Упражнение №5 по САА

Теория на графите – част 1

Основни понятия

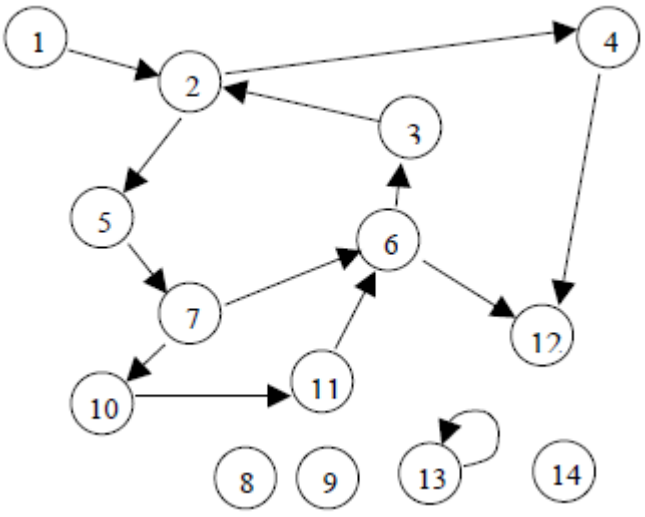
Теорията на графите е клон на съвременната математика, претърпял впечатляващо развитие през последните няколко десетилетия. Графите са едни от най-полезните абстрактни структури от данни и в информатиката. Много задачи от различни области на науката и практиката могат да бъдат моделирани с граф и решени с помощта на съответен алгоритъм върху него. Поради тази причина ще им отделим цяла глава, като до края на тази книга те ще присъстват в още много задачи. Преди да посочим някои примери, ще дефинираме строго понятието.

Дефиниция 5.1. *Краен ориентиран граф* се нарича наредената двойка (V, E), където:

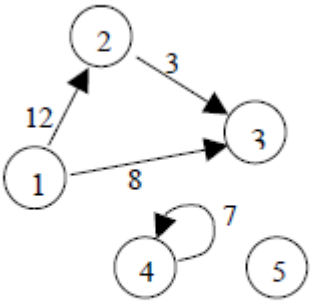
- $V = \{v_1, v_2, ..., v_n\}$ е крайно множество от *върхове*
- $E = \{e_1, e_2, ..., e_m\}$ е крайно множество от *ориентирани ребра*. Всеки елемент $e_k \in E$ $(k=1,2, ..., m)$ е *наредена* двойка (v_i, v_j), $v_i, v_j \in V$, $1 \leq i, j \leq n$.

Ако в допълнение е дадена числова функция $f: E \rightarrow R$, съпоставяща на всяко ребро e_k *тегло* $f(e_k)$, графът се нарича *претеглен*. Понякога, когато няма опасност от двусмислие, ще казваме само ребра вместо ориентирани ребра. Ще отбележим, че някои автори използват термина *дъга* за ориентирано ребро, а *ребро* – само за неориентирано ребро.

Най-често, ориентиран граф се представя графично в равнината чрез множество от точки (кръгчета), означаващи върховете му, и свързващи ги стрелки — ребрата на графа.



Фиг. 1. Ориентиран граф.



Фиг. 2. Ориентиран претеглен граф.

На *фигура 1* е показан граф с 14 върха и 13 ребра. Всеки връх $v_i \in V$ сме изобразили като кръгче и сме му съпоставили номер — уникално естествено число. Така, в случаите, когато това няма да доведе до допълнителни недоразумения, вместо произволно множество V, ще приемем, че V е множеството от първите n естествени числа. Това не намалява общността на разглежданите алгоритми и същевременно води до редица удобства при реализацията — например, можем да използваме върховете като индекси на масив и др.

Всяко ребро $(i,j) \in E$ сме изобразили като стрелка, насочена от връх i към връх j. Забележете, че е допустимо ребро да излиза и влиза в същия връх: например връх 13. Такива ребра се наричат *примки*. Ако графът е претеглен, теглото на всяко ребро ще записваме до съответната стрелка, както на *фигура 2*.

В редки случаи, при разглеждането на някои специфични задачи, е възможно да из ползва- ме и други начини за индексиране на върховете (например с малки латински букви).

Почти всяка съвкупност от обекти с дефинирани връзки между тях може да бъде представена като граф. Ето няколко примера:

1) Да разгледаме транспортната карта на България и по-големите градове. Те могат да бъдат представени като върхове на граф, а преките пътища между тях — като ребра. Теглата на ребрата ще бъдат дължините на преките пътища. Илюстрация на примера е *фигура 3.*, където графът е неориентиран, като е възможно да разглеждаме и ориентиран граф.

2) Компютърна мрежа може да бъде представена чрез неориентиран граф, в който компютрите са върхове, а всяко ребро между два върха показва, че съответните компютри са пряко свързани в мрежа.

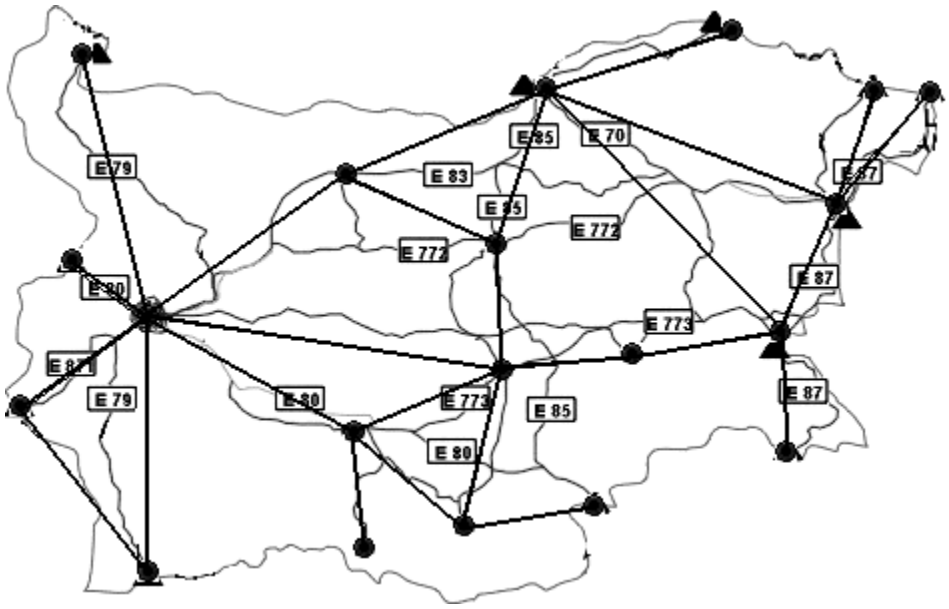
3) Множеството от страниците в Интернет и връзките помежду им могат да бъдат представени като ориентиран граф.

4) Няколко химични съединения могат да бъдат представени като върхове на неориентиран граф: Всяко ребро ще показва дали съответните химични съединения могат да си взаимодействат. Аналогичен пример е върховете да представят видове декоративни риби, а ребрата да показват дали два вида риби могат да съжителстват заедно, или — не.

5) Етапите в изработването на едно изделие могат да се представят като върхове на ориентиран граф. Ребрата показват кой етап кого предхожда в процеса на изработване на изделието.

Могат да бъдат посочени още много примери за графи, както и да се формулират задачи върху тях. Така например, може да се интересуваме от най-късия или най-евтиния път между два града в пример 1), или от минималното време за изработване на цялото изделие в пример 5).

За пълноценното разглеждане на материала по-нататък е необходимо да дефинираме още някои понятия от теория на графите. Не е нужно читателят да се опитва да запомни всички дефиниции сега. Достатъчно е при разглеждането на съответната част по-късно той да се върне тук и да си припомни, каквото му е необходимо.



Фиг. 3. Транспортна карта на България.

Дефиниция 5.3. Даден е ориентиран (неориентиран) граф $G(V,E)$. Ако в множеството на ребрата му се допуска повторение (т.е. E е *мултимножество*), G се нарича *мултиграф*.

Забележете, че ако мултиграфът е претеглен, то на различните ребра (i,j) се съпоставят от-делни (в общия случай различни) тегла $f(1)(i,j)$, $f(2)(i,j)$, ...

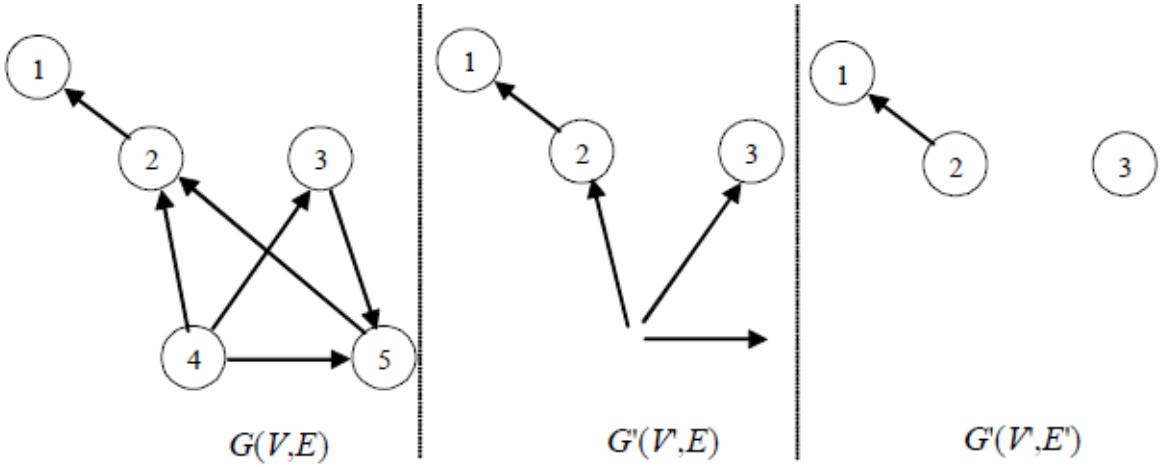
Дефиниция 5.4. Даден е ориентиран граф $G(V,E)$. Два върха i и j ($i,j \in V$) се наричат *съсед-ни*, ако поне едно от ребрата (i,j) и (j,i) принадлежи на E. В такъв случай още казваме, че i и j са *краища* за ребрата (i,j) и (j,i). За всяко ребро (i,j) върхът i се нарича *предшественик* на j, а j — *наследник* на i. Всеки от върховете i и j се нарича *инцидентен* с реброто (i,j). Казваме, че две ребра са *инцидентни*, когато са инцидентни с един и същ връх. В случай на неориентиран граф, понятията *съседен* връх, *край* на ребро, *инцидентност* на върхове и ребра се дефинират аналогично.

Дефиниция 5.5. Даден е ориентиран граф $G(V,E)$. *Полустепен на изхода на връх* i, $i \in V$ се нарича броят на ребрата (i,j), $j \in V$. Аналогично, броят на ребрата (j,i), $j \in V$ се нарича *полустепен на входа на* i. Сборът от полустепента на входа и полустепента на изхода се нарича *степен* на върха i. *Изолиран* се нарича връх, чиято степен е 0. При неориентиран граф степен на връх i се нарича броят на ребрата (i,j), инцидентни с него.

На *фигура 5.1а.* степента на връх *2* е 4, а връх *14* е изолиран.

Дефиниция 5.6. Нека е даден ориентиран (неориентиран) граф $G(V, E)$ и $V' \subseteq V$. Ако от E се изключат всички ребра (i, j), такива че $i \in V'$ или $j \in V'$, а останалите се запазят, то

казваме, че $G'(V',E')$ е *индуциран от V' подграф* (или само *подграф*) на G (*виж фигура 5.1д.*).

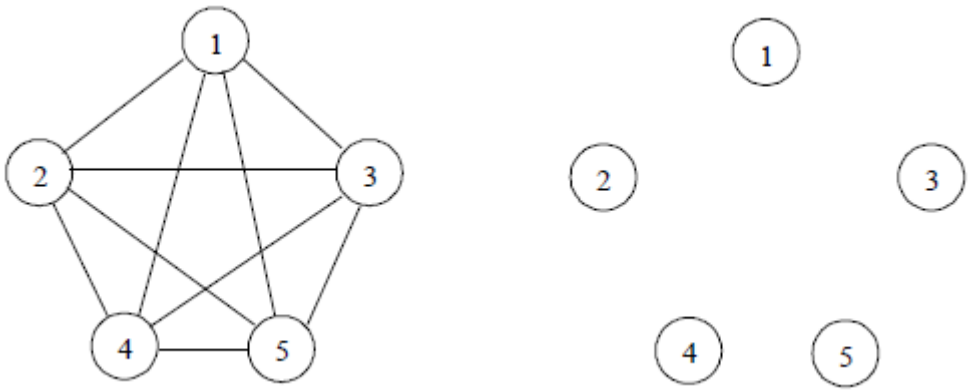


Фиг. 4. Индуциран от множеството V' = {1, 2, 3} подграф G' на G.

Дефиниция 5.7. *Път* в ориентиран (неориентиран) граф $G(V,E)$ се нарича последовател- ност от върхове $v1$, $v2$,..., vk такава, че за всяко $i = 1, 2, \ldots, k–1$ е изпълнено $(vi, vi+1) \in E$. Върховете $v1$ и vk се наричат *краища* на пътя. Ако $v1 = vk$, пътят се нарича *цикъл*. Ако за всяко $i \neq j$ $(1 \leq i, j \leq k)$ следва $vi \neq vj$, то пътят се нарича *прост*. Съответно, ако $v1 = vk$, и всички останали върхове са различни, цикълът се нарича *прост*. Когато граф съдържа поне един цикъл, той се нарича *цикличен*, а в противен случай — *ацикличен*.

Дефиниция 5.8. Нека са дадени графите $G(V, E)$ и $G'(V, E')$, такива че за всяко $i, j \in V$, $i \neq j$, наредената двойка (i, j) принадлежи на точно едно от множествата E' или E. Тогава G' се нарича *допълнение* на G. Граф, несъдържащ ребра, се нарича *празен* (*фигура 5.1е.*). Ако за даден граф е изпълнено $(i, j) \in E$ за всяко $i, j \in V$, $i \neq j$, графът се нарича *пълен*.

Забележете, че последната дефиниция е симетрична, т.е. ако G' е допълнение на G, то и G е допълнение на G'.



Фиг. 5. Пълен и празен неориентиран граф.

Дефиниция 5.9 Ако граф $G'(V', E')$ с t върха е индуциран от G, и G' е пълен, казваме че G' е *t-клика на G*. Максималното t, за което G има *t-клика* се нарича *кликово число* за G.

Дефиниция 5.10. Ориентиран граф се нарича *слабо свързан*, ако всеки два върха i и j са краища на поне един път (т.е. съществува поне един път от i до j или от j до i). Когато в ориенти- ран граф за всеки два върха i, jсъществува път както от iдо j, така и от jдоi, то графът се нарича *силно свързан*. Неориентиран граф се нарича *свързан*, ако съществува път между всяка двойка не-гови върхове i, j.

Дефиниция 5.11. *Диаметър* на граф се нарича максималното число k, такова че най-крат-кия прост път (пътят, в който участват минимален брой върхове) между произволни два върха $i,j \in V$ съдържа поне k върха.

Така например, ако разглеждаме страниците в Интернет и връзките помежду им като ори-ентиран граф, то диаметърът му k е максималният брой страници, през които трябва да преминем, започвайки от произволна страница и следвайки хипервръзки, за да достигнем до произволна друга страница в мрежата. Проучванията показват, че "диаметърът на Интернет" е приблизително равен на 19 [*Web-d*].

Дефиниция 5.12. *Силно* (*слабо*) *свързана компонента* в ориентиран граф $G(V, E)$ се нарича всеки *подграф* $G'(V', E')$, за който е изпълнено:

1) G' е силно (слабо) свързан граф.

2) Не съществува друг силно (слабо) свързан собствен подграф $G''(V'',E'')$ на G, такъв, че G' да бъде подграф на G''.

Аналогично се дефинира *свързана компонента* в неориентиран граф.

Дефиниция 5.13. Неориентиран свързан граф без цикли се нарича *дърво*. Ако допълни-телно изберем някой връх от дървото за корен, то получената структура се нарича *кореново дърво*.

Дефиницията е алтернативна на разгледаната в *2.3.* дефиниция на кореново дърво).

Дефиниция 5.14. *Покриващо* (*обхващащо*) *дърво* в свързан неориентиран граф $G(V,E)$ се нарича всеки свързан ацикличен подграф $G'(V, E')$ на G.

Представяне и прости операции с граф

Досега използвахме два начина за представяне на граф:

- Директно следвайки дефиницията — чрез дадените множества V и E.
- Графично: чрез множество от точки (кръгчета) и връзки (стрелки) между тях.

Второто представяне (макар и нагледно) е неприложимо, а първото — често неудобно, за компютърно обработване. Затова съществуват други начини за представяне на граф — те ще бъдат предмет на настоящия параграф. Изборът на най-подходящото

компютърно представяне зависи от конкретната задача — добре е да се използва такова, при което операциите, които се извършват най-често, да имат по-малка алгоритмична сложност (или да е необходимо по-малко памет).

Списък на ребрата

Най-близко до дефиницията на граф е представяне със списък от ребрата му. Ориентиран граф представяме като списък от наредени двойки (i,j). В случай на неориентиран граф, двойките са ненаредени. При претеглен (ориентиран) граф разглеждаме (наредени) тройки (i,j,k), където реалното число k е теглото $f(i,j)$ на съответното ребро. Последното бихме могли да реализираме на Си, като използваме масив float A[3][m], където m е броят на ребрата на графа.

Очевидно, при това представяне необходимата памет ще бъде $\Theta(m)$, каквато е и сложността на проверката дали два върха са съседни, което се счита за една от най-често извършваните операции. В случай, че сортираме ребрата, ще можем прилагаме двоично търсене и така да намалим сложността на проверката до $\Theta(\log 2m)$. Както ще видим по-долу, съществуват представяния, при които последната сложност е доста по-малка: функция на n, и дори константа. Така, това представяне е приложимо предимно за *разредени* графи, т.е. графи, които имат относително малък брой ребра: $m \in O(n)$.

Матрица на съседство, матрица на теглата

Матрицата на съседство е един от най-често използваните начини за представяне на граф в паметта. При него на ориентиран граф с n върха се съпоставя квадратна матрица A[n][n]. Стойността на A[i][j] е равна на 1, когато съществува реброто (i,j), и A[i][j] = 0 □ в противен случай. За удобство при реализирането на някои алгоритми, когато не съществува реброто (i, j), но съществува (j, i), на позиция A[i][j] в матрицата се записва стойност –1 (и съответно 1 □ на A[j][i]).

Ако графът е неориентиран, то A[i][j] = 1, когато съществува реброто (i, j) или (j, i), и A[i][j] = 0 в противен случай. При непретеглен мултиграф, в полето A[i][j] записва не 1, а броят на ребрата между върховете i и j. Когато графът е претеглен, на позиция A[i][j] се записва теглото $f(i, j)$, и матрицата A[][] се нарича *матрица на теглата*.

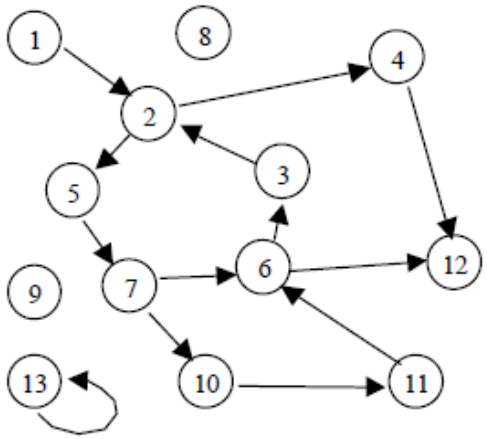
В някои задачи се дефинират и тегла за *върховете* на графа. Те могат да бъдат записани на позициите A[i][i] за всеки връх i от графа. Тези полета от матрицата няма да бъдат заети, единствено в случай, че графът не съдържа примки (ребра от вида (i,i)).

При неориентиран граф матрицата на съседство е *симетрична* относно главния си диагонал. В този случай може да се приложи по-специфично запазване в паметта (например линеаризиране, с цел да се спести дублиращата се част, която заема излишно повече клетки памет [*Рахнев, Гъров, Гаврилов-1995*]. 2) 1(□n n

Важно преимущество на представянето чрез матрица на съседство е, че проверката за съществуване на ребро между два върха става с единствена операция. От друга страна намирането на всички *наследници* на даден връх има сложност □(n), независимо от това

колко наследника има върхът (т.е. дори върхът да има единствен наследник, пак ще трябва да се извършат всичките *n* проверки). Пример за ориентиран граф и съответстващата му матрица на съседство е даден на *фигура 6*. Забележете, че докато графичната номерация на върховете започва от 1, то при реализация на Си индексирането на елементите започва от 0. Тази специфична особеност трябва да се има в предвид при разглеждането на всички реализации на алгоритми от настоящата глава.

	0	1	2	3	4	5	6	7	8	9	10	11	12
0	0	1	0	0	0	0	0	0	0	0	0	0	0
1	-1	0	-1	1	1	0	0	0	0	0	0	0	0
2	0	1	0	0	0	-1	0	0	0	0	0	0	0
3	0	-1	0	0	0	0	0	0	0	0	0	1	0
4	0	-1	0	0	0	0	1	0	0	0	0	0	0
5	0	0	1	0	0	0	-1	0	0	0	-1	1	0
6	0	0	0	0	-1	1	0	0	0	1	0	0	0
7	0	0	0	0	0	0	0	0	0	0	0	0	0
8	0	0	0	0	0	0	0	0	0	0	0	0	0
9	0	0	0	0	0	0	-1	0	0	0	1	0	0
10	0	0	0	0	0	1	0	0	0	-1	0	0	0
11	0	0	0	-1	0	-1	0	0	0	0	0	0	0
12	0	0	0	0	0	0	0	0	0	0	0	0	1



Фиг. 6. Матрица на съседство и съответният ѝ граф.

Списък на наследниците (списък на инцидентност)

При това представяне за всеки връх *i* от графа се пази списък с наследниците му. За графа от *фигура 6.* представянето ще има следният вид:

1 → {*2*}; *2* → {*4,5*}; *3* → {*2*}; *4* → {*12*}; *5* → {*7*}; *6* → {*3,12*}; *7* → {*6,10*}; *8* → {};
9 → {}; *10* → {*11*}; *11*→{*6*}; *12* → {}; *13* → {*13*}.

За компютърната реализация най-често се използват *динамични свързани списъци*. При тях сложността за намиране на всички наследници на даден връх е линейна по броя на наследниците: предимство пред матрицата на съседство, където намирането на наследниците е със сложност □(*n*). Недостатъкът обаче е, че проверката дали съществува ребро между два върха е също със сложност, линейно зависеща от броя на наследниците.

Възможна е и статична реализация на това представяне: едномерен масив с големина броя на наследниците за всеки един от върховете на графа. В този случай проверката дали има ребро между два върха се извършва значително по-ефективно — с логаритмична сложност. Това се постига, като се пазят наследниците в сортиран вид и се използва двоично търсене.

Ясно е, че както можем да поддържаме списък на наследниците, така можем да построим и *списък на предшествениците*, като в този случай сложностите на основните операции ще бъдат същите. На практика представяне на граф чрез списък на предшествениците се използва много рядко.

Построяване и прости операции с графи

Основните операции, свързани с построяване и модифициране на граф, са следните:

- *създаване* на празен граф.
- *добавяне/премахване* на връх.
- *добавяне/премахване* на ребро.

По-нататък следват операциите, за които стана въпрос и в предходните точки:

- *проверка за съществуване на връх*
- *проверка за съществуване на ребро*
- *намиране на наследниците на даден връх*

Няма да уточняваме как се реализират тези операции при всеки един вид представяне в паметта. Като пример ще покажем реализация за граф, представен с *матрица на съседство* (*ма-трица на теглата*).

```
/* Максимален брой върхове в графа */
#define MAXN 200

/* Брой върхове в графа */
unsigned n;

/* Матрица на теглата на графа */
int A[MAXN][MAXN];

/* Модифициране на теглото на върха i */
A[i][i] = k;

/* Добавяне на ребро с тегло k, свързващо върховете i и j */
A[i][j] = k;
A[j][i] = k; /* ако графът не е ориентиран */

/* Премахване на ребро, свързващо върховете i и j */
A[i][j] = 0;

/* Проверка дали има ребро между върховете i и j */
if (A[i][j] != 0) { /* има ребро */; }
else { /* няма ребро */; }

/* Намиране на всички наследници на върха i */
for (k = 0; k < n; k++) if (k != i) {
if (A[i][k] != 0) { /* върхът k е наследник на i */ ; }
}
```

Обхождане на граф

Под *обхождане* на неориентиран (ориентиран) граф ще разбираме последователно *посе-щаване* (разглеждане) на всеки връх от графа точно по веднъж. *Стратегията за*

обхождане определя реда, в който ще се разглеждат върховете.

Съществуват две фундаментални стратегии за обхождане на граф (предмет на следващите две точки), които са известни като *обхождане в ширина* и *обхождане в дълбочина*. Изключител-ната важност на тези два алгоритъма се обуславя от:

- Широкия периметър от задачи, в които те намират приложение.
- Възможността да се решават сложни алгоритмични проблеми, с прилагане на малки модификации на основните алгоритми.
- Добрата алгоритмична сложност, която се постига в повечето случаи.

Обхождане в ширина

Обхождането в ширина от даден връх i ще наричаме следната стратегия за обхождане на граф: започваме от върха *i*, разглеждаме всички негови непосредствени *съседи*, и едва след това преминаваме към по-нататъшно обхождане (*обхождане в ширина от всеки един от съседите му*). По този начин се постига последователно обхождане “по нива”, започвайки от стартовия връх, докато се обходят всички върхове на графа, достижими от *i*. *Обхождане в ширина на граф* наричаме преминаването през всички негови върхове по описания начин — т.е. последователно избиране на (произволен) стартов връх, докато всички върхове на графа не бъдат обходени.

От сега нататък, ще използваме съкращението *BFS*(*i*) (от англ. *Breadth-First-Search*), за да означаваме функцията за обхождане в ширина от връх *i*.

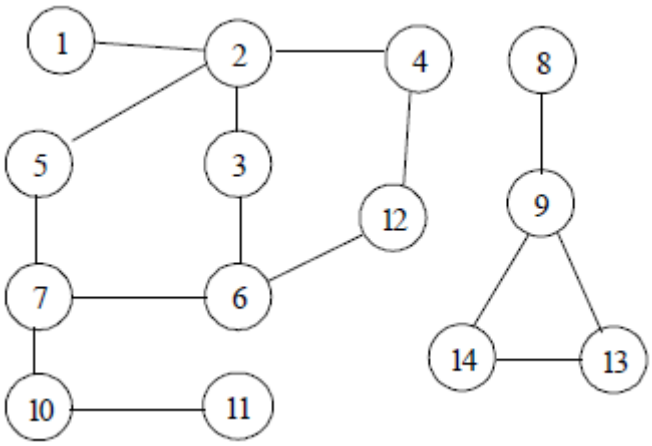
Като пример ще приложим описаната стратегия върху графа от *фигура 7*.

Ако за стартов връх изберем *1*, резултатът от обхождането ще изглежда по следния начин:

BFS(1):
ниво 1: *1*
ниво 2: *2*
ниво 3: *3, 4, 5*
ниво 4: *7, 6, 12*
ниво 5: *10*
ниво 6: *11*

Ако вместо *1* изберем *3* за начален връх, при изпълнението на *BFS*(*3*) получаваме:

BFS(3):
ниво 1: 3
ниво 2: 2, 6
ниво 3: 1, 4, 5, 7, 12
ниво 4: 10
ниво 5: *11*



Фиг. 7. Неориентиран граф.

Веднага се забелязват някои възможни приложения на обхождането в ширина. Например, да се намери минималният по брой участващи върхове път между два върха или да се намерят *компонентите на свързаност* на граф. Тези и други примери ще разгледаме малко по-нататък.

Обхождането в ширина намира косвено приложение и в други задачи, при които се търси минимално разстояние. Така например, *методът на вълната* (*виж задача 5.110.*), приложим при търсене на минимален път между две клетки на матрица и др., може да се разглежда и като обхождане на специален вид граф в ширина.

Реализация на обхождане в ширина от даден връх

Ще представим графа с матрица на съседство, като това представяне ще използваме при почти всички задачи от графи. То е достатъчно ефективно при повечето задачи и същевременно не нарушава четимостта на кода с динамично заделяне на памет, обемисти реализации на сложни структури от данни и др.

Разбира се, както при обхождането в ширина, така и при задачите, разглеждани по-нататък, ще се стараем да представяме на читателя винаги най-добрите (от алгоритмична гледна точка) известни до момента решения.

В процеса на обхождане в ширина се налага да намираме наследниците на даден връх i. От таблица *5.2.6.*, се вижда, че ако използваме матрица на съседство, сложността ще бъде $\Theta(n)$. Това е така, тъй като за всеки от n-те върха на графа се извършва проверка дали е наследник на i (за сравнение, при реализация със списък на наследниците можем да получим директно наследниците на i чрез обхождане на съответния списък). Така общата сложност на обхождането е $\Theta(m+n)$ — при представяне със списъци на съседство, и $\Theta(n2)$ — при представяне с матрица на съседство.

Ще поддържаме опашка, в която първоначално се намира единствено стартовият връх. След това, докато в опашката има поне един връх, извършваме следното: изваждаме върха, намиращ се в началото на опашката, разглеждаме го и добавяме в опашката всички негови непосетени до момента наследници. Върховете ще маркираме като посетени в момента, в който ги добавяме в опашката:

```
BFS(i)
{ Създаваме_празна_опашка_Queue;
  Добавяме_към_опашката_върха_i;
  for (k = 1,2,…,n) used[k] = 0;
  used[i] = 1;
  while (Опашката_не_е_празна) {
    p = Извличаме_елемент_от_началото_на_опашката;
    Анализираме_върха_p;
    for (за всеки наследник j на p)
      if (0 == used[j]) { /* ако върхът j не е обходен */
        Добавяме_към_опашката_върха_j;
        used[j] = 1; /* маркираме j като обходен */
      }
  }
}
```

Следва пълна реализация. Входните данни, използвани в реализацията по-долу, са зададе-ни като константи в началото на програмата и са за графа, показан на *фигура 7*. Броят върхове на графа е n, а A[MAXN][MAXN] е матрицата му на съседство. Стартовият за обхождането връх се задава с константата v.

В показания пример графът е неориентиран, но програмата по-долу ще приложи правилно алгоритъма и в случаите на ориентиран граф.

```
#include <stdio.h>
/* Максимален брой върхове в графа */
#define MAXN 200
/* Брой върхове в графа */
const unsigned n = 14;
/* Обхождане в ширина с начало връх v */
const unsigned v = 5;
/* Матрица на съседство на графа */
const char A[MAXN][MAXN] = {
{0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
{1, 0, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0},
{0, 1, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0},
{0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0},
{0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0},
{0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0},
{0, 0, 0, 0, 1, 1, 0, 0, 0, 1, 0, 0, 0, 0},
{0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0},
{0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 1},
{0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0},
{0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0},
{0, 0, 0, 1, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0},
{0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1},
{0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 1, 0}
};
char used[MAXN];
/* Функция за обхождане в ширина от даден връх */
void BFS(unsigned i)
{
	unsigned k, j, p, queue[MAXN], currentVert, levelVertex, queueEnd;
	for (k = 0; k < n; k++) queue[k] = 0;
	for (k = 0; k < n; k++) used[k] = 0;
	queue[0] = i; used[i] = 1;
	currentVert = 0; levelVertex = 1; queueEnd = 1;
	while (currentVert < queueEnd) { /* докато опашката не е празна */
		for (p = currentVert; p < levelVertex; p++) {
			/* p – вземаме поредния елемент от опашката */
			printf("%u ", queue[p]+1);
			currentVert++;
			/* за всеки необходен наследник j на queue[p] */
			for (j = 0; j < n; j++)
				if (A[queue[p]][j] && !used[j]) {
					queue[queueEnd++] = j;
					used[j] = 1;
				}
		}
		printf("\n");
		levelVertex = queueEnd;
	}
```

```
}
int main(void) {
	printf("Обхождане в ширина от връх %u: \n", v);
	BFS(v-1);
	return 0;
}
```

Резултат от изпълнението на програмата:

```
Обхождане в ширина от връх 5:
5
2 7
1 3 4 6 10
12 11
```

Задачи за упражнение:

1. Горната програма реализира алгоритъм за обхождане на граф в ширина от даден връх. В кои случаи, изпълнена за произволен връх, тя ще обходи *всички* върхове от графа?
2. Как трябва да се модифицира програмата така, че в случаите, когато останат необходени върхове, да се изпълнява ново обхождане в ширина върху тях?

Обхождане в дълбочина

Обхождането в дълбочина (ще съкращаваме с *DFS* — от англ. *Depth-First-Search*) наред с това, че представлява неразделна част от някои по-сложни алгоритми, е основна идея при един от фундаменталните методи за решаване на изчерпващи задачи — *търсене с връщане* (на англ. *backtracking*).

Докато обхождането в ширина разглежда върховете на графа *последователно* по нива, об-хождането в дълбочина от даден връх се стреми да “се спусне” колкото се може “по-надълбоко” при обхождането. Алгоритъмът за обхождане на граф в дълбочина се описва най-лесно рекур-сивно: започваме от избрания начален връх $i \in V$, маркираме го като посетен и продължаваме ре-курсивно обхождането в дълбочина за всеки негов непосетен наследник, т.е. функцията за обхождане в дълбочина от връх i ($DFS(i)$) изглежда по следния начин:

1) Разглеждаме i.
2) Маркираме i като обходен.
3) За всяко инцидентно с i ребро $(i,j) \in E$ такова, че j е необходен връх от графа, изпълняваме рекурсивно $DFS(j)$.

Ще приложим *DFS* върху примера от *фигура 5.3.1*. В случаите, когато повече от едно ребро е инцидентно с разглеждан връх (стъпка 2), ще продължаваме последователно от върховете с по-малък към върховете с по-голям номер. Така резултатът от $DFS(1)$ ще бъде: *1, 2, 3, 6, 7, 5, 10, 11, 12, 4*, а от $DFS(3)$: *3, 2, 1, 4, 12, 6, 7, 5, 10, 11*.

Програмата, извършваща обхождането, ще реализираме директно по описания по-горе начин (чрез рекурсивната функция DFS(i)). В булев масив used[] ще маркираме обходените върхове: в началото го инициализираме с нули, а при посещаване на върха *i*

извършваме присвояването used[i] = 1.

```
#include <stdio.h>
/* Максимален брой върхове в графа */
#define MAXN 200
/* Брой върхове в графа */
const unsigned n = 14;
/* Обхождане в дълбочина с начало връх v */
const unsigned v = 5;
/* Матрица на съседство на графа */
const char A[MAXN][MAXN] = {
	{0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
	{1, 0, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0},
	{0, 1, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0},
	{0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0},
	{0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0},
	{0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0},
	{0, 0, 0, 0, 1, 1, 0, 0, 0, 1, 0, 0, 0, 0},
	{0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0},
	{0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 1},
	{0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0},
	{0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0},
	{0, 0, 0, 1, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0},
	{0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1},
	{0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 1, 0}
};
char used[MAXN];
/* Обхождане в дълбочина от даден връх */
void DFS(unsigned i)
{
	unsigned k;
	used[i] = 1;
	printf("%u ", i+1);
	for (k = 0; k < n; k++)
		if (A[i][k] && !used[k]) DFS(k);
}
int main(void) {
	unsigned k;
	for (k = 1; k < n; k++) used[k] = 0;
	printf("Обхождане в дълбочина от връх %u: \n", v);
	DFS(v-1);
	printf("\n");
	return 0;
}
```

Резултат от изпълнението на програмата:

```
Обхождане в дълбочина от връх 5:
5 2 1 3 6 7 10 11 12 4
```

Литература

1. Програмиране = ++Алгоритмни, Наков, Добринков – Можете да свалите безплатно книгата на адрес: http://www.programirane.org