Popis

Nechť máme na vstupu m řádek ve tvaru:

Počátek	*Cíl*	*Doba jízdy*
Počátek	*Cíl*	*Doba jízdy*
	...	

Navíc také máme zadaný počáteční uzel a počáteční čas. *Počátek* a *Cíl* jsou názvy uzlů/měst. Je rozumné předpokládat, že název města je omezený řetězec, který lze efektivně hashovat. Dále máme na vstupu $ř$ řádek ve tvaru:

Počátek	*Cíl*	*Odjezd*
Počátek	*Cíl*	*Odjezd*
	...	

přičemž každá řádka popisuje cestu právě jednoho vlaku. Nyní si budeme stavět speciální orientovaný graf, ve kterém budou hrany cesty vlakem a časem a uzly budou nádraží v čase. Ukázka konkrétního vstupu a vytvořeného grafu:
Doby jízdy:

A	*B*	*15*
A	*C*	*25*
B	*D*	*18*
C	*D*	*03*

Odjezdy:

A	*B*	*05*
B	*D*	*25*
A	*C*	*05*
C	*D*	*40*

Výsledný graf:

$$\begin{tikzcd}
\text{A (2)} \arrow[r] & \text{A (5)} \arrow[r] \arrow[drr] & \text{B(20)} \arrow[r] & \text{B(25)} \arrow[r] & \text{D(43)} \\
 & & & \text{C(30)} \arrow[r] & \text{C(40)} \arrow[u]
\end{tikzcd}$$

Budeme rozlišovat *město* a *uzel - město v čase*. Budeme mít slovníkovou strukturu *cities*, která bude překládat názvy měst na objekty *město*. Pokud budeme po *cities* chtít doposud neexistující město, tak *cities* takový objekt vytvoří. Slovník použijeme také pro dobu jízdy, kde pod klíčem *Počátek#Klíč* budeme uchovávat číselnou hodnotu doby jízdy. Objekt *město* bude obsahovat AVL strom z grafovými uzly (myšleno na mapě v čase). Mapové uzly jsou řazeny dle časů, které k nim náleží. Při procházení odjezdů budeme vytvářet hrany mezi mapovými uzly. Vyhledáme si první patřičné město a v daném AVL stromě hledáme uzel s konkrétním časem. Pokud existuje, tak tam vytvoříme hranu (s dalším, získaným podobně), pokud ne, tak na daném místě jej můžeme vytvořit, vybalancovat a přidat hranu. Pak ale nesmíme zapomenout, že kromě pohybu vlaky se lze pohybovat v čase i čekáním na nádraží. Stačí tedy projít každý AVL strom zleva doprava a pomocí následníků vytvářet hrany.
Na konci stačí už na takový graf spustit třeba Dijkstrův algoritmus, který najde nejrychlejší cestu.

Pseudokód

```
# doby
doby = dic()
for i in range(m):
        from, to, time = input()
        dic[from + "#" + to] = time

# departures
cities = special_dic()
for i in range(r):
        from, to, departure = input()
        city_f = cities[from]
        city_t = cities[to]
        node_f = city_f.avl.get_create(departure)
        node_t = city_t.avl.get_create(departure+doby[from + "#" + to])
        node_f.add_edge(node_t)

# special
start, goal, start_time = input()
node_start = cities[start].avl.get_create(start_time)

# waiting
for city in cities:
        cur = city.avl.get_min()
        while next = city.avl.get_succ(cur):
                cur.add_edge(next)
                cur = next

output(spec_dijkstra(node_start, goal));
```

Upozorňuji, že operátor [ ] je v případě objektu *cities* přetížený (vytváří, pokud neexistuje). Dále pak objektu *node_start* obsahuje tranzitivně odkaz ke zbytku grafu (pakliže je souvislý). Hrany mají váhy dle rozdílu času v uzlech (tj. každá hrana má přiřazený čas, který na ní strávíme). V tomto smyslu je definovaná funkce w

Nejméně přestupů

Podmínku co nejméně přestupů zajistíme tak, že si uvědomíme, jakým způsobem Dijkstra prohledává a vybírá hrany a vrcholy k relaxaci. Místo ukládání atributu vzdálenost od zdroje do každého vrcholu si můžeme ukládat dvojici (vzdálenost, počet navštívených hran) a při vybírání lepší cesty (relaxace) to zohlednit. Tj. relax můžeme předefinovat zhruba na:

```
relax(u, v, G):
        if (v.d > u.d + w(u,v)) or (v.d == u.d + w(u,v) and v.steps > u.steps + 1) :
                v.d = u.d + w(u, v)
                v.pred = u
                v.steps = u.steps + 1
```

Základní verzi to nepokazí, neboť úprava se týká pouze situací, kdy bychom si nepolepšili a doby by byly nerozhodně.

Korektnost

Pokud by existoval lepší způsob dopravy, tak by tomu musela odpovídat nějaká cesta uvnitř našeho komplexního grafu, neboť ten zahrnuje všechny cesty vlakem a všechny možné čekání. Taková cesta by ale byla nejkratší - dorazila by do cíle v nejkratším čase. Avšak na takový graf jsme pustili Dijkstrův algoritmus, takže tu stejnou cestu musí najít též, pokud existuje. Zároveň pakliže je nejlepší, že tak algoritmus lepší nenajde a najde právě tu. Nejméně přestupů bylo argumentováno o odstavec výš.

Složitost

Postavení slovníku trvání jízd trvá $O(m)$, neboť hashování názvů měst můžeme považovat za konstantní záležitost (omezení z reálného života). V pamětí zabere $O(m)$. Dále vybudování základního grafu mezi městy za pomocí odjezdů a příjezdů trvá $O(\check{r} \cdot \log(\check{r}))$ (logaritmus pochází z vyhledávání a vkládání do AVL stromu). V nejhorším případě se vytvoří pokaždé nový mapový uzel v co největším stromě (velikost lineárně $\check{r}$). V paměti to bude zabírat $O(\check{r})$, neboť nejhůře ř-krát provedeme do nějakého stromu insert. Zároveň jsme zde vytvořili právě $\check{r}$ hran. Výsledkem je tedy graf o nejhůře $\check{r}$ hranách a $\check{r}$ vrcholech.
Pro čekání na nádraží musíme projít každý strom právě jednou a přidat ke každému uzlu patřičnou hranu na svého následovníka. To trvá $O(\check{r})$ a vytváříme zhruba $\check{r}$ hran. Na výsledný graf o nejhůře $\check{r}$ hranách a $\check{r}$ vrcholech spouštíme jen lehce upraveného Dijkstru, což je $(\check{r} \cdot \log(\check{r}))$ (implementace za pomocí Fibonacciho haldy/rozumných sebevyvažovacích stromů).
Celkově jsme zabrali $O(\check{r} + m)$ paměti a $O(\check{r} \cdot \log(\check{r}) + m)$ času, kde $\check{r}$ je počet vlakových cest a m počet dvojic, mezi kterými vlak jede.

Mezistanice

Pokud by vlaky zastavovaly v mezistanicích, tak si to můžeme představit tak, že jeden vlak do stanice přijede, kde skončí a vzápětí odjede ze stejné stanice nový vlak. Z uzlu budou tedy vést jak čekací hrana na stejné nádraží v budoucím čase, tak i hrana *pokračujícího* vlaku. Nyní už počet řádků odjezdu není totožný s počtem vlakových cest, avšak pokud $\check{r}$ bude značit počet vlakových cest, tak je složitost stejná.
Tento přístup však narušuje integritu pro řešení nejmenšího počtu přestupů. Toho lze však docílit tak, že se v relaxu podíváme, zdali jsme nepřijeli stejným vlakem. To poznáme tak, že každá vlaková hrana bude mít přiřazeno *id*, což může být dost dobře pořadí vlaku na vstupu a to budeme jednoduše porovnávat. Čekací hrany mohou mít všechny *id = -1*, takže pokud budeme budeme cestovat: *Otrokovice 12:20* → *Otrokovice 12:35* → *Otrokovice 13:01*, tak se nám počítadlo přestupů nezvětší, neboť tato podmínka je ošetřena. Úprava by mohla vypadat následovně:

```
relax(u, v, G):
          if (v.d > u.d + w(u,v)) or (v.d == u.d + w(u,v) and v.steps > u.steps + 1) :
                    v.d = u.d + w(u, v)
                    v.pred = u
                    # not a station and not the same train
                    if (id(u, v) != -1 and id(u, v) != id(v. v.p):
                              v.steps = u.steps + 1
```

Alternativní řešení

Úlohu by šlo řešit i modifikováním Dijkstry. Graf byl obyčejný (sestavený z tabulky doby jízdy) a hrany by u každého města byly v AVL stromu, takže by se pro daný čas dalo zjistit, jaké vlaky mají ještě odjet (nalezne se nejmenší větší než aktuální čas a pak se projdou následníci). Každé město by též u sebe mělo čas, v kolik se do něj dá nejdříve dostat.
Popsaný graf by však mohl být multigraf (není zaručeno, že mezi dvěma městy nejede stejným směrem dva a více vlaků). Na multigrafech klasický Dijkstra nefunguje a bylo by třeba jej ještě dál upravit.