Записки за алгоритмите

Г. Георгиев (Скелета)

6 януари 2013 г.

Съдържание

Глава 1

Въведение

Започнах тези записки при подготовката ми за изпита за гл. асистент към Факултета по Математика и Информатика (ФМИ) на Софийския университет през пролетта на 2012г.

Ще се опитам да ги разширя по време на преподаването ми в курса 'Създаване и изследване на алгоритми' (това е името, което ми харесва, официалното име е 'Дизайн и анализ на алгоритми', съкратено ДАА).

По-голямата част от материала е преразказ на части от книгите посочени в библиографията. Наблягам на теми, които не са изложени или са слабо застъпени в учебниците, публикувани на български език.

Надявам се записките да бъдат полезни на студентите, които изучават курса ДАА и на всички, които изпитват естетическа наслада от заниманията с алгоритми.

Скелета

1.1 Обозначения

Съветваме читателя първо да прочете учебника на К. Манев [1], за да е запознат с термините и обозначенията, обичайни при описване на дискретни структури и алгоритми.

Ето кратко изброяване на математически понятия и области, които ползваме:

Множества и операции върху множества, декартово произведение, релации, релации на еквивалентност и на частична наредба, функции над множества.

Комбинаторика. Изброителна комбинаторика. Основни комбинаторни конфигурации.

Графи, мултиграфи и хиперграфи. Пътища, прости пътища, контури и цикли в графи. Силна и слаба свързаност на ориентирани графи. Допълнения на графи. Оцветяване на графи. Дървета. Ойлерови цикли и Хамилтонови цикли.

1.2 Инструменти за анализ на алгоритмите

1.2.1 Асимптотични нотации

Функциите по-долу са асимптотично положителни, примерно $\exists n_0 \forall n,\ n > n_0 \to g(n) > 0$

$\Theta(g(n)) = \{f(n) : \exists n_0, \exists c_1 > 0, \exists c_2 > 0, \forall n,\ n > n_0 \to c_1 g(n) \leq f(n) \leq c_2 g(n)\}$

$O(g(n)) = \{f(n) : \exists n_0, \exists c > 0, \forall n,\ n > n_0 \to f(n) \leq cg(n)\}$

$\Omega(g(n)) = \{f(n) : \exists n_0, \exists c > 0, \forall n,\ n > n_0 \to cg(n) \leq f(n)\}$

$o(g(n)) = \{f(n) : \forall c > 0, \exists n_0, \forall n,\ n > n_0 \to f(n) \leq cg(n)\}$

$\omega(g(n)) = \{f(n) : \forall c > 0, \exists n_0, \forall n,\ n > n_0 \to cg(n) \leq f(n)\}$

Подробно и ясно изложение на свойствата на асимптотичните нотации е дадено в книгата с решени задачи на Минко Марков [2].

Асимптотичните нотации са метод за приближено боравене с функции. Идеята е да считаме две функции приблизително равни, ако се различават с множител някаква константа при нарастване на аргумента.

Това приближено представяне е удобно при оценка на сложността на алгоритмите.

Константата, която игнорираме има смисъла на разлика в хардуернати платформи на които изпълняваме един и същ алгоритъм.

Друга интерпретация е, че тази константа скрива разликата в ефективността на компилатора при превода на една програма от език на високо ниво към машинен код.

1.2.2 Мастър теорема

Нека е дадено рекурентно отношение $T(n) = aT(n/b) + f(n),\ a \geq 1, b > 1$. В следните 3 случая отношението има решение:

1) $f(n) = O(n^{\log_b a - \varepsilon})$ за някое $\varepsilon > 0$. Тогава $T(n) = \Theta(n^{\log_b a})$.

2) $f(n) = \Theta(n^{\log_b a})$. Тогава $T(n) = \Theta(n^{\log_b a} \lg n)$.

3) $f(n) = \Omega(n^{\log_b a + \varepsilon})$ за някое $\varepsilon > 0$. Нека $\exists c < 1, \exists n_0, n > n_0 \to af(n/b) < cf(n)$. Тогава $T(n) = \Theta(f(n))$.

Доказателството изследва нивата в рекурсивното дърво, породено от $T(n)$. Изброяват се възлите и допълнителната работа, зададена от $f(n)$.

1.2.3 Линейни рекурентни отношения с крайна история

Хомогенно рекурентно отношение: $a_n = c_1 a_{n-1} + c_2 a_{n-2} + \cdots + c_k a_{n-k}$.

Характеристично уравнение за отношението: $x^k - c_1 x^{k-1} - c_2 x^{k-2} - \cdots - c_k = 0$

Нека $\alpha_1, \alpha_2, \ldots, \alpha_s$ са различните корени на уравнението с кратности $k_i, \sum_i k_i = k$.

Тогава общото решение се задава от формулата:

$$a_n = \sum_{i=1}^{s} P_i(n) \alpha_i^n$$

където $P_i(n)$ са полиноми от степен $< k_i$. Общо те имат точно k коефицента, които могат да се определят ако знаем първите k члена на редицата $(a_0, a_1, a_2, \ldots)$.

Доказателството може да се извърши така:

1) Проверяваме, че ако α е корен на уравнението, $a_n = \alpha^n$ е решение на рекурентното отношение.

2) Ако α е кратен корен, тогава и $a_n = n\alpha^n$ е решение, защото α е корен на уравнението за производната на характеристичния полином. Правим тази сметка за многократен корен.

3) От 1) и 2) получаваме всички базови решения $n^j\alpha_i^n$, $j < k_i$. Тези решения са k на брой и са линейно независими, а общото решение е тяхна линейна комбинация. Коефицентите в линейната комбинация се определят еднозначно от първите k члена. Линейната независимост за различни решения следва от свойствата на матрицата на Вандермонд, (при кратни корени пак се получава линейна независимост).

1.3 Структури от данни

Стек, опашка, списък.

Структурите от данни са блокове за ефективна реализация на алгоритми. Обикновено нямат самостоятелно значение.

Когато ги ползваме, ги разглеждаме като черна кутия с предварително уговорени методи за достъп (интерфейси). Такава абстрактна структура може да има различни конкретни реализации.

При реализацията им пък определяме вътрешната структура и ефективността на отделните алгоритми за структурата.

1.3.1 Приоритетна опашка: пирамида

Приоритетната опашка е абстрактна структура, в която вкарваме обекти от типа $x =< key, value >$, а извличаме екстремален обект. Тук ще обсъждаме извличане на обект с максимален ключ, алгоритмите при работа с минимален са аналогични.

Методите за работа с приоритетна опашка са:
1. $Insert(x)$ вкарва нов обект в опашката
2. $Extract\_max$ изважда обекта с максимален ключ
3. Get_max връща обекта с максимален ключ (без да го вади от структурата)
4. $Modify(i, new\_key)$ променя ключа на обект, който се намира на позиция i.

Последният метод изисква по-сложни връзки между основният алгоритъм и структурата.

При наивните реализации едната от двете основни операции е със сложност $O(n)$, където n е броят на обработените обекти.

Най-често използваната ефективна реализация е (двоична) пирамида (binary heap).

Плътно двоично дърво ще наричаме такова кореново двоично дърво, при което всеки слой освен последният е наситен, а в последния са запълнени левите върхове. Такова дърво с n върха се разполага в обикновен масив $A[1..n]$ така: коренът е в $A[1]$, наследниците му са в $A[2], A[3]$, техните наследници в $A[4], \ldots, A[7]$ и т.н.

В плътно двоично дърво номерата на роднините на връх i се изчисляват така: родителят му е $P(i) = \lfloor \frac{i}{2} \rfloor$, левият му син е $L(i) = 2i$, а десният $R(i) = 2i + 1$.

Слой с номер k (корена е в слой 0, децата му в слой 1) се състои от всички върхове, чиито номера се записват с $k + 1$ цифри в двоичен запис: коренът има номер 1, децата му 10 и 11, техните деца $100, 101, 110, 111$ и т.н.

1.3.2 Непресичащи се множества

Нека $X = \{x_1, x_2, \ldots, x_n\}$ е множество от обекти, а $S = \{S_1, S_2, \ldots, S_k\}$ е динамично разбиване на X на непресичащи се подмножества. Множествата от разбиването ще отъждествяваме с някой техен елемент (представител) $y_i \in S_i$.

Разглеждаме следните операции над обекти от X:
$Make\_set(x)$ създава множество в разбиването с единствен елемент и представител x.
$Find\_set(x)$ връща представител на множество от разбиването, което съдържа x.
$Union(x, y)$ обединява множествата, съдържащи x и y.

Предполагаме, че се извършват m извиквания на някоя от 3-те операции, като първо се вика n пъти $Make\_set(x)$, за да се създаде начално разбиване, а след това в произволен ред се викат другите 2 операции. Очевидно $Union(x, y)$ може да се изпълни само $n - 1$ пъти, после в разбиването има само едно множество.

Реализации:

1) Списъци. За всяко S_i правим линеен списък от обектите му, от всеки обект има линк към началото, от началото има линк към края. При такава структура $Find\_set(x)$ и $Make\_set(x)$ ще се изпълняват за $\Theta(1)$. Ако реализираме $Union(x, y)$ така, че да залепва по-късия списък към по-дългия, всеки обект ще бъде преместван най-много $\lg(n)$ пъти (множеството му ще нараства поне двойно). Амортизираната сума на тези разместванията е $O(n \lg(n))$ за всичките операции $Union(x, y)$. Общата сложност при m извиквания на 3-те операции ще бъде $O(m + n \lg(n))$.

2) Коренови дървета. За всяко S_i правим кореново дърво от обектите му, коренът е представител на множеството и сочи себе си, поддържаме атрибут ранг $r(S_i)$, който има смисъл на дълбочина на кореновото дърво. При сливане на дървета корена на дървото с по-малък ранг насочваме към корена на другото. Така осигуряваме максимаен растеж на ранга до $\lg(n)$. Другата техника е компресия на пътищата - при изпълнение на $Find\_set(x)$ при обхождането на пътя от x до корена пренасочваме всички обекти по този път да сочат към корена:

$Find\_set(x)$
1) if $x \neq x.parrent$
2) $\quad x.parrent \leftarrow Find\_set(x.parrent)$
3) върни $x.parrent$

Ползването на рангове и компресиране на пътища заедно дават амортизирано време при m извиквания на 3-те операции $O(m\alpha(n))$, където $\alpha(n)$ расте много бавно, тя е сходна с обратна функция на Акерман и за всички обозрими случаи $\alpha(n) < 5$.

Глава 2

Алгоритми върху графи

2.1 Представяния на графи

5. Алгоритми върху графи. Представяния на графи – сравнителен анализ. Обхождане на графи в ширина и в дълбочина. Изчислителни проблеми, свеждащи се до обхождане в ширина или в дълбочина.

Нека $G(V, E)$ е граф и $|V| = n,\ |E| = m$.

G е плътен (dense), когато $m = \Omega(n^2)$. G е разреден (sparse), когато $m = O(n)$. Може да има и междинни състояния. Много алгоритми върху графи са най-ефективни в случая на плътен или разреден граф.

Представяне на ребрата E:
1) Матрица на съседства (добро при плътен граф).
2) Списъци на съседства $Adj(v),\ v \in V$ (добро при разреден граф).

Сравняване на двете представяния - заемана памет, скорост на операциите, вкарване на теглова функция и други атрибути в представянето и пр.

2.2 Обхождане на графи

Обхождане на граф е алгоритмична схема за систематично посещаване на всички върхове на графа и евентуално обхождане на ребрата му. Може да има самостоятелно значение, ако директно решава поставената задача, а може да е част от по-сложен алгоритъм.

2.2.1 Обхождане в ширина

Обхождане в ширина (Breadth-first search, BFS):

Тази схема разбива графа на нива (слоеве) на достижимост спрямо предварително избран връх s. Прилага се както за ориентиран, така и за неориентиран граф.

За всеки връх v поддържаме атрибутите цвят (бял, сив, черен) $v.c$, разстояние от източника $v.d$ и предшественик $\pi[v]$. Алгоритъмът използва обикновена опашка Q.

Съществени за работата на алгоритъма са цветовете (без черния) и опашката Q. Разстоянията $v.d$ и кореновото дърво π са полезни резултати.

Алгоритъм $BFS(G(V, E), s)$:
1) for $v \in V$ do $v.c \leftarrow white;\ v.d \leftarrow \infty;\ \pi[v] \leftarrow NIL$
2) $s.c \leftarrow gray; s.d \leftarrow 0$
3) $Q.Init; Q.EnQueue(s)$
4) while $Q \neq \emptyset$

5) $u \leftarrow Q.DeQueue$
6) for $v \in Adj(u)$
7) if $v.c = white$
8) $v.c \leftarrow gray;\ v.d \leftarrow u.d + 1;\ \pi[v] \leftarrow u$
9) $Q.EnQueue(v)$
A) $u.c \leftarrow black$

Инварианта: В цикъла на ред 4) всички сиви върхове са точно върховете от опашката Q.

Доказателство: Всеки връх преминава през състояния бял $\rightarrow$ сив $\rightarrow$ черен, като тези преходи са синхронизирани с вкарването и изкарването от опашката.

Скорост: Всеки връх се вкарва най-много веднъж в опашката и после се вади, а ребрата, излизащи от него се обработват веднъж на ред 6), следователно скоростта е $\Theta(n+m)$.

Памет: За работата си алгоритъмът използва $\Theta(n)$ клетки допълнителна памет за съхранение на 3-те атрибута на върховете (цвета е задължителен!).

Разстояния: Верни са следните твърдения:
1) Стойността на $v.d$ расте монотонно по реда на пъхане в опашката Q. (индукция по опашката).
2) Във всеки момент $v_r.d \leq v_l.d + 1$, където v_l е първият, а v_r е последният връх в опашката.
3) $v.d$ е най-краткият път от s до всеки връх v при единични дължини на ребрата, а π е кореново дърво, представящо най-кратките пътища (индукция по $v.d$).

Употреба:

Лека модификация на BFS разпознава дали входният граф G е двуделен (двуцветен) и разделя върховете на 2-та дяла.

Ползва се в бързите варианти на търсене на максимален поток за построяване на слоеста мрежа.

При единични дължини на ребрата BFS е най-бързият алгоритъм за намиране на най-кратки пътища от източник s до другите върхове.

2.2.2 Обхождане в дълбочина

Обхождане в дълбочина (Depth-first search, DFS):

Обикновено тази схема се ползва върху ориентиран граф с няколко силно свързани компоненти. Затова типичната реализация строи гора от дървета на достижимост. Присъщи атрибути на DFS са цветът, π (това май не се ползва ?) и 2 времена за всеки връх v - време на откриване $v.d$ и време на закриване $v.f$.

Алгоритъм $DFS(G(V,E))$:
1) for $v \in V$ do $v.c \leftarrow white;\ \pi[v] \leftarrow NIL$
2) $time \leftarrow 0$
3) for $v \in V$ if $v.c = white$
4) $DFS\_visit(G(V,E), v)$

$DFS\_visit(G(V,E), u)$
1) $time \leftarrow time + 1$
2) $u.c \leftarrow gray;\ u.d \leftarrow time$
3) for $v \in Adj(u)$ if $v.c = white$
4) $\pi[v] \leftarrow u$
5) $DFS\_visit(G(V,E), v)$
6) $time \leftarrow time + 1$
7) $u.c \leftarrow black;\ u.f \leftarrow time$

Тази реализация използва рекурсия, но би трябвало да може като BFS да се направи в цикъл, само че да използва стек вместо опашка.

Инварианта: Всички сиви върхове са точно върховете от стека на рекурсивните извиквания на DFS_visit.

Доказателство: Всеки връх преминава през състояния бял → сив → черен, като тези преходи са синхронизирани с влизането и излизането в рекурсивно извикване.

Скорост: За всеки връх се вика точно веднъж DFS_visit (когато е бял), а ребрата, излизащи от него се обработват веднъж в цикъла 3-5) на DFS_visit, следователно скоростта е $\Theta(n+m)$.

Памет: За работата си алгоритъмът използва $\Theta(n)$ клетки допълнителна памет за съхранение на атрибутите на върховете и стека на DFS_visit (цвета е задължителен!).

Класификация на ребрата:
A) **Дървено** е ребро от текущия към бял връх, по тези ребра върви рекурсията.
B) **Обратно** е ребро от бял към сив връх, тези ребра затварят цикли.
C) Ребро **напред** свързва връх с негов наследник в поддърво на алгоритъма.
D) Ребро **настрани** покрива останалите случаи - може да е връзка към някой друг връх в поддървото на алгоритъма, или връзка към вече изследвано поддърво.

Ребрата от вид C) и D) свързват текущият връх с черен връх.

Лема за скобите: За всеки два върха $u, v \in V$ следната верига неравенства е невъзможна: $u.d < v.d < u.f < v.f$, или казано неформално, времената на откриване и закриване на връх са като скоби в правилно написан аритметичен израз.

Доказателството следва от рекурсивната природа на алгоритъма, като допускаме $u.d < v.d$ и разглеждаме 2 подслучая - $v.d < u.f$, в който следва $v.f < u.f$, щото v е наследник на u и го изследваме изцяло преди да сме завършили изследването на u. Другият случай - $v.d > u.f$ директно нарушава въпросната верига неравенства.

Лема за белият път: От u до v има път в гората, която строи $DFS \iff$ в момента $u.d$ на достигане на u има път от бели върхове от u до v.

Доказателство: → следва от логиката на алгоритъма, ← доказваме от противното, вземаме v да е най-близкия до u, до който не достига път в гората и мислим какво става, когато DFS достигне върха предшестващ v по белия път.

Лема за неориентираният граф:
Ако графът е неориентиран, DFS открива само ребра от вида A) и B).

Доказателство: Нека $((u,v) \in E) \wedge (u.d < v.d)$. От лемата за скобите следва, че $v.f < u.f$. Ако алгоритъмът срещне най-напред реброто (u,v), трябва v да е бял (иначе ще се окаже, че по време на изследването на v е било обработено реброто (v,u)) - случай A). Ако пък най напред се обработи (v,u), u е сив - случай B).

Следствие: Малка модификация на DFS намира цикъл в неориентиран граф за време $O(n)$. Трикът е да усетим, че най-много n ребра ще изследваме до намирането на цикъла.

DAG наричаме ориентиран граф без цикли. Всеки DAG представя частична наредба, която може да се разшири до пълна наредба. Такова разширяване наричаме топологично сортиране.

Лема DAG:
Графът G е $DAG \iff DFS$ пуснат върху G не открива ребра от вида B).

Доказателство:
→ ако има обратно ребро, ще има и цикъл в G.
← ако има цикъл в G, проследяваме как DFS изследва върховете му и ще получим обратно ребро.

Алгоритъм TopologicalSort(G(V,E))
1) $l \leftarrow NIL$ (l е обикновен списък)
2) Извикваме DFS, при всяко изчисляване на $v.f$ пъхаме v в началото на l.
3) l съдържа топологична наредба на върховете на G.

Коректност: Нека $TopologicalSort$ изследва ребро (u,v). От лема DAG следва, че то не е

обратно, тоест v е или бял или черен връх. Ако v е бял, ще бъде рекурсивно изследван и от лемата за скобите ще имаме $u.d < v.d < v.f < u.f$. Ако пък е черен, той е вече приключен, а u е още сив, следователно $v.f < u.f$. И за двата случая получаваме $(u,v) \in E \to v.f < u.f$, следователно списъкът l задава пълната наредба (в него са всички върхове v в намалящ ред на $v.f$).

Означаваме $u \rightsquigarrow v$, когато има път от u до v.

Върховете u и v са силносвързани, когато $u \rightsquigarrow v$ и $v \rightsquigarrow u$. Силната свързаност е релация на еквивалентност, ще наричаме класовете ѝ на еквивалентност силно свързани компоненти (strongly connected component, SCC) на графа G.

Ако кондензираме силносвързаните компоненти на граф G, полученият граф е DAG (туй е очевидно).

Обърнат (транспониран) граф на $G(V,E)$ ще наричаме графът $G^T(V,E^T)$, за който ребрата са наобратно: $E^T = \{(u,v) : (v,u) \in E\}$.

Списъкът на съседства на $G^T(V,E^T)$ може да се построи за време $\Theta(n+m)$, а силносвързаните компоненти на G и G^T са едни и същи.

Алгоритъм $SCCsearch(G(V,E)$:
1) Пускаме DFS върху G и правим списък l за V в намалящ ред по $v.f$.
2) Построяваме $G^T(V,E^T)$.
3) Пускаме DFS върху $G^T(V,E^T)$, като в основният му цикъл вадим върховете от l.
4) Всяко поддърво в получената гора на достижимост е силно свързана компонента.

По долу предполагаме, че атрибутите $v.d$ и $v.f$ се отнасят за първото изпълнение на DFS в алгоритъма $SCCsearch(G(V,E)$.

Нека $d(C) = min\{v.d : v \in C\},\ f(C) = max\{v.f : v \in C\}$, където $C \subseteq V$.

Лема за компонентите:
Нека C и C' са различни силно свързани компоненти на G и реброто (u,v) излиза от C и влиза в C'. Тогава $f(C) > f(C')$.

Доказателство: Разглеждаме 2 случая.

Ако алгоритъмът първо достига връх $x \in C$, той ще обиколи и двете компоненти C и C' докато изследва x (туй следва от лемата за белия път и за скобите и от дефиницията на SCC), значи всички $v.f, v \in C'$ ще се окажат по-малки от $x.f$, следователно $f(C) > f(C')$.

Ако пък първият DFS достига най-напред $y \in C'$, той ще завърши изследването на y, а всички върхове от C ще са още бели, защото няма път от y до C (кондензираният граф е DAG). Чак след това ще почне изследване на C, значи пак $f(C) > f(C')$.

Следствие G^T: В обърнатият граф $G^T(V,E^T)$ е вярно обратното неравенство. В него SCC са същите като в G, но ребрата в кондензираният граф са наобратно.

Коректност на $SCCsearch$: Разглеждаме работата на ред 3), там вторият DFS работи върху $G^T(V,E^T)$, започвайки от върха с максимално $v.f$, тоест от компонента C_1, от която не излизат ребра към други компоненти в обърнатият граф (Следствие G^T). Така алгоритъмът ще създаде поддърво, съдържащо точно върховете от C_1. В оставащият граф са запазени същите условия, защото върховете от C_1 са черни и няма да се разглеждат повече, а в l оставащите върхове стоят наредени по намаляване на $v.f$. Следователно на следващата стъпка ще бъде отделена нова компонента C_2 и т.н. Строгото доказателство е с индукция по броят k на отделените силно свързани компоненти.

Сложност на $SCCsearch$: $\Theta(m+n)$

2.3 Минимално покриващо дърво

Нека $G(V, E)$ е неориентиран граф с теглова функция $w : E \to \mathbb{R}$. Нека $|V| = n$, $|E| = m$.

Искаме да намерим минималното (спрямо тегловата функция w) покриващо дърво $T(V, E_T)$ за G.

Ще казваме че $A \subseteq E$ е хубаво множество от ребра, ако A е подмножество на някое минимално покриващо дърво.

Ще казваме, че реброто (u, v) разширява хубавото множество A, ако $A \cup \{(u, v)\}$ е хубаво множество от ребра.

Лема *MST_cut* **2.1.** *Нека A е хубаво множество, $S \cup T = V$, $S \cap T = \emptyset$ е разрез на G, за който $u \in S$, $v \in T \to (u, v) \notin A$. Нека (u', v') е най-лекото ребро, което свързва S и T. Тогава (u', v') разширява множеството A.*

Доказателство: Допускаме противното, добавяме реброто (u', v') към MST-то, което покрива A, правим цикъл и смяна на ребро през разреза с (u', v') и получаваме по-леко дърво. □

Алгоритъм 2.1 Базов алгоритъм за намиране на MST

```
1: procedure MST(G(V,E), w)          ▷ G е неориентиран граф с тегла
2:     A ← ∅
3:     while |A| < |V| − 1 do
4:         търсим ребро (u, v) което разширява A
5:         A ← A ∪ {(u, v)}
6:     return A
```

Коректност: Инвариантa на цикъла: A е хубаво множество от ребра.

Искането новото ребро да е разширяващо гарантира липсата на цикъл. При всяко добавяне броят на несвързаните компоненти ще намалява с 1, т.е точно $n-1$ пъти ще има поне 2 компонента и ще може, съгласно механизма на лема 2.1 да се намери разширяващо ребро и добавянето му да запази верността на инвариантата.

Накрая A ще стане дърво и всичко е наред ! □

2.3.1 Алгоритъм на Крускал

Този алгоритъм избира най-лекото ребро, което разширява A. За да работи ефективно, алгоритъмът ползва структура 'непресичащи се множества', като обектите в нея са върховете на графа, а множествата от разбиването са дърветата в гората A:

Алгоритъм 2.2 Крускал: намира минимално покриващо дърво

```
1: procedure MST_Kruskal(G(V,E), w)          ▷ G е неориентиран граф с тегла
2:     A ← ∅
3:     for v ∈ V do Make_set(v)
4:     Sort(E) order by w(e)
5:     for (u,v) ∈ E do
6:         if Find_set(u) ≠ Find_set(v) then
7:             A ← A ∪ {(u,v)}
8:             Union(u,v)
9:     return A
```

Сложност: При ползване на ефективна структура, сложността на цикъла в редове 6-8 е $O(m\alpha(n))$, където $\alpha(n)$ расте много бавно (за практически възможните задачи $\alpha(n) < 5$). Сложността на сортировката в ред 4 е $O(m \lg(m)) \approx O(m \lg(n))$, което доминира алгоритъма. □

2.3.2 Алгоритъм на Прим-Ярник

Този алгоритъм е описан най-напред в чешко научно списание от Vojtěch Jarník през 1930. По-късно независимо е описан от Robert C. Prim през 1957 и отново от Edsger Dijkstra през 1959.

Този алгоритъм строи A свързано (растящо дърво) и избира най-лекото ребро, което е в разреза, образуван от върховете в A: V_A и останалите върхове $V_{\bar{A}} = V \setminus V_A$.

За всеки връх от $v \in V_{\bar{A}}$ има смисъл да следим само най-лекото ребро, което идва от дървото $T_A(V_A, A)$, защото останалите по-тежки ребра няма да бъдат избрани. Да означим с Q_A множеството от тези ребра, те са най-много n на брой.

Когато графът $G(V, E)$ е плътен, най-ефективната реализация е да представим Q_A чрез масив. Първият връх в дървото можем да изберем произволно, означаваме го с r. За всеки връх v ще поддържаме 3 атрибута: $v.key$ ще е тежестта на най-лекото ребро, свързващо v със строящото се дърво, $v.\pi$ ще е върхът от дървото, за който минимума се достига или NIL, а $v.marked$ ще е $true$ ако върхът е вече включен в дървото и $false$, ако още не е.

Така двойката $< v, v.\pi >$ представя ребро от Q_A.

В края на алгоритъма дървото ще се състои от ребрата $A = \{(v, v.\pi) | v \in V \setminus \{r\}\}$, а теглото на ребро $(v, v.\pi)$ ще бъде $v.key$.

Алгоритъм 2.3 Прим-Ярник: намира минимално покриващо дърво

```
1:  procedure MST_PrimA(G(V,E), w, r)          ▷ G е неориентиран граф с тегла
2:      for v ∈ V do                            ▷ Начални присвоявания
3:          v.key ← ∞
4:          v.π ← NIL
5:          v.marked ← false                    ▷ Всички върхове са извън дървото
6:      r.key ← 0                               ▷ Започваме да строим дървото от връх r
7:      cnt ← 0
8:      while cnt < n do                        ▷ cnt брои върховете в дървото
9:          u ← min_{v.key} {v : v.marked = false}   ▷ u е на най-лекото ребро в среза
10:         u.marked ← true                     ▷ Включваме u в дървото
11:         for v ∈ {x|((u,x) ∈ E) ∧ (x.marked = false) ∧ (w(u,x) < x.key)} do
12:             v.key ← w(u,v)
13:             v.π ← u
```

Сложност: Очевидно $O(n^2)$. □

Когато графът не е плътен, по-ефективно е да представим Q_A чрез приоритетна опашка (пирамида):

Алгоритъм 2.4 Прим-Ярник: намира минимално покриващо дърво

```
1:  procedure MST_PrimB(G(V,E), w, r)          ▷ G е неориентиран граф с тегла
2:      for v ∈ V do                            ▷ Начални присвоявания
3:          v.key ← ∞
4:          v.π ← NIL
5:      r.key ← 0                               ▷ Започваме да строим дървото от връх r
6:      for v ∈ V do
7:          Q.Insert(< v.key, v.π >)
8:      while Q ≠ ∅ do
9:          u ← Q.Extract_min
10:         for v ∈ {x|((u,x) ∈ E) ∧ (x ∈ Q) ∧ (w(u,x) < x.key)} do
11:             v.key ← w(u,v)
12:             v.π ← u
13:             Q.Move_up(< v.key, v.π >)
```

Сложност: Ако ползваме двоична пирамида при реализацията на Q, ред 7 ще се изпълни за $O(n \lg(n))$, цикълът започващ от ред 8 ще се изпълни n пъти, следователно ред 9 ще има сумарна сложност $O(n \lg(n))$, а редовете 11-13 ще се изпълнят по веднъж за всяко ребро на G. Еднократно изпълнение на ред 13 става за време $O(lg(n))$, а всички изпълнения на ред 13 ще дадат сложност $O(m \lg(n))$, което доминира цялата сложност и е същото като в описната по-горе реализация на алгоритъма на Крускал.

Ако ползваме пирамида на Фибоначи, всички изпълнения на ред 13 ще дадат сложност $\Theta(m)$ и тогава оценката за сложността на алгоритъма става $O(m + n \lg(n))$. Тази структура е сложна и май не се използва на практика. □

2.3.3 Второ най-леко дърво

Искаме всички тегла да са различни. Доказваме следната поредица твърдения:

1) Нека T е минималното покриващо дърво за графа G. T е единствено.

Доказателство: Допускаме че не е единствено, нека T е построено от алгоритъм (примерно на Крускал), а T' е друго минимално и $(u,v) \in T - T'$. Разглеждаме разреза, при който алгоритъма добавя реброто (u,v). В този разрез всички ребра са по-тежки от (u,v) и ще можем в $T' + \{(u,v)\}$ да махнем по-тежко ребро (процедурата от теорема MST_cut) и да получим по-леко дърво от T'. Второто дърво може да не е единствено - лесно се прави пример.

2) $\exists (u,v) \in T,\ \exists (x,y) \notin T,\ T' = T - \{(u,v)\} + \{(x,y)\}$ е второ дърво.

Доказателство: нека T' е второ дърво и $(u,v) \in T - T'$. В $T' + \{(u,v)\}$ има цикъл и ребро (x,y) в този цикъл, такова че $(x,y) \in T' - T$. $w(x,y) > w(u,v)$, щото T е минимално. Нека $T'' = T' - \{(x,y)\} + \{(u,v)\}$. Очевидно $w(T'') < w(T')$, демек T'' е минимално. Ама T е единствено, следователно $T = T''$ и се различава от T' само с едно ребро.

3) Алгоритъма за намиране на T' се основава на свойството 2).

Първо построяваме минималното дърво T, после за всяко ребро $(x,y) \notin T$ търсим най-тежкото ребро в маршрута, свързващ x и y в T, да го означим с $e_{(x,y)}$. Търсим (x_0, y_0), за което се достига $min_{(x,y)}[w(x,y) - w(e_{(x,y)})]$. Търсеното второ минимално дърво е $T' = T - \{e_{(x_0,y_0)}\} + \{(x_0,y_0)\}$. Всички сметки можем да направим за време $O(n^2)$.

2.4 Оптимални пътища в графи

2.4.1 Най-къс път, алгоритъм на Дейкстра

2.4.2 Алгоритъм на Флойд-Уоршел

2.4.3 Най-надеждни и най-широки пътища

Нека тегловата функция $w : E \to [0, 1]$ представя надеждността на ребрата в графа, тоест $w_{(u,v)}$ интерпретираме като вероятност за успешно преминаване на нещо (пътник, мрежови пакет или друг багаж) по реброто $(u, v) \in E$.

Когато разлеждаме път $P = (v_1, v_2, \ldots, v_k)$, вероятността за успешно преминаване по него w_P ще бъде произведение на надежностите на ребрата му $w_P = \prod_{i=1}^{k-1} (w_{(v_i,v_{i+1})})$.

Функцията $\ln(x)$ е монотонно растяща и отрицателна за $x \in [0, 1)$, следователно w_P ще достига максимум когато $-\ln(w_P)$ достига минимум.

$$-\ln(w_P) = -\ln(\prod_{i=1}^{k-1} (w_{(v_i,v_{i+1})})) = \sum_{i=1}^{k-1} -\ln(w_{(v_i,v_{i+1})})$$

Последното равенство свежда задачата за намиране на най-надежден път до задача за намиране на най-кратък път в същия граф, като единствено заменяме тегловата функция $w_{(u,v)}$ с $-\ln(w_{(u,v)})$ за всяко ребро на изходния граф.

Нека сега тегловата функция $w : E \to \mathbb{R}^+$ представя ширина на ребрата в графа (пропускна способност на мрежова връзка, ширина на пътна магистрала, канал и пр.).

За път $P = (v_1, v_2, \ldots, v_k)$ определяме ширината му $w_P = \min_{i=1}^{k-1} (w_{(v_i,v_{i+1})})$ като най-тясното ребро от пътя.

Задачата за намиране на най-широк път (пътища) се свежда до задачата за намиране на най-кратък път (пътища) с малка модификация на съответния алгоритъм.

Оказва се, че когато графът $G(V, E)$ е неориентиран, намирането на най-широк път в него е по-лесна задача от намирането на най-кратък път:

Лема за широкия път 2.1. *Ако в неориентиран теглови граф $G(V, E)$ построим максималното покриващо дърво T, всички пътища в него са най-широки.*

Доказателство: Нека P е най-широк път от s до t.

Нека (u, v) е най-лекото ребро в него, което не е от T. Нека C е цикълът, който се получава, ако добавим (u, v) към T. Понеже (u, v) не е от T, то е най-лекото ребро в C.

Образуваме сега графчето $P \cup C \setminus \{(u, v)\}$ (в него може да има преплитания). Това графче е свързано и в него има най-кратък път P_1 от s до t. P_1 е поне толкова широк, колкото е широк P и освен това сме намалили с едно ребрата в него, които не са от максималното дърво T. Това е индуктивната стъпка, която повтаряме докато намалим до 0 ребрата извън T. Ще получим път P_k, който свързва s и t и съдържа ребра само от T. □

От лема 2.1 следва, че можем да намерим бързо най-широките пътища в графа. Построяваме максимално покриващо дърво T за време $O(n^2)$ с алгоритъм на Прим за плътни графи 2.3 (с модификация за търсене на максимално дърво).

Нека T е представено със списъци на съседства. За всеки фиксиран връх v можем да пуснем някакъв алгоритъм за систематично бързо обхождане на T (примерно BFS), така ще намерим всички пътища от v до останалите върхове в дървото. Те са единствени в дървото T и са най-широки в G.

Можем да пуснем модификация на BFS, да я наречем BFS_w, която да изчислява ширините на намерените пътища. Понеже в дървото има $n-1$ ребра, за всеки връх ще намерим ширините на всички пътища към останалите върхове за време $O(n)$. Ако направим това за всички върхове, ще изчислим ширините на пътищата между всички двойки върхове за време $O(n^2)$.

Алгоритъм 2.5 Намира ширините на всички най-широки пътища

```
1: procedure WidestPaths(G(V,E), w)        ▷ G е неориентиран граф с тегла-ширини
2:     T ← MaxSpaningTree(G(V,E), w)       ▷ T - максимално покриващо дърво за G
3:     for v ∈ V do                        ▷ За всеки връх v намираме ширините на всички
4:         BFS_w(T, v)                     ▷ пътища от v до останалите върхове на T
```

Алгоритъмът 2.5 може да запише намерените ширини в матрица когато изпълнява ред 4. Тази матрица е подобна на генерираната от алгоритъма на Флойд-Уоршел, но се изчислява по-бързо.

Извеждането на самите пътища обаче не може да стане толкова бързо, защото отделният път в най-лошият случай ще има дължина $\Theta(n)$, така сумарната дължина на всички изведени пътища ще бъде $\Theta(n^3)$.

Когато търсим най-широк път между два конкретни върха s и t можем да го направим по-бързо от известните алгоритми за намиране на най-къс път. Има алгоритъм със сложност $O(m), m = |E|$, но поради сравнителната му сложност няма да го излагам тук, ето линк към него:

Volker Kaibel, Matthias A. F. Peinhardt: On the Bottleneck Shortest Path Problem

2.5 Потоци в графи

7. Потоци в графи. Метод на Ford-Fulkerson. Граф породен от поток – остатъчни капацитети. Увеличаващи пътища (augmenting paths). Срезове (cuts) в графи с потоци. Анализ на алгоритъма на Ford-Fulkerson. Алгоритъм на Edmonds-Karp.

Нека $G(V, E)$ е ориентиран граф, а $cap : E \to \mathbb{R}^+$ е теглова функция, задаваща максимална пропусквателна способност (капацитет) за всяко ребро.

Нека s и t са два специални върха - източник (извор) и цел (сифон).

Поток в $G(V, E)$ ще наричаме всяка функция $f : E \to \mathbb{R}^+$, за която:
1) $u \notin \{s, t\} \to \sum_u f(u, v) = \sum_u f(v, u)$ (запазване на потока - закон на Кирхов за ток в мрежа)
2) $f(u, v) \leq cap(u, v)$ (ограничение на капацитета)

Задача за максималния поток: при горните определения търсим поток който максимизира влизащият в t поток $Q = max_f(\sum_u f(u, t))$

Нека f е поток в $G(V, E)$. Дефинираме остатъчен граф $G_f(V, E_f)$, $cap_f : E_f \to \mathbb{R}^+$ така:
1) Ако $f(u, v) < cap(u, v)$ добавяме в E_f ребро (u, v), $cap_f(u, v) = cap(u, v) - f(u, v)$
2) Ако $f(u, v) > 0$ добавяме в E_f ребро (v, u), $cap_f(v, u) = f(u, v)$

$s - t$ разрез (cut) ще наричаме разбиване (S, T) на върховете на графа G, за което $s \in S$, $t \in T$, $S \cup T = V$, $S \cap T = \emptyset$.

Дефинираме обем на потока $f(S, T)$ и капацитет на разреза $cap(S, T)$ така:

$$f(S,T) = \sum_{u \in S} \sum_{v \in T} f(u,v) - \sum_{u \in S} \sum_{v \in T} f(v,u)$$
$$cap(S,T) = \sum_{u \in S} \sum_{v \in T} cap(u,v)$$

Лема 1: Обемът на потока $f(S, T)$ не зависи от S и T и не надминава $cap(S, T)$.

Доказателство: Нека прехвърлим връх u, различен от s, t между S и T (ще получим нов разрез (S', T')). От нулевият баланс на потока през u след малко смятане ще се получи еднаквост на потоците за двата разреза: $f(S, T) = f(S', T')$. Освен това:

$$\begin{aligned} f(S,T) &= \sum_{u \in S} \sum_{v \in T} f(u,v) - \sum_{u \in S} \sum_{v \in T} f(v,u) \\ &\leq \sum_{u \in S} \sum_{v \in T} f(u,v) \leq \sum_{u \in S} \sum_{v \in T} cap(u,v) = cap(S,T) \end{aligned}$$

Метод на Форд-Фалкерсон:
1) Създаваме нулев начален поток f.
2) Докато в графа $G_f(V, E_f)$ има път от s към t, увеличаваме потока с капацитета на пътя.
3) Връщаме f, той е максималният поток в графа.

Теорема (минималният разрез съвпада с максималният поток): При изпълнени условия на задачата за максималният поток следните 3 твърдения са еквивалентни:
1) f е максимален поток в графа G.
2) В $G_f(V, E_f)$ няма път от s към t.
3) Съществува разрез (S, T) и поток f, за които $cap(S, T) = f(S, T)$. ((S, T) е минимален разрез, а f е максимален поток)

Доказателство:

(1) $\Rightarrow$ (2) е очевидно.

(2) $\Rightarrow$ (3) Нека S съдържа всички върхове, достижими от s в $G_f(V, E_f)$. Няма обратни ребра, а всички прави са наситени, т.е. ще има равенство на потока и разреза (втората сметка от Лема 1).

(3) $\Rightarrow$ (1) Потокът е по-малък или равен от капацитета на всеки разрез (Лема 1). От равенството следва, че са оптимални и двете неща едновременно.

Анализ на метода:

При произволен избор на път в стъпка 2), потокът ще се увеличава с поне 1, т.е. скоростта ще зависи от големината на потока $F = |f|$ (предполагаме че са цели капацитетите). Лош пример - ромбоиден граф.

Избор на най-широк път (Edmonds-Karp #1). В този случай има път с ширина поне F/m (ако махнем всички по-тесни ребра и получим несвързан граф, ще получим разрез с капацитет $cap(S,T) < F$). Следва, че на всяка стъпка намаляме оставащият поток поне с коефицент $(1 - 1/m)$, което след малко сметки дава оценка $O(m^2 \lg n \lg F)$.

Избор на най-къс път (Edmonds-Karp #2). В този случай алгоритъмът прави най-много mn изпълнения на стъпка 2) или сложността му е $O(m^2 n)$

Доказателство: Разглеждаме слоестата мрежа, получена от алгоритъма BFS при търсене на път от s към t. Всяка итерация на Форд-Фалкерсон ще изтрива поне едно ребро и евентуално ще добавя ребра назад в G_f. Итерациите ще се изпълняват, докато се изчерпи потока в слоестата мрежа, т.е. най-много m пъти. После ще се премине към по-дълбока слоеста мрежа. Възможните мрежи са най-много n, следователно максималният брой итерации е mn, всяка итерация намира път за най-много m стъпки, откъдето получаваме оценката за сложност.

8. Слоести мрежи. Алгоритъм MPM (Malhotra, Pramodh-Kumar and Maheshwari). **Приложения на потоците в графи.**

Всички неща по-долу полват дефинициите и свойствата от предния въпрос.

Слоеста мрежа: Нека е даден граф $G(V,E)$ и поток f между специалните върхове s и t. Нека сме построили графът от остатъчни капацитети $G_f(V,E_f)$. Строим множество от нива $\{U_0,U_1,\ldots,U_q\}$ така:
1) $U_0=\{s\}$
2) $U_{i+1}=\{v\in V|\ \exists(u,v)\in E_f,(u\in U_i)\wedge(v\notin U_j,j\leq i)\}$
3) $U_q=\{t\}$

Можем да построим множествата U_i' с алгоритъма BFS, всяко от тях включва точно тези върхове, които са на разстояние i от източника s, ако разглеждаме ребрата в G_f като преходи с единични тегла. От U_i' получаваме U_i като премахнем върхове, от които няма пътища към t. Това може да се направи като пуснем BFS наобратно да търси пътища от t към s. После към получените върхове добавяме всички ребра от U_i към U_{i+1}.

Слоеста мрежа ще наричаме графът $G_l(V_l,E_l)$, където $V_l=\bigcup_i U_i,\ E_l=\{(u,v)|\ u\in U_i,v\in U_{i+1}\}$.

Свойства на слоестата мрежа:
1) Най-краткият път от s до t е с дължина q ребра.
2) Всички най-кратки пътища в G_f минават само по ребра от G_l.
3) В G_f няма ребра от U_i до U_{i+k} за $k>1$.

Блокиращ поток за слоестата мрежа ще наричаме поток, който минава само по ребрата й и я насища, т.е. след премахването на ребрата наситени от потока, $G_l(V_l,E_l)$ се разкъсва.

При строежа на такъв поток ще се премахват някои ребра напред в слоестата мрежа и ще се добавят (или увеличават капацитетите) на ребра назад (спрямо BFS) в G_f. Когато слоестата мрежа се насити, всички пътища в остатъчният граф ще са с дължина поне $q+1$. Това гарантира, че най-много n пъти ще циклим, ако търсим поток така:

Алгоритъм MPM
1) Създаваме нулев начален поток f.
2) Строим слоеста мрежа G_l спрямо G_f.
3) Използваме алгоритъм MPM_2, за да намерим блокиращ поток f' за G_l, увеличавме f с f' и отиваме на 2), докато в G_f има път от s до t.

Остава да направим бърз алгоритъм за намиране на блокиращ поток в G_l.

Определяме капацитет на връх $v\in V_l$ в G_l така:
$cap_l(v)=min(\sum_{u\in V_l}cap_f(u,v),\sum_{u\in V_l}cap_f(v,u))$

За s и t ще разглеждаме само изходящите/входящите ребра при дефинирането на $cap_l(v)$. Капацитетът на връх $cap_l(v)$ изразява максималното количество поток, което можем да прекараме през върха, без да се интересуваме от останалите върхове.

Алгоритъм MPM_2 Търсим блокиращ поток в G_l така:
1) Премахваме от G_l всички върхове v и свързаните с тях ребра ако $cap_l(v)=0$.
2) Избираме връх v_0, за който $cap_l(v_0)=f_0$ е минимално.
3) Прекарваме поток f_0 от v_0 нагоре по слоевете до t, като насищаме плътно ребрата.
4) Прекарваме поток f_0 от v_0 надолу по слоевете до s, като насищаме плътно ребрата.
5) Ако след модификация на потока в G_l все още има път от s до t, отиваме на 1).

Коректност и оценка на сложността на MPM_2

Тъй като на стъпка 2) избираме връх с най-малка пропускна способност, на стъпки 3) и 4) ще можем да удовлетворим тази пропускна способност и да увеличим текущия поток, като гарантирано избраният връх ще бъде премахнат от графа при следващото достигане на стъпка 1). Така MPM_2 ще изпълни основният си цикъл най-много n пъти.

В процеса на насищане раглеждаме два варианта - пълно и частично насищане на ребро. При пълното насищане реброто еднократно се обработва, т.е. най-много m такива обработки ще се извършат за цялата работа на MPM_2.

При частичното насищане пък за всеки връх най-много едно частично ребро ще има при обработка в стъпки 3) и 4), при едно минаване на основния цикъл това дава n обработки или за цялата работа на MPM_2 - n^2 обработки, а сумарно двата типа ребра ще добавят сложност $O(m+n^2) = O(n^2)$.

Капацитетите $cap_l(v)$ могат да се сметнат в началото и после само да се модифицират за време, пропорционално на времето за обработка на ребрата.

Така сложността на MPM_2 е O(n^2), а сложността на MPM е O(n^3), тъй като най-бавният му компонент е извикването на MPM_2, а броят извиквания не превишава n.

Специални случаи:

Когато всички ребра имат капацитет 1, сложността на MPM е $O(n^{\frac{2}{3}}m)$ защото:

1) MPM_2 прави само пълно насищане на ребра, т.е. неговата сложност е $O(m)$.

2) Броят цикли в MPM е $O(n^{\frac{2}{3}})$ (тънка сметка, следва от неравенство свързващо дълбочината на слоестата мрежа q и максимален поток в нея f_q, $q \leq 2n/\sqrt{|f_q|}$)

Проста мрежа наричаме мрежа с капацитети на ребрата 1 и максимален поток 1 през всеки връх, различен от s, t.

В проста мрежа сложността на MPM е $O(n^{\frac{1}{2}}m)$ Както и в предния случай, MPM_2 има сложност е $O(m)$.

Тъй като потокът f_q през всеки слой U_i е ограничен от броя на върховете в слоя $|U_i|$, имаме

$$n \geq \sum_{i=1}^{q} |U_i| \geq \sum_{i=1}^{q} |f_q| = q|f_q|$$

или $q \leq n/|f_q|$.

Ако $|f_q| \leq \sqrt{n}$, броят оставащи цикли в MPM е най-много $\sqrt{n}$ (на всеки цикъл поне с 1 нараства потока).

Нека в началото сме направили $\sqrt{n}$ цикъла в MPM. Тъй като на всеки цикъл получаваме слоеста мрежа с по-голяма дълбочина, след тези първи $\sqrt{n}$ цикъла имаме $q \geq \sqrt{n}$ или $|f_q| \leq n/q \leq n/\sqrt{n} = \sqrt{n}$. Този малък поток ще бъде наситен с най-много $\sqrt{n}$ оставащи цикъла в MPM. Така общият брой цикли става по-малък от $2\sqrt{n}$, откъдето получаваме сложността $O(n^{\frac{1}{2}}m)$.

2.6 Съчетания и реброви покрития

9. Съчетания и реброви покрития в графи. Увеличаващи пътища, съчетания в двуделен и произволен граф (алгоритъм на Edmonds). Свеждане на реброво покритие към съчетание.

Нека $G(V, E)$ е неориентиран граф. M е съчетание, когато $M \subseteq E, ((u, v_1) \in M) \wedge (u, v_2) \in M) \rightarrow v_1 = v_2$ (всеки връх е покрит от най-много едно ребро на M)

M е съвършено, когато $|M| = \lfloor |V/2| \rfloor$

Увеличващ път наричаме път $p = [u_1, u_2, \ldots, u_{2k}]$, за който първият и последният връх са свободни спрямо съчетанието M (не са покрити) и $(u_{2i}, u_{2i+1}) \in M, 0 < i < k$.

Лема 1 Ако P е множеството на ребрата в увеличаващ път $p = [u_1, u_2, \ldots, u_{2k}]$ за съчетанието M. Тогава $M' = M \oplus P$ е съчетание с мощност $|M| + 1$.

Теорема 1 Съчетанието M е максимално за графа $G \iff$ в G не съществува увеличаващ път за M.

Евристики за начално съчетание

Глава 3

Други бързи алгорими

3.1 Търсене в текст

10. Търсене в текст (string matching). Наивен алгоритъм и алгоритъм на Rabin-Karp. Търсене в текст чрез краен автомат. Алгоритъм на Knuth-Morris-Pratt.

Обозначения и дефиниции:

Ще търсим срещане на образец $P[1..m]$ в текст $T[1..n]$, $m < n$. И двата масива се състоят от символи в азбука Σ, $|\Sigma| = d$. Нека $P_k = P[1..k]$. $x \sqsupset y$ означава че низът x е суфикс на y. $x \sqsubset y$ означава че низът x е префикс на y.

Алгоритми:

Наивен Сложност $O(m(n-m+1))$

Рабин-Карп (1987)

Нека q е просто число, такова, че dq да е по-малко от размера на думата.

Нека $H_{RK}(S[1..m]) = (S[1]d^{m-1} + S[2]d^{m-2} + \cdots + S[m]) \ mod \ q$ е хеш функция върху низ S с дължина m. Да означим $t_i = H_{RK}(T[i+1..i+m]),\ p = H_{RK}(P[1..m]),\ h = d^{m-1}$

Можем бързо да изчисляваме t_i, когато пълзим по текста:

$t_{i+1} = (d(t_i - T[i+1]h) + T[i+m+1]) \ mod \ q$

$t_i \neq p \rightarrow T[i+1..i+m] \neq P[1..m]$, което дава основание за алгоритъма:
1) Изчисляваме $h, p, t_0, i = 0$.
2) Ако $t_i = p$, сравняваме $T[i+1..i+m]$ и $P[1..m]$
3) Ако $i < n - m$, пресмятаме t_{i+1} чрез t_i, увеличаваме i и отиваме на 2)

Сложността е най-лоша - $O(m(n-m+1))$, когато текстът и образецът съдържат еднаква повтаряща се буква.

В средният случай тя е $O(n+m\nu+cn/q)$, където ν е броят срещания на образеца в претърсвания текст, n/q е средният брой съвпадения на хеш-стойностите в т. 2), а c е средното време за което се установява, че въпреки съвпадението на хеш-стойностите, няма срещане на образеца.

В текст с малко срещания ν средната сложност ще бъде $O(n+m)$.

Краен автомат

Да си направим КДА $A_P =< Q, \Sigma, q_0, \delta, F >$ с $m+1$ състояния $q_0, q_1, \ldots, q_m$, които имат следния смисъл:

$\Delta(x) = q_i \iff i = max\{k | P_k \sqsupset x\}$ или i е най-дългото начало на P, което е суфикс на x, при работа на автомата върху стринга x.

Очевидно, ако имаме такава конструкция и пуснем автомата да работи върху текста T, всяко достигане до състояние q_m е еквивалентно на открито срещане на образеца. Последното ни дава

$F = \{q_m\}$

Определяме $\delta(q_i, a) = q_l$, където $l = max\{k | P_k \sqsupset P_i a\}$.

Непосредствено от дефиницията следва $\delta(q_i, a) = q_l \rightarrow l \leq i + 1$.

Теорема 1 Нека Δ е разширената функция на преходите на A_P.
Тогава: $\Delta(x) = q_i \iff i = max\{k | P_k \sqsupset x\}$

Доказателство: Провеждаме индукция по дължината на x и внимателно разглеждаме случаи.

Алгоритъм $FAM(T[1..n], A_P, m)$ (Finite Automation Matcher)
1) $q \leftarrow 0$
2) for $i \leftarrow 1$ to n
3) $\quad q \leftarrow \delta(q, T[i])$
4) $\quad$ if $q = m$
5) $\quad\quad$ Print 'Pattern found at position:' $i - m$

Кнут-Морис-Прат KMP

Много полезна е следната префиксна функция за P (едномерен масив):

$$\pi[q] = max\{k | (k < q) \wedge (P_k \sqsupset P_q)\}$$

Алгоритъм $KMP(T[1..n], P[1..m])$
1) $\pi[1..m] \leftarrow EvalPrefixFun(P)$
2) $q \leftarrow 0$
3) for $i \leftarrow 1$ to n
4) $\quad$ while $(q > 0) \wedge (P[q+1] \neq T[i])$
5) $\quad\quad q \leftarrow \pi[q]$
6) $\quad$ if $P[q+1] = T[i]$
7) $\quad\quad q \leftarrow q + 1$
8) $\quad$ if $q = m$
9) $\quad\quad$ Print 'Pattern found at position:' $i - m$
A) $\quad\quad q \leftarrow \pi[q]$

Изчисляването на префиксната функция π е подобно на алгоритъма KMP, с тази разлика, че образецът P се сравнява със себе си, като се почва от втора позиция:

Алгоритъм $EvalPrefixFun(P[1..m])$
1) $\pi[1] \leftarrow 0$
2) $q \leftarrow 0$
3) for $i \leftarrow 2$ to m
4) $\quad$ while $(q > 0) \wedge (P[q+1] \neq P[i])$
5) $\quad\quad q \leftarrow \pi[q]$
6) $\quad$ if $P[q+1] = P[i]$
7) $\quad\quad q \leftarrow q + 1$
8) $\quad \pi[i] \leftarrow q$

Сложност на KMP:

Освен ред 5), другите редове в цикъла 3-A се изпълняват за време $\Theta(n)$. Нека t_5 е броят на изпълненията на ред 5) по време на работата на алгоритъма.

Доказваме по индукция неравенството $t_5 \leq i - q$. В началото $t_5 = 0$, $i - q = 1$. Тъй като $q < \pi(q)$, при всяко изпълнение на ред 5) t_5 ще нараства с 1, а $i - q$ с едно или повече, тоест неравенството се запазва. При преминаване на ред 7) (ако е изпълнено условието от ред 6) $i - q$ ще намалее с едно, но при достигане края на цикъла i ще се увеличи с 1, така че в редове 6-A $i - q$ не намалява. Това завършва доказателството на неравенството $t_5 \leq i - q$, накрая i достига n, оттам сложността на цикъла while (редове 5-6) е $O(n)$, а сложността на алгоритъма без ред 1) е $\Theta(n)$.

За $EvalPrefixFun$ сметките са същите и неговата сложност е $\Theta(m)$. Следователно общата сложност на KMP е $\Theta(n)$.

Коректност на KMP:

Теорема 2 За всяка стойност на i алгоритмите FAM и KMP достигат съответни редове 4) и 8) при еднаква стойност на q.

Доказваме с индукция по i. Досадно, дълго и скучно доказателство.

От теорема 2 следва, че FAM и KMP са еднакво коректни, а коректността на FAM вече е доказана в теорема 1.

Бойер-Муур-Галил

3.2 Хеш таблици

11. Хеш таблици. Универсални хеш функции. Перфектно хеширане.

Дефиниции:

Хеширане: Алгоритмична техника за търсене, вмъкване, замяна в таблица (речник) T от обекти от вида $< key, value >$, $key \in U$ за средно време $\Theta(1)$.

Хеш функция: Функция $h : U \to \{0, \ldots, m-1\}$ от множеството на всички ключове към редовете в $T[0..m-1]$.

Колизия: Обикновено $|U| \gg m$. Колизия наричаме случай на ключове x, y на обекти в речника, за които $h(x) = h(y)$. Реализацията на хеширане трябва да разреши колизиите и да използва хеш функция пораждаща минимум колизии.

Универсално множество хеш-функции: Крайното множество $\mathcal{H}$ от хеш функции $h : U \to \{0, \ldots, m-1\}$ е универсално, ако за всеки 2 ключа $k, l \in U, k \neq l$ броят на функциите даващи колизия $h(k) = h(l)$ е най-много $|\mathcal{H}|/m$.

Съществуване на универсални хеш-функции (1979 г.):

Нека U е множество от числа и p е просто число, $p > |U|$. Така можем да разглеждаме всеки ключ k като число в интервала $0, 1, \ldots, p-1$. Нека m е размера на таблицата.

Нека $a \in \mathbb{Z}_p^* = \{1, 2, \ldots, p-1\}$, и $b \in \mathbb{Z}_p = \{0, 1, \ldots, p-1\}$.

Дефинираме:
$h_{ab}(k) = ((k.a + b) \ mod \ p) \ mod \ m$
$\mathcal{H}_{pm} = \{h_{ab} \mid a \in \mathbb{Z}_p^*, \ b \in \mathbb{Z}_p\}$

$\mathbf{T_1}$ $\mathcal{H}_{pm}$ е универсално множество хеш-функции.

Доказателство: Първо разглеждаме свойствата на функциите $h'_{ab}(k) = ((k.a + b) \ mod \ p)$

Те са пермутации на $\mathbb{Z}_p$, освен това ако $k \neq l$, по образите им $r = h_{ab}(k), \ s = h_{ab}(l)$ можем еднозначно да определим $a, \ b$:

$$a = (r - s).((k - l)^{-1} \ mod \ p) \ mod \ p$$
$$b = (r - a.k) \ mod \ p$$

Броят на възможните двойки $< a, b >$ и $< r, s >$, $r \neq s$ е точно $p.(p-1)$, демек съвсем равновероятно са разпределени попаденията на $r, \ s$. Като вземем по модул m да смятаме, се вижда че вероятността за съвпадение $Pr(r = s \ mod \ m) \leq 1/m$, което завършва доказателството.

Разрешаване на колизиите:

Единият метод е чрез списъци. В таблицата за всяка хеш-стойност k се съпоставя списък от обекти, за които $h(x_i) = k$. Ако входовете в таблицата са m, а сме вкарали n обекта, запълнеността означаваме с $\alpha = n/m$.

Другият метод наричаме **отворено адресиране**. При него всички обекти са в таблицата, като при колизия имаме някакъв метод за разполагане на ново място, а празните места са заети от празен обект NIL. При отвореното адресиране задължително $\alpha < 1$.

Неописано: техника за по-добро запълване на отворена таблица !!!

Перфектно хеширане (Renzo Sprugnoli 1977, тук - 1984 г.):

Когато множеството от ключове е статично, можем да направим двустепенна схема за хеширане без колизии, т.е. хеширане с най-лоша оценка $\Theta(1)$. Първата степен е хеширане към таблица със списъци, а вместо списъци правим вторични таблици с размер n_i^2, където n_i е броят на колизиите за хеш-стойност i.

Нека имаме таблица с размер n и $k+1$ запълнени клетки. Вероятността в тази таблица да няма колизии да означим с $NC_{n,k}$. Очевидно $NC_{n,k} = 1.(1-\frac{1}{n}).(1-\frac{2}{n})\dots(1-\frac{k}{n})$.

$\mathbf{T_2}$ $NC_{n^2,n-1} > \frac{1}{2}$

Доказателство: Ще направим по-силна оценка.

От неравенството $(1-\frac{i}{n}).(1-\frac{k-i}{n}) \geq (1-\frac{k}{n})$ следва:

$$NC_{n,k} = \prod_{i=0}^{k}(1-\frac{i}{n}) \geq \prod_{i=0}^{\frac{k}{2}}(1-\frac{k}{n}) = (1-\frac{k}{n})^{\frac{k}{2}} = ((1-\frac{k}{n})^{\frac{n-k}{k}})^{\frac{k}{n-k}\frac{k}{2}} > \Big(\frac{1}{e}\Big)^{\frac{k^2}{2(n-k)}}$$

защото $((1-\frac{k}{n})^{\frac{n-k}{k}}) = ((1-\frac{k}{n})^{\frac{n}{k}-1} > 1/e)$.

За $NC_{n^2,n-1}$ получаваме оценката:

$$NC_{n^2,n-1} > \Big(\frac{1}{e}\Big)^{\frac{(n-1)^2}{2(n^2-n+1)}} > \Big(\frac{1}{e}\Big)^{\frac{1}{2}} = \frac{1}{\sqrt{e}} > \frac{3}{5}$$

От тази оценка следва, че при случаен избор на хешираща функция от универсално множество при хеширане на вторичните таблици с размер n_i^2 и n_i запълвания, вероятността да няма колизии е над $\frac{3}{5}$.

$\mathbf{T_3}$ $\mathrm{E}\big[\sum n_i^2\big] < 2n$

Нека основната таблица при перфектното хеширане има n слота и n запълвания.

$$\sum_{i=1}^{n} n_i^2 = \sum_{i=1}^{n} n_i + 2\sum_{i=1}^{n}\binom{n_i}{2} = n + 2\sum_{i=1}^{n}\binom{n_i}{2}$$

Последната сума $\sum\binom{n_i}{2}$ е точно броят на всички колизии. Понеже използваме универсални хеш-функции очакването за този брой е:

$$\mathrm{E}\big[\binom{n}{2}/n\big] \leq \frac{n-1}{2}$$

или:

$$E\big[\sum_{i=1}^{n} n_i^2\big] \leq n + 2\frac{n-1}{2} < 2n$$

Сега прилагаме неравенство на Марков:

$$Pr\big\{\sum_{i=1}^{n} n_i^2 > 4n\big\} \leq \frac{E\big[\sum_{i=1}^{n} n_i^2\big]}{4n} < \frac{2n}{4n} = \frac{1}{2}$$

Последното неравенство гарантира, че след няколко опита ще изберем успешно хеш функция, която ползва $\Theta(n)$ памет за двете нива на перфектната хеш таблица.

12. Дървета за двоично търсене и операции върху тях. Червено-черни дървета и B-дървета.

Дърво за двоично търсене е кореново дърво T с указатели в двете посоки. За всеки връх v ще означаваме указателите съответно към родителя $v.p$ и към двата наследника с $v.l$ и $v.r$. За липсващи обекти слагаме указател NIL, а с n ще означаваме броят на върховете в T. Всеки връх съдържа структурната информация $< key, value >$. Основното свойство на дървото е:

Нека v е произволен връх. За всеки връх u от лявото поддърво на v е изпълнено $v.key \geq u.key$. За всеки връх u от дясното поддърво на v е изпълнено $v.key \leq u.key$.

Ще наричаме атрибута key ключ, той определя наредбата на върховете в дървото T.

Такава структура позволява обхождане на върховете на дървото в сортиран ред за време $O(n)$:

Алгоритъм $TreeWalk(v)$
1) if $v \neq NIL$
2) $\quad TreeWalk(v.l)$
3) $\quad$ print $< v.key, v.value >$
4) $\quad TreeWalk(v.r)$

Ако височината на дървото е h, можем да извършваме следните бързи (за време $O(h)$) запитвания:
$TreeSearch(x, k)$ - намира връх с ключ k в поддървото с корен x.
$TreeMin(x)$ - намира връх с минимален ключ.
$TreeMax(x)$ - намира връх с максимален ключ.
$TreeSucc(x)$ - намира връх следващ по големина на ключа върха x.
$TreePred(x)$ - намира връх предшестващ по големина на ключа върха x.

Две операции променят структурата на дървото - вмъкване и триене на връх.

$TreeSearch(x, z)$ вмъква нов връх z (с начални указатели NIL) в поддърво с корен x. Процедурата търси в дървото указател NIL, който да разрязва дървото по ключа $z.key$ и го замества със z.

$TreeDelete(x, z)$ премахва връх z от поддърво с корен x. Тя разглежда 3 случая:
1) Ако z няма деца, заменяме го с NIL.
2) Ако z има само едно дете y, заменяме го с y.
3) Ако z има 2 деца, завъртаме примерно надясно дясното му дете y така, че $v = TreeSucc(z)$ да се окаже корен на детето и тогава заменяме z с v. Друга възможна техника е да намерим $v = TreeMin(y)$, да заменим v с дясното му дете, а z да заменим с v.

Завъртането е хитра процедура, която не нарушава основното свойство на дърво за двоично търсене. Псевдокода е дългичък, затова словесно само описвам завъртане надясно: Нека връх v има ляво дете u, дясно дете γ, а u има деца α и β. Правим следните промени:
1) u става родител на v и корен на поддървото
2) Новите деца на u са α и v
3) Новите деца на v са β и γ

Завъртане наляво се дефинира аналогично.

Глава 4

Сложност на изчислителни проблеми

15. Машина на Тюринг. Универсална машина на Тюринг. Недетерминирана машина на Тюринг. Тезис на Тюринг—Чърч. Алгоритмично нерешими изчислителни проблеми.

17. Долни граници върху сложността на проблеми. Основни техники за доказване на долни граници: дървета за вземане на решения (decision trees), игри срещу противник (adversary argument), редукции.

4.1 Сложност по време

18. Изчислителни проблеми с отговор ДА-НЕ. Класове на сложност на изчислителните проблеми. Класове на сложност P, NP и co-NP . Класове на сложност NP-complete и NP-hard.

19. Съществуване на проблеми в класа NP-complete. Теорема на Кук (Cook). NP пълнота на 3-Sat.

20. Апроксимиращи алгоритми за оптимизационни проблеми от NP-complete. Примери: алгоритми за Върхово Покритие (Vertex Cover) и Задачата за Търговския Пътник в Метричен Вариант (Metric Travelling Salesman Problem). Разлика между евристика и апроксимиращ алгоритъм. Условна невъзможност за апроксимиране.

4.2 Сложност по памет

21. Класове на сложност L, NL, PSpace. Теорема на Савич (Savitch). Теорема на Immerman–Szelepcsényi, следствия. Взаимоотношения между класовете на сложност по време и по памет.

Функцията $S : \mathbb{N} \to \mathbb{N}$ е констуируема по време/памет (time/space constructible), когато съществува TM_S, която получава на входа си 1^n и изчислява $1^{S(n)}$ за време/памет $O(S(n))$. Обичайните функции $n^k, \lg(n), 2^n$ са конструируеми и по време и по памет.

Сложност по памет: Нека $S(n)$ е конструируема по памет и $L \subseteq \{0,1\}^*$. Ще казваме, че $L \in \mathbf{SPACE}(S(n))$ (съответно $L \in \mathbf{NSPACE}(S(n))$), когато съществува TM_L (съответно NTM_L), която разпознава езика L, като за всяка входна дума x използва памет $O(S(|x|))$.

Изискването S да е конструируема по памет позволява на машините от класа да 'знаят' колко памет използват по време на изчислението.

При сложността по време е смислено да се разглеждат само случаите $S(n) \geq n$, поради факта, че машината ще прочете входните данни за време поне n.

При сложност по памет за прочитането на входа е достатъчно да имаме указател (брояч) към него, той заема памет $\lg(n)$, затова ще разглеждаме класове по памет, за които $S(n) \geq \lg(n)$.

При малките класове, за които $S(n) < n$ входните данни са повече от паметта, в която е разрешено да правим изчисления. Затова разделяме паметта на 2 части - памет само за четене (в която са записани входните данни) и памет за четене/запис (в която се извършва изчислението). Във формалният модел който ползваме, това ще е машина на Тюринг с две ленти - едната read_only за входа x, другата read_write с размер $O(S(|x|))$.

Нека M е (определена или размножаваща се) машина на Тюринг. Конфигурация на M ще наричаме информацията, точно определяща състоянието на машината в някакъв момент от изчислението - съдържанието на клетките от паметта (за лентите read_write), мястото на главата (главите), състоянието на машината.

Граф на конфигурациите на M при входна дума x наричаме графът $G_{M,x}$ с върхове възможните конфигурации и ребра преходите на M между тези конфигурации при изпълнение на един преход (стъпка) от изчислението. За определена (обикновена) машина от всеки връх излиза едно ребро, при недетерминирана (размножаваща се) машина ще има върхове с по-висока степен.

Началната конфигурация бележим с C^x_{start}, а при достигане на крайно състояние, ако трием лентата ще достигаме единствена крайна конфигурация C_{accept}.

Лема 1 Нека $G_{M,x}$ е конфигурационният граф на машина M, ограничена по памет от $S(n)$ и $|x| = n$. Тогава:

1) Всяка конфигурация на M (връх на $G_{M,x}$) може да се представи в памет $cS(n)$ за някоя константа c, оттам броят върхове на $G_{M,x}$ е по-малък от $2^{cS(n)}$.

2) Ако C, C' са 2 конфигурации, може в памет $O(S(n))$ да се изчисли дали $(C, C') \in E_{G_{M,x}}$ (дали от C машината може да премине към C').

Доказателството е пипкаво, но интуитивно ясно.

Теорема 1 $\mathbf{DTIME}(S(n)) \subseteq \mathbf{SPACE}(S(n)) \subseteq \mathbf{NSPACE}(S(n)) \subseteq \mathbf{DTIME}(2^{O(S(n))})$

Доказателство: Първите 2 включвания са ясни, последното следва от Лема 1.

Дефинираме естествените класове по памет така:
$\mathbf{PSPACE} = \bigcup_{c>0} \mathbf{SPACE}(n^c)$
$\mathbf{NPSPACE} = \bigcup_{c>0} \mathbf{NSPACE}(n^c)$
$\mathbf{L} = \mathbf{SPACE}(\lg(n))$
$\mathbf{NL} = \mathbf{NSPACE}(\lg(n))$

Масова задача $stCON$:

Даден е ориентиран граф $G(V,E)$, $s,t \in V$. Има ли път от s към t ?

Задачата *stCON* е централна за класовете по памет заради свойствата на графа на конфигурациите $G_{M,x}$ и Лема 1 - изчисляването в ограничена памет е сводимо до търсенето на път от C^x_{start} до C_{accept} в графа $G_{M,x}$.

Лема 2 $stCON \in \mathbf{NL}$ (пускаме недетерминистка разходка от s до t).

Лема 3 $\mathbf{NL} \subseteq \mathbf{P}$ (пускаме алгоритъмът BFS да търси път в $G_{M_L,x}$ за $L \in \mathbf{NL}$).

Алгоритъм на Савич:
$stCON(G,s,t) \leftarrow Reach(G,s,t,n)$, където $n = |V|$

$Reach(G,u,v,k)$
1) if $k < 2$ return $(u = v) \vee ((u,v) \in E)$
2) for $w \in V$
3) if $Reach(G,u,w,\lfloor k/2 \rfloor) \wedge Reach(G,w,v,\lceil k/2 \rceil)$
4) return $TRUE$
5) return $FALSE$

Сложност по памет: Рекурсивната функция $Reach(G,u,v,k)$ ползва стек с дълбочина $\lg(n)$ и пъха в него 3-4 променливи - u,v,k,w, всяка с размер $\lg(n)$. Следователно използваната памет е $\Theta(\lg^2(n))$.

Коректност: С индукция по k доказваме, че $Reach(G,u,v,k)$ връща $TRUE$, ако има път от u до v с дължина не повече от k.

Теорема на Савич:
Нека $S(n)$ е конструируема по памет и $S(n) \geq \lg(n)$. Тогава $\mathbf{NSPACE}(S(n)) \subseteq \mathbf{SPACE}(S^2(n))$.

Доказателство: Пускаме алгоритъма на Савич да търси път в графа от конфигурации, като му даваме да ползва памет $\Theta(S^2(n))$.

Следствие: $\mathbf{PSPACE} = \mathbf{NPSPACE}$ защото полином на квадрат пак е полином.

Сводимостта в класа **NL** трябва да е в по-слаб клас и понеже няма избор, трябва да се реализира с функции от **L**. Тия функции нямат място за писане, та се дефинират така, че да изчисляват динамично - или искаме да изчисляват i-тия бит на резултата, или пък да пишат на виртуална лента на която главата се движи само в едната посока (write_once). Виртуалната лента може да се реализира като двойка $< i,b >$, като i е позицията на главата върху лентата за запис, а b е битът, който е записан последно. Означаваме тази сводимост в логаритмична памет с $\propto_L$.

Лема 4 Очаквани свойства на сводимостта $\propto_L$:
1) $(A \propto_L B) \wedge (B \propto_L C) \rightarrow A \propto_L C$
2) $(A \propto_L B) \wedge (B \in \mathbf{L}) \rightarrow A \in \mathbf{L}$

И двете условия на лемата се свеждат до пресмятане на композиция на две функции, ограничени в логаритмична памет. Нека примерно f свежда A до B, а g свежда B до C. Реализираме композицията $g(f(x))$ така:

Започваме да изчисляваме g и когато се наложи да се чете бит i от входната лента на машината свеждаща B към C, пускаме f да изчисли този бит, като ползва реалната входна лента x, която съдържа индивидуален вход на задачата A.

Лема 5 *stCON* е **NL**-пълна задача.

Доказателство: От лема 2 знаем, че $stCON \in \mathbf{NL}$. Всяка задача $A \in \mathbf{NL}$ свеждаме към *stCON* като строим матрица на съседствата на нейния граф от конфигурации $G_{M_A,x}$. От Лема 1 следва, че всеки бит от тази матрица може да се изчисли в памет $O(\lg(n))$.

Сертификат в **NL**: Поради липса на памет, сертификатът в **NL** няма къде да бъде записан. Както и при логаритмичната сводимост $\propto_L$ ще ползваме допълнителна лента, но този път за четене само в едната посока (read_once). За интерпретацията на такава лента е нужен един бит r. Когато искаме да четем от сертификата, пускаме размножаване на недетерминираната машина на Тюринг, като двете копия изписват съответно 0 и 1 в бита r, а следващата инструкция прочита

r.

Масова задача not_stCON:
Даден е ориентиран граф $G(V,E)$, $s,t \in V$. Отговорът на задачата е ДА $\iff$ няма път от s към t.

Теорема на Immerman-Szelepcseny: $not\_stCON \in \mathbf{NL}$

Доказателство: Нека върху read_once лентата да е изписан път $(v_0, v_1, \ldots, v_k)$. Такъв път може да бъде потвърден в памет $\Theta(\lg(n))$, достатъчно е при последователното изчитане на върховете да проверим за всяко $i = 0, 1, \ldots, k-1$ условието $(v_i, v_{i+1}) \in E$.

Нека с C_k означим множеството от върхове, достижими от s чрез пътища, не по-дълги от k, а броят им с $r_k = |C_k|$.

Нека разгледаме сертификат S_k, който се състои от поредица пътища с дължина не-повече от k, започващи от s и достигащи различни върхове на графа, зададени в нарастващ ред по номера на крайния връх. Можем да проверим в памет $\Theta(\lg(n))$ коректността на тези пътища, а също и да ги преброим. Нарастващият номер на крайният връх гарантира, че всички пътища в S_k достигат различни върхове. Следователно $|S_k| \leq r_k$. Ако знаем предварително r_k и се окаже, че в S_k има точно r_k пътища, то S_k ще се окаже валиден сертификат за множеството C_k.

Алгоритъмът на Immerman-Szelepcseny използва идеята за индуктивно преброяване на пътищата, ако знаем r_{k-1}, можем да изчислим r_k така:

$r_0 = 1$, защото само s е достижим чрез път с дължина 0.

Нека сме изчислили r_{k-1}. Нека $Q_k = nS_{k-1}$ е поредица от n валидни сертификата за множеството C_{k-1}. Тъй като знаем r_{k-1}, тези сертификати могат да бъдат проверени при прочитането.

Преди прочитането на Q_r присвояваме начална стойност на $r_k \leftarrow 0$. При прочитането на i-тия сертификат S_{r-1} освен проверката му за валидност ще правим следното:

1) Следим дали върхът с номер i е достижим за $k-1$ или k стъпки от s. Ако този връх е край на някой от пътищата в S_{k-1}, той ще е достижим за $k-1$ стъпки, ако пък има ребро от края на някой път в S_{k-1} до върха i, той ще достижим за k стъпки. Така в края на изчитането на Q_k ще имаме тази информация и можем да увеличим r_k с 1, ако i е достижим за не повече от k стъпки от s.

2) Следим дали върхът t е достижим. Ако това се случи, убиваме копието на програмата. Така ще осигурим завършване на алгоритъма само в случай, че няма път от s към t.

След изчитането и анализа на Q_k, r_k ще съдържа точния брой върхове в C_k.

Целият сертификат на алгоритъма е редицата $(Q_1, Q_2, \ldots, Q_{n-1})$. Приемането му означава, че алгоритъмът е изчислил коректно всички r_i, проверил е достижимостта на всички върхове и не е намерил път до t. Лесно се вижда, че през цялото време се използват само краен брой указатели и броячи, тоест изчислението се извършва в памет $\Theta(\lg(n))$. Това завършва доказателството на теоремата.

Следствие 1:
Нека $S(n)$ е конструируема по памет и $S(n) \geq \lg(n)$.
Тогава $\mathbf{NSPACE}(S(n)) = \mathbf{coNSPACE}(S(n))$.

Доказателство: Смятаме за графа на конфигурациите, като провеждаме същите разсъждения, както в теоремата на Immerman-Szelepcseny.

Следствие 2 Някои връзки между класовете на сложност:
1) $\mathbf{L} \subseteq \mathbf{NL} = \mathbf{coNL} \subseteq \mathbf{P} \subseteq \mathbf{NP}$
2) $\mathbf{DTIME}(S(n)) \subseteq \mathbf{NTIME}(S(n)) \subseteq \mathbf{SPACE}(S(n)) \subseteq$
$\subseteq \mathbf{NSPACE}(S(n)) = \mathbf{coNSPACE}(S(n)) \subseteq \mathbf{DTIME}(2^{O(S(n))})$

Следствие 3: $2\text{-}SAT \in \mathbf{NL}$.

4.3 Алгоритми, ползващи случайна поредица

22. Класове на сложност BPP, RP, coRP, ZPP. Тест на Милер-Рабин за прости числа. Намиране на минимален срез в неориентиран граф (Karger).

Има два алтернативни начина да дефинираме алгоритми, ползващи случайност - като машини на Тюринг с вероятностна функция на преходите (примерно с две различни функции на преходите, като при изчислението всеки път се избира случайно коя да се ползва), или като машина на Тюринг, която ползва допълнителна *read_once* лента, на която има записана редица случйни битове, които могат да се ползват при изчислението. Лесно се свеждат един към друг тези модели.

Ще означаваме с $M(x)$ случайно изчисление на машина на Тюринг M върху вход x, а с $M(x,r)$ случайно изчисление на M върху вход x при използване на лента от случайни битове r.

Ще казваме, че M е PPT, когато M е машина на Тюринг, която ползва случайност и работи полиномиално време.

Класът **RP** се състои от езиците L, за които има PPT M, такава че:

$$x \in L \to Pr[M(x) = 1] > \frac{1}{2}$$
$$x \notin L \to Pr[M(x) = 0] = 1$$

Тълкуваме горната дефиниция така: ако случайното изчисление върху x даде отговор 'да', тогава е вярно че $x \in L$, ако пък отговорът е 'не', не е ясно какво става.

$\mathbf{RP} \subseteq \mathbf{NP}$, защото има много изчисления $M(x, r_i)$, които приемат x и всяка от лентите r_i е сертификат, че $x \in L$.

Класът **coRP** се състои от езиците L, за които има PPT M, такава че:

$$x \in L \to Pr[M(x) = 1] = 1$$
$$x \notin L \to Pr[M(x) = 0] > \frac{1}{2}$$

Да означим с $L(x)$ характеристичната функция на езика L: $L(x) = 1$ когато $x \in L$ и $L(x) = 0$, когато $x \notin L$.

Класът **BPP** се състои от езиците L, за които има PPT M, такава че:

$$Pr[M(x) = L(x)] > \frac{2}{3}$$

Очевидно класът **BPP** включва **RP** и **coRP**. Не са ясни отношенията му с **NP**. Съвременното очакване е $\mathbf{BPP} = \mathbf{P}$, но теориите по тази тема са сложни.

По-малък клас в тази йерархия е **ZPP**, той е от езици L, за които има PPT M, която дава 3 възможни отговора $0, 1$ и ? (последното интерпретираме като 'не знам') и за всеки вход x:

$$Pr[M(x) = '?'] < \frac{1}{2}$$
$$Pr[M(x) = L(x) \vee M(x) = '?'] = 1$$

Следствие 1: $\mathbf{ZPP} = \mathbf{RP} \cap \mathbf{coRP}$

Доказателство: Нека $L \in \mathbf{RP} \cap \mathbf{coRP}$ и машините M_{RP} и M_{coRP} представят L в двата класа. Пускаме ги да смятат и двете и ако дадат противоречив резултат връщаме $'?'$, иначе връщаме сигурния резултат. Новата машина вкарва L в **ZPP**.

Ако пък $L \in \mathbf{ZPP}$ пускаме представящата го машина M_{ZPP} да смята и резултатът $'?'$ заменяме с 0 или 1 за да получим алгоритъм, който вкарва езика в **RP** или **coRP**.

Следствие 2: Класът **ZPP** се състои от езиците L, за които има вероятностна машина M_L, за която очакваната сложност (средното време за работа) е ограничена от полином, такава че:

$$Pr[M(x) = L(x)] = 1$$

Доказателство: В едната посока пускаме алгоритъмът M_{ZPP} няколко пъти докато получи точен отговор, в другата пускаме M_L да смята докато мине ограничаващият я полином и ако не е свършила сметката връщаме $'?'$.

Лема за повторенията: (става дума за клас **RP** или подобна ситуация)
Нека алгоритъмът A разпознава език L с вероятност $Pr[A(x) = L(x)] > f(n)^{-1}$, където $n = |x|$, а $f(n)$ расте неограничено. Означението $A(x) = L(x)$ тълкуваме 'A разпознава правилно, че $x \in L$ при вход x и зададена редица от случайни числа'. Нека стартираме алгоритъма $cf(n)\lg(n)$ пъти с различни случайни редици. Тогава вероятността нито едно от изпълненията да не успее $Pr[Err_A] < 1/n^c$.

Доказателство: Вероятността за единичен неуспех е по-малка от $(1-1/f(n))$, а за многократен ще е по малка съответно от $(1-1/f(n))^{cf(n)\lg(n)} \approx (1/e)^{c\lg(n)} = 1/n^c$.

От тази лема се вижда, че е достатъчно в дефинициите на **RP** и **coRP** да сменим константата $\frac{1}{2}$ с малка функция от вида $f(n)^{-1}$, където f е полином. Тази техника на многократно пускане на алгоритъм, ползващ случайност се нарича **намаляване (редукция)** на грешката.

Тест на Милер-Рабин за намиране на прости числа:

$Check\_Composite(a, n)$ //отговор $TRUE$ гарантира, че n е съставно
1) Определяме t и u такива, че u е нечетно и $n - 1 = 2^t u$.
2) $x_0 \leftarrow a^u \ mod \ n$
3) for $i \leftarrow 1$ to t
4) $\quad x_i \leftarrow x_{i-1}^2 \ mod \ n$
5) $\quad$ if $(x_i = 1) \wedge (x_{i-1} \neq 1) \wedge (x_{i-1} \neq (n-1))$
6) $\quad\quad$ return $TRUE$ //нетривиален корен на единицата
7) if $(x_t \neq 1)$ return $TRUE$ //малка теорема на Ферма
8) return $FALSE$ //вероятност числото да е просто

Алгоритъм $Miler\_Rabin(n, s)$
1) for $i \leftarrow 1$ to s
2) $\quad a \leftarrow Random(1, n-1)$
3) $\quad$ if $Check\_Composite(a, n)$
4) $\quad\quad$ return $COMPOSITE$ //гарантирано
5) return $PRIME$ //с вероятност за грешка най-много $\frac{1}{2^s}$ или $\frac{1}{4^s}$?

Минимален срез: Karger

Нека $G(V, E)$ е неориентиран граф с теглова функция $w : E \to \mathbb{R}^+$. Нека $|V| = n$, $|E| = m$.

Искаме да намерим минимален (спрямо тегловата функция w) срез (разрез) (A, B), $A \cup B = V$, $A \cap B = \emptyset$ за G.

В някои случаи за удобство ще разглеждаме G като мултиграф с единични тегла, като броят на ребрата между два върха u и v е равен на $w(u, v)$ (предполагаме, че w приема цели значения).

Сгъване на ребро (u, v) ще наричаме преобразуване на графа G, при което върховете u и v се слепват в един нов метавръх (ще приемаме, че v изчезва, а u е резултатът от сгъването), реброто (мултиребрата) между u и v изчезват, а всички останали ребра остават закачени за върхът-наследник.

Резултатът от сгъването на (u, v) ще отбелязваме с $G/(u, v)$. Ако сгъваме група ребра F, резултатът ще отбелязваме G/F. Лесно се вижда, че редът на сгъване на група ребра няма значение с точност до изоморфизъм.

Идеята на алгоритъмът на Karger е да избира и сгъва случайно избрано ребро, ако разглеждаме мултиграф с еднични ребра, или да избира ребро с вероятност, пропорционална на тегловата

функция. Тъй като ребрата в минималният срез са малко (или леки), вероятноста да бъдат избрани за сгъване е малка и може да се очаква, че графът ще се преобразува до граф с 2 метавърха, представящи множества A и B, които определят минимален срез.

Алгоритъм $Karger\_0(G(V,E),w)$
1) while $|V|>2$
2) Избери ребро (u,v) с вероятност, пропорционална на $w(u,v)$.
3) $G \leftarrow G/(u,v)$
4) return G

Лема K_1: Нека допуснем, че алгоритъмът конструира минимален срез и сумата от теглата на ребрата между 2-та последни върха на G в стъпка 4) е c (или това е едно метаребро, със сумарно тегло). c е величината на минималния срез и по време на работата на алгоритъма от всяко подмножество на G излиза сумарен поток (срез) не по-малък от c.

Доказателството провеждаме индуктивно по цикъла на алгоритъма. Когато сгъваме ребро (u,v), множеството от 2-та му върха има сумарен поток $c_{(u,v)} \geq c$, обединеният връх ще има същият капацитет и няма да наруши баланса на никое подмножество в новият граф $G/(u,v)$.

Лема K_2: Вероятността алгоритъмът $Karger\_0$ да открие минимален срез е не по-малка от $\binom{n}{2}^{-1} = \Omega(n^{-2})$.

Доказателство: Интерпретираме G като мултиграф с единични тегла на ребрата. Да допуснем, че алгоритъмът на всяка стъпка избира ребро, което не е от конкретен минимален срез (A,B) с капацитет c. Съгласно лема K_1 капацитетът на всеки връх ще е поне c, следователно за броят на всички ребра имаме неравенството $|E| \geq ck/2$, където k е броят на върховете в някой момент от работата на алгоритъма.

Вероятността да изберем ребро от среза е $\frac{c}{|E|} \leq \frac{2}{k}$, а вероятността да изберем ребро, което не е от среза ще е поне $1-\frac{2}{k}$.

За всички преминавания през цикъла на $Karger\_0$ вероятността да не сгънем ребро от среза ще е поне

$$(1-\frac{2}{n})(1-\frac{2}{n-1})(1-\frac{2}{n-2})\dots(1-\frac{2}{3}) = (\frac{n-2}{n})(\frac{n-3}{n-1})(\frac{n-4}{n-2})\dots(\frac{1}{3}) = \frac{2}{n(n-1)} = \binom{n}{2}^{-1}$$

По-нататък ще ползваме леко модифицирана версия на $Karger\_0$, тя ще цикли докато в графа останат не 2, а k върха (параметърът k е малък спрямо n):

Алгоритъм $Karger\_1(G(V,E),w,k)$
1) while $|V|>k$
2) Избери ребро (u,v) с вероятност, пропорционална на $w(u,v)$.
3) $G \leftarrow G/(u,v)$
4) return G

Лема K_3: Вероятността алгоритъмът $Karger\_1$ да не сгъне ребро от конкретен минимален срез е поне $\binom{k}{2}/\binom{n}{2} = \Omega(k^2/n^2)$.

Доказателство: Повтаряме сметката от лема K_2.

Сложност на $Karger\_0$ и $Karger\_1$:
Ако представим ребрата и тегловата функция с матрица на съседства $W_{i,j}$ и поддържаме масив от сумарните тегла $D[i] = \sum_j W_{i,j}$, всяко преминаване през цикъла избор, сгъване на ребро може да се извърши за време $\Theta(n)$ така:
1) Първо избираме случайно ред i с вероятност пропорционална на $D[i]$.

2) Избираме ребро (i, j) с вероятност пропорционална на $W_{i,j}$.
3) Сгъването свеждаме до сумиране на ред/стълб j към i,
4) последвано от нулиране на ред/стълб j в матрицата $W_{i,j}$.

Всяка от стъпките по-горе се извършва за време $\Theta(n)$, основният цикъл на двата алгоритъма се върти $\Theta(n)$ пъти, оттук общата сложност на алгоритъма е $\Theta(n^2)$.

За да намерим минимален срез с висока вероятност, трябва да пускаме $Karger\_0$ много пъти, примерно $cn^2 \lg(n)$.

От сметката в лема K_2 се вижда, че когато върховете в G са много, вероятността за грешка е малка и расте при намаляване на графа.

Можем да спестим изчислителна работа, ако при голямо n пускаме малко на брой копия на алгоритъма, а с намалянето на n увеличаваме броят на работещите копия.

Резултатът от тази идея е:

Алгоритъм $Karger(G(V, E), w)$
1) if $n < 6$
2) $\quad G' \leftarrow Karger\_1(G(V, E), w, 2)$
3) $\quad$ return G' // G' има 2 метавърха, вероятен минимален срез
4) for $i \leftarrow 1$ to 2 // Стартираме рекурсивно 2 копия
5) $\quad G' \leftarrow Karger\_1(G(V, E), w, \lceil \frac{n}{\sqrt{2}} \rceil + 1)$
6) $\quad G_i \leftarrow Karger(G'(V, E), w)$
7) return $min_{cut(G_i)}\{G_1, G_2\}$

Сложност:
Да означим с $T(n)$ сложността по време на алгоритъма $Karger$.
Тя удовлетворява рекурентната формула $T(n) = 2T(\lceil \frac{n}{\sqrt{2}} \rceil + 1) + cn^2$.
Решението на тази зависимост (Мастер теорема) е $T(n) = \Theta(n^2 \lg(n))$.

Броят върхове $\lceil \frac{n}{\sqrt{2}} \rceil + 1$, които остават несвити на ред 5) не е случаен, той дава вероятност за оцеляване на конкретен минимален срез поне $1/2$ при преминаването на сгъванията от ред 5). Лема K_3 се прилага тук.

Лема K_4: Вероятността алгоритъмът $Karger$ да не сгъне ребро от конкретен минимален срез е $\Omega(1/\lg(n))$.

Доказателството на лемата използва тънки рекурентни сметки и посоченото по-горе съображение за оцеляване на среза при преминаване на ред 5) с вероятност поне $1/2$. Няма да го пиша тука, сложничко е !

Теорема за сложността на алгоритъма на Каргер: За да достигнем произволно малка грешка е достатъчно да стратираме алгоритъма $Karger$ $c \lg^2(n)$ пъти. Общата сложност на многократното изпълнение е $\Theta(n^2 \lg^3(n))$.

Доказателство: Следва от анализа на сложността на единичното изпълнение, лема K_4 и лемата за повторенията.

Каргер посочва най-добро време за детерминиран алгоритъм към 1996г. оценка $O(mn+n^2 \lg(n))$ за алгоритъм от 1992г. на Nagamochi-Ibaraki.

Библиография

[1] К. Манев: Увод в дискретната математика, КЛМН, четвърто издание, 2005.

[2] Minko Markov: Problems with solutions in the Analysis of Algorithms, Draft, 2012

[3] Cormen, Leiserson, Rivest, Stein: Introduction to Algorithms

[4] Herbert S. Wilf: Algorithms and Complexity

[5] Avrim Blum: лекции CMU 15-451 (Algorithms), Fall 2009

[6] Dasgupta, Papadimitriou, and Vazirani: Algorithms, 2006

[7] Пападимитриу, Стайглиц: Комбинаторная оптимизация, 1985 (превод от английски)

[8] Michael Garey, David Johnson: Computers and Intractability (A guide to the theory of NP-completeness), 1979

[9] Скелет: Въведение в изчисленията и алгоритмите

[10] Sanjeev Arora, Boaz Barak: Computational Complexity, draft, 2007