6. domácí úkol | Vilém Zouhar

1

1.1 Popis

Budeme procházet graf DFS a v některých klíčových momentech budeme asfaltovat. Kdykoliv budeme chtítu uzavřít hranu, tak se podíváme na to, kolik má nevyasfaltovaných hran směrem k uzavřeným vrcholům:

- 0 Nedá se nic dělat, nebudeme asfaltovat nic. Hranu, směrem sem vyasfaltuje rodič.
- 1 Vyasfaltujeme tuto hranu + hranu vedoucí do rodiče. Pokud toto byla jediná hrana rodiče, ten bude mít při zpravování číslo 0 a graf je tedy vyasfaltovaný.
- 2 a více Po dvojicích vyasfaltujeme hrany a zbude nám buď 0, nebo 1 (viz předchozí kroky).

Při stoupání nahodu vyasfaltujeme všechno. Pokud máme lichý počet hran, tak v posledním kroku nám zbude jedna nevyasfaltovaná hrana. (viz. *Navazující otázka*)

1.2 Korektnost

Rozebráno v popisu.

1.3 Pseudokód

```
DFS(G, root, close_vertex_fun = function(v) {
        switch v.gh {
                0: break;
                1: asphalt(v.g[0], v); asphalt(v, v.parent); break;
                2: for i in range(v.gh - v.gh % 2)
                        asphalt(v.g[i], v);
                        asphalt(v, v.g[i+1])
                   this()
                   break;
        }
)
```

1.4 Složitost

Děláme DFS s pár věcmi navíc. Uperace u uzavírání jsou závislé na počtu hran, tedy celkově $O(m+n)$.

1.5 Navazující otázka

Sudý graf := graf se sudým počtem hran; lichý graf := graf s lichým počtem hran
Graf s 0 nebo 2 hranami vyasfaltujeme triviálně. Pro spor předpokládejme, že existuje nějaký sudý graf s alespoň dvěma hranami, který nedokážeme vyasfaltovat. Vezmeme libovolnou hranu e, která ze souvislosti a sudosti má alespoň jednu další hranu vedle sebe.
Pokud má jednu, tak dohromady tvoří u grafu ocásek, který když vyasfaltujeme (odstřihneme), tak zbude sudý souvislý graf, který opět vyasfaltovat nejde, ale je menší než náš vybraný $\rightarrow$ spor.
Pokud má dvě hrany vedle sebe, tak si vezmeme tu první v_1. Pokud odebráním ev_1 vzniknou dva souvislé sudé grafy, tak jsme vyhráli. Pokud nám vzniknou dva liché grafy, tak místo toho odebereme ev_2. V obou případech spor.

2

2.1 Popis

Nejprve si uvědomíme, že most nikdy nemůžeme zorientovat. Kdyby ano, tak to rozhodně není most (existuje jiná cesta přes věc, co taky vypadá trochu jako most, ale není). Stačí si tedy graf rozsekat na tyto *mostové komponenty* (Tarjanův algoritmus) a pak v těchto komponentách orientovat co se dá. Tarjanův algoritmus dělá DFS a v průběhu kromě samotných mostů (které nás zas tolik nezajímají) jsme schopni dostat i hledané *mostové komponenty*.
Na každé takové komponentě provedeme operaci *direct*, která ji začne procházet DFS, kde za sebou orientujeme cestu. Kdykoliv narazíme na příčnou, nebo zpětnou hranu, tak ji zorientuje ve směru hledání, pokračuje z ní dál směrem ke kořeni (za sebou stále orientuje), kde narazíme už na náš zorientovaný kousek a tím víme, že se tudy máme opět vydat.

2.2 Korektnost

Vyplývá z popisu. Využíváme toho, že kdyby cesta DFS vedla do části komponenty, která vypadá jako strom, tedy bychom na jeho dně museli narazit na nějaký list, což by způsobilo problém (neboť jsme právě zorientovali cestu do listu). Tento list by ale rozhodně musel být klasifikovaný předchozí částí algoritmu jako most, tedy jsme ho tady nemohli potkat.

2.3 Pseudokód

```
komponenty = Tarjan(G)
for k in komponenty:
        direct(k)
output << G
```

2.4 Složitost

Tarjanův algoritmus je lineární k součtu počtu vrcholů a hran. Následné DFS na všech komponentách nemůžou projít řádově víc, než celý graf dohromady, tedy složitost je $O(n+m)$.

3

3.1 Popis

Vytvoříme nový orientovaný graf, kde vcholy budou nádraží *v daném čase* a orientované hrany budou značit odjezdy vlaků. Hrany mezi stejným nádražím (akorát v jiném čase) budou představovat čekání na nádraží a hrany mezi různými nádražími cestu vlakem. Každá hrana tedy symbolizuje určitý časový úsek, ať už na nádraží, nebo ve vlaku. Tedy budeme to brát jako její cenu (např. v minutách).
Na tento graf stačí pustit Dijkstrův algoritmus a máme nejrychlejší cestu.
Stavění grafu může být trochu problém. Předpokládejme, že nádraží dokážeme šikovně indexovat do nějakého rozsahu (nebo pěkně hashovat). Tedy máme slovník/strukturu, která nám v konstantním čase přeloží název stanice na pointer někam. To někam je pole (den má 1440 minut, což je konstanta; pokud povolíme cesty delší než den, tak tady použijeme haldu, nebo prostě něco, co nám v logaritmu vyhledá a vloží prvek; jízdní řád je setřízený a to nám může pomoct), kde prvky ukazují na *null*, pokud se hodnota ještě nepoužila a pointer na vrchol v novém grafu, pokud k němu již někdo přistoupil. Procházením jízdního řádu takto začneme vyplňovat tabulky a postupně si pospojujeme celý graf.

3.2 Korektnost

Na pouštění Dijkstry na graf výše popsaný není co zkazit.

3.3 Pseudokód

```
for radek in jizdni_rad:
        dic[rad.A][rad.odjezd].add_edge(dic[rad.B][rad.prijezd], rad.prijezd-rad.odjezd)
output << Dijkstra(dic.get_graph())
```

3.4 Složitost

Dijkstra je lineární k součtu počtu hran a vrcholů. Problematická je část s přidáváním do slovníku/struktury. Pokud by názvy měst byly opravdu ohavné, tak se zhoršíme o logaritmus (velmi nepravděpodobné). Horší je to s tím trikem 1440 minut. Pokud chceme cestovat déle než jeden den, nebo čas je z nějakého důvodu spojitý (nemůžeme nasekat na konečně mnoho minut), tak se musíme uchýlit k binarní haldě, která nás zhorší o logaritmus (třídit nemusíme, vstup je setřízený). Tedy složitost je snad $O(N \cdot V)$ (Nádraží, Vlaky), v horším případě $O(N \cdot V \cdot \log(N \cdot V))$. Paměť vždy $O(N \cdot V)$.