Задачи върху графи

Зад. 1

Даден е ориентиран граф $G = \langle V, E \rangle$. Да се състави ефикасен алгоритъм, който проверява дали G съдържа цикъл.

```
1. hasCycleFrom(G = ⟨V, E⟩, v, col[1 .. n]) : // V = {1, …, n}, v ∈ V
2.       if col[v] = black then
3.                return FALSE
4.       s ← Stack.Init()
5.       s.push(v)
6.       while s.isEmpty() = FASLE do
7.                u ← s.top()
8.                if col[u] = gray then
9.                         col[u] ← black
10.                        s.pop()
11.               else  // guaranteed white
12.                        col[u] ← gray
13.                        for each ⟨u, v⟩ ∈ E do
14.                                 if col[v] = white then
15.                                          s.push(v)
16.                                 else if col[v] = gray then
17.                                          return TRUE
18.      return FALSE
```

```
1. hasCycle(G = ⟨V, E⟩) : // V = {1, …, n}
2.       col[1 .. n] ← [white, …, white]
3.       for each v ∈ V do
4.                if hasCycleFrom(G, v, col) then
5.                         return TRUE
6.       return FALSE
```

Зад. 2

Даден е неориентиран граф $G = \langle V, E \rangle$. Да се състави ефикасен алгоритъм, който проверява дали G е двуделен.

```
1. isBipartite(G = ⟨V, E⟩) : // V = {1, …, n}
2.       col[1 .. n] ← [blue, NULL, …, NULL]
3.       q ← Queue.Init()
4.       q.push(V[1])
5.       while q.isEmpty() = FALSE do
6.                u ← q.pop()
7.                for each ⟨u, v⟩ ∈ E do
8.                         if col[v] = NULL then
9.                                  col[v] ← other(col[u]) //other(blue)⟼red; other(red)⟼blue
10.                                 q.push(v)
```

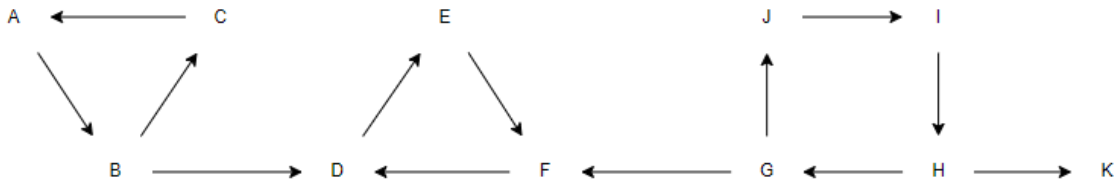
```
11.                 else if col[v] = col[u] then
12.                         return FALSE
13.     return TRUE
```

Зад. 3

Даден е ориентиран граф $G = \langle V, E \rangle$. Да се състави ефикасен алгоритъм, който намира силно свързаните компоненти на G.

Пример:

Instance : G=



Solution : {{A,B,C},{D,E,F},{G,J,I,H},{K}}

```
1. dsf(G = ⟨V, E⟩, u, s, visited[1 .. n]) : // V = {1, …, n}
2.     if visited[u] then
3.         return
4.     visted[u] ← TRUE
5.     for each ⟨u, v⟩ ∈ E do
6.         if visited[v] = FALSE then
7.             dfs(G, v, s, visited)
8.     s.push(u)
```

```
1. dfs2(G = ⟨V, E⟩, u, comp[1 .. n], compCnt, visited[1 .. n]) : // V = {1, …, n}
2.     if visited[u] then
3.         return
4.     visited[u] ← TRUE
5.     comp[u] ← compCnt
6.     for each ⟨u, v⟩ ∈ E do
7.         if visited[v] = FALSE then
8.             dfs2(G, v, comp, compCnt, visited)
```

```
1. SCC(G = ⟨V, E⟩) : // V = {1, …, n}
2.     visited[1 .. n] ← [FALSE, …, FALSE]
3.     comp[1 .. n] ← [NULL, …, NULL]
4.     compCnt ← 0
5.     s ← Stack.Init() // stack of vertices
6.     for each u ∈ V do
7.         dfs(G, u, s, visited)
8.     G' ← reverseGraph(G) // returns graph G' = ⟨V, E'⟩ : (∀ u ∈ V) (∀ v ∈ V) (⟨u, v⟩ ∈ E ↔ ⟨v, u⟩ ∈ E')
9.     visited[1 .. n] ← [FALSE, …, FALSE]
10.    while s.isEmpty() = FALSE do
11.        u ← s.pop()
12.        if visited[u] then
13.            continue
```

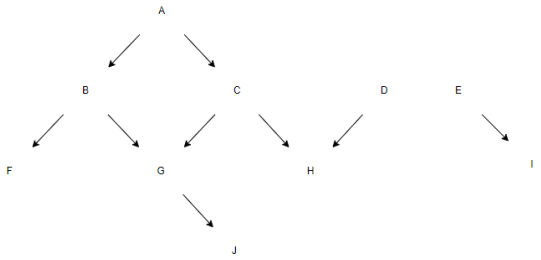
```
14.             dfs2(G, u, comp, compCnt, visited)
15.             compCnt ← compCnt + 1
16.     return comp
```

Зад. 4

Даден е ацикличен ориентиран граф $G = \langle V, E \rangle$. Да се състави ефикасен алгоритъм, който сортира топологически върховете на G.

Пример:

Instance :



Solution : ⟨E,A,B,I,F,C,G,J,D,H⟩ (*има много други възможности*)

```
1. dfs(G = ⟨V, E⟩, u, s, visited[1 .. n]) : // V = {1, …, n}
2.     if visited[u] then
3.         return
4.     visited[u] ← TRUE
5.     for each ⟨u, v⟩ ∈ E do
6.         if visited[v] = FALSE then
7.             dfs(G, v, s, visited)
8.     s.push(u)
```

```
1. topSort(G = ⟨V, E⟩) : // V = {1, …, n}
2.     s ← Stack.Init() // stack of vertices
3.     ans ← List.Init() // list of vertices
4.     visited[1 .. n] ← [FALSE, …, FALSE]
5.     for each u ∈ V do
6.         dfs(G, u, s, visited)
7.     while s.isEmpty() = FALSE do
8.         ans.push_back (s.pop())
9.     return ans
```

Зад. 5

Даден е свързан неориентиран граф $G = \langle V, E \rangle$. Да се състави алгоритъм, построяващ МПД T на G със следните сложности:

a) $O(m\log(n))$ - чрез Kruskal's algorithm

b) $O(n^2)$ - чрез Prim's algorithm (масив)

c) $O(m\log(n))$ - чрез Prim's algorithm (binary heap)

d) $O(m + n\log(n))$ - чрез Prim's algorithm (Fibonacci heap)

Забележка Сложностите горе са илюстративни.. ясно е, че ако направим решение на *d*), то няма нужда от подточки изобщо.. също

очевидно *a*) и *c*) имат еднаква времева сложност.. идеята е да се видят 4 различни решения

***a*)** Kruskal's algorithm

Преди да покажем алгоритъма на крускал, трябва да разгледаме така наречениете *структури от данни за непресичащи се множества (disjoint-set data structure)*. Както се предполага от името тя съдържа колекция от непресичащи се множества. Това, което не предполага името е, че тези множества са динамични (т.е. ще ги променяме в работата на програмта) и всяко множество ще си има представител, който идентифицира множеството. Има три основни операции които тази структура поддържа:

· union - обединява две множества в едно (общия брой множества в колекцията намалява с единица/остава същият)

· find - връща представителя на множеството в което се намира елемента

· initSet - образува n на брой непресичащи се множества с по един елемент

```
1. Struct DS:
2.     parent ← NULL
3.     find(a)
4.     union(a, b)
5.     initSet(n)
```

```
1. DS.initSet(n) : // n ∈ ℕ⁺
2.     parent ← [1, 2, …, n]
3.     return parent
```

```
1. DS.find(a) : // a ∈ {1, …, n}
2.     if parent[a] ≠ a then
3.         return find(parent[a])
4.     return parent[a]
```

```
1. DS.union(a, b) : // a, b ∈ {1, …, n}
2.     a ← find(a)
3.     b ← find(b)
4.     if a = b then
5.         return FALSE
6.     parent[b] ← a
7.     return TRUE
```

По този начин обаче може да получим дървета с много голяма височина и няма да имаме добра сложност.. правим следната оптимизация на метода DS.find(a)

```
1. DS.find(a) : // a ∈ {1, …, n}
2.     if parent[a] ≠ a then
3.         parent[a] ← find(parent[a])
4.     return parent[a]
```

Така при повикване на DS.find(a) дори ако е много висок клона по който търсим корена на дървото (представителя на множеството), то всеки един такъв връх вече ще ни се намира на височина 2 (дете на корена). Няма да разглеждаме подробно сложността на тези операции, тъй като са съвсем нетривиални. Нека обаче ги видим набързо:

· DS.initSet(n) очевидно е $\theta(n)$

· DS.find(a) има средна и най-лоша сложност $\alpha(n)$, където α е *inverse Ackermann function*.

· DS.union(a, b) има средна и най-лоша сложност $\alpha(n)$, където α е *inverse Ackermann function*.

Забележка *Ackermann function* расте толкова бързо, че за всички практични цели можем да ограничим стойностите на обратната ѝ по следния начин: $(\forall\, n \in \mathbb{N}_{\text{practical}})[\alpha(n) \le 4]$. Тоест ще го взимаме за константа.

Деф (локална) $\mathbb{N}_{\text{practical}} \stackrel{\text{def}}{=} \{0,\, 1,\, \ldots,$ *броя атоми във видимата вселената*$\}$

```
1. MSTKruskal(G = ⟨V, E⟩) : // V = {1, …, n}
2.     E ← sortByWeight(E, order = incrementing)
3.     E' ← List.Init()
4.     ds ← DS.initSet(n)
5.     for each ⟨{u, v}, w⟩ ∈ E : u < v do //in the sorted order!
6.         if ds.union(u, v) then
7.             E'.push_back (⟨{u, v}, w⟩)
8.     return G' = ⟨V, E'⟩
```

***b*)** Prim' *s* algorithm (масив)

```
1. getMinDistIdx(dist[1 .. n], visited[1 .. n]) : // dist ∈ ℝ⁺, visited ∈ {0, 1}ⁿ
2.     idx ← 0
3.     for i ← 1 to n
4.         if visited[i] = FALSE then
5.             idx ← i
6.             break
7.     for i ← idx + 1 to n
8.         if visited[i] = FALSE and dist[i] < dist[idx] then
9.             idx ← i
10.    return idx
```

```
1. MSTPrimArray(G = ⟨V, E⟩) : // V = {1, …, n}
2.     E' ← List.Init()
3.     dist[1 .. n] ← [∞, …, ∞]
4.     from[1 .. n] ← [NULL, …, NULL]
5.     visited[1 .. n] ← [TRUE, FALSE, FALSE, …, FALSE]
6.     for each ⟨{1, v}, w⟩ ∈ E do
7.         from[v] ← 1
8.         dist[v] ← w
9.     for i ← 2 to n
10.        u ← getMinDistIdx(dist, visited) // БОО (∀ k ∈ {1, …, n})[V[k] = k]
11.        E'.push_back (⟨{from[u], u}, dist[u]⟩)
12.        visited[u] ← TRUE
13.        for each ⟨{u, v}, w⟩ ∈ E do
14.            if w < dist[v] then
15.                from[v] ← u
16.                dist[v] ← w
17.    return G' = ⟨V, E'⟩
```

Тази реализация е асимптотично по-добра от *a*) и *c*), когато $m = \theta\left(n^2\right)$.. тоест когато графа е *dense graph*. Проверете го (тривиално заместване)!

c) чрез Prim' *s* algorithm (binary heap)

```
1. MSTPrimBinaryHeap(G = ⟨V, E⟩) : // V = {1, …, n}
2.     E' ← List.Init()
3.     visited[1 .. n] ← [TRUE, FALSE, FALSE, …, FALSE]
4.     pq ← PriorityQueue.Init() // priority queue of weighted edges,  compared by weight
5.     for each ⟨{1, v}, w⟩ ∈ E do
6.         pq.push(⟨{1, v}, w⟩)
7.     for i ← 2 to n
8.         ⟨{u, v}, w⟩ ← pq.pop()
9.         while visited[u] and visited[v] do
10.            ⟨{u, v}, w⟩ ← pq.pop()
11.        E'.push_back (⟨{u, v}, w⟩)
12.        if visited[u] then
13.            for each ⟨{v, v'}, w'⟩ ∈ E do
14.                pq.push(⟨{v, v'}, w'⟩)
15.            visited[v] ← TRUE
16.        else  // visited[v] = TRUE
17.            for each ⟨{u, v'}, w'⟩ ∈ E do
18.                pq.push(⟨{u, v'}, w'⟩)
19.            visited[u] ← TRUE
20.    return G' = ⟨V, E'⟩
```

d) Prim' *s* algorithm (Fibonacci heap)

Operation	find-min	delete-min	insert	decrease-key	meld
Binary[8]	Θ(1)	Θ(log *n*)	*O*(log *n*)	*O*(log *n*)	Θ(*n*)
Leftist	Θ(1)	Θ(log *n*)	Θ(log *n*)	*O*(log *n*)	Θ(log *n*)
Binomial[8][9]	Θ(1)	Θ(log *n*)	Θ(1)[a]	Θ(log *n*)	*O*(log *n*)[b]
Fibonacci[8][2]	Θ(1)	*O*(log *n*)[a]	Θ(1)	Θ(1)[a]	Θ(1)
Pairing[10]	Θ(1)	*O*(log *n*)[a]	Θ(1)	*o*(log *n*)[a][c]	Θ(1)
Brodal[13][d]	Θ(1)	*O*(log *n*)	Θ(1)	Θ(1)	Θ(1)
Rank-pairing[15]	Θ(1)	*O*(log *n*)[a]	Θ(1)	Θ(1)[a]	Θ(1)
Strict Fibonacci[16]	Θ(1)	*O*(log *n*)	Θ(1)	Θ(1)	Θ(1)
2–3 heap[17]	*O*(log *n*)	*O*(log *n*)[a]	*O*(log *n*)[a]	Θ(1)	?

a. ^ a b c d e f g h i Amortized time.
b. ^ *n* is the size of the larger heap.
c. ^ Lower bound of $\Omega(\log \log n)$,[11] upper bound of $O(2^{2\sqrt{\log \log n}})$.[12]

Може да забележим, че за разлика от binary heap, Fibonacci heap поддържа операцията decrease – key за амортизирано $O(1)$ време. Тогава можем да подобрим ***c)*** като използваме decrease – key, вместо да слагаме повтарящи се върхове.. по този начин ще имаме, че пирамидата ни ще се състои от максимално $O(n)$ елемента, вместо $O(m)$ при ***c)***. Оттук времвата сложност спада до $O(m + n\log(n))$ и то не амортизирана. Няма да правим псевдокод, нито ще доказваме сложност. Fibonacci heap не се разглежда в рамките на текущия курс. Въпреки това е показано идейно за обща култура.

Заключение Kruskal' *s* algorithm се пише по-лесно и е с добра времева сложност. Prim' *s* algorithm изисква повече съобразителност, но е по-силно средство за намиране на MST (т.е. има добра версия за dense graph и добра версия за sparse graph, но има и "екстра добра" версия).

Зад. 6

Даден е тегловен и ориентиран граф $G = \langle V, E \rangle$. Даден е също така $v_{st} \in V$. Да се намерят теглата на най-кратките пътища от v_{st} до всички върхове $u \in V$, ако:

***a*)** няма отицателни тегла (бонус за поддръжка на минималните пътища)

***b*)** има отрицателни тегла (бонус за поддръжка на минималните пътища и бонус за проверка за отрицателен цикъл)

***a*)** Dijkstra's algorithm

```
1. Dijkstra(G = ⟨V, E⟩, v_st) : // V = {1, …, n}, v_st ∈ V, (∀ e ∈ E)[w(e) ≥ 0]
2.     visited[1 .. n] ← [FALSE, …, FALSE]
3.     dist[1 .. n] ← [∞, …, ∞]
4.     pred[1 .. n] ← [NULL, …, NULL]
5.     pq ← PriorityQueue.Init() // priority queue of 3 − tuples ⟨v_pred, v, w_total⟩ compared by w_total,
       where v_pred is the predecessor of v and w_total is the combined weight of the path v_st ↦ … ↦ v_pred ↦ v
6.     pq.push(⟨v_st, v_st, 0⟩) // we make up an edge − initial value for the priority queue
7.     while pq.isEmpty() = FALSE do
8.         ⟨v_pred, v, w_total⟩ ← pq.pop()
9.         if visited[v] then
10.            continue
11.        visited[v] ← TRUE
12.        dist[v] ← w_total
13.        pred[v] ← v_pred
14.        for each ⟨v, v_next, w⟩ ∈ E do
15.            if visited[v_next] = FALSE then
16.                pq.push(⟨v, v_next, dist[v] + w⟩) // equally ⟨v, v_next, w_total + w⟩
17.    return ⟨dist, pred⟩
```

Времева сложност: (аналогична на Prim's algorithm)

· Binary heap $-$ $O((m + n)\log(n))$

· Fibonacci heap $-$ $O(m + n\log(n))$ // прилагаме същата оптимизация както при Prim's algorithm

b)

```
1. BellmanFord(G = ⟨V, E⟩, v_st) : // V = {1, …, n}, v_st ∈ V
2.     dist[1 .. n] ← [∞, …, ∞]
3.     pred[1 .. n] ← [NULL, …, NULL]
4.     dist[v_st] ← 0
5.     for i ← 1 to n − 1
6.         for each ⟨u, v, w⟩ ∈ E do
7.             if dist[u] + w < dist[v] then
8.                 dist[v] ← dist[u] + w
9.                 pred[v] ← u
10.    return ⟨dist, pred⟩
```

Ако искаме да проверим дали имаме отрицателен цикъл, то изпълняваме тялото на for цикъла на ред 5 още веднъж (n-ти път) и ако имаме подобрение (т.е влезем в тялото на if на ред 7), то ни е гарантирано, че имаме отрицателен цикъл. Защо?

Времева сложност: $O(mn)$

Зад. 7

Даден е тегловен и ориентиран граф $G = \langle V, E \rangle$ без отрицателни цикли. Да се намерят теглата на най-кратките пътища от всеки до всеки връх

```
1. FloydWarshall(G = ⟨V, E⟩) : // V = {1, …, n}
2.       dist[1 .. n][1 .. n] ← [[∞, …, ∞], …, [∞, …, ∞]]
3.       for each ⟨u, v, w⟩ ∈ E do
4.                dist[u][v] ← w
5.       for cap ← 1 to n
6.                for u ← 1 to n
7.                         for v ← 1 to n
8.                                  dist[u][v] ← min(dist[u][v], dist[u][cap] + dist[cap][v])
9.       return dist
```

Времева сложност:

$\cdot O(n^3)$

Зад. 8

Дадени са ориентиран граф $G = \langle V, E \rangle$, естествено число $k \in \mathbb{N}_0$ и много на брой заявки от вида:

a)
{ Instance : $u, v \in V$
{ Solution : брой маршрути от u до v с дължина **точно** k

b)
{ Instance : $u, v \in V$
{ Solution : брой маршрути от u до v с дължина **не повече от** k

Предложете наредена двойка от максимално бързи алгоритми ⟨*индекс*, *заявка*⟩ за двете подточки.

Подсказка Използвайте следната теорема от ДС:

Th Нека $G = \langle V, E \rangle$ е ориентиран мултиграф. Нека $M_{n \times n}$ е матрицата на съседство, отговаряща на G. Тоест имаме, че $M[i, j]$ = *броя ребра от* v_i *до* v_j. Тогава $M^k[i, j]$ = *броя маршрути* с *дължина* k *от* v_i *до* v_j.