1 Struktur

1.1 Allmänt

Programmet är uppbyggt så att huvudklassen (Lab3) läser in informationen om linjer och hållplatser med hjälp av Lab3Help och skickar sedan in detta till MyPath (implements Path) som skapar ett nätverk (Network) med hjälp av denna datan. Sedan skickar Lab3 en förfrågan av kortaste väg mellan de två hållplatser som angivits i indatan (argument 3 och 4). MyPath räknar ut detta med hjälpklassen Dijkstra, som helt enkelt innehåller en implementation av Dijkstras algoritm

1.2 Network

Används för att bygga upp (och representera) grafen. Varje element är en Node som innehåller elementets värde (t.ex. hållplatsens namn) samt en lista med grannoder (och kostnaden för att ta sig till dem).

1.3 Dijkstra

Vår implementation är anpassad för oändliga grafer ([1]). Den består huvudsakligen av en PriorityQueue (frontier) och ett set av besökta element (explored). Varje element består av den underliggande noden, kostnaden för att ta sig till den från startelementet samt föregående element i den bästa vägen. I varje steg plockar man elementet i frontier med lägst kostnad, går igenom alla dess grannar och för varje granne gör vi följande:

- om den redan är explored så går vi bara vidare
- om den är i frontier så uppdaterar vi dess kostnad omm den nya kostnaden är lägre
- annars lägger vi till den i frontier.

Algoritmen avslutas när elementet man plockar från frontier är samma som målet (hittad väg). För att hämta ut vägen går man bara ”baklänges” till start tack vare att vi har sparat föregående bästa nod.

Om frontier blir tom utan att vi hittat målet innebär det att det inte finns någon väg.

1.4 Pair

Detta är en hjälpklass för att para ihop två värden, till exempel en grannnod och kostnaden att ta sig dit.

2 Tidskomplexitet

Uppgiften är att visa att vi uppfyller tidskomplexiteten:

$$O(v + (e + l)log(v)) \tag{1}$$

där e är antalet kanter, v är antalet noder och l är största nodetikettens storlek.

Eftersom att vår implementation av dijkstras algoritm inte initialiserar för alla noder, så slipper vi den fristående v-termen. Dessutom räknar vi ut hashen i förväg så vi slipper även l-termen (om vi får anta att vi inte har några kollisioner).

Sammanfattningsvis måste vi i värsta fall gå igenom alla kanter och för varje kant så måste i värsta fall modifiera/sätta in ett element i vår priorityqueue (vilket är $O(\log(v))$). Slutligen måste vi bygga upp den bästa vägen vilket är $O(\min(v, e))$. Tidskomplexiteten blir alltså i vårt fall:

$$O(\min(v, e) + e\log(v)) = O(e\log(v)) \tag{2}$$

"Stora framsteg inom datavetenskapen"! :)

2.1 Förtydling kring uppbyggnad av en path

När målet har nåtts måste den bästa vägen returneras. Detta görs genom att man stegar "baklänges" längs varje nods länk till sin bästa föregångare. Tidskomplexiteten för detta är O(V) då vi vägen maximalt innehåller alla noder en gång. Den är dock även O(E) då vi endast lägger till noder om de är kopplade till en känd nod via en kant, dvs. vägen kan aldrig vara längre än antalet kanter (+1). Sammanfattningsvis får vi att den är uppåt begränsad av både e och v och alltså har vi: $O(\min(v, e))$.

Ett resonemang kring varför detta "bakas in" i $O(e\log(v))$ följer här:

1. $v < e$. Då får vi $O(V + e\log(v)) = O(elog(v))$ eftersom att $e\log(v) > v$ om $e > v$
2. $v >= e$ Då får vi $O(e + e\log(v)) = O(e\log(v))$ av uppenbara skäl

I båda fallen har vi alltså $O(e\log(v))$.

3 Testning

Vi har testat programmet med de uppsättningar hållplatser och linjer som var givna (förutom att köra testprogrammet så klart). Vi har även testat olika felaktiga inputs (t.ex. hållplatser som inte finns) för att se att programmet hanterar detta rätt.

Referenser

[1] Dijkstras algorith, https://en.wikipedia.org/wiki/Dijkstra%27s_algorithm#Practical_optimizations_and_infinite_graphs