**Оригиналният файл е на колегата ми Герасим, който води на 6та група, но аз го взех малко да го туйкна. С този шрифт са мои добавки.**

**В какви ситуации използваме основните алгоритми?**

**DFS (обхождане в дълбочина):**

**-** намиране на някакъв прост път между два върха;

- намиране на цикъл в граф

+ в ориентиран граф -> O(n+m)

+ в неориентиран -> O(n);

- намиране на свързаните компоненти на граф;

- топологично сортиране на граф (в обратен ред на излизане от връх ги колекционираме); (дотук прилагането му е почти непосредствено)

- използва се в алгоритъма за намиране на силно свързаните компоненти на ориентиран граф;

- използва се в алгоритъма за намиране на артикулационни върхове и мостове в граф;

- сложност по време в най-лошия случай **Theta(n+m)**..

**BFS (обхождане в широчина):**

**-** изграждане на слоеста мрежа в граф – на ниво k се намират върховете на разстояние k ребра от началния;

- намиране на най-къс прост път между два върха в безтегловен граф или цените на всички ребра/върхове са еднакви;

- може да се използва и за намиране на свързаните компоненти – в общи линии BFS може да се използва като заместител на DFS във всички алгоритми, в които използваме DFS за намиране на някакъв прост път или за установяване на достижимост между два върха (това, че намереният път е най-къс път, се явява просто бонус);

**-** дали граф е двуделен – лека модификация на BFS – всеки път поставяме маркировка L или R на върховете, като съседните върхове на текущия трябва да бъдат с обратната на неговата маркировка. Ако се окаже, че съседен на текущия връх е маркиран със същата маркировка като него, значи графът не е двуделен (Th на Кьонинг: Един граф е двуделен <-> не съдържа цикъл с нечетна дължина);

- сложност по време в най-лошия случай **Theta(n+m)**.

**Алгоритми върху тегловни графи:**

**Алгоритъм на Форд-Белман:**

- намира на най-къс път в тегловен граф с ребра произволни реални числа (положителни, 0 или отрицателни), **НО БЕЗ НАЛИЧИЕ НА ОТРИЦАТЕЛНИ ЦИКЛИ (при наличие на отрицателен цикъл задачата за намиране на най-къс път не е добре дефинирана, защото можем безкрайно да се въртим по него и да намаляваме пътя)** от **даден връх v до останалите върхове;  
-** можем да го използваме и за да установим наличие на отрицателен цикъл в графа;

- идеята му е |V|-1 пъти да „релаксираме“ всички ребра, като след стъпка номер i имаме дължините на най-късите пътища от връх v до всички останали с не повече от i ребра;

- сложност по време в най-лошия случай **O(n.m)**;

- малка оптимизация – ако след дадена стъпка не сме променили нищо, тогава можем спокойно да спрем процеса – със сигурност и на следващите стъпки няма да променим нищо. Оптимизацията не променя сложността в най-лошия случай.

**Алгоритъм на Флойд-Уаршал:**

– намиране на най-късите пътища в тегловен граф **с произволни ребра** без наличие на отрицателни цикли;

- работи на принципа на динамичното програмиране. Въпрос – нали казахме, че динамично програмиране можем да прилагаме само върху графи от тип DAG, защо тук го прилагаме върху произволен? Отговорът е, че прилагаме DP върху графи от тип DAG, но само когато се възползваме от топологична сортировка на този DAG и провеждаме алгоритъма върху нея. Тук динамичното не използва топологичната наредба, а “номерацията”, която сме въвели над върховете, и строим пътища с номера <= k;

- можем да приложим модификация на алгоритъма и за намиране на транзитивното затваряне на матрицата на съседство, т.е. за превръщане на матрицата на съседство в матрица на достижимост – операцията „събиране“ става операция „конюнкция“, а операцията „минимум“ става операция „дизюнкция“ (за транзитивно затваряне можем да ползваме и повдигане на матрица в степен, помислете как);

- сложност по време в най-лошия случай **O(n^3)**.

**Алгоритъм на Дийкстра:**

– намиране на най-късия път в тегловен граф с **ребра неотрицателни числа (безсмислено е да уточняваме без отрицателни цикли, тъй като такива не може да има);**

**-** намира от най-къс тегловен път от връх i до всички останали върхове и неявно изгражда дърво на най-късите пътища, сходен е с алгоритъма на Прим за намиране на минимално покриващо дърво;

- сложност по време – зависи от имплементацията – наивно - **O(n^2)**, с приоритетна опашка – **O((n+m)logn)**, с Фибоначиева пирамида - **O(n+mlogn)**.

**Задачата MST (minimum spanning tree, минимално покриващо дърво) разглеждаме само върху неориентирани графи.**

**Алгоритъм на Крускал за решаване на задачата за MST:**

**-** намира минимално покриващо дърво с всякакви ребра;

- също е и алгоритъм, който намира покриващо дърво, **минимизиращо на най-лекото ребро** в дървото;

- евристики **Union by rank** и **Path compression**;

- сложност по време – наивно - O(n^2), с използване на евристиките Union by rank + Path compression – O(m lgm) – сложността се определя от сложността на сортирането, сложността на union-find операциите е O(m\*alpha(n)), където alpha(n) е обратната функция на Акерман – изключително бавно растяща функция, за практически цели реално не надвишава 5;

- лека модификация на алгоритъма – сортиране на ребрата в началото в намаляващ ред - може да се използва и за намиране на максимално покриващо дърво, може и просто да умножим ребрата по -1 и да го подадем така на оригиналният Крускал, след това пак ги умножаваме по -1 и имаме максимално покриващо дърво.

**+Widest path problem**

**Алгоритъм на Прим-Ярник за решаване на задачата за MST:**

- изгражда дърво на най-късите пътища от даден връх v до всички останали;

- дървото на най-късите пътища от v се оказва и минимално покриващо дърво за графа;

**Практически постановки:**

- MST – задачата с велоалеите, която дадох за пример, има я и като задача 7 от основните задачи в дизайн 2;

- топологична сортировка – инсталиране на dependency-та (външните библиотеки, които ви се налага да използвате, когато разработвате дадено приложение – вие описвате в даден конфигурационен файл кои библиотеки ви трябват и като build-вате проекта, те почват да се свалят от някакво глобално хранилище, всяка библиотека обаче трябва да инсталира други библиотеки, които саматя тя ползва, те пък - свои и т.н., самата структура на зависимостите се моделира с граф, чиито върхове трябва да се сортират топологично и библиотеките следва да се инсталират в реда на топологичната им наредба. Ако някой от вас е ползвал Maven automation tool в Java-базирани проекти - като напишете в промпта **mvn clean install**, де факто отзад се случва това.); изчисляване на формули в Excel; хронологична подредба на изпълнението на задачи, зависещи една от друга по следния начин – задача Х трябва да се изпълни преди задача Y.

- доминиращо множество – да изберем в кои градове от дадена област да направи болници, така че във всеки град да има болница или ако няма, то да има такава в негов съседен.

**!NB! При моделиране на задача с граф трябва да изясните добре:**

**- кои са върховете и кои са ребрата - за оценка на сложността ще трябва да изясните и колко са те (по-често се изисква само O-асимптотиката);**

**- ориентиран или неориентиран е графът;**

**- имат ли тегла ребрата или нямат? Ако имат тегла – с какъв домейн са? Ако нищо не е казано по въпроса, приемаме ги за произволни реални числа (т.е. положителни, отрицателни и нула. Също това включва и целочислени, рационални и ирационални – в алгоритмите, които разглеждаме, няма разделение някои от тях да работят само за целочислени ребра, а други за всякакви. По-скоро въпросът, който стои, е дали са положителни или не.). Принципът е „всеки домейн, който не е изрично забранен, е разрешен“.**

**- какъв алгоритъм прилагаме и защо имаме право да го приложим? Каква е сложността му?**

Често се използват разсъждения от рода на:

- „разглеждаме подграф на дадения, в който сме изключили ребрата по-големи/по-малки от дадено тегло“ и правим разсъждения върху „орязания“ подграф – вижте BST (Bottleneck Spanning Tree) задачата по-долу, също тук вижте зад. 1 от изпит 2018 г. КН2;

- „допълнение“ на ребрата, т.е. на база изходния граф съставяме нов граф, чиито ребра са V2\E – вижте сводимостта на CLIQUE към INDEPENDET SET;

- “разглеждаме подграф на изходния без определени върхове или определени ребра” – например така може да се състави бавен алгоритъм за намиране на артикулационните върхове или мостове на граф (има го в статията, която съм реферирал за артикулационни върхове по-надолу), също задачата „MST в граф с две възможни стойности за теглата“.

**!NB! За решения в стил „Прилагаме еди-си-какъв алгоритъм“ без да е уточнено върху какъв граф, дори и да е верен алгоритъмът, се отнемат със сигурност (немалко) точки!!**

Отговор на въпроса „По задачите за графи, които ще имаме на изпита, длъжни ли сме да пишем псевдокод на алгоритмите и да го верифицираме?“ - и тук правилата са същите: **можете да ползвате** **директно без верификация само стандартни алгоритми, изучавани на лекции**. Ако използвате модификация на стандартен алгоритъм, тогава задължително трябва да кажете в коя част е модификацията, което ще повлече със себе си и нуждата от някакъв код и обяснения защо модификацията е коректна. Ако пък алгоритъмът е съвсем различен, ваш си - тогава е ясно, всичко трябва да е съвсем детайлно.

**Дизайн 2 – Моделиране на задачи с графи и прилагането на основни алгоритми върху графи**

**Книгата „Програмиране=++Алгоритми“:** [**http://www.programirane.org/download-now/**](http://www.programirane.org/download-now/) **– оттук може да се свали, в нея има доста разгледани задачи върху графи и върху динамично програмиране.**

**Файлът дизайн 2 – графи, във вашия курс е качен с име “Задачи от графи (2)”:** [**https://learn.fmi.uni-sofia.bg/pluginfile.php/137963/mod\_resource/content/2/Nina\_Design2\_Graphs.pdf**](https://learn.fmi.uni-sofia.bg/pluginfile.php/137963/mod_resource/content/2/Nina_Design2_Graphs.pdf)

**Основни задачи** – всеки от вас **трябва да може** да ги решава и да ги разпознава, т.е. да може да моделира практически постановки с тях. Считайте ги за абсолютен минимум на подготовката по темата.

**Други задачи** – препоръчително е да можете да ги решавате и разпознавате.

**Някои основни дефиниции и твърдения**

1. Да допуснем, че не е цикличен. Това означава, че от n-те ребра може да се състави непресичащ се път, т.е. прост път (в него не се съдържа цикъл). Тогава обаче този път трябва да има n+1 различни върха. Противоречие с условието, че върховете са n на брой.

2. Поне един **sink** и поне един **source** – да допуснем, че няма източник, т.е. няма връх, за който да е дадено, че в него не влизат ребра. Т.е. във всеки връх влиза поне едно ребро. Тогава, ако започнем от произволен връх, ще е ясно, че можем да направим „обратен път“, отивайки към някой негов предшественик, после към предшественика на последно добавения връх и т.н. И така докъде? Според това, което сме допуснали, всеки връх има поне един предшественик, т.е. процедурата може да продължава безкрайно. Да, обаче броят на върховете е краен, което означава, че в някакъв момент ще повторим връх. Тогава фрагментът от пътя, тръгвайки от повторения връх, и стигайки пак до него, ще образува цикъл. Противоречие с това, че графът е DAG. Следователно допускането ни е грешно. Т.е. има поне един източник. Аналогично и за доказателството за поне един sink.

**Задачи с директно прилагане на основните алгоритми**

1. **Броят на свързаните компоненти на неориентиран граф** – ясно е как. С колкото е нужно DFS-и – итерираме през върховете на графа, като при необходен пускаме DFS от него. Колкото пъти се наложи да пуснем DFS, толкова са компонентите на графа.

2. **Дали граф е цикличен –** отново DFS. Ако в даден момент можем да отидем във връх, който вече сме обходили, и той е предшественика на текущия в DFS-дървото, следва, че можем да затворим цикъл в този връх, тоест графът е цикличен.

3. Най-късият път по брой ребра от i до j – BFS.

4. A – Дийкстра – граф с **неотрицателни** тегла, B – Форд-Белман – граф с **произволни ребра** и **без отрицателни цикли**

5. Флойд-Уаршал – най-късите пътища от всеки връх до всеки един в граф с **произволни ребра** и **без отрицателни цикли**

6. Повдигане на матрицата на съседство в степен **k** ни дава броя на маршрутите с дължина **k –** по ДС мисля, че сте го споменавали.

**Задачи, решаващи се с търсене с връщане (backtracking)**

**Търсене с връщане** – алгоритъм от тип „с груба сила“ (brute-force). Търси някакво множество от „решения“ из множество от „потенциални решения“, подредени в дърво на решенията. Де факто обхожда цялото дърво на решенията, като, ако се намира на стъпка, на която е ясно, че няма да стигне до решение, не продължава обхождането, а се връща една стъпка нагоре. Поради тази причина сложността му е експоненциална. Като код е подобно на DFS, но разликата е, че търсенето с връщане след като обходи някой от наследниците си, демаркира всичко посетено от него, т.е. продължава обхождането от идентично състояние на това, в което се е намирало преди да влезе. Така сложността скача от линейна до експоненциална. Използва се за решаване на доказано NP-пълни задачи - за тях де факто друга алтернатива нямаме.

**Други задачи, зад. 1. Всички прости пътища между два върха в граф** – с търсене с връщане. В „Програмиране=++Алгоритми“ е показано с код.

**Друзи задачи, зад. 2.** **Проверка дали граф е Хамилтонов** – класическа NP-пълна задача. Може да се реши с търсене с връщане или с проверка на всички пермутации на върховете (със сложност O(n.n!)), което e по-голяма от полиномиална сложност.

**Хамилтонов цикъл с минимална дължина (Travelling Salesman Problem, TSP, задача за пътуващия пощальон)** – също NP-пълна задача, има интересен подход с динамично програмиране ( <https://en.wikipedia.org/wiki/Held>[–Karp\_algorithm](https://en.wikipedia.org/wiki/Held–Karp_algorithm) ) със сложност O(2n n2) - както виждате, отново сложността му е експоненциална. Ще го разгледаме на упражнения, защото е интересен е като идея. За алгоритми с груба сила тук могат да се ползват същите, каквито и за намиране на Хамилтонов цикъл в граф.

**Задачи върху DAG:**

**Топологично сортиране на DAG**

**Основни задачи, зад. 8.**

A - топологично сортиране. За А е важно, че ако графът е цикличен, няма как да направим топологично сортиране. Такова може да се направи само върху графи от тип DAG – **ориентиран ацикличен граф**.

В – постановка на задача за топологично сортиране – говорихме го на упражнения.

**Методи за топологично сортиране:**

- добавяме последователно на върхове към сортирания списък, започвайки от връх без предшественици (тоест източник - вече доказахме, че в DAG има поне един такъв), изтриваме го и продължаваме по същия начин за новополучения граф (който също е DAG, евентуално може вече да не е свързан, но това не е проблем), докато имаме все още върхове в графа;

- аналогичен алгоритъм, отстраняващ обаче последователно върховете без наследници (сифони) – очевидно тя ще ни води до анти-топологично сортиране, накрая трябва да го обърнем;

- топологична сортировка можем да получим и след сортиране на върховете в намаляващ ред на post-order числата им, получени при DFS обхождането.

**Проверка дали DAG има единствена топологична наредба** – DAG има единствена топологична наредба, тогава и само тогава когато тя е Хамилтонов път. Тоест правим една топологична наредба и проверяваме дали тя е Хамилтонов път. Ако е, значи няма как да получим друга топологична наредба, тъй като върховете са подредени във „верижка“. Следователно топологичната наредба е единствена. Можем да проверим по същия начин дали DAG съдържа Хамилтонов път – строим негова топологична наредба. Ако тя образува Хамилтонов път, то тогава графът има Хамилтонов път. В противен случай няма.

В една от задачите, давана на изпит при Скелета (2017 год. на поправителния изпит КН1), се иска по зададено n да се построи граф с n върха, който има: а) минимален брой топологични сортировки, б) максимален брой топологични сортировки. Помислете върху нея. А ако в а) въпросът е минимален строго положителен брой топологични сортировки?

**Динамично програмиране върху DAG**

**Други задачи, зад. 3.**

A – **брой пътища от даден връх до всички останали** - динамично програмиране върху DAG-а – правим топологична сортировка на подграфа, състоящ се от достижимите от i върхове (ако връх не е достижим от i, просто не участва в нея) и в реда на топологичната сортировка пресмятаме броя на пътищата от i до текущия, като:

**- в началото countPaths[i] = 1;**

**- countPaths[v] = sum{ countPaths[u] | има ребро (u, v) в графа } за v != i.**

Тази дефиниция е коректна, тъй като за предшествениците u на връх v функцията countPaths[] вече е изчислена, тъй като се движим по реда на топологичната сортировка, а в нея предшествениците на връх v сe намират преди v. За върховете v, недостижими от i – countPaths[v] = 0.

В – **най-дълъг път в DAG** – напомням, че задачата LONGEST PATH за произволен граф е NP-пълна, но за DAG има решение с динамично програмиране – вж. стр. 286 от „Програмиране = ++Алгоритми“ (правихме го на упражнения, браво на Елиан, че предложи и алгоритъм от източниците към сифоните).

**Мостове и артикулационни върхове (също наричани критични върхове, срязващи върхове) на неориентиран граф (разглеждаме задачата само за неориентиран граф, нали така?)**

**Други задачи, зад. 5.** А – артикулационни точки, B – мостове на граф – и двете се решават с изчисляване на lowest[] и preorder[] масивите в DFS обхождането с цел проверка за наличие на **обратни ребра.** Първо прафим DFS от произволен връх и разсъждаваме върху дървото, построено от него. Eдин връх **v** е артикулационен, ако е:

- или коренът на дървото и има поне 2 деца (това означава, че между тези две деца няма друг път, неминаващ през **v** – ако имаше, нямаше да са в различни поддървета спрямо **v**, ами едното щеше да е в поддървото на другото);

- или не е коренът на дървото и за него е в сила, че **lowest[u]>=preorder[v]** за някой негов пряк наследник **u**,

където за всеки връх **cur** изчисляваме

**lowest[cur] = min {**

**preorder[cur],**

**min { preorder[next] | next е наследник на cur в дървото },**

**min { lowest[next] | next не е наследник на cur в дървото, но има ребро от cur до next в графа }**

**}**

Ясно е, че масива lowest[] може да се изчислява в хода на изграждането на DFS дървото след връщането от DFS извикването за наследник на cur (ако въобще сме направили такова; ако по текущо разглежданото ребро не влизаме в DFS, това значи, че върхът в другия край вече е посетен, т.е. вече за него е изчислена lowest[] стойността). Предназначението на lowest[] масива е да ни каже следното: **lowest[u]** **е най-ранният preorder момент (момент на влизане във връх) за върховете, до които може да ни върне някой от върховете в T\_u (поддървото с корен u).** Тоест ако някой от върховете в поддървото **T\_u** може да ни върне във връх по-нагоре от u в дървото (т.е. неговият lowest e по-малък от preorder-a на u), то тогава имаме **наличие на „обратен път“**, т.е. можем да се върнем по-нагоре от **u** в дървото по път, който не минава през **u**. Следователно **u** не е артикулационен. В противен случай – **u** е артикулационен, защото не можем да се върнем някъде по-нагоре в дървото по път, неминаващ през **u**, т.е. всеки път в графа от връх в T\_u до връх по-нагоре от **u** в дървото задължително минава през него. Имайки алгоритъма за намиране на артикулационни върхове, мостове се намират с лека модификация, погледнете каква.

В „Програмиране=++Алгоритми“ същото е обяснено малко по-формално.

Статия по въпроса: <https://www.hackerearth.com/practice/algorithms/graphs/articulation-points-and-bridges/tutorial/>

**(**

**За любознателните:**

**Двусвързана компонента на граф** – подграф на даден граф, за който е изпълнено, че в него няма артикулационен връх, т.е. който и връх да премахнем, подграфът ще остане свързан. (Двусвързаната компонента е **обобщение на k-свързаната компонента** – **k-свързана компонента на граф** се нарича негов свързан подграф, за който е изпълнено, че **които и да е негови (k-1) върха да премахнем**, той ще продължи да бъде свързан. А **един граф се нарича k-свързан, ако при премахването на кои да е (k-1) негови върха, той остава да бъде свързан**. Тоест самият граф представлява k-свързана компонента на себе си. Не бъркайте k-свързаната компонента с компонентата на свързаност заради общата дума “компонента”, която използват! Факт е, че k-свързаната компонента е компонента на свързаност, но обратното не задължително е вярно - не задължително съществува някакво k, което да определя произволна компонента на свързаност като k-свързана.) Как намираме двусвързаните компоненти на граф? – Ще го допълня по-нататък.

)

Практическа постановка на задачата за намиране на артикулационни върхове в граф – да проверим например дали в мрежа от рутери има такъв, на който, ако му спре електрозахранването, мрежата ще се раздели на две отделни компоненти (т.е. ще престане да бъде свързана). Ако има такъв, трябва да гарантираме, че дори и то да спре, той ще продължи да работи на батерии. В общи линии артикулационните върхове представят **уязвимите точки** на една мрежа, на които трябва по някакъв начин да подсигурим допълнително.

**Силно-свързани компоненти на ориентиран граф**

**Други задачи, зад. 9. -** това са твърденията от лекции за силно-свързани компоненти, които имахте и на семестриалното контролно. В подточка В има грешка – във второто изречение G трябва да е GT, надявам се сте го забелязали.

**Покриващи дървета. Минимално покриващо дърво (MST – minimum spanning tree, покриващо дърво с минимално сумарно тегло)**

**Основни задачи, зад. 7.** - практическа постановка на задачата за MST в граф.

**Зад.** Даден е неориентиран свързан граф, чиито ребра са в множеството {a, b} (0<a<b). Нека **r** e теглото на минималното му покриващо дърво. Намерете стойността на r със сложност O(n+m) (а не самото MST!) – давана е преди години на контролни и на държавен изпит. Първо намираме свързаните компоненти на графа, образуван само от леките ребра. Ако тя е само една, т.е. той е свързан – супер, значи **(n-1)a** е теглото на MST-то. Ако не е, трябва за k-те образувани свързани компоненти, да добавим (k-1) ребра, с които да ги свържем – те ще са от тежките, защото очевидно как няма да са от леките. Т.е. MST-то е с тегло **(n-k)a + (k-1)b**. Има тук и обяснение от Андрей Дренски: [https://learn.fmi.uni-sofia.bg/pluginfile.php/140330/mod\_resource/content/3/Малко%20контролно%20№2%20-%20Условия%20и%20решения.pdf](https://learn.fmi.uni-sofia.bg/pluginfile.php/140330/mod_resource/content/3/Малко контролно №2 - Условия и решения.pdf)

**Други задачи, 11.**

А – С алгоритъма на Крускал, както говорихме **минималното покриващо дърво е и покриващо дърво, минимизиращо най-лекото ребро** (За който не разбира какво означава “покриващо дърво, минимизиращо най-лекото ребро” - това е покриващо дърво T, такова че **няма друго покриващо дърво T’, за което най-лекото ребро на T’ да има по-малка стойност** **от тази на T**.

В – **MBST** (**Minimum Bottleneck Spanning Tree** – покриващо дърво, минимизиращо най-тежкото ребро) – използва твърдението, **че всяко MST е и MBST**. Доказателство може да прочетете в задачата за хижите – зад. 1 от изпита 2017 година КН2 - т.е. спокойно можем да си построим MST отново с Крускал. Конкретно за построяване на MBST има известен линеен алгоритъм на Camerini, но той е само за любознателните.

С – **BST** (**Bottleneck Spanning Tree**) - проверка дали може да се изгради покриващо дърво с най-тежко ребро (наричано **bottleneck**), по-малко или равно на дадено **c**, където c e положителна константа – в зад. 1 от изпита 2017 КН2 е обяснено как става – игнорираме ребрата на графа, по-големи от с, и строим минимално покриващо дърво с алгоритъма на Крускал. Ако след края на тази процедура, графът е свързан, значи може, в противен случай - не може.

**Други задачи, зад. 12.** Второ минимално покриващо дърво – само за любознателните - в „Записки върху алгоритмите“ на Скелета.

**Други задачи, зад. 13.** Частично минимално покриващо дърво с **k** върха – в „Записки върху алгоритмите“ на Скелета:

за всеки връх **r** строим по алгоритъма на Прим минимално покриващо дърво в **r**, спирайки, когато в него са добавени (**k-1**) ребра, т.е. когато то съдържа **k** върха. Взимаме за частично минимално покриващо дърво с **k** върха това от получените **n** покриващи дървета с **k** върха, което има минимално тегло.

**Намиране на най-широк път в неориентиран граф (път, минимизиращ теглото на най-тежкото си ребро) – не е лошо да го погледнете**: <https://en.wikipedia.org/wiki/Widest_path_problem> . Може да се докаже, че най-широкият път в графа е път в **максималното** му покриващо дърво. Тоест построяваме го с модификация на алгоритъма на Крускал и търсим път в него.

**Ойлерови цикли и пътища – известна задача от класа P, за разлика от задачата за Хамилтонов цикъл, която е NP-пълна.** Учили сте я по ДС, дадена е тук, за да ви припомни необходимите и достатъчни условия за:

**–** наличие на Ойлеров цикъл в неориентиран свързан (мулти)граф - всички върхове да бъдат от четна степен;

- наличие на Ойлеров път в неориентиран свързан (мулти)граф - всички върхове без два да бъдат от четна степен, а тези два да бъдат от нечетна (A).

– наличие на Ойлеров цикъл в ориентиран (мулти)граф - всички върхове да имат равна входна и изходна степен и графът да е (слабо или силно - тук трябва да се помисли) свързан

-наличие на Ойлеров път в ориентиран (мулти)граф - всички върхове да имат равна входна и изходна степен с изключение на два - началният, който да има входна степен с 1 по-голяма от изходната, както и крайният, който да има изходна степен, която е с 1 по-голяма от входната и графът да е (слабо или силно - тук също трябва да се помисли) свързан (B).

На тях се базират и алгоритмите за намиране на Ойлеров цикъл и Ойлеров път в граф.

**За любознателните: Задачи върху дървета – диаметър, LCA:**

Първо малко дефиниции:

**Разстояние в дърво между u и v –** дължината на пътя (по брой ребра) между u и v – не казваме „най-късия път“, тъй като между всеки два върха в дърво съществува единствен път.

**Най-дълъг път в дърво, започващ във връх v –** пътят от v до върха, отдалечен на най-голямо разстояние от v.

**Ексцентрицитет на връх v –** дължината на пътя до най-отдалечения връх от **v** в дървото, тоест дължината на най-дългия път в дървото, започващ във връх **v** (неформално казано, колко стъпки „най-далече“ можем да отидем от върха **v**).

**Радиус на дърво –** минимумът от ексцентрицитетите на върховете в дървото. Как се намира радиус на дърво? - Ще го допълня по-късно.

**Диаметър на дърво –** максимумът от ексцентрицитетите на върховете в дървото или дължината на най-дългия път между два върха в дървото. В **зад. 4 от Други задачи** търсим диаметъра на дърво, това става по следния начин: за произволен връх намираме най-отдалечения от него – т.е. един от тези, отдалечени на разстояние, равно на ексцентрицитета му. Нека той е v. За връх v търсим пак най-отдалечения от него по същия начин. Нека той е u. Тогава пътят от v до u e диаметърът на дървото. Сложността по време е O(n). Ще допълня по-късно обяснението на коректността.

Основни задачи / 9. **LCA (Lowest Common Ancestor, най-близък общ предшественик на два върха, предшественик на два върха от най-голямо ниво)** – Нека върховете са u и v. За всеки връх в началото си изчисляваме дълбочината му – разстоянието до него от корена. След това намираме пътя от u до корена. След това се движим по пътя от v до корена, като намерим ли връх, който се среща в първоначално намерения път – супер, той е търсеният общ предшественик от най-голяма дълбочина. Решението е със сложност O(n). Самата задача за LCA става интересна, когато имаме множество заявки върху фиксирано дърво. Има цяла теория, посветена на това – тя разглежда връзката на LCA проблема с RMQ (Range Minimum Query, най-малък елемент в отрез на масив) проблема, с други структури от данни като Декартово дърво (Cartesian tree), Интервално дърво (Interval tree) – на когото му е интересно, в курса по „Бързи алгоритми върху структури от данни“ около една трета от курса е посветена на теорията около тази задача.

**Клика, антиклика, върхово покритие, доминиращо множество – дефиниции и основни техни свойства. Темата се отнася съществено и към „NP-пълни задачи“ - докато се подготвяте върху „Графи“, може да не четете тази част.**

**Клика на граф G(V, E)** – подмножество на върховете на G, които индуцират пълен подграф на G. Често пъти под „клика“ се разбира и самият индуциран пълен подграф.

**Максимална клика на графа G(V, E)** – клика на графа, такава че ако добавим връх на G, несъдържащ се в кликата, индуцираният подграф няма да бъде клика. Или с други думи това е клика на граф, която не е собствено подмножество на (тоест не може да бъде вложена в) друга клика на графа. Казано неформално, не може да се разширява повече, продължавайки да бъде клика.

**Максимална по мощност клика** – клика измежду кликите на графа, която има най-голяма мощност, т.е. всички клики в графа (в частност всички максимални клики) са с мощност, по-малка или равна на нейната. Ясно е, че всяка максимална по мощност клика е и максимална клика. Обратното не задължително е вярно.

**Независимо множество, антиклика** – подмножество на върховете на графа, такова, че индуцираният от него подграф е празен. Всяка антиклика е допълнение на клика на графа - очевидно.

**Максимална антиклика** – антиклика, която не може да бъде вложена в антиклика с по-голяма от нея мощност (т.е. такава, която не може да бъде собствено подмножество на друга антиклика). Неформално казано, не може повече да се разширява, запазвайки свойството си на антиклика.

**Максимална по мощност антиклика** – антиклика на граф с мощност, по-голяма или равна на мощността на всяка друга антиклика на графа.

**Свойство.** Всяка максимална по мощност клика/антиклика е и максимална клика/антиклика. Обратното обаче не задължително е вярно.

**Оптимизационните задачи за намиране на клика и антиклика (съответно CLIQUE и INDEPENDENT SET) са максимизационни.** Следователно**, кликово число на графа** ще наричаме максималната мощност на клика на графа, а **число на независимост на графа** ще наричаме максималната мощност на антиклика на графа.

**Намирането на кликово число и число на независимост на произволен граф са NP-пълни задачи,** т.е. не са известни полиномиални решения за тях до момента – можем да ги решим само с алгоритми с експоненциална сложност – например търсене с връщане. За дървета намирането на число на независимост има полиномиално решение с динамично програмиране, което ще разгледаме по-долу. За дървета кликовото число е най-много 2.

**Върхово покритие на графа G(V, E)** - подмножество на върховете на G, такова че всяко ребро в графа е инцидентно с поне един връх от него.

**Доминиращо множество на графа G(V, E)** - подмножество на V, такова че всеки връх или е в това множество, или някой от съседите му е в това подмножество. Някои ребра може да не се покриват от доминиращото множество – това не представлява проблем. Ясно е, че всеки изолиран връх на G задължително ще бъде в доминиращото множество, защото няма съседи.

**Оптимизационните задачи за върхово покритие и доминиращо множество на граф (съответно VERTEX COVER и DOMINATING SET) са минимизационни.** Следователно, **число на върхово покритие** ще наричаме минималната мощност на върхово покритие на графа, а **число на доминиране** ще наричаме минималната мощност на доминиращо множество на графа.

**Твърдение**. Всяко върхово покритие е и доминиращо множество в граф без изолирани върхове. Обратното не задължително е вярно.

**Следствие.** В граф без изолирани върхове числото на върхово покритие е по-голямо или равно на числото на доминиране.

**Намирането на числото на върхово покритие и числото на доминиране на произволен граф са NP-пълни задачи** – можем да ги решим само чрез експоненциален алгоритъм. За дървета, намирането на число на доминиране може да се реши полиномиално чрез алгоритъм с динамично програмиране, а намирането на число на върхово покритие може да се реши полиномиално чрез алчен алгоритъм.

**Твърдение.** Ако подмножество на върховете на даден граф образува антиклика, то комплементарното му (допълващото до V) подмножество образува върхово покритие. Обратното твърдение също е вярно.

**Следствие.** За всеки граф сборът от числото на независимост плюс числото на върхово покритие е n.

**Следствие.** За всеки граф минималното по мощност върхово покритие индуцира максимално по мощност независимо множество (което е допълнението му). Обратното също е вярно**.**

**Графът на Петерсен** – сам по себе си е интересен с това, че се явява контрапример в доста ситуации. Може да се упражните да намерите на ръка кликовото му число, числото на независимост, числото на върхово покритие и числото на доминиране - по ДС би трябвало да сте го правили.

**Други задачи, зад. 6.**

**А**

Кликово число на дърво – тривиално, то е най-много 2.

Число на независимост на дърво – чрез **динамично програмиране**, има я решена в сборника на Марков на страница 193.

**B**

Кликово число и число на независимост **на произволен граф** – NP-пълни задачи, решават се с търсене с връщане.

Всички максимални по включване антиклики **на произволен граф** – NP-пълна задача, отново с търсене с връщане.

**Други задачи, зад. 7.**

A. Число на доминиране **на граф** – NP-пълна задача, отново с търсене с връщане.

B. Число на доминиране **на дърво** – в сборника на Марков на стр. 194 я има решена с **DP.**

Число на върхово покритие на **дърво** - в сборника на Марков на стр. 192 долу е решена с алчен алгоритъм.

Число на върхово покритие на **граф** - отново NP-пълна задача.