Kortste paden

Een in de praktijk veel voorkomend probleem is het bepalen van de (minimale) afstand tussen knopen in een graaf. Tom-Tom verdient hier bijvoorbeeld veel geld mee.

In dit probleem kunnen de takken in de graaf gericht zijn of ongericht. Iedere tak heeft een gewicht. Als (A, B) de tak van knoop A naar knoop B is, dan is Gewicht(A, B) het gewicht van die tak. De lengte van een pad tussen twee knopen is gelijk aan de som van de gewichten van de takken op het pad.

We hebben al een algoritme gezien (met dynamisch programmeren) voor het geval dat we een gerichte, acyclische graaf hebben (blz. 89-91 van het boek). Maar dat is wel een heel speciaal geval. Veel grafen zijn niet acyclisch: je kunt er in een kringetje lopen. Dan heb je dus een ander algoritme nodig.

Het algoritme van Dijkstra

Een geschikt algoritme voor het bepalen van kortste paden is het algoritme van Dijkstra. Om dit algoritme te kunnen toepassen mogen de gewichten *niet negatief* zijn, maar dat is in de praktijk niet vaak een beperking: als het gewicht van een tak tussen twee knopen echt een soort afstand/kosten/tijd voorstelt, is dit bijna automatisch niet negatief.

In principe berekent het algoritme van Dijkstra de afstand vanaf de beginknoop naar alle andere knopen in de graaf. Desgewenst kunnen we het algoritme (voortijdig) stoppen, als we de gewenste eindknoop hebben bereikt.

Afstanden vanuit een gegeven knoop

Stel dat we in een graaf vanuit een gegeven knoop (de beginknoop) naar elk van andere knopen een kortste pad aan willen geven. Het is niet al te moeilijk om in te zien dat deze paden dan een boomstructuur zullen gaan vormen. Immers het kortste pad naar een knoop loopt via andere knopen die dichter bij de beginknoop liggen. De verste knoop wordt dus door een tak verbonden met de kortste afstanden boom voor de overige knopen. Dit levert een boomstructuur op.

Het levert ook een idee om deze boom te construeren. Net als bij het algoritme van Prim (voor het bepalen van een minimale opspannende boom, zie blz. 87-89 van het boek) voegen we steeds takken aan een boom toe, nú echter steeds de tak naar de knoop die het dichtst bij de beginknoop ligt. Dit is weer een gretig algoritme.

Hieronder geven we de pseudo-code voor het algoritme van Dijkstra. Hierin
- bevat het array BoomAfstand voor iedere knoop in de boom de (definitieve) afstand vanuit de beginknoop;
- staat ∞ voor ‘oneindig’,
- is V de verzameling van alle knopen,
- is TV de verzameling van knopen die we voorlopig aan de boom hebben toegevoegd,
- is TE de verzameling van takken die we voorlopig aan de boom hebben toegevoegd.

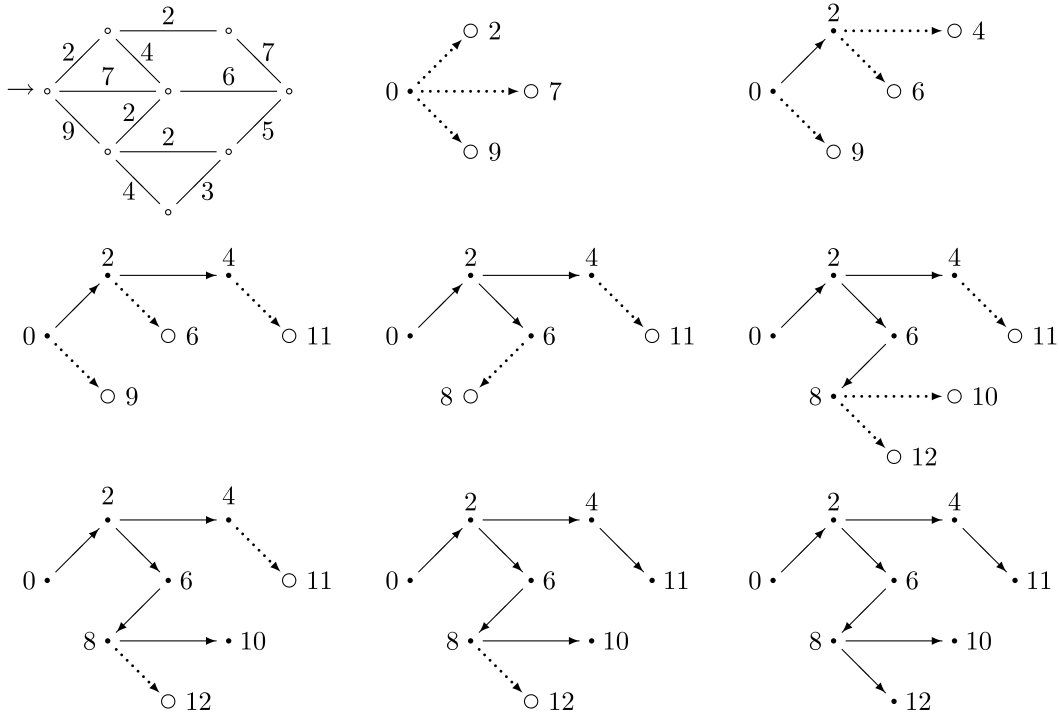
Algoritme : kortste paden volgens Dijkstra

```
      // bouw boom van kortste paden op;
      // begin met een enkele knoop:
    Kies een beginknoop X;
    TV = {X};
    TE = ∅;  // leeg, nog geen takken dus
    BoomAfstand[X] = 0;
      // ga nu knopen en takken toevoegen:
    while (TV ≠ V)
    do  Kies tak (X,Y) met X ∈ TV, Y ∉ TV,
        waarbij Afstand = BoomAfstand[X]+Gewicht(X,Y) minimaal is;
        TV = TV ∪ {Y};
```

```
        TE = TE ∪ {(X,Y)};
        BoomAfstand[Y] = Afstand;
    od
```

Voorbeeld

De constructie van een boom van kortste afstanden volgens Dijkstra's algoritme. De getallen bij de knopen geven de 'boomafstand' voor de knoop weer. Knopen met boomafstand ∞ zijn weggelaten. Tevens is de tak getekend via welke de boomafstand minimaal is.



2 • ——► • 4 tak in boom van kortste afstanden, met afstanden

8 • ·······► ○ 10 Kandidaat-tak, afstand via deze tak (tot nu toe kortste)

□

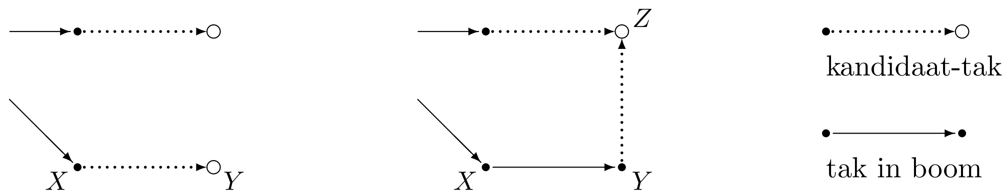
Implementatie

Om het algoritme van Dijkstra te implementeren kunnen we voor iedere knoop Y die nog niet aan de boom is toegevoegd, een kandidaat-afstand bijhouden. Dat is de kortste afstand van de beginknoop naar Y, waarbij alleen takken van de boom gebruikt mogen worden, plus één tak die nog niet in de boom zit. Die laatste tak noemen we de kandidaat-tak van Y. Knopen Y buiten de boom, die niet bereikbaar zijn met slechts één zo'n extra tak, krijgen kandidaat-afstand ∞.

In iedere iteratie van het algoritme kiezen we de knoop Y buiten de boom uit die de kleinste kandidaat-afstand heeft. Samen met zijn kandidaat-tak wordt Y aan de boom toegevoegd.

Neem nu aan dat we tak (X, Y) aan de boom toevoegen, waarbij X al binnen de boom zat en Y aanvankelijk buiten de boom. Dan wordt de kandidaat-afstand van Y nu zijn definitieve afstand. Korter kan kennelijk niet voor Y.

Voor elke knoop Z buiten de boom moeten we nu nagaan of de tak (Y, Z) (als die bestaat) een betere kandidaat is. De afstand vanaf de beginknoop tot Z via de tak (Y, Z) is gelijk aan de afstand van Y plus het gewicht van (Y, Z), zie het volgende plaatje:



Wanneer we bovenstaande opmerkingen verwerken, krijgen we het volgende stuk programma. Hierin
• bevat het array KandidaatAfstand voor iedere knoop buiten de boom de kandidaat-afstanden vanuit de beginknoop,
• bevat het array KandidaatOuder voor iedere knoop buiten de boom de knoop via welke zijn kandidaat-afstand gehaald kan worden (zijn 'kandidaat-ouder').

```
void PasKandidatenAan (Y)
  // pas kandidaten aan na toevoeging van knoop Y aan de boom
{
     for elke tak (Y, Z)
       // elke uitgaande tak vanuit Y dus
     do   if (Z zit nog niet in de boom)
          then Afstand = BoomAfstand[Y] + Gewicht(Y,Z);
               if (Afstand < KandidaatAfstand[Z])
               then KandidaatAfstand[Z] = Afstand;
                    KandidaatOuder[Z] = Y;
               fi
          fi
     od
}    // PasKandidatenAan
```

Om geheugen te besparen kunnen we de arrays BoomAfstand en KandidaatAfstand samenvoegen tot één array. Voor de duidelijkheid hebben we dat hierboven nog niet gedaan.

Een beetje complexiteit

Veronderstel dat de gerepresenteerde graaf N knopen en M takken heeft. Elke tak wordt in het algoritme van Dijkstra één maal gebruikt om te controleren of die tak een kandidaat-tak moet gaan vervangen (dat gebeurt in de functie PasKandidatenAan). Dit kost dus $\mathcal{O}(M)$ stappen bij de *adjacency-list* representatie.

De hoofdlus van het programma wordt $N-1$ maal doorlopen, voor elk toe te voegen knooppunt één maal. Elke keer moet de knoop buiten de boom met minimale kandidaat afstand bepaald worden. Omdat er N knopen zijn kost dit uiteindelijk totaal $\mathcal{O}(N^2)$ stappen. Het helpt slechts een beetje om de knopen buiten de boom in een lijst te hangen. Het totaal aantal knopen wat dan wordt bekeken (op zoek naar het minimum) is $(N-1)+(N-2)+\ldots+1=\frac{1}{2}N\cdot(N-1)$. Dit is ook kwadratisch.

Omdat (voor een samenhangende graaf) $N-1 \leq M \leq N\cdot(N-1)$, is de totale complexiteit voor het aanpassen van kandidaten en het selecteren van de minimale kandidaat $\mathcal{O}(M+N^2)$, wat $\mathcal{O}(N^2)$ is.

Alternatief (geen tentamenstof)

Het kan soms beter. Stop de kandidaat-takken in een heap, die geordend is op de kandidaat-afstand. De heap bevat ten hoogste voor elke knoop één element, dus maximaal N elementen. Een heap operatie (verandering van een waarde, weghalen van het minimum) kost daarom maximaal $\lg(N)$ stappen. Er moet maximaal M maal een kandidaat-tak vervangen worden door een nieuwe, en er moet N maal de minimale kandidaat gekozen worden. Totaal zijn dit $M+N \leq 2\cdot M$ heap operaties. De complexiteit is daarmee $\mathcal{O}(M\cdot\lg(N))$. Dit is beter dan $\mathcal{O}(N^2)$ mits het aantal takken van de graaf relatief klein is, in ieder geval minder dan $\frac{N^2}{\lg(N)}$. Wanneer het aantal takken ongeveer dit aantal is, hangt de keuze voornamelijk van de precieze implementatie af.