АЛГОРИТМИ ВЪРХУ ГРАФИ

Граф се нарича наредена двойка $G = \big(V; E\big)$, където $V = \big\{v_1\ ;\ v_2\ ;\ v_3\ ;\ \ldots\ ;\ v_{n-1}\ ;\ v_n\big\}$ е непразно крайно множество, съставено от произволни елементи, а E е крайно множество, съставено или само от наредени, или само от ненаредени двойки от различни елементи на V; графът съответно се нарича ориентиран или неориентиран. Елементите на множествата V и E се наричат съответно върхове и ребра на графа; $n = |V|$ е броят на върховете на графа, $n > 0$; $m = |E|$ е броят на ребрата на графа, $m \geq 0$. Числата n и m са цели. В сила са неравенствата $m \leq \dfrac{n(n-1)}{2}$ при неориентиран граф и $m \leq n(n-1)$ при ориентиран граф; винаги $m \preceq n^2$; в първите две има равенство само за пълни графи — които съдържат всички възможни ребра. Асимптотичното неравенство се обръща в равенство за всички пълни и за някои непълни графи; графите, за които е изпълнено асимптотичното равенство $m \asymp n^2$, се наричат плътни графи. Граф без ребра се нарича празен граф: $E = \varnothing$, $m = 0$.

Определението не допуска да има две или повече еднопосочни ребра между една и съща двойка върхове. Ако тази забрана отпадне, получаваме мултиграф; той съдържа кратни ребра и E е мултимножество, а m е неограничено отгоре. Тук няма да разглеждаме мултиграфи.

Определението забранява ребра със съвпадащи краища (примки). В теорията на графите се допускат такива ребра, но тук ще разглеждаме само графи без примки.

В теорията се изучават също безкрайни графи, обаче тук няма да се занимаваме с тях, а ще работим само с крайни графи.

Дотук говорим за нетегловни графи, но има и тегловни — на ребрата (по-рядко на върховете) се съпоставят числа, наречени техни тегла. Теглата могат да се тълкуват по различни начини: дължини, пропускателни способности, цени и т.н.

Върховете на граф се изобразяват с точки, а ребрата — с дъгички, свързващи точките. Но графът е нещо много повече от своето нагледно представяне. Той не е геометричен обект. Графите са подходящи за моделиране на различни задачи от всички области на практиката: върховете представят някакви обекти, а ребрата — двучленни отношения между тези обекти. Някои от тези практически задачи са доста трудни, което обяснява нуждата от алгоритми.

Алгоритмите върху графи имат сложност по време $T(n; m)$ и сложност по памет $M(n; m)$, които са функции на два аргумента.

Някои алгоритми върху неориентирани графи предполагат, че входният граф е свързан, тоест че има път между всеки два различни върха на графа (иначе алгоритъмът се изпълнява върху отделните компоненти на свързаност). Ако неориентиран граф е свързан, то $m \geq n - 1$ (равенство се достига само ако графът е дърво), следователно $m = \Omega(n)$. Съчетаваме този извод с асимптотичното неравенство по-горе и получаваме двойното неравенство

$$n \preceq m \preceq n^2.$$

След логаритмуване то приема вида

$$\log n \preceq \log m \preceq 2 \log n,$$

откъдето следва, че

$$\log m \asymp \log n.$$

Затова при анализ на асимптотичния порядък на сложността на алгоритми върху свързани графи не се прави разлика между $\log m$ и $\log n$.

За ориентирани графи е по-подходящо понятието силна свързаност. Два различни върха се наричат силно свързани, ако има път и от първия до втория, и от втория до първия връх. Тази релация е симетрична и транзитивна, а ако всеки връх се смята за силно свързан със себе си, релацията ще бъде и рефлексивна. Тоест силната свързаност е релация на еквивалентност. Нейните класове на еквивалентност носят името компоненти на силна свързаност на графа. Един граф се нарича силно свързан, ако притежава само една компонента на силна свързаност. За всеки силно свързан граф важи неравенството $m \geq n$; равенство се достига само за граф, който е ориентиран цикъл. Оттук, както по-горе, следва, че $\log m \asymp \log n$.

ПРЕДСТАВЯНИЯ НА ГРАФ

В компютърните програми се използват най-често следните две представяния на граф: матрица на съседство и списъци на съседство. Съседни се наричат върховете, свързани с ребро.

Матрица на съседство

Матрицата на съседство има n реда и n стълба и се състои от цели неотрицателни числа: в пресечната клетка на ред № i и стълб № j стои броят на ребрата, които излизат от връх № i и влизат във връх № j на графа. По този начин могат да се представят и мултиграфи с примки.

Най-важното свойство на матрицата на съседство е, че елементът в ред № i и стълб № j на нейната k-та степен е точно броят на пътищата с дължина k, които започват от връх № i и завършват във връх № j. Това свойство се доказва с индукция по k. Дължина на път наричаме броя на ребрата в него (върховете са с един повече). Пътят може да повтаря върхове и ребра.

Графите без кратни ребра имат двоична матрица на съседство (само нули и единици). Ако графът е без примки, то матрицата на съседство съдържа само нули по главния диагонал. Неориентираните графи имат симетрична матрица на съседство.

Дотук говорихме за нетегловни графи. Тегловните графи също могат да се представят с матрица на съседство. Сега тя съдържа теглата на ребрата, а те са произволни реални числа (където няма ребро, стои някаква служебна стойност на теглото). Такава матрица на съседство не притежава важното свойство, цитирано по-горе.

Независимо от конкретните подробности, матрицата на съседство изразходва памет $\Theta\big(n^2\big)$. За разредени графи (тоест за графи с малко ребра) това е твърде неикономично представяне: има много нули и малко единици. Ще спестим памет, ако представим само единиците.

Списъци на съседство

Друго представяне на граф е да пазим за всеки връх u един списък на съседство $\mathrm{Adj}(u)$, съставен от ребрата, излизащи от върха u. Ако те не съдържат други данни (например тегла), списъкът може да се състои просто от другите краища на ребрата (краищата, различни от u). Това представяне изисква памет $\Theta(n)$ за указателите към списъците (по един за всеки връх) плюс памет $\Theta(m)$ за елементите на всички списъци общо (по един елемент за всяко ребро); получава се количество памет $\Theta(n+m)$ за цялата структура от данни.

Тъй като $m = \mathrm{O}\big(n^2\big)$, то $n+m = \mathrm{O}\big(n^2\big)$, тоест списъците на съседство са по-икономични от матрицата на съседство. Ето защо ще представяме графите само чрез списъци на съседство. Тогава за алгоритъм, обработващ граф, дължината на входа е от порядък $\Theta(n+m)$. Затова, ако сложността на алгоритъма е $\Theta(n+m)$, тя се смята за линейна, нищо че може $m = \Theta\big(n^2\big)$, което влече $n+m = \Theta\big(n^2\big)$; това е линейна, а не квадратична сложност (при плътни графи), защото сложността се мери спрямо дължината на входа. Обаче $\Theta\big(n^2\big)$ е квадратична сложност за разредени графи — когато $m = \Theta(n)$ и $n+m = \Theta(n)$.

ОБХОЖДАНЕ НА ГРАФ

Съществуват два вида обхождане на граф: обхождане в ширина и обхождане в дълбочина. Те се прилагат по еднакъв начин за всякакви графи — както ориентирани, така и неориентирани. За някои класове от графи (например за дървета) има и други видове обхождане.

Двата вида обхождане на граф често се наричат търсене в ширина и търсене в дълбочина. Причината е, че понякога търсим връх, ребро или път с едни или други свойства.

При търсенето в ширина първо обхождаме преките наследници на текущия връх на графа, а чак след това — техните преки наследници.

При търсенето в дълбочина, когато обхождаме пряк наследник на текущия връх на графа, първо обхождаме наследниците на наследника, а после — другите наследници на текущия връх.

Целта на обхождането е да преминем по веднъж през всеки връх и през всяко ребро. Може да попаднем в един връх много пъти, но го обработваме веднъж — при първото достигане. Ако за всеки връх v запазим указател $\pi[v]$ към върха, от който за пръв път сме дошли във v, ще получим гора от коренови дървета; всяко от тях е представено чрез указатели към родителите: $\pi[v]$ е родител на v; корен е върхът, от който е започнало обхождането; коренът няма родител. Получената структура се нарича гора на обхождането на графа.

Ако графът е ориентиран, гората на обхождането е безполезна. За неориентиран граф обаче гората на обхождането показва кои са компонентите на (слаба) свързаност на графа: всяко дърво съдържа всички върхове от една компонента на свързаност. При свързан неориентиран граф гората се състои от едно-единствено дърво, което се нарича дърво на обхождането.

За да обработваме всеки връх по веднъж, трябва да помним кои върхове са били посетени. Често можем да минем с две състояния за всеки връх — посетен и непосетен. Но в някои задачи е по-удобно да използваме по три състояния за всеки връх — непосетен, отворен и затворен. Отначало всички върхове на графа са непосетени. Отваряме връх при първото му достигане и го затваряме при завършване на обработката му. Когато отваряме някой връх, го добавяме в подходяща структура от данни за бъдеща обработка. Когато извадим връх от тази структура, го обработваме и го затваряме. При обхождане в ширина подходящата структура е опашка, а при обхождане в дълбочина — стек. В практиката рядко използваме явно деклариран стек; по-често разчитаме на програмния стек, който при рекурсия се попълва автоматично.

В общия случай върховете и ребрата са обекти с много характеристики — имена, тегла и др. Но за простота на кода можем да приемем, че върховете са целите числа от 1 до n включително, а ребрата са само двойки от върхове. Тоест множеството от върховете е $V = \{ 1 ; 2 ; 3 ; \ldots ; n \}$. Функцията $\mathrm{Adj}(\bullet)$ представлява обикновен масив $\mathrm{Adj}[1 \ldots n]$, а самото наименование Adj може да се схваща като псевдоним на E — множеството от ребрата на графа. Когато е даден граф, ще смятаме, че са дадени също броят n на върховете му и броят m на ребрата му.

Обхождане в ширина $\big(G(V; E)$: граф$\big)$

```
 1)  Q ← празна опашка от върхове
 2)  π[1 ... n]: масив от върхове
 3)  σ[1 ... n]: масив от състояния
 4)  for each r ∈ V do
 5)      σ[r] ← непосетен
 6)  for each r ∈ V do
 7)      if σ[r] ≠ непосетен
 8)          continue
 9)      π[r] ← NIL
10)      σ[r] ← отворен
11)      Q.Add(r)
12)      while not Q.IsEmpty do
13)          u ← Q.Extract
14)          print "Връх u: начало."
15)          for each v ∈ Adj[u] do
16)              if σ[v] ≠ непосетен
17)                  continue
18)              π[v] ← u
19)              σ[v] ← отворен
20)              print "Ребро (u; v)."
21)              Q.Add(v)
22)          print "Връх u: край."
23)          σ[u] ← затворен
24)  return π[1 ... n]
```

Обхождане в дълбочина $\big(G(V; E)$: граф$\big)$

```
 1)  // Рекурсията използва програмния стек.
 2)  π[1 ... n]: масив от върхове
 3)  σ[1 ... n]: масив от състояния
 4)  for each r ∈ V do
 5)      σ[r] ← непосетен
 6)  for each r ∈ V do
 7)      if σ[r] ≠ непосетен
 8)          continue
 9)      π[r] ← NIL   // нулев указател
10)      Rec(G(V; E), r, π[1 ... n], σ[1 ... n])
11)  return π[1 ... n]
```

Rec $\big(G(V; E)$, u, $\pi[1 \ldots n]$, $\sigma[1 \ldots n]\big)$

```
 1)  // Рекурсивно обхождане от връх u.
 2)  σ[u] ← отворен
 3)  print "Отваряне на връх u."
 4)  for each v ∈ Adj[u] do
 5)      if σ[v] ≠ непосетен
 6)          continue
 7)      π[v] ← u
 8)      print "Ребро (u; v)."
 9)      Rec(G(V; E), v, π[1 ... n], σ[1 ... n])
10)  print "Затваряне на връх u."
11)  σ[u] ← затворен
```

За масива $\sigma[1 \ldots n]$ изразходваме допълнителна памет $\Theta(n)$ при всякакви входни данни. Не броим паметта за масива $\pi[1 \ldots n]$, защото този масив е изход и се използва само за писане. За опашката или стека трябва допълнителна памет в размер $\Theta(n)$ при най-лоши входни данни. Другите променливи изразходват незначително количество памет (колкото за един-два върха). Следователно сложността по памет и на двата вида обхождане на граф е $M(n; m) = \Theta(n)$ при всякакви входни данни. Но докато опашката и масива от състоянията можем да заделим от динамичната памет, която обикновено е налична в голямо количество, то програмният стек най-често е с малки размери и лесно се препълва при дълбока рекурсия. Затова е препоръчително обхождането в дълбочина да се реализира с явно деклариран стек, използващ динамична памет.

Сложността по време и на двата вида обхождане на граф е линейна: $T(n; m) = \Theta(n + m)$ при всякакви входни данни, ако са изпълнени някои изисквания:

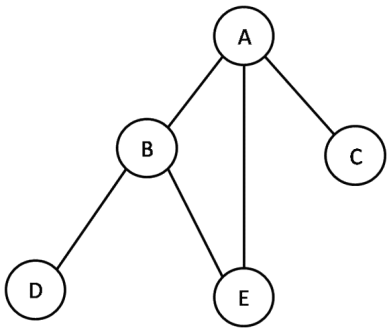
— Първо, командите IsEmpty (проверка за празна опашка), Add (добавяне на връх в края ѝ) и Extract (извличане на връх от началото ѝ) трябва да работят за време $\Theta(1)$ при всякакви данни. Същото важи за добавянето и премахването на елементи от стека, ако се използва явен стек. Това изискване може да се гарантира от програмната реализация на типовете опашка и стек.

— Второ, обхожда се целият граф. Ако е възможно търсенето да се прекрати предсрочно, тогава времевата сложност може да е по-малка или същата, но само при най-лоши входни данни (когато алгоритъмът претърси графа изцяло, за да установи, че не съдържа връх или ребро с желаните характеристики).

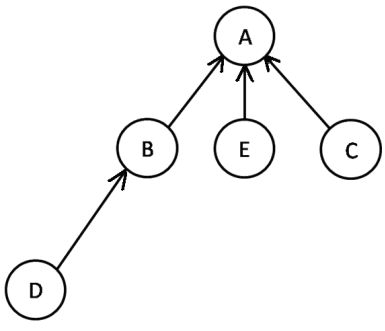
— Трето, всеки връх и всяко ребро се обработват за време $\Theta(1)$. Иначе времевата сложност може да бъде по-голяма от посочената.

Обхождането в дълбочина и обхождането в ширина са алгоритмични схеми, а не алгоритми. От тези схеми се получават алгоритми, когато бъде уточнена обработката на върховете и ребрата. В псевдокода по-горе липсва истинска обработка; тя е заменена символично с оператори за печат. В истински алгоритъм трябва да заменим операторите **print** с действителна обработка на данни. Допустимо е да премахнем някои оператори за печат, без да ги заменяме с други команди (ако например търсим връх, можем да не обработваме ребрата; и обратно). Разрешени са също някои малки размествания, например редове № 18, № 19, № 20 и № 21 от обхождането в ширина, както и редове № 7 и № 8 от рекурсивния подалгоритъм Rec на обхождането в дълбочина могат да се изпълняват в произволен ред. Същото важи за редове № 22 и № 23 от търсенето в ширина, както и редове № 2 и № 3 от Rec, а също редове № 10 и № 11 от Rec. Освен това всеки алгоритъм може да добави инициализация на собствени (т.е. допълнителни) локални променливи.

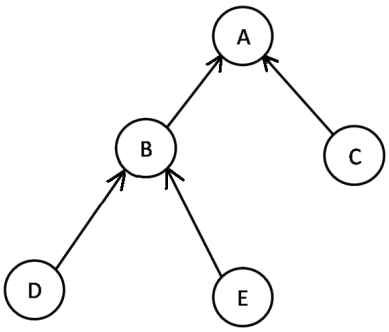
Пример:



Входни данни:
неориентиран граф.



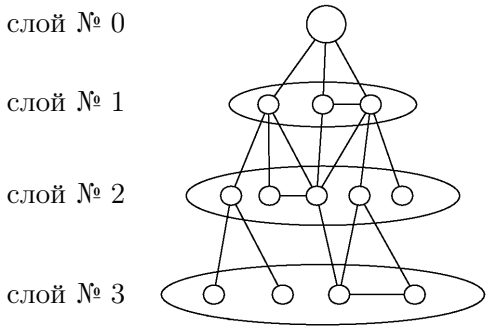
Обхождане в ширина:
дърво на обхождането.
Върховете се обхождат
в реда A, B, E, C, D.



Обхождане в дълбочина:
дърво на обхождането.
Върховете се обхождат
в реда A, B, D, E, C.

Обхождането в ширина разделя върховете по слоеве според отдалечеността им от корена; така намира най-къси пътища от корена до всички други върхове. Корен в примера е върхът A, който е сам в слой № 0; върховете B, E и C от слой № 1 се намират на разстояние 1 от върха A; върхът D от слой № 2 е на разстояние 2 от корена A. Обхождането в дълбочина не гарантира намирането на най-къси пътища (в примера то открива път от A до E с дължина 2 вместо 1).

Разделянето на върховете на графа в слоеве, което е резултат от обхождането в ширина, притежава следното свойство: краищата на всяко ребро са от един слой или от съседни слоеве.



Доказателство: Номерът на всеки слой е равен на дължината на произволен най-къс път от корена до кой да е връх от този слой. Нека краищата на едно ребро са от слоеве № k и № ℓ. Ако $k = \ell$, двата края са от един слой. Нека $k \neq \ell$. Едното число е по-голямо от другото. Без ограничение нека $\ell > k$. Следователно от корена до върха на реброто, който е от слой № k, има път с дължина k. Присъединяваме реброто и получаваме път с дължина $k+1$ от корена до върха от слой № ℓ. Тъй като ℓ е най-малката дължина на път от корена до този връх, то следва, че $\ell \leq k+1$. Неравенствата $k < \ell \leq k+1$ заедно с факта, че k и ℓ са цели числа, водят до извода, че $\ell = k+1$, тоест двата върха на реброто са от съседни слоеве.

АЛГОРИТМИ ВЪРХУ НЕТЕГЛОВНИ ГРАФИ

В този раздел ще разгледаме някои алгоритми върху нетегловни графи. Тези алгоритми ще извършват обхождане в ширина или в дълбочина, затова ще притежават характеристиките на двете алгоритмични схеми. По-конкретно, сложността по памет и по време ще бъде линейна: $M(n;m) = \Theta(n)$ и $T(n;m) = \Theta(n+m)$ при всякакви входни данни, освен ако е казано друго.

Алгоритми, независещи от вида на обхождането

При някои алгоритми няма значение как обхождаме графа — в ширина или в дълбочина:

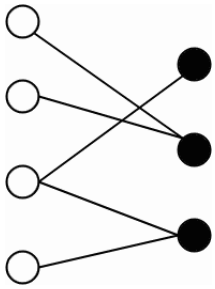
- *Обръщане на посоката на ребрата.* Даден е ориентиран граф. Да се построи друг граф със същите върхове, но ребрата му да сочат в обратна посока. Тази задача възниква, когато се търси обратната на дадена бинарна релация (родител — дете; длъжник — заемодател). Задачата се решава с обхождане на дадения граф: за $\forall v \in \mathrm{Adj}(u)$ добавяме u към $\mathrm{Adj}^*(v)$, където Adj^* са списъците на съседство на обърнатия граф (отначало те са празни).

- *Превръщане на ориентиран граф в неориентиран.* Обхождаме дадения ориентиран граф и за $\forall v \in \mathrm{Adj}(u)$ добавяме върха v към $\mathrm{Adj}^*(u)$, а върха u — към $\mathrm{Adj}^*(v)$, където Adj^* са списъците на съседство на неориентирания граф (отначало те са празни).

- *Търсене по ключ.* Еднократно обхождаме дадения граф (ориентиран или неориентиран) и намираме върха или реброто със стойност, равна на търсения ключ (ако съществуват). Тук стойността не се схваща като тегло (дължина, ширина и т.н.), а просто се пази в графа. Времето е $\Theta(n+m)$ при претърсване на ребрата и $\Theta(n)$ при претърсване на върховете, и то при най-лоши входни данни — например когато графът не съдържа търсения ключ.

- *Компонентите на (слаба) свързаност* на неориентиран граф съответстват на дърветата от гората на обхождане: имат едни и същи върхове (дърветата може да са с по-малко ребра). Ако графът е ориентиран, трябва първо да се премахне ориентацията на ребрата.

- *Проверка за двуделност.* Времето ѝ е $\Theta(n+m)$ при най-лоши данни — двуделен граф.



Един граф (ориентиран или неориентиран) се нарича двуделен, ако върховете му могат да се оцветят с два цвята по такъв начин, че краищата на всяко ребро да са с различен цвят. (За празния граф е достатъчен един цвят, но този случай е тривиален.)

Можем да намерим оцветяването на върховете (ако има такова) чрез обхождане на графа, като видът на обхождането няма значение. Когато присвояваме нулев указател към родителя на произволен връх, оцветяваме върха в бяло. Когато правим върха u родител на върха v, оцветяваме v в черно, ако върхът u е бял, и обратно; това се прави само ако върхът v току-що е бил посетен за пръв път. Ако не е така, т.е. ако v е бил посетен по-рано, не го оцветяваме повторно, а проверяваме наличния цвят на v: ако е еднакъв с цвета на u — спираме обхождането предсрочно, защото графът не е двуделен; ако u и v имат различни цветове — продължаваме обхождането, докато намерим оцветяване.

Този алгоритъм работи правилно само върху неориентирани графи.

Всеки свързан двуделен граф има точно две оцветявания: началния връх на обхождането можем да оцветим не само в бяло, но и в черно; променят се цветовете и на другите върхове, тоест едното оцветяване е негатив на другото.

Ето защо всеки двуделен граф с k компоненти на свързаност има точно 2^k оцветявания. Няма бърз алгоритъм за намиране на всичките (тривиална долна граница по размера на изхода). Алгоритъмът, описан по-горе, е бърз, защото намира само едно оцветяване.

Въпреки че ориентираните графи могат да бъдат двуделни, все пак посоката на ребрата няма никакво значение за оцветяването, затова често я пренебрегваме.

Теорема на Кьониг: Един граф не е двуделен тогава и само тогава, когато съдържа неориентиран цикъл с нечетна дължина.

Доказателство: Достатъчността е очевидна, защото, ако допуснем, че графът е двуделен, то всеки неориентиран цикъл редува цветовете на върховете, т.е. съдържа четен брой върхове, а това противоречи на съществуването на неориентиран цикъл с нечетна дължина.

Необходимостта следва от предложения алгоритъм за оцветяване. Ако графът не е двуделен, алгоритъмът намира ребро с едноцветни краища. Щом са едноцветни, те са били достигнати от началния връх за брой стъпки с еднаква четност. Пътищата, по които са били достигнати, и реброто между двата едноцветни върха образуват цикъл с нечетна дължина.

Приложения на търсенето в ширина

Обхождането в ширина има следните приложения:

- *Търсене на най-къс път.* Върху свързан нетегловен граф (ориентиран или неориентиран) търсенето в ширина намира най-къси пътища от първия връх в списъка на върховете до всички други върхове на графа. Тъй като можем да променим подредбата на списъка, търсенето в ширина е способно да намери най-къси пътища от произволно зададен връх до всички други върхове. По-точно, дървото на обхождането в ширина е същевременно дърво на най-къси пътища, но те се получават в обратна посока — от крайните върхове към началния връх (корена), защото дървото е представено чрез указатели към родителите (това има значение, когато графът е ориентиран).

  Ако графът е несвързан, тогава обхождането в ширина построява гора на най-къси пътища от един връх във всяка компонента на свързаност до всички други върхове в нея.

  За горните два варианта времевата сложност е $\Theta(n+m)$ при всякакви входни данни.

  Възможно е да се търси най-къс път от един даден връх до друг даден връх на графа; сега търсенето се прекратява при достигане на крайния връх, затова времевата сложност е $\Theta(n+m)$ при най-лоши входни данни.

- *Търсене на максимален поток в граф.* Извършва се с алгоритъма на Форд—Фалкерсон, а той използва обхождане в ширина, но е твърде сложен за настоящия кратък обзор.

Приложения на търсенето в дълбочина

Обхождането в дълбочина има следните приложения:

- *Търсене на цикъл.*

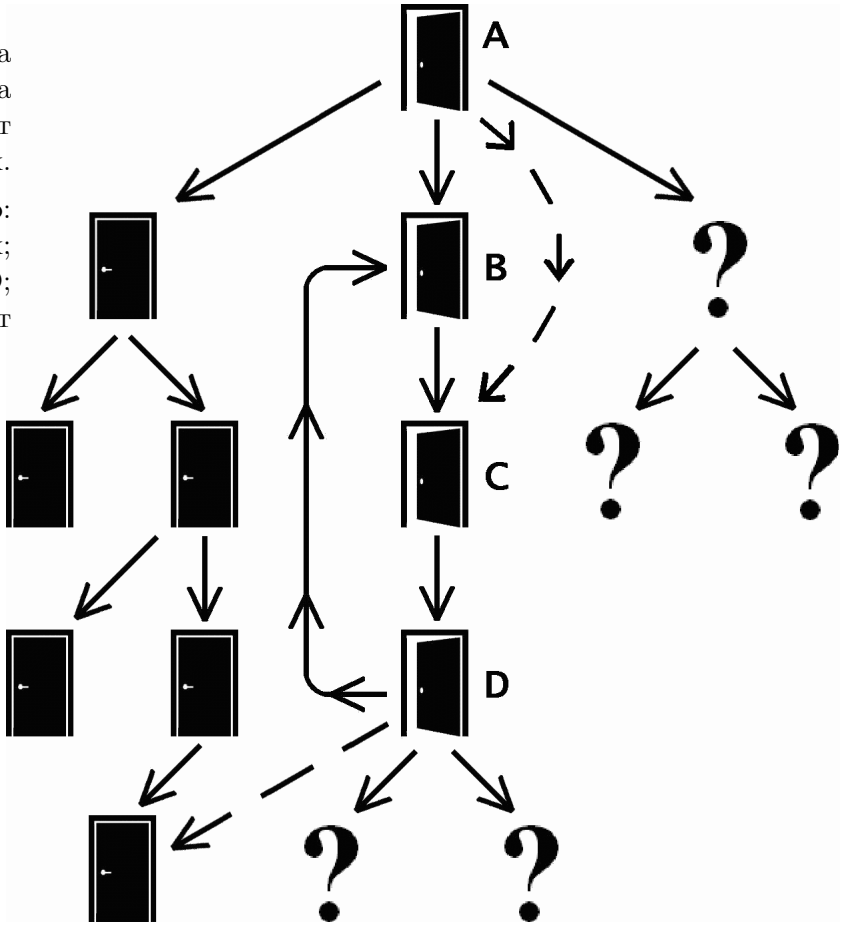
  При обхождане в дълбочина отворените върхове на графа във всеки миг образуват път от корена до текущия връх.

  На чертежа е даден пример: A — корен; D — текущ връх; отворени върхове: A, B, C, D; въпросителните изобразяват непосетените върхове.

  Ребрата са четири вида: ребра напред, дървесни, обратни и странични.

  Дървесни са тези ребра, които образуват дървото на обхождането. Те сочат към преките наследници в дървото (а не в графа). На чертежа са показани с плътни прави стрелки (например AB).

  Обратните ребра сочат към преките и косвените предшественици в дървото. На чертежа такова ребро е DB (плътната заоблена линия).



  Ребрата напред сочат косвени наследници в дървото (прекъснатата заоблена линия AC).

  Страничните ребра указват върхове, които не са нито наследници, нито предшественици в дървото — било преки, било косвени. На чертежа странично ребро излиза от връх D (прекъснатата права линия).

  Ако едно ребро на графа излиза от текущия връх, то се класифицира така:
  — дървесно ребро — ако сочи към непосетен връх;
  — обратно ребро — ако сочи към отворен връх;
  — ребро напред или странично ребро — ако сочи към затворен връх.

  Откриването на обратно ребро е признак за наличието на цикъл. Цикълът е образуван от обратното ребро и от дървесните ребра по пътя до текущия връх от предшественика, посочен от обратното ребро. На чертежа обратно ребро е DB; то затваря цикъла DBCD. (Когато се разглеждат графи с примки, за обратни ребра се смятат и примките.)

  Този алгоритъм търси цикли не само в ориентирани, но също и в неориентирани графи, само че при работа върху неориентирани графи няма ребра напред и странични ребра и трябва да внимаваме да не приемем за обратно реброто, по което току-що сме минали.

  Алгоритъмът има сложност по памет $\Theta(n)$ заради състоянията на върховете. Това важи при всякакви входни данни и независимо дали графът е ориентиран, или неориентиран. Но времевата сложност зависи от входните данни: за време $\Theta(1)$ се открива къс цикъл, чиито върхове са разположени в началото на списъците, които ги съдържат. Ето защо е подходящо да разгледаме времевата сложност при най-лоши входни данни.

Най-лоши входни данни са ацикличните графи (както ориентирани, така и неориентирани), защото единственият начин алгоритъмът да установи липсата на цикъл е чрез обхождане на всички ребра (следователно и всички върхове) на графа, а това изисква време $\Theta(n+m)$. Обхождането не може да се прекрати предсрочно, защото до последния миг е невъзможно алгоритъмът да бъде сигурен в липсата на цикъл: всяко ребро, включително последното, може да се окаже обратно ребро (да сочи например към корена).

— Ако графът е ориентиран, той може да бъде ацикличен при всички допустими n и m: номерираме върховете и насочваме всички ребра от по-малък към по-голям номер. Затова времевата сложност е $\Theta(n+m)$ при всички допустими n и m.

— Ако графът е неориентиран, той може да е ацикличен само при $m < n$. Причината е, че всеки ацикличен неориентиран граф е гора, а всяка гора съдържа $m = n - k$ ребра, където k е броят на дърветата. При $m < n$ важи направеният анализ на най-лошия случай и времевата сложност е $\Theta(n+m)$, обаче от $m < n$ следва $m = \mathrm{O}(n)$ и $n + m = \Theta(n)$, затова времевата сложност излиза $\Theta(n)$. Ако $m \geq n$, то графът непременно е цикличен и направеният по-горе анализ на най-лошия случай губи сила: алгоритъмът не е длъжен да провери всичките m ребра. Най-късно на n-тото ребро редицата от отворените върхове ще се зацикли, така че алгоритъмът приключва работа за време $\Theta(n)$ в най-лошия случай.

Окончателно, времевата сложност при най-лоши входни данни на разгледания алгоритъм е $\Theta(n+m)$ за ориентирани графи и $\Theta(n)$ за неориентирани.

Забележка 1: Алгоритъмът търси един цикъл, а не всички цикли. Броят на всички цикли може да е експоненциален, така че няма бърз алгоритъм за откриване на всички цикли.

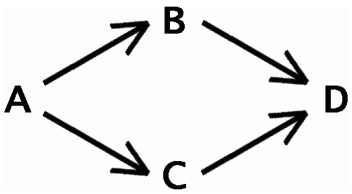
Забележка 2: Един цикъл може да бъде търсен и с обхождане в ширина, но алгоритъмът е по-труден. Времевата му сложност е линейна, обаче константният множител пред нея е по-голям в случай на ориентиран граф, защото се правят няколко обхождания.

- *Топологично сортиране* означава да номерираме върховете на ориентиран ацикличен граф по такъв начин, че всяко ребро да сочи от връх с по-малък към връх с по-голям номер. За граф, съдържащ цикъл, задачата няма решение; за неориентиран граф тя няма смисъл. За ориентиран ацикличен граф задачата се решава така: обхождаме графа в дълбочина и подреждаме върховете в ред, обратен на затварянето. Времевата сложност на този метод е $\Theta(n+m)$ при най-лоши входни данни — когато графът е ацикличен. За граф с къс цикъл алгоритъмът може да завърши бързо, ако рано попадне на цикъла.

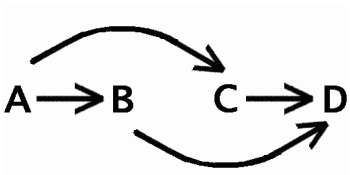
  Вместо да номерираме върховете на графа, ще ги подреждаме водоравно по такъв начин, че всички ребра да сочат отляво надясно. Всеки ориентиран ацикличен граф притежава поне една подредба; тя е единствена, ако и само ако графът съдържа хамилтонов път.

  Броят на възможните подредби може да е експоненциален, затова няма бърз алгоритъм за откриване на всички подредби. Описаният метод е бърз, защото открива една подредба. Коя именно — зависи от реда на върховете и ребрата в списъците V и Adj.

  П р и м е р :



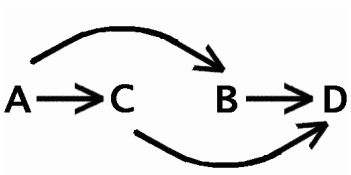
Входни данни:
ориентиран ацикличен граф.



Възможно обхождане:

((( )) ( ))

A C D D C B B A



Възможно обхождане:

((( )) ( ))

A B D D B C C A

Обхождането в дълбочина поражда низ от правилно вложени отварящи и затварящи скоби, съответстващи на отварянето и затварянето на върхове.

Макар че топологично сортиране може да се извърши само за ацикличен ориентиран граф, не е нужна предварителна проверка за ацикличност: това води до две обхождания на графа. Топологичното сортиране и проверката за ацикличност могат да се проведат с едно и също обхождане в дълбочина: то или обхожда графа докрай, подреждайки върховете правилно, или приключва предсрочно, откривайки цикъл.

```
Топологично сортиране с проверка за ацикличност (G(V; E): ориентиран граф)
 1)  ℓ[1...n]: масив от върхове      // изход
 2)  π[1...n]: масив от върхове      // локален масив
 3)  σ[1...n]: масив от състояния    // локален масив
 4)  for each v ∈ V do
 5)      σ[v] ← непосетен
 6)  k ← n
 7)  for each v ∈ V do
 8)      if σ[v] ≠ непосетен
 9)          continue
10)      π[v] ← NIL   // нулев указател
11)      k ← Rec(G(V; E), v, k, π[1...n], ℓ[1...n], σ[1...n])
12)      if k < 0
13)          k ← −k
14)          return false, ℓ[k...n]
15)  return true, ℓ[1...n]

Rec(G(V; E), u, k, π[1...n], ℓ[1...n], σ[1...n])
 1)  σ[u] ← отворен
 2)  for each v ∈ Adj[u] do
 3)      switch
 4)          case σ[v] = отворен  // обратно ребро ⇒ има цикъл
 5)              k ← n
 6)              while u ≠ v do
 7)                  ℓ[k] ← u
 8)                  k ← k − 1
 9)                  u ← π[u]
10)              ℓ[k] ← u
11)              return −k
12)          case σ[v] = непосетен  // дървесно ребро
13)              π[v] ← u
14)              k ← Rec(G(V; E), v, k, π[1...n], ℓ[1...n], σ[1...n])
15)              if k < 0
16)                  return k
17)  σ[u] ← затворен
18)  ℓ[k] ← u
19)  return k − 1
```

Както по-рано, $\pi[1 \ldots n]$ пази дървото на обхождането чрез указатели към родителите, но сега π е локална променлива, а не изход. Изход на главния алгоритъм е масивът $\ell[1 \ldots n]$ (или част от него) плюс логическа стойност: "истина" — ако даденият граф е ацикличен (тогава $\ell[1 \ldots n]$ съдържа подредбата на върховете в резултат на топологичното сортиране); "лъжа" — ако има цикъл в графа (тогава $\ell[k \ldots n]$ съдържа цикъла). Изходът се попълва отзад напред и k е текущият индекс — вход-изход на рекурсивния подалгоритъм, който връща отрицателен индекс като сигнал за откриването на цикъл.

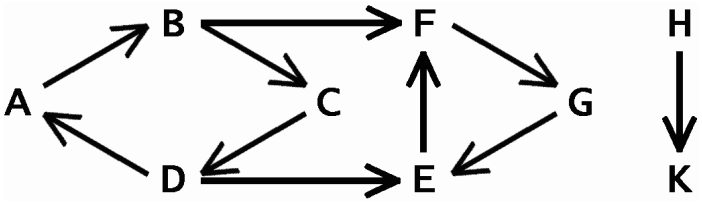
- *Компоненти на силна свързаност.* Търсят се по следния алгоритъм:
  1) Обхождаме графа в дълбочина и подреждаме върховете в ред, обратен на затварянето.
  2) Обхождаме графа втори път (няма значение как) и обръщаме посоките на ребрата.
  3) Обхождаме графа в дълбочина, спазвайки реда на върховете, получен на стъпка № 1. Всяко изпълнение на ред № 10 от псевдокода на алгоритъма обхождане в дълбочина пробягва всички върхове от една компонента на силна свързаност.

  Времевата сложност на този алгоритъм е $\Theta(n+m)$ при всякакви входни данни.

  Стъпка № 1 прилича на топологичното сортиране, само че сега графът се обхожда докрай, дори да съдържа цикъл. Резултатът е масив (или списък) ℓ от всички върхове на графа, подредени по специален начин.

  Между стъпка № 1 и стъпка № 3 има две разлики. Едната е, че стъпка № 3 се изпълнява върху граф с обърнати посоки на ребрата. Другата разлика е, че на входа на стъпка № 1 върховете на графа са подредени във V по произволен начин, а на входа на стъпка № 3 върховете са подредени в ℓ по точно определен начин. Тоест стъпка № 1 се изпълнява върху дадения граф $G\big(V;\,E\big)$, а стъпка № 3 се изпълнява върху друг граф $G'\big(\ell;\,E^{-1}\big)$, където V и ℓ се различават само по реда на върховете. Все едно че кодът на стъпка № 3 обхожда върховете с цикъл от вида **for each** $v \in \ell$ вместо **for each** $v \in V$.

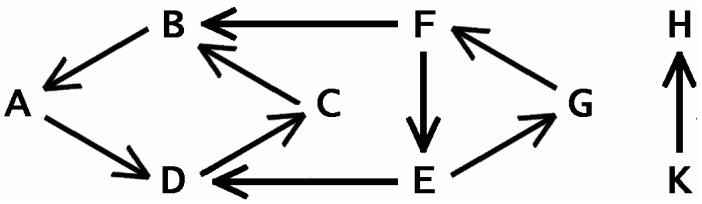
  П р и м е р :



  Този граф има две компоненти на слаба свързаност: $\{\mathrm{A};\,\mathrm{B};\,\mathrm{C};\,\mathrm{D};\,\mathrm{E};\,\mathrm{F};\,\mathrm{G}\}$ и $\{\mathrm{H};\,\mathrm{K}\}$. Но компонентите на силна свързаност са четири: $\{\mathrm{A};\,\mathrm{B};\,\mathrm{C};\,\mathrm{D}\}$, $\{\mathrm{E};\,\mathrm{F};\,\mathrm{G}\}$, $\{\mathrm{H}\}$ и $\{\mathrm{K}\}$. Описаният алгоритъм, използващ обхождане в дълбочина, ще ги намери по следния начин. Нека върховете са дадени отначало в следния ред: $V = \big(\mathrm{F};\,\mathrm{E};\,\mathrm{G};\,\mathrm{D};\,\mathrm{B};\,\mathrm{H};\,\mathrm{C};\,\mathrm{K};\,\mathrm{A}\big)$. Тогава обхождането в дълбочина от стъпка № 1 изглежда така:

$$\big(\ \big(\ \big(\ \big)\ \big)\ \big)\ \big(\ \big(\ \big(\ \big(\ \big)\ \big)\ \big)\ \big)\ \big(\ \big(\ \big)\ \big)$$
F G E E G F D A B C C B A D H K K H

  Подреждаме върховете в ред, обратен на затварянето: $\ell = \big(\mathrm{H};\,\mathrm{K};\,\mathrm{D};\,\mathrm{A};\,\mathrm{B};\,\mathrm{C};\,\mathrm{F};\,\mathrm{G};\,\mathrm{E}\big)$. На стъпка № 2 обръщаме посоките на ребрата:



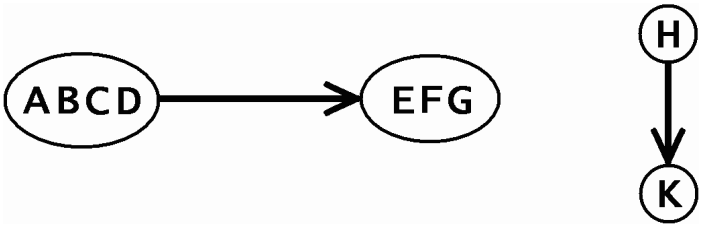
  Получения граф обхождаме в дълбочина по време на стъпка № 3, като спазваме реда от ℓ:

$$\underbrace{\big(\ \big)\ \ \big(\ \big)}_{\text{№ 1}\quad\text{№ 2}}\ \ \underbrace{\big(\ \big(\ \big(\ \big(\ \big)\ \big)\ \big)\ \big)}_{\text{№ 3}}\ \ \underbrace{\big(\ \big(\ \big(\ \big)\ \big)\ \big)}_{\text{№ 4}}$$
H H K K D C B A A B C D F E G G E F

  Компонентите на силна свързаност съответстват на скоби от най-високо ниво (тоест такива, които не са вложени в други скоби). Иначе казано, с който връх започва някоя компонента, със същия връх трябва и да завърши.

За всеки ориентиран граф G (със или без цикъл) построяваме друг ориентиран граф $\overset{*}{G}$, чиито върхове са компонентите на силна свързаност на G. Ребрата на $\overset{*}{G}$ представляват копия на онези ребра на G, които свързват върхове от различни компоненти. Обикновено се получават кратни ребра, които можем да слеем, та $\overset{*}{G}$ да бъде граф, а не мултиграф. Можем също да запазим кратните ребра: отстраняването им само би забавило алгоритъма, а повечето алгоритми работят и върху мултиграфи (например топологичното сортиране). Ребрата на G, които свързват върхове от една и съща компонента, не се пренасят в $\overset{*}{G}$, за да не се получават примки.

Ако G е ацикличен граф, всеки негов връх е отделна компонента на силна свързаност, следователно $\overset{*}{G}$ е изоморфен на G. Графът $\overset{*}{G}$ е ацикличен, дори ако G съдържа цикъл: иначе компонентите, образуващи цикъл, биха били силно свързани, тоест биха се слели в една компонента.



Описаното построение е полезно в задачи, за които съществува ефективен алгоритъм в случая на ориентиран ацикличен граф, но не и в общия случай. При такъв тип задачи графът от компонентите на силна свързаност свежда общия случай до частния.

- *Срязващи върхове и ребра.* Връх или ребро на неориентиран граф се наричат срязващи, ако премахването им увеличава броя на компонентите на свързаност. Срязващите ребра се наричат още мостове.

  Срязващите върхове и ребра са уязвими места в графа. Например в съобщителна мрежа срязващите върхове и ребра съответстват на възли и връзки, чието повреждане отнема възможността за изпращане на съобщения поне между някои възли, които по-рано са имали такава възможност.

  Два върха на неориентиран граф се наричат върхово двусвързани, ако между тях има поне два пътя без общи върхове освен краищата. Два върха на неориентиран граф се наричат реброво двусвързани, ако между тях има поне два пътя без общи ребра. Очевидно е, че ако два върха са двусвързани върхово, то те са двусвързани и реброво. Обратното не е вярно. В определенията се предполага, че графът съдържа поне три върха. (Ако се разрешават кратни ребра, то в определението за реброва двусвързаност се допуска мултиграфът да съдържа само два върха).

  Ясно е, че ако един граф е двусвързан, то той е свързан. Обратното не е вярно.

  Двусвързана компонента на граф се нарича максимален породен подграф, чиито върхове са два по два двусвързани (в единия или другия смисъл). Един граф се нарича двусвързан, ако целият е една двусвързана компонента.

  Нека G е свързан неориентиран граф с поне три върха. В сила са следните твърдения:
  1) G е върхово двусвързан $\iff$ G няма срязващ връх.
  2) G е реброво двусвързан $\iff$ G няма срязващо ребро.

  Реброво двусвързаните компоненти не притежават нито общи върхове, нито общи ребра. Върхово двусвързаните компоненти могат да имат най-много един общ връх и той винаги е срязващ. Може да има върхове, които не принадлежат на никоя двусвързана компонента (например върхове от първа степен).

Алгоритъм на Шмит за търсене на срязващите върхове и ребра
на свързан неориентиран граф с поне три върха:

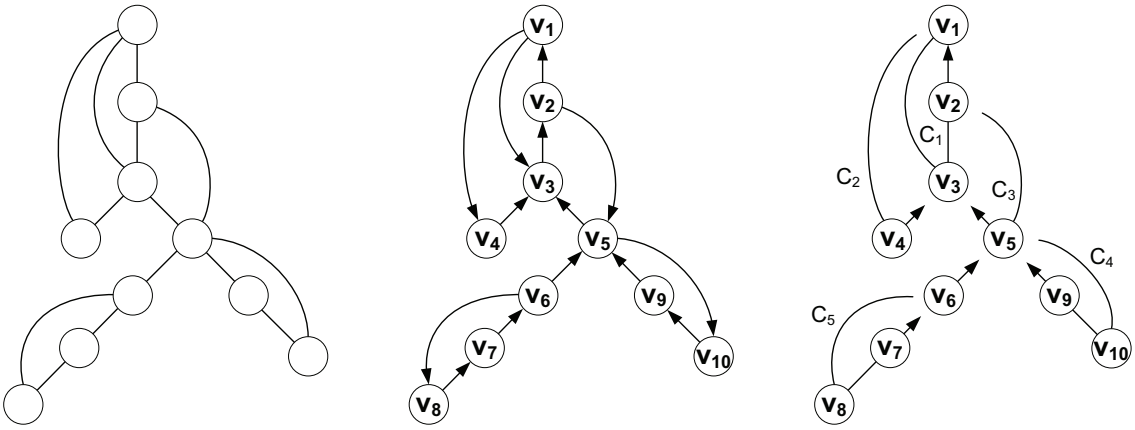
1) Обхождаме графа в дълбочина, класифицираме всяко ребро като дървесно или обратно и номерираме върховете в реда на отваряне.
2) Ориентираме дървесните ребра към корена, а обратните — към листата. Иначе казано, насочваме дървесните ребра към малките номера, а обратните ребра — към големите.
3) Създаваме празен списък, в който ще пазим т. нар. вериги.
4) Обявяваме всички върхове и дървесни ребра за непосетени.
5) Обхождаме върховете по нарастващ ред на номерата и за всеки връх u правим следното:
   а) Отбелязваме върха u като посетен.
   б) Обхождаме в произволен ред обратните ребра, излизащи от u, и за всяко от тях:
      — Преминаваме към другия край v на обратното ребро.
      — От v се движим по дървесните ребра към корена, докато намерим посетен връх (най-ранният възможен такъв връх е v, а най-късният — u). При това движение отбелязваме преминатите върхове и дървесни ребра като посетени, а изминатия път, наричан верига в този конкретен алгоритъм, добавяме към списъка на веригите. Едни вериги са цикли, а други — не. Текущата верига е цикъл тогава и само тогава, когато последното ѝ ребро сочи към върха u, от който е започнала.

   Така графът се разлага на вериги, номерирани в реда на образуване: C_1, C_2 и т.н. Може да няма вериги (ако графът е дърво). Ако има поне една верига, то C_1 е цикъл.
6) Обхождаме дървесните ребра и гледаме дали са посетени. Всяко непосетено ребро е мост. Всеки от краищата на мост, ако е поне от втора степен, е срязващ връх.
7) Обхождаме списъка от вериги без първата: C_2, C_3, C_4 и т.н. За всяка верига (без C_1) проверяваме дали е цикъл. Ако да — началото ѝ е срязващ връх.

Забележка: Не е изключено началото на цикъла C_1 да бъде срязващ връх. Това става, когато то е начало и на друг цикъл от списъка с вериги, а също и когато е край на мост.

Времевата сложност на този алгоритъм е $\Theta(n+m)$ при всякакви входни данни.

П р и м е р :



На чертежа вляво е показан свързан неориентиран граф след обхождането в дълбочина. Дървесните ребра са изобразени с прави линии, а обратните ребра — със заоблени линии.

На чертежа в средата са показани номерацията на върховете и ориентацията на ребрата. Това е резултатът от първите две стъпки от алгоритъма на Шмит.

Чертежът вдясно показва разлагането на графа в съвкупност от вериги и номерацията им. Дървесното ребро $v_6\,v_5$ не участва в никоя верига, т.е. остава непосетено на петата стъпка, следователно то е мост. Краищата му са от степени 3 и 5, затова са срязващи върхове. Веригите C_1, C_4 и C_5 са цикли, а C_2 и C_3 не са. Начала на циклите C_4 и C_5 са съответно върховете v_5 и v_6. Така че те са срязващи върхове и на това основание.

АЛГОРИТМИ ВЪРХУ ТЕГЛОВНИ ГРАФИ

В този раздел ще разгледаме някои от най-използваните алгоритми върху тегловни графи. Теглата са реални числа — по едно за всяко ребро на графа. Могат да имат различни значения: дължина на шосе, цена за преминаване, надеждност на съобщителна връзка и т.н.

Динамично програмиране при ориентирани ациклични графи

Това е метод за решаване на оптимизационни и комбинаторни задачи. При първия тип често се иска т. нар. възстановяване на решението — намиране на самото оптимално решение, а не просто стойността му. По принцип възстановяването става в обратен ред на пресмятането на оптималната стойност. Разглеждаме следните оптимизационни задачи:

- *Най-къси пътища от всички върхове до даден връх t.* Извършваме топологично сортиране, след което обработваме върховете отзад напред (тоест обратно на посоките на ребрата). На всеки връх u приписваме по едно число $\ell(u)$ — дължината на най-къс път от u до t. Ако искаме да намерим не само дължината на най-къс път, но и самия път, то за всеки връх пазим още първата стъпка от най-късия път — реброто от пътя, излизащо от този връх.

  Принцип на Белман: Всеки участък от най-къс път е също най-къс път.

  Принципът е очевиден: ако можехме да скъсим някой участък, това би скъсило целия път в противоречие с факта, че е най-къс. На практика принципът на Белман се прилага така: за да намерим най-къс път от u до t, първо намираме най-късите пътища от всички v до t, където v пробягва преките наследници на u. Тъкмо затова обработваме върховете на графа отзад напред — срещу посоките на ребрата. Числото на всеки връх пресмятаме така:

$$\ell(u) = \begin{cases} +\infty, & \text{ако } u \text{ е зад } t; \\ 0, & \text{ако } u = t; \\ \min\limits_{v \in \mathrm{Adj}(u)} \Big\{ w(u\,;\,v) + \ell(v) \Big\}, & \text{ако } u \text{ е пред } t; \end{cases}$$

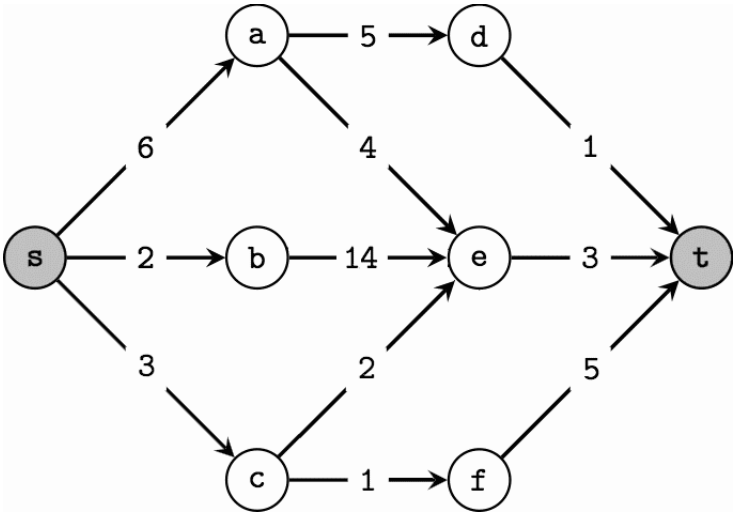
  където $w(u\,;\,v)$ е теглото на реброто от u към v. Тук теглото се тълкува като дължина. Въпреки това се допуска теглото да бъде не само положително, но и нулево, и отрицателно. Дължината на път е сборът от дължините на неговите ребра. Последният ред от формулата изразява принципа на Белман, а другите два реда са инициализацията на алгоритъма. Върха t инициализираме с 0, защото най-късият път от t до t се състои само от върха t и не съдържа ребра. Върховете зад t получават условна стойност $+\infty$, понеже от тях няма път до t, а стойността $+\infty$ ги изключва от кандидатите за минимум. Нужно е също за всеки връх u да пазим реброто, излизащо от u към онзи връх v, за който се достига най-малката стойност във формулата; по тази информация ще намерим самия най-къс път, а не просто дължината му.

- *Най-къси пътища от даден връх s до всички върхове.* Извършваме топологично сортиране, след това правим пресмятания, подобни на горните. Разлики: движим се в права посока (спазваме посоките на ребрата); минимумът се взима по входящите ребра, т.е. $u \in \mathrm{Adj}(v)$; $\ell(u)$ е дължината на най-къс път от s до u; заменяме "пред t" със "зад s" и "зад t" с "пред s"; навсякъде другаде заменяме t с s.

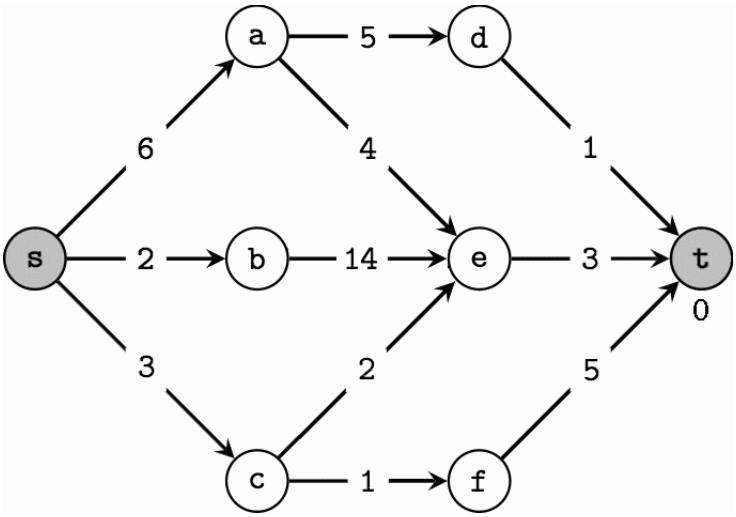
- *Най-къс път от даден връх s до даден връх t.* Прилагаме някоя от горните две процедури.

- *Най-дълъг път.* Тази задача съществува в три разновидности, подобни на горните три. Решава се по аналогичен начин, само заменяме минимумите с максимуми и $+\infty$ с $-\infty$.

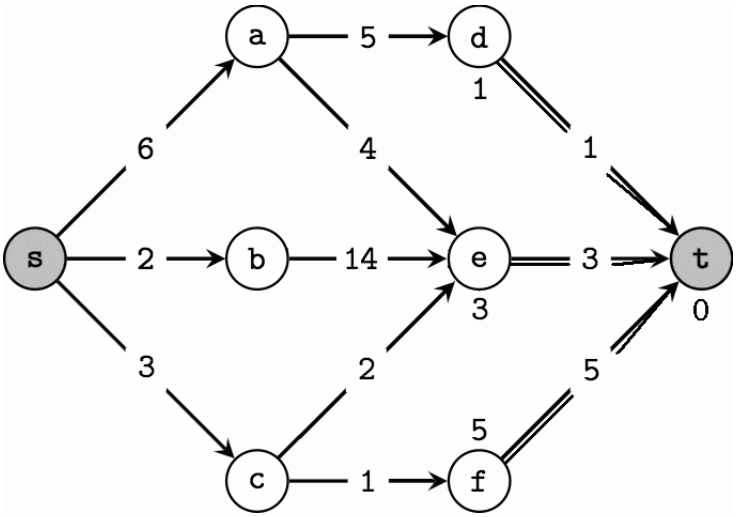
Пример: Търсим най-къс път от s до t в следния граф:



Избираме посока на извършване на сметките, например от t към s. Инициализираме t с 0:

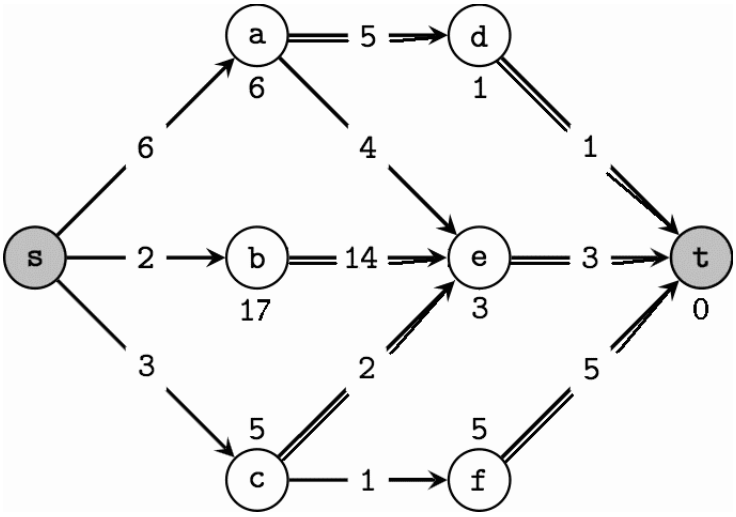


Стойностите на върховете d, e и f се пресмятат тривиално:



Чрез удвояване са показани "избраните" ребра, излизащи от d, e и f. Точно за тези три върха изборът е тривиален: всеки от тях притежава само едно изходящо ребро.

По-интересни са пресмятанията при следващите върхове — a, b и c:

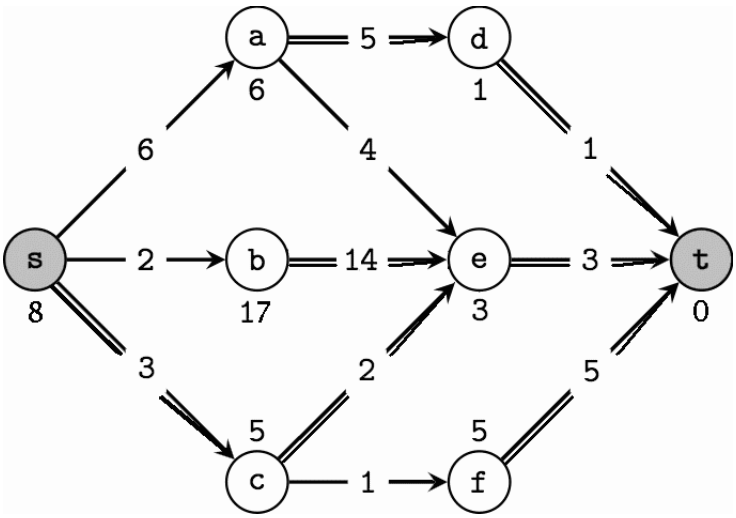


Стойността на върха a например е пресметната така:

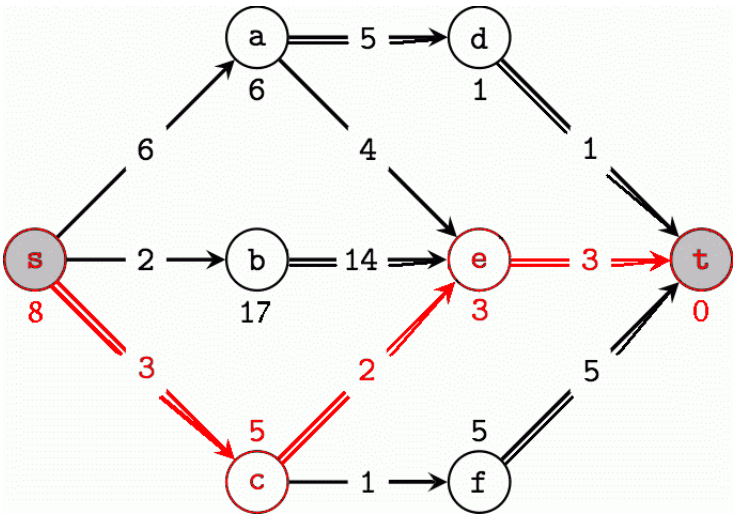
$$w(a\,;\,d) + \ell(d) = 5 + 1 = 6, \quad w(a\,;\,e) + \ell(e) = 4 + 3 = 7, \quad 6 < 7 \quad \Rightarrow \quad \ell(a) = 6.$$

Реброто от a към d е удвоено, защото най-късият път от a до t минава през d, а не през e.

На последната стъпка намираме дължината на най-късия път от s до t:



Най-късия път намираме в обратна посока на пресмятанията — от s до t по избраните ребра:



Окончателно, най-късият път от s до t има дължина 8 и това е пътят s — c — e — t.

Подробен анализ на сложността ще направим в края на раздела, общо за всички алгоритми. Тук ще отбележим само, че времевата сложност на разгледания алгоритъм е линейна, защото той обхожда графа два пъти. Показахме второто обхождане — намирането на най-късите пътища. Не беше показано първото обхождане — топологичното сортиране. На чертежите се вижда само крайният му резултат — подреждането на върховете (отляво надясно): първи е върхът s, след него идват върховете a, b и c (в някакъв ред), после d, e и f (също в произволен ред) и последен е върхът t.

Пълното изчерпване, т.е. изследването на всички възможни пътища от s до t, е коректно, но твърде бавно: броят им като общо расте експоненциално с нарастване на графа.

От друга страна, съществува много бърз алгоритъм от класа на т. нар. алчни алгоритми: на всяка стъпка избираме решението, което е най-добро за тази стъпка. Тръгваме от върха s, сравняваме теглата на излизащите ребра $(2 < 3;\ 2 < 6)$ и избираме най-лекото ребро — sb. По-нататък нямаме избор: от b отиваме в e, а от e — в t. От този пример добре проличава силната страна на алчния алгоритъм: той е изключително бърз, дори не обхожда целия граф! Обаче този алгоритъм е некоректен: предлага пътя $s — b — e — t$ с дължина $2 + 14 + 3 = 19$, която не е най-малката възможна (динамичното програмиране намери път с дължина 8).

Динамичното програмиране е едновременно коректно и бързо. Коректността на метода следва от принципа на Белман, а бързината — от това, че всички върхове и ребра се посещават само два пъти: при топологичното сортиране и при пресмятането на дължините.

Произходът на названието "динамично програмиране" няма нищо общо със съставянето на компютърни програми. Думата "програма" е употребена в най-общ смисъл на някакъв план, например производствена програма. А прилагателното "динамично" уточнява, че се разглеждат процеси, протичащи на няколко етапа.

Разгледаният граф може да бъде изтълкуван като модел на следната практическа задача. Искаме да купим от чужбина стока, която се произвежда в три различни държави — a, b и c. В държавата a стоката се предлага за 6000 долара, в b струва 2000 долара, а в c — 3000 долара (това са теглата на ребрата, излизащи от върха s). Избираме между трима превозвачи: d, e и f. Фирмата d може да превози стоката само от държавата a (тъй като няма клонове в b и c) и това ще струва 5000 долара. Фирмата f ще превози стоката за 1000 долара, обаче само от c. Превозвачът e има клонове навсякъде, но за различни разстояния предлага различни цени: 4000, 14000 и 2000 долара за превоз от a, b и c съответно. За разни допълнителни услуги (съхраняване на стоката при специални условия, например при точно определена температура) превозвачите начисляват допълнителни такси, които не зависят от пропътуваното разстояние: d — 1000 долара; e — 3000 долара; f — 5000 долара. Как да се снабдим със стоката най-евтино?

При това тълкуване имаме процес, който протича в дискретно време — четири момента, съответстващи на четирите стълба, в които са подредени върховете на графа. Той е ацикличен, защото всички ребра сочат надясно: времето тече в една посока — от миналото към бъдещето. В началния момент не сме извършили все още никакво действие; намираме се във върха s. Във втория момент сме купили стоката, но не сме я превозили; това са върховете a, b и c. Третият момент е, когато сме избрали превозвач; това са върховете d, e и f от третия стълб. Четвъртият момент съответства на върха t — стоката е пристигнала, процесът приключва. Между четирите момента има три прехода, което значи, че процесът протича на три етапа: избор на производител и купуване на стоката; избор на превозвач и заплащане на превоза; заплащане на допълнителните услуги, свързани с превоза.

В теорията се разглеждат и процеси с непрекъснато време. При тях етапите са безброй много, тоест възникват безкрайни графи, каквито не разглеждаме тук. Графите не са подходящ модел за такива задачи. По-добре е да се използва т. нар. управляваща функция, която въплъщава стратегията за избор на действие на всяка стъпка в зависимост от момента и от състоянието, достигнато от процеса. (Казано в термините на графи, управляващата функция съпоставя по едно изходящо ребро на всеки връх на графа.) Стъпките са безкрайно малки, което води до получаването на диференциални уравнения — тема извън обхвата на настоящия раздел.

Дотук разгледахме само оптимизационни задачи върху ориентирани ациклични графи. Но за такива графи могат да се решават и комбинаторни задачи:

- *Брой пътища от всички върхове до даден връх t.* Извършваме топологично сортиране, след което обработваме върховете отзад напред (тоест обратно на посоките на ребрата). На всеки връх u съпоставяме по едно цяло число $\ell(u)$ — броя на пътищата от u до t. Тези числа се пресмятат така:

$$\ell(u) = \begin{cases} 1, & \text{ако } u = t; \\ \displaystyle\sum_{v \in \mathrm{Adj}(u)} \ell(v), & \text{ако } u \neq t. \end{cases}$$

  Празните сборове се смятат за нули.

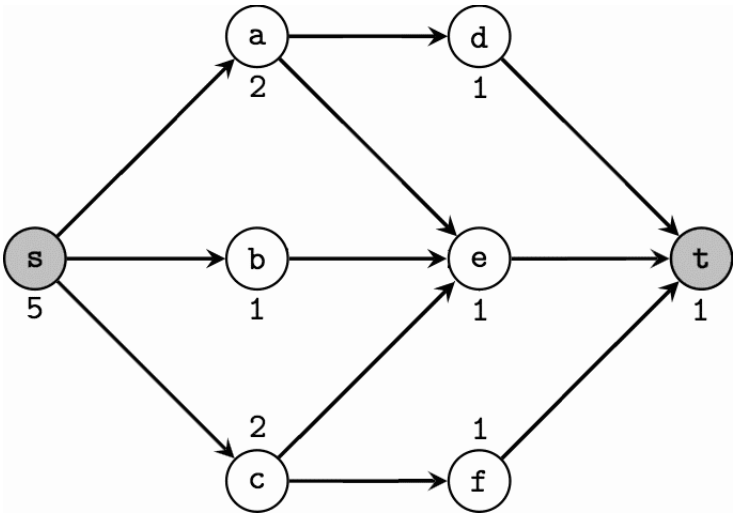
- *Брой пътища от даден връх s до всички върхове.* Извършваме топологично сортиране, след това правим пресмятания, подобни на горните. Разлики: движим се в права посока (спазваме посоките на ребрата); сборът се пресмята по входящите ребра, т.е. $u \in \mathrm{Adj}(v)$; $\ell(u)$ е броят на пътищата от s до u; навсякъде заменяме t с s.

- *Брой пътища от даден връх s до даден връх t.* Прилагаме някоя от горните две процедури.

Изброените комбинаторни задачи се отнасят за нетегловни ориентирани ациклични графи, обаче ги разглеждаме в този раздел, защото динамичното програмиране се прилага към тях по същия начин, както към оптимизационните задачи.

При комбинаторните задачи не може да се говори за възстановяване на решение.

П р и м е р : Броя на пътищата от s до t (общо пет) пресмятаме отзад напред:



Първата фаза от динамичното програмиране (тоест топологичното сортиране) изразходва време $\Theta(n+m)$ и памет $\Theta(n)$ при всякакви входни данни заради обхождането в дълбочина. Втората фаза (пресмятането на стойностите на върховете) също притежава линейна сложност: изразходва време $\mathrm{O}(n+m)$ и памет $\mathrm{O}(n)$ при всякакви входни данни: като обхождаме графа, преминаваме най-много веднъж през всеки връх и всяко ребро. Окончателно, целият алгоритъм притежава времева сложност $\Theta(n+m)$ и сложност по памет $\Theta(n)$ при всякакви входни данни.

Тези изводи търпят промяна в някои случаи. Графът може да бъде даден в сортиран вид. Тогава остава само втората фаза от алгоритъма; времевата сложност е пак $\Theta(n+m)$ по порядък, обаче константният множител пред порядъка намалява. Ако при това са дадени два върха s и t, можем да прекратим обхождането при достигане на желания връх, така че времевата сложност е $\Theta(n+m)$ само при най-лоши входни данни — когато желаният връх бъде обходен последен. Ако графът е даден в сортиран вид и притежава хубава структура, можем да спестим и памет. Да кажем, в последния пример не пазим по едно ребро за всеки връх, а и числата на върховете не са нужни постоянно: намерим ли $\ell(a)$, $\ell(b)$ и $\ell(c)$, то $\ell(d)$, $\ell(e)$ и $\ell(f)$ не ни трябват повече.

Минимално покриващо дърво

Всеки свързан неориентиран граф притежава поне едно покриващо (обхващащо) дърво — такова, което съдържа всички върхове на графа и всяко ребро на дървото е ребро и на графа. Едно покриващо дърво е например дървото на обхождането на графа в ширина или в дълбочина. Графът може да има повече от едно покриващо дърво.

Да предположим допълнително, че графът е тегловен, тоест на всяко от неговите ребра е приписано тегло — реално число (положително, отрицателно или нула). Тегло на едно дърво наричаме сбора от теглата на ребрата му. Минимално покриващо (обхващащо) дърво наричаме онова покриващо дърво на графа, чието тегло е възможно най-малко. Тъй като множеството от покриващите дървета е крайно и непразно, то непременно има минимално покриващо дърво, а може да съществуват и няколко такива (с еднакво тегло).

Намирането на минимално покриващо дърво на даден свързан неориентиран тегловен граф представлява значителен практически интерес. Ако върховете на графа са например селища, ребрата — шосета на непостроена пътна мрежа, а теглата — цени за построяване на шосетата, то минималното покриващо дърво е най-евтината пътна мрежа, свързваща всички селища.

За решаването на тази задача са съставени бързи алгоритми. Ще разгледаме два от тях. Навсякъде n е броят на върховете, а m е броят на ребрата на графа.

Алгоритъм на Крускал:

1) Сортираме по тегло множеството E от ребрата на графа.
2) $A \leftarrow \varnothing$, където A е множеството от ребрата на търсеното дърво.
3) Обхождаме сортирания масив E от най-лекото към най-тежкото ребро.
4) Поредното ребро от масива E се добавя към A, ако не затваря цикъл в A.
5) Алгоритъмът връща полученото множество A.

Тъй като всяко дърво с n върха има точно $n-1$ ребра, то можем да прекратим алгоритъма при добавяне на $(n-1)$-ото ребро към A: за всички следващи ребра и без проверка е сигурно, че ще затварят цикли в A.

Стъпки № 3 и № 4 образуват цикъл по брояч — индекса на поредното ребро от масива E. Броячът е полуинвариант: расте с единица при всяко изпълнение на стъпка № 4. Следователно тя се изпълнява не повече от m пъти, след което цикълът завършва.

Инвариант: Във всеки миг множеството A е подмножество на множеството от ребрата на някое минимално покриващо дърво.

Доказателство: с помощта на математическа индукция по поредния номер на проверката за край на цикъла. За простота ще бележим по един и същи начин кое да е дърво и множеството от ребрата му.

База: Първата проверка за край се изпълнява при влизане в цикъла. От стъпка № 2 следва, че в този миг е изпълнено равенството $A = \varnothing$. По-горе доказахме, че съществува поне едно минимално покриващо дърво T. В сила е включването $A \subseteq T$, тъй като празното множество е подмножество на всяко множество.

Индуктивна стъпка: Нека при някоя проверка за край, която не е последна, съществува минимално покриващо дърво T, такова че $A \subseteq T$. Ще докажем, че при следващата проверка ще се намери минимално покриващо дърво T_2, такова че $A' \subseteq T_2$, където A' е новото множество от избрани ребра.

Ако при изпълнението на стъпка № 4 поредното ребро не е било добавено към A, то $A' = A$ и можем да изберем $T_2 = T$.

Сега нека по време на стъпка № 4 реброто e е било добавено към A, тоест $A' = A \cup \{e\}$. Ако $A' \subseteq T$, инвариантът е изпълнен и при новата проверка. Затова нека $A' \nsubseteq T$. Да приложим индуктивното предположение: $A \subseteq T$. Значи, $e \notin T$. Щом T е покриващо дърво, добавянето на e затваря цикъл C в T. От друга страна, важи включването $T \cup \{e\} \supseteq A \cup \{e\} = A'$ и тъй като A' не съдържа цикъл (защото иначе стъпка № 4 не би добавила реброто e към множеството A), то $C \nsubseteq A'$, въпреки че $C \subseteq T$.

И така, цикълът C съдържа поне едно ребро, което не принадлежи на A'. Такива ребра има краен брой; да означим с f най-тежкото от тях. Ако има няколко най-тежки ребра в $C \setminus A'$, няма значение кое от тях ще бъде f. От $e \in A'$ и $f \notin A'$ следва, че $f \neq e$.

Нека $T_2 = T \cup \{e\} \setminus \{f\}$. Покриващото дърво T е свързан граф с n върха и $n-1$ ребра. Понеже $e \notin T$, то $T \cup \{e\}$ е свързан граф с n върха и n ребра. Премахването на ребро от цикъл не нарушава свързаността, затова T_2 е свързан неориентиран граф с n върха и $n-1$ ребра, тоест покриващо дърво.

От $T \cup \{e\} \supseteq A'$ и $f \notin A'$ следва, че $T_2 \supseteq A'$. Ако докажем, че T_2 има минимално тегло, тогава ще следва, че T_2 е търсеното минимално покриващо дърво.

Нека w означава теглото на ребро или дърво. Очевидно $w(T_2) = w(T) + w(e) - w(f)$. Минималността на T_2 се изразява с равенството $w(T_2) = w(T)$, което пък е равносилно на $w(e) = w(f)$.

Допускането, че $w(f) > w(e)$, влече неравенството $w(T_2) < w(T)$, което противоречи на минималността на T.

Да допуснем, че $w(f) < w(e)$. Следователно реброто f е било обходено от цикъла преди e. Щом $f \notin A'$, то $f \notin A$, тоест алгоритъмът е отхвърлил реброто f, значи f затваря цикъл в A. С други думи, множеството от ребра $A \cup \{f\}$ съдържа цикъл. Но $A \subseteq T$ и $f \in T$, следователно $A \cup \{f\} \subseteq T$. Това означава, че T съдържа цикъл, което противоречи на определението за дърво.

Получените противоречия показват, че за ребрата e и f остава единствена възможност: $w(e) = w(f)$, което трябваше да се докаже.

Вече установихме, че цикълът завършва. При последната проверка за край на цикъла всички ребра на дадения граф $G(V; E)$ са обходени. Да разгледаме неговия подграф $G'(V; A)$ и да допуснем, че G' в този миг е несвързан граф, т.е. има поне две компоненти на свързаност. Между тях съществува поне едно ребро e във входния граф G, защото той е свързан. Но $e \notin A$, щом краищата на e са в различни компоненти на G'. Добавянето на e към A би намалило броя на компонентите на свързаност на G', но не би затворило цикъл в A, тоест в G'. Тогава защо алгоритъмът не е добавил e към A? Възможни са само две обяснения, но никое от тях не върши работа:

— Реброто e не е добавено към множеството A, защото цикълът още не е стигнал до e. Това обяснение не е правилно: щом алгоритъмът се намира в края на цикъла, значи е обходил всички ребра на G.

— Реброто e е било отхвърлено, защото добавянето му към A би затворило цикъл в A. Това също не е вярно, защото реброто e не затваря цикъл дори в окончателното множество A, камо ли в онова A, което е било актуално към момента, в който алгоритъмът е разглеждал e (онова A е било подмножество на окончателното A).

И в двата случая се стига до противоречие, а други случаи няма. Следователно не е вярно допускането, че в края на цикъла множеството от ребра A образува несвързан граф $G'(V; A)$. Тоест G' е свързан граф с n върха. Затова $|A| \geq n-1$.

Сега нека приложим доказания инвариант към последната проверка за край на цикъла. Съществува минимално покриващо дърво T, такова че $A \subseteq T$, като в това включване T означава множеството от ребрата на дървото съгласно с направената уговорка. Понеже T покрива G, то T има n върха, а щом T е дърво, то T съдържа точно $n-1$ ребра. От включването $A \subseteq T$ следва неравенството $|A| \leq |T| = n-1$. По-горе доказахме, че $|A| \geq n-1$, затова $|A| = n-1$. Следователно A съвпада с множеството от ребрата на T, тоест графът G' съвпада с дървото T и алгоритъмът, връщайки A, връща ребрата на T, което е минимално покриващо дърво на G. Ето защо алгоритъмът на Крускал коректно построява минимално покриващо дърво.

С подобни разсъждения можем да докажем две важни твърдения:
— Ако между теглата на ребрата няма равни, то минималното покриващо дърво е единствено.
— Ако между теглата на ребрата има равни, тогава е възможно графът да притежава няколко минимални покриващи дървета. Всяко от тях може да бъде получено по алгоритъма на Крускал, стига да се зададат приоритети на ребрата с равни тегла (тоест да се укаже на алгоритъма в какъв ред да ги обхожда).

Анализ на алгоритъма на Крускал: Входният граф е зададен със списъци на съседство, затова дължината на входа е $\Theta(n+m)$. Графът е свързан, следователно изразите $\log m$ и $\log n$ са равни по порядък и $m \geq n-1$, поради което $m = \Omega(n)$ и дължината на входа е $n+m = \Theta(m)$. Ако използваме пирамидално сортиране за ребрата, ще трябва време $\Theta(m \log m) = \Theta(m \log n)$ и памет $\Theta(1)$. За тази цел е нужно да представим множеството на ребрата като масив $E\big[1 \ldots m\big]$, но това не е проблем: става с едно обхождане на графа за време $\Theta(n+m) = \Theta(m) = \mathrm{o}(m \log m)$, тоест времето за конвертиране е незначително спрямо времето за сортиране. Масивът $E\big[1 \ldots m\big]$ се явява локална променлива, заемаща памет $\Theta(m)$. Тъй като след конвертирането се използва само този масив, но не и входните данни, възниква въпросът дали можем да спестим памет, преобразувайки входните данни на място и бързо. Очевидно не, ако ребрата от входните списъци са разпръснати из паметта. Ако пък заемат непрекъснат отрязък от паметта, има значение как точно са записани в този отрязък. Възможността за спестяване на памет зависи чувствително от физическото представяне на входните данни. За да не навлизаме в технически подробности, ще приемем оценката $\Theta(m)$.

Анализът на втората фаза (изпробването на ребрата) зависи от структурите от данни.

Наивната реализация не е достатъчно бърза. Дали поредното ребро от E затваря цикъл в A, можем да проверим чрез стандартния алгоритъм за търсене на цикъл (с обхождане в дълбочина). Отделната проверка има линейна времева сложност; всичките m проверки общо — квадратична.

По-ефективна структура от данни ще получим, ако забележим следните подробности. Неориентираният граф $G'\big(V;\, A\big)$ не съдържа цикъл, тоест той е гора. Отначало G' няма ребра, всеки връх е отделно дърво и гората съдържа n дървета. С добавянето на все повече ребра дърветата се сливат и броят им намалява. Едно ребро се добавя само ако не затваря цикъл, тоест тогава и само тогава, когато краищата му са от различни дървета. Всяко добавено ребро намалява с единица броя на дърветата; след добавяне на $(n-1)$-ото ребро остава едно дърво — минималното покриващо дърво на входния граф.

Алгоритъм на Крускал $\big(G\big(V$: върхове; E: ребра$\big)$: неориентиран свързан тегловен граф$\big)$

```
 1)  Сортиране (E) по тегла   // Например пирамидално сортиране.
 2)  for each v ∈ V do
 3)      MakeSet(v)
 4)  n ← |V|   // n > 0, защото V ≠ ∅
 5)  A ← ∅
 6)  k ← 1
 7)  for each e = {u; v} ∈ E do
 8)      ũ ← Find(u)
 9)      ṽ ← Find(v)
10)      if ũ ≠ ṽ
11)          Union(ũ; ṽ)
12)          A ← A ∪ {e}
13)          k ← k + 1
14)          if k = n
15)              return A   // Списък от ребрата на минимално покриващо дърво на G.
16)  return NIL   // Грешни входни данни: несвързан граф.
```

Забележка: Последният ред може да се замени с **return** A; при несвързан граф ще върне гора от минимални покриващи дървета — по едно за всяка компонента на свързаност на графа.

За ефективна реализация на алгоритъма са нужни две бързи операции: сливане на дърветa и проверка дали краищата на дадено ребро са от едно и също дърво, или от различни дървета. Казаното представлява описание на абстрактен тип данни за работа с непресичащи се множества, наречен Union-Find по имената на двете процедури (съществува и трета процедура — MakeSet; тя служи за инициализиране). Има поне две бързи реализации на този абстрактен тип данни, които се използват не само за дървета, а за произволна съвкупност от непресичащи се множества.

Първата реализация използва списъци от елементи (в случая това са върхове на графа). Всяко от непресичащите се множества (т.е. всяко дърво) се представя като едносвързан списък. За всеки списък пазим броя на елементите и два указателя: към началото и към края на списъка. Обратно, за всеки елемент пазим някакъв идентификатор на списъка, който съдържа елемента, например указател към първия елемент на списъка. Отначало списъците имат по един елемент. Проверката дали два елемента са от един списък, се извършва за константно време с помощта на списъчните идентификатори, пазени от елементите. Най-бавно е сливането на два списъка: по-късият списък се залепя за по-дългия, актуализират се всички списъчни идентификатори, пазени от елементите на късия списък; актуализира се и броят на елементите на дългия списък.

Трите процедури за работа с тази структура от данни са достатъчно елементарни, затова ще ги опишем не отделно, а направо в кода на алгоритъма на Крускал:

Алгоритъм на Крускал $\big(G$: неориентиран свързан тегловен граф с n върха и m ребра$\big)$

1) // Както обикновено, ребрата са представени чрез списъци на съседство и $n > 0$.
2) // За простота приемаме, че върховете на графа са целите числа от 1 до n вкл.
3) $E\big[1 \ldots m\big]$: масив от ребрата на графа.
4) $E\big[1 \ldots m\big] \leftarrow$ Конвертиране $\big(G\big)$
5) Сортиране $\big(E\big)$ по тегла // Например пирамидално сортиране.
6) $F\big[1 \ldots n\big]$: гора — масив от дървета (едносвързани списъци от върхове).
7) $\ell\,\big[1 \ldots n\big]$: масив от цели числа — идентификаторите на дърветата.
8) $s\,\big[1 \ldots n\big]$: масив от цели числа — бройките на върховете на дърветата.
9) **for** $v \leftarrow 1$ **to** n **do**
10) $\quad F\big[v\big] \leftarrow \big\{v\big\}$
11) $\quad \ell\,\big[v\big] \leftarrow v$
12) $\quad s\,\big[v\big] \leftarrow 1$
13) $A \leftarrow \varnothing$
14) $k \leftarrow 1$
15) **for** $i \leftarrow 1$ **to** m **do**
16) $\quad e \leftarrow E\big[i\big]$
17) $\quad \big\{u\,;\,v\big\} \leftarrow e\,.\,\text{endpoints}$
18) $\quad \widetilde{u} \leftarrow \ell\,\big[u\big]$
19) $\quad \widetilde{v} \leftarrow \ell\,\big[v\big]$
20) $\quad$ **if** $\widetilde{u} \neq \widetilde{v}$
21) $\quad\quad$ **if** $s\,\big[\widetilde{u}\big] < s\,\big[\widetilde{v}\big]$
22) $\quad\quad\quad$ swap $\big(\widetilde{u}\,;\,\widetilde{v}\big)$ // Размяна на стойностите на две променливи.
23) $\quad\quad F\big[\widetilde{u}\big] \leftarrow F\big[\widetilde{u}\big] \cup F\big[\widetilde{v}\big]$ // Слепване на списъци.
24) $\quad\quad s\,\big[\widetilde{u}\big] \leftarrow s\,\big[\widetilde{u}\big] + s\,\big[\widetilde{v}\big]$
25) $\quad\quad$ **for each** $x \in F\big[\widetilde{v}\big]$ **do**
26) $\quad\quad\quad \ell\,\big[x\big] \leftarrow \widetilde{u}$
27) $\quad\quad A \leftarrow A \cup \big\{e\big\}$
28) $\quad\quad k \leftarrow k + 1$
29) $\quad\quad$ **if** $k = n$
30) $\quad\quad\quad$ **return** A // Списък от ребрата на минимално покриващо дърво на G.
31) **return** NIL // Грешни входни данни: несвързан граф.

Тази реализация на алгоритъма на Крускал изразходва памет $\Theta(m)$ за масива $E\big[1 \ldots m\big]$ и $\Theta(n)$ за другите три масива и за списъците (те имат общо n елемента — върховете на графа). Понеже графът е свързан, то $m = \Omega(n)$, следователно общото количество допълнителна памет е равно на $\Theta(m) + \Theta(n) = \Theta(m)$ при всякакви входни данни.

Първата фаза от алгоритъма (редовете до № 5 включително) изисква време $\Theta(m \log m)$ заради сортирането на масива E (конвертирането е бързо и времето му може да се пренебрегне). Понеже $\log m$ и $\log n$ са равни по порядък, когато графът е свързан, то времето на първата фаза може да се запише и като $\Theta(m \log n)$. То се отнася за най-лоши входни данни.

Времето на втората фаза се определя от циклите (другите оператори изискват време $\Theta(1)$). Цикълът с начало на ред № 9 изразходва време $\Theta(n)$, а този с начало на ред № 15 — време $\Theta(m)$, тъй като всички оператори в тялото му се изпълняват точно m пъти с изключение на ред № 24, който се намира във вложен цикъл. Командата на ред № 24 премества върха x от списъка $F[v]$ в списъка $F[u]$. Щом залепваме по-късия към по-дългия списък, то за всеки преместен елемент дължината на новия списък е поне два пъти по-голяма от дължината на стария. Всеки списък има от 1 до n елемента, затова всеки елемент може да се премести най-много $\lfloor \log_2 n \rfloor$ пъти, а всички елементи общо — не повече от $n \lfloor \log_2 n \rfloor$ пъти. Поради това ред № 24 от алгоритъма се изпълнява $O(n \log n)$ пъти.

И така, времето на втората фаза е от порядъка на $m + n \log n = O(m \log n)$. Следователно определяща за времето на алгоритъма е първата фаза: то е равно на $\Theta(m \log n) = \Theta(m \log m)$ при най-лоши входни данни — когато ребрата на графа са подредени отначало по такъв начин, че максимално да затруднят сортирането.

Втората реализация на абстрактния тип данни Union-Find (сама често наричана така) използва коренови дървета. Тези дървета не съвпадат с дърветата от гората, която се получава при работа на алгоритъма на Крускал. Множеството от върховете на всяко кореново дърво съвпада с множеството от върховете на съответното дърво от гората от алгоритъма на Крускал. Обаче ребрата на кореновото дърво нямат нищо общо с ребрата на дадения граф.

Всяко кореново дърво от структурата Union-Find се описва чрез указатели към родителите. Родител на корена е той самият. Коренът пази една числова характеристика, наречена ранг, която в идеалния случай съвпада с височината на кореновото дърво. При сливане на две дървета коренът на дървото с по-голям ранг става родител на корена на дървото с по-малък ранг; ако двата ранга са равни, то новият ранг е с единица по-голям от тях. Така рангът нараства най-много до $\lfloor \log_2 n \rfloor$. Идентификатор на кореново дърво е коренът. Търсенето му по даден връх изисква изминаване на целия път от върха до корена. Очевидно за бързодействието е добре пътищата да бъдат възможно най-къси, тоест всяко кореново дърво да е възможно най-ниско. Това се постига чрез компресиране на пътища — при изминаване на път от някой връх до корена се пренасочват указателите на всички върхове от изминатия път: коренът става техен родител. Компресирането на пътищата може да намали височината на дървото под стойността на ранга. Въпреки това рангът не се актуализира: за целта би трябвало да обходим цялото кореново дърво, а не само един негов клон, тоест забавянето, причинено от актуализацията на ранга, ще надвиши ползата от компресирането на пътищата. Може да се покаже, че неточни стойности на ранга възникват рядко, тоест имат пренебрежимо малко влияние върху бързината на алгоритъма (и изобщо не влияят на коректността на крайния резултат).

```
MakeSet (x)
 1)  parent[x] ← x
 2)  rank[x] ← 0
```

```
Find (x)
 1)  r ← x
 2)  p ← parent[r]
 3)  while p ≠ r do
 4)      r ← p
 5)      p ← parent[r]
 6)  p ← parent[x]
 7)  while p ≠ r do
 8)      parent[x] ← r
 9)      x ← p
10)      p ← parent[x]
11)  return r
```

```
Union (x ; y)
 1)  rx ← rank[x]
 2)  ry ← rank[y]
 3)  switch
 4)      case rx < ry
 5)          parent[x] ← y
 6)      case rx > ry
 7)          parent[y] ← x
 8)      case rx = ry
 9)          parent[y] ← x
10)          rank[x] ← rx + 1
```

Параметрите на процедурите MakeSet и Find са произволни елементи (върхове на графа). Параметрите на процедурата Union представляват идентификатори на различни множества (т.е. корени на различни коренови дървета). Проверката, че двете множества са различни, е пропусната в кода на процедурата Union, тъй като е направена преди нейното извикване (вж. ред № 10 от алгоритъма на Крускал преди три страници).

Процедурите използват два масива: $\mathrm{rank}\big[1 \ldots n\big]$ и $\mathrm{parent}\big[1 \ldots n\big]$. Първият масив се състои от цели числа — ранговете на множествата. Вторият масив съдържа върхове на дадения граф, но те според уговорката са цели числа от 1 до n, така че в крайна сметка и този масив е числов. Двата масива трябва да се добавят в началото на втората фаза, тоест между ред № 1 и ред № 2 от алгоритъма на Крускал. Заради тези масиви втората фаза от алгоритъма изисква памет $\Theta(n)$ (те са локални променливи за алгоритъма на Крускал, но са нелокални за трите процедури; локалните променливи на процедурите са от примитивен тип, тоест използват малко памет). Освен това алгоритъмът се нуждае от памет $\Theta(m)$ за масива $E\big[1 \ldots m\big]$. Графът е свързан, затова $m = \Omega(n)$ и сложността по памет на тази реализация на алгоритъма на Крускал е $\Theta(m)$ при всякакви входни данни.

Времевата сложност се изследва трудно, поради което няма да правим подробен анализ. Само ще отбележим, че процедурите MakeSet и Union се изпълняват за константно време, а бързината на процедурата Find зависи от височината на дърветата. Поддържаме ги ниски с помощта на ранговете и компресирането на пътища. С амортизиран анализ може да се докаже, че втората фаза от алгоритъма на Крускал изразходва време $\mathrm{O}\big(m\,\alpha(n)\big)$, където $\alpha(n)$ означава обратната функция на Акерман. Теоретично тя расте неограничено, обаче толкова бавно, че на практика е ограничена отгоре, и то от малка константа: неравенството $\alpha(n) < 5$ е в сила за всички практически възможни n. По-големи функционални стойности могат да се получат само за астрономически стойности на аргумента, а толкова големи графи не могат да се поберат в паметта на никое материално устройство. Така че втората фаза на алгоритъма притежава почти линейна времева сложност, следователно е по-бърза от първата фаза. Така първата фаза определя времето на целия алгоритъм: $\Theta(m \log m) = \Theta(m \log n)$ при най-лоши входни данни (когато ребрата на графа са подредени отначало така, че максимално да затруднят сортирането).

Виждаме, че последните две реализации на алгоритъма на Крускал са еднакво ефективни: и двете притежават времева сложност $\Theta(m \log m) = \Theta(m \log n)$ при най-лоши входни данни и сложност по памет $\Theta(m)$ при всякакви входни данни.

Тази памет се изразходва предимно от масива $E\big[1 \ldots m\big]$ за конвертирането на данните, което е само техническа подробност. Можем да премахнем конвертирането, задавайки графа като масив от ребра — наредени тройки от начален връх, краен връх и тегло. Тази промяна няма да намали значително времето, обаче ще спести памет: масивът $E\big[1 \ldots m\big]$ сега става вход на алгоритъма на Крускал и не се брои към допълнителната памет, използвана от алгоритъма; така нейната асимптотична сложност намалява до $\Theta(n)$ при всякакви входни данни.

Алгоритъм на Прим—Ярник

Построява минимално покриващо дърво на свързан неориентиран тегловен граф $G\big(V; E\big)$, V съдържа върховете, а E — ребрата на графа. Графът притежава n върха и m ребра, $n > 0$; представен е чрез списъци на съседство Adj, тоест Adj може да се схваща като псевдоним на E. Теглото на реброто между u и v бележим с $w(u\,;\,v)$; то може да бъде произволно реално число. Освен графа алгоритъмът получава като вход и един негов връх s, от който да започне работа. Изборът на s не влияе на коректността на алгоритъма, така че върхът s може да бъде произволен.

Алгоритъмът на Прим—Ярник се основава на идеята за релаксация — редица от решения, всяко от които е по-добро от предишното. Започваме с върха s и постепенно добавяме върхове; през цялото време имаме дърво, което расте, докато обхване всички върхове на дадения граф. За всеки връх пазим едно число — текущата цена за включване на този връх към дървото; т.е. теглото на някое ребро, влизащо в този връх. С добавяне на все повече върхове към дървото расте броят на възможностите за присъединяване на въпросния връх и цената за включването му може само да намалява при откриване на по-добра възможност. Всеки път добавяме към дървото най-евтиния от чакащите върхове.

Алгоритъм на Прим—Ярник $\big(G\big(V\,;\,E\big),\ s\big)$

1) $\pi\big[1\ldots n\big]$: масив от върхове
2) $\sigma\big[1\ldots n\big]$: масив от състояния
3) $d\big[1\ldots n\big]$: масив от реални числа — цените за включване на върховете към дървото
4) **for each** $v \in V$ **do**
5) $\quad\pi\big[v\big] \leftarrow$ NIL // нулев указател
6) $\quad\sigma\big[v\big] \leftarrow$ чакащ връх
7) $\quad d\big[v\big] \leftarrow +\infty$
8) $d\big[s\big] \leftarrow 0$
9) **while** $\exists u \in V\colon\ \sigma\big[u\big] =$ чакащ връх **do**
10) $\quad u \leftarrow$ чакащ връх с най-малка стойност d
11) $\quad$**for each** $v \in \mathrm{Adj}\,(u)$ **do**
12) $\quad\quad$**if** $\sigma\big[v\big] =$ чакащ връх **and** $w(u\,;\,v) < d\big[v\big]$
13) $\quad\quad\quad d\big[v\big] \leftarrow w(u\,;\,v)$
14) $\quad\quad\quad \pi\big[v\big] \leftarrow u$
15) $\quad\sigma\big[u\big] \leftarrow$ присъединен връх
16) **return** $\pi\big[1\ldots n\big],\ d\big[1\ldots n\big]$

Присвояването от ред № 10 се извършва едновременно с търсенето на върха u от ред № 9. В псевдокода тези две операции са разделени само за удобство.

Релаксацията се осъществява от редове № 13 и № 14.

Масивът $d\big[1\ldots n\big]$ има важно значение за релаксацията, но не е особено полезен като изход и може да бъде премахнат от ред № 16. Сборът на числата от d при излизане от алгоритъма е равен на теглото на минималното покриващо дърво.

Масивът $\pi\big[1\ldots n\big]$ е по-важната част от изхода. Той описва минималното покриващо дърво като кореново дърво, представено чрез указатели към родителите: s е коренът на дървото, $\pi\big[u\big]$ е родителят на u в кореновото дърво. Само коренът не притежава родител: $\pi\big[s\big] =$ NIL. С други думи, в края на алгоритъма $\pi\big[u\big]$ указва другия край на реброто, чрез което върхът u се включва в дървото. А докато алгоритъмът работи и върхът u не е присъединен към дървото, $\pi\big[u\big]$ указва най-доброто известно до момента ребро, чрез което върхът u може да се присъедини.

Циклите с начала на редове № 4 и № 11 завършват винаги, защото обхождат крайни списъци. Те са цикли по брояч. Техни полуинварианти са анонимните им броячи.

Ред № 9 е начало на цикъл по условие. Негов полуинвариант е броят на чакащите върхове, който намалява с единица при всяко изпълнение на ред № 15. Следователно тялото на този цикъл се изпълнява точно n пъти.

И така, алгоритъмът на Прим—Ярник завършва винаги. Твърденията за неговия изход, изказани по-горе, се доказват с инварианти — по един за всеки цикъл. Двата цикъла по брояч са достатъчно ясни, така че ще пропуснем техните инварианти. Важен е цикълът по условие.

Инвариант: Всеки път, когато се извършва проверката на ред № 9, важат едновременно следните твърдения:

— За върха s са в сила равенствата $d\big[s\big] = 0$ и $\pi\big[s\big] =$ NIL.

— $\forall x \in V \setminus \big\{s\big\}\colon\ d\big[x\big] = +\infty \iff \pi\big[x\big] =$ NIL.

— За всеки присъединен връх $x \neq s$ е в сила неравенството $\pi\big[x\big] \neq$ NIL.

— $\forall x \in V\colon\ \pi\big[x\big] \neq$ NIL $\Rightarrow\ \pi\big[x\big]$ е присъединен връх $\wedge\ \big\{\pi\big[x\big]\,;\,x\big\} \in E\ \wedge\ d\big[x\big] = w\big(\pi\big[x\big]\,;\,x\big)$.

— Ако съществуват присъединени върхове, тогава те заедно с всички ребра от вида $\big\{\pi\big[x\big]\,;\,x\big\}$, където x пробягва присъединените върхове без s, образуват кореново дърво с корен s.

— Множеството от ребрата на това кореново дърво е подмножество на множеството от ребрата на някое минимално покриващо дърво на G.

— За всеки чакащ връх $x \neq s$: $\quad d\big[x\big] = \min\Big\{w\big(y\,;\,x\big)\colon \big\{y\,;\,x\big\} \in E\ \wedge\ y$ е присъединен връх$\Big\}$, където $\min\varnothing = +\infty$; и ако $d\big[x\big] \neq +\infty$, то $\pi\big[x\big]$ е онзи връх y, за който се достига минимумът.

Доказателство: чрез математическа индукция по поредния номер на проверката за край на цикъла с начало на ред № 9.

База: Първата проверка за край се извършва при влизане в цикъла с начало на ред № 9. Първото и второто твърдение от инварианта са непосредствени следствия от командите на редове № 4, № 5, № 7 и № 8 от алгоритъма. Заради ред № 6 всички върхове са чакащи, следователно третото твърдение е тривиално вярно: кванторът за всеобщност е върху празно множество. Четвъртото твърдение е вярно като импликация с неверен антецедент (това следва от ред № 5). Петото твърдение е вярно като импликация с неверен антецедент: всички върхове са чакащи заради ред № 6. По същата причина важи и шестото твърдение, което е продължение на петото (заради думите "това кореново дърво"), така че се подразбира същото условие; двете твърдения са отделени с цел по-лесно възприемане. Тъй като всички върхове са чакащи, то равенството от седмото твърдение е равносилно на $d\big[x\big] = \min \varnothing$, тоест $d\big[x\big] = +\infty$, за всеки връх $x \neq s$, което е изпълнено заради ред № 7. Частта от седмото твърдение след знака точка и запетая е вярна като импликация с неверен антецедент.

Индуктивна стъпка: Нека седемте твърдения са в сила при някоя проверка за край, която не е последна. Ще докажем, че те ще продължат да важат и при следващата проверка, тоест след еднократно изпълнение на тялото на цикъла с начало на ред № 9.

— Първото твърдение е било вярно според индуктивното предположение, а ще бъде в сила след изпълнение на тялото на цикъла, тъй като $d\big[s\big]$ и $\pi\big[s\big]$ няма да се променя̀т. Това е така, защото само редовете № 13 и № 14 промѐнят стойности в масивите d и π, но тези две команди никога не се изпълняват с $v = s$. При първото изпълнение на тялото на цикъла имаме $u = s$, следователно $v \neq s$, защото $v \neq u$: не може $v = u$, понеже $v \in \mathrm{Adj}(u)$, а графът няма примки. Първото изпълнение на тялото на цикъла превръща s от чакащ в присъединен връх (ред № 15) и s остава присъединен връх до края на алгоритъма, тъй като обратното превръщане не се прави никъде в цикъла. При следващи изпълнения на тялото на цикъла $v \neq s$ на редове № 13 и № 14, защото те се изпълняват само върху някой чакащ връх v (заради първата проверка на ред № 12), а пък s е присъединен връх.

— Второто твърдение е било вярно според индуктивното предположение и ще остане вярно, ако $d\big[x\big]$ и $\pi\big[x\big]$ запазят стойностите си. А те могат да се променя̀т само от редове № 13 и № 14, които винаги се изпълняват заедно. След такава промяна $d\big[x\big]$ ще притежава крайна стойност, а пък $\pi\big[x\big]$ ще бъде ненулев указател, така че второто твърдение на инварианта пак ще е вярно: двете страни на еквивалентността ще бъдат неистинни.

— Нека x е върхът от третото твърдение, $x \neq s$. Ако x е бил присъединен към дървото преди текущото изпълнение на тялото на цикъла, то $\pi\big[x\big] \neq \mathrm{NIL}$ в началото на това изпълнение съгласно с индуктивното предположение и ще остане ненулев до следващата проверка за край, защото не може да бъде променен в тялото на цикъла: само ред № 14 би могъл да стори това, но неговото изпълнение ще бъде предотвратено от проверката на ред № 12. Ако пък върхът x се присъединява при текущото изпълнение на тялото на цикъла, то $x = u$ съгласно с ред № 15. Затова $u \neq s$. Указателят $\pi\big[u\big]$ не се променя никъде в цикъла, вкл. на ред № 14, тъй като $v \neq u$: $v \in \mathrm{Adj}(u)$ според ред № 11, а графът не съдържа примки. За да докажем третото твърдение, остава да установим, че този указател е бил ненулев от по-рано. Нека допуснем противното: че $\pi\big[u\big] = \mathrm{NIL}$ в началото на текущото изпълнение на тялото на цикъла. Тогава имаме право да използваме индуктивното предположение. От $\pi\big[u\big] = \mathrm{NIL}$ и $u \neq s$ по второто твърдение следва, че $d\big[u\big] = +\infty$. От ред № 10 правим извод, че $d\big[z\big] = +\infty$ за всеки чакащ връх z, в това число за всеки чакащ връх $z \neq s$ (има поне един такъв връх z, а именно u). Още веднъж прилагаме индуктивното предположение, обаче този път се позоваваме на седмото твърдение: няма ребра между чакащите върхове, различни от s, и присъединените върхове. Понеже $u \neq s$, то текущото изпълнение на тялото на цикъла не е първото му изпълнение и от разсъжденията в доказателството на първото твърдение заключаваме, че s е присъединен връх. Затова можем да опростим получения извод така: няма ребра между чакащите и присъединените върхове. Това означава, че графът е несвързан, което противоречи на изискванията към входните данни.

— Според индуктивното предположение четвъртото твърдение е било вярно при проверката непосредствено преди текущото изпълнение на тялото на цикъла **while** с начало на ред № 9. Твърдението ще остане в сила и при следващата проверка, ако $\pi[x]$ и $d[x]$ не се променят, тоест ако редовете № 13 и № 14 от кода не се изпълнят с $v = x$. Ако пък се изпълнят с $v = x$, то при новата проверка на условието за край на цикъла ще важат равенствата $\pi[x] = \pi[v] = u$ и $d[x] = d[v] = w(u\,;\,v) = w(\pi[x]\,;\,x)$. Следователно $\pi[x] \neq \text{NIL}$, тоест при новата проверка четвъртото твърдение ще има верен антецедент. Остава да докажем истинността на консеквента. Той е конюнкция от три съждения. Преди малко проверихме едно от тях: $d[x] = w\big(\pi[x]\,;\,x\big)$. Второто гласи: $\big\{\pi[x]\,;\,x\big\} \in E$, което е равносилно на $\big\{u\,;\,v\big\} \in E$, а то е вярно, защото $v \in \text{Adj}\,(u)$ съгласно с ред № 11. Третото съждение казва, че върхът $\pi[x]$, тоест u, е присъединен връх. Според ред № 10 това не е така в момента, когато се изпълняват редовете № 13 и № 14. Обаче не е задължително инвариантът да важи през цялото време на изпълнение на цикъла **while**. Истинността му е задължителна единствено когато се извършва проверката за край на ред № 9. А след като цикълът изпълни ред № 13 и ред № 14, преди да отиде на ред № 9 за нова проверка, ще изпълни ред № 15, тоест ще обяви върха u за присъединен към нарастващото кореново дърво. Това действие е неотменимо: цикълът **while** никога не превръща присъединен връх в чакащ. Ето защо следващия път, когато алгоритъмът достигне до ред № 9, върхът u ще принадлежи на множеството на присъединените върхове.

— Седмото твърдение се отнася за минималните цени. Предполагаме, че то е било вярно в началото на текущото изпълнение на тялото на цикъла **while**. Ще докажем, че то ще важи и в края на изпълнението. Действително, дясната страна на равенството се мени единствено при присъединяване на връх към дървото, а текущото изпълнение на тялото на цикъла **while** присъединява един връх и това е u. Променливата y, пробягвайки присъединените върхове, ще приеме всички стари стойности (защото веднъж присъединен връх не може да стане чакащ) и една нова стойност: $y = u$. Обаче в дясната страна на формулата може да настъпи промяна само за онези чакащи върхове x, за които съществува ребро от вида $\big\{y\,;\,x\big\}$, тоест $\big\{u\,;\,x\big\}$. Цикълът с начало на ред № 11 обхожда ребрата $\big\{u\,;\,x\big\}$, но променливата се казва v вместо x. Докато v пробягва чакащите върхове от $\text{Adj}\,(u)$, втората проверка на ред № 12 от алгоритъма търси по-малка стойност от текущия минимум; ако вътрешният цикъл намери такава стойност, ред № 13 актуализира числото $d[v]$, тоест $d[x]$, по такъв начин, че новата му стойност да е равна на минимума от дясната страна на равенството в седмото твърдение; а пък ред № 14 актуализира указателя $\pi[v]$, тоест $\pi[x]$, по такъв начин, че отново да е вярна частта от седмото твърдение след знака точка и запетая.

— Петото и шестото твърдение образуват една импликация. Нейният антецедент е истина при проверките на ред № 9 след втората включително: има присъединени върхове, например s. Така че всъщност доказваме консеквента. Една тънкост: трябва да отделим втората проверка. (Ако доказвахме само тези две твърдения, втората проверка би била в базата на индукцията. Но ние доказваме конюнкцията на всичките седем твърдения.) След като тялото на цикъла се изпълни за първи път, възниква един и за момента единствен присъединен връх — това е s. Ето защо не съществуват ребра от вида, за който става дума в петото и шестото твърдение. Тоест разглежданият подграф се състои само от върха s и няма ребра, следователно е дърво. Върхът s е негов корен, защото не притежава родител: $\pi[s] = \text{NIL}$ според първото твърдение, което вече е доказано за новата проверка на условието за край на цикъла **while**. Освен това кореновото дърво има празно множество от ребра, а то е подмножество на всяко множество, включително на множеството от ребрата на произволно избрано минимално покриващо дърво (съществува такова дърво, защото графът е свързан). Така че петото и шестото твърдение важат при втората проверка за край на цикъла **while**.

За следващите проверки ще се наложи да използваме индуктивното предположение. Нека за краткост да въведем следните обозначения: P е множеството от присъединените върхове, A е множеството от ребрата на G от вида $\big\{\pi[x]\,;\,x\big\}$, където x пробягва множеството $P \setminus \big\{s\big\}$, при текущата проверка за край на цикъла **while**, която не е последна (за да има следваща) и не е първа (вече доказахме, че петото и шестото твърдение ще важат при втората проверка). Нека $H\big(P\,;\,A\big)$ е графът с върхове P и ребра A.

Понеже текущата проверка не е първата, то $P \neq \varnothing$; по-точно, $s \in P$. Възможно е $A = \varnothing$. За да е правилно определен графът H, трябва краищата на ребрата от A да принадлежат на P. За краищата от вида x това е вярно по определение. За краищата от вида $\pi[x]$ при $x \neq s$ това следва от индуктивното предположение: $\pi[x] \neq \mathrm{NIL}$ според третото твърдение и $\pi[x] \in P$ според четвъртото. Пак от четвъртото твърдение следва, че $A \subseteq E$, а по определение $P \subseteq V$, т.е. H е подграф на G. Според индуктивното предположение за петото и шестото твърдение H е дърво с корен s, представено чрез указатели π към родителите, а пък A е подмножество на множеството от ребрата на някое минимално покриващо дърво T на G.

Нека P', A' и $H'(P'; A')$ са съответните множества и граф при следващата проверка за край на цикъла **while**. За третото и четвъртото твърдение вече установихме, че ще важат при следващата проверка, затова можем да ги използваме по същия начин: от тях следва, че H' е правилно определен подграф на G.

Тъй като текущата проверка за край на цикъла **while** не е нито първа, нито последна, тялото на цикъла ще се изпълни (т.е. има чакащ връх u), и то не за първи път (значи, $u \neq s$). Върхът u (и само той) ще се присъедини към H на ред № 15. По-рано присъединените върхове ще останат такива. Формален запис: $u \notin P$, $u \in P'$, $P' = P \cup \{u\}$, $e \notin A$, $e \in A'$, $A' = A \cup \{e\}$, където $e = \{\pi[u]; u\}$. Това, че $\pi[u] \neq \mathrm{NIL}$ и $e \in E$, вече беше доказано, когато установихме, че H' е правилно определен подграф на G.

Графът H е дърво, следователно е свързан и не съдържа цикли. Добавянето на реброто e не създава цикли, защото u е нов връх ($u \notin P$). Освен това графът G не съдържа примки, затова за всяко $v \in \mathrm{Adj}(u)$ от ред № 11 важи неравенството $v \neq u$ и ред № 14 не променя $\pi[u]$. Понеже $\pi[u] \neq \mathrm{NIL}$ при следващата проверка за край (след текущото изпълнение на тялото на цикъла **while**), то същото неравенство е било вярно и преди това — при текущата проверка за край на цикъла. От индуктивното предположение за четвъртото твърдение правим извод, че върхът $\pi[u]$ е бил присъединен по-рано, тоест $\pi[u] \in P$. Ето защо добавянето на реброто e не нарушава свързаността на графа H.

Тези разсъждения показват, че H' е свързан неориентиран граф без цикли, тоест дърво. То е представено с указатели към родителите: за върха u и реброто e го доказахме току-що; за върховете и ребрата от H прилагаме индуктивното предположение за петото твърдение. От индуктивното заключение, вече доказано за първото и третото от седемте твърдения, следва, че измежду върховете на H' единствено s няма родител. Значи, s е коренът на дървото H'.

Според индуктивното предположение за шестото твърдение, A се съдържа сред ребрата на някое минимално покриващо дърво T на графа G. Ако $e \in T$, то T съдържа $A' = A \cup \{e\}$ и няма какво повече да се доказва. Нека $e \notin T$. Тогава $T \cup \{e\}$ съдържа поне един цикъл, а оттам — поне един цикъл C, неповтарящ върхове. T е дърво и няма цикли, ето защо $e \in C$. Понеже H' е дърво, то H' не съдържа цикли, следователно поне едно ребро на C не е от A'. Ще докажем, че сред ребрата от $C \setminus A'$ съществува ребро f, не по-леко от e.

Да обходим C в посока от $\pi[u]$ към u, тоест $\pi[u]$, u, a, b, c, $\ldots$, $\pi[u]$. Ребрата на C са ребра на $T \cup \{e\}$, а значи и на G. Щом G не съдържа примки, съседните върхове в редицата са различни. В частност $a \neq u$ и $\{a; u\}$ е ребро на G. Освен това G не съдържа кратни ребра, така че върховете през един в редицата също са различни. В частност $a \neq \pi[u]$.

Първи случай: $a \notin P'$. Понеже $P' \supseteq P$, то $a \notin P$. По-горе доказахме, че $\pi[u] \in P$, $u \notin P$. Следователно в редицата a, b, c, $\ldots$, k, ℓ, $\ldots$, $\pi[u]$ има два поредни върха $k \notin P$ и $\ell \in P$. Оказва се, че реброто $f = \{\ell; k\}$ е търсеното. Щом u липсва в скъсената редица от върхове, то $k \neq u$ и понеже $k \notin P$, то $k \notin P'$, следователно $f \notin A'$. Тъй като $s \in P$, но $k \notin P$, то $k \neq s$. От индуктивното предположение за седмото твърдение в частта преди знака точка и запетая, приложено при $x = k \neq s$, извличаме неравенството $d(k) \leq w(\ell; k) = w(f)$, а от втората част на същото твърдение, приложено при $x = u \neq s$, следва равенството $d(u) = w(\pi[u]; u) = w(e)$. Понеже прилагаме индуктивното предположение, тези стойности на $d(k)$ и $d(u)$ са актуални в началото на текущото изпълнение на тялото на цикъла **while**. Върховете u и k в този миг са чакащи, защото $u \notin P$, $k \notin P$. От ред № 10 следва, че $d(u) \leq d(k)$, тоест $w(e) \leq d(k) \leq w(f)$.

Втори случай: $a \in P'$. Тъй като $a \neq u$, то $a \in P$. По-горе доказахме, че $\pi\big[u\big] \in P$, $u \notin P$. Оказва се, че реброто $f = \big\{a\,;\,u\big\}$ е търсеното. При $x = u \neq s$ индуктивното предположение за седмото твърдение в частта след знака точка и запетая води до неравенството $w\big(e\big) \leq w\big(f\big)$. Освен това $f \notin A'$, защото инак върхът u би бил поне от втора степен в H', а това е невъзможно: току-що присъединен към дървото, този връх е листо, следователно е от първа степен.

Доказахме, че във всички случаи съществува ребро $f \in C \setminus A'$, за което $w\big(e\big) \leq w\big(f\big)$. Полагаме $T_2 = T \cup \big\{e\big\} \setminus \big\{f\big\}$. Покриващото дърво T е свързан граф с n върха и $n-1$ ребра. Понеже $e \notin T$, то $T \cup \big\{e\big\}$ е свързан граф с n върха и n ребра. Премахването на ребро от цикъл не нарушава свързаността, затова T_2 е свързан неориентиран граф с n върха и $n-1$ ребра, т.е. покриващо дърво. От съотношенията $w\big(T_2\big) = w\big(T\big) + w\big(e\big) - w\big(f\big)$ и $w\big(e\big) \leq w\big(f\big)$ следва неравенството $w\big(T_2\big) \leq w\big(T\big)$. Строго неравенство не е възможно поради минималността на T. Затова е изпълнено равенството $w\big(T_2\big) = w\big(T\big)$, тоест T_2 също е минимално покриващо дърво.

Тъй като $T \supseteq A$, то $T \cup \big\{e\big\} \supseteq A \cup \big\{e\big\} = A'$. А понеже $f \notin A'$, то $T_2 \supseteq A'$, което доказва индуктивното заключение и за шестото твърдение от инварианта.

С това индуктивната стъпка е завършена и инвариантът е доказан.

Преди инварианта установихме, че алгоритъмът на Прим—Ярник завършва непременно. Последната проверка на ред № 9 прекратява цикъла **while**, т.е. няма повече чакащи върхове. Следователно всички върхове са присъединени към дървото, представено чрез масива $\pi\big[1\,\ldots\,n\big]$ от указатели към родителите. Че този масив наистина представя кореново дърво с корен s, следва от петото твърдение на инварианта. В края на алгоритъма това дърво, както казахме, съдържа всички върхове, значи е покриващо дърво на графа G. Според шестото твърдение ребрата на това дърво са подмножество на ребрата на някое минимално покриващо дърво на G. Тъй като двете дървета са покриващи и едното е подграф на другото, те съвпадат, следователно масивът $\pi\big[1\,\ldots\,n\big]$, върнат от алгоритъма, представя минимално покриващо дърво на G.

Остана да докажем коректността на другата част от изхода — числовия масив $d\big[1\,\ldots\,n\big]$. Сборът от числата му трябва да е равен на теглото на намереното минимално покриващо дърво. От първото твърдение на инварианта следва, че можем да пренебрегнем събираемото $d\big[s\big] = 0$. Понеже всички върхове на G са присъединени, то от третото твърдение на инварианта следва, че всички върхове без s имат родители в дървото, а от четвъртото твърдение — че числото $d\big[x\big]$ на всеки връх $x \neq s$ е равно на теглото на реброто между x и неговия родител. Следователно сборът на числата от масива $d\big[1\,\ldots\,n\big]$ е равен на сбора от теглата на ребрата на намереното минимално покриващо дърво.

Сложността на алгоритъма на Прим—Ярник зависи от реализацията.

Ако се използват единствено трите масива от редовете № 1, № 2 и № 3 на алгоритъма (без никакви други структури от данни), то сложността по памет е $\Theta(n)$ във всички случаи. Тук влиза паметта за масива от състояния $\sigma\big[1\,\ldots\,n\big]$, който представлява локална променлива. Масивът $\pi\big[1\,\ldots\,n\big]$ не се брои: той е изход, използван само за писане и може да се реализира чрез отпечатване на стойностите или изпращане на съобщения. Обаче масивът $d\big[1\,\ldots\,n\big]$ се брои: въпреки че е изход, той се използва и за четене, тоест служи и като локална променлива. Другите локални променливи (u и v) са от примитивен тип и изразходват съвсем малко памет. Сложността по време е сбор от времената на отделните операции. Ред № 8 отнема време $\Theta(1)$; цикълът с начало на ред № 4 — време $\Theta(n)$; ред № 15 — време $\Theta(n)$ общо за целия алгоритъм, защото този ред се изпълнява n пъти (по веднъж за всеки връх на графа). Ред № 9 и ред № 10 изразходват време $\Theta\big(n^2\big)$ общо за целия алгоритъм, тъй като се изпълняват точно по n пъти и всеки път обхождат масивите $\sigma\big[1\,\ldots\,n\big]$ и $d\big[1\,\ldots\,n\big]$. Конюнкцията на ред № 12 се проверява $2m$ пъти — по два пъти за всяко ребро $\big\{u\,;\,v\big\}$, защото то участва в два списъка на съседство: $v \in \mathrm{Adj}\,(u)$ и $u \in \mathrm{Adj}\,(v)$. Следователно всеки от редовете № 13 и № 14 може да бъде изпълнен не повече от $2m$ пъти (тази горна граница може да се намали, но по-точна оценка не е нужна). Понеже $m = \mathrm{O}\big(n^2\big)$ за всеки граф, то най-голямото от всички събираеми е $\Theta\big(n^2\big)$; то определя асимптотичния порядък на сбора. Затова времевата сложност на тази реализация на алгоритъма е равна на $\Theta\big(n^2\big)$ при всякакви входни данни.

Както виждаме, най-бавна операция е търсенето на чакащ връх с минимално число d. Можем да я ускорим с помощта на приоритетна опашка, реализирана с двоична пирамида по такъв начин, че ключът на всеки елемент на пирамидата да е не по-голям от ключовете на наследниците му. Следователно на върха на пирамидата стои елементът с най-малък ключ. Елементите на пирамидата са наредени тройки $\langle d[u]\,;\,\pi[u]\,;\,u\rangle$, където u е връх на графа, а $d[u]$ е ключ на тройката. Редовете от № 4 до № 8 вкл. се изпълняват пак за общо време $\Theta(n)$: на върха на пирамидата поставяме тройката $\langle 0\,;\,\mathrm{NIL}\,;\,s\rangle$, а останалите $n-1$ места попълваме в произволен ред с тройки $\langle +\infty\,;\,\mathrm{NIL}\,;\,u\rangle$ за всеки връх $u \neq s$. Всеки път, когато прочитаме елемента от върха на пирамидата, го изтриваме от нея, така че няма нужда от масива $\sigma[1\ldots n]$ и можем да изтрием редове № 2, № 6 и № 15: чакащи върхове са тези от приоритетната опашка; присъединени към дървото са върховете, извадени от опашката. За сметка на това е нужен допълнителен масив с n елемента, чрез който да намираме мястото на всеки връх в пирамидата: релаксацията намалява ключовете и елементите се придвижват към върха на пирамидата. Допълнителният масив, масивът d и самата опашка имат по n елемента, откъдето следва, че сложността по памет е отново $\Theta(n)$ при всякакви входни данни. Времевата сложност се променя. Ред № 9 се изпълнява вече за константно време всеки път (проверява дали пирамидата е празна), а общо — за време $\Theta(n)$. Ред № 10 извлича елемента, който се намира на върха на пирамидата, така че еднократно изпълнение на ред № 10 изразходва време $\Theta(\log n)$ в най-лошия случай: когато новият елемент на върха трябва да бъде спуснат чак до дъното на двоичната пирамида. Понеже ред № 10 се изпълнява n пъти (по веднъж за всеки връх на графа), то общият му принос към времевата сложност на алгоритъма е $\mathrm{O}(n \log n)$. Пишем O вместо Θ, защото този израз представлява само горна граница на времето: броят на елементите на пирамидата не остава n, а намалява с извличането им един по един. Събираемото $\Theta\bigl(n^2\bigr)$ отпада, заместено от $\mathrm{O}(n \log n)$. За сметка на това се увеличава времето на ред № 13: присвояването на стойност намалява ключа на елемент от пирамидата и може да се наложи елементът да се придвижи към нейния връх. Това отнема време $\Theta(\log n)$ при еднократно извикване в най-лошия случай: когато елементът се придвижва от дъното до върха на пирамидата; общо за целия алгоритъм — време $\mathrm{O}(m \log n)$, защото намаляване на ключ се прави не повече от $2m$ пъти — по два пъти за всяко ребро (множителят 2 може да бъде намален до 1, но това не е нужно при оценка само по порядък). Ето защо времевата сложност на тази реализация на алгоритъма е равна на $\mathrm{O}\bigl((m+n) \log n\bigr)$ при всякакви входни данни. Друг запис: $\mathrm{O}(m \log n)$, тъй като $m = \Omega(n)$ за свързан граф.

Доказахме горна граница на времевата сложност. Ще докажем, че тя е и долна граница, като за всички допустими n и m построим свързан неориентиран граф с n върха и m ребра, който кара алгоритъма да изразходва време $\Omega(m \log n)$. Трудно е да зададем графа наведнъж, вместо това ще го задаваме стъпка по стъпка. Тоест всеки входен елемент ще се конкретизира едва когато алгоритъмът поиска да го прочете. Нека ред № 10 на алгоритъма току-що е изтрил елемент u от върха на пирамидата. Ред № 11 започва да иска входни данни за ребрата от u. Можем да си представяме обхождането на списъка $\mathrm{Adj}(u)$ като поредица от запитвания, отправени от алгоритъма към генератора на входните данни: “Дай ми следващото ребро $\{u\,;\,v\}$.” Генераторът поддържа списък на дадените досега отговори и оставащ брой ребра (отначало m). Най-напред генераторът подава върховете v, за които вече е създал ребро $\{v\,;\,u\}$ с разменени роли на краищата, тоест $u \in \mathrm{Adj}(v)$; в такъв случай задава същото тегло: $w(u\,;\,v) = w(v\,;\,u)$, защото входните данни не бива да са противоречиви, и не намалява броя на оставащите ребра, понеже това ребро вече е броено. За такива ребра върхът v е вече обработен (присъединен), т.е. има най-много една релаксация на ребро (и можем да заменим $2m$ с m в горната граница). След това генераторът на входни данни подава в ролята на v всеки от чакащите върхове без u, като намалява броя на оставащите ребра, докато изчерпи броя на ребрата или чакащите върхове. Важен е редът на чакащите върхове: първи са листата на двоичната пирамида. За всяко листо v генераторът избира тегло $w(u\,;\,v)$, което е по-малко от цените на всички чакащи върхове, вкл. v. (Ред № 13 превръща това тегло в цена на v и следващото тегло трябва да бъде още по-малко.) Така генераторът не само предизвиква много релаксации (по една за всеки чакащ съсед на u), но и принуждава елементите от листата на пирамидата да се изкачат един след друг до върха ѝ; всеки от тях изминава път с дължина около $\log_2 k$, където k е броят елементи на пирамидата.

Броят k на чакащите върхове на графа намалява с течение на времето.

Първи случай: ребрата (m на брой) се изчерпват, преди k да намалее до $n/2$. Това означава, че всички $k \geq n/2$, ето защо всички елементи, които са били листа на пирамидата, са изминали път с дължина поне $\log_2(n/2) = (\log_2 n) - 1$. Но за всяко k в листата стоят около половината от елементите на пирамидата, тоест около половината от всички ребра на графа предизвикват такъв брой разместванията. Следователно общият брой разместванията на елементи на пирамидата е поне $(m/2)\big(-1 + \log_2 n\big)$, тоест $\Omega(m \log n)$. Малките неточности се дължат на закръгляване и не влияят на порядъка.

Втори случай: разполагаемият брой m ребра се изчерпва, след като k е намаляло до $n/2$. Да разгледаме само първите $n/2$ върха на графа, обработени в ролята на u от цикъла **while**. От всеки такъв връх излизат ребра към всеки от останалите $n/2$ върха на графа, което прави поне $n^2/4$ ребра, за поне половината от които са били извършени поне $\log_2(n/2) = (\log_2 n) - 1$ разместванията на елементи в двоичната пирамида. Следователно в този случай са извършени общо поне $\big(n^2/8\big)\big(-1 + \log_2 n\big)$, тоест $\Omega\big(n^2 \log n\big)$, разместванията. Освен това $n^2/4 \leq m \leq n(n-1)/2$, откъдето следва асимптотичното равенство $m = \Theta\big(n^2\big)$. Въз основа на това равенство можем да заместим n^2 с m в долната граница. Така получаваме отново $\Omega(m \log n)$.

С това са разгледани всички случаи и е завършено доказателството на долната граница. По порядък тя съвпада с горната граница, затова времевата сложност на втората реализация при най-лоши входни данни е равна на $\Theta(m \log n) = \Theta(m \log m)$.

Алгоритъмът на Прим—Ярник притежава и трета версия — отново с приоритетна опашка, но този път реализирана с пирамида на Фибоначи. Тази структура от данни е твърде сложна за настоящото изложение — не само описанието на структурата и разните операции върху нея, а също и обосновката на нейните свойства (доказват се с анализ на амортизираната сложност). Затова привеждаме само изводите: сложността по памет е $\Theta(n)$ при всякакви входни данни, а сложността по време е $\Theta(m + n \log n)$ при най-лоши входни данни.

И тъй, трите разгледани реализации на алгоритъма на Прим—Ярник имат една и съща сложност по памет: $\Theta(n)$ при всякакви входни данни. (Еднакъв е само асимптотичният порядък, константните множители може да се различават.) Времевата сложност е с различен порядък:
— за реализацията без приоритетна опашка: $\Theta\big(n^2\big)$ при всякакви входни данни;
— за реализацията с двоична пирамида: $\Theta(m \log n)$ при най-лоши входни данни;
— за реализацията с пирамида на Фибоначи: $\Theta(m + n \log n)$ при най-лоши входни данни.

Нека сравним трите версии по порядък на бързодействието при най-лоши входни данни. Първата версия на алгоритъма е по-бърза от втората при $m = \Theta\big(n^2\big)$, тоест при плътни графи. Обратно, втората версия изпреварва първата при $m = \mathrm{o}\big(n^2/\log n\big)$, тоест при неплътни графи. Третата версия е най-бърза при всякакво съотношение между броя на ребрата и на върховете, защото $m + n \log n = \mathrm{O}\big(n^2\big)$ и $m + n \log n = \mathrm{O}\big(m \log n\big)$.

Бързите реализации на алгоритъма на Крускал имат времева сложност от същия порядък като втората реализация на алгоритъма на Прим—Ярник, следователно отстъпват на третата. На практика се използват и двата алгоритъма — в първоначалния си вид или в променена форма.

Двата алгоритъма работят и с отрицателни тегла, обаче този случай не е особено полезен. Най-често се интересуваме от минимален покриващ свързан граф (той не е непременно дърво). Ако всички ребра имат положителни тегла, задачите съвпадат: изтриването на ребро от цикъл намалява теглото на покриващия граф, без да нарушава неговата свързаност, следователно минималният покриващ свързан граф няма цикли, тоест той е минимално покриващо дърво. Ако всички тегла са неотрицателни и между тях се срещат нули, понякога може да се намери минимален покриващ свързан граф, който не е дърво, но все пак ще има и такъв, който е дърво (премахваме нулеви ребра от циклите). Но ако графът съдържа ребра с отрицателни тегла, двете задачи не са равносилни: добавяйки отрицателно ребро към минимално покриващо дърво, може да получим покриващ свързан граф с по-малко тегло, който не е дърво.

Най-къси пътища
в граф с неотрицателни тегла на ребрата

Задачата за търсене на най-къс път в граф е изключително важна. Върховете на графа представят някакви състояния, а ребрата — преходи между тях. Според значението на теглата най-късият път се тълкува като най-бърз или най-евтин преход от едно състояние в друго.

Постановка на задачата: Даден е ориентиран тегловен граф $G\big(V;\,E\big)$ с n върха и m ребра. V е множеството от върховете, E е множеството от ребрата на G, $n = |V| > 0$, $m = |E| \geq 0$. Теглото на реброто $(u\,;\,v)$ ще означаваме с $w(u\,;\,v)$ и ще тълкуваме като дължина на реброто; теглото е реално число. Даден е също един връх s на графа. Търсят се най-къси пътища от s до всички върхове на G. Дължината на път е равна на сбора от дължините на ребрата му.

За нетегловни графи дължината на път е равна на броя на ребрата му, а задачата се решава чрез търсене в ширина за време $\Theta(n+m)$ при всякакви входни данни. Обхождането в ширина се пуска от върха s.

За тегловни ациклични графи задачата се решава с помощта на динамично програмиране. Времевата сложност по порядък е пак $\Theta(n+m)$ при всякакви входни данни, обаче ненаписаният константен множител пред порядъка е двойно по-голям, защото графът се обхожда два пъти: за подреждането на върховете (топологично сортиране) и за търсенето на най-къси пътища. Теглата могат да бъдат произволни реални числа — положителни, отрицателни, нули.

За тегловни графи, които не са ациклични, динамичното програмиране не е приложимо. За тях не е известен алгоритъм с линейна времева сложност. Има алгоритми с различна бързина в зависимост от някои предположения за теглата на ребрата. Ако всички тегла са неотрицателни, съществува алгоритъм, чиято времева сложност е почти линейна — алгоритъмът на Дейкстра. Използва се релаксация — скъсяване, когато е възможно, на текущия най-къс път до някой връх. Ограничението за неотрицателни тегла не намалява практическата значимост на алгоритъма: цените, дължините, сроковете и въобще почти всички величини, срещани в практически задачи, са неотрицателни числа.

Алгоритъм на Дейкстра $\big(G\big(V;\,E\big),\;s\big)$

```
 1)  π[1...n]: масив от върхове
 2)  σ[1...n]: масив от състояния
 3)  d[1...n]: масив от реални числа — дължините на най-късите пътища от s до всеки връх
 4)  for each v ∈ V do
 5)      π[v] ← NIL   // нулев указател
 6)      σ[v] ← чакащ връх
 7)      d[v] ← +∞
 8)  d[s] ← 0
 9)  while ∃u ∈ V: σ[u] = чакащ връх and d[u] < +∞ do
10)      u ← чакащ връх с най-малка стойност d
11)      for each v ∈ Adj(u) do
12)          if σ[v] = чакащ връх and d[u] + w(u; v) < d[v]
13)              d[v] ← d[u] + w(u; v)
14)              π[v] ← u
15)      σ[u] ← присъединен връх
16)  return π[1...n], d[1...n]
```

Алгоритъмът на Дейкстра построява дърво на най-къси пътища от s до всички върхове. Дървото е представено чрез масива $\pi\big[1\ldots n\big]$ от указатели към родителите. За всеки връх u най-къс път от s до u се намира чрез проследяване на редицата от указатели към родителите: u, $\pi\big[u\big]$, $\pi\big[\pi\big[u\big]\big]$, … , s. Така върховете от пътя се получават в обратен ред. В действителност при спазване на посоката на ребрата на G пътят води от s до u, тоест s, … , $\pi\big[\pi\big[u\big]\big]$, $\pi\big[u\big]$, u, а дължината му е $d\big[u\big]$. Ако алгоритъмът не е открил път от s до u, то $d\big[u\big] = +\infty$, $\pi\big[u\big] = \text{NIL}$.

Алгоритъмът на Дейкстра работи и върху мултиграфи, но този вариант не е особено полезен. Винаги можем да премахнем еднопосочните ребра между една и съща двойка върхове на графа, оставяйки само едно от тях — най-късото. Това не променя дървото на най-късите пътища. Затова не е съществено ограничение, ако предположим, че G е обикновен граф, не мултиграф. (Разрешаваме по две противопосочни ребра между една и съща двойка върхове; така представяме неориентираните графи.) Поради липсата на кратни ребра $m \le n^2$, тоест $m = \mathrm{O}\big(n^2\big)$.

Примките не са пречка за работата на алгоритъма, но са излишни и можем да ги махнем: примка с положително тегло не може да участва в най-къс път (без нея пътят би се скъсил още); примка с нулево тегло може да е част от най-къс път, но след изтриването ѝ се получава друг път със същата дължина. Затова можем да предполагаме, че графът не съдържа примки.

Ако графът е несвързан, може да има пътища от s единствено до върховете от неговата компонента на слаба свързаност. Другите компоненти са излишни и можем да предполагаме, че графът е слабо свързан. Понеже е ориентиран, това предположение изобщо не гарантира съществуването на пътища дори за върховете от компонентата на s. Следва само, че $m = \Omega(n)$.

Сравнението между алгоритмите на Дейкстра и на Прим—Ярник показва само една разлика между техните описания: ред № 12 сравнява $d\big[v\big]$ съответно с $d\big[u\big] + w(u\,;\,v)$ или само с $w(u\,;\,v)$, като същото се отнася и за присвояването на стойност на ред № 13. Тази прилика гарантира съвсем еднаква сложност по памет на двата алгоритъма и еднаква по порядък времева сложност (само константният множител на времевата сложност е по-голям при алгоритъма на Дейкстра заради допълнителната операция събиране). Еднакви са също съотношенията между m и n. Така че анализът на сложността на алгоритъма на Прим—Ярник се пренася без никаква промяна към алгоритъма на Дейкстра. И този алгоритъм притежава три реализации с една и съща сложност по памет: $\Theta(n)$ при всякакви входни данни. (Еднакъв е само асимптотичният порядък, константните множители може да се различават.) Времевата сложност е с различен порядък:
— за реализацията без приоритетна опашка: $\Theta\big(n^2\big)$ при всякакви входни данни;
— за реализацията с двоична пирамида: $\Theta(m \log n)$ при най-лоши входни данни;
— за реализацията с пирамида на Фибоначи: $\Theta(m + n \log n)$ при най-лоши входни данни.
Първата версия на алгоритъма е по-бърза от втората при $m = \Theta\big(n^2\big)$, тоест при плътни графи. Обратно, втората версия изпреварва първата при $m = \mathrm{o}\big(n^2/\log n\big)$, тоест при неплътни графи. Третата версия е най-бърза при всякакво съотношение между броя на ребрата и на върховете, защото $m + n \log n = \mathrm{O}\left(n^2\right)$ и $m + n \log n = \mathrm{O}\left(m \log n\right)$.

Заради допълнителната операция събиране и изискването теглата да бъдат неотрицателни, има малка промяна в доказателството на долната граница $\Omega(m \log n)$ на времевата сложност на втората реализация при най-лоши входни данни. При поискване генераторът на входни данни трябва да даде такова тегло на реброто $\big(u\,;\,v\big)$, че да бъдат удовлетворени неравенствата

$$0 < w(u\,;\,v) < \min_z\ d(z) - d(u),$$

където минимумът се взима по всички чакащи върхове $z \ne u$. Това може да бъде постигнато, ако между крайните цени на чакащите върхове няма равни. Отначало това е така, защото има единствен чакащ връх с крайна цена: $d\big[s\big] = 0$. По-нататък самото правило за теглата на ребрата гарантира, че изискването за различни цени е изпълнено. Тоест коректността на тази стратегия следва по индукция.

Между алгоритмите на Прим—Ярник и на Дейкстра има доста съществени разлики:
— Алгоритъмът на Прим—Ярник намира минимално покриващо дърво на даден свързан граф, а алгоритъмът на Дейкстра построява дърво на най-къси пътища.
— Входният параметър s и масивът $d\big[1 \ldots n\big]$ като изход са важни за алгоритъма на Дейкстра, но не и за алгоритъма на Прим—Ярник.
— Алгоритъмът на Прим—Ярник обработва неориентирани графи, а алгоритъмът на Дейкстра е предназначен за ориентирани графи, въпреки че може да работи и върху неориентирани, ако всяко ребро $\big\{u\,;\,v\big\}$ се замени с две ребра — $\big(u\,;\,v\big)$ и $\big(v\,;\,u\big)$.
— Алгоритъмът на Дейкстра работи вярно само за графи с неотрицателни тегла на ребрата, а алгоритъмът на Прим—Ярник обработва правилно и графи с отрицателни тегла.
— Заради различното си предназначение двата алгоритъма имат различни инварианти.

За алгоритъма на Дейкстра няма да съставяме подробен инвариант, както постъпихме при алгоритъма на Прим—Ярник. Поради приликата между описанията на двата алгоритъма ще приемем за доказано, че присъединените върхове винаги образуват кореново дърво с корен s, представено чрез масива $\pi\big[1 \ldots n\big]$ от указатели към родителите. Това доказателство е еднакво с доказателството на съответстващата му част от инварианта на алгоритъма на Прим—Ярник. Ще пропуснем също и формалното доказателство, че $d\big[u\big]$ е дължината на пътя от s до u, образуван от ребрата на дървото (за чакащите върхове u може още да не е открит такъв път, а ако е открит, то последното му ребро не е от дървото). Пропускаме това доказателство, защото твърдението е достатъчно очевидно: ред № 13 от алгоритъма събира дължините на ребрата:

$$w(s — \cdots — u — v) = w(s — \cdots — u) + w(u\,;\,v).$$

Очевидно е също, че указателят на всеки връх, ако е ненулев, сочи към присъединен връх: след ред № 14, преди да достигне до проверката на ред № 9, алгоритъмът изпълнява ред № 15. Ще формулираме изрично само най-съществената част от инварианта — какво представляват пътищата от построеното до момента кореново дърво.

Инвариант: При всяко достигане на ред № 9 от алгоритъма:
— Ако z е присъединен връх, то пътят от s до z, образуван от ребрата на кореновото дърво, е най-къс измежду всички пътища от s до z.
— Ако z е чакащ връх, то пътят от s до z, образуван от ребрата на текущото кореново дърво (ако има такъв път, тоест ако $\pi\big[z\big] \neq \mathrm{NIL}$), е най-къс измежду всички онези пътища от s до z, които съдържат само присъединени върхове с изключение на крайния връх z.

Доказателство: чрез математическа индукция по поредния номер на проверката за край.

База: Първата проверка за край на цикъла се извършва при влизане в него. В този момент твърденията на инварианта са тривиално верни:
— първото — защото няма присъединени върхове (ред № 6 от кода);
— второто — защото няма чакащ връх с ненулев указател към родител (ред № 5 от кода).

Индуктивна стъпка: Нека твърденията са верни при някоя проверка, която не е последна. Ще докажем, че те ще останат в сила и при следващата проверка за край на цикъла **while**. Проследяваме едно изпълнение на тялото на цикъла; u е единственият нов присъединен връх.

Нека z е някой присъединен връх при новата проверка за край на цикъла **while** и $z \neq u$. Следователно върхът z е бил присъединен още преди текущото изпълнение на тялото на цикъла и според индуктивното предположение за първото твърдение в кореновото дърво вече се е пазел най-къс измежду всички пътища от s до z. Първото твърдение ще важи и при новата проверка, защото въпросният път съдържа само присъединени върхове (вкл. z), чиито d и π не се менят: редовете № 13 и № 14 променят d и π само на чакащи върхове.

Върхът u се присъединява към кореновото дърво (на ред № 15) при текущото изпълнение на тялото на цикъла **while**, а преди това е бил чакащ връх (ред № 10), за който имаме право да приложим индуктивното предположение за първото твърдение: алгоритъмът е бил открил най-къс път от s до u, съставен само от присъединени върхове. Техните d и π не се променят, а също $d\big[u\big]$ и $\pi\big[u\big]$: редовете № 13 и № 14 не се изпълняват с $v = u$ дори ако u има примка, защото втората проверка на ред № 12 не допуска това:

$$d\big[u\big] + w(u\,;\,v) < d\big[v\big] \iff d\big[u\big] + w(u\,;\,u) < d\big[u\big] \iff w(u\,;\,u) < 0,$$

което е невъзможно, защото теглата на всички ребра са неотрицателни. Ето защо пътят от s до u, намерен по-рано и пазен в кореновото дърво, не се променя, обаче след присъединяването на u трябва да докажем за пътя второто, а не първото твърдение на инварианта. Второто твърдение е по-силно: че пътят е най-къс не само сред пътищата, съставени само от присъединени върхове, а измежду всички пътища от s до u, включително пътищата, които съдържат чакащи върхове. Ако $u = s$, второто твърдение е очевидно: алгоритъмът е намерил път с дължина 0 (ред № 8), а по-къс няма, защото всички тегла са неотрицателни. Нека $u \neq s$ и да допуснем противното: че съществува по-къс път $s — \cdots — x — y — \cdots — u$, който съдържа поне един чакащ връх (различен от u) и нека y е първият, т.е. върховете от s до x вкл. са присъединени (може $x = s$).

Следователно са изпълнени съотношенията

$$d(u) > w(s - \cdots - x - y - \cdots - u) = w(s - \cdots - x - y) + w(y - \cdots - u) \geq d(y) + 0 = d(y).$$

Тук с d са означени цените на върховете преди текущото изпълнение на тялото на цикъла а с w са означени дължините на пътищата (те не се променят). Първото съотношение изразява направеното допускане (за $d(u)$ използваме индуктивното предположение за второто твърдение). Второто съотношение просто казва, че дължината на път е сбор от дължините на частите му. Третото съотношение се получава чрез събиране на неравенството $w(s - \cdots - x - y) \geq d(y)$, което следва от индуктивното предположение за второто твърдение, приложено за върха y, и неравенството $w(y - \cdots - u) \geq 0$, което е вярно, тъй като всички тегла са неотрицателни. От получената верига от равенства и еднопосочни неравенства следва неравенството $d(u) > d(y)$, което важи в началото на текущото изпълнение на тялото на цикъла **while**. Това неравенство противоречи на избора на върха u от ред № 10.

Нека v е някой чакащ връх след изпълнението на тялото на цикъла **while**. Следователно v е бил чакащ връх и преди това. От индуктивното предположение за второто твърдение следва, че преди текущото изпълнение на тялото на цикъла **while** пътят $s - \ldots - \pi\big[\pi\big[v\big]\big] - \pi\big[v\big] - v$ е бил най-къс път измежду онези пътища от s до v, които съдържат само присъединени върхове с изключение на крайния връх v (ако $\pi\big[v\big]$ е било NIL, то следва, че не е имало такива пътища). Превръщането на u от чакащ в присъединен връх разширява множеството от възможни пътища и може да намали текущия минимум. Ако u е предпоследен връх на такъв път, то $v \in \mathrm{Adj}(u)$; в такъв случай релаксацията на редовете № 13 и № 14 ще се погрижи да актуализира минимума. Ако u не е предпоследният връх на пътя, няма никакво значение дали има ребро от u към v: редовете № 13 и № 14 не могат да променят $d\big[v\big]$ и $\pi\big[v\big]$, а и най-късият път от s до v, минаващ само през присъединени върхове, остава същият, защото единственият начин да бъде скъсен е да се намери по-къс път от s до някой присъединен съсед на v, а този съсед (щом не е u), е бил присъединен още преди текущото изпълнение на тялото на цикъла **while**, следователно индуктивното предположение за първото твърдение не допуска скъсяването. В крайна сметка, щом не са се променили $d\big[v\big]$, $\pi\big[v\big]$ и най-късият път от s до v само през присъединени върхове, то второто твърдение на инварианта остава в сила при новата проверка.

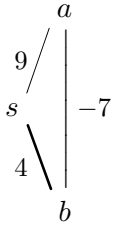
С това е доказан инвариантът на цикъла **while**. Че алгоритъмът на Дейкстра завършва, се доказва със същите полуинварианти, които използвахме при алгоритъма на Прим—Ярник. Към последната проверка за край на цикъла **while** да приложим инварианта. Щом е последна, значи няма чакащи върхове с крайни цени. Ако има чакащи върхове, те са с безкрайни цени; от второто твърдение на инварианта следва, че няма път от s до никой от тях. Другите върхове са присъединени към дървото, а според първото твърдение на инварианта дървото съдържа най-къс път от s до всеки присъединен връх, така че изходът на алгоритъма е коректен.

Забележки:

— На ред № 9 от алгоритъма на Прим—Ярник липсва проверка от алгоритъма на Дейкстра: дали $d\big[u\big] < +\infty$. Тя е нужна за алгоритъма на Дейкстра: може да няма път от s до някой връх дори ако графът е слабо свързан (понеже е ориентиран). Обаче за алгоритъма на Прим—Ярник тази проверка е излишна: в свързан неориентиран граф има път от s до всеки връх.

— Ако търсим най-къс път от s до даден връх t, можем да спрем алгоритъма на Дейкстра веднага щом присъедини t към дървото на най-късите пътища. Тогава времевите сложности важат при най-лоши входни данни — когато t е последният присъединен връх.

— От доказателството на инварианта на алгоритъма на Дейкстра се вижда колко е важно предположението, че теглата на всички ребра са неотрицателни. Ребро с отрицателно тегло застрашава не само доказателството на инварианта, но и коректността на самия алгоритъм:



Най-къси пътища в граф с отрицателни тегла на ребрата

Когато търсим най-къс път в граф, който съдържа поне едно ребро с отрицателно тегло, не можем да използваме алгоритъма на Дейкстра. Ако графът е ацикличен, задачата се решава с динамично програмиране, ала това е частен случай. Нужен е по-общ алгоритъм. Има такъв, но той е по-бавен от алгоритъма на Дейкстра. Преди да го опишем, нека разграничим две задачи, които досега бяха отъждествявани: търсене на най-къс път, който може да повтаря върхове, и търсене на най-къс прост път (тоест неповтарящ върхове). За графи с положителни тегла тези задачи съвпадат: всеки цикъл е излишен, само удължава пътя. При неотрицателни тегла двете задачи отново могат да се отъждествят: циклите с нулева дължина не удължават пътя, но и не го скъсяват, така че можем спокойно да ги премахнем; по този начин може да изгубим някои най-къси пътища (тези, които съдържат нулев цикъл), но ще останат други, прости пътища със същата дължина, а нашата цел е да намерим един най-къс път, не всички.

В предишния абзац са важни не толкова теглата на ребрата, колкото дължините на циклите. Горните разсъждения са приложими и за графи с отрицателни ребра, но без отрицателни цикли. Когато има такива цикли, двете задачи се различават съществено: ако изобщо има най-къс път (например от s до t), то има и най-къс прост път, тъй като простите пътища са краен брой; обаче най-къс път може и да няма, защото всяко ново изминаване на отрицателния цикъл поражда път с по-малка дължина, намаляваща неограничено.

Задачата за търсене на най-къс прост път е NP-пълна, следователно за нея не съществува алгоритъм с полиномиална времева сложност, ако $\mathbf{P} \neq \mathbf{NP}$. Напротив, търсенето на най-къс път (който може да повтаря върхове) е лесна задача — може да бъде решена за полиномиално време: ако графът съдържа отрицателен цикъл, то липсват най-къси пътища (поне до някои върхове); ако отрицателен цикъл не съществува, то има най-къси пътища и можем да ги намерим бързо по алгоритъма, който предстои да опишем.

Търсенето на цикъл с отрицателна дължина е важно и самò по себе си, а не е само проверка за съществуване на най-къси пътища.

Началният връх на пътищата може да бъде зададен или не. Тоест задачата за най-къс път се среща в два варианта. Те се решават по различен начин.

Алгоритъм на Белман—Форд

По даден ориентиран (или неориентиран) тегловен граф $G\big(V; E\big)$ с произволни реални, включително отрицателни, тегла на ребрата, алгоритъмът построява дърво на най-къси пътища от даден връх s до всички върхове на графа. Броят на върховете е n; броят на ребрата е m. V е множеството от върховете, E е множеството от ребрата на G, $n = |V| > 0$, $m = |E| \geq 0$. Теглото на реброто $(u\,;\,v)$ ще означаваме с $w(u\,;\,v)$ и ще тълкуваме като дължина на реброто; теглата са произволни реални числа — положителни, отрицателни или нули.

Ако графът съдържа цикли с отрицателна дължина, то алгоритъмът на Белман—Форд намира поне един такъв цикъл. В такъв случай не съществува дърво на най-къси пътища.

Алгоритъмът започва работа с предположението, че в графа няма отрицателни цикли, тоест може да се построи дърво на най-къси пътища. За произволен връх v да означим с $\delta(v)$ дължината на най-къс път от s до v; ако няма път от s до v, то $\delta(v) = +\infty$ по определение.

За върховете, до които има път от s, дължината на най-къс такъв път се изчислява така:

$$\delta(s) = 0, \quad \delta(v) = \min_u \Big\{\delta(u) + w(u\,;\,v)\Big\} \text{ за всеки връх } v \neq s,$$

където минимумът се взима по всички ребра, влизащи във v.

Алгоритъмът връща два масива — числов масив $d\big[1 \ldots n\big]$ и масив от върхове $\pi\big[1 \ldots n\big]$. При излизане от алгоритъма $d\big[v\big] = \delta\big[v\big]$ и $\pi\big[v\big] = u$, където u е онзи връх, при който се достига минимумът във втората формула за δ. Освен това $\pi\big[s\big] = \text{NIL}$ и за всеки връх $v \neq s$ са в сила следните еквивалентности: $d\big[v\big] = +\infty \iff \pi\big[v\big] = \text{NIL} \iff$ няма път от s до v.

Когато има поне един път от s до v, указателят $\pi\big[v\big] = u$ сочи към предпоследния връх u от някой най-къс път от s до v и u е родител на v в дървото на най-къси пътища с корен s.

Двете формули за δ са аналог на начално условие и рекурентно уравнение. Ако знаехме в какъв ред да изчисляваме стойностите на δ, бихме ги получили с едно обхождане на графа. Но тъй като не знаем това, обхождаме графа няколко пъти. Инициализираме $\pi\big[1\ldots n\big]$ с NIL, а $d\big[1\ldots n\big]$ с безкрайни стойности, само $d\big[s\big] = 0$ поради началното условие за функцията δ. Неравенството $\delta(v) \le \delta(u) + w(u\,;\,v)$ е в сила за всеки връх $v \ne s$ и за всяко ребро $\big(u\,;\,v\big)$; това следва от рекурентното уравнение. Ето защо, ако при обхождане на ребрата срещнем ребро $(u\,;\,v)$, за което $d(v) > d(u) + w(u\,;\,v)$, извършваме релаксация — присвояваме на $d(v)$ стойността на дясната страна на неравенството и пренасочваме указателя $\pi(v)$ към върха u. Релаксация правим и при $v = s$, но тогава тя е признак за наличие на отрицателен цикъл през s и води до прекратяване на алгоритъма.

Ако не открием отрицателен цикъл през върха s, продължаваме обхождането на ребрата. Колко пъти трябва да ги обходим, зависи съществено от подредбата им във входния списък E. Информацията тече по ребрата на графа от дадения начален връх s към останалите върхове. Ако списъкът E е подреден еднопосочно с ребрата, то правилните стойности на функцията δ ще намерим при първото обхождане, а при второто ще установим това по липсата на релаксации. Но ако списъкът E е подреден обратно на ребрата, то информацията ще тече много бавно — с една стъпка на всяко обхождане. Тъй като прост път може да има най-много $n-1$ ребра, то може да съществува връх, до който информацията да достигне чак на $(n-1)$-ото обхождане. Ето защо са задължителни поне $n-1$ обхождания, освен ако намерим отрицателен цикъл през s или констатираме липса на релаксация при някое обхождане. В последния случай правим извод, че информацията за дължините на най-късите пътища е достигнала до всички върхове.

Ако не настъпи нито едно от двете събития, можем да спрем след $(n-1)$-ото обхождане, тъй като дотогава информацията със сигурност ще е стигнала до всеки връх, достижим от s. По-точно, можем да спрем, ако знаем отнапред, че няма отрицателен цикъл, достижим от s. Иначе трябва да обходим ребрата още веднъж, за да проверим ще се получи ли релаксация: тя е признак за наличието на отрицателен цикъл през върха, при който е възникнала.

Алгоритъм на Белман—Форд $\big(G\big(V\,;\,E\big),\ s\big)$

```
 1)  π[1...n]: масив от върхове
 2)  d[1...n]: масив от реални числа — дължините на най-късите пътища от s до всеки връх
 3)  for each v ∈ V do
 4)      π[v] ← NIL   // нулев указател
 5)      d[v] ← +∞
 6)  d[s] ← 0
 7)  for k ← 1 to n do
 8)      NoChange ← true
 9)      for each (u; v) ∈ E do
10)          if d[u] + w(u; v) < d[v]
11)              d[v] ← d[u] + w(u; v)
12)              π[v] ← u
13)              if v = s or k = n
14)                  return v, π[1...n], d[1...n]
15)              NoChange ← false
16)      if NoChange
17)          return NIL, π[1...n], d[1...n]
```

Алгоритъмът връща наредена тройка:
— указател към връх;
— дърво на най-къси пътища с корен s, представено чрез указатели към родителите;
— дължините на най-късите пътища от s до другите върхове.
Първият елемент на тройката е указател към някой връх от отрицателен цикъл, достижим от s. Нулевият указател е знак, че няма отрицателен цикъл, достижим от s.

Когато графът съдържа отрицателен цикъл, достижим от s, алгоритъмът връща указател към един връх v от цикъла. Самия цикъл можем да намерим, като тръгнем от посочения връх v и следваме указателите към родителите, докато попаднем отново във v.

Когато няма отрицателен цикъл, достижим от s, можем да прочетем най-късия път от s до някой връх v, тръгвайки от v и следвайки указателите към родителите, докато стигнем в s.

И в двата случая се движим срещу посоките на ребрата, тоест правилният ред на върховете е обратен на получения.

По този начин не се откриват отрицателни цикли, недостижими от s. Това не е проблем, защото алгоритъмът бездруго е предназначен да изследва само върховете, достижими от s. Затова можем без ограничение да предполагаме, че всички върхове на графа са достижими от s (пренебрегваме недостижимите върхове). От друга страна, това изискване не е необходимо, тоест можем да подадем граф, в който някои върхове са недостижими от s, само че по този начин забавяме алгоритъма с излишни данни. Аналогично, той може да обработва несвързани графи, но за да облекчим работата му, е по-добре да отстраним върховете, недостижими от s.

Алгоритъмът работи и върху мултиграфи, тоест кратните ребра не са пречка за него. Обаче те са излишни: от всички еднопосочни ребра между една и съща двойка от върхове е достатъчно да запазим единствено най-късото; останалите само забавят алгоритъма.

Примките също не представляват пречка, но и те са излишни, затова обикновено работим с графи без примки.

Сложността по памет на алгоритъма на Белман—Форд е $\Theta(n)$ при всякакви входни данни. Тя се изразходва за масива $d\big[1 \ldots n\big]$, който е изход, но се използва не само за запис, а и за четене. Масивът $\pi\big[1 \ldots n\big]$ не се брои, защото той представлява изход и се използва само за писане. Останалите променливи (k, u, v и NoChange) са от примитивни типове, затова изразходват пренебрежимо малко памет.

Цикълът с начало на ред № 3 отнема време $\Theta(n)$ за инициализиране на двата масива.

За цикъла с начало на ред № 9 е нужно време $\Theta(n+m)$, защото ребрата на графа са дадени в n списъка на съседство — по един за всеки връх; обхождаме не само ребрата, а и върховете. При $m = \Omega(n)$ събираемото n е несъществено по порядък, затова можем да го пренебрегнем, тоест можем да запишем времевата сложност на цикъла като $\Theta(m)$. Обратно, ако $m = \mathrm{O}(n)$, събираемото n значително увеличава порядъка на времето и е добре да се освободим от него. Можем да постигнем това, като в началото на алгоритъма обходим графа еднократно и съставим списък E от всички ребра (така че да има един списък, а не n списъка). За това обхождане е нужно време $\Theta(n+m) = \Theta(n)$, което може да се причисли към времето за инициализация, и памет $\Theta(m) = \mathrm{O}(n)$ за новия списък E. Допълнителното време и допълнителната памет не увеличават досегашната сложност по порядък, а само константните множители отпред. За сметка на това времевата сложност на вътрешния цикъл спада до порядък $\Theta(m)$.

Виждаме, че във всички случаи цикълът с начало на ред № 9 може да бъде реализиран с времева сложност $\Theta(m)$. Изпълнява се n пъти, защото е вложен в цикъла с начало на ред № 7, следователно времевата сложност на целия алгоритъм е $\Theta(n) + \Theta(nm) = \Theta(nm)$.

Като обобщим резултатите от извършения дотук анализ, стигаме до следното заключение: алгоритъмът на Белман—Форд притежава сложност по памет $\Theta(n)$ при всякакви входни данни и сложност по време $\Theta(nm)$ при най-лоши входни данни — например когато графът е ацикличен и съдържа хамилтонов път, който има за начало s и е най-къс сред всички пътища с начало s, а подредбата на ребрата в списъците от входните данни е обратна на посоките на самите ребра (тоест обратна на посоката на споменатия хамилтонов път).

Алгоритъм на Флойд—Уоршал

Понякога възниква необходимост да намерим най-къси пътища от всеки до всеки връх на даден ориентиран (или неориентиран) граф с n върха. Ако се случи графът да е нетегловен или тегловен ацикличен, или с неотрицателни тегла, пускаме n пъти (по веднъж от всеки връх) съответния метод: търсене в ширина, динамично програмиране или алгоритъма на Дейкстра. Но когато са допустими отрицателни тегла на ребрата, тези алгоритми не могат да се приложат, а n-кратното пускане на алгоритъма на Белман—Форд не е най-бързият начин.

Алгоритъмът на Флойд—Уоршал намира най-къси пътища от всеки връх до всеки връх на даден ориентиран (или неориентиран) тегловен граф с n върха. Теглата на ребрата на графа могат да бъдат произволни реални числа — положителни, отрицателни или нули. Тази задача има решение тогава и само тогава, когато графът не съдържа цикъл с отрицателна дължина. В противен случай алгоритъмът намира един такъв цикъл.

Да номерираме върховете на графа с целите числа от 1 до n вкл. Разглеждаме редица от $n+1$ числови матрици: $D^{(0)},\ D^{(1)},\ D^{(2)},\ \ldots\ ,\ D^{(n)}$. Всяка от тях има n реда и n стълба. Числата, поставени в скоби, са горни индекси, а не степени. Ще бележим с $d_{ij}^{(k)}$ онзи елемент, който стои в ред № i и стълб № j на матрицата $D^{(k)}$. По определение приемаме, че числото $d_{ij}^{(k)}$ е дължината на най-къс път от върха i до върха j сред пътищата, чиито междинни върхове (тоест всички върхове без краищата) имат номера, ненадвишаващи k. Ако освен дължината е нужен и самият път, правим още една редица от $n+1$ матрици: $P^{(0)},\ P^{(1)},\ P^{(2)},\ \ldots\ ,\ P^{(n)}$, всяка от които има n реда и n стълба; елементът $\pi_{ij}^{(k)}$ от ред № i и стълб № j на матрицата $P^{(k)}$ е указател към предпоследния връх на споменатия най-къс път.

Матриците се пресмятат в нарастващ ред на индекса k.

Инициализация: при $k=0$ не може да има междинни върхове, т.е. разглеждаме пътища, съставени от едно ребро. Затова $d_{ij}^{(0)}$ е дължината (теглото) на реброто от връх i към връх j и $\pi_{ij}^{(0)}=i$, ако има такова ребро; в противен случай $d_{ij}^{(0)}=+\infty$ и $\pi_{ij}^{(0)}=\mathrm{NIL}$.

За всяко цяло число k от 1 до n включително използваме следните рекурентни формули:
— ако $i=k$ или $j=k$, то $d_{ij}^{(k)}=d_{ij}^{(k-1)}$ и $\pi_{ij}^{(k)}=\pi_{ij}^{(k-1)}$;
— ако $i\neq k$ и $j\neq k$, то $d_{ij}^{(k)}=\min\left\{d_{ij}^{(k-1)},\ d_{ik}^{(k-1)}+d_{kj}^{(k-1)}\right\}$, а $\pi_{ij}^{(k)}=\pi_{ij}^{(k-1)}$ или $\pi_{ij}^{(k)}=\pi_{kj}^{(k-1)}$ според това, кой от двата елемента е по-малък — първият или вторият съответно. Тези формули се получават от следните две възможности: връх № k съответно да не участва или да участва в най-къс път от i до j. Във втория случай пътят се състои от две части: от i до k и от k до j. Ако сравняваните дължини са равни, няма значение указателя към кой предходник ще изберем: и двата пътя са най-къси. Единственото изключение е, когато двете дължини са безкрайни. Тогава породеният подграф не съдържа път от i до j, затова взимаме $\pi_{ij}^{(k)}=\pi_{ij}^{(k-1)}=\mathrm{NIL}$.

Ако върховете от № 1 до № $k-1$ вкл. пораждат подграф, съдържащ отрицателен цикъл, то рекурентните формули не важат за съответното k. Ето защо алгоритъмът спира работа, ако открие такъв цикъл. Признак за съществуването на отрицателен цикъл е получаването на отрицателна стойност където и да било върху главния диагонал на някоя матрица $D^{(k)}$: ако $d_{ii}^{(k)}<0$, то съществува отрицателен цикъл през връх № i. Можем да открием самия цикъл, като тръгнем от връх № i и следваме указателите към предходните върхове, записани в $P^{(k)}$: $i,\ j=\pi_{ii}^{(k)},\ h=\pi_{ij}^{(k)}$ и тъй нататък, докато стигнем отново до връх № i. Така се придвижваме срещу посоките на ребрата, тоест получаваме върховете на цикъла в обратен ред.

Когато в графа няма отрицателен цикъл, алгоритъмът извършва пресмятанията докрай. Понеже никой номер на връх не надвишава n, то матриците $D^{(n)}$ и $P^{(n)}$ съдържат информация за най-късите пътища, съставени от всякакви върхове без ограничение. Тоест тези две матрици са резултатът от работата на алгоритъма: $d_{ij}^{(n)}$ е дължината на най-къс път от връх i до връх j. Самия път ще намерим, ако тръгнем от върха j и следваме указателите към предходните върхове: $j,\ f=\pi_{ij}^{(n)},\ g=\pi_{if}^{(n)},\ h=\pi_{ig}^{(n)}$ и т.н. до връх № i. Върховете се получават в обратен ред.

Няма път от i до j $\iff$ $d_{ij}^{(n)}=+\infty$ $\iff$ $\pi_{ij}^{(n)}=\mathrm{NIL}$.

Разсъжденията от последните два абзаца важат и при $i=j$. Числото $d_{ii}^{(n)}$ е дължината на най-къс цикъл през връх № i. Цикълът се открива чрез указателите към предходниците. Ако $d_{ii}^{(n)}=+\infty$, то върхът i не участва в цикъл.

От рекурентното уравнение се вижда, че новите стойности на елементите на матриците могат да се записват върху старите. Затова са достатъчни една матрица D и една матрица P, и двете от тип $n\times n$. Следователно сложността по памет е $\Theta\left(n^2\right)$ при всякакви входни данни. Попълваме тези матрици $n+1$ пъти (за k от 0 до n вкл.), затова времевата сложност е $\Theta\left(n^3\right)$ при най-лоши входни данни — когато няма отрицателен цикъл.