Тук са описани някои полезни похвати при имплементиране на алгоритми с графи. По време на практикума се забеляза, че има проблем с писането на алгоритмите и най-вече проблемно беше това, че се усложняват излишно реализациите. Тук ще се опитам накратко да опиша какво видях като реализации и как би било правилно да се направят.

DFS (Depth First Search)

Обикновно този алгоритъм се имплементира рекурсивно. Така се получава особено прост кода за написването му. Нещо от типа:

```
void DFS(int node)
{
	if(visited[node])
		return;

	visited[node] = true;

	for(int i=0;i<numChildren[node];i++)
		DFS(childred[node][i]);
}
```

Тук приемаме, че за представяне на графа се използва списък на наследниците - `childred[node][i]`. По първото измерение имаме номер на върха, по второто поредния наследник. Нужен ни е и масив, в който пазим броя наследници на даден връх – `numChildren[]`. Разбира се бихме могли да ползваме и масив от vector, за да пазим същата информация, без да има нужда от допълнителен масив с броя наследници. При рекурсивната имплементация трябва да внимаваме за дълбочината на рекурсията, която може да се дотигне. Ако например графа може да е с 100000 върха и може да бъде във формата на верижка би могло да се получи рекурсия с дълбочина 100000. Тогава стека, който се използва за рекурсията няма да е достатъчен. Въобще използваме ли рекурсия трябва да внимаваме колко дълбока може да е тя, защото стека, в който се записва всяко поредно извикване на функцията може да не достигне и да изгърми програмата ни. Ако много държим на DFS можем да го напишем итеративно със стек.

BFS (Breadth First Search)

Обикновно го реализираме итеративно с опашка. За разлика от DFS то ни дава най-кратък път по брой ребра от един връх (от който сме го пуснали) до всички останали. Виждаха се реализации с динамични опашки или пък рекурсивни реализации на алгоритъма. Но според мен няма нужда. Аз лично използвам масив за опашката и две променливи. Една показва къде може да се слага в опашката, а другата – от къде може да се взима елемент.

```
int tail[MAXN];
int tailPut;
int tailGet;
// празна опашка
tailPut = tailGet = 0;

// insert
tail[tailPut++] = newElement;
```

```
// pop
result = tail[tailGet++];

// проверка дали е пълна опашката
if(tailPut > tailGet){...}
```

Така съвсем просто можем да реализираме опашка за BFS-то, което пишем. Трябва да се внимава константата MAXN да е достатъчно голяма, за да стигне опашката. В повечето случаи няма нужда от динамични опашки с указатели и разни template-и. Така можеби сте писали тези структури по УП и ООП. В случая сами виждате, че това само усложнява нещата. Някои хора ползват структурите предлагани например от STL библиотеката. Това е много полезно в повечето случаи, защото със сигурност работи вярно и спестява писане на код. Трябва само да се внимава да не забавя решението ви.

Иначе като цяло представянията на графите обикновено се правят без динамични структури използващи указатели. Ако използвате списък на наследниците може просто да се използва масив от вектори. Всеки вектор се пълни с наследниците на някой връх. Ако искате можете да използвате матрица на съседство, тя се реализира с двумерен масив. Може да се наложи за списък на наследниците да се пази изформацията за всеки наследник в структура. Това ще стане в случая, когато за всеки наследник освен самия него трябва да се пази и теглото на реброто до него.

Dijkstra

Този алгоритъм достатъчно е разискван на лекции и по дискретни структури. Дадохме две задачи, които го използват в решенията си. Целта беше да се покаже, че той може да се използва не само за намиране на най-къс път в граф. Това разбиране е напълно грешно. Това може да звучи странно, но алгоритъма на Дейкстра се използва за намиране на **оптимални** пътища в граф. Това означава, че имаме дефиния за оптимален път и търсим такъв път. Оптимален може да означава най-къс, с най-голям капацитет (както е в задача PIPESYSTEM) или нещо друго. Някои студенти бяха учудени, че PIPESYSTEM може да се реши отново с алгоритъма на Дейкстра, но е така. Нужни са само малки модификации, за да проработи. По този начин с този алгоритъм биха могли да се решат и други задачи за оптимален път в граф.

Извода е че при решавани на подобни алгоритмични проблеми е хубаво преди да започнем да пишен код добре да се замислим как ще представим нещатата в програмата си. Важно е да измислим решение, но това не е последната стъпка преди писането на код. Обикновено използването на указатели и динамични структури може да се избегне. Това не винаги е така, но в повечето случаи е вярно. Добре обмислете коя е най-добрата реализация на решението ви преди да започнете писането на код.

Антон Димитров