναι ίση με τη χωρητικότητά τους έχουν $c_f(u, v) = 0$ και δεν μπαίνουν στο G_f

Ford-Fulkerson Method (5/17)

Residual networks

- Ο αλγόριθμος προσπαθεί σε κάθε βήμα να αυξήσει την τιμή της συνολικής ροής
- Υπάρχει περίπτωση να προσπαθήσει να μειώσει τη ροή σε κάποιες ακμές
- Για την αναπαράσταση της μείωσης μιας θετικής ροής f(u,v) τοποθετούμε την ακμή (v,u) στο G_f με χωρητικότητα $c_f(v,u) = f(u,v)$
- Με αυτό τον τρόπο ‘ακυρώνουμε’ τη ροή της ακμής (u,v)
- Αυτές οι ακμές ονομάζονται αντίστροφες ακμές αφού στέλνουν πίσω τη ροή

Ford-Fulkerson Method (6/17)

Residual networks

- Η εναπομείνουσα χωρητικότητα $c_f(u,v)$ ορίζεται ως εξής:

$$c_f(u,v) = \begin{cases} c(u,v) - f(u,v) & \text{if } (u,v) \in E \\ f(v,u) & \text{if } (v,u) \in E \\ 0 & \text{otherwise} \end{cases}$$

- Παράδειγμα:
  - c(u,v)=16, f(u,v)=11; Η ροή μπορεί να αυξηθεί το πολύ κατά $c_f(u,v) = 5$
- Το residual network ορίζεται ως εξής:

$$E_f = \{(u,v) \in V \times V : c_f(u,v) > 0\} \qquad |E_f| \leq 2\,|E|$$

Ford-Fulkerson Method (7/17)

Residual networks

- Αν f είναι μια ροή στο G και f’ είναι μια ροή στο residual network G_f ορίζουμε $f \uparrow f'$ την αυξητική ροή από την f στην f’ ως μια συνάρτηση από το VXV στο R ως εξής:

$$(f \uparrow f')(u,v) = \begin{cases} f(u,v) + f'(u,v) - f'(v,u) & \text{if } (u,v) \in E \\ 0 & \text{otherwise} \end{cases}$$

Ford-Fulkerson Method (8/17)

Lemma

Let $G = (V, E)$ be a flow network with source s and sink t, and let f be a flow in G. Let G_f be the residual network of G induced by f, and let f' be a flow in G_f. Then the function $f \uparrow f'$ is a flow in G with value $|f \uparrow f'| = |f| + |f'|$.

Ford-Fulkerson Method (9/17)

Augmenting paths

- Ένα augmenting path είναι ένα απλό μονοπάτι από το s στο t στο residual network G_f

Lemma

Let $G = (V, E)$ be a flow network, let f be a flow in G, and let p be an augmenting path in G_f. Define a function $f_p : V \times V \to \mathbb{R}$ by

$$f_p(u, v) = \begin{cases} c_f(p) & \text{if } (u, v) \text{ is on } p \text{ ,} \\ 0 & \text{otherwise .} \end{cases}$$

Then, f_p is a flow in G_f with value $|f_p| = c_f(p) > 0$.

Ford-Fulkerson Method (10/17)

Corollary

Let $G = (V, E)$ be a flow network, let f be a flow in G, and let p be an augmenting path in G_f. Let f_p be defined as in equation (26.8), and suppose that we augment f by f_p. Then the function $f \uparrow f_p$ is a flow in G with value $|f \uparrow f_p| = |f| + |f_p| > |f|$.

Ford-Fulkerson Method (11/17)

Cuts

- Το βασικό ερώτημα στον αλγόριθμο είναι το πώς θα καταλάβουμε ότι πρέπει να σταματήσουμε
- Με βάση το θεώρημα **max-flow min-cut**, **μια ροή είναι η μέγιστη όταν και μόνο όταν αν το residual network δεν περιέχει κάποιο augmenting path**
- Μια τμηματοποίηση (cut) (S, T) είναι ένας διαχωρισμός των κόμβων V σε S & T=V-S τέτοιος ώστε το s να ανήκει στο S και το t να ανήκει στο T
- Αν η f είναι μια ροή, τότε η ροή δικτύου (net flow) f(S,T) μεταξύ της τμηματοποίησης (S, T) ορίζεται ως εξής:

$$f(S,T) = \sum_{u \in S} \sum_{v \in T} f(u,v) - \sum_{u \in S} \sum_{v \in T} f(v,u)$$

Ford-Fulkerson Method (12/17)

Cuts

- Η χωρητικότητα της τμηματοποίησης είναι

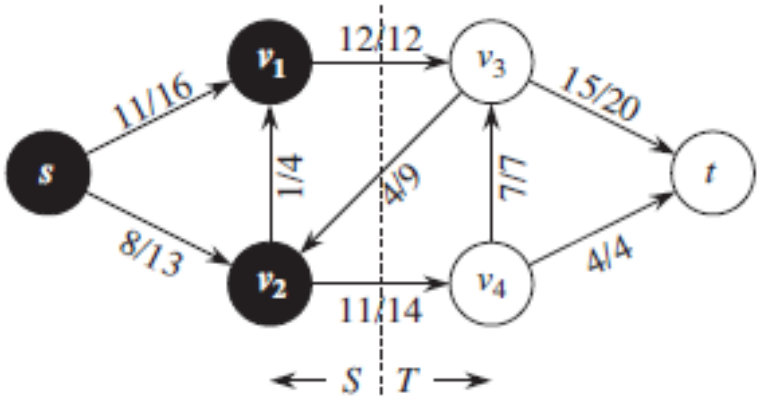
$$c(S,T) = \sum_{u \in S} \sum_{v \in T} c(u,v)$$

- Η ελάχιστη τμηματοποίηση ενός δικτύου κόμβων είναι μια τμηματοποίηση της οποίας η χωρητικότητα είναι η ελάχιστη σε σχέση με όλους τους διαχωρισμούς του δικτύου

Ford-Fulkerson Method (13/17)

Cuts

- Παράδειγμα



$$
\begin{aligned}
f(v_1, v_3) + f(v_2, v_4) - f(v_3, v_2) &= 12 + 11 - 4 \\
&= 19 \\
c(v_1, v_3) + c(v_2, v_4) &= 12 + 14 \\
&= 26
\end{aligned}
$$

Ford-Fulkerson Method (14/17)

Lemma

Let f be a flow in a flow network G with source s and sink t, and let (S, T) be any cut of G. Then the net flow across (S, T) is $f(S, T) = |f|$.

Corollary

The value of any flow f in a flow network G is bounded from above by the capacity of any cut of G.

Theorem (Max-flow min-cut theorem)

If f is a flow in a flow network $G = (V, E)$ with source s and sink t, then the following conditions are equivalent:

1. f is a maximum flow in G.
2. The residual network G_f contains no augmenting paths.
3. $|f| = c(S, T)$ for some cut (S, T) of G.

Ford-Fulkerson Method (15/17)

Βασικός αλγόριθμος

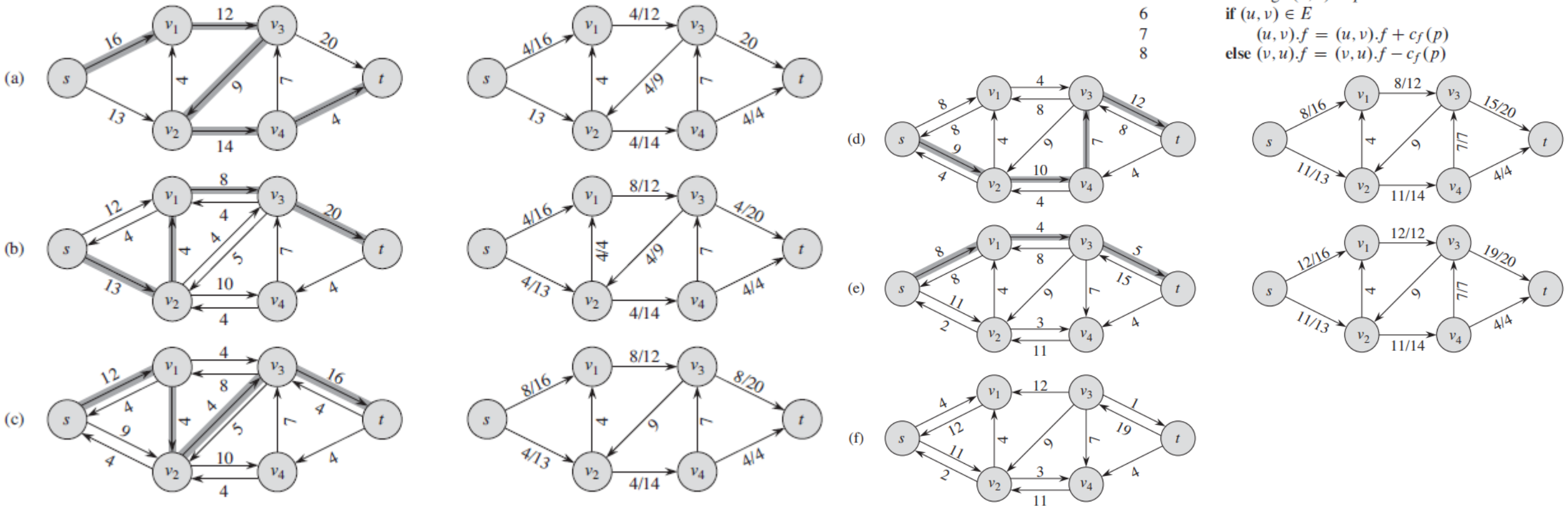
- Σε κάθε βήμα βρίσκουμε ένα augmenting path και τροποποιούμε τη ροή f
- Αντικαθιστούμε το f με $f \uparrow f_p$ και παίρνουμε μια νέα ροή με τιμή $|f| + |f_p|$

```
FORD-FULKERSON(G, s, t)
1  for each edge (u, v) ∈ G.E
2      (u, v).f = 0
3  while there exists a path p from s to t in the residual network G_f
4      c_f(p) = min {c_f(u, v) : (u, v) is in p}
5      for each edge (u, v) in p
6          if (u, v) ∈ E
7              (u, v).f = (u, v).f + c_f(p)
8          else (v, u).f = (v, u).f − c_f(p)
```

Ford-Fulkerson Method (16/17)

Παράδειγμα εκτέλεσης

```
FORD-FULKERSON(G, s, t)
1  for each edge (u, v) ∈ G.E
2      (u, v).f = 0
3  while there exists a path p from s to t in the residual network G_f
4      c_f(p) = min {c_f(u, v) : (u, v) is in p}
5      for each edge (u, v) in p
6          if (u, v) ∈ E
7              (u, v).f = (u, v).f + c_f(p)
8          else (v, u).f = (v, u).f − c_f(p)
```



Ford-Fulkerson Method (17/17)

Ανάλυση αλγορίθμου

- Ο χρόνος εκτέλεσης εξαρτάται από το χρόνο εύρεσης του augmenting path
- Με χρήση του BFS ο αλγόριθμος έχει πολυωνυμικό χρόνο εκτέλεσης
- Ο χρόνος για την εύρεση ενός μονοπατιού είναι O(E) είτε με BFS ή με DFS
- Ο συνολικός χρόνος εκτέλεσης του αλγορίθμου είναι $O(E|f^*|)$

Edmonds-Karp Algorithm (1/2)

Αποτελεί βελτίωση του Ford-Fulkerson

Βρίσκει το augmenting path μέσω αναζήτησης κατά πλάτος

Επιλέγει το augmenting path ως το συντομότερο μονοπάτι από το s στο t

Ο αλγόριθμος εκτελείται σε $O(V E^2)$

Edmonds-Karp Algorithm (2/2)

Lemma

If the Edmonds-Karp algorithm is run on a flow network $G = (V, E)$ with source s and sink t, then for all vertices $v \in V - \{s, t\}$, the shortest-path distance $\delta_f(s, v)$ in the residual network G_f increases monotonically with each flow augmentation.

Theorem

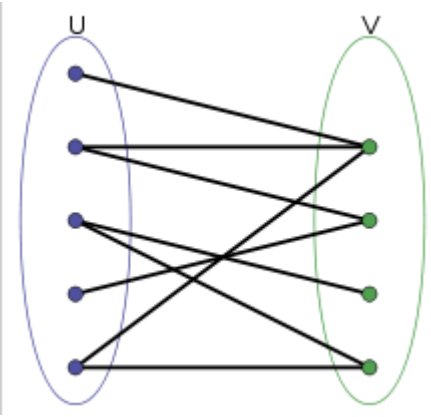
If the Edmonds-Karp algorithm is run on a flow network $G = (V, E)$ with source s and sink t, then the total number of flow augmentations performed by the algorithm is $O(VE)$.

Maximum Matching in Bipartite Graphs

Μέγιστο Διμερές Ταίριασμα (1/5)

Πρόβλημα

- Εύρεση του μέγιστου ταιριάσματος σε ένα **διμερή γράφο (bipartite graph)**
- Ένας διμερής γράφος είναι ένας γράφος του οποίου οι κόμβοι μπορούν να διαιρεθούν σε δύο αμοιβαία αποκλειόμενα και ανεξάρτητα σύνολα U, V τέτοια ώστε κάθε ακμή συνδέει ένα κόμβο του U με ένα κόμβο του V
- Παράδειγμα



Μέγιστο Διμερές Ταίριασμα (2/5)

Δοσμένου ενός γράφου G=(V,E), ένα **ταίριασμα (matching)** είναι ένα υποσύνολο ακμών M τέτοιο ώστε για όλους τους κόμβους που ανήκουν στο v ∈ V το πολύ μια ακμή είναι προσπίπτουσα προς τον κόμβο v

Λέμε πως ο v έχει ταιριάξει αν υπάρχει μια ακμή στο M ενώ σε αντίθετη περίπτωση λέμε πως ο κόμβος δεν έχει ταιριάξει

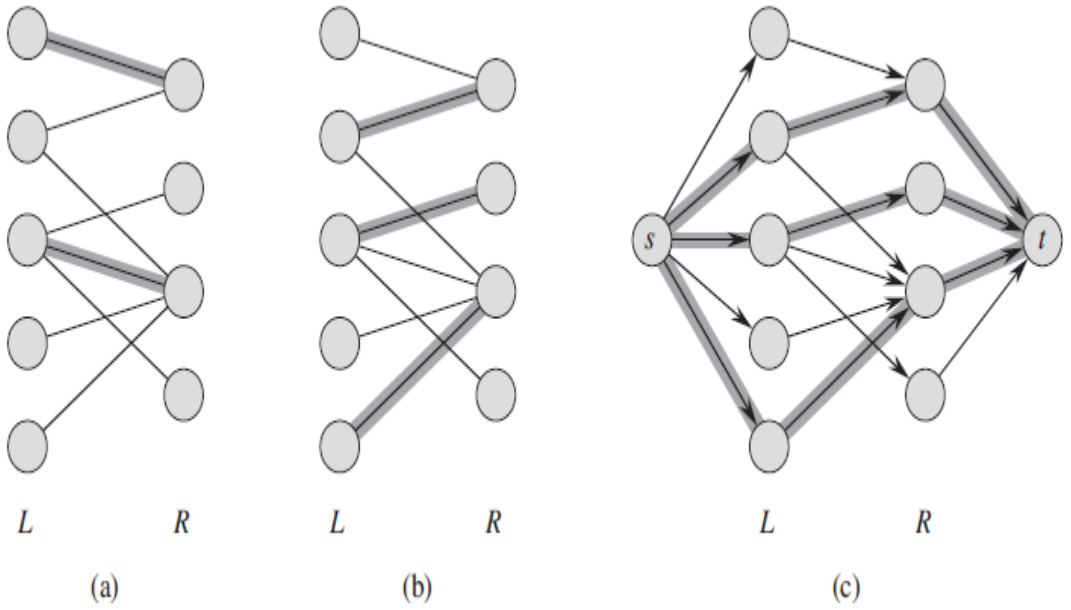
Ένα μέγιστο ταίριασμα είναι ένα ταίριασμα με τη μέγιστη πληθικότητα (cardinality), τέτοιο ώστε |M| >= |M'| για κάθε M'

Εστιάζουμε σε διμερείς γράφους

Μέγιστο Διμερές Ταίριασμα (3/5)

Λύση

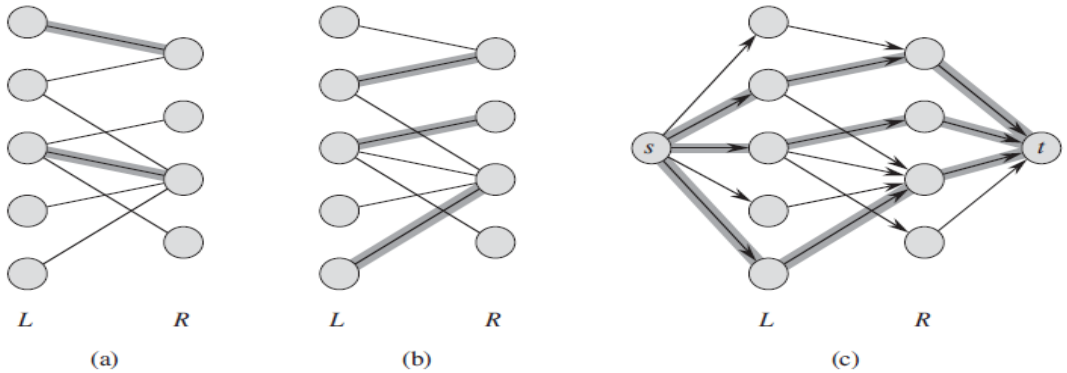
- Μπορούμε να χρησιμοποιήσουμε τον αλγόριθμο Ford-Fulkerson για την εύρεση του μέγιστου ταιριάσματος
- Κατασκευάζουμε ένα δίκτυο ροής όπου η ροές αντιστοιχούν με ταιριάσματα



Μέγιστο Διμερές Ταίριασμα (4/5)

Βήματα

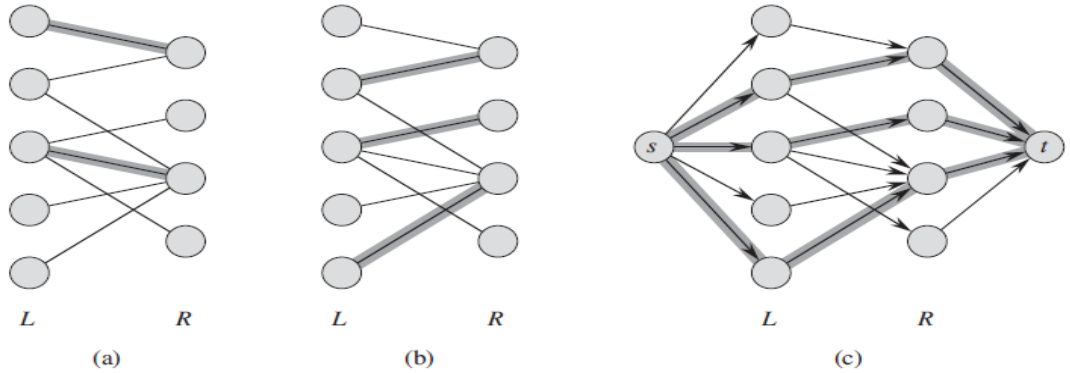
- Ορίζουμε δύο νέους κόμβους s, t
- Το νέο σύνολο των κόμβων του γράφου είναι V’=V ∪ {s,t}
- Αν ο διαχωρισμός των κόμβων είναι V = L ∪ R, οι κατευθυνόμενες ακμές του νέου γράφου G’ θα είναι οι ακμές που ανήκουν στο E που κατευθύνονται από το L στο R
- Προσθέτουμε |V| νέες ακμές στο E ώστε να παραχθεί το νέο σύνολο των ακμών E’
- Το E’ ορίζεται ως εξής: $E' = \{(s,u) : u \in L\} \cup \{(u,v) : (u,v) \in E\} \cup \{(v,t) : v \in R\}$



Μέγιστο Διμερές Ταίριασμα (5/5)

Βήματα (συνέχεια)

- Καθορίζουμε τη χωρητικότητα κάθε ακμής που ανήκει στο E’
- Αφού κάθε κόμβος στο V έχει τουλάχιστον μια προσπίπτουσα ακμή, τότε |E| >= |V|/2
- Ισχύει: |E|<=|E’|=|E|+|V|<= 3 |E|
- Συνεπώς |E’| = Θ(E)



Hash Tables

Hash Tables (1/24)

Η έννοια του **κατακερματισμού (hashing)** βασίζεται στην ιδέα της διασποράς κλειδιών σε ένα μονοδιάστατο πίνακα που ονομάζεται **πίνακας κατακερματισμού (hash table)**

Η κατανομή γίνεται μέσα από την εφαρμογή μιας συνάρτησης h που ονομάζεται **συνάρτηση κατακερματισμού (hash function)**

Για κάθε κλειδί, η συνάρτηση αποδίδει ένα ακέραιο σε ένα διάστημα [0..m-1] που ονομάζεται **διεύθυνση κατακερματισμού (hash address)**

Παράδειγμα: h(K)=K mod m

Hash Tables (2/24)

Απαιτήσεις για την hash function

- Το μέγεθος του hash table δεν πρέπει να είναι πολύ μεγάλο συγκρινόμενο με το πλήθος των κλειδιών προς κατακερματισμό και κατά συνέπεια να μην διακινδυνεύει την αύξηση του χρόνου εκτέλεσης
- Μια hash function πρέπει να ισοκατανέμει (όσο αυτό γίνεται) τα κλειδιά σε όλες τις θέσεις του πίνακα
- Μια hash function θα πρέπει να υπολογίζεται εύκολα

Hash Tables (3/24)

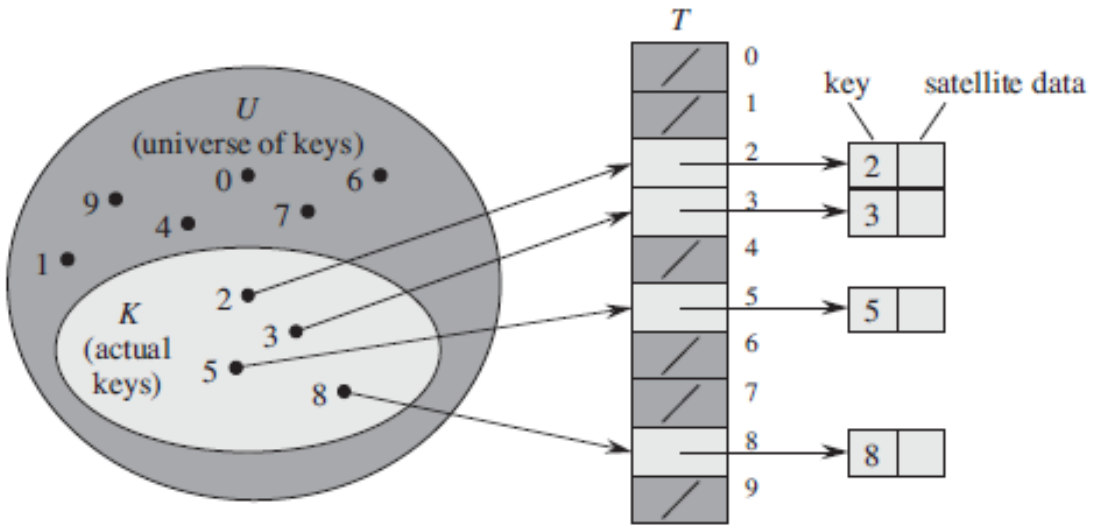
Για ένα hash table T[0..m-1] σε κάθε θέση (**slot**) του οποίου αντιστοιχεί ένα κλειδί k του περιβάλλοντός μας U μπορούμε να εκτελέσουμε τις ακόλουθες ενέργειες:

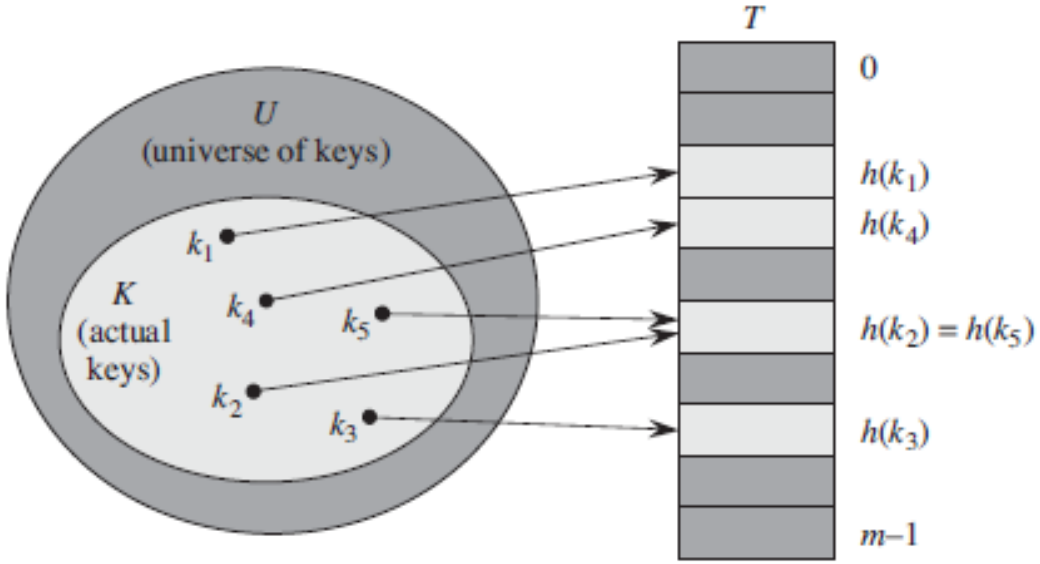
```
DIRECT-ADDRESS-SEARCH(T, k)
1  return T[k]

DIRECT-ADDRESS-INSERT(T, x)
1  T[x.key] = x

DIRECT-ADDRESS-DELETE(T, x)
1  T[x.key] = NIL
```

Κάθε μια από τις ενέργειες αυτές απαιτεί O(1) χρόνο

Hash Tables (4/24)

Παράδειγμα





Hash Tables (5/24)

Πρόβλημα

- Δύο κλειδιά μπορεί να κατακερματιστούν στην ίδια θέση του πίνακα
  - **Σύγκρουση (collision)**

Προσπαθούμε να βρούμε κατάλληλη και αποδοτική hash function

Εφαρμόζουμε διάφορες αποδοτικές τεχνικές

Hash Tables (6/24)

Λύση: chaining

Τοποθετούμε τα κλειδιά που κατακερματίζονται στην ίδια θέση του πίνακα σε μια συνδεδεμένη λίστα

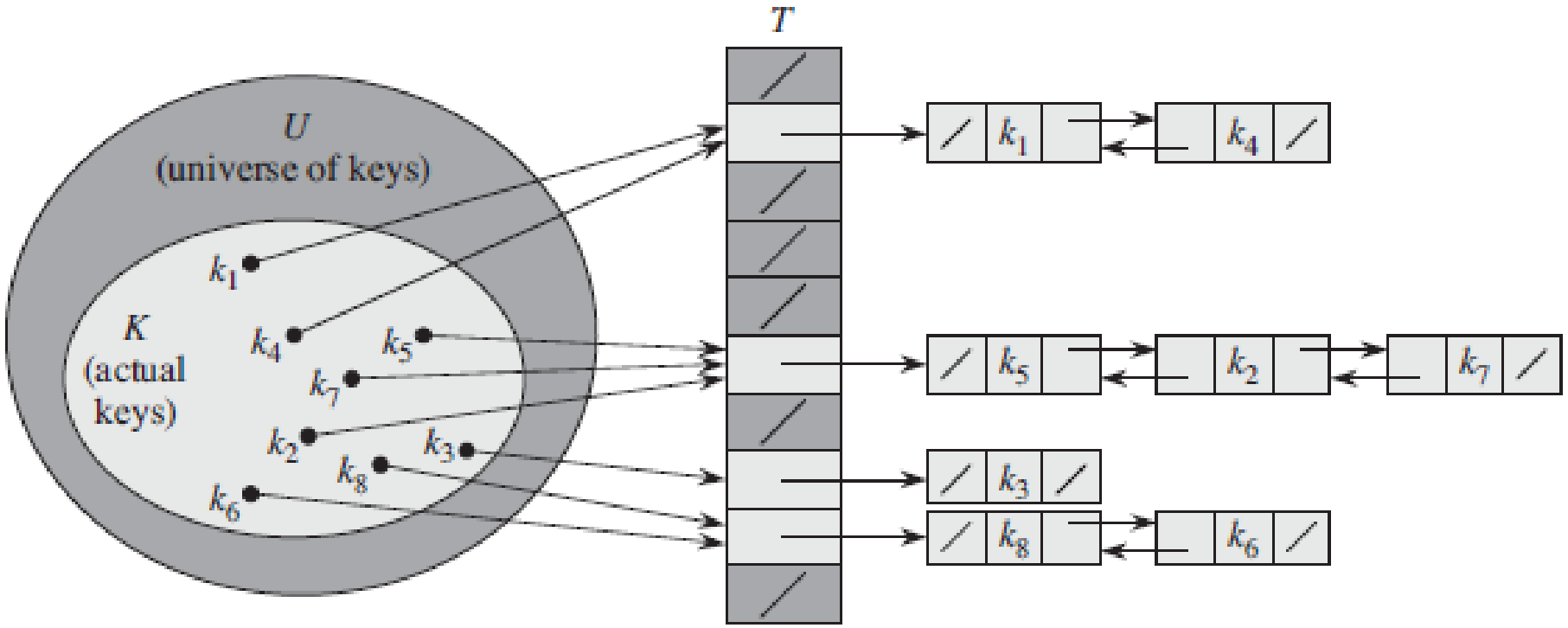
Κάθε θέση του πίνακα περιέχει ένα δείκτη προς την κορυφή της λίστας

Οι λειτουργίες γίνονται ως εξής:

```
CHAINED-HASH-INSERT(T, x)
1  insert x at the head of list T[h(x.key)]

CHAINED-HASH-SEARCH(T, k)
1  search for an element with key k in list T[h(k)]

CHAINED-HASH-DELETE(T, x)
1  delete x from the list T[h(x.key)]
```

Hash Tables (7/24)

Παράδειγμα



Hash Tables (8/24)

Ο χρόνος εισαγωγής στη χειρότερη περίπτωση είναι Ο(1)

Ο χρόνος αναζήτησης στη χειρότερη περίπτωση εξαρτάται από το μέγεθος της λίστας

Ορίζουμε τον **παράγοντα φόρτου (load factor)** του Τ ως το λόγο n/m όπου n είναι τα στοιχεία που πρόκειται να κατακερματιστούν και m είναι οι θέσεις του Τ

Η χειρότερη περίπτωση περιλαμβάνει την τοποθέτηση των n στοιχείων στην ίδια θέση

Η αναζήτηση θα γίνει σε Θ(n) επιπλέον του χρόνου υπολογισμού της συνάρτησης κατακερματισμού

Hash Tables (9/24)

Η μέση περίπτωση περιλαμβάνει την κατανομή των n κλειδιών σε διάφορες θέσεις του T

Αν κάθε κλειδί μπορεί με την ίδια πιθανότητα να τοποθετηθεί σε οποιαδήποτε θέση του T τότε έχουμε τον απλό ομοιόμορφο κατακερματισμό (simple uniform hashing)

Έστω n_j το μέγεθος της λίστας της θέσης T[j]

Συνεπώς έχουμε $n = n_0 + n_1 + \ldots + n_{m-1}$

Επίσης $E[n_j] = \alpha = n/m$

Hash Tables (10/24)

Η αναζήτηση εξαρτάται από τον αναμενόμενο πλήθος των στοιχείων σε κάθε λίστα

Θεώρημα 1: Σε ένα πίνακα κατακερματισμού Τ μια ανεπιτυχής αναζήτηση απαιτεί στη μέση περίπτωση χρόνο Θ(1+α) (Ο(1) είναι ο υπολογισμός του h(k))

Απόδειξη

Κάθε κλειδί κατακερματίζεται σε κάθε θέση με την ίδια πιθανότητα. Ο αναμενόμενος χρόνος για την ανεπιτυχή αναζήτηση είναι ίσος με τον αναμενόμενο χρόνο για να φτάσουμε στο τέλος της λίστας T[h(k)] με k να είναι το κλειδί που ψάχνουμε. Η συγκεκριμένη λίστα έχει αναμενόμενο μέγεθος nh(k). Οπότε ο αναμενόμενος αριθμός στοιχείων στα οποία ψάχνουμε είναι α.

Hash Tables (11/24)

Θεώρημα 2: Σε ένα πίνακα κατακερματισμού Τ, στη μέση περίπτωση, ο χρόνος επιτυχούς αναζήτησης είναι ίσος με Θ(1+α)

Απόδειξη

Έστω ότι έχουμε την ίδια πιθανότητα να αναζητήσουμε το κάθε ένα από τα n κλειδιά. Έστω x το στοιχείο που ψάχνουμε. Αφού κάθε νέο στοιχείο μπαίνει στην κορυφή της λίστας, αυτά τα στοιχεία θα μπαίνουμε προστά από το x και συνεπώς θα πρέπει να ψάχνουμε στο πλήθος των στοιχείων πριν από το x συν 1.

Πρέπει να βρούμε το αναμενόμενο πλήθος των στοιχείων που θα εξετάσουμε.

Hash Tables (12/24)

Απόδειξη (συνέχεια)

Αν x_i είναι το i στοιχείο που εισάγεται στον πίνακα και $k_i = x_i.key$. Ορίζουμε την τυχαία μεταβλητή $X_{ij}=I\{h(k_i)=h(k_j)\}$. Αφού υιοθετούμε ομοιόμορφη κατανομή ισχύει $Pr\{h(k_i)=h(k_j)\}=1/m$ και $E[X_{ij}]=1/m$.

Ο αναμενόμενος αριθμός των στοιχείων που εξετάζονται σε μια επιτυχή αναζήτηση είναι:

Hash Tables (13/24)

$$\mathrm{E}\left[\frac{1}{n}\sum_{i=1}^{n}\left(1+\sum_{j=i+1}^{n}X_{ij}\right)\right]$$

$$= \frac{1}{n}\sum_{i=1}^{n}\left(1+\sum_{j=i+1}^{n}\mathrm{E}\left[X_{ij}\right]\right)$$

$$= \frac{1}{n}\sum_{i=1}^{n}\left(1+\sum_{j=i+1}^{n}\frac{1}{m}\right)$$

$$= 1+\frac{1}{nm}\sum_{i=1}^{n}(n-i)$$

$$= 1+\frac{1}{nm}\left(\sum_{i=1}^{n}n-\sum_{i=1}^{n}i\right)$$

$$= 1+\frac{1}{nm}\left(n^2-\frac{n(n+1)}{2}\right)$$

$$= 1+\frac{n-1}{2m}$$

$$= 1+\frac{\alpha}{2}-\frac{\alpha}{2n}$$

$$\Theta(2+\alpha/2-\alpha/2n)=\Theta(1+\alpha)$$

Αναμενόμενος αριθμός συγκρίσεων σε επιτυχή αναζήτηση

Προσθέτουμε 1 για τον υπολογισμό της συνάρτησης

α=n/m ➔ m=n/α

Hash Tables (14/24)

Οι περισσότερες hash functions υποθέτουν πως τα κλειδιά είναι φυσικοί αριθμοί

Αν τα κλειδιά δεν είναι φυσικοί αριθμοί τότε εφαρμόζουμε μεθόδους για τη μετατροπή τους σε φυσικούς αριθμούς

Παράδειγμα: ένα σύνολο χαρακτήρων μπορεί να αναπαρασταθεί σαν μια ακολουθία ακεραίων αριθμών

Hash Tables (15/24)

The division method

- Υιοθετούμε το υπόλοιπο της διαίρεσης του κλειδιού k με το πλήθος των θέσεων m
- h(k) = k mod m
- Συχνά αποφεύγουμε κάποιες τιμές για το m
- Για παράδειγμα το m δεν πρέπει να είναι πολλαπλάσιο του 2 αφού m = 2^p και το h(k) είναι τα p κατώτερα bits του k
- Καλή επιλογή επιλογή για το m είναι ένας πρώτος αριθμός όχι κοντά σε μια δύναμη του 2

Hash Tables (16/24)

The multiplication method

- Υιοθετεί δύο βήματα
  - Πολλαπλασιάζουμε το k με μια σταθερά A και παίρνουμε ένα τμήμα του γινομένου
  - Πολλαπλασιάζουμε το αποτέλεσμα με το m και παίρνουμε το floor του αποτελέσματος
- h(k) = floor(m (k A mod 1)), το k A mod 1 απεικονίζει το k A – floor(k A)
- Εδώ η τιμή του m δεν είναι κρίσιμη οπότε την επιλέγουμε να είναι πολλαπλάσιο του 2

Hash Tables (17/24)

Όταν χρησιμοποιούμε μια σταθερή hash function κάποιος μπορεί να επιλέξει κλειδιά ώστε όλα να κατακερματιστούν στην ίδια θέση

Η λύση είναι να επιλέξουμε μια συνάρτηση που να είναι ανεξάρτητη από τα κλειδιά

Η προσέγγιση αυτή ονομάζεται **universal hashing**

Κατά την εκτέλεση επιλέγουμε τυχαία τη συνάρτηση μέσα από ένα σύνολο συναρτήσεων

Αφού γίνεται τυχαία επιλογή, τα αποτελέσματα μπορεί να είναι διαφορετικά ακόμα και για την ίδια είσοδο

Hash Tables (18/24)

Open addressing

- Κάθε στοιχείο του T περιέχει είτε ένα κλειδί ή NIL
- Στην αναζήτηση ψάχνουμε τα στοιχεία του πίνακα ώστε είτε να βρούμε το κλειδί που επιθυμούμε ή να καταλήξουμε ότι το κλειδί δεν υπάρχει
- Δεν χρησιμοποιούμε λίστες εκτός του πίνακα όπως γίνεται με την μέθοδο του chaining
- Ο T μπορεί να γεμίσει, οπότε δεν μπορούν να μπουν νέα στοιχεία

Hash Tables (19/24)

Εισαγωγή κλειδιού στο open addressing

- Εξετάζουμε τον T ώστε να βρούμε μια κενή θέση (probing)
- Η ακολουθία των θέσεων στις οποίες κάνουμε αναζήτηση εξαρτάται από το κλειδί που πρόκειται να εισαχθεί
- Επεκτείνουμε τη συνάρτηση κατακερματισμού ώστε να περιλαμβάνει τον αριθμό αναζήτησης ως δεύτερη είσοδο
- Απαιτείται μια ακολουθία αναζήτησης ως εξής: <h(k,0), h(k,1), …, h(k,m-1)> που είναι ένα permutation των <1,2, ..., m-1>
- Κάθε θέση εξετάζεται ισοπίθανα

Hash Tables (20/24)

```
HASH-INSERT(T, k)
1  i = 0
2  repeat
3      j = h(k, i)
4      if T[j] == NIL
5          T[j] = k
6          return j
7      else i = i + 1
8  until i == m
9  error "hash table overflow"
```

Hash Tables (21/24)

Αλγόριθμος αναζήτησης

- Αναζητά στις θέσεις που εξετάζει ο αλγόριθμος εισαγωγής όταν εισάγεται ένα κλειδί k
- Η αναζήτηση τερματίζεται όταν φτάσουμε σε μια κενή θέση

```
HASH-SEARCH(T, k)
1  i = 0
2  repeat
3       j = h(k, i)
4       if T[j] == k
5            return j
6       i = i + 1
7  until T[j] == NIL or i == m
8  return NIL
```

Hash Tables (22/24)

Linear probing

- Χρησιμοποιούμε μια βοηθητική συνάρτηση κατακερματισμού
- h(k,i) = (h’(k) + i) mod m
- Αρχικά αναζητούμε στο T[h’(k)]
- Υποφέρει από την **πρωταρχική συσταδοποίηση (primary clustering)**
  - Μεγάλες εκτελέσεις σε κατειλλημένες θέσεις αυξάνουν το μέσο χρόνο
  - Άδειες θέσεις ακολουθούνται από i γεμάτες θέσεις με πιθανότητα (i+1)/m

Hash Tables (23/24)

Quadratic probing

- $h(k,i) = (h'(k) + c_1 i + c_2 i^2) \bmod m$
- Η h' είναι μια βοηθητική συνάρτηση κατακερματισμού
- Τα c_1, c_2 είναι βοηθητικές σταθερές
- Αποδίδει καλύτερα από το linear probing αλλά για να γίνει η πλήρης χρήση του Τ θα πρέπει να περιορίσουμε τις σταθερές και το m
- Αν δύο κλειδιά έχουν την ίδια αρχική θέση αναζήτησης, τότε η ακολουθία τους θα είναι επίσης η ίδια
- Το φαινόμενο αυτό οδηγεί στη **δευτερεύουσα συσταδοποίηση (secondary clustering)**

Hash Tables (24/24)

Double hashing

- $h(k,i) = (h_1(k) + i\, h_2(k)) \bmod m$
- Οι h_1 kai h_2 είναι βοηθητικές συναρτήσεις κατακερματισμού
- Η αρχική θέση είναι η $T[h_1(k)]$
- Η ακολουθία αναζήτησης εξαρτάται δύο φορές στο κλειδί k
- Η τιμή $h_2(k)$ πρέπει να είναι ένας πρώτος σε σχέση με το μέγεθος m
  - Μια καλή λύση είναι να θέσουμε το m ως πολλαπλάσιο του 2 και την h_2 να βγάζει περιττούς αριθμούς
  - Άλλη λύση είναι να θέσουμε
    - $h_1(k) = k \bmod m$
    - $h_2(k) = 1 + (k \bmod m')$, m' επιλέγεται να είναι κατά τι μικρότερο του m (m'=m-1)

Ασκήσεις

Linear Probing

- Πρόκειται για την πιο απλή τεχνική
- Αν μια τιμή αποθηκευτεί στη θέση h(k) τότε υιοθετούμε την ακόλουθη συνάρτηση κατακερματισμού για να επιλύσουμε τη σύγκρουση

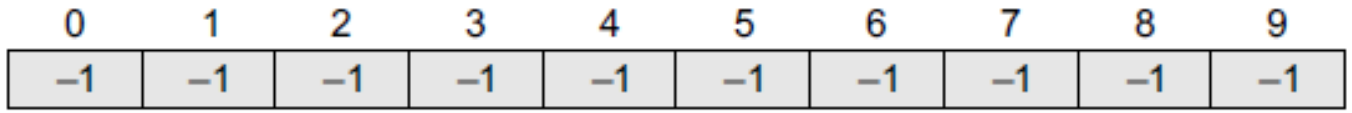
$$h(k,i)=[h'(k)+i] \bmod m$$

$$h'(k)=k \bmod m$$

- Το i απεικονίζει τον probe αριθμό που κινείται από 0 μέχρι m-1
- Για ένα κλειδί k υπολογίζεται η θέση $h'(k)=k \bmod m$ διότι την πρώτη φορά έχουμε i=0
- Αν η θέση είναι ελεύθερη, τότε το κλειδί αποθηκεύεται σε αυτή τη θέση
- Αν όχι, τότε υπολογίζουμε το δεύτερο probe $[h'(k)+1] \bmod m$

Linear Probing

- Παράδειγμα: να κατακερματιστούν τα κλειδιά 72, 27, 36, 24, 63, 81, 92, 101 σε πίνακα 10 θέσεων
- h’(k) = k mod m, m = 10



Linear Probing

- Παράδειγμα: να κατακερματιστούν τα κλειδιά 72, 27, 36, 24, 63, 81, 92, 101 σε πίνακα 10 θέσεων

Step 1

```
Key = 72
h(72, 0) = (72 mod 10 + 0) mod 10
         = (2) mod 10
         = 2
```

Since `T[2]` is vacant, insert key 72 at this location.

0	1	2	3	4	5	6	7	8	9
–1	–1	72	–1	–1	–1	–1	–1	–1	–1

Step 2

```
Key = 27
h(27, 0) = (27 mod 10 + 0) mod 10
         = (7) mod 10
         = 7
```

Since `T[7]` is vacant, insert key 27 at this location.

0	1	2	3	4	5	6	7	8	9
–1	–1	72	–1	–1	–1	–1	27	–1	–1

Linear Probing

- Παράδειγμα: να κατακερματιστούν τα κλειδιά 72, 27, 36, 24, 63, 81, 92, 101 σε πίνακα 10 θέσεων

Step 3 Key = 36

h(36, 0) = (36 mod 10 + 0) mod 10
= (6) mod 10
= 6

Since T[6] is vacant, insert key 36 at this location.

0	1	2	3	4	5	6	7	8	9
–1	–1	72	–1	–1	–1	36	27	–1	–1

Step 4 Key = 24

h(24, 0) = (24 mod 10 + 0) mod 10
= (4) mod 10
= 4

Since T[4] is vacant, insert key 24 at this location.

0	1	2	3	4	5	6	7	8	9
–1	–1	72	–1	24	–1	36	27	–1	–1

Linear Probing

- Παράδειγμα: να κατακερματιστούν τα κλειδιά 72. 27. 36. 24. 63. 81. 92. 101 σε πίνακα 10 θέσεων

Step 5 Key = 63

h(63, 0) = (63 mod 10 + 0) mod 10
= (3) mod 10
= 3

Since T[3] is vacant, insert key 63 at this location.

0	1	2	3	4	5	6	7	8	9
–1	–1	72	63	24	–1	36	27	–1	–1

Step 6 Key = 81

h(81, 0) = (81 mod 10 + 0) mod 10
= (1) mod 10
= 1

Since T[1] is vacant, insert key 81 at this location.

0	1	2	3	4	5	6	7	8	9
0	81	72	63	24	–1	36	27	–1	–1

Linear Probing

- Παράδειγμα: να κατακερματιστούν τα κλειδιά 72, 27, 36, 24, 63, 81, 92, 101 σε πίνακα 10 θέσεων

Step 7

Key = 92

h(92, 0) = (92 mod 10 + 0) mod 10
= (2) mod 10
= 2

Key = 92

h(92, 1) = (92 mod 10 + 1) mod 10
= (2 + 1) mod 10
= 3

Key = 92

h(92, 2) = (92 mod 10 + 2) mod 10
= (2 + 2) mod 10
= 4

Key = 92

h(92, 3) = (92 mod 10 + 3) mod 10
= (2 + 3) mod 10
= 5

Since T[5] is vacant, insert key 92 at this location.

0	1	2	3	4	5	6	7	8	9
–1	81	72	63	24	92	36	27	–1	–1

Linear Probing

- Παράδειγμα: να κατακερματιστούν τα κλειδιά 72, 27, 36, 24, 63, 81, 92, 101 σε πίνακα 10 θέσεων

- Μπορείτε να εκτελέσετε τον κατακερματισμό του τελευταίου αριθμού;

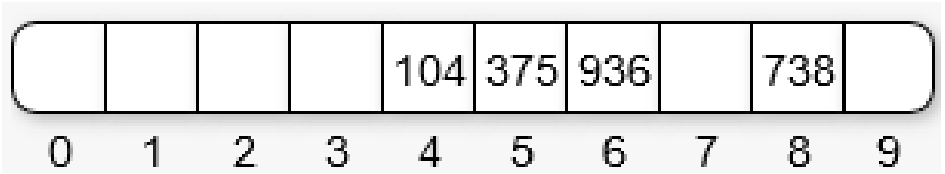
0	1	2	3	4	5	6	7	8	9
–1	81	72	63	24	92	36	27	–1	–1

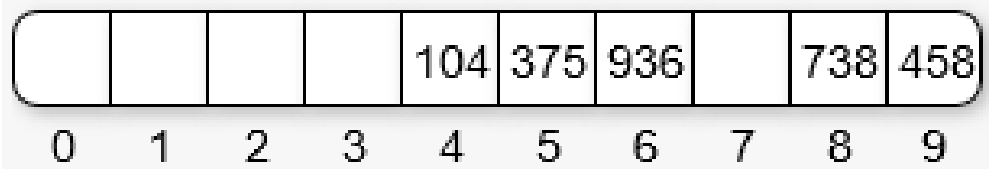
Linear Probing

- Αναζήτηση τιμής
  - Η διαδικασία είναι η ίδια με τη διαδικασία εισαγωγής της τιμής στον πίνακα
  - Επανυπολογίζουμε την τιμή της θέσης του πίνακα
  - Αν η τιμή που ψάχνουμε δεν ταιριάξει με την τιμή που πήραμε από τη θέση τότε εκκινούμε μια σειριακή αναζήτηση
  - Τα αποτελέσματα της διαδικασίας μπορεί να είναι:
    - Να βρεθεί η τιμή
    - Να καταλήξουμε σε θέση που έχει το -1 οπότε η τιμή δεν υπάρχει
    - Να φτάσουμε στο τέλος του πίνακα

Linear Probing

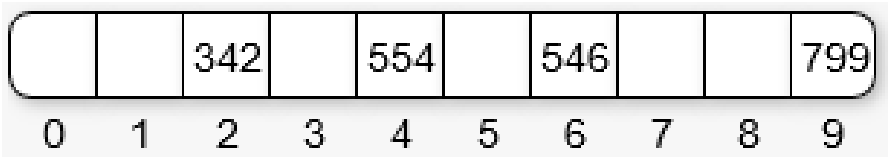
- Άσκηση:
- Στον επόμενο πίνακα σε ποια θέση θα τοποθετηθεί το 458 όταν h(k)=k mod 10;

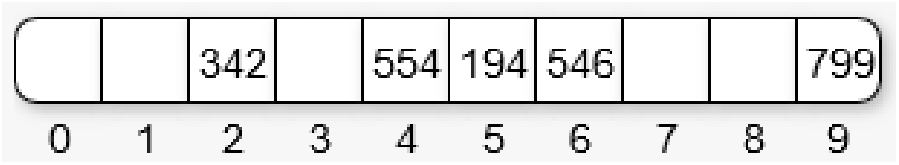




Linear Probing

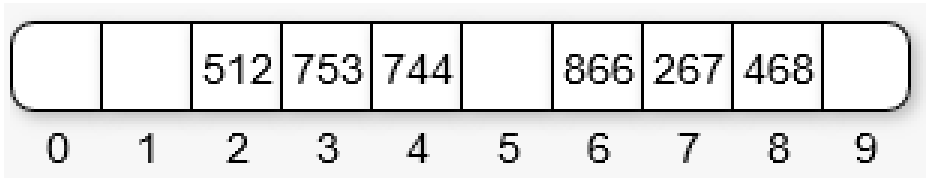
- Άσκηση:
- Στον επόμενο πίνακα σε ποια θέση θα τοποθετηθεί το 194 όταν h(k)=k mod 10;

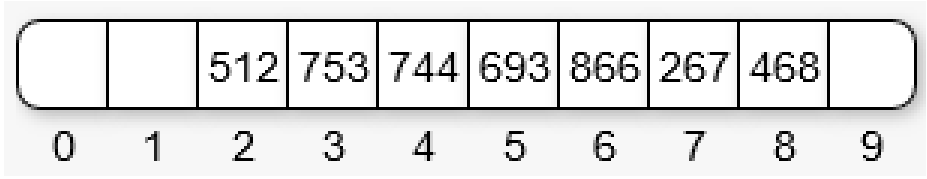




Linear Probing

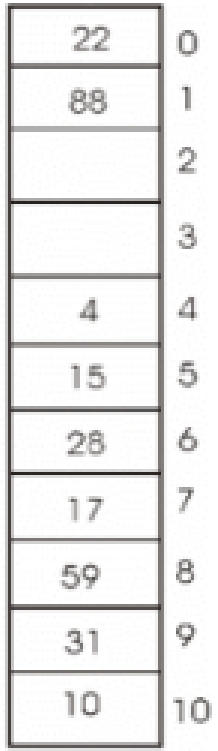
- Άσκηση:
- Στον επόμενο πίνακα σε ποια θέση θα τοποθετηθεί το 693 όταν h(k)=k mod 10;





Linear Probing

- Άσκηση:
- Να κατακερματίσετε τα κλειδιά: 10, 22, 31, 4, 15, 28, 17, 88, 59 σε πίνακα μεγέθους 11



Quadratic Probing

- Σε αυτή την τεχνική, αν μια τιμή υπάρχει ήδη στη θέση h(k) τότε ψάχνουμε μια ελεύθερη θέση με την ακόλουθη συνάρτηση κατακερματισμού:

$$h(k,i)=[h'(k) + c_1 i + c_2 i^2] \bmod m$$

$$h'(k)=k \bmod m$$

- Τα c_1, c_2 είναι σταθερές

Quadratic Probing

- Παράδειγμα:
- Να κατακερματιστούν τα ακόλουθα κλειδιά σε πίνακα 10 θέσεων: 72, 27, 36, 24, 63, 81, 101 (c_1=1, c_2=3)

0	1	2	3	4	5	6	7	8	9
–1	–1	–1	–1	–1	–1	–1	–1	–1	–1

Quadratic Probing

- Παράδειγμα:
- Να κατακερματιστούν τα ακόλουθα κλειδιά σε πίνακα 10 θέσεων: 72, 27, 36, 24, 63, 81, 101 (c_1=1, c_2=3)

Step 1 Key = 72

h(72, 0) = [72 mod 10 + 1 × 0 + 3 × 0] mod 10

= [72 mod 10] mod 10

= 2 mod 10

= 2

Since T[2] is vacant, insert the key 72 in T[2]. The hash table now becomes:

0	1	2	3	4	5	6	7	8	9
–1	–1	72	–1	–1	–1	–1	–1	–1	–1

Quadratic Probing

- Παράδειγμα:
- Να κατακερματιστούν τα ακόλουθα κλειδιά σε πίνακα 10 θέσεων: 72, 27, 36, 24, 63, 81, 101 (c_1=1, c_2=3)

Step 2

Key = 27

h(27, 0) = [27 mod 10 + 1 × 0 + 3 × 0] mod 10

= [27 mod 10] mod 10

= 7 mod 10

= 7

Since T[7] is vacant, insert the key 27 in T[7]. The hash table now becomes:

0	1	2	3	4	5	6	7	8	9
–1	–1	72	–1	–1	–1	–1	27	–1	–1

Step 3

Key = 36

h(36, 0) = [36 mod 10 + 1 × 0 + 3 × 0] mod 10

= [36 mod 10] mod 10

= 6 mod 10

= 6

Since T[6] is vacant, insert the key 36 in T[6]. The hash table now becomes:

0	1	2	3	4	5	6	7	8	9
–1	–1	72	–1	–1	–1	36	27	–1	–1

Quadratic Probing

- Παράδειγμα:
- Να κατακερματιστούν τα ακόλουθα κλειδιά σε πίνακα 10 θέσεων: 72, 27, 36, 24, 63, 81, 101 (c_1=1, c_2=3)

Step 4 Key = 24

$h(24, 0) = [24 \bmod 10 + 1 \times 0 + 3 \times 0] \bmod 10$

$= [24 \bmod 10] \bmod 10$

$= 4 \bmod 10$

$= 4$

Since T[4] is vacant, insert the key 24 in T[4]. The hash table now becomes:

0	1	2	3	4	5	6	7	8	9
–1	–1	72	–1	24	–1	36	27	–1	–1

Step 5 Key = 63

$h(63, 0) = [63 \bmod 10 + 1 \times 0 + 3 \times 0] \bmod 10$

$= [63 \bmod 10] \bmod 10$

$= 3 \bmod 10$

$= 3$

Since T[3] is vacant, insert the key 63 in T[3]. The hash table now becomes:

0	1	2	3	4	5	6	7	8	9
–1	–1	72	63	24	–1	36	27	–1	–1

Quadratic Probing

- Παράδειγμα:
- Να κατακερματιστούν τα ακόλουθα κλειδιά σε πίνακα 10 θέσεων: 72, 27, 36, 24, 63, 81, 101 (c_1=1, c_2=3)

Step 6 Key = 81

$$h(81,0) = [81 \bmod 10 + 1 \times 0 + 3 \times 0] \bmod 10$$
$$= [81 \bmod 10] \bmod 10$$
$$= 81 \bmod 10$$
$$= 1$$

Since T[1] is vacant, insert the key 81 in T[1]. The hash table now becomes:

0	1	2	3	4	5	6	7	8	9
–1	81	72	63	24	–1	36	27	–1	–1

Quadratic Probing

- Παράδειγμα:
- Να κατακερματιστούν τα ακόλουθα κλειδιά σε πίνακα 10 θέσεων: 72, 27, 36, 24, 63, 81, 101 (c_1=1, c_2=3)

Step 7

Key = 101

h(101,0) = [101 mod 10 + 1 × 0 + 3 × 0] mod 10
= [101 mod 10 + 0] mod 10
= 1 mod 10
= 1

Key = 101

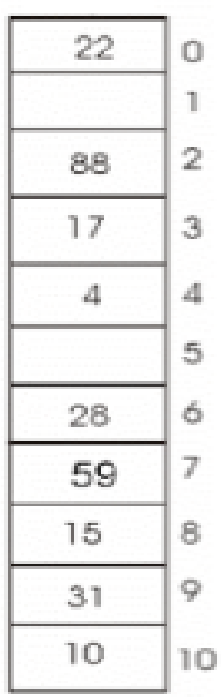
h(101,0) = [101 mod 10 + 1 × 1 + 3 × 1] mod 10
= [101 mod 10 + 1 + 3] mod 10
= [101 mod 10 + 4] mod 10
= [1 + 4] mod 10
= 5 mod 10
= 5

Since T[5] is vacant, insert the key 101 in T[5]. The hash table now becomes:

0	1	2	3	4	5	6	7	8	9
–1	81	72	63	24	101	36	27	–1	–1

Quadratic Probing

- Να κατακερματιστούν τα ακόλουθα κλειδιά σε πίνακα 11 θέσεων: 10, 22, 31, 4, 15, 28, 17, 88, 59 (c_1=1, c_2=3)



Quadratic Probing

- Να κατακερματιστούν τα ακόλουθα κλειδιά σε πίνακα 11 θέσεων: 12,44,13,88,23,94,11,39,20,16 (c_1=1, c_2=3) με τη hash function h(k)=(2k+5) mod m

23	**20**	**11**		**16**	**44**	**94**	**12**	**88**	**13**	**39**

Double Hashing

- Η τεχνική αυτή εξάγει μια τιμή και στη συνέχεια επαναληπτικά προχωρά μέχρι να βρει κενή θέση όπως ακριβώς και οι δύο προηγούμενες τεχνικές
- Υιοθετεί δύο συναρτήσεις κατακερματισμού που είναι ανεξάρτητες
- Η δεύτερη υπολογίζει το βήμα μετακίνησης πάνω στις θέσεις του πίνακα
- Η τελική συνάρτηση κατακερματισμού έχει ως εξής:

$$h(k,i)=[h_1(k) + ih_2(k)] \bmod m$$

$$h_1(k)=k \bmod m$$

$$h_2(k)=k \bmod m'$$

Double Hashing

- Το m’ επιλέγεται να είναι μικρότερο του m
- Μπορούμε να επιλέξουμε m’=m-1, m’=m-2

Double Hashing

- Παράδειγμα:
- Να κατακερματιστούν τα ακόλουθα κλειδιά σε πίνακα 10 θέσεων: 72, 27, 36, 24, 63, 81, 92, 101 με h_1 = (k mod 10) και h_2 = (k mod 8).

0	1	2	3	4	5	6	7	8	9
–1	–1	–1	–1	–1	–1	–1	–1	–1	–1

Double Hashing

- Παράδειγμα:
- Να κατακερματιστούν τα ακόλουθα κλειδιά σε πίνακα 10 θέσεων: 72, 27, 36, 24, 63, 81, 92, 101 με h_1 = (k mod 10) και h_2 = (k mod 8).

Step 1 Key = 72

$h(72, 0) = [72 \bmod 10 + (0 \times 72 \bmod 8)] \bmod 10$

$= [2 + (0 \times 0)] \bmod 10$

$= 2 \bmod 10$

$= 2$

Since T[2] is vacant, insert the key 72 in T[2]. The hash table now becomes:

0	1	2	3	4	5	6	7	8	9
–1	–1	72	–1	–1	–1	–1	–1	–1	–1

Step 2 Key = 27

$h(27, 0) = [27 \bmod 10 + (0 \times 27 \bmod 8)] \bmod 10$

$= [7 + (0 \times 3)] \bmod 10$

$= 7 \bmod 10$

$= 7$

Since T[7] is vacant, insert the key 27 in T[7]. The hash table now becomes:

0	1	2	3	4	5	6	7	8	9
–1	–1	72	–1	–1	–1	–1	27	–1	–1

Double Hashing

- Παράδειγμα:
- Να κατακερματιστούν τα ακόλουθα κλειδιά σε πίνακα 10 θέσεων: 72, 27, 36, 24, 63, 81, 92, 101 με h_1 = (k mod 10) και h_2 = (k mod 8).

Step 3 Key = 36

$h(36, 0) = [36 \bmod 10 + (0 \times 36 \bmod 8)] \bmod 10$

$= [6 + (0 \times 4)] \bmod 10$

$= 6 \bmod 10$

$= 6$

Since T[6] is vacant, insert the key 36 in T[6]. The hash table now becomes:

0	1	2	3	4	5	6	7	8	9
–1	–1	72	–1	–1	–1	36	27	–1	–1

Step 4 Key = 24

$h(24, 0) = [24 \bmod 10 + (0 \times 24 \bmod 8)] \bmod 10$

$= [4 + (0 \times 0)] \bmod 10$

$= 4 \bmod 10$

$= 4$

Since T[4] is vacant, insert the key 24 in T[4]. The hash table now becomes:

0	1	2	3	4	5	6	7	8	9
–1	–1	72	–1	24	–1	36	27	–1	–1

Double Hashing

- Παράδειγμα:
- Να κατακερματιστούν τα ακόλουθα κλειδιά σε πίνακα 10 θέσεων: 72, 27, 36, 24, 63, 81, 92, 101 με h_1 = (k mod 10) και h_2 = (k mod 8).

Step 5 Key = 63

$h(63, 0) = [63 \bmod 10 + (0 \times 63 \bmod 8)] \bmod 10$

$= [3 + (0 \times 7)] \bmod 10$

$= 3 \bmod 10$

$= 3$

Since T[3] is vacant, insert the key 63 in T[3]. The hash table now becomes:

0	1	2	3	4	5	6	7	8	9
–1	–1	72	63	24	–1	36	27	–1	–1

Step 6 Key = 81

$h(81, 0) = [81 \bmod 10 + (0 \times 81 \bmod 8)] \bmod 10$

$= [1 + (0 \times 1)] \bmod 10$

$= 1 \bmod 10$

$= 1$

Since T[1] is vacant, insert the key 81 in T[1]. The hash table now becomes:

0	1	2	3	4	5	6	7	8	9
–1	81	72	63	24	–1	36	27	–1	–1

Double Hashing

- Παράδειγμα:
- Να κατακερματιστούν τα ακόλουθα κλειδιά σε πίνακα 10 θέσεων: 72, 27, 36, 24, 63, 81, 92, 101 με h_1 = (k mod 10) και h_2 = (k mod 8).

Step 7

Key = 92

h(92, 0) = [92 mod 10 + (0 × 92 mod 8)] mod 10
= [2 + (0 × 4)] mod 10
= 2 mod 10
= 2

Key = 92

h(92, 1) = [92 mod 10 + (1 × 92 mod 8)] mod 10
= [2 + (1 × 4)] mod 10
= (2 + 4) mod 10
= 6 mod 10
= 6

Key = 92

h(92, 2) = [92 mod 10 + (2 × 92 mod 8)] mod 10
= [2 + (2 × 4)] mod 10
= [2 + 8] mod 10
= 10 mod 10
= 0

Since T[0] is vacant, insert the key 92 in T[0]. The hash table now becomes:

0	1	2	3	4	5	6	7	8	9
92	81	72	63	24	–1	36	27	–1	–1

Double Hashing

- Παράδειγμα:
- Να κατακερματιστούν τα ακόλουθα κλειδιά σε πίνακα 10 θέσεων: 72, 27, 36, 24, 63, 81, 92, 101 με h_1 = (k mod 10) και h_2 = (k mod 8).

Step 8

Key = 101

$h(101, 0) = [101 \bmod 10 + (0 \times 101 \bmod 8)] \bmod 10$

$= [1 + (0 \times 5)] \bmod 10$

$= 1 \bmod 10$

$= 1$

Key = 101

$h(101, 1) = [101 \bmod 10 + (1 \times 101 \bmod 8)] \bmod 10$

$= [1 + (1 \times 5)] \bmod 10$

$= [1 + 5] \bmod 10$

$= 6$

- Συνεχίζουμε με τον ίδιο τρόπο .....

Double Hashing

- Να κατακερματιστούν τα κλειδιά: 14, 17, 25, 37, 34, 16, 26 με M=11, $h_1(k) = k \bmod 11$, $h_2(k) = k \bmod 7 + 1$

	34		**14**	**37**	**16**	**17**		**25**		**26**

Linear Programming

Linear Programming (1/27)

Πολλά προβλήματα έχουν τη μορφή της μεγιστοποίησης ή ελαχιστοποίησης μιας (συνήθως) συνάρτησης κάτω από προϋποθέσεις

Η συνάρτηση είναι συνήθως γραμμική

Οι προυποθέσεις / περιορισμοί (constraint) παίρνουν τη μορφή ισοτήτων ή ανισοτήτων

Σε αυτές τις περιπτώσεις έχουμε **ένα πρόβλημα γραμμικού προγραμματισμού (linear programming problem)**

Linear Programming (2/27)

Η γενική μορφή του προβλήματος είναι να προσπαθήσουμε να βελτιστοποιήσουμε μια γραμμική συνάρτηση όταν ισχύει ένα σύνολο γραμμικών ανισοτήτων

Δοσμένου ενός συνόλου αριθμών $\alpha_1, \alpha_2, \ldots, \alpha_n$ και ενός συνόλου μεταβλητών $x_1, x_2, \ldots, x_n$, ορίζουμε μια γραμμική συνάρτηση πάνω σε αυτές τις μεταβλητές ως εξής:

$$f(x_1, x_2, \ldots, x_n) = a_1x_1 + a_2x_2 + \cdots + a_nx_n = \sum_{j=1}^{n} a_j x_j$$

Αν b είναι ένας πραγματικός αριθμός και f μια γραμμική συνάρτηση τότε οι γραμμικές ισότητες / ανισώσεις έχουν ως εξής

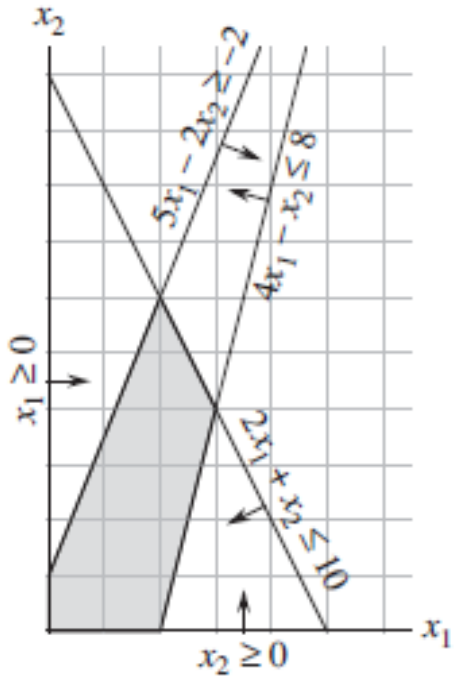
$$f(x_1, x_2, \ldots, x_n) = b \quad f(x_1, x_2, \ldots, x_n) \leq b \quad f(x_1, x_2, \ldots, x_n) \geq b$$

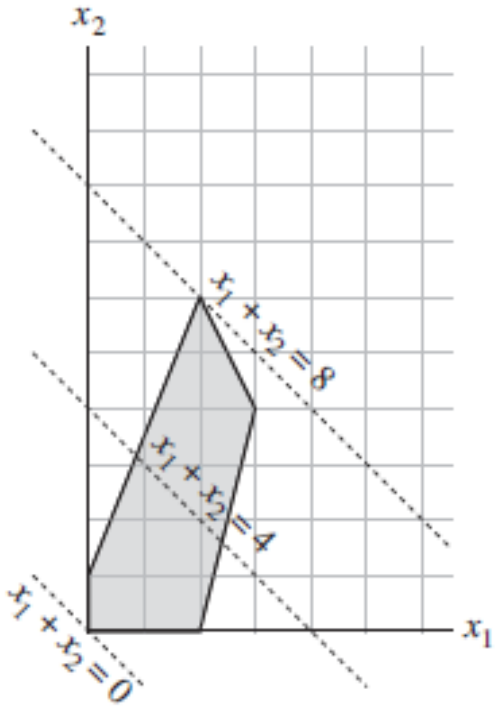
Linear Programming (3/27)

Παράδειγμα

$$
\begin{array}{lrcrcr}
\text{maximize} & x_1 & + & x_2 & & \\
\text{subject to} & & & & & \\
& 4x_1 & - & x_2 & \leq & 8 \\
& 2x_1 & + & x_2 & \leq & 10 \\
& 5x_1 & - & 2x_2 & \geq & -2 \\
& x_1, x_2 & & & \geq & 0
\end{array}
$$

Πιθανές λύσεις





Linear Programming (4/27)

Παραδείγματα εφαρμογών

- Αεροπορική εταιρεία επιθυμεί να διαχειριστεί το πρόγραμμα των πληρωμάτων της. Οι αρχές θέτουν ένα σύνολο περιορισμών σχετικά με τις ώρες εργασίας καθώς και τον τύπο αεροπλάνου όπου θα εργαστεί κάθε πλήρωμα. Η εταιρεία επιθυμεί να μεγιστοποιήσει το όφελος όταν θα αναθέσει εργασίες στον ελάχιστο αριθμό πληρώματος
- Μια εταιρεία εξόρυξης πετρελαίου επιθυμεί να εντοπίσει το σημείο όπου θα κάνει την επόμενη γεώτρηση. Θα πρέπει να λάβει υπόψιν της το κόστος τοποθέτησης των μηχανημάτων και το αναμενόμενο όφελος. Η εταιρεία έχει περιορισμένους χρηματικούς πόρους για να κάνει την εξόρυξη και επιθυμεί να μεγιστοποιήσει την ποσότητα που θα εξάγει και το όφελος.

Linear Programming (5/27)

Η μέθοδος simplex είναι ο κλασικός τρόπος για την επίλυση προβλημάτων γραμμικού προγραμματισμού

Η μέθοδος, πρακτικά, είναι πολύ γρήγορη

Για να εφαρμόσουμε τη μέθοδο σε προβλήματα γραμμικού προγραμματισμού πρέπει να εκφράσουμε το πρόβλημα στην **τυποποιημένη μορφή (standard form)**

Linear Programming (6/27)

Η standard form έχει τις ακόλουθες απαιτήσεις:

- Πρέπει να ορίζεται ένα πρόβλημα **μεγιστοποίησης**
- **Όλοι οι περιορισμοί** (εκτός από τους μη αρνητικούς περιορισμούς) πρέπει να έχουν τη μορφή γραμμικών εξισώσεων με μη αρνητικά δεξιά μέρη
- **Όλες οι μεταβλητές** πρέπει να είναι μη αρνητικές

Το κύριο πλεονέκτημά της είναι στον απλό μηχανισμό που προσφέρει για να αναγνωρίσει τα οριακά σημεία της περιοχής των λύσεων

Linear Programming (7/27)

Υποθέτουμε ότι έχουμε **m** περιορισμούς και **n** μεταβλητές (n>=m)

Η γενική μορφή ενός προβλήματος γραμμικού προγραμματισμού έχει ως εξής:

$$\begin{aligned} \text{maximize} \quad & c_1x_1 + \cdots + c_nx_n \\ \text{subject to} \quad & a_{i1}x_1 + \cdots + a_{in}x_n = b_i, \quad b_i \geq 0 \text{ for } i = 1, 2, \ldots, m \\ & x_1 \geq 0, \ldots, x_n \geq 0 \end{aligned}$$

ή με την υιοθέτηση συμβολισμού πινάκων

$$\begin{aligned} \text{maximize} \quad & cx \\ \text{subject to} \quad & Ax = b \\ & x \geq 0 \end{aligned}$$

Linear Programming (8/27)

Όπου

$$c = [c_1\ c_2\ \ldots\ c_n],\quad x = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix},\quad A = \begin{bmatrix} a_{11} & a_{12} & \ldots & a_{1n} \\ \vdots & \vdots & & \vdots \\ a_{m1} & a_{m2} & \ldots & a_{mn} \end{bmatrix},\quad b = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_m \end{bmatrix}$$

Οποιοδήποτε πρόβλημα γραμμικού προγραμματισμού μπορεί να γραφεί στην standard form

Linear Programming (9/27)

Όταν πρέπει να ελαχιστοποιήσουμε μια συνάρτηση, μπορεί να αντικατασταθεί από ένα ισοδύναμο πρόβλημα μεγιστοποίησης της ίδιας συνάρτησης αλλά έχουμε αντικαταστήσει τα c_j με $-c_j$

Όταν ένας περιορισμός δίνεται σαν μια ανίσωση, μπορεί να αντικατασταθεί από μια ισοδύναμη εξίσωση προσθέτωντας μια **χαλαρή μεταβλητή** (slack variable) που παριστάνει τη διαφορά των δύο μερών της αρχικής ανίσωσης

Παράδειγμα

$$\sum_{j=1}^{n} a_{ij}x_j \leq b_i \quad \rightarrow \quad \begin{aligned} s &= b_i - \sum_{j=1}^{n} a_{ij}x_j \\ s &\geq 0 \end{aligned}$$

Linear Programming (10/27)

Παράδειγμα

$$\begin{aligned} \text{maximize} \quad & 3x + 5y \\ \text{subject to} \quad & x + y \le 4 \\ & x + 3y \le 6 \\ & x \ge 0, \quad y \ge 0 \end{aligned}$$

→

$$\begin{aligned} \text{maximize} \quad & 3x + 5y + 0u + 0v \\ \text{subject to} \quad & x + y + u = 4 \\ & x + 3y + v = 6 \\ & x, y, u, v \ge 0 \end{aligned}$$

Άλλη μορφή συμβολισμού

$$x\begin{bmatrix}1\\1\end{bmatrix} + y\begin{bmatrix}1\\3\end{bmatrix} + u\begin{bmatrix}1\\0\end{bmatrix} + v\begin{bmatrix}0\\1\end{bmatrix} = \begin{bmatrix}4\\6\end{bmatrix}$$

Linear Programming (11/27)

Αν το σύστημα που προκύπτει έχει μια μοναδική λύση, αυτή ονομάζεται **βασική λύση (basic solution)**

Οι συντεταγμένες (στο δισδιάστατο χώρο) που τίθενται ίσες με το 0 πριν τη λύση του συστήματος καλούνται **μη βασικές (non basic)**

Οι συντεταγμένες (στο δισδιάστατο χώρο) που αποτιμούνται με τη λύση του συστήματος καλούνται **βασικές (basic)**

Μια βάση στο δισδιάστατο χώρο αποτελείται από δύο διανύσματα τα οποία είναι ανάλογα μεταξύ τους

Όταν επιλεγεί η βάση, κάθε διάνυσμα μπορεί να εκφραστεί ως ένα άθροισμα πολλαπλασίων των διανυσμάτων βάσης

Linear Programming (12/27)

Οι βασικές και μη βασικές συντεταγμένες δείχνουν ποια από τα δοσμένα διανύσματα περιλαμβάνονται ή αποκλείονται από την επιλογή μιας βάσης

Αν όλες οι συντεταγμένες μιας βασικής λύσης είναι μη αρνητικές, αυτή καλείται **βασική εφικτή λύση (basic feasible solution)**

Για παράδειγμα αν θέσουμε τις μεταβλητές x και y ίσες με 0 και λύσουμε ως προς τις u, υ παίρνουμε

(0, 0, 4, 6)

$$
\begin{aligned}
\text{maximize} \quad & 3x + 5y + 0u + 0v \\
\text{subject to} \quad & x + y + u = 4 \\
& x + 3y + v = 6 \\
& x, y, u, v \geq 0
\end{aligned}
$$

Linear Programming (13/27)

Αν θέσουμε τις μεταβλητές x και u ίσες με 0 και λύσουμε ως προς τις y, υ παίρνουμε

(0, 4, 0, -6)

η οποία δεν είναι βασική εφικτή λύση

Οι βασικές εφικτές λύσεις έχουν αντιστοιχία ένα-προς-ένα με τα οριακά σημεία της περιοχής εφικτών λύσεων

$$
\begin{aligned}
\text{maximize} \quad & 3x + 5y + 0u + 0v \\
\text{subject to} \quad & x + y + u = 4 \\
& x + 3y + v = 6 \\
& x, y, u, v \geq 0
\end{aligned}
$$

Linear Programming (14/27)

Η μέθοδος simplex προχωρά μέσα σε μια σειρά γειτονικών οριακών σημείων (βασικές εφικτές λύσεις) με αυξανόμενες τιμές της συνάρτησης που προσπαθούμε να μεγιστοποιήσουμε

Κάθε ένα από αυτά τα σημεία μπορεί να αναπαρασταθεί από ένα **simplex tableau**

Ένα simplex tableau είναι ένας πίνακας που αποθηκεύει πληροφορίες για τις εφικτές λύσεις που αντιστοιχούν στα οριακά σημεία του χώρου των λύσεων

Linear Programming (15/27)

Για παράδειγμα, για τη λύση (0, 0, 4, 6) του προηγούμενου συστήματος ο simplex tableau έχει ως εξής:

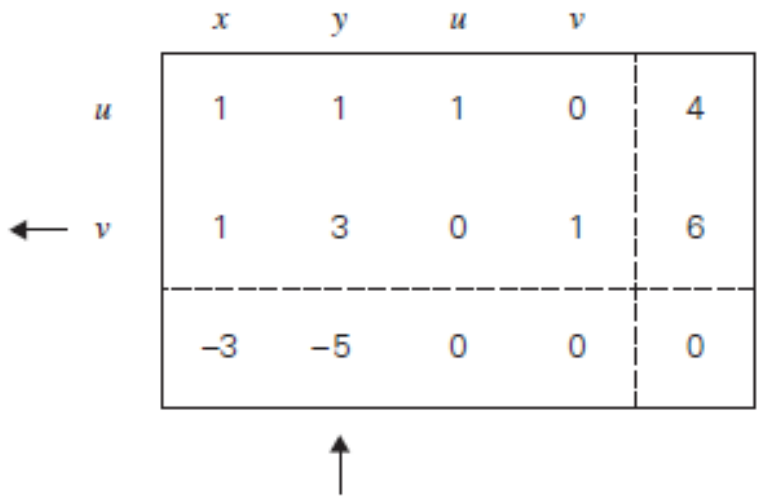
	x	y	u	v	
u	1	1	1	0	4
← v	1	3	0	1	6
	−3	−5	0	0	0
	↑				

$$
\begin{aligned}
\text{maximize} \quad & 3x + 5y + 0u + 0v \\
\text{subject to} \quad & x + y + u = 4 \\
& x + 3y + v = 6 \\
& x, y, u, v \geq 0
\end{aligned}
$$

Linear Programming (16/27)

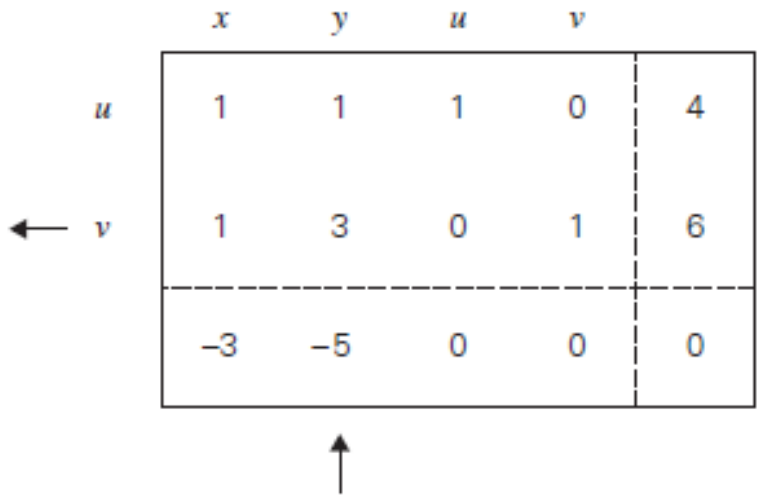
Ένας simplex tableau έχει τα ακόλουθα χαρακτηριστικά:

- Περιλαμβάνει m+1 γραμμές και n+1 στήλες
- Κάθε γραμμή αντιστοιχεί σε περιορισμούς και οι στήλες αντιστοιχούν στις μεταβλητές
- Κάθε γραμμή περιλαμβάνει τα coefficients του κάθε περιορισμού
- Η τελευταία στήλη δείχνει το δεξιό μέρος των περιορισμών
- Οι γραμμές έχουν ετικέτες των βασικών μεταβλητών της εφικτής λύσης



Linear Programming (17/27)

- Η τελευταία γραμμή ονομάζεται **objective row**
- Αρχικοποιείται με τα coefficients της συνάρτησης με αντίστροφο πρόσημο (για τις πρώτες n στήλες) και την τιμή της συνάρτησης στην τελευταία στήλη
- Σε επόμενες επαναλήψεις η objective row μετασχηματίζεται με τον ίδιο τρόπο όπως και οι υπόλοιπες γραμμές
- Η συγκεκριμένη γραμμή χρησιμοποιείται από τη μέθοδο για να διαπιστώσει αν η τρέχουσα λύση είναι η βέλτιστη

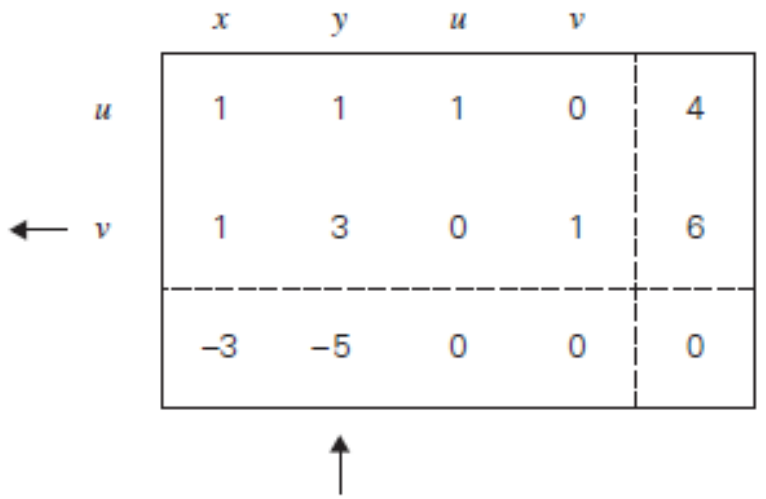


	x	y	u	v	
u	1	1	1	0	4
← v	1	3	0	1	6
	–3	–5	0	0	0

↑

Linear Programming (18/27)

- Η τρέχουσα λύση είναι μια βέλτιστη λύση αν όλες οι τιμές της εκτός ίσως από την τελευταία στήλη είναι μη αρνητικές
- Αν δεν ισχύει αυτό, τότε κάθε αρνητική τιμή αντιπροσωπεύει μια μη βασική μεταβλητή η οποία πρόκειται να γίνει βασική στο επόμενο tableau
- Η εφικτή λύση (0, 0, 4, 6) του πίνακα στο παράδειγμα, δεν είναι βέλτιστη
- Η αρνητική τιμή στη στήλη του x σημαίνει πως μπορούμε να αυξήσουμε την τιμή της συνάρτησης, αυξάνοντας την τιμή της συντεταγμένης x



	x	y	u	v	
u	1	1	1	0	4
← v	1	3	0	1	6
	−3	−5	0	0	0

↑

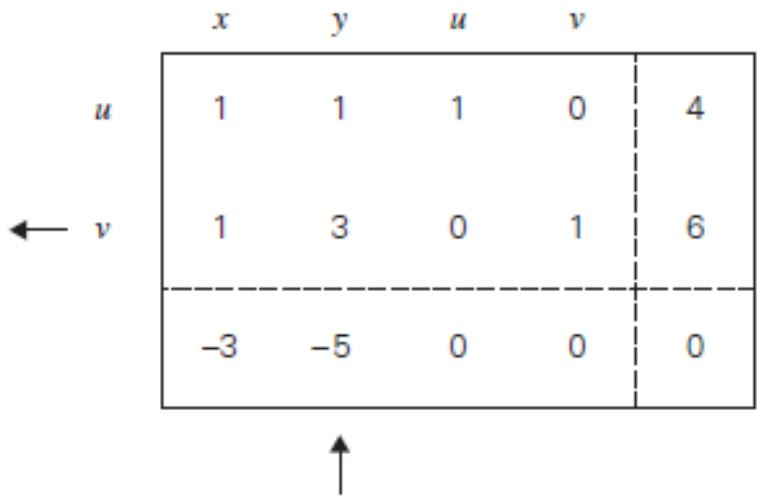
Linear Programming (19/27)

- Αφού ο συντελεστής του x στη συνάρτηση είναι θετικός, όσο μεγαλύτερο είναι το x τόσο θα αυξάνει η τιμή της συνάρτησης
- Η αύξηση του x θα εξαρτηθεί και από τις τιμές των u, υ έτσι ώστε το νέο σημείο της συνάρτησης να είναι εφικτό
- Πρέπει να ικανοποιούνται οι ακόλουθες συνθήκες

$$x + u = 4 \quad u \geq 0$$
$$x + v = 6 \quad v \geq 0$$

- άρα

$$x \leq \min\{4, 6\} = 4$$



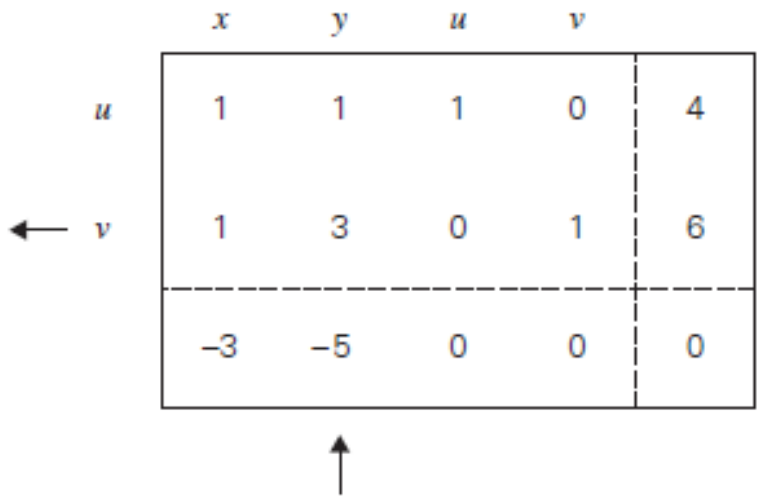
Linear Programming (20/27)

- Αν αυξήσουμε το x από το 0 στο 4, βρίσκουμε το σημείο (4, 0, 0, 2) που είναι γειτονικό στο (0, 0, 4, 6) με την τιμή της συνάρτησης να είναι ίση με 12
- Όμοια επεξεργασία ακολουθούμε και για την αρνητική τιμή του y
- Η αύξηση της τιμής του y απαιτεί

$$y + u = 4 \quad u \geq 0$$
$$3y + v = 6 \quad v \geq 0$$

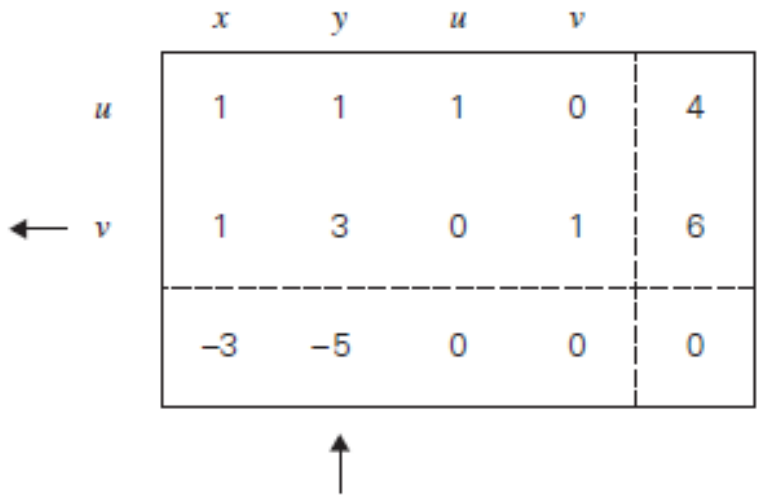
- άρα

$$y \leq \min\{\frac{4}{1}, \frac{6}{3}\} = 2$$



Linear Programming (21/27)

- Συνεπώς μπορούμε να αυξήσουμε το y κατά 2 και θα βρούμε τη λύση (0, 2, 2, 0) που είναι ακόμη ένα γειτονικό σημείο στο (0, 0, 4, 6) με τη συνάρτηση να παίρνει την τιμή 10
- Αν υπάρχουν αρκετές αρνητικές τιμές στην objective row, εφαρμόζεται ο ίδιος κανόνας με πρώτη επιλογή το μεγαλύτερο, σε απόλυτη τιμή, αρνητικό αριθμό
- Η τακτική αυτή στοχεύει στη μεταβλητή που έχει την μεγαλύτερη πιθανότητα για μεγαλύτερη αύξηση



	x	y	u	v	
u	1	1	1	0	4
← v	1	3	0	1	6
	−3	−5	0	0	0

↑

Linear Programming (22/27)

Οι περιορισμοί θέτουν διαφορετικά κριτήρια ως προς το μέγεθος αύξησης κάθε μεταβλητής

Μια νέα βασική μεταβλητή καλείται **μεταβλητή εισόδου (entering variable)** ενώ η στήλη της ονομάζεται **pivot column** (παίρνουμε τη μεταβλητή που προκαλεί τη μεγαλύτερη αύξηση, π.χ., y)

Ας δούμε όμως πως επιλέγουμε μια **departing variable** (μια βασική μεταβλητή για να γίνει μη βασική)

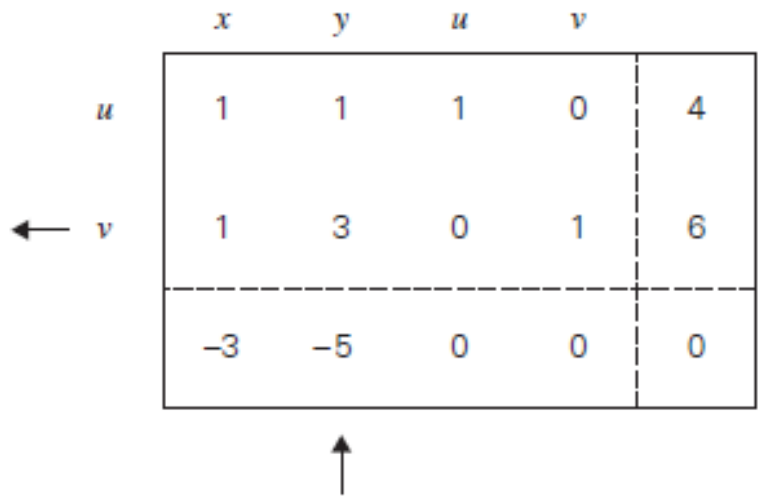
Βασιζόμαστε στην παρατήρηση για να πάρουμε ένα γειτονικό σημείο στις λύσεις που οδηγεί σε μεγαλύτερες τιμές της συνάρτησης, πρέπει να αυξήσουμε την μεταβλητή εισόδου κατά τη μέγιστη δυνατή ποσότητα

Linear Programming (23/27)

Με αυτό τον τρόπο θα μπορέσουμε να θέσουμε μια από τις παλιές μεταβλητές ίσες με 0 διατηρώντας το κριτήριο της ύπαρξης μη αρνητικών τιμών για τις όλες υπόλοιπες

Προκύπτει ο ακόλουθος κανόνας για την επιλογή της **departing variable**

- Για κάθε θετική τιμή στη pivot column, υπολογίζουμε το **θ-ratio** διαιρώντας την τελευταία τιμή της γραμμής με την τιμή στην pivot column. Στο παράδειγμα έχουμε:



	x	y	u	v	
u	1	1	1	0	4
← v	1	3	0	1	6
	−3	−5	0	0	0

↑

$$\theta_u = \frac{4}{1} = 4, \qquad \theta_v = \frac{6}{3} = 2$$

Linear Programming (24/27)

- Η γραμμή **με το μικρότερο θ-ratio** καθορίζει την departing variable, δηλαδή τη μεταβλητή που θα γίνει μη βασική
- Στο παράδειγμα μας είναι η μεταβλητή **υ**
- Αν δεν υπάρχει κάποια θετική τιμή στην pivot column, δεν μπορούμε να υπολογίσουμε το θ-ratio που σημαίνει ότι το πρόβλημα δεν φράσσεται και ο αλγόριθμος σταματά
- Στη συνέχεια μαρκάρουμε τη γραμμή της μεταβλητής που είναι η pivot row

	x	y	u	v	
u	1	1	1	0	4
← v	1	3	0	1	6
	−3	−5	0	0	0
		↑			

$$\theta_u = \frac{4}{1} = 4, \qquad \theta_v = \frac{6}{3} = 2$$

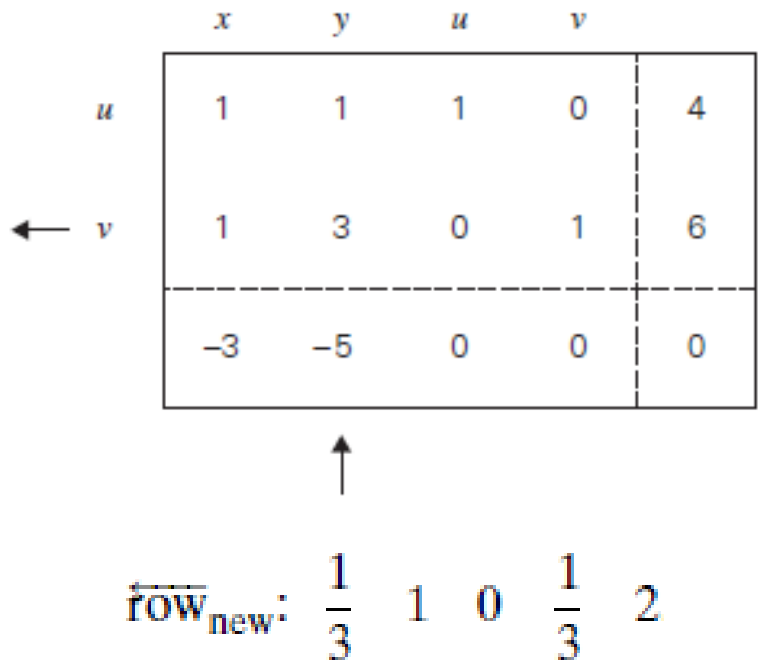
Linear Programming (25/27)

Τα επόμενα βήματα αφορούν για το μετασχηματισμό του τρέχοντος tableau στο νέο

Αρχικά διαιρούμε όλες τις τιμές της pivot row με την τιμή pivot (την τιμή της pivot column) και παίρνουμε νέες τιμές για τη γραμμή

Έπειτα αντικαθιστούμε κάθε μια από τις άλλες γραμμές, μαζί με την objective row, με τη διαφορά $\text{row} - c \cdot \overline{\text{row}}_{\text{new}}$

όπου c είναι η τιμή της γραμμής στην pivot column



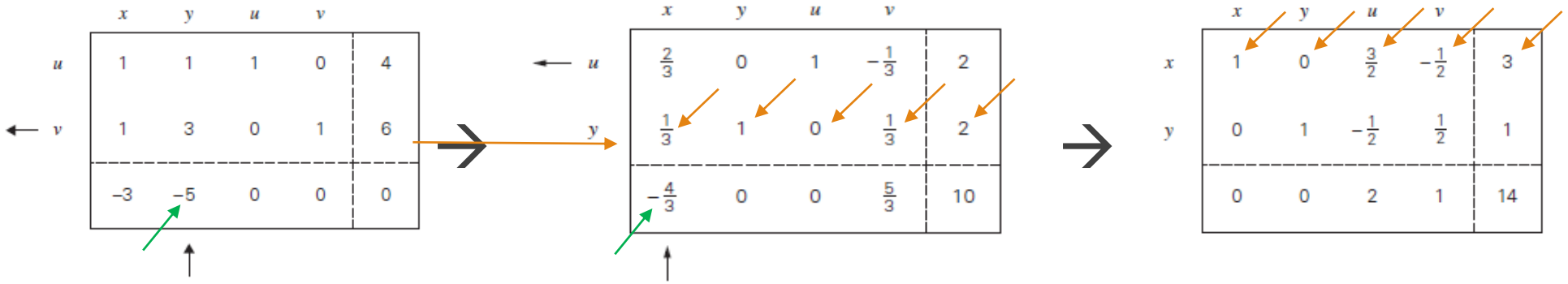
Linear Programming (26/27)

Για το παράδειγμά μας έχουμε $\text{row} - c \cdot \overleftarrow{\text{row}}_{\text{new}}$

$$\text{row } 1 - 1 \cdot \overleftarrow{\text{row}}_{\text{new}}: \quad \frac{2}{3} \quad 0 \quad 1 \quad -\frac{1}{3} \quad 2$$

$$\text{row } 3 - (-5) \cdot \overleftarrow{\text{row}}_{\text{new}}: \quad -\frac{4}{3} \quad 0 \quad 0 \quad \frac{5}{3} \quad 10$$

Και ο πίνακας μετασχηματίζεται ως εξής:



[-3-(-5)1/3 -5-(-5)1 0-(-5)0 0-(-5)1/3] 0-(-5)2

[-4/3-(-4/3)1 0-(-4/3)0 0-(-4/3)3/2 5/3-(-4/3))-1/2)] 10-(-4/3)3

Linear Programming (27/27)

Αλγόριθμος

Step 0 *Initialization* Present a given linear programming problem in standard form and set up an initial tableau with nonnegative entries in the rightmost column and m other columns composing the $m \times m$ identity matrix. (Entries in the objective row are to be disregarded in verifying these requirements.) These m columns define the basic variables of the initial basic feasible solution, used as the labels of the tableau's rows.

Step 1 *Optimality test* If all the entries in the objective row (except, possibly, the one in the rightmost column, which represents the value of the objective function) are nonnegative—stop: the tableau represents an optimal solution whose basic variables' values are in the rightmost column and the remaining, nonbasic variables' values are zeros.

Step 2 *Finding the entering variable* Select a negative entry from among the first n elements of the objective row. (A commonly used rule is to select the negative entry with the largest absolute value, with ties broken arbitrarily.) Mark its column to indicate the entering variable and the pivot column.

Step 3 *Finding the departing variable* For each positive entry in the pivot column, calculate the θ-ratio by dividing that row's entry in the rightmost column by its entry in the pivot column. (If all the entries in the pivot column are negative or zero, the problem is unbounded—stop.) Find the row with the smallest θ-ratio (ties may be broken arbitrarily), and mark this row to indicate the departing variable and the pivot row.

Step 4 *Forming the next tableau* Divide all the entries in the pivot row by its entry in the pivot column. Subtract from each of the other rows, including the objective row, the new pivot row multiplied by the entry in the pivot column of the row in question. (This will make all the entries in the pivot column 0's except for 1 in the pivot row.) Replace the label of the pivot row by the variable's name of the pivot column and go back to Step 1.

Linear Programming - Examples

http://www.phpsimplex.com/en/simplex_method_example.htm

https://math.libretexts.org/Bookshelves/Applied_Mathematics/Applied_Finite_Mathematics_(Sekhon_and_Bloom)/04%3A_Linear_Programming_The_Simplex_Method/4.02%3A_Maximization_By_The_Simplex_Method

http://www.universalteacherpublications.com/univ/ebooks/or/Ch3/simplex.htm