1. Дефиниция на минимално покриващо дърво (МПД) на свързан граф с тегла на ребрата.

Определение 80: Тегло на ПД. Минимално ПД.

Нека G е свързан тегловен граф с тегловна функция w. Нека T е ПД на G. *Теглото на* T е $w(T) = \sum_{e \in E(T)} w(e)$. Минимално ПД, съкратено МПД, е такова ПД T', че за всяко ПД T'' е вярно, че $w(T') \leqslant w(T'')$.

2. Формулировка и доказателство на теоремата за съгласуваното множество (условия за нарастване на подмножество на МПД).

Определение 82: Срез в граф.

Срез в граф G е всяко разбиване на $V(G)$ на два дяла. Нека $S = \{V_1, V_2\}$ е срез в G. Ребро $e = (u, v)$ *прекосява* S, ако $u \in V_1$ и $v \in V_2$. Нека $E' \subseteq E(G)$. S е *съобразен с* E', ако нито едно ребро от E' не го прекосява. Ако G е тегловен и реброто e прекосява S, казваме, че реброто e е *леко*, ако e има минимално тегло измежду всички ребра, които прекосяват S.

Теорема 58: Theorem 23.1 от [31, стр. 627] (МПД теоремата)

Нека $G = (V, E)$ е свързан тегловен граф с тегловна функция $w : E \to \mathbb{R}$. Нека $A \subseteq E$ е такова, че съществува МПД T, че $A \subseteq E(T)$. Нека $S = \{V_1, V_2\}$ е срез в G, който е съобразен с A. Нека $e = (u, v)$ е произволно леко ребро, прекосяващо S. Тогава e е сигурно за A.

Доказателство: Ако $e \in E(T)$, то ние сме готови с доказателството. Да допуснем, че $e \notin E(T)$. Ще покажем, че има МПД T', такова че $A \cup \{e\} \subseteq E(T')$.

Добавяме e към T и получаваме уницикличен граф U. Нещо повече, със сигурност e е ребро от цикъла c на U. За реброто e знаем, че прекосява среза. Със сигурност съществува ребро e' в c, различно от e, което прекосява среза. Изтриваме e' от U и получаваме ПД T', различно от T. Да сравним $w(T)$ с $w(T')$. T' се получи от T с добавяне на e и изтриване на e'. Тогава

$$w(T') = w(T) + w(e) - w(e')$$

Но e е леко по конструкция, така че $w(e) \leqslant w(e')$, следователно $w(e) - w(e') \leqslant 0$. Тогава $w(T') \leqslant w(T)$. Но T е МПД, така че $w(T) \leqslant w(T')$ по дефиниция. Тогава $w(T) = w(T')$ и T' също е МПД. Което съдържа както ребрата от A, така и e. □

МПД теоремата????

3. Алгоритми на Прим и Крускал, имплементации и оценка на сложността.

- Прим

PRIM GENERIC($G = (V, E)$: неор. свързан граф, w: тегл. ф-я върху G, s: стартов връх)
1 $V(T) \leftarrow \{s\}$
2 $A \leftarrow \varnothing$
3 **while** $V \backslash V(T) \neq \varnothing$ **do**
4 намери леко ребро $e = (x, y)$, прекосяващо среза $\{V(T), V \backslash V(T)\}$
5 нека $x \in V(T)$
6 $V(T) \leftarrow V(T) \cup \{y\}$
7 $A \leftarrow A \cup \{e\}$
8 **return** A

Известно подобрение би било на всяка итерация да не преглеждаме всички ребра на графа, а само тези, които прекосяват среза $\{V(T), V \backslash V(T)\}$. В асимптотичния смисъл обаче това не е никакво подобрение. В най-лошия случай, общият брой на прегледани ребра (за цялото изпълнение на алгоритъма) се оценява със сумата

$$\sum_{k=1}^{n-1} k(n-k) = \Theta(n^3)$$

тъй като има ребро между всеки връх от единия дял и всеки връх от другия дял, на всяка итерация (какъв граф трябва да е G, за да изпълнено това?). И така, отново имаме, в най-лошия случай, кубичен алгоритъм. Трябва да направим нещо по-умно, за да избираме ефикасно леко прекосяващо ребро.

- върховете от $V(T)$ са *дървесните върхове* (на английски, *tree vertices*),
- върховете от $N(V(T))$ са *граничните върхове* (на английски, *fringe vertices*),
- върховете от $V \backslash N[V(T)]$ са *неизвестните върхове* (на английски, *unseen vertices*).

МПД PRIM1($G = (V, E)$: неор. свързан граф, w: тегл. ф-я върху G, s: стартов връх)
1 $V(T) \leftarrow \{s\}$
2 $A \leftarrow \varnothing$
3 **foreach** $u \in V$
4 status$[u] \leftarrow$ неизвестен

```
5   status[s] ← дървесен
6   foreach y ∈ adj[s]
7       status[y] ← граничен
8   x ← s
9   while V\V(T) ≠ ∅ do
10      (* x е последният сложен в дървото връх *)
11      foreach y ∈ adj[x]
12          if status[y] = граничен
13              ако w((x, y)) е по-малко от граничното тегло на y,
                    смени кандидат-реброто на y с (x, y)
14          if status[y] = неизвестен
15              status[y] ← граничен
16              кандидат-реброто на y става (x, y)
17      намери кандидат-ребро e с минимално тегло
18      x ← граничният връх на e
19      status[x] ← дървесен
20      V(T) ← V(T) ∪ {x}
21      A ← A ∪ {e}
22  return A
```

Коректност и сложност. Подробно доказателство за коректност няма да правим. Инвариантът е, при всяко достигане на управлението на ред 9, ребрата от A задават МПД на подграфа на G, индуциран от $V(T)$. Използвайки Теорема 58, лесно показваме, че реброто e на ред 17 е сигурно за A.

Сложността по време в най-лошия случай е $\Theta(n^2)$.

- Първо, всички намирания на кандидат-ребра, по време на цялата работа на алгоритъма, става във време $\Theta(n^2)$. Ето защо. Най-лошият случай е графът да е пълен. Тогава unseen върхове изобщо няма, понеже всеки връх е съседен на всеки друг, така че на всяка итерация на **while**-а (ред 9) върховете на графа се разбиват на дървесни и гранични. Тогава на всяка итерация на **while**-а се разглеждат $|V \setminus V(T)|$ (гранични) върхове, за да се намери кандидат-ребро с минимално тегло на ред 17. Тогава следната сума дава асимптотиката на общата (по цялото изпълнение на алгоритъма) работа за намирането на кандидат-ребрата:

  $$\Theta(n-1) + \Theta(n-2) + \cdots + \Theta(2) + \Theta(1)$$

  Тривиално се пресмята, че $\Theta(n-1) + \Theta(n-2) + \cdots + \Theta(1) = \Theta(n^2)$.

- Освен намирането на кандидат-ребрата, алгоритъмът обхожда списъците на съседство чрез **for**-цикъла на редове 11–16. Това става във време $\Theta(n+m)$, за цялата работа на алгоритъма.

Тъй като $\Theta(m+n) = O(n^2)$, имаме обща асимптотична оценка за сложността $\Theta(n^2)$. Забележе, че по отношение на най-лошия случай, тази сложност по време е оптимална, защото в най-лошия случай списъците на съседство имат размер $\Theta(n^2)$.

```
МПД PRIM2(G = (V, E): неор. свързан граф, w: тегл. ф-я върху G, s: стартов връх)
 1  foreach u ∈ V
 2      u.key ← ∞
 3      π[u] ← Nil
 4  s.key ← 0
 5  създай празна приоритетна min опашка Q
 6  Q ← V
 7  while not isempty(Q) do
 8      x ← EXTRACT-MIN(Q)
 9      foreach y ∈ adj[x]
10          if y ∈ Q and w((x, y)) < y.key
11              y.key ← w((x, y))
12              π[y] ← x
13  return π[1 .. n]
```

Този псевдокод следва плътно псевдокода от [31, стр. 634]. Две забележки по имплементацията. Първо, проверката $y \in Q$ на ред 10 може да бъде имплементирана, като всеки връх има булев флаг за принадлежност към опашката; в началото всички флагове са TRUE, а при всяко вадене на връх от опашката на ред 8, правим флага му FALSE. Второ, смяната на ключа на y на ред 11 не е просто смяна на едно число с друго, както може човек да си помисли първо. Връх y е в опашката и смяната на ключа му, тоест, намаляването на ключа му, трябва да бъде съпроводено с изпълнение на DECREASE-KEY, която е огледален аналог на INCREASE-KEY от Подсекция 5.4.4. Това има значение за анализа на сложността.

- **Крускал**

Алгоритъмът на Kruskal е ефикасен алгоритъм за намиране на МПД, който е основан на съвсем различна идея от тази на алгоритъма на Prim. Най-общо казано, първо сортираме ребрата по тегло, и после **в ненамаляващия ред** на теглата слагаме първите $n - 1$ ребра, за всяко от които, в този ред, е вярно, че не образува цикъл с вече сложените ребра. Ако обаче имплементираме тази идея буквално, ще получим кубичен алгоритъм.

- Сортирането на ребрата става във, и иска, време $\Theta(m \lg m)$, което е същото като $\Theta(n^2 \lg n)$ ако $m \asymp n^2$.

- В най-лошия случай се налага да проверим всички m ребра. За да се убедим в това, нека най-тежкото ребро е мост. Всеки мост задължително участва във всяко МПД, защото всеки мост участва във всяко ПД. Ерго, в най-лошия случай може да се наложи да стигнем до края на сортираната редица от ребра.

  За всяко ребро в сортирана редица тестваме дали образува цикъл с вече сложените ребра. Тоест, за всяко ребро тестваме дали подграфът, индуциран от вече сложените ребра плюс него е цикличен. Ако тестът се направи с BFS или DFS, той става във време $\Theta(n)$ в най-лошия случай[†]. Така че тези тестове ще станат във време $O(nm)$, като може да се покаже, че границата е точна, тоест, $\Theta(nm)$.

При $m \asymp n^2$, това е $\Theta(n^2 \lg n) + \Theta(n^3) = \Theta(n^3)$.

Как да подобрим сложността по време? Сортирането на ребрата отнема $\Omega(m \lg n)$ време и това е неизбежно заради долната граница на сортирането (Подсекция 13.2.2)[‡]. Ерго, имаме долна граница $\Omega(m \lg n)$ за алгоритъма на Kruskal, независимо от това колко ефикасно имплементираме слагането на ребрата.

Слагането на ребрата става итеративно, като в най-лошия случай се изпълняват m итерации (ако най-тежкото ребро е мост). Имайки предвид това, целта ни е тестването за едно ребро да става във време $O(\lg n)$. Ако успеем да постигнем това, ще имаме $\Theta(m \lg n)$ имплементация на алгоритъма на Kruskal; дори да постигнем тестване за едно ребро във време $o(\lg n)$, пак ще имаме $\Theta(m \lg n)$ имплементация на алгоритъма на Kruskal заради сортирането в началото.

Ключовото наблюдение е, че ако имаме гора с повече от едно дърво и добавим ново ребро между два върха:

- ако те са от различни дървета на гората, няма да се образува цикъл, но тези две дървета плюс новото ребро ще станат едно дърво, а графът ще продължи да е гора;

- ако те са от едно и също дърво, ще се образува цикъл, а графът ще престане да е гора.

МПД KRUSKAL($G = (\{1, \ldots, n\}, E)$: неориентиран свързан граф, w: тегловна ф-я върху G)

```
1   A ← ∅
2   направи разбиването {{1}, {2}, ..., {n}}
3   сортирай E по тегла
4   count ← 0
5   while count < n − 1 do
6       нека e = (u, v) е следващото ребро в сортираната редица
7       if component(u) ≠ component(v)
8           A ← A ∪ {e}
9           identify(component(u), component(v))
10          count++
11  return A
```

Сложност по време. За да бъде ефикасен МПД KRUSKAL, трябва проверката дали $\boxed{\text{component}(u) \neq \text{component}(v)}$ (ред 7) и операцията $\boxed{\text{identify}(\text{component}(u), \text{component}(v))}$ (ред 9) да се вършат във време $O(\lg n)$. Ако успеем да постигнем това, **while**-цикълът (редове 5–10) ще се изпълнява във време $O(m \lg n)$ и алгоритъмът ще има сложност по време $\Theta(m \lg n)$ заради сортирането (ред 3).

Всяка компонента, или дял, на разбиването трябва да има идентификатор; тоест, свое име. Нека са дефинирани две примитивни функции: Find и Union. Ако u е елемент от опорното множество (в случая, върх на графа), Find(u) връща името на дяла на разбиването, в който е u. Ако i и j са имената на два различни дяла, Union(i, j) слива тези два дяла в един и му дава име i. Ако разполагаме с такива примитиви, можем да реализираме $\boxed{\text{component}(u) \neq \text{component}(v)}$ така:

component(u) ≠ component(v)

```
1   i ← Find(u)
2   j ← Find(v)
3   if i ≠ j
4       return TRUE
5   else
6       return FALSE
```

и да реализираме $\boxed{\text{identify}(\text{component}(u), \text{component}(v))}$ така:

identify(component(u), component(v))

```
1   i ← Find(u)
2   j ← Find(v)
3   Union(i, j)
```

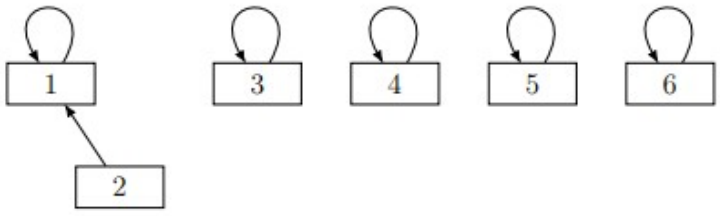
Тези лидери са важни. Функцията Find(u) връща лидера на компонентата, в която се намира u, а функцията Union(i, j) работи само върху лидери на компоненти.

Ето псевдокодът на Find. $\pi[u]$ е родителят на u.

```
Find(u)
1  if u ≠ π[u]
2      return Find(π[u])
3  else
4      return u
```

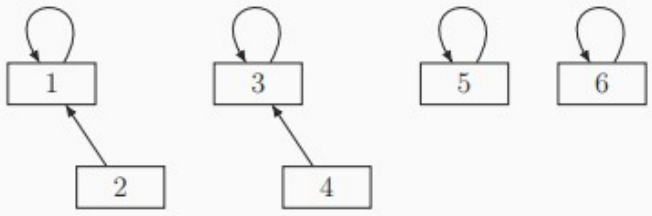
Union by rank. Винаги, когато правим Union, "наказваме" върховете на едно от дърветата с увеличаване на дълбочината им с единица. Ако извършваме Union с просто пренасочване на указателя на единия корен към другия корен, това е неизбежно. Но кое от двете дървета да накажем?

Логично е да накажем дървото с по-малко върхове. В това се състои тази евристика: всяка от компонентите съдържа и информация за броя на елементите в себе си, която е *рангът* на дървото. При Union се променя указателя на корена на дървото с по-малък ранг (ако са с еднакви рангове, решението се взема произволно).



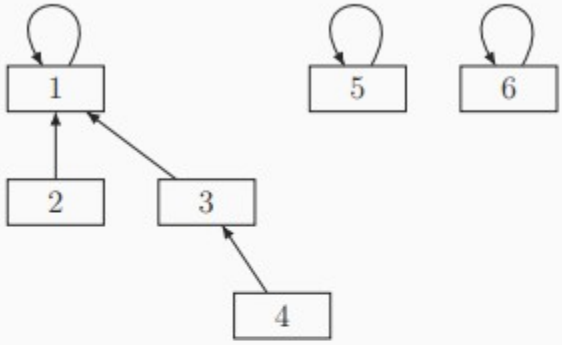
Да кажем, че следващото сливане е Union(3, 4). Резултатът е показан на Фигура 10.2.

Фигура 10.3 : След Union(1, 2) и Union(3, 4).

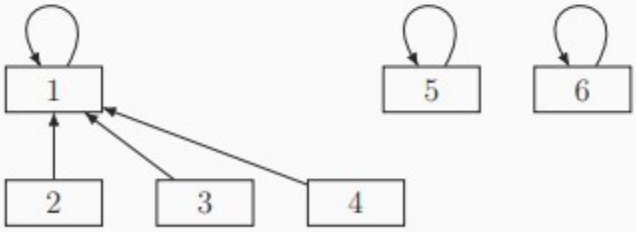


Нека следващото сливане е Union(1, 3). Викането е коректно, защото и 1, и 3 са лидери[↑]. Както се вижда на Фигура 10.3, има два начина да направим дървото–резултат от сливането.

Фигура 10.3 : Два варианта за Union$(1,3)$ върху гората от Фигура 10.2.



Вариант 1.



Вариант 2.

Теорема 61: Максимална височина на дърво след Union by rank

Започвайки от дървета с по един връх (разбиване на едноелементни множества), след произволна серия от изпълнения на Union by rank, за всяко дърво в колекцията от дървета е вярно, че $\mathtt{h} = \mathsf{O}(\lg \mathtt{n})$, където $\mathtt{h}$ е височината на дървото, а $\mathtt{n}$ е броят на **неговите** върхове.

4. Задачи за най-къс път в граф с тегла на ребрата.

Определение 83: Тегло на път

Теглото на пътя p е $w(p) = \sum_{e \in E(p)} w(e)$.

Определение 84: Отрицателен цикъл

Нека G е тегловен граф. Ако G е ориентиран, *отрицателен цикъл* е всеки прост цикъл в G, който има отрицателна сума от теглата на ребрата. Ако G е неориентиран, отрицателен цикъл е всеки прост цикъл в G, който има отрицателна сума от теглата на ребрата, както и всяко ребро с отрицателно тегло.

Наблюдение 53: Проблем при отрицателните цикли.

Ако теглата са само положителни, "най-къс път от u до v" е добре дефинирано понятие. Обаче за всеки отрицателен цикъл c, за всеки път p, който съдържа c, съществува път p', такъв че $w(p') < w(p)$, като p' "минава през" c повече пъти от p. Ерго, ако поне един път от $\text{Paths}(u, v)$ съдържа поне един връх от отрицателен цикъл, не можем да дефинираме "най-къс път от u до v".
Ако няма отрицателни цикли, този проблем не съществува дори при наличие на отрицателни тегла.

Наблюдение 54: Когато няма отрицателни цикли.

При липса на отрицателни цикли, за всеки два върха u и v, всеки най-къс път от u до v задължително е прост път.

Определение 85: Теглото на най-къс път

Теглото на най-къс път от u *до* v е

$$\delta(u, v) = \begin{cases} \infty, \text{ ако } \text{Paths}(u, v) = \varnothing \\ -\infty, \text{ ако } \text{Paths}(u, v) \neq \varnothing \text{ и } \exists p \in \text{Paths}(u, v) : p \text{ съдържа отрицателен цикъл} \\ \min\{w(p) \mid p \in \text{Paths}(u, v)\}, \text{ ако } \text{Paths}(u, v) \neq \varnothing \text{ и } \neg\exists p \in \text{Paths}(u, v) : p \text{ съд. отр. цик.} \end{cases}$$

Известни са следните варианти на задачата за най-къс път в тегловен граф.

Изч. Задача 24: Single Pair Shortest Path

екземпляр: Ориентиран граф $G = (V, E)$, тегловна функция w, начален връх s, краен връх t.
решение: $\delta(s, t)$

Изч. Задача 25: Single Source Shortest Paths

екземпляр: Ориентиран граф $G = (V, E)$, тегловна функция w, начален връх s.
решение: $\forall v \in V : \delta(s, v)$.

Изч. Задача 26: Single Destination Shortest Paths

екземпляр: Ориентиран граф $G = (V, E)$, тегловна функция w, краен връх t.
решение: $\forall v \in V : \delta(v, t)$.

Изч. Задача 27: All Pairs Shortest Paths

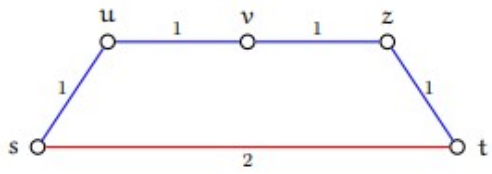
екземпляр: Ориентиран граф $G = (V, E)$, тегловна функция w.
решение: $\forall u, v \in V : \delta(u, v)$.

5. Дърво на най-късите пътища.

Ако искаме да изчислим най-къс път $s \rightsquigarrow t$, паметта, която е необходима за записването на пътя е $\Theta(n)$ в най-лошия случай, защото, в най-лошия случай, дължината на пътя е $\Theta(n)$ и не можем да избегнем записването на всеки връх от него. На пръв поглед, ако искаме да запишем по един най-къс път $s \rightsquigarrow v$ за всеки $v \in V$, ще ни трябва общо $O(n^2)$ памет, като може да се покаже, че асимптотичната горна граница $O(n^2)$ е точна; ерго, имаме право да кажем, че ще ни трябва $\Theta(n^2)$ памет.

Но това е само на пръв поглед. Всъщност, можем "да минем" само с $\Theta(n)$ памет за всички пътища, защото можем да ги представим с ориентирано кореново дърво-арборесценция, която може да се представи с масив на предшествията $\pi[1 \mathinner{..} n]$, също както при BFS и DFS. Ще покажем, че най-къси пътища от s могат да се представят с една арборесценция с корен s. Нека p и q са най-къси пътища от s съответно до u и v, където u и v са различни върхове. Ако единственият общ връх на p и q е s, няма какво да се показва. Нека p и q имат поне един общ връх освен s. Нека z е най-отдаленият в p и в q връх от s, който е общ за p и q. Очевидно не може $z = u = v$, понеже u и v са различни. Ако $z = u \neq v$ или $z = v \neq u$, няма какво повече да показваме; това са случаите, в който съответно p е част от q или q е част от p. Остава да разгледаме случая, в който $z \neq u$ и $z \neq v$. Нека подпътят на p от s до z е p', а подпътят на q от s до z е q':

Задачата за най-късите пътища и задачата за МПД са различни. Естествено, те са задачи върху различни видове графи, като МПД е върху неориентирани тегловни графи, а най-късите пътища са върху ориентирани тегловни графи, но съществената разлика не е в това. Дори да решаваме задачата за най-късите пътища върху неориентирани тегловни графи, тя остава принципно различна от задачата за МПД. Ето малък пример:



Най-късият път между s и t е реброто (s, t) с тегло 2 (в червено). Но МПД-то (то е само едно) се състои от четирите сини ребра, всяко с тегло 1. Ако си представим работата на алгоритъма на Kruskal върху този граф, сортирайки ребрата, той ще постави в началото четирите ребра с тегло 1 и след това реброто с тегло 2. Когато започне да слага ребра, той ще сложи четирите ребра с тегло 1 и ще спре, без да достигне до реброто с тегло 2, което реализира най-късия път.

Наблюдение 55

Задачите за намиране на МПД и намиране на дърво на най-късите пътища върху неориентирани тегловни графи са принципно различни. Всяко МПД, за всеки два върха s и t, дава (уникален) път p между s и t, но в общия случай p не е най-къс път в графа между s и t.

6. Алгоритъм за намиране на дърво на най-къси пътища в граф с константни тегла по ребрата и алгоритъм на Дейкстра.

Ще разгледаме един от най-популярните алгоритми за намиране на най-къс път в ориентиран граф: алгоритъмът на Dijkstra. Нека тегловната функция е $w : E \to \mathbb{R}^+$. Изискването да няма отрицателни тегла е съществено! Алгоритъмът на Dijkstra **не работи коректно**, ако има отрицателни тегла, дори да няма отрицателни цикли. Намерете сами малък пример за тегловен граф с точно едно ребро с отрицателно тегло, в който няма отрицателни цикли и въпреки това който алгоритъмът на Dijkstra не работи коректно.

Ще разгледаме два варианта на алгоритъма на Dijkstra: базов и изтънчен. Базовият вариант намира следващия връх x с последователно търсене, а изтънченият вариант ползва АТД приоритетна опашка, от която бързо изважда следващия връх x. Забележете аналогията с алгоритъма на Prim, който също беше имплементиран в два варианта—МПД PRIM1 на стр. 426 и МПД PRIM2 на стр. 428—с подобна разлика между тях.

```
SSSHP DIJKSTRA1(G: ориентиран тегловен граф, w: тегл. ф-я върху G, s: стартов връх)
 1  ISS(G, s)
 2  S ← ∅
 3  while ∃u ∈ V\S : u.d < ∞ do
 4      избери x ∈ V\S, такъв че x.d е минимална
 5      S ← S ∪ {x}
 6      foreach y ∈ adj[x]
 7          RELAX((x, y))
```

```
SSSHP DIJKSTRA2(G: ориентиран тегловен граф, w: тегл. ф-я върху G, s: стартов връх)
 1  ISS(G, s)
 2  S ← ∅
 3  създай празна приоритетна min опашка Q
 4  Q ← V
 5  while not isempty(Q) do
 6      x ← EXTRACT-MIN(Q)
 7      S ← S ∪ {x}
 8      foreach y ∈ adj[x]
 9          RELAX((x, y))
```

Нещо друго?

7. Оценка на сложността.

Сложността по време е същата, в асимптотичния смисъл, като на съответните имплементации на Prim. Базовата имплементация има сложност $\Theta(n^2)$ в най-лошия случай, а изтънчената има сложност $\Theta((n + m) \lg n)$ в най-лошия случай, ако Q е имплементирана с двоична пирамида.

Да разгледаме псевдокодовете на изтънчените варианти на алгоритмите на Prim и Dijkstra един до друг. Има следните разлики между показаните тук псевдокодове и тези, които вече видяхме съответно на стр. 428 и на предната страница.

- За удобство тук описваме Dijkstra подборно, без ISS и RELAX.
- Множеството S не се ползва истински от алгоритъма на Dijkstra в изтънчения вариант. То присъства в псевдокода (редове 2 и 7) за удобство в доказателството за коректност. Тук няма S.
- На ред 10 има проверка дали $y \in Q$. Това не е съществено за коректността, въпреки че на практика има смисъл. Върховете извън Q са точно върховете от досега изграденото дърво. Нас ни интересуват само ключовете на върховете от Q, понеже това са върховете извън дървото и алгоритъмът "решава" кой от тях да добави към дървото въз основа на ключовете им. Ако $y \notin Q$, няма смисъл да променяме ключа на y, защото този връх е вече в дървото и неговият ключ няма да се използва повече за променяне (намаляване) на ключове на върхове извън дървото.

  Поради това тук на ред 10 не правим проверка дали $y \in Q$.

```
МПД PRIM2(G, w, s)
 1  foreach v ∈ V(G)
 2      v.key ← ∞
 3      π[v] ← Nil
 4  s.key ← 0
 5  празна приор. min опашка Q
 6  Q ← V
 7  while not isempty(Q) do
 8      x ← EXTRACT-MIN(Q)
 9      foreach y ∈ adj[x]
10          if y.key > w((x,y))
11              y.key ← w((x,y))
12              π[y] ← x
```

```
SSSHP DIJKSTRA2(G, w, s)
 1  foreach v ∈ V(G)
 2      v.d ← ∞
 3      v.π ← Nil
 4  s.d ← 0
 5  празна приор. min опашка Q
 6  Q ← V
 7  while not isempty(Q) do
 8      x ← EXTRACT-MIN(Q)
 9      foreach y ∈ adj[x]
10          if y.d > x.d+w((x,y))
11              y.d ← x.d+w((x,y))
12              y.π ← x
```

Ясно се вижда, че единствената формална разлика между двата псевдокода е числото, с което се сравнява ключът на y на ред 10. В алгоритъма на Prim това число е теглото на реброто (x, y). В алгоритъма на Dijkstra това число е **сумата** от теглото на (x, y) и ключа на x. Ерго, алгоритъмът на Prim взема своите решения въз основа на предвалително фиксирани числа – теглата на ребрата не се менят по време на работата на алгоритъма. Докато алгоритъмът на Dijkstra взема решенията въз основа на числа, които (може да) се менят (намаляват) от итерация на итерация.

8. Алгоритъм на Флойд за намиране на всички двойки най-кратки пътища.

$$D^{(0)}[i,j] = \begin{cases} 0, & \text{ако } i = j \\ \infty, & \text{ако } i \neq j \end{cases} \tag{12.2}$$

$$D^{(t)}[i,j] = \min\{D^{(t-1)}[i,k] + W[k,j] \mid 1 \leqslant k \leqslant n\}, \text{ за } t > 0 \tag{12.3}$$

Тогава можем да изчисляваме рекурсивно така:

$$D^{(0)} \leftarrow W \tag{12.4}$$

$$D^{(k)}[i,j] \leftarrow \min\{\underbrace{D^{(k-1)}[i,j]}_{\text{k не помага}}, \underbrace{D^{(k-1)}[i,k] + D^{(k-1)}[k,j]}_{\text{k помага}}\}, \text{ за } k > 0 \tag{12.5}$$

Забележете алгоритмичното предимство на (12.4) и (12.5) пред (12.2) и (12.3)! В (12.4) и (12.5) се пресмята минимум на **2** числа, докато в (12.2) и (12.3) се пресмята минимум на n числа.

Естествено, ако имплементираме (12.4) и (12.5) директно, ще получим алгоритъм с експоненциална сложност по време, защото при разклоненост 2 и височина $\Theta(n)$ на дървото на рекурсията сложността по време е $\Omega(2^n)$. Но, също както при предишния алгоритъм, ние няма да имплементираме (12.4) и (12.5) директно, а ще съобразим, че всъщност има само $\Theta(n^3)$ различни стойности, които се налага да изчислим, които са разпределени в матрици $D^{(0)}$, $D^{(1)}$, ..., $D^{(n)}$, всяка от които е $n \times n$ и се пресмята от предишната. Тези матрици ще запълваме отдолу-нагоре, точно както диктува схемата **Динамично Програмиране**.

Всичко това води до следния кубичен алгоритъм за задачата APShP, който е публикуван от Robert Floyd [45], но се базира на теорема на Stephen Warshall [143].

FLOYD-WARSHALL(W: матрица $n \times n$, представяща тегловен ориентиран граф)

```
1  D^(0) ← W
2  for k ← 1 to n
3      for i ← 1 to n
4          for j ← 1 to n
5              D^(k)[i,j] ← min(D^(k-1)[i,j], D^(k-1)[i,k] + D^(k-1)[k,j])
6  return D^(n)
```

Коректност и сложност по време и памет. Коректността на FLOYD-WARSHALL е очевидна, ако читателят е убеден в коректността на (12.4) и (12.5). Сложността му по време е $\Theta(n^3)$. Сложността му по памет, ако бъде реализиран по точно този начин, е $\Theta(n^3)$, но може лесно да бъде подобрена до $\Theta(n^2)$, избягвайки създаването на $n+1$ матрици и използвайки само две матрици, една, от която четем стойности (тя отговаря на $D^{(k-1)}$) и една, в която пишем (тя отговаря на $D^{(k)}$).

И така, ако в началото на алгоритъма не копираме W в D, а работим **директно върху** W, ще реализираме алгоритъм с **константна сложност по памет**, иначе казано, in-place версия на **Floyd-Warshall**. Да си припомним (Подсекция 2.2.4), че при изследването на сложността по памет отчитаме само допълнителната памет, а входа игнорираме. Така че лесно може да направим версия на **Floyd-Warshall** със сложност по памет $\Theta(1)$, при условие, че сме готови да загубим входа, който ще се окаже презаписан.