ΕΠΛ 232: Αλγόριθμοι και Πολυπλοκότητα

Κατ'οίκον Εργασία 1 – Σκελετοί Λύσεων

1. (α) Έστω δρομολόγηση $\langle e_1, e_2, \ldots, e_n \rangle$ των εργασιών $\langle 1, 2, \ldots, n \rangle$. Τότε οι χρόνοι συμπλήρωσης των εργασιών είναι

$e_1 \quad \Rightarrow \quad d_{e_1}$

$e_2 \quad \Rightarrow \quad d_{e_1} + d_{e_2}$

$e_3 \quad \Rightarrow \quad d_{e_1} + d_{e_2} + d_{e_3}$

…

$e_n \quad \Rightarrow \quad d_{e_1} + d_{e_2} + d_{e_3} + \ldots + d_{e_n}$

Συνεπώς, για τη συγκεκριμένη δρομολόγηση έχουμε

$$\mathrm{E} = \frac{1}{n}\left(nd_{e_1} + (n-1)d_{e_2} + (n-3)d_{e_3} + \ldots + d_{e_n}\right)$$

(i) Το κριτήριο αυτό είναι κατάλληλο για τη δημιουργία άπληστου αλγορίθμου. Θα το αποδείξουμε με τη μέθοδο της επαγωγής:

Έστω δρομολόγηση $\Delta = \langle e_1, e_2, \ldots, e_n \rangle$ μια βέλτιστη λύση στο πρόβλημα, δηλαδή, μια δρομολόγηση που ελαχιστοποιεί το μέσο χρόνο συμπλήρωσης των εργασιών $\langle 1, 2, \ldots, n \rangle$. Έστω f η εργασία με το μικρότερο χρόνο εκτέλεσης. Αν $f = e_1$, τότε η λύση περιέχει τη βέλτιστη επιλογή. Διαφορετικά, έστω Δ' η δρομολόγηση όπου η εργασίες f και e_1 ανταλλάσσουν θέσεις, δηλαδή

αν $\Delta = \langle e_1, e_2, \ldots, f, \ldots, e_n \rangle$ τότε $\Delta' = \langle f, e_2, \ldots, e_1, \ldots, e_n \rangle$.

Έστω E_Δ και $E_{\Delta'}$ οι μέσοι χρόνοι συμπλήρωσης των Δ και Δ', αντίστοιχα. Τότε

$$\begin{aligned}
E_\Delta - E_{\Delta'} &= \frac{1}{n}\left(nd_{e_1} + (n-1)d_{e_2} + (n-3)d_{e_3} + \ldots + (n-i)d_f + \ldots + d_{e_n}\right) \\
&\quad - \frac{1}{n}\left(nd_f + (n-1)d_{e_2} + (n-3)d_{e_3} + \ldots + (n-i)d_{e_1} + \ldots + d_{e_n}\right) \\
&= \frac{1}{n}\left(nd_{e_1} - nd_f + (n-i)(d_f - d_{e_1})\right) \\
&= \frac{i}{n}\left(d_{e_1} - d_f\right) \geq 0
\end{aligned}$$

Το τελευταίο βήμα έπεται από την υπόθεση ότι η εργασία f έχει το μικρότερο χρόνο εκτέλεσης από όλες τις εργασίες, δηλαδή, $d_{e_1} \geq d_f$. Επομένως, αφού, $E_\Delta - E_{\Delta'} \geq 0$, τότε $E_\Delta \geq E_{\Delta'}$. Επιπλέον, αφού γνωρίζουμε ότι η Δ είναι μια βέλτιστη λύση στο πρόβλημα, τότε $E_\Delta = E_{\Delta'}$ και συνεπώς η Δ' είναι μια βέλτιστη δρομολόγηση που ξεκινά με την άπληστη επιλογή.

Μπορούμε να χρησιμοποιήσουμε τα ίδια επιχειρήματα και για την εργασία με το δεύτερο μικρότερο χρόνο εκτέλεσης, και ούτω καθεξής (επαγωγή).

(ii) Το κριτήριο αυτό είναι ακατάλληλο για το πρόβλημα. Αντιπαράδειγμα: $n = 2$, $d_1 = 3$ και $d_2 = 5$. Επιλέγοντας σύμφωνα με το κριτήριο, παίρνουμε τη δρομολόγηση $\langle 2,1 \rangle$, με $c_1 = 8$, $c_2 = 5$ και $E = 13/2$. Υπάρχει όμως δρομολόγηση με μικρότερο μέσο χρόνο συμπλήρωσης: η δρομολόγηση $\langle 1,2 \rangle$ με $c_1 = 3$, $c_2 = 8$ και $E=11/2$.

(β) Σε αυτή την παραλλαγή του προβλήματος, υποθέτουμε ότι οι εργασίες δεν είναι διαθέσιμες από την αρχή αλλά αποδεσμεύονται σε κάποιους προκαθορισμένους χρόνους. Το κριτήριο απληστίας στην προκειμένη περίπτωση μεταβάλλεται ως εξής:

Ανά πάσα στιγμή επέλεξε την εργασία με το μικρότερο υπολειπόμενο χρόνο από αυτές που είναι διαθέσιμες.

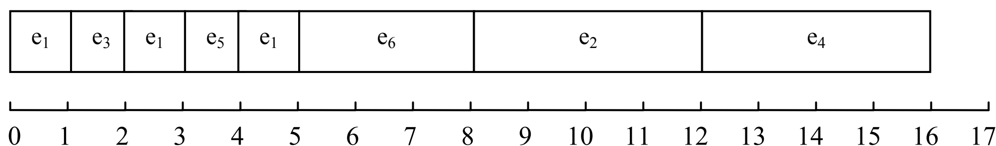
Αν κατά την εκτέλεση κάποιας εργασία αποδεσμευτεί άλλη εργασία με μικρότερο χρόνο εκτέλεσης τότε διέκοψε την εργασία που τρέχει και εκτέλεσε τη νέα εργασία με τον ελάχιστο χρόνο εκτέλεσης.

Έστω $r_1, r_2, \ldots, r_k$, οι χρόνοι αποδέσμευσης εργασιών σε αύξουσα σειρά, και έστω $\Sigma_1, \Sigma_2, \ldots, \Sigma_k$, τα σύνολα εργασιών που αποδεσμεύονται σε κάθε ένα από αυτούς τους χρόνους, αντίστοιχα. Υποθέτουμε ότι κάθε ένα από τα σύνολα περιέχει τις εργασίες ταξινομημένες σε αύξουσα σειρά ως προς το χρόνο εκτέλεσής τους.

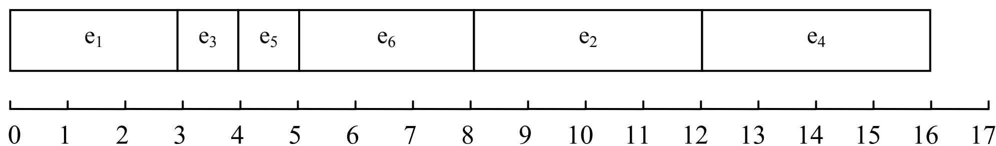
Τότε, δυνατό στιγμιότυπο του προβλήματος είναι το

$r_1 = 0$, $\Sigma_1 = \{(e_1, 3), (e_2, 4)\}$
$r_2 = 1$, $\Sigma_2 = \{(e_3, 1), (e_4, 4)\}$
$r_3 = 3$, $\Sigma_3 = \{(e_5, 1), (e_6, 3)\}$

όπου ο αλγόριθμος θα έχει το πιο κάτω αποτέλεσμα:



Παρατηρείστε ότι η δρομολόγηση αυτή έχει μικρότερο μέσο χρόνο συμπλήρωσης από τη πιο κάτω δρομολόγηση (όπου δεν εφαρμόζεται διακοπή εργασιών):



Ακολουθεί ο αλγόριθμος σε ψευδοκώδικα. Υποθέτει την ύπαρξη δύο δομών δεδομένων:

- SStack – Ταξινομημένη στοίβα: Λίστα ζευγών, ταξινομημένη σε αύξουσα σειρά ως προς το δεύτερο στοιχείο, που υποστηρίζεται από τις πράξεις: IsEmpty(S), (έλεγχος αν η S είναι κενή), Pop(S) (επιστροφή του κόμβου κορυφής), Push(S, x) (εισαγωγή του x στην κορυφή της S), Merge(S, T) (συγχώνευση των S και T)

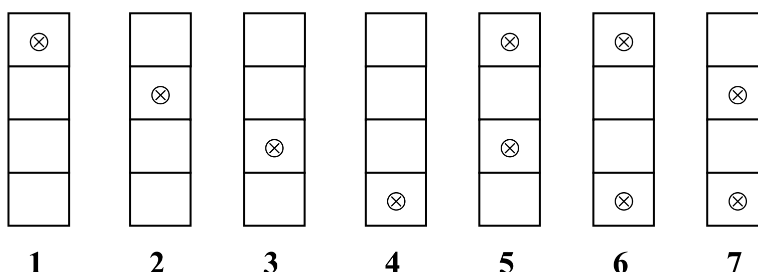
- Queue – Ουρά: Λίστα ζευγών που υποστηρίζεται από τις πράξεις Enqueue(Q,x) (εισαγωγή του x στο τέλος της Q) και Append(Q, S) (εισαγωγή των στοιχείων της S μετά από τα στοιχεία της Q).

```
int i = 1;
int time = r_i;
SStack jobs = Σ_i;
Queue schedule = ⟨⟩;

while (!IsEmpty(jobs) OR i<=m)
   if (i == m)
      Append(schedule, jobs);
   else
      if IsEmpty(jobs)
           i++;
           time = r_i;
           jobs = Σ_i;
      else
           (e,d) = Pop(jobs);
           if ( d + time < r_i+1)
                Enqueue(schedule, (e,d));
                time = time + d;
           else
             if (d  >= r_i+1 - time )
                Enqueue(schedule, (e, r_i+1 - time);
             if (d  > r_i+1 - time)
                Push(jobs, (e, d - (r_i+1 - time));
                Merge(jobs, Σ_i+1);
                i++;
                time = r_i;
}
```

Ο χρόνος εκτέλεσης της διαδικασίας είναι O(n·XE(Merge)) και επομένως εξαρτάται από την υλοποίηση της δομής SStack. Αν η υλοποίηση αυτή γίνει με γραμμική δομή δεδομένων ο χρόνος εκτέλεσης της Merge (XE(Merge)) είναι της τάξης O(n) διαφορετικά, με χρήση δενδρικής δομής, είναι της τάξεως O(log n). Η απόδειξη της ορθότητας αφήνεται ως άσκηση για τον αναγνώστη.

2. (α) Οι νόμιμες τοποθετήσεις είναι οι εξής:



(β) Αναδρομικός ορισμός της βέλτιστης λύσης.

Έστω C(i,k) η μέγιστη τιμή τοποθέτησης από τη στήλη 1 μέχρι τη στήλη i που τελειώνει με στήλη στην τοποθέτηση υπ'αριθμό k. Τότε το C(i,k) δίνεται αναδρομικά ως εξής:

$$C(i,k) = \begin{cases} 0 & \text{if } \quad i=0 \\ gain(1,i) + \max_{m\in\{2,3,4,7\}} C(i-1,m), & \text{if } \quad k=1, i>0 \\ gain(2,i) + \max_{m\in\{1,3,4,5,6\}} C(i-1,m), & \text{if } \quad k=2, i>0 \\ gain(3,i) + \max_{m\in\{1,2,4,6,7\}} C(i-1,m), & \text{if } \quad k=3, i>0 \\ gain(4,i) + \max_{m\in\{1,2,3,5\}} C(i-1,m), & \text{if } \quad k=4, i>0 \\ gain(1,i) + gain(3,i) + \max_{m\in\{2,4,7\}} C(i-1,m), & \text{if } \quad k=5, i>0 \\ gain(1,i) + gain(4,i) + \max_{m\in\{2,3\}} C(i-1,m), & \text{if } \quad k=6, i>0 \\ gain(2,i) + gain(4,i) + \max_{m\in\{1,3,5\}} C(i-1,m), & \text{if } \quad k=7, i>0 \end{cases}$$

Από αυτή την αναδρομική σχέση παρατηρούμε ότι για υπολογισμό κάθε C(i,k) απαιτείται γνώση κάποιων C(i-1,k). Η βασική περίπτωση δίνεται από το C(0,k) και το ζητούμενο είναι το max(C(n,1), C(n,2), …, C(n,7)). Επομένως χρησιμοποιούμε ένα πίνακα C διαστάσεων n×7 και υπολογίζουμε τις θέσεις του ξεκινώντας με τα C[0,k] και προχωρώντας προς τα C[1, k], C[2,k], …, και C[n,k].

Σε ψευδοκώδικα ο αλγόριθμος έχει ως εξής:

```
maxgain (int gain[4,n])
  for (k=1; k<=7, k++)
     C[0,k]= 0;

  for (i = 1; i <= n; i++){
         max = C[i-1,2];
         if (max < C[i-1,3])
                  max = C[i-1,3];
         if (max < C[i-1,4])
                  max = C[i-1,4];
         if (max < C[i-1,7])
                  max = C[i-1,7];
         C[i,1] = max + gain[i,1];

         max = C[i-1,1];
         if (max < C[i-1,3])
                  max = C[i-1,3];
         if (max < C[i-1,4])
                  max = C[i-1,4];
         if (max < C[i-1,5])
                  max = C[i-1,5];
         if (max < C[i-1,6])
                  max = C[i-1,6];
```

```
            C[i,2] = max + gain[i,2];

            …
            C[i,3] = max + gain[i,3];

            …
            C[i,4] = max + gain[i,4];

            …
            C[i,5] = max + gain[i,1] + gain[i,3];

            …
            C[i,6] = max + gain[i,1] + gain[i,4];

            …
            C[i,7] = max + gain[i,2] + gain[i,4];
  }
  max = C[n,1]);
  for (i= 2; i<=7; i++)
       if (C[n,i]> max)    max = C[n,i];
  return max;
  }
```

Ο χρόνος εκτέλεσης του αλγόριθμου είναι O(n·7·c) = O(n).

(δ) Για να επιστρέψουμε την ακριβή τοποθέτηση με τη μέγιστη τιμή, θα πρέπει να επεκτείνουμε τον αλγόριθμο έτσι ώστε να διατηρεί πληροφορίες για τη δομή της βέλτιστης λύσης κάθε υποπροβλήματος. Συγκεκριμένα, εκτός από τον πίνακα C(i,k) θα χρησιμοποιήσουμε ένα δεύτερο πίνακα Last(i,k) όπου θα φυλάγουμε τον αριθμό της τοποθέτησης της προτελευταίας στήλης στη βέλτιστη τοποθέτηση από τη στήλη 1 μέχρι τη στήλη i που τελειώνει με στήλη στην τοποθέτηση με αριθμό k.

Ο καινούριος αλγόριθμος φαίνεται πιο κάτω όπου οι προσθήκες εμφανίζονται με σκούρα γράμματα.

```
  maxgain (int gain[4,n])
    for (k=1; k<=7, k++)
       C[0,k]= 0; D[0,k] = 0;

    for (i = 1; i <= n; i++){
            max = C[i-1,2]; D[i,1] = 2;
            if (max < C[i-1,3])
                      max = C[i-1,3]; D[i,1] = 3;
            if (max < C[i-1,4])
                      max = C[i-1,4]; D[i,1] = 4;
            if (max < C[i-1,7])
                      max = C[i-1,7]; D[i,1] = 7;
            C[i,1] = max + gain[i,1];

            max = C[i-1,1]; D[i,2] = 1;
            if (max < C[i-1,3])
```

```
                max = C[i-1,3]; D[i,1] = 3;
          if (max < C[i-1,4])
                max = C[i-1,4]; D[i,1] = 4;
          if (max < C[i-1,5])
                max = C[i-1,5]; D[i,1] = 5;
          if (max < C[i-1,6])
                max = C[i-1,6]; D[i,1] = 6;
          C[i,2] = max + gain[i,2];

          …
          C[i,3] = max + gain[i,3];

          …
          C[i,4] = max + gain[i,4];

          …
}
max = C[n,1]); last = 1;
for (i= 2; i<=7; i++)
     if (C[n,i]> max)
          max = C[n,i];
          last = i;

printf(last);

while (i > 0)
     last= D[i,last];
     printf(last);
     i--;

return max;
}
```

Τυπώνει ανάποδα τη βέλτιστη τοποθέτηση.

3. (α) Από τον ορισμός του γινομένου δύο πολυωνύμων, έχουμε ότι

$$C(x) = c_0 + c_1x + c_2x^2 + c_3x^3 + c_4x^4$$

όπου

$$c_0 = a_0 \cdot b_0$$
$$c_1 = a_0 \cdot b_1 + a_0 \cdot b_1$$
$$c_2 = a_0 \cdot b_2 + a_1 \cdot b_1 + a_0 \cdot b_2$$
$$c_3 = a_1 \cdot b_2 + a_1 \cdot b_2$$
$$c_4 = a_2 \cdot b_2$$

Κατά συνέπεια, για υπολογισμό του γινομένου απαιτούνται 9 πολλαπλασιασμοί και 4 προσθέσεις.

(β) Χρησιμοποιώντας τα γινόμενα P_1, …, P_6, το πρόβλημα ανάγεται στους πιο κάτω υπολογισμούς:

$$c_0 = P_1$$
$$c_1 = P_4 - P_1 - P_2$$
$$c_2 = P_6 - P_1 - P_3 + P_2$$

$c_3 = P_5 – P_2 – P_3$
$c_4 = P_3$

Αυτή η μέθοδος προφανώς απαιτεί τη χρήστη των 6 γινομένων και 13 προσθαφαιρέσεων.

(γ) Ο αλγόριθμος διαίρει και βασίλευε για το πρόβλημα έχει ως εξής:

- Έστω τα πολυώνυμα $f(x) = a_0 + a_1x + \ldots + a_{n-1}x^{n-1}$, $g = b_0 + b_1x + \ldots + b_{n-1}x^{n-1}$.
- Μοίρασε τα πολυώνυμα σε τρία τρίτα:
  $f_0(x)= a_0+\ldots+a_{n/3-1}x^{n/3-1}$, $f_1(x)= a_{n/3}+\ldots+a_{2n/3-1}x^{n/3-1}$, και $f_2(x)= a_{2n/3}+\ldots+a_{n-1}x^{n/3-1}$,
  $g_0(x)= b_0+\ldots+b_{n/3-1}x^{n/3-1}$, $g_1(x)= b_{n/3}+\ldots+b_{2n/3-1}x^{n/3-1}$, και $g_2(x)= a_{2n/3}+\ldots+b_{n-1}x^{n/3-1}$,
- Υπολόγισε αναδρομικά τα γινόμενα

$$P_1 = f_0\cdot g_0$$
$$P_2 = f_1\cdot g_1$$
$$P_3 = f_2\cdot g_2$$
$$P_4 = (f_0 + f_1)\cdot(g_0 + g_1)$$
$$P_5 = (f_1 + f_2)\cdot(g_1 + g_2)$$
$$P_6 = (f_0 + f_2)\cdot(g_0 + g_2)$$

- Συνδύασε τα πιο πάνω γινόμενα για να λάβεις το γινόμενο πολυώνυμο ως εξής:

$$f(x)\cdot g(x) = P_1 + x^{n/3}(P_4 – P_1 – P_2) + x^{2n/3}(P_6 – P_1 – P_3 + P_2) + x^n(P_5 – P_2 – P_3) + x^{4n/3} P_3$$

Έστω T(n) ο χρόνος εκτέλεσης της διαδικασίας σε πολυώνυμα βαθμού n. Το T(n) δίνεται από την αναδρομική εξίσωση

$T(1) = 1$
$T(n) = 6\cdot T(n/3) + cn,$

Λύνοντας την αναδρομική εξίσωση με τη μέθοδο της αντικατάσταση παίρνουμε:

$$\begin{aligned} T(n) &= 6\cdot T(n/3) + n \\ &= 6^2\cdot T(n/3^2) + 2n + n \\ &= 6^3\cdot T(n/3^3) + 2^2n + 2n + n \\ &= 6^i\cdot T(n/3^i) + 2^{i-1}n + \ldots + 2^2n + 2n + n \end{aligned}$$

Θέτοντας $k = \log_3 n$…

$$\begin{aligned} T(n) &= 6^k\cdot T(n/3^k) + 2^{k-1}n + \ldots + 2^2n + 2n + n \\ &= 6^k \cdot T(n/n) + n(2^{k-1}+ \ldots + 2^2 + 2 + 1) \\ &= 6^k + n (2^k – 1) \\ &= 6^k + 3^k (2^k – 1) \\ &= 2\cdot 6^k – 3^k \end{aligned}$$

$= 2 \cdot 6^{\log_6 n / \log_6 3} - n$

$= 2 \cdot 6^{\log_6 n \cdot \log_3 6} - n$

$= 2 \cdot (6^{\log_6 n})^{\log_3 6} - n$

$=2n^{\log_3 6} - n$

$\in \Theta(n^{\log_3 6})$

(δ) Ο προτεινόμενος αλγόριθμος έχει χρόνο εκτέλεσης που δίνεται από την πιο κάτω αναδρομική εξίσωση:

$T(n) = m \cdot T(n/4) + cn$
$T(1) = 1$

Χρησιμοποιώντας το θεώρημα γενικής χρήσης συμπεραίνουμε ότι

- Αν $n \in O(n^{\log_4 m - \varepsilon})$, δηλαδή m>4, $T(n) \in O(n^{\log_4 m})$.
- Αν $n \in \Theta(n^{\log_4 m})$, δηλαδή m=4, $T(n) \in O(n^{\log_4 4} \lg n) = O(n \lg n)$.
- Αν $n \in \Omega(n^{\log_4 m + \varepsilon})$, δηλαδή m<4, $T(n) \in O(n)$

Επομένως, για m≤4, δηλαδή στις δύο τελευταίες πιο πάνω περιπτώσεις, ο αλγόριθμος είναι αποδοτικότερος από τον αλγόριθμο που προτείνεται στο (γ) (n log n, n $\in$ O($(n^{\log_3 6})$. Επίσης, το ζητούμενο ισχύει (πρώτη περίπτωση) όταν m>4 και
$n^{\log_4 m} < n^{\log_3 6}$
Δηλαδή

$$\log_4 m < \log_3 6 \Leftrightarrow 4^{\log_4 m} < 4^{\log_3 6} \Leftrightarrow m < 4^{\log_3 6} \Leftrightarrow m < 9.59$$

΄Αρα m=9 είναι ο μεγαλύτερος ακέραιος που ικανοποιεί το ζητούμενο.

4. Μοντελοποιούμε τη σκακιέρα ως ένα δισδιάστατο πίνακα Board[7,7] με αρχική κατάσταση:

Board[1,1] = Board[1,2] = Board[1,6] = Board[1,7] = INVALID
Board[2,1] = Board[2,2] = Board[2,6] = Board[2,7] = INVALID
Board[6,1] = Board[6,2] = Board[6,6] = Board[6,7] = INVALID
Board[7,1] = Board[7,2] = Board[7,6] = Board[7,7] = INVALID
Board[1,3] = Board[1,4] = Board[1,5] = FULL
Board[2,3] = Board[2,4] = Board[2,5] = FULL
Board[6,3] = Board[6,4] = Board[6,5] = FULL
Board[7,3] = Board[7,4] = Board[7,5] = FULL
Board[3,1] = Board[3,2] = … = Board[3,7] = FULL
Board[4,1] = Board[4,2] = … = Board[4,7] = FULL
Board[5,1] = Board[5,2] = … = Board[5,7] = FULL
Board[4,4] = EMPTY

Ορίζουμε μια κίνηση του παιχνιδιού ως μια τριάδα (i, j, m), $1 \le i, j \le 7$, $m \in \{U, D, R, L\}$, αν Board[i,j] = FULL, και επιπλέον ένα από τα ακόλουθα:

- m = U (up), Board[i – 1, j] = FULL, Board[i – 2, j] = EMPTY,

- m = D (down), Board[i + 1, j] = FULL, Board[i + 2, j] = EMPTY,
- m = R (right), Board[i, j + 1] = FULL, Board[i , j + 2] = EMPTY,
- m = L (left), Board[i , j – 1] = FULL, Board[i, j – 2] = EMPTY,

Ονομάζουμε μια ακολουθία από k νόμιμες κινήσεις του παιχνιδιού, $0 \le k \le 31$, k-υποσχόμενη αν

- $0 \le k < 31$, και εφαρμογή των k κινήσεων οδηγούν τη σκακιέρα σε κατάσταση όπου υπάρχουν δυνατές κινήσεις, ή
- k = 31 και εφαρμογή των k κινήσεων οδηγούν τη σκακιέρα σε κατάσταση με Board[4,4] = FULL.

Λύση στο πρόβλημα είναι ένα 31-υποσχόμενο διάνυσμα. Θεωρώντας ότι ξεκινούμε με το 0-υποσχόμενο διάνυσμα και με δεδομένο ένα i-υποσχόμενο διάνυσμα υπάρχουν τρεις περιπτώσεις

- είτε αυτό μπορεί να γίνει (i+1)-υποσχόμενο διάνυσμα με την προσθήκη μιας νέας κίνησης,
- είτε, αν i = 30, αυτό μπορεί να γίνει 31-υποσχόμενο διάνυσμα με την προσθήκη μιας τελευταίας κίνησης,
- είτε έχουμε φτάσει σε αδιέξοδο, δηλαδή σε μια τοποθέτηση στην οποία είτε δεν υπάρχουν δυνατές κινήσεις, είτε δεν έχουμε φθάσει στην επιθυμητή τελική κατάσταση. Σε τέτοια περίπτωση ο αλγόριθμος θα οπισθοχωρήσει, θα ματαιώσει την τελευταία κίνηση επαναφέροντας τη σκακιέρα στην προηγούμενη κατάσταση, και θα θεωρήσει μια άλλη κίνηση από τη συγκεκριμένη κατάσταση, αν υπάρχει.

Υποθέτουμε ότι έχουμε υλοποιημένες τις διαδικασίες DoMove(move, Board) η οποία εκτελεί την κίνηση move στη σκακιέρα Board και μας επιστρέφει το σύνολο των νέων κινήσεων που έχουν γίνει εφικτές μετά από την κίνηση, και UndoMove(move, Board) η οποία ματαιώνει την κίνηση move στη σκακιέρα Board. Ο αλγόριθμος έχει ως εξής:

```
board(){
   i = 0;
   game = ⟨⟩;
   moves[32]= [⟨⟩, … , ⟨⟩]
   moves[0] = ⟨(2,4,D),(4,2,R),(4,6,L),(6,4,U)⟩;
   while (i ≤ 32)
        if (i == 32 AND Board[4,4] != FULL)
             return game;
        else if (!IsEmpty(moves[i])
                  x = Pop (moves[i]);
                  moves[i+1] = DoMove(x,Board);
                  game = Push(game,move)
             else
                  UndoMove(Pop(game), Board);
                  i--;
}
```

5. Τα δύο προβλήματα μπορούν να λυθούν με αλγόριθμους δυναμικού προγραμματισμού, παραλλαγές του αλγόριθμου των Floyd-Warshall, και χρόνους

εκτέλεσης της τάξης $O(n^3)$. Πιο κάτω δίνονται οι αναδρομικοί ορισμοί των ζητούμενων βέλτιστων λύσεων. Η διατύπωση των αλγορίθμων αφήνεται ως άσκηση στον αναγνώστη.

(i) Συμβολισμός: Γράφουμε paths[i,j,m] για το μέγιστο αριθμό μονοπατιών από την κορυφή i στην κορυφή j με ενδιάμεσες κορυφές που ανήκουν στο σύνολο {1,2,…m}. Τότε

$$paths[i,j,0] = \begin{cases} 1, \text{ if } w(i,j) \neq \infty \\ 0, \text{ if } w(i,j) = \infty \end{cases}$$

$$paths[i,j,m] = paths[i,j,m-1] + paths[i,m,m-1] \cdot paths[m,j,m-1])$$

Παρατηρείστε πως για να ορίζεται η έννοια του "μέγιστου αριθμού μονοπατιών" ανάμεσα στους κόμβους ενός γράφου, ο γράφος πρέπει να μην περιέχει κύκλους.

(ii) Συμβολισμός: Γράφουμε maxpaths[i,j,m] για τον αριθμό των βραχύτερων μονοπατιών από την κορυφή i στην κορυφή j με ενδιάμεσες κορυφές που ανήκουν στο σύνολο {1,2,…m}. Τότε

$$maxpaths[i,j,0] = \begin{cases} 1, \text{ if } w(i,j) \neq \infty \\ 0, \text{ if } w(i,j) = \infty \end{cases}$$

$$maxpaths[i,j,m] = \begin{cases} maxpaths[i,j,m-1], \\ \qquad \text{if } c(i,j,m-1) < c(i,m,m-1) + c(m,j,m-1) \\ maxpaths[i,m,m-1] \cdot maxpaths[m,j,m-1], \\ \qquad \text{if } c(i,j,m-1) > c(i,m,m-1) + c(m,j,m-1) \\ maxpaths[i,j,m-1] + maxpaths[i,m,m-1] \cdot maxpaths[m,j,m-1], \\ \qquad \text{if } c(i,j,m-1) = c(i,m,m-1) + c(m,j,m-1) \end{cases}$$

όπου τα c[i,j,m] είναι τα βάρη βραχύτερων μονοπατιών όπως ορίζονται στον αλγόριθμο των Floy-Warshall.