ДАА Семинар 10

Алгоритми върху графи

Деф. Граф

Граф е наредена двойка $G=(V,E)$, $V\neq\emptyset$ е м-во от върховете на G, а E - множество от ребрата на G, където $E\subseteq\{X\subseteq V : |X|=2\}$

Деф. Мултиграф

Мултиграф е наредена тройка $G=(V,E,f_G)$, $V\neq\emptyset$ е м-во от върховете на G, E - множество от ребрата на G, $E\cap V=\emptyset$ и

$f_G: E\to\{X\subseteq V : |X|=2\}$

Деф. Ориентиран граф

Ориентиран граф е наредена двойка $G=(V,E)$, където $V\neq\emptyset$ е м-вото от върховете на G, E - м-вото от ребрата, като $E\subseteq(V\times V)\setminus\{(u,u)\mid u\in V\}$

Деф. Свързаност в граф $G=(V,E)$

За всеки два върха $u,v\in V$ казваме че са свързани, ако съществува $u\overset{w/j}{-}v$ път. G е свързан, ако всеки два върха в него са свързани

Деф. Релацията на свързаност $Q_G\subseteq V\times V$ - рел. на екв.

$\forall u,v\in V: uQ_Gv \leftrightarrow u,v$ са свързани

Свързаните компоненти на G са максималните по включване свързани подграфи на G

Деф. Силна свързаност в ориентиран граф $G=(V,E)$

За всеки два върха $u,v\in V$ казваме че са силно свързани, ако съществува път от u до v и съществува път от v до u.

- G е силно свързан, ако всеки два върха в него са силно свързани.
- Силно свързаните компоненти са подграфите, индуцирани от класовете на еквивалентност на релацията силна свързаност
- Слабо свързаните компоненти съответстват на свързаните компоненти на съответния неориентиран граф

- Представяния на графи - $G=(V,E)$

1) Матрица на съседство

Може да представим G като булева матрица $M_{n\times n}$, където

$M[i,j]=1$, ако $(i,j)\in E$ и $M[i,j]=0$, ако $(i,j)\notin E$

- не е добро представяне за разредени графи / Дори да използваме триъгълна матрица (за неориентирани графи)

2) Списъци на съседство

-за всеки връх е даден списък от неговите съседи (за неориентиран граф) / деца (за ориентиран граф)

Критерий	Избор
По-бързо тестване $(x,y) \in E$	матрица на съседство
По-малко памет в/у разредени графи	списъци на съседство
По-бързо изтриване/ добавяне на ребро	матрица на съседство
По-бързо обхождане на графа	списъци на съседство

• Задачи върху графи

1) Обхождане на графи - в/у ориентирани мултиграфи с възможни примки

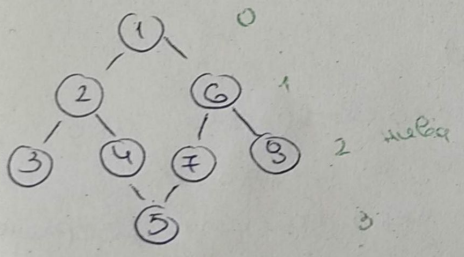
Обхождат се върховете и ребрата

BFS - пуска се с начален стартов връх

„обхожда по нива" / „строи" дърво на най-късите пътища (в нетегловни графи) от стартовия връх до всички останали

// Пример

Нека G е (неор.) представен чрез списъци на съседство:



Стартов връх: 1

1: 2, 6
2: 1, 3, 4
3: 2
4: 2, 5, 7
5: 4, 7
6: 1, 7, 8
7: 5, 6
8: 6

Тогава BFS открива върховете в следния ред:

1, 2, 6, 3, 4, 7, 8, 5

0 1 2 3 нива

Зад. ① Да се напише псевдокода на BFS

(G-ор.нуг.с.в.приики, V={1..n}, S-стартов)
for i←1 to n
2. color[i]←white, d[i]←∞, π[i]←nil
3. color[s]←gray
4. d[s]←0
5. Enqueue(Q,s)
6. while !Q.empty()
7. x←Dequeue(Q)
8. for y∈adj[x]
9. if color[y]=white
10. color[y]←gray
11. d[y]←d[x]+1
12. π[y]←x
13. Enqueue(Q,y)
14. color[x]←black

зад ② По даден граф G, представен чрез списъци на съседство, да се определи дали G е циклитен (дали съдържа цикъл), използвайки BFS

Решение:

Идея: Нека изпълнението е на ред 7 и текущият връх е X. Разглеждаме съседите на X. Ако X има посетен съсед Y и Y не е родител на X в строеното от BFS дърво (тоест, не сме открили X чрез Y), то това означава, че в текущото изпълнение на BFS има път от стартовия връх S до Y, който не минава през X, но също така има път от стартовия връх S до Y, който минава през X ⇒ има цикъл

HasCycle(G-неориент. граф чрез сп.н.с.)
1. for i←1 to n
2. visited[i]←0, π[i]←Nil
3. for i←1 to n
4. if !visited[i]
5. if checkComp(G,i,visited)
6. return true
7. return false

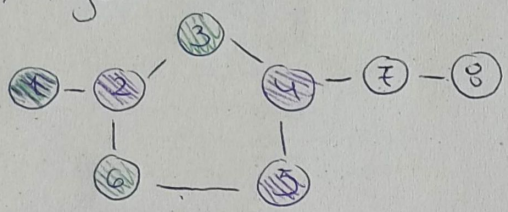
checkComp(G,S,visited)
1. visited[S]←true/1
2. Enqueue(Q,x)
3. while !Q.Empty()
4. x←Dequeue(Q)
5. for y∈adj[x]
6. if visited[y] && π[x]!=y
7. return true
8. if !visited[y]
9. visited[y]←true
10. π[y]←x
11. Enqueue(Q,y)

не сме открили x чрез y

Зад ③ По даден граф $G=(V,E)$, представен чрез списъци на съседство да се определи дали G е двуделен.

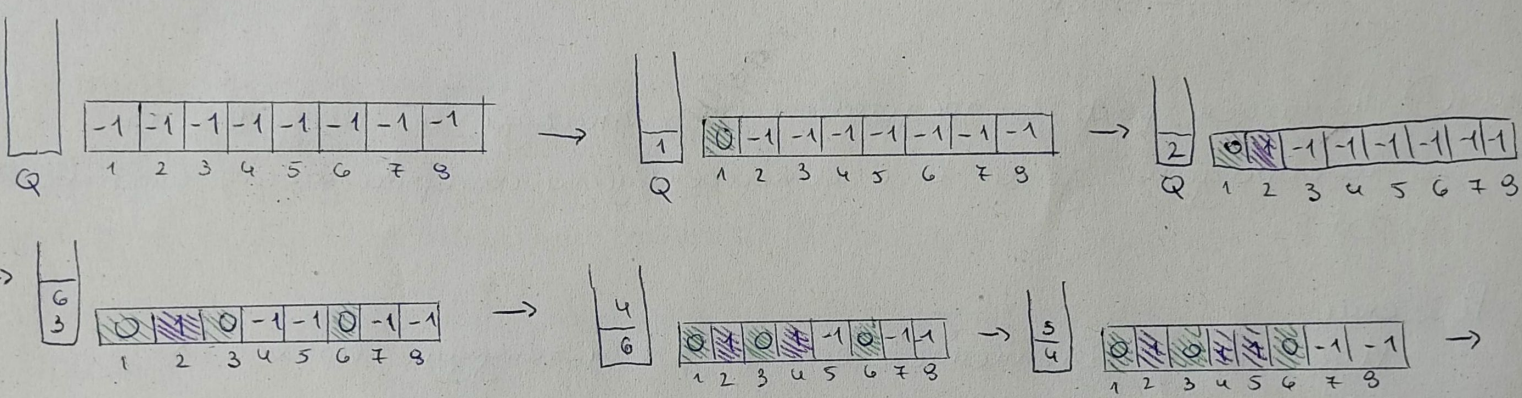
Решение: G да е двуделен е същото като да е двуоцветим.
Ще използване BFS за да определим дали G е двуоцветим

// Идея



1 → {2}
2 → {1, 3, 6}
3 → {2, 4}
4 → {3, 5, 7}
5 → {4, 6}
6 → {2, 5}
7 → {4, 8}
8 → {7}

Ще имаме масив color, който ще има първи ст-сти -1



Изваждане връх 4 от опашката и гледане дали цветовете на посетените му съседи са различни от неговия. Виждаме, че color[4] = color[5] → алгоритъмът приключва и връща лъжа

```
CheckCurrComp(S-стартов, G-срез.списъци, color[1...n])
1. color[s] ← 0
2. Enqueue(Q,s)
3. while !Q.empty()
4.         x ← Dequeue(Q)
5.         for y ∈ adj[x]
6.             if color[y] = -1
7.                 color[y] = !color[x]
8.                 Enqueue(Q,y)
9.             else if color[y] = color[x]
10.                return false
return true
```

```
IsBipartite(G-срез.списъци)
1. for i ← 1 to n
2.         color[i] ← -1
3. for i ← 1 to n
4.     if color[i] = -1
5.         if !CheckCurrComp(i, G, color)
6.             return false
7. return true
```

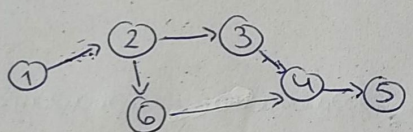
Да се напише псевдокода на DFS

```
DFS(G - чрез спис. на със.)
1. for i←1 to n
2.     π[i]←Nil, color[i]←white
3. time←0
4. for i←1 to n
5.     if color[i]=white
6.         DFS-VISIT(G,i)
```

```
DFS-VISIT(G,s)
1. color[s]←gray
2. time←time+1
3. d[s]←time
4. for y∈adj[s]
5.     if color[y]=white
6.         π[y]←s
7.         DFS-VISIT(G,y)
8. color[s]←black
9. time←time+1
10 f[x]←time
```

зад (5) Даден е ориентиран граф G, представен чрез списъци на съседство. Казваме, че връх $v \in V$ е безопасен връх, ако всеки прост път с начало v е начало на прост път, водещ до сифон.
Да се изведе сортиран (в нарастващ ред) масив, съдържащ всичките безопасни върхове на G

// Уточнение "най-дълъг прост път с начало v"



Нека v=1 ① → ② е прост път, но не е най-дълъг

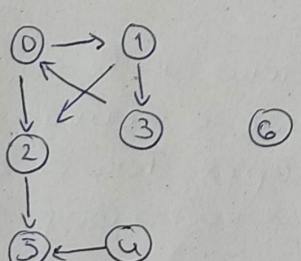
① → ② → ③ → ④ → ⑤ е прост път с дължина 4

① → ② → ⑥ → ④ → ⑤ е прост път с дължина 4

и двата пътя са най-дълги и завършват в сифон

Решение:

// Пример:

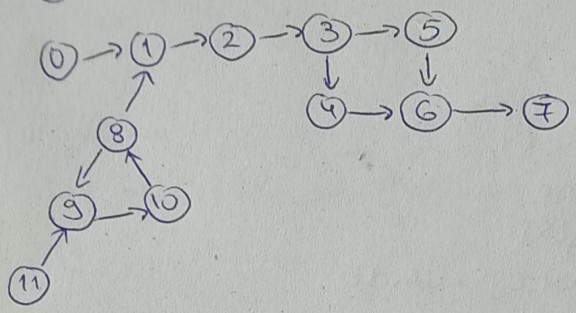


→ тук сигурните върхове са 6, 2, 5, 4

- 0 не е сигурен път, тъй като ⓪ → ① → ③ (①) е прост път с начало 0, който не е начало на прост път водещ до сифон - не може да разширим (①) и да получим прост път
- 1 не е сигурен/безопасен връх, тъй като ① → ③ → ⓪ „—"
- 3 не е безопасен връх, тъй като ③ → ⓪ → ① „—"

Всеки връх от, който има път до цикъл, не е безопасен връх
Всеки друг връх е безопасен връх

// Идея



0 → {1}
1 → {2}
2 → {3}
3 → {4, 5}
4 → {6}
5 → {6}
6 → {7}
7 → {}
8 → {1, 9}
9 → {10}
10 → {8}
11 → {9}

Ще поддържаме два масива - visited, които ще указва посетените върхове и leadsToCycle, които ще указва кои върхове водят до цикъл
В началото и двата масива са запълнени.
При откриването на нов връх v: visited[v] ← 1 ∧ leadsToCycle[v] ← 1.
Когато свършим с даден връх v и не сме стигнали до цикъл: leadsToCycle[v] ← 0

ощe не сме сигурни дали някъде няма да ни отведе до цикъл

DFS(0) ✓
DFS(1) ✓
DFS(2) ✓
DFS(3) ✓
DFS(4) ✓ DFS(5) ✓ → след като се обработи лявото тост
DFS(6) ✓
DFS(7) ✓
сифон

↳ има единствено дете 6, което е обработено - маркирано като безопасно → 5 е безопасен

DFS(8) ✗
↳ НЕ извиква DFS(1), тъй като 1 е обработен връх - маркиран като безопасен. Трябва да проверим 9.
DFS(9) ✗
DFS(10) ✗

↳ има единствено дете 8, което е посетено и в текущия момент leadsToACycle[8] = true е истина, което значи, че 10 не е безопасен връх. Тогава и всеки негов предшественик (връх, който води до 10) не е безопасен

DFS(11) ✗
↳ има единствено НЕбезопасно дете 9 => 11 НЕ е безопасен

В края на алгоритъма: leadsToACycle:

0	0	0	0	0	0	0	0	1	1	1	1
0	1	2	3	4	5	6	7				

Всичките върхове с нулеви стойности са безопасни.

Задача:

```
FindSafeNodes (G - граф сп. на с.)
1. for i ← 1 to n
2.     visited[i] ← 0/false
3.     leadsToACycle[i] ← 0/false
4. for i ← 1 to n
5.     if !visited[i]
6.         DFSCheck(i, G, visited, leadsToACycle)
```

→ проверява дали х „води" до цикъл - връща false, ако НЕ води до цикъл

```
DFSCheck(x, G, visited, leadsToACycle)
1. visited[x] ← 1
2. leadsToACycle[x] ← 1 // Синоним до доказване на противното
3. for y ∈ adj[x]
4.     if !visited[y]
5.         if DFSCheck(y, G, visited, leadsToACycle) → някое от децата на х „води" до цикъл => х „води" до цикъл
6.             return true
7.     else if leadsToACycle[y]
8.             return true
9. leadsToACycle[x] ← 0/false
10. return false
```