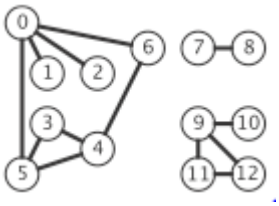
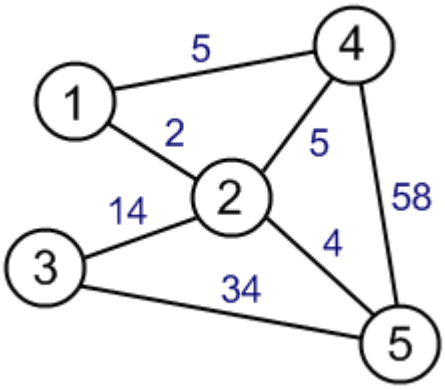
Графи

Графът е структура от данни изградена от върхове (възли) и връзки между тях, които се наричат ребра. Ако ребрата имат посока графът е ориентиран, в противен случай е неориентиран. Ребрата могат да имат и тегло (или дължина). Тогава се казва, че графът е претеглен. Свързаните списъци и дърветата са частни случаи на графи.

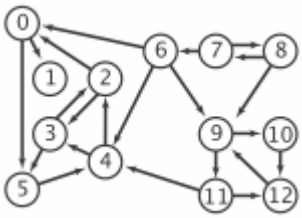
Видове



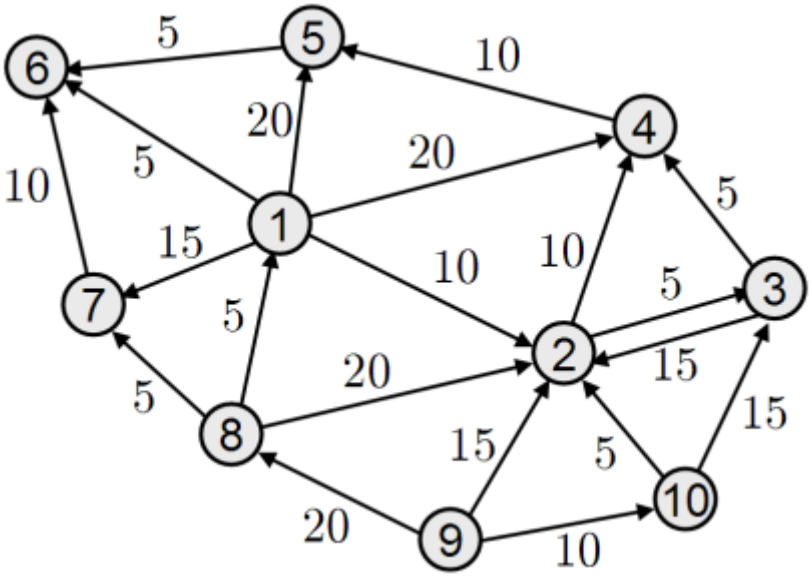
Неориентиран



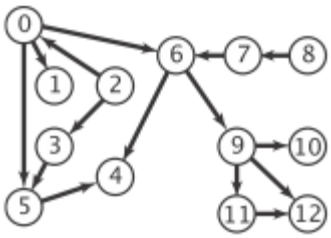
Неориентиран, претеглен



Ориентиран (насочен)

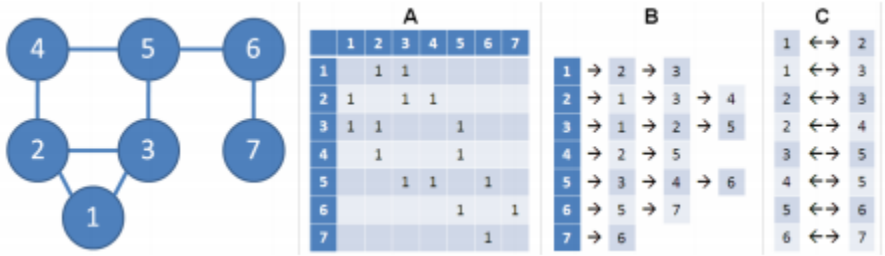


Ориентиран, претеглен



*Насочен, ацикличен граф – DAG (**D**irected **A**cyclic **G**raphs) е граф, в който няма цикли*

Представяне



Матрица на съседство (Adjacency-matrix)

Един възможен начин за представяне на граф е чрез **матрицата на съседство**. Тя се състои от **V** реда и **V** стълба, където **V** е броят на върховете в графа. Всеки ред и стълб съответстват на конкретен връх. Ако има ребро между връх **u** и връх **v** то елементът на позиция [**u**][**v**] е 1, а ако няма ребро - 0. Ако графа е неориентиран, то тогава маркираме и клетка [**v**][**u**] с 1. Такова представяне на графите изисква $\mathbf{V}^2$ памет, както и повечето алгоритми, ползващи такава матрица също са квадратични.

Списък на съседите (Adjacency-lists)

Ако забелязвате в матриците на съседство има много елементи със стойност 0. Такива матрици се наричат разредени. При големи графи тези матрици заемат много излишна памет, затова е по-удачно да използваме динамична памет за представянето на графите. Това може да стане, чрез **списъци на съседство**. За всеки връх от графа има списък, в който се намират номерата на съседните му върхове. Списъкът на съседство представлява масив (да го кръстим G) от свързани списъци(или динамични масиви), където G[**u**] съдържа всички съседни върхове на **u**.

Това представяне се използва много често, защото е подходящо за разредени графи, тъй като използва по-малко памет от матрицата на съседство, макар и операциите триене и добавяне на ребро да са по-бавни.

Също така е възможно всеки елемент на G да се поддържа в сортиран вид, което е удобно, ако G[**u**] е динамичен масив, защото тогава за проверка за съседство на **u** и **v** може да се направи двоично търсене в G[**u**], което води до операция със логаритмична сложност.

Списък на ребрата (Array of edges)

При това представяне поддържаме списък на ребрата, обикновено в някакъв ред. Ползва се по-рядко, най-вече при алгоритъм на Крускал за **MST**.

Сложности

	array of edges	adjacency matrix	adjacency lists
space	E	V^2	$V + E$
initialize empty	1	V^2	V
copy	E	V^2	E
destroy	1	V	E
insert edge	1	1	1
find/remove edge	E	1	V
is v isolated?	E	V	1
path from u to v?	$E \lg^* V$	V^2	$V + E$

Worst-case cost of graph-processing operations

Обхождане

Ще разгледаме две широко използвани алгоритмични схеми за обхождане на графи. Това са обхождане в дълбочина и обхождане в широчина. В представените програмни реализации графът се представя чрез ***списък на съседите***. При такова представяне и двете алгоритмични схеми имат алгоритмична сложност **O(V + E)**, където **V** е броят на върховете в графа, а **E** е броят на ребрата му. Ако за представянето на графа използваме матрица на съседство, то сложността им ще бъде **O(V^2).**

За представянето на графа, чрез списък на съседите ще използваме динамичният масив **vector** от **STL**. По-точно ще използваме **vector** от **vector**-и:

```
#define MAXV 1000

vector < vector < int > > Graph(MAXV);
vector < vector < pair < int, int> > > Graph(MAXV); // при претеглен граф
```

Обектът Graph е масив от масиви с големина MAXV. Неговото значение е следното: в `Graph[i]` (това е динамичен масив) ще съхраняваме всички наследници на върха `i`. Изчитаме графа, с функция подобна на показаната:

```
void read()
{
	int from, to;
	//V - брой върхове, E - брой ребра
	scanf("%d%d", &V, &E);
	for (int i = 0; i < E; i++)
	{
		scanf("%d%d", &from, &to);
		from--, to--;
		Graph[from].push_back(to);
		Graph[to].push_back(from); //if the graph is b-directional
```

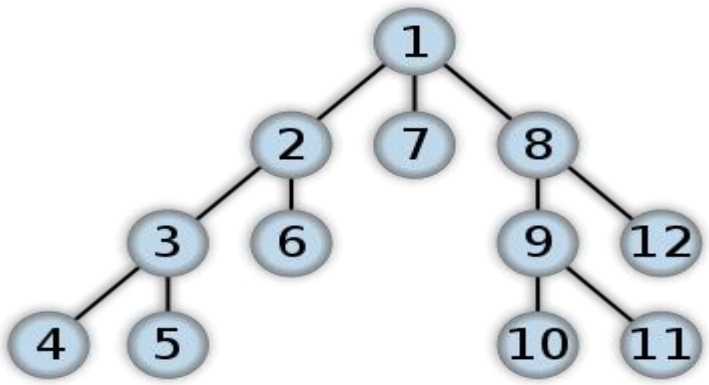
```
	}
}
```

Обхождане в дълбочина (Depth-first search)

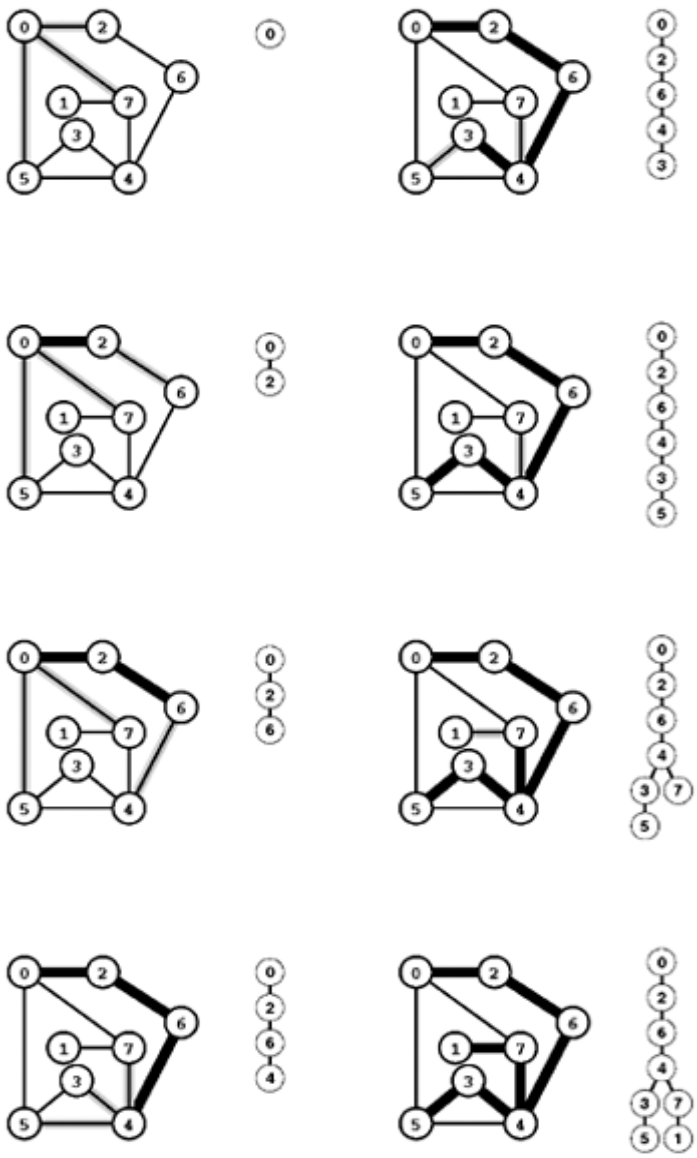
При обхождането в дълбочина се тръгва от някакъв начален връх и се преминава към негов необходен наследник, след което към наследник на наследника и така, докато се стигне връх без наследници (от който не излизат ребра). Тогава се прави връщане назад (*backtrack*) към предишния връх.

Обхождането в дълбочина от даден връх се стреми да се спуска, колкото се може по-надълбоко, в процеса на обхождането, за разлика от обхождането в широчина, където върховете са обхождат по нива. Идеята е рекурсивно до обхождаме всеки непосетен съсед на текущия връх. Ето как би изглеждало обхождане в дълбочина на граф, започвайки от най-горния връх:

```
void dfs(int startV)
{
	printf("%d ", startV + 1);
	viz[startV] = true; // маркираме текущия връх като обходен
	// разглеждаме всички негови съседи
	for (int i = 0; i < Graph[startV].size(); i++)
	{
		const int toVisit = Graph[startV][i];
		if(!viz[toVisit]) // ако поредния съсед не е бил посещаван
			dfs(toVisit); // рекурсия
	}
}
```



Върховете са номерирани, според реда в който ще бъдат посетени



Топологическо сортиране

Топологическо сортиране на **ориентиран ацикличен граф** (DAG), наричаме такава подредба на върховете му, за която е изпълнено, че ако съществува път от *i* до *j*, то*i* трябва да предшества *j*. Топологическото сортиране използва алгоритмичната схема обхождане в дълбочина. Идеята е на всяка стъпка да търсим връх без съседи и така ще получим топологическото сортиране в обратен ред, използвайки обратния ход на рекурсията.

```
void dfs(int from)
{
	//маркираме текущия връх за посетен
	vis[from] = true;

	//за всеки негов непосетен съсед
	for(int i = 0;i < p[from].size();i++)
		if(!vis[p[from][i]])
			dfs(p[from][i]);//стартираме обхождане в дълбочина

	//при изплуване от рекурсията запазваме стартовия връх
	topol.push_back(from);
}

void solve()
{
	for(int i = 0;i < n;i++)
		if(!vis[i])//за всеки непосетен връх
			dfs(i);//стартираме обхождане в дълбочина
}
```

Примерна задача – Родословно дърво

Родствени връзки на планетата ФМИ са доста объркани. Те се събират в групи, така че фмитяните могат да имат както един, така и десет родителя и никои не е учуден от стотина деца. Фмитяните са свикнали с техния начин на живот, и го намират за много естествен, но в планетарния парламент това объркано родословно дърво води до объркване. Там се събират най-достойните фмитяни и затова, за да не се засегне някои по време на дискусии е решено първо да се дава думата на най-възрастните фмитяни, след това на по-младите и най-накрая на най-младите и бездетни фмитяни. Разбира се спазването на това решение не е тривиална задача. Не винаги фмитяните познават всичките си родители. Ако по грешка, някой говори преди родителите си или прародителите си, тогава става голям скандал. Задачата е да се напише програма която, да определи реда на изказване във фмитянския парламент, и да прекратят скандалите веднъж за винаги.

Входа съдържа множество тестови примери. Първия ред на всеки от тях съдържа числото N ($1 \le N \le 10\,000$) – броя на членовете на фмитянския парламент. Според вековните традиции членовете на парламента са номерирани с естествените числя от 1 до N. Следват точно N реда, като i-тия ред съдържа списък с децата на i-тия член на парламента. Списъка с децата с поредица от номерата на децата в произволен ред разделени с интервали. Списъкът може да е празен. Списъкът, дори и празен, завършва с 0. Ако N е извън обявените граници това означава край на входа.

За всеки от тестовите примери изведете по една единствена линия, списък с членовете на фмитянския парламент по реда на изказване, разделени с интервали. Ако задачата има няколко решения, отпечатайте кое да е от тях. Поне едно решение винаги съществува.

Вход	Изход
4	2 4 3 1

0 4 1 0 1 0 3 0 -45	

Свързани компоненти в граф

Свързани компоненти в ***неориентиран*** граф са такова подмножества от свързани помежду си възли, като всяко от тях не е свързано с друго множество, т.е. различните компоненти на графа не са свързани по никакъв начин. В една свързана компонента има път (директен или индиректен) между всеки два възела. Това означава, че с алгоритъма за намиране на свързани компоненти можем да отговорим на въпроса, дали даден граф е свързан. Отговора е да, ако той съдържа само една свързана компонента, т.е. в него има път между веки два върха.

Примерна задача – Приятели

В един град живеят N жители, за някои двойки от които се знае, че са приятели. От известната максима "Приятелите на моите приятели са и мои приятели" следва, че ако A и B са приятели, и B и C са приятели, то A и C също са приятели. Напишете програма, която намира броя на гражданите в най-голямата група от приятели.

На първия ред на стандартния вход е зададен броят на тестовете. Всеки тест започва с ред, на който са зададени числата N и M, където N е броят на жителите ($10 \le N \le 10000$), а M е броят на двойките, за които се знае, че са приятели ($0 < M \le 50000$). На всеки от следващите M реда има по две числа – номерата на двойка приятели A, B ($1 \le A \le N$, $1 \le B \le N$, $A \ne B$), като между дадените двойки може да има и повтарящи се.

За всеки тестов пример програмата трябва да изведе на стандартния изход едно число – броя на гражданите в най-многобройната група от приятели.

Вход	**Изход**
1 10 12 1 2 3 1 3 4 5 4 3 5 4 6 5 2 2 1 7 10 1 2 9 10 8 9	6

Тук трябва да се намери свързаната компонента на графа, с най-много върхове. За целта просто намираме всички свързани компоненти в дадения граф и броим, кой от тях има най-много върхове.

Алгоритъма за намирането им е лека модификация на търсенето в дълбочина. Тъй като **dfs(u)** ще посети възлите, който са свързани с върха **u**, то просто трябва да пуснем **dfs** от всеки непосетен връх:

```
void dfs(int from)
{
	//маркираме текущия връх за посетен
	vis[from] = true;

	//за всеки негов непосетен съсед
	for(int i = 0;i < p[from].size();i++)
		if(!vis[p[from][i]])
			dfs(p[from][i]);//стартираме обхождане в дълбочина

	//увеличаваме броя на върховете, които са в текущия компонент
	numVertInComp++;
}

void solve()
{
	for(int i = 0;i < n;i++)
	{
		if(!vis[i])//за всеки непосетен връх
		{
			numVertInComp = 0; //зануляваме

			dfs(i);//стартираме обхождане в дълбочина

			//актуализираме, ако сме намерили по-добро решение
			if(numVertInComp > ans)
				ans = numVertInComp;
		}
	}
}
```

Оцветяване на свързани компоненти в граф

С лека модификация на горния алгоритъм можем да маркираме компонентите. Или казано с други думи да зададем различен „цвят“ на върховете, които се в различни компоненти. Следната модификация е позната като **Flood Fill**:

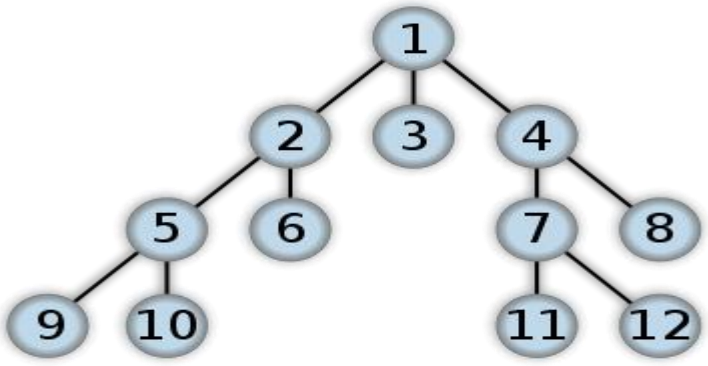
```
void floodFill(int from, int color)
{
	//маркираме текущия връх за посетен и му задаваме цвят
	dfsColor[from] = color;

	//за всеки негов непосетен съсед
	for(int i = 0;i < p[from].size();i++)
	   if(!dfsColor[p[from][i]])
```

```
            floodFill(p[from][i], color);//стартираме обхождане в дълбочина
}

void solve()
{
      numComponents = 0; //зануляваме
      for(int i = 0;i < n;i++)
        if(!dfsColor[i])//за всеки непосетен връх
             floodFill(i, ++numComponents);//стартираме обхождане в дълбочина

      for(int i = 0;i < n;i++)
            printf("Vertex %d has color %d\n", i, dfsColor[i]);
}
```

Вход	Изход
1 10 12 1 2 3 1 3 4 5 4 3 5 4 6 5 2 2 1 7 10 1 2 9 10 8 9	Vertex 0 has color 1 Vertex 1 has color 1 Vertex 2 has color 1 Vertex 3 has color 1 Vertex 4 has color 1 Vertex 5 has color 1 Vertex 6 has color 2 Vertex 7 has color 2 Vertex 8 has color 2 Vertex 9 has color 2

Обхождане в ширина (Breadth-first search)

Обхождането в ширина от даден връх i наричаме стратегията при, която започвайки от връх *i*, разглеждаме всички негови непосредствени *съседи*, и едва след това преминаваме към по-нататъшно обхождане в широчина от всеки един от съседите му. По този начин постигаме обхождане на графа по нива на всички достижими от стартови връх върхове. Ето как би изглеждало обхождане в широчина на граф, започвайки от най-горния връх:

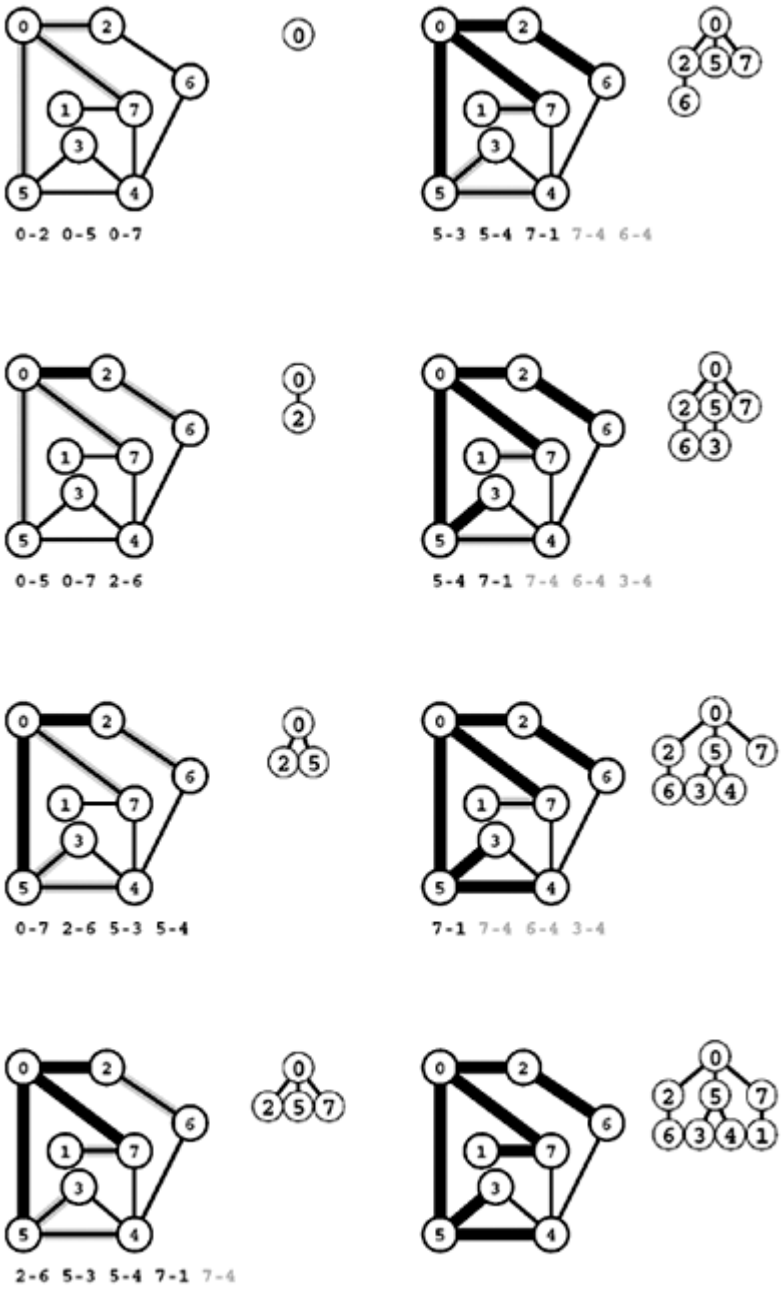


Върховете са номерирани, според реда в който ще бъдат посетени

Поддържаме опашка, в която първоначално се намира единствено стартовия връх. След това, докато в опашката има поне един връх, правим следното: вадим върха, който се намира в началото на опашката, разглеждаме го и добавяме в опашката всички негови непосетени до момента съседи. Върховете маркираме като посетени в момента, в който ги вадим от опашката.

```
void bfs(int startV)
{
	queue <int> bfs;

	viz[startV] = true; // маркираме стартовия връх като обходен
	bfs.push(startV); // и го добавяме в опашката

	// докато в опашката има елементи (върхове)
	while (!bfs.empty())
	{
		// взимаме върха, който е най-напред в опашката
		const int vertex = bfs.front();

		// махаме го от опашката
		bfs.pop();
		printf("%d ", vertex + 1);

		// за всеки негов съсед
		for (int i = 0; i < Graph[vertex].size(); i++)
		{
			const int toVisit = Graph[vertex][i];
			// ако е необходен
			if (!viz[toVisit])
			{
				viz[toVisit] = true; // маркираме го като обходен
				bfs.push(toVisit); // добавяме го в опашката
			}
		}
	}

	printf("\n");
}
```

Обхождането в дълбочина може да се използва за намиране на най-къс път в граф, ако теглата на ребрата са 1!



Примерна задача – Игра с плочки

Върху част от клетките на правоъгълна мрежа от квадрати с N реда и M стълба са поставени плочки. Напишете програма, която да провери дали е възможно плочка да бъде преместена от зададено място на дъската до друго зададено място на дъската. Плочката може да бъде местена от квадрата, в който се намира, в някой от съседните **празни** квадрати отляво, отдясно, отгоре или отдолу и не може да напуска мрежата.

На първия ред на стандартния вход е зададен броят на тестовете. Всеки тест започва с ред, на който са зададени целите числа N и M (3 < N, M ≤ 500). Всеки от следващите N реда съдържа по един низ с дължина M, описващ поредния ред от дъската – знакът ’*’, означава, че в съответния квадрат има плочка, а знакът ’.’ (точка) – че квадратът е празен. Последният ред съдържа четири цели положителни числа – координатите на квадрата с плочката, която искаме да преместим, и координатите на празен квадрат, в който искаме да я преместим.

За всеки тест, на отделен ред на стандартния изход програмата трябва да изведе YES, ако преместването е възможно или NO, ако не е възможно.

Вход	**Изход**
2 4 5 ***** *...* ***.* ...*. 3 1 4 3 4 5 ***** *...* ***.* ...*. 1 2 4 5	YES NO

Забележка. Редовете са номерирани от 1 до N в реда по който са зададени на входа, а стълбовете от 1 до M – отляво надясно.

Примерна задача – Дънна платка

Освен велик математик, Станчо е и млад техник. Едно от любимите му занимания за времето, в което се води, че трябва да прави нещо смислено е да модифицира домашния си компютър. При последното разглобяване Станчо реши да оптимизира тотално дънната си платка, тъй като не беше доволен от производителността ѝ, а не му се даваха пари за нова. За разликата от повечето компютри, при Станчовият дънната платка представлява матрица N на M от пинове, редовете и колоните на която са на разстояние 1. Всеки пин е или положителен или отрицателен. За да работи дънната платка, всеки положителен пин трябва да се свърже свързан към един отрицателен и всеки отрицателен към един положителен. Поради технически ограничения, връзките между пиновете са еднопосочни и трябва да са изградени от отсечки които са успоредни на редовете или колоните на матрицата. За сметка на това обаче, няма никакви проблеми тези връзки да се пресичат. Напишете програма, която по дадена дънна платка да определя най-малката възможна сумарна дължина на връзки които да се изградят, така че дънната платка да заработи.

На стандартния вход са дадени няколко описания на дънни платки. Всяко от тях започва с ред с двойка числа N и M (1 ≤ N, M ≤ 1000). Следват N реда с по M знака ‘+’ или ‘-’ описващи дали съответния пин е положителен или отрицателен.

За всяка дънна платка вашата програма трябва да изведе на стандартния изход ред с число — минималната обща дължина на връзки които да се изградят. В случай, че не е възможно платката да се свърже изведете 0.

Вход	Изход	Пояснение
2 2 +- -+ 2 3 +-- --- 1 3 +++ 3 6 +++-++ ++---+ +-----	4 10 0 28	При първия пример всеки пин може да се свърже към съседния При втория трябва да се изградят връзки с дължина 1, 1, 2, 1, 2 и 3 Третата платка не може да се свърже, тъй като няма отрицателни пинове

Решение:

При прочитане на задачата първото нещо, което трябва да бъде забелязано е, че се иска **най-кратък път от всеки връх до някой различен по вид връх**(пин) и се пита колко е сумата на всички тези най-кратки пътища. Този граф е с тегла между върховете 1, което ни подсказва, че задачата се решава с обхождане в ширина(**BFS**).

Интересното и полезното в тази задача е, че обхождането на графа не започва от един връх. Преди да започнем да обхождаме графа трябва да си направим таблица, в която ще пазим минималните пътища до различен по вид връх. Инициализирането на таблицата се извършва като попълваме с 1 всички клетки който имат различен по вид съседен пин. Примерно за графа:

+	-	-
-	-	-

Първоначалното състояние ва таблицата изглежда така:

1	1	0
1	0	0

Обхождаме в широчина с **начални върхове всички**, които са 1-ца в таблица. За всеки непосетен връх **u** проверяваме не обходените му съседи. Маркираме ги като посетени, слагаме ги в опашката и записваме теглото на пътя до тях по формулата A[v] = A[u] + 1. След като обхождането приключи таблицата има вида:

1	1	2
1	2	3

Отговора на задачата е сумата на всички клетки.