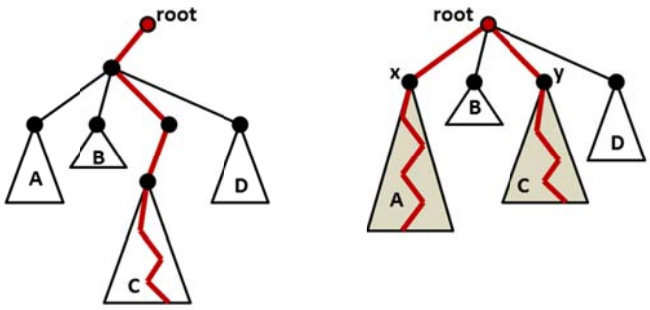
Průměr minimální kostry úplného grafu

Předpokládejme, že daný graf má N uzlů.
Budeme postupovat odzadu a nejprve budeme diskutovat, jak nalézt průměr minimální kostry, to jest průměr libovolného kořenového stromu.

Průměr souvislého grafu je délka nejkratší cesty mezi dvěma jeho navzájem nejvzdálenějšími uzly. Ve stromu vede mezi každými dvěma uzly jen jediná cesta, tudíž průměr stromu je pouze je délka cesty mezi dvěma navzájem nejvzdálenějšími uzly. Připomeňme že výška uzlu *x* je rovna vzdálenosti z *x* do jemu nejvzdálenějšího listu, který leží v podstromu s kořenem *x*.

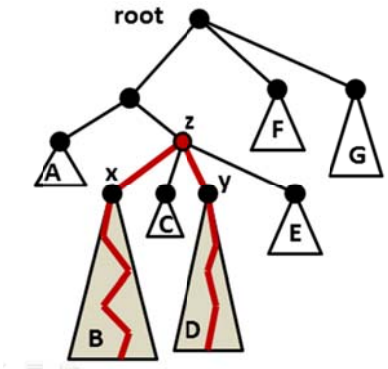
V kořenovém stromu nejdelší cesta buď prochází nebo neprochází kořenem.
I. Pokud nejdelší cesta kořenem prochází, mohou nastat dvě možnosti, neformálně naznačené na obr. 1."
a) Kořen má jediného potomka, délka nejdelší cesty je rovna výšce kořene.
b) Kořen má alespoň dva potomky, délka nejdelší cesty je rovna součtu výšek dvou potomků (*x* a *y*) s největší výškou mezi všemi potomky kořene a je dále zvětšená o 2, díky hranám (kořen, *x*) a (kořen , *y*) .



a) b)

Obr. 1.

II. Pokud nejdelší cesta neprochází kořenem, což je naznačeno na obr. 2., existuje jiný uzel z, a je právě jeden, takový, že nejdelší cesta prochází uzlem z a navíc ještě dvěma jeho podstromy. V tomto případě je, podobně jako v předchozím případě b) délka nejdelší cesty rovna součtu výšek kořenů těchto dvou podstromů, opět zvětšených o 2.



Obr. 2.

Stačí tedy znát výšku každého uzlu z a v každém uzlu pomocí výšek jeho potomků zkonstruovat kandidáta na délku nejdelší cesty v podstromu s kořenem z, podle varianty a) nebo b). Z kandidátů vypočtených ve všech uzlech pak jen vybereme největšího.

Výšku každého uzlu z snadno určíme rekurzivně jako 1 + max{výšky všech bezprostředních potomků z} nebo 0, pokud z je listem. Složitost této akce je Θ(N). Výpočet kandidáta na délku maximální cesty procházející uzlem z má složitost lineárně závislou na počtu potomků z, tj na stupni z. Ve stromu je součet stupňů všech uzlů úměrný počtu uzlů, takže i výpočet kandidátů a výběr největšího má složitost Θ(N). Tudíž i nalezení průměru kořenového stromu má složitost Θ(N).

Abychom mohli efektivně implementovat postup popsaný výše, musíme mít seznam uzlů a ke každému uzlu seznam jeho bezprostředních potomků. Algoritmus hledání minimální kostry může vracet seznam hran kostry, z něj příslušné seznamy vygenerujeme snadno v čase $\Theta(N)$. Minimální kostru můžeme rovnou získat ve stromové reprezentaci. Např. Primův algoritmus určí každému uzlu přímo jeho rodiče ve stromu minimální kostry, stačí pak obrátit hrany v této (orientované) kostře. I to lze učinit v čase $\Theta(N)$.

Alternativní postup nalezení průměru stromu představuje následující algoritmus:
Zvol libovolný uzel *x*. Spusť BFS z uzlu *x* a každému uzlu stromu urči vzdálenost od *x* při jednotkové váze každé hrany. Zvol libovolný uzel *z* s maximální vzdáleností od *x*. Spusť BFS z uzlu *z* a každému uzlu stromu urči vzdálenost od *z* při jednotkové váze každé hrany. Vzdálenost nejvzdálenějšího uzlu od *z* je rovna průměru stromu.
Je třeba upozornit, že tento algoritmus je funkční pouze pro stromy, při existenci byť i jediné kružnice v grafu může selhat a vydat chybný výsledek..

Uzavíráme, že při výpočtu průměru minimální kostry souvislého grafu zbývá po nalezení minimální kostry vykonat jen $\Theta(N)$ operací, což je asymptoticky méně než vyžaduje samotný výpočet minimální kostry, takže podstatné bude především co nejefektivněji nalézt minimální kostru.

Graf, jehož minimální kostru hledáme, je úplný, má tedy $\Theta(N^2)$ hran. Každou hranu je nutno navštívit alespoň jednou, takže složitost hledání minimální kostry je $\Omega(N^2)$. Nejjednodušší způsob, jak dosáhnout složitosti právě $\Theta(N^2)$, je modifikovat Primův algoritmus tak, aby nepoužíval prioritní frontu. Nalezení otevřeného uzlu s nejnižší cenou, to jest toho uzlu, který chceme v průběhu algoritmu aktuálně připojit ke vznikající kostře, uděláme triviálně tak, že projdeme v cyklu všechny uzly grafu a z otevřených uzlů vybereme ten s nejmenší cenou. To má složitost $\Theta(N)$. Asymptoticky tím nic neztrácíme, protože vzápětí v dalším kroku algoritmu musíme probrat a případně v konstantním čase aktualizovat všechny sousedy tohoto nově vybraného uzlu a jeho sousedů je v úplném grafu právě $N-1$, čili také $\Theta(N)$.
Výhodou uvedené modifikace Primova algoritmu je rovněž fakt, že váhu každé hrany potřebujeme použít v průběhu algoritmu jen jednou, takže si váhy hran nemusíme nikam ukládat a stačí ukládat pouze informace o uzlech. Při očekávané velikosti N = 10000, má graf cca 50 milionů hran, což znamená že ušetříme přinejmenším stovky MB paměti a ovšem také čas na její alokaci a případnou další manipulaci s hranami. Pro N = 10000 tak budeme pracovat s maximálně cca 1MB paměti obsahující pouze údaje o uzlech grafu resp. hranách minimální kostry, jichž je také přibližně N.
Pokud použijeme Kruskalův algoritmus, bude jeho rychlost závislá na rychlosti řazení množiny všech hran podle jejich ceny. Typicky při použití zabudovaného řazení je to $\Theta(E \cdot \log(E))$, kde E je počet hran, což je ale rovno $\Theta(N^2 \cdot \log(N^2)) = \Theta(N^2 \cdot \log(N))$. Proti uvedené modifikaci Primova algoritmu je tento postup tedy horší o faktor $\log(N)$, což pro N = 10000 představuje zhruba alespoň desetinásobné zpomalení.

Dále uvádíme tabulky s časy řešení celé úlohy pro jednotlivé velikosti grafu a použité algoritmy výpočtu minimální kostry. Jak plyne z rozboru výše, výpočet průměru kostry zahrnutý v těchto časech nemá zásadní vliv na délku jednotlivých časů, zejména při objemnějších datech.

Intel Core 2 Duo L7500 1.60 Ghz	Prim klasicky	Prim bez fronty	Kruskal
N = 500	31	15	110
N = 1000	78	78	610
N = 2000	234	265	2750
N = 5000	1484	1625	24047
N = 8000	3797	4156	67610
N = 9000	4844	5516	88718

Intel Core i7-2620M 2.70 Ghz	Prim klasicky	Prim bez fronty	Kruskal
N = 500	8	6	32
N = 1000	19	18	205
N = 2000	66	70	928
N = 5000	388	430	8483
N = 8000	968	1084	24465
N = 9000	1251	1439	32693

Tab 1. a 2. Časy řešení v milisekundách, N představuje počet uzlů grafu.

Poznámka

Z dat je zřejmé, že klasická implementace Primova algoritmu je pro tento typ dat o jednu až dvě desítky procent rychlejší než námi uvažovaná implementace bez fronty, a že s rostoucí velikostí dat se tento poměr zvolna zlepšuje ve prospěch klasické varianty. I přesto je původně zvolená varianta řešení dobře přijatelná, především pro svou jednoduchou implementaci. Klasická varianta obsahuje manipulaci s frontou, mnohdy v teoretickém rozboru decentně zamlčenou, kdy je nutno aktualizovat pozice libovolných prvků ve frontě, to jest nejen prvku na čele nebo chvostu fronty. Ne každá knihovní prioritní fronta to dokáže, a pak je nutno vymýšlet další nadbytečné okliky, které tento nedostatek nějak nahradí, pravděpodobně za cenu ztráty času a paměťových prostředků. Naopak, při vlastní implementaci fronty se vystavujeme nutnosti vytvořit objemnější kód nesoucí s sebou větší riziko chyb a překlepů.