Примерни решения на домашно № 3

Задача 1: В тази задача разглеждаме построяването на транзитивното затваряне на ориентиран граф $G = (V, E)$. Транзитивното затваряне на G, което за краткост ще записваме с "$TrC(G)$", е реброво-минималният граф с множество от върхове V, който съдържа G като подграф и е транзитивен. Ориентиран граф H е транзитивен, ако за всеки $u, v, w \in V(H)$ е вярно, че $(u, v) \in E(H)$ и $(v, w) \in E(H)$ влече $(u, w) \in E(H)$.

Иначе казано, ако $R \subseteq V \times V$ е релацията, съответстваща на G, то $TrC(G)$ съответства на транзитивното затваряне на R.

4 т.
- Нека G е представен с матрица на съседство. Конструирайте алгоритъм за построяване на $TrC(G)$, който първо пуска еднократно алгоритъма на Floyd-Warshall и после използва матрицата D, която алгоритъмът на Floyd-Warshall връща, за да изчисли $TrC(G)$, представен чрез матрица на съседство. Вашият алгоритъм трябва да има сложност по време, не по-лоша в асимптотичния смисъл от тази на алгоритъма на Floyd-Warshall. За пълен брой точки трябва да обясните какъв вход подавате на алгоритъма на Floyd-Warshall.

16 т.
- Задача 25.2-8 в учебника на Cormen, Leiserson и Rivest (страница 700 в третото издание) иска да се конструира алгоритъм със сложност $O(|V| \cdot |E|)$ за намиране на $TrC(G)$. Обяснете защо не можете да ползвате алгоритъма от предната подточка за тази цел.

  Конструирайте алгоритъм със сложност $O(|V| \cdot |E|)$ за намиране на $TrC(G)$. Сега G е описан чрез списъци на съседство и полученият $TrC(G)$ също трябва да е описан чрез списъци на съседство. Обосновете накратко коректността и сложността по време на Вашия алгоритъм. В тази подточка можете да допуснете, че G е слабо свързан.

Решение: Първо да се убедим, че в $TrC(G)$ примки няма. Причината е, че в G няма примки – за да има възможност за примки, трябва да е казано изрично. Ерго, G като релация е антирефлексивна. Тогава и транзитивното ѝ затваряне е антирефлексивна.

Нека M е матрицата на съседство на G. От нея изчисляваме квадратна $n \times n$ матрица W, която е подходящ вход за алгоритъма на Floyd-Warshall, така:

```
for i from 1 to n
  for j from 1 to n
    if i = j
       W[i,j] := 0
    else
      if M[i,j] = 1
         W[i,j] := 1
      else
         W[i,j] := \infinity
```

Тук "`\infinity`" означава ∞. Иначе казано, слагаме тегла единици, без да добавяме ребра.

Пускаме алгоритъма на Floyd-Warshall с вход W и той връща матрица D. Нея я преобразуваме така:

```
for i from 1 to n
  for j from 1 to n
    if not (i = j)
      if D[i,j] = \infinity
         D[i,j] := 0
      else
         D[i,j] := 1
```

Да видим защо това е коректно. Матрицата, която F-W връща, има нули по главния диагонал, а матрицата на съседство на $TrC(G)$ трябва да има нули по главния диагонал, така че клетките `D[i,i]` са наред. За останалите клетки:

- `D[i,j] = \infinity` означава недостижимост на `j` от `i` в G, което на свой ред означава липса на ребро $(\texttt{i}, \texttt{j})$ в $TrC(G)$.
- `D[i,j]` $\neq$ `\infinity` означава достижимост на `j` от `i` в G, което на свой ред означава наличие на ребро $(\texttt{i}, \texttt{j})$ в $TrC(G)$.

Алгоритъм със сложност $O(|V| \cdot |E|)$ за намиране на $TrC(G)$ може да се направи, като пускаме алгоритъм за обхождане—или BFS, или DFS—от всеки връх u и слагаме в списъка на u всеки връх, който бива обходен, без самия u (за да не сложим примка в $TrC(G)$). Получените по този начин списъци са списъците на съседство на $TrC(G)$.

Обосновката на коректността отново е, че транзитивното затваряне е релацията на достижимост. А ние знаем, че BFS е коректен алгоритъм за обхождане и обхожда точно върховете, които са достижими от даден начален връх.

Сложността по време на едно пускане на BFS е $O(n + m)$, което знаем от лекции. Тъй като пускаме BFS от всеки връх, общата сложност е $O(n(n + m))$, което е $O(n^2 + mn)$. Тъй като G е слабо свързан, $m = \Omega(n)$, така че можем да запишем сложността като просто $O(mn)$. Ако $m = \Theta(n^2)$, това става $O(n^3)$, тоест, колкото намирането на транзитивното затваряне чрез F-W. Но ако графът е разреден, $O(mn)$ не става $O(n^3)$ и този алгоритъм е по-бърз в асимптотичния смисъл.

Задача 2: Даден е ориентиран цикличен граф $G = (V, E)$. Нека $\mathcal{C}$ е множеството от циклите на G. Нека

$$A \stackrel{\text{def}}{=} \bigcap \{E(c) \,|\, c \in \mathcal{C}\}$$

Конструирайте алгоритъм с колкото е възможно по-ниска сложност по време, който връща 0, ако $A = \varnothing$, или връща произволен елемент на A, в противен случай. Обосновете коректността и сложността по време на Вашия алгоритъм.

Не е необходимо алгоритъмът да е написан с псевдокод. Достатъчно е да е напълно ясно и недвусмислено какво имате предвид, както и да става ясно каква е сложността по време.

Бонус 10 точки Как се променя задачата, ако G е неориентиран граф?

Решение: Първо да видим какво се иска. На прост български, иска се ребро, през което минава всеки цикъл в графа, или индикация, че такова ребро няма.

Има различни решения. Всяко ребро e, което се съдържа във всеки цикъл в графа, се характеризира с това, че $G - e$ е ацикличен, тъй като всички цикли престават да съществуват с изтриването на e. Ако това се "преведе" директно в алгоритъм, получаваме алгоритъм със сложност $\Theta(m(n+m))$, което в най-лошия случай е квартичен алгоритъм. Такова решение се оценява с **десет (10) точки**.

По-добро решение е с DFS да се намери един цикъл c. Знаем, че DFS открива цикличност чрез откриване на поне едно ребро назад (x, y). Тогава в стека има върхове, в този ред отгоре надолу, $y, u_1, \ldots, u_k, x$, където $k \geqslant 0$. Тогава $c = y, u_1, \ldots, u_k, x, y$, като $E(c) = \{(y, u_1), (u_1, u_2), \ldots, (u_k, x), (x, y)\}$. Достатъчно е за ребрата от $E(c)$ да тестваме дали изтриването на някое от тях прави графа ацикличен: ако да, връщаме това ребро, ако ли не, връщаме нула. Броят на тези ребра е $O(n)$, понеже $|c| \leqslant n$, така че този алгоритъм има сложност $O(n(n+m))$, като в най-лошия случай е $\Theta(n(n+m))$, което в най-лошия случай е кубичен алгоритъм. Такова решение се оценява с **двадесет (20) точки**.

Ето едно още по-бързо решение. За него или решение със сходна сложност по време се дават **четиридесет (40) точки**. Ако сумата от точките на домашното е над 60, това ще е бонус, който не се губи.

БОО, можем да смятаме, че G е силно свързан и няма примки.

- Ако G не е силно свързан и поне две силно свързани компоненти G_1 и G_2 съдържат цикли c_1 и c_2, решение няма, понеже c_1 и c_2 нямат дори общи върхове. Ако G не е силно свързан и точно една силно свързана компонента G_1 има цикли, разглеждаме само G_1: решение за G има тстк има решение върху G_1.
- Ако G има примки, решение има тстк примката е само една и нейното изтриване води до даг; с други думи, тя е единственият цикъл. Ако това е така, решението е реброто-примка. В противен случай решение няма и връщаме нула.

Пускаме DFS върху G еднократно от произволен връх s, обхождайки целия G. Тъй като G е цикличен, DFS открива поне едно ребро назад. Нека множеството от ребрата назад е

$$B = \{(x_1, y_1), (x_2, y_2), \ldots, (x_k, y_k)\}$$

Внимание! – възможно е $x_i = x_j$ за $i \neq j$ или $y_i = y_j$ за $i \neq j$. Не може обаче $x_i = x_j$ и $y_i = y_j$ за различни i и j, понеже G не е мултиграф. Също така, $x_i \neq y_i$ за $1 \leqslant i \leqslant k$, понеже няма примки. Нека $S = \{x_1, \ldots, x_k, y_1, \ldots, y_k\}$. Както стана ясно, може $|S| = 2k$ или $|S| < 2k$.

Нека $G' = G - B$; тоест, изтриваме ребрата назад и получаваме G'. Очевидно G' е даг, защото върху него DFS не открива ребра назад. Освен това $V(G') = V$. Както знаем, върховете на G', обратно подредени по времена на финализиране от DFS (който вече пуснахме), представляват топологическо сортиране на G'. Нещо повече. По отношение на G, ребрата, които са с "неправилна посока" спрямо топосортировката (краят е вляво от началото), са точно ребрата от B. Нека t е функцията на това топологическо сортиране. Както знаем, t е биекция $t : V \to \{1, \ldots, n\}$.

Твърдим, че ако съществуват x_i и y_j, такива че $t(x_i) < t(y_j)$, решение няма. Да кажем, че има x_i и y_j, такива че $t(x_i) < t(y_j)$. Очевидно $i \neq j$, защото, ако беше изпълнено $i = j$, щеше да има ребро от B, на което началото е вляво от края в топосортировката. И така, $i \neq j$ и сме допуснали $t(x_i) < t(y_j)$. Забелязваме, че $t(y_i) < t(x_i)$ и $t(y_j) < t(x_j)$, понеже става дума за ребра от B. Тогава y_i, x_i, y_j и x_j са четири различни върха, които в топологическото сортиране се появяват в този ред:

$$\ldots\ldots\ldots y_i \ldots\ldots\ldots x_i \ldots\ldots\ldots y_j \ldots\ldots\ldots x_j \ldots\ldots\ldots$$

Сега е напълно очевидно, че в G има поне два цикъла без общи върхове, така че решение няма. Ерго, ако поне един от хиксовете е вляво от някой от игреците, връщаме нула.

Остава да разгледаме случая, в който всички върхове от $\{y_1, \ldots, y_k\}$ са преди всички върхове от $\{x_1, \ldots, x_k\}$ в топологическото сортиране. Взаимното разположение на игреците няма значение, също така няма значение взаимното разположение на хиксовете. Това, че е игреците са преди хиксовете в топосортировката обаче не гарантира, че има решение. Може да има, може да няма. БОО, нека

$$t(y_1) \leqslant t(y_2) \leqslant \cdots \leqslant t(y_k)$$
$$t(x_1) \leqslant t(x_2) \leqslant \cdots \leqslant t(x_k)$$

Да дефинираме y-x-път като всеки прост път в G', който започва в някой y_i и завършва в някой x_j. Тези пътища са ключови, защото решение има тстк има ребро, което принадлежи на всички тях. Тъй като те са пътища в G', в топосортировката техните ребра са само отляво надясно. Ако за всяка "празнина" между два съседни върха в топосортировката запишем колко ребра от y-x-пътища "прескачат" тази "празнина", решение има тстк някоя празнина вдясно от y_k и вляво от x_1 има само едно такова прескачащо ребро: тогава то очевидно е във всеки цикъл на G. Нула прескачащи ребра не може да има, защото, ако имаше "празнина" с нула прескачащи ребра, нямаше да има цикли в G.

Нека $W = \{u \in V \,|\, u \text{ е връх от някой } y\text{-}x\text{-път}\}$. Тогава можем да получим решение, ако разгледаме само върховете от W, игнорирайки останалите. В топосортировката върховете от W са между y_1 и x_k, но не всеки връх между y_1 и x_k е непременно от W. Ако някак сме намерили и маркирали върховете от W—всеки връх има булев флаг, който е 1 тстк този връх е от W—лесно можем да намерим ребро, за каквото стана дума. По отношение на подграфа на G', индуциран от върховете от W—да наречем този подграф H—това ребро е нещо като мост, само че ориентиран. Ние знаем как се намират всички мостове на неориентиран граф, и то в линейно време. И така, ако конструираме неориентирания граф $\widehat{H}$, съответен на H, което можем да сторим в линейно време, можем да намерим мостовете на $\widehat{H}$ (в линейно време) после да видим дали в топосортировката на H на някой мост съответства ориентирано ребро с начало вдясно от, или съвпадащо с, y_k и край вляво от, или съвпадащ с x_1. Топосортировката на H е просто наредбата на върховете на H, която следва от t.

Остава да видим как може да генерираме W, и то във време $\Theta(n + m)$. Разглеждаме само върховете на V, които в топосортировката са между y_1 и x_k включително. Всеки връх маркираме с наредена двойка от булеви флагове, левият елемент от която показва дали върхът е достижим от някой y_i, а десният – дали от него е достижим някой x_j. Ако имаме тези наредени двойки, върховете от W са точно тези, които са маркирани с $(1, 1)$. За левите компоненти на флаговете можем да приложим идея, аналогична на построяването на най-къси пътища в топосортирани дагове. Поначало тези компоненти са нули, а само игреците имат единици. После обхождаме върховете от y_1 до x_k в реда на топосортировката и за всеки връх, който има единица, записваме единици в левите компоненти на върховете от списъка му на съседство. Гарантирано те са вдясно от него и така записаните единици индикират достижимост от някой игрек.

За десните компоненти, които показват наличие на път до някой хикс, можем да направим аналогичното нещо в обратната посока, но върху транспонирания граф.

Ако се имплементира внимателно, тази идея води до линеен алгоритъм.

Ако G е неориентиран граф, задачата става тривиална. В неориентиран цикличен граф има ребро, което се съдържа във всеки цикъл тстк цикълът е само един. Ако има два цикъла, независимо от това дали имат общо ребро или не, няма как едно ребро да се съдържа във всеки цикъл: ако нямат общо ребро, това е очевидно, а ако имат общо ребро, тогава има и трети цикъл, който не съдържа общите ребра на двата.

И така, за да има ребро, което е във всеки цикъл, графът трябва да има една свързана компонента, която е уницикличен граф, а останалите, ако има такива, да са дървета. Това може да се установи във време $\Theta(n)$, и то само с бройките на ребрата в свързаните компоненти – за компонента с k върха, тя е дърво тстк има $k-1$ ребра и е уницикличен граф тстк има k ребра.

Задача 3: Даден е масив от цели числа $A[1,\ldots,n]$, където $n \geqslant 2$. Предложете алгоритъм със сложност по време $O(n)$ и сложност по памет $O(1)$, който изчислява

$$\max\{A[i] - A[j] \,|\, 1 \leqslant i < j \leqslant n\}$$

Обосновете коректността и сложността по време на Вашия алгоритъм.

Решение: Следният алгоритъм решава задачата.

```
Alg1(A[1,...,n]: int, където n ⩾ 2)
1  Δ ← A[1] − A[2]
2  Max ← max{A[1], A[2]}
3  for i ← 3 to n
4      if Max − A[i] > Δ
5          Δ ← Max − A[i]
6      if A[i] > Max
7          Max ← A[i]
8  return Δ
```

При всяко достигане на ред 3:

1. Max съдържа $\max\{A[i] \,|\, 1 \leqslant i \leqslant i-1]\}$,
2. а Δ съдържа $\max\{A[p] - A[q] \,|\, 1 \leqslant p < q \leqslant i-1\}$.

Това е очевидно вярно при първото достигане. Нека е вярно при някое достигане, което не е последното. Ще докажем поотделно първата и втората част на инварианта, понеже редове 4 и 5 касаят втората част и само нея, а редове 6 и 7 касаят първата част и само нея.

1. Да допуснем, че

$$\max\{A[p] \,|\, 1 \leqslant p \leqslant i-1\} - A[i] > \max\{A[p] - A[q] \,|\, 1 \leqslant p < q \leqslant i-1\} \tag{1}$$

Но това е същото като

$$\max\{A[p] - A[i] \,|\, 1 \leqslant p \leqslant i-1\} > \max\{A[p] - A[q] \,|\, 1 \leqslant p < q \leqslant i-1\} \tag{2}$$

Очевидно множеството $X = \{(p,q) \,|\, 1 \leqslant p < q \leqslant i\}$ се разбива на $Y = \{(p,q) \,|\, 1 \leqslant p < q \leqslant i-1\}$ и $Z = \{(p,i) \,|\, 1 \leqslant p \leqslant i-1\}$. В (2) установихме, че $\max Z > \max Y$. Но тогава $\max X = \max Z$. Тоест,

$$\underbrace{\max\{A[p] - A[q] \,|\, 1 \leqslant p < q \leqslant i\}}_{\max X} = \underbrace{\max\{A[p] - A[i] \,|\, 1 \leqslant p \leqslant i-1\}}_{\max Z} \tag{3}$$

Съгласно първата част на индуктивното предположение, при допускането (1) имаме

$$\text{Max} - A[i] > \Delta$$

Тогава условието на ред 4 е истина и изпълнението отива на ред 5, където Δ получава стойност $\text{Max} - A[i]$, което е равно на $\max\{A[p] - A[q] \,|\, 1 \leqslant p < q \leqslant 1\}$ съгласно (3). И така, $\Delta = \max\{A[p] - A[q] \,|\, 1 \leqslant p < q \leqslant 1\}$. При следващото достигане на ред 3, в термините на новото i отново е вярно, че Δ съдържа $\max\{A[p] - A[q] \,|\, 1 \leqslant p < q \leqslant i-1\}$. Първата част на инварианта се запазва.

Сега да допуснем негацията на (1), а именно

$$\max\{A[p] \,|\, 1 \leqslant p \leqslant i-1\} - A[i] \leqslant \max\{A[p] - A[q] \,|\, 1 \leqslant p < q \leqslant i-1\} \tag{4}$$

Но това е същото като

$$\max\{A[p] - A[i] \,|\, 1 \leqslant p \leqslant i-1\} \leqslant \max\{A[p] - A[q] \,|\, 1 \leqslant p < q \leqslant i-1\} \tag{5}$$

Както отбелязахме вече, множеството $X = \{(p,q) \,|\, 1 \leqslant p < q \leqslant i\}$ се разбива на $Y = \{(p,q) \,|\, 1 \leqslant p < q \leqslant i-1\}$ и $Z = \{(p,i) \,|\, 1 \leqslant p \leqslant i-1\}$. В (5) установихме, че $\max Z \leqslant \max Y$. Но тогава $\max X = \max Y$. Тоест,

$$\underbrace{\max\{A[p] - A[q] \,|\, 1 \leqslant p < q \leqslant 1\}}_{\max X} = \underbrace{\max\{A[p] - A[q] \,|\, 1 \leqslant p < q \leqslant i-1\}}_{\max Y} \tag{6}$$

Съгласно индуктивното предположение, при допускането (4) имаме

$$\text{Max} - A[i] \leqslant \Delta$$

Тогава условието на ред 4 е лъжа и изпълнението не отива на ред 5. Δ си остава със стойност $\max\{A[p] - A[q] \,|\, 1 \leqslant p < q \leqslant i-1\}$ съгласно първата част на индуктивното предположение, което е равно на $\max\{A[p] - A[q] \,|\, 1 \leqslant p < q \leqslant 1\}$ съгласно (6). При следващото достигане на ред 3, в термините на новото i отново е вярно, че Δ съдържа $\max\{A[p] - A[q] \,|\, 1 \leqslant p < q \leqslant i-1\}$. Първата част на инварианта се запазва.

2. Втората част на инварианта се доказва по-лесно. Първо да допуснем, че $A[i] > \max A[1, \ldots, i-1]$. Но съгласно втората част на индуктивното предположение, това е същото като $A[i] > \text{Max}$. Условието на ред 6 е истина и на ред 7 Max получава стойност $A[i]$, което е равно на $\max A[1, \ldots, i]$ при текущото допускане. След инкрементирането на i на ред 3, отново е вярно, че Max съдържа $\max A[1, \ldots, i-1]$. Втората част на инварианта се запазва.

Сега да допуснем, че $A[i] \leqslant \max A[1, \ldots, i-1]$. Но съгласно втората част на индуктивното предположение, това е същото като $A[i] \leqslant$ Max. Условието на ред 6 е лъжа и Max остава със стойността—съгласно втората част на индуктивното предположение—$\max A[1, \ldots, i-1]$, което е същото като $\max A[1, \ldots, i]$ при текущото допускане. След инкрементирането на i на ред 3, отново е вярно, че Max съдържа $\max A[1, \ldots, i-1]$. Втората част на инварианта се запазва.

Доказахме инварианта. При последното достигане на ред 3 е вярно, че $n + 1$. Заместваме във втората част на инварианта и получаваме "Δ съдържа $\max\{A[p] - A[q] \mid 1 \leqslant p < q \leqslant n\}$". Алгоритъмът наистина връща желаната максимална разлика на ред 8.

Сложността по време очевидно е $O(n)$, а по памет е $O(1)$.