
Задача 1: *Граф на Кирхоф* ще наричаме всеки ориентиран тегловен граф $G = (V, E, w)$, за който са изпълнени следните свойства:

1. Всяко ребро от E участва в поне един цикъл.
2. Цената на всеки цикъл в G е 0.

Предложете ефикасен алгоритъм, който при подаден граф на Кирхоф намира цената на най-дълъг (най-тежък) път в графа.

Решение: Нека G е граф на Кирхоф. Ще докажем няколко леми за G.

Лема 1. *Всяка от слабо свързаните компоненти в G е силно свързана.*

Доказателство: Да допуснем противното. Нека u и v са два върха от G, за които съществува път $(u = x_1, x_2, \ldots, x_{i-1}, x_i = v)$ от u до v, но не съществува път от v до u и нека $1 < k \leq i$ е най-малкият индекс, за който съществува път от v до x_k[1]. Да разгледаме реброто (x_{k-1}, x_k). От свойство 1 знаем, че то участва в цикъл. Тогава, премахвайки реброто (x_{k-1}, x_k) от цикъла, получаваме път от x_k до x_{k-1}. Ако слепим този път към пътя от v до x_k, ще получим път от v до x_{k-1}. От дефиницията на k обаче такъв път не съществува. Така получихме противоречие, тоест допускането ни е грешно и всяка от слабо свързаните компоненти на G е силно свързана.

□

Лема 2. *Ако u и v са в една свързана компонента на G, то цената на произволен път от u до v е обратна на цената на произволен път от v до u. Като следствие всички пътища между два върха в G имат еднаква цена, тоест най-дългият път между два върха е всъщност и най-късият.*

Доказателство: Нека p е произволен път от u до v и нека q е произволен път от v до u. Тогава, слепвайки двата пътя, получаваме цикъл pq. От Свойство 2 този цикъл има цена 0, тоест

$$0 = cost(pq) = cost(p) + cost(q)$$

Оттук $cost(p) = -cost(q)$.

□

Лема 3. *Нека u е произволен връх от G и нека s и t са върхове от свързаната компонента на u. Тогава:*

$$cost(s \rightsquigarrow t) = cost(u \rightsquigarrow t) - cost(u \rightsquigarrow s)$$

[1]тъй като сме допуснали, че не съществува път от v до $u = x_1$, то k трябва да е по-голямо от 1.

Доказателството: Нека p е произволен път от s до u, и нека q е произволен път от u до t (от Лема 1 такива пътища съществуват). Тогава pq е път от s до t и ще бъде изпълнено, че:

$$\begin{aligned} cost(s \rightsquigarrow t) &= cost(pq) \ // \text{ от Лема 2} \\ &= cost(p) + cost(q) \\ &= cost(s \rightsquigarrow u) + cost(u \rightsquigarrow t) \ // \text{ от Лема 2} \\ &= -cost(u \rightsquigarrow s) + cost(u \rightsquigarrow t) \ // \text{ от Лема 2} \\ &= cost(u \rightsquigarrow t) - cost(u \rightsquigarrow s) \end{aligned}$$

□

Да се върнем обратно към задачата. Нека помислим как можем да намерим цената на най-дълъг път в една свързана компонента. Ако u е произволен връх от компонентата, то тази цена може да се запише като

$$\max\{cost(s \rightsquigarrow t) \mid s, t \in \text{SCC}(u)\}$$

По Лема 3 последното може да се развие по следния начин:

$$\begin{aligned} &\max\{cost(s \rightsquigarrow t) \mid s, t \in \text{SCC}(u)\} \\ &= \max\{cost(u \rightsquigarrow t) - cost(u \rightsquigarrow s) \mid s, t \in \text{SCC}(u)\} \\ &= \max\{cost(u \rightsquigarrow t) \mid t \in \text{SCC}(u)\} - \min\{cost(u \rightsquigarrow s) \mid s \in \text{SCC}(u)\} \end{aligned}$$

Така, за да намерим цената на най-дълъг път в свързана компонента, е достатъчно да намерим цените на пътищата от един връх от компонентата до всички останали, след което да извадим минималната цена от максималната.

Следната модификация на BFS реализира точно тази идея:

```
FindLongestPath(G = (V, E, w): граф на Кирхоф)
 1  създай нулиран булев масив visited[1...n]
 2  създай празна опашка Q
 3  result ← −∞
 4  for u ∈ V
 5      if visited[u] = 1
 6          continue
 7      visited[u] ← 1
 8      max_cost ← 0
 9      min_cost ← 0
10      enqueue(Q, (u, 0))
11      while Q is not empty do
12          (x, c) ← dequeue(Q)
13          max_cost ← max(max_cost, c)
14          min_cost ← min(min_cost, c)
```

```
15            for y ∈ adj(x)
16                if visited[y] = 0
17                    visited[y] ← 1
18                    enqueue(Q, (y, c + w(x, y)))
19        result ← max(result, max_cost − min_cost)
20    return result
```

Сложността по време на FindLongestPath е $O(m + n)$, както при BFS.

Задача 2: Представете си, че играете следната игра. На маса са наредени n кутии, като върху всяка кутия има положително номерче. За един ход се случва следното:

- Избирате три от кутиите на масата.
- Трите кутии, заедно с кутиите между тях, биват премахнати от масата.
- Печелите толкова точки, колкото е произведението на номерчетата на трите кутии, които сте избрали.

Играта приключва, когато кутиите станат по-малко от 3. Предложете алгоритъм със сложност по време $O(n^3)$, който при подаден масив $A[1\,..\,n]$ от номерчетата на кутиите намира максималния брой точки, който можете да спечелите.

Решение: Задачата може да се реши по схемата **Динамично Програмиране**. Нека $\mathrm{dp}[i,j]$ е отговорът за входен масив $A[i\,..\,j]$. Искаме да построим $\mathrm{dp}[1,n]$.

Базата на рекурсивната декомпозиция е, когато $j-i<2$, тоест когато играта е с две или по-малко кутии. Тогава

$$\mathrm{dp}[i,j]=0$$

тъй като играта приключва веднага и не могат да бъдат спечелени никакви точки.

В случая, когато $j-i\geq 2$, тоест когато играта е с поне три кутии, има два варианта:

1. На някой ход взимаме първата и последната кутия и някоя кутия на позиция $i<k<j$. Тъй като това би премахнало всички кутии, този ход всъщност е последният. Освен това на никой предишен ход не може да сме избрали кутии разположени от различни страни на k-тата кутия, тъй като това би я премахнало. Така можем да разбием предишните ходове на ходове в интервала $[i+1\,..\,k-1]$ и ходове в интервала $[k+1\,..\,j-1]$. Тъй като максималните точки, които могат да бъдат спечелени от двата интервала са съответно $\mathrm{dp}[i+1,k-1]$ и $\mathrm{dp}[k+1,j]$, то максималният брой точки за целия интервал $[i\,..\,j]$ ще бъде:

$$\mathrm{dp}[i+1,k-1]+\mathrm{dp}[k+1,j-1]+A[i]*A[j]*A[k]$$

2. Никога не взимаме едновременно първата и последната кутия. Тогава съществува число $i\leq k<j$, за което можем да разбием ходовете, които сме изиграли на ходове в интервала $[i\,..\,k]$ и ходове в интервала $[k+1\,..\,j]$. Тъй като максималните точки, които могат да бъдат спечелени от двата интервала са съответно $\mathrm{dp}[i,k]$ и $\mathrm{dp}[k+1,j]$, то максималният брой точки за целия интервал $[i\,..\,j]$ ще бъде:

$$\mathrm{dp}[i,k]+\mathrm{dp}[k+1,j]$$

Естествено трябва да изберем най-оптималната от по-горните възможности. Така, когато $j-i\geq 2$, рекурсивната декомпозиция ще бъде:

$$\mathrm{dp}[i,j]=\max\left\{\begin{array}{l}\max\limits_{i<k<j}\big(\mathrm{dp}[i+1,k-1]+\mathrm{dp}[k+1,j-1]+A[i]\cdot A[j]\cdot A[k]\big),\\ \max\limits_{i\leq k<j}\big(\mathrm{dp}[i,k]+\mathrm{dp}[k+1,j]\big)\end{array}\right\}$$

Следният алгоритъм със сложност $O(n^3)$ реализира рекурсивната декомпозиция:

```
MaximumScore(A[1..n]: масив от цели числа)
 1  създай нулиран двумерен масив dp[1..n][1..n]
 2  for len ∈ 3 to n
 3      for i ∈ 1 to n − len + 1
 4          j ← i + len − 1
 5          best ← 0
 6          for k ∈ i + 1 to j − 1
 7              best ← max(best, dp[i+1][k−1] + dp[k+1][j−1] + A[i] * A[j] * A[k])
 8          for k ∈ i to j − 1
 9              best ← max(best, dp[i][k] + dp[k+1][j])
10          dp[i][j] ← best
11  return dp[1][n]
```

Задача 3: Дадено е кореново дърво $T = (V, E)$ и тегловна функция $w : V \to \mathbb{N}$ на върховете. За всеки връх u ще наричаме *u добър*, ако съществува по-тежък от u връх, който е извън поддървото на u. Предложете алгоритъм със сложност $O(n)$, който намира максималната тежест на добър връх в T.

Упътване: Пуснете обхождане в дълбочина от корена на дървото и разгледайте реда на откриване и финализиране на върховете.

Решение: Да си представим, че сме пуснали обхождане в дълбочина от корена на дървото. От лекции знаем, че такъв тип обхождане задава ред на откриване и финализиране на обходените върховете. Нека $d[u]$ е редът на откриване на връх u и нека $f[u]$ е редът на финализиране на връх u. За всяко $1 \leq i \leq n$ дефинираме dpref$[i]$ и fsuff[i] по следния начин:

$$\text{dpref}[i] \coloneqq \max\{w(v) \mid d[v] \leq i\}$$

$$\text{fsuff}[i] \coloneqq \max\{w(v) \mid f[v] \geq i\}$$

тоест dpref$[i]$ е максималната тежест на връх измежду първите i открити върха, а fsuff$[i]$ е максималната тежест на връх измежду последните $n - i + 1$ финализирани върха.

Да забележим, че масивите $d, f,$ dpref и fsuff могат да се построят по време на обхождането в дълбочина за време $O(m + n)$. В сила е следното наблюдение:

Наблюдение 1. *Един връх v е извън поддървото на u точно тогава, когато $d[v] < d[u]$ или $f[v] > f[u]$.*

Следващата лема е следствие от горното наблюдение. Накратко, това, което ни казва тя, е че за да проверим дали един връх е добър, е достатъчно да проверим дали максималната тежест на върховете, които са открити преди него или финализирани след него, е по-голяма от неговата собствена тежест.

Лема 4. *Нека u е връх от T. Тогава u е добър точно тогава, когато*

$$(\text{dpref}[d[u] - 1] > w(u)) \vee (\text{fsuff}[f[u] + 1] > w(u))$$

Доказателство: Като разпишем по дефинициите на dpref и fsuff, получаваме, че изразът от лемата е еквивалентен на

$$(\max\{w(v) \mid d[v] \leq d[u] - 1\} > w(u)) \vee (\max\{w(v) \mid f[v] \geq f[u] + 1\} > w(u))$$

Тъй като за всяко $1 \leq i \leq n$ е вярно, че $d[i] \in \mathbb{N}$, то последното е еквивалентно на

$$(\max\{w(v) \mid d[v] < d[u]\} > w(u)) \vee (\max\{w(v) \mid f[v] > f[u]\} > w(u))$$

което на свой ред е еквивалентно на

$$\max\{w(v) \mid d[v] < d[u] \vee f[v] > f[u]\} > w(u)$$

От Наблюдение 1 последното е същото като

$$\max\{w(v) \mid v \text{ е извън поддървото на } u\} > w(u)$$

Горното е изпълнено точно когато съществува по-тежък от u връх, който е извън поддървото на u, тоест точно когато u е добър връх.

□

Горната лема ни дава константна $\Theta(1)$ проверка дали един връх е добър. Така, за да намерим най-тежкия добър връх, можем просто да итерираме през всички върхове, да филтрираме добрите и да вземем максималната тежест на филтриран връх. Следният алгоритъм реализира точно тази идея:

```
HeaviestGood(T = (V, E, r, w): кореново дърво с тегловна функция на върховете)
1  създай нулирани глобални масиви d[1..n], f[1..n], dpref[0..n+1], fsuff[0..n+1]
2  InitTraversalArrays(T)
3  best ← −∞
4  for u ∈ V
5      if dpref[d[u] − 1] > w(u)  or fsuff[f[u] + 1] > w(u)
6          best ← max(best, w(u))
7  return best
```

```
InitTraversalArrays(T = (V, E, r, w))
1  създай нулирани глобални променливи dtime и ftime
2  DFS-Visit(T, r)
3  for i ∈ 1 to n
4      dpref[i] ← max(dpref[i − 1], dpref[i])
5  for i ∈ n downto 1
6      fsuff[i] ← max(fsuff[i + 1], fsuff[i])
```

```
DFS-Visit(T = (V, E, r, w), u ∈ V)
1  dtime ← dtime + 1
2  dpref[dtime] ← w(u)
3  d[u] ← dtime
4  for v ∈ adj[u]
5      if d[v] = 0
6          DFS-Visit(T, v)
7  ftime ← ftime + 1
8  fsuff[ftime] ← w(u)
9  f[u] ← ftime
```

От лекции знаем, че сложността на DFS-Visit е $O(m+n)$. Тъй като работим над дървета, това е просто $O(n)$. Алгоритъмите HeaviestGood и InitTraversalArrays извършват линейна допълнителна работа, затова тяхната сложност също е $O(n)$.