Radboud Universiteit Nijmegen

Bachelorscriptie Wiskunde

Stelste routes bepalen op wegennetwerken

Auteur:
Alex Brouwers
0214019

Begeleider:
dr. W. Bosma

30 Augustus 2012

Probleem beschrijving

Definitie van een snelste route.

We zijn we op zoek naar een snelste route tussen twee kruispunten, onder een snelste route verstaan we een route die de minste tijd kost. Een *route* van kruispunt A naar kruispunt B is een lijst van gerichte wegen $w_1, \ldots, w_n$, zodat:

- A is het beginpunt van w_1.
- $\forall i \in \{1, \ldots, n-1\}$: het eindpunt van w_i is gelijk aan het beginpunt van w_{i+1}.
- B is het eindpunt van w_n.

Daarnaast is $\{w\} = \emptyset$ een route als $A = B$.
Vervolgens is een *snelste route* van kruispunt A naar kruispunt B een lijst van wegen $w_1, \ldots, w_n$, zodat:

- $w_1, \ldots, w_n$ is een route van A naar B,
- Voor elke $v_1, \ldots, v_{n_v}$, route van A naar B, $\sum_{i=1}^{n_v} \text{tijd}(v_i) \geq \sum_{i=1}^{n} \text{tijd}(w_i)$.

Aangezien we ervan uitgaan dat het wegennetwerk samenhangend is, bestaat er voor elk begin- en eindpunt minstens één route. Aangezien we er ook vanuit gaan dat het gebruiken van een weg altijd een positief aantal tijd kost, bestaat er een snelste route. Het is alleen niet zo dat er maar één snelste route is, aangezien er twee routes kunnen zijn die dezelfde tijd kosten. We kunnnen dus niet praten over de snelste route, wel over de kleinste tijd.

Modelleren van een wegennet naar een graaf.

Om op een wegennetwerk snelste routes te kunnen berekenen, is maar een beperkt deel van alle eigenschappen van het wegennetwerk nodig. Namelijk:

- De verzameling van kruispunten;
- De verzameling van wegen;
- De lengte, beginpunt en eindpunt van elk van deze wegen.

Deze informatie is weer te geven in een samenhangende gerichte kantgelabelde graaf. Hierbij zijn de knopen de representanten van de kruisingen, en de kanten representeren de wegen tussen deze kruisingen.
Voor elk kruispunt is er een knoop v die deze representeert, V de verzameling van alle knopen. Voor elke (gerichte) weg is er een gerichte kant $e = (v_1, v_2)$ met label t, waarbij $v_1, v_2 \in V$. v_1 is het beginpunt van de weg, v_2 het eindpunt, en t de tijd die het kost om de weg af te leggen. E is de verzameling van alle kanten. De graaf $G = (V, E)$ representeerd het wegennetwerk.
Het verschil in het vinden tussen de snelste en kortste route is slechts een verschil van de waarde op het label. Omdat het makkelijker is over afstand te praten dan over tijdsduur, worden veel van de algoritmes daarmee beschreven. Wij zullen dat ook doen, en het label zien als de afstand van de kant.
Om informatie snel voor handen te hebben, koppelen we de data van de kanten (de objecten) aan de bijbehorende knopen. Bij een knoop slaan we zowel de kanten die ervan weglopen, als de kanten die er naar toe lopen op.

Dijkstra's algoritme

In 1959 beschreef Dijkstra een iteratief algoritme, dat de kortste afstand bepaalt tussen twee knopen. De algoritme heeft als invoer een samenhangende gerichte graaf, met op de kanten labels die de lengte (≥ 0) aangeven. Daarnaast twee knopen, beginknoop A en eindknoop B, waartussen de kortste afstand wordt bepaald.
Elke knoop k van de graaf krijgt een label $A(k)$ met een afstand toegewezen, deze geeft de kortst gevonden afstand vanaf de beginknoop aan. In de initi ele stand heeft enkel de beginknoop afstand 0, de rest heeft afstand ∞. Elke iteratie wordt er een afstand vastgezet.

- 1. Bepaal de nog niet vastgezette knoop x met de kortste gevonden afstand $A(x)$. Zet deze knoop vast.
- 2. Is x de eindknoop B, dan stopt het algoritme. De kortste afstand is $A(x)$.
- 3. Voor elk van de kanten (x, y) met beginknoop x; als $A(y) > A(x) + t(x, y)$, update het label van y naar de kortere afstand. Ga verder met stap 1.

Dat kortste gevonden afstand is de werkelijk kortste afstand, wanneer een vastgezette knoop niet meer wordt geupdate. We gaan aantonen dat: Alle afstanden van knopen zijn kleiner dan (of gelijk aan) de afstanden van niet vastgezette afstanden.
Met behulp van inductie:

- Beginstap: Er geen vastgezette knopen.
- Inductiestap: Stel voor stap 1 geldt de uitpsraak. In stap 1 zetten we de knoop met de kleinste afstand vast. Oftewel, als voor stap 1 de uitspraak klopt, dan hierna ook. Merk hierbij op dat de vastgezette knoop, die we voor het gemak de pivot noemen, de grootste afstand

heeft van alle vastgezette knopen. In stap 3 worden afstanden van knopen geupdate, maar ze worden alleen kleiner. We hoeven dus alleen in te zien dat een niet vastgezette knoop een afstand moet houden die groter is dan die van de vastgezette knopen, dus groter dan de afstand van de pivot. Voor elke kant (pivot,x) geldt: $A(\text{pivot}) + t(\text{pivot}, x) \geq A(\text{pivot})$, want alle afstandlabels van kanten zijn positief. Oftewel de afstand van de niet vastgezette knoop x blijft groter dan de afstand naar de pivot. Hierdoor geldt de uitspraak nog steeds na stap 3. □

We hebben nu gezien dat alle afstanden van vastgezette knopen kleiner zijn dan de afstand naar de pivot knoop. Knopen kunnen alleen geupdate worden, als de oude afstand strikt groter is dan die van de pivot knoop. Dit kan dus geen al vastgezette knoop zijn.
Door bij elke knoop ook de kant op te slaan van waaruit de kortste afstand is bereikt, kun je vanuit de eindknoop teruglopen en zo de kortste route vinden, met maar een kleine aanpassing. Deze kant wordt terugelijkertijd met de gelabelde afstand geupdate.

Binaire queue

De algoritme is in de tussentijd verbeterd. Zij $\#V$ het aantal knopen van de graaf, en U het maximale aantal uitgaande kanten van een knoop. Per iteratie stap moeten hoogstens V waardes vergelijkt worden om de kleinste niet vastgezette afstand te vinden en hoogstens U waardes van knopen geupdate worden. Elke iteratiestap zet een knoop vast, dus er zijn maximaal V iteratiestappen. De snelheid is daarom $O(V \cdot (V + U))$. Aangezien bij wegennetwerken U vele malen kleiner is dan V, kost vooral het opzoeken van de kleinste waarde veel tijd.
Bij een binaire queue wordt gebruik gemaakt van een geordende gebalanceerde binaire boom. Het kleinste element bevind zich in de stam, en elk kind is groter dan zijn ouder. Hierdoor kan de knoop met de kleinste afstand verwijderd worden in $O(\log(V))$. Het toevoegen of updaten van een afstand duurt langer dan eerst, namelijk ook $O(\log(V))$ in plaats van $O(1)$, om de ordening in stand te houden. De totale complexiteit van het Dijkstra algoritme met binaire queue wordt hierdoor $O(V(U+1) \cdot \log(V)) = O(V \cdot U \cdot \log(V))$.
Er zijn methodes die een kleinere (gemiddelde) complexiteit hebben. Een voorbeeld hiervan is met behulp van een Fibonacci Queue, wat bestaat uit een bos van geordende (niet binaire) bomen. Hoewel deze methode in theorie, en bij zeer grote grafen, sneller is; blijkt dit niet zo te zijn voor praktische

toepassingen in het algemeen, en voor bestaande wegennetwerken in het bijzonder.

A* algoritme

Dijkstra's Algoritme vindt meer dan enkel de afstand naar de eindknoop. Iteratief worden alle afstanden naar de knopen die dichter bij de beginknoop liggen gevonden. Het algoritme zoekt daarbij in alle richtingen tegelijkertijd. Door de algoritme een richtingsgevoel te geven, zal hij minder iteraties nodig hebben voor hij het aankomstpunt vindt, waardoor het resultaat sneller gevonden wordt. Voor het richtingsgevoel gebruiken we een *heursitiek* h(x), een schatting van de kortste afstand tot de eindknoop.
Stap 1. in Dijkstra's Algoritme wordt vervangen door:

- 1. Bepaal de nog niet vastgezette knoop x waarbij de kortste gevonden afstand plus de *heuristiek*, $A(x) + h(x)$, minimaal is. Zet deze knoop vast.

We willen de eigenschap behouden dat een vastgezette knoop nooit meer geupdate wordt. We zullen laten zien dat hiervoor *Bellman's Condition* moet gelden: $h(x) \leq t(x, y) + h(y)$.
Dit doen we door de labels op de kanten aan te passen, en te laten zien dat het te reduceren is tot Dijkstra's algoritme. We vervangen elke
We beginnen weer met aantonen dat alle vergelijkingswaarden $(A(x)+h(x))$ van vastgezette knopen groter zijn dan of gelijk zijn aan de vergelijkswaarden van niet vastgezette knopen.
Met inductie:

- Beginstap: Er zijn geen vastgezette knopen.
- Inductiestap: Stel de uitspraak geldt voor stap 1. In stap 1 wordt de knoop met de kleinste vergelijkswaarde vastgezet, hierna de pivot genoemd. De uitspraak geldt dus ook na stap 1. In stap 3 wordt mogelijkerwijs de afstand van een knoop geupdate. Dit verlaagt alleen de waarde, we zijn dus klaar als de nieuwe vergelijkswaarde van de knoop groter is dan die van de pivot. Stel x wordt geupdate. Dan

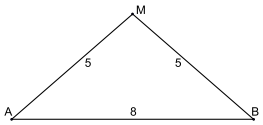
volgens *Bellman's Conndition* volgt: $A(\text{pivot}) + h(\text{pivot}) \leq A(\text{pivot}) + t(\text{pivot}, x) + h(x)$. Aangezien x geupdate wordt, is $A(\text{pivot}) + t(\text{pivot}, x) = A(x)$. Zodoende is de vergelijkingswaarde van de pivot: $A(\text{pivot}) + h(\text{pivot})$ kleiner dan de nieuwe vergelijksingswaarde van de knoop x: $A(x) + h(x)$. □

We hebben nu gezien dat alle vergelijkingswaarden van vastgezette knopen kleiner zijn dan de vergelijkinswaarde van de pivot knoop.Knopen kunnen alleen geupdate worden, als de oude vergelijkingswaarde strikt groter is dan die van de pivot knoop. Dit kan dus geen al vastgezette knoop zijn.
We kunnen A^* ook reduceren tot Dijkstra's algoritme, als we de afstand bij de kanten aanpassen. De nieuwe lengte $v(x, y)$ wordt gedefinieerd door $t(x, y) + h(y) - h(x) \geq 0$.
Stel we vinden een kortste route op deze graaf met aangepaste afstanden, $w_1, \ldots, w_n = (x_1, y_1), \ldots, (x_n, y_n)$, waarbij x_1 de beginknoop, y_n de eindknoop, en voor elke $i_{\in\{1,\ldots,n-1\}}$ geldt: $y_i = x_{i+1}$. $\sum_{i=1}^{n} v(w_i) = \sum_{i=1}^{n} v(x_i, y_i) = \sum_{i=1}^{n}(t(x_i, y_i) + h(y_i) - h(x_i)) = \sum_{i=1}^{n-1}(t(x_i, y_i) + h(x_{i+1}) - h(x_i)) + t(x_n, y_n) + h(y_n) - h(x_n)) = h(y_n) - h(x_1) + \sum_{i=1}^{n} t(x_i, y_i)$. De begin- en eindknoop staan vast, en daardoor ook $h(y_n) - h(x_1)$. Aangezien $\sum_{i=1}^{n} v(x_i, y_i)$ minimaal is, is $h(y_n) - h(x_1) + \sum_{i=1}^{n} t(x_i, y_i)$ het ook, en dus is $\sum_{i=1}^{n} t(x_i, y_i)$ minimaal. Dus $w_1, \ldots, w_n$ is ook een kortste route in de oorspronkelijke graaf.
We hebben het nog niet gehad over het nut van de *heuristiek*. Zonder verlies van algemeenheid kunnen we kijken naar *heuristieken* waarbij de *heuristiek* van de eindknoop gelijk is aan 0. Dijkstra's algoritme is te beschrijven als A^* met als *heuristiek* de nulfunctie. Voor elke knoop geldt, dat als de *heuristiek* van de knoop hoger is, hij minder snel als pivot wordt aangewezen, en hopelijk later dan de eindknoop. De maximale *heuristiek* heeft op elk punt de afstand tot de eindknoop, anders voldoet hij niet aan de *Bellman's Condition*. Alle kanten naar de eindknoop hebben dan een aangepaste afstand van 0. Voor een goede *heuristiek* moet een balans gevonden worden tussen de preciesheid van de afschatting en de tijd die het kost om hem te berekenen.
Voor onze metriek gebruiken we de coordinaten van de kruispunten. Zonder dat we weten waar de wegen liggen, weten we zeker dat de afstand over de weg langer is dan de hemelsbrede afstand. We kunnen bij elke knoop bepalen wat de hemelsbrede afstand is naar de eindknoop. De meeste wegennetten hebben overal wegen in elke richting, waardoor dit een goede afschatting is. Aangezien we op zoek zijn naar de snelste route, hebben we niet genoeg aan alleen de hemelsbrede afstand. We zijn namelijk geintresseerd in een

onderschatting van de kleinste tijdsduur. Hiervoor hebben we ook de maximale snelheid nodig waarop de wegen afgelegt mogen worden. We nemen als *heuristiek* de tijdsduur van de hemelsbrede afstand tot de eindknoop afgelegt met de maximale snelheid. Voor het berekenen van snelste routes voor personenauto's in Nederland is de maximale snelheid bijvoorbeeld 130 km/u. Omdat er relatief weinig snelwegen zijn, en het verschil tussen 130 en 80 km/u groot is, is de afschatting voor de snelste route veel slechter. Hierdoor zal A^* veel minder nuttig zijn.

Bidirectionele variant

In plaats van alleen vanuit het beginpunt Dijkstra's algoritme aan te roepen, kunnen we vanuit twee richtingen gaan rekenen. Hiervoor gebruiken we de gespiegelde graaf, waarbij elke kant omgedraait wordt. Hierop roepen we Dijkstra's algoritme aan vanuit de eindknoop naar de beginknoop. Om de twee Dijkstra's algoritmes gelijk te laten lopen, wisselen we de iteraties af. Wanneer bij een van de Dijkstra's algoritmes een knoop vastgezet wordt die al vastgezet is in de andere algoritme, hebben we een route gevonden door de twee gevonden halve routes aan elkaar te plakken. We hebben hier nog niet te maken met een kortste route, aangezien het niet noodzakelijk is dat deze langs de knoop gaat die het meest in het midden ligt. In onderstaand voorbeeld wordt vanuit beide richtingen knoop M vastgezet, terwijl de kortste route tussen A en B niet langs knoop M gaat.



Wat we nu wel hebben, is een bovenschatting van de kortste afstand. Laat voor knooppunt x, $A(x)$ de afstand tot x zijn vanaf de beginknoop, en $B(x)$ de afstand vanaf x tot de eindknoop; M het gevonden midden, de knoop die in beide Dijkstra's algoritmes vastgezet is. We gaan kijken naar een knooppunt N, die op de kortste route ligt. De gevonden bovenschatting van de afstand is $A(M)+B(M)$. Stel $A(N)+B(N) < A(M)+B(M)$, dan geldt dat $A(N) < A(M)$ of $B(N) < B(M)$. Oftewel elke knoop N op de kortste route is in een van de twee algoritmes vastgezet. Wanneer we het midden gevonden hebben, hoeven we alleen nog te kijken naar wegen tussen de twee verzamelingen vastgezette knopen.

Bidirectionele A^*

Aangezien we nu ook teruguit willen rekenen, werkt de *heuristiek* niet meer. De beide algoritmes hebben namelijk een ander aankomstpunt waar de *heuristiek* tot wordt bepaald. Wanneer met twee verschillende *heuristieken* gewerkt wordt, kunnen de halve routes niet aan elkaar worden geplakt. Daarom maken we een *heuristiek* die twee richtingen op werkt. Namelijk het gemiddelde van beide *heuristieken*.
Stel h_1 de *heuristiek* naar de eindknoop, h_2 de *heuristiek* naar de beginknoop. Voor iedere kant (x,y) geldt: $h_1(x) \leq t(x,y) + h_1(y)$ en $h_2(x) \leq t(x,y) + h_2(y)$. Hierdoor geldt $h_1(x) + h_2(x) \leq 2t(x,y) + h_1(y) + h_2(y)$, waaruit volgt $1/2(h_1(x) + h_2(x)) \leq t(x,y) + 1/2(h_1(y) + h_2(y))$. $1/2(h_1(x) + h_2(x))$ voldoet dus aan de *Bellman's Condition*, en is vanuit twee richtingen te gebruiken. Merk op dat nu voor de eindknoop (en de beginknoop) de *heuristiek* niet langer 0 is. De kwaliteit van de afschatting neemt hierdoor af met gemiddeld ongeveer $1/4$.

Het filteren van de wegen

Niet elke weg van een wegennetwerk is even belangrijk. De meeste woonwijken worden enkel gebruikt als je in de buurt iets te doen hebt. Voor snelwegen maakt het juist weinig uit of je daarbij in de buurt vertrekt of aankomt. Om te zorgen dat er geen rekentijd wordt gebruikt voor onbelangrijke kanten, gaan we een nieuw begrip invoeren. de *invloed* van een kant. De *invloed* geeft aan hoe ver weg knopen moeten liggen, wil deze kant niet op de kortste route liggen. Een voorberekening berekend eenmalig voor een graaf de *invloed* van alle wegen. Daarna is het te gebruiken bij het bepalen van de kortste route tussen elk tweetal knopen.

- Voor elke kant (x, y) kunnen we een verzameling W defineren van kortste routes die (x, y) bevatten.
- Voor w uit W, kunnen we de kortste afstand bepalen tussen beginknoop en x en tussen y en eindknoop. Het minimum van deze twee afstanden noemen we de *lokale invloed* van x op w.
- De globale *invloed* van x is de maximale *lokale invloed* van x op w met w uit W.

Stel we willen de kortste route berekenen van beginknoop A naar eindknoop B. Voor knoop k is $A(k)$ de afstand vanaf de beginknoop tot k en $B(k)$ de afstand vanaf k tot de eindknoop. Als kant $w = (x, y)$ op de kortste route tussen A en B ligt, weten we zeker dat A(x) of B(y) kleiner is dan *invloed*(w). Het is een kleine verandering om in het bidirectionele algoritme eerst naar de *invloed* van een kant te kijken voordat je de kant gebruikt om een punt te updaten.
We vervangen stap 3 in Dijkstra's algoritme door:

- Voor elk van de kanten $w = (x, y)$ met beginknoop x; wanneer $A(x) \leq$ *invloed*$(w))$ en $A(y) > A(x) + t(x, y)$, update het label van y naar de kortere afstand. Ga verder met stap 1.

Om daarnaast zeker te zijn dat een kant van de kortste route altijd wordt gebruikt, is het belangrijk beter na te denken over de manier om de twee richtingen af te wisselen. In plaats van beurtelings, zoeken we uit welke richting de kleinste nog niet vastgezette afstand bevat. Hierdoor zal het nooit kunnen gebeuren dat een kant aan in een richting genegeerd wordt, en daarna in de andere richting niet.

Het berekenen van de *invloed*

De *invloed* hoeft maar een keer per graaf berekend te worden. Om de invloed efficient te kunnen berekenen gebruiken we Dijkstra's algoritme, waarbij we met een berekening vanuit één beginknoop kortste routes naar iedere eindknoop kunnen berekenen. Daarna kan, voor elk van de kortste routes, de *invloed* van de kanten opgehoogt worden tot de *lokale invloed* van de kant binnen de route.
Wanneer de *invloed* berekend wordt op een graaf waar met een *heuristiek* gerekend gaat worden, worden de kortste routes bepaald zonder *heursitiek*. Daarna wordt de *lokale invloed* bepaald met de door de *heuristiek* aangepaste afstanden.
Wanneer de graaf aangepast wordt, moet de *invloed* opnieuw berekend worden.
Wanneer we op zoek zijn naar de snelste route in plaats van de kortste route, heeft *invloed* meer effect. Dit omdat snelle wegen vaak alle *invloed* van de omgeving verlagen, omdat het de enige weg in de omgeving is die gebruikt wordt voor lange-afstandsverkeer.

Computionele resultaten

Om de verschillen in rekentijd te laten zien tussen de verschillende methodes, laten we elk algoritme een aantal kortste routes berekenen. De graaf die we hiervoor gebruiken is een representatie van de gemeente Utrecht, bestaande uit 20007 knopen en 44421 kanten. Hierop hebben we elk van de algoritmes de kortste routes laten bepalen tussen een vaste 100 willekeurig gekozen knopen. Alle algoritmes rekende op dezelfde computer, en zijn in dezelfde stijl geschreven in dezelfde programmeertaal. Allen gebruikte binaire queues voor het sneller vinden van een kleinste waarde. Onderstaande tabel geeft de resultaten, waarbij het aantal iteraties aangeeft hoe vaak een iteratie werd uitgevoerd, oftewel een knoop werd vastgezet.

	rekentijd (sec)	iteraties
Dijkstra	436	98488027
A^*	325	67480631
bidir. Dijkstra	250	66926514
bidir. A^*	219	51987320
bidir. Dijkstra met *invloed*	70	16990296
bidir. A^* met *invloed*	65	14702896

Hierbij moet gemeld worden dat het bereken van de *invloed* 22 936 sec, oftewel ongeveer 6 uur en 23 minuten duurde.

Conclusie

Wanneer je de snelste route wilt weten tussen twee kruispunten op een wegennetwerk, kun je het beste gebruik maken van een bidirectioneel A^* algoritme. Wanneer je vaker snelste routes wilt berekenen op hetzelfde wegennetwerk met dezelfde snelheden, kun je de *invloed* van de wegen rekenen. Dit kost een relatief zeer lange tijd, en is alleen nuttig in speciale gevallen:

- Er moeten heel vaak snelste routes berekend worden.
- Van tevoren is alle tijd beschikbaar, en enkel respondstijd na het vaststellen van begin- en eindpunt is belangrijk.
- De *invloed* kan op een veel snellere computer van tevoren berekend worden, waarna de routes op langzamere computers moet worden berekend.

Vervolgstappen

Wanneer een graaf aangepast wordt, hoeft niet overal de *invloed* opnieuw berekend te worden. Wanneer een kant weggehaald wordt, hoeven alleen de knopen en kanten op een afstand binnen zijn *invloed* opnieuw berekend te worden. Wanneer een kant toegevoegd wordt, is het een lastiger verhaal. Men kan eerst kijken welke knopen aan welke uitgang van de kant liggen. Hierdoor kun je alle kanten selecteren die deze groepen met elkaar verbinden. Met alleen de afstanden berekend vanuit die kanten is de *invloed* van de toegevoegde kant te bepalen. Daarna zal alleen de *invloed* van andere kanten mogelijk verlaagt moeten worden, wat voor grote problemen kan zorgen.

Referenties

- E.W. Dijkstra: *A note on two problems in connexion with graphs.* Numer Math 1 (1959), 269-271.
- J. W. J. Williams: *Algorithm 232 - Heapsort.* Communications of the ACM(1964) 7(6), 347-348.
- R. Bellman: *Dynamic programming.* Princeton University Press, Princeton, NJ, 1957.
- I. Pohl: *Bi-directional search.* Machine Intelligence, Edinburgh University Press, Edinburgh, 1971.
- M.L. Fredman; R.E. Tarjan: *Fibonacci heaps and their uses in improved network optimization algorithms.* J Assoc Comput Machin 34(3) (1987), 596-615.
- G.A. Klunder; H.N. Post: *The Shortest Path Problem on Large-Scale Real-Road Networks.* Wiley InterScience, 2006.

Bijlages

Code: Graaf

```
public class Graph<TVertex> : IGraph<TVertex>
        where TVertex : IIndexable {
  private readonly Quictionary<TVertex, Edge<TVertex>> vertexInEdges;
  private readonly Quictionary<TVertex, Edge<TVertex>> vertexOutEdges;

  public Graph(IIndexedReadOnlyCollection<TVertex> vertexCollection) {
    vertexInEdges = new EasyQuictionary<TVertex, Edge<TVertex>>(vertexCollection);
    vertexOutEdges = new EasyQuictionary<TVertex, Edge<TVertex>>(vertexCollection);
      foreach (var vertex in vertexCollection) {
        vertexInEdges[vertex] = null;
        vertexOutEdges[vertex] = null;
    }
  }

  public IIndexedReadOnlyCollection<TVertex> Vertices() {
    return vertexInEdges.Keys;
  }

  public void AddEdge(InputStreet street, TVertex source, TVertex target) {
    var edge = new Edge<TVertex>(street, source, target,
          vertexOutEdges[source], vertexInEdges[target]);
    vertexOutEdges[source] = edge;
    vertexInEdges[target] = edge;
  }

  public IEnumerable<Edge<TVertex>> OutEdges(TVertex v) {
    var edge = vertexOutEdges[v];
    while (edge != null) {
      yield return edge;
      edge = edge.OutTail;
    }
  }

  public IEnumerable<Edge<TVertex>> InEdges(TVertex v) {
    var edge = vertexInEdges[v];
    while (edge != null)
    {
      yield return edge;
      edge = edge.InTail;
    }
  }
}
```

Code: Dijkstra's Algoritme

```
public class Dijkstra {
  private readonly Graph<Crossing> graph;
  private readonly BinaryQueueAndBin<Crossing, Tuple<double, Edge<Crossing>>> queue;
  private Func<Edge<Crossing>, double> edgeResistanceFunc;
```

```
  public Dijkstra(Graph<Crossing> graph) {
    this.graph = graph;
    queue = new BinaryQueueAndBin<Crossing, Tuple<double, Edge<Crossing>>>(graph.Vertices(), (x,y) => x.Key < y.Key);
  }

  public List<InputStreet> CalculateRouting(Crossing sourceCrossing, Crossing targetCrossing, Func<Edge<Crossing>, double> edgeResistanceFunc) {
    //Set ResistanceFunc
    this.edgeResistanceFunc = edgeResistanceFunc;
    //Cleaning up last calculation (so it can be used multiple times).
    queue.Clear();
    //Set source point.
    var dummyEdge = new Edge<Crossing>(null, 0, 0, null, sourceCrossing, null, null, false);
    queue.Add(sourceCrossing, new Tuple<double, Edge<Crossing>>(0, dummyEdge));
    Crossing foundCrossing;
    while ( ComputeStep(out foundCrossing)) {
      if (targetCrossing.Equals(foundCrossing)) {
        //Found targetCrossing.
        var resultList = new List<GraphBuildingClass.InputStreet>();
        var tempCrossing = targetCrossing;
        Edge<Crossing> tempEdge;
        while(tempCrossing != null) {
          if (!queue.TryGetResistance(tempCrossing, out tempEdge)) {
            throw new ApplicationException("Kan de gevonden weg niet terugvinden");
          }
          if (tempeEdge.Value.Origin == null) {
            //Found begin of route.
            return resultList.Reverse;
          }
          resultList.Add(tempEdge.Value.Street);
          tempCrossing = tempEdge.Value.Origin;
        }
      }
    }
    //No result. There is no route from sourceCrossing to targetCrossing. Garbage in, garbage out: return empty list.
    return new  List<InputStreet>();
  }

  private bool ComputeStep(out Crossing addedCrossing) {
    if (queue.Count == 0) {
      addedCrossing = default(Crossing);
      return false;
    }
    var parent = queue.RemoveMinimum();
    addedCrossing = parent.Key;
    var edgeList = graph.OutEdges(addedCrossing);
    foreach (var edge in edgeList) {
      var edgeResistance = edgeResistanceFunc(edge);
      if (edgeResistance < 0) {
        throw new ArgumentException("Afstanden mogen niet negatief zijn.");
      }
      queueForward.ImproveResistance(edge.Destination, new Tuple<double, Edge<Crossing>>(parent.Value.Key + edgeResistance, edge));
    }
    return true;
  }
}
```

Code: Binaire queue

```
public class BinaryHeapAndBin<TKey, TResistance>
        where TKey : IIndexable {
  private readonly EasyQuictionary<TKey, int> positionTable;
  private readonly KeyValuePair<TKey, TResistance>[] keyDistancePairs;
  public Func<TResistance, TResistance, bool> LessFunc { get; private set; }
  public int Count { get; private set; }
  public int BinCount { get; private set; }


  public BinaryHeapAndBin(IIndexedReadOnlyCollection<TKey> collection, Func<TResistance, TResistance, bool> lessFunc) {
    positionTable = new EasyQuictionary<TKey, int>(collection);
    keyDistancePairs = new KeyValuePair<TKey, TResistance>[collection.Count];
    LessFunc = lessFunc;
    Count = 0;
    BinCount = collection.Count - 1;
```

```
}

public void Add(TKey key, TResistance resistance) {
  positionTable.Add(key, Count);
  Count++;
  keyDistancePairs[Count - 1] = new KeyValuePair<TKey, TResistance>(key, resistance);
  MinHeapifyDown(Count - 1);
}

public KeyValuePair<TKey, TResistance> Minimum() {
  if (Count == 0)
    throw new InvalidOperationException();
  return keyDistancePairs[0];
}

public KeyValuePair<TKey, TResistance> RemoveMinimum() {
  if (Count == 0) {
    throw new InvalidOperationException();
  }
  var result = keyDistancePairs[0];
  Place(BinCount, result);
  BinCount--;
  Count--;
  if (Count != 0) {
    Place(0, keyDistancePairs[Count]);
    MinHeapifyUp(0);
  }
  return result;
}

public bool Contains(TKey key) {
  return positionTable.ContainsKey(key);
}

public bool ContainsInHeap(TKey key) {
  int index;
  return positionTable.TryGetValue(key, out index) && index < Count;
}

public bool ContainsInBin(TKey key) {
  int index;
  return positionTable.TryGetValue(key, out index) && index >= Count;
}

public bool ImproveDistance(TKey key, TResistance newResistance) {
  int index;
  if (positionTable.TryGetValue(key, out index)) {
    if (LessFunc(newResistance, keyDistancePairs[index].Value)) {
      keyDistancePairs[index] = new KeyValuePair<TKey, TResistance>(keyDistancePairs[index].Key, newResistance);
      MinHeapifyDown(index);
      return true;
    }
    return false;
  }
  Add(key, newResistance);
  return true;
}

public bool TryGetResistance(TKey key, out TResistance resistance) {
  int index;
  if (positionTable.TryGetValue(key, out index)) {
    resistance = keyDistancePairs[index].Value;
    return true;
  }
resistance = default(TResistance);
return false;
}

private void MinHeapifyUp(int index) {
  var left = 2 * index + 1;
  var right = 2 * index + 2;
  var keyDistanceIndex = keyDistancePairs[index];
  while (
    (left < Count && Less(left, keyDistanceIndex)) ||
    (right < Count && Less(right, keyDistanceIndex))
  ) {
```

```
      if (right >= Count || Less(left, right)) {
        Place(index, keyDistancePairs[left]);
        index = left;
      } else {
        Place(index, keyDistancePairs[right]);
        index = right;
      }
      left = 2 * index + 1;
      right = 2 * index + 2;
    }
    Place(index, keyDistanceIndex);
  }

  private void MinHeapifyDown(int index) {
    int parent = (index - 1) / 2;
    var keyDistanceIndex = keyDistancePairs[index];
    while (index != 0 && Less(keyDistanceIndex, parent)) {
      Place(index, keyDistancePairs[parent]);
      index = parent;
      parent = (index - 1) / 2;
    }
    Place(index, keyDistanceIndex);
  }

  private bool Less(int index1, int index2) {
    return LessFunc(keyDistancePairs[index1].Value, keyDistancePairs[index2].Value);
  }

  private bool Less(KeyValuePair<TKey, TResistance> keyDistance1, int index2) {
    return LessFunc(keyDistance1.Value, keyDistancePairs[index2].Value);
  }

  private bool Less(int index1, KeyValuePair<TKey, TResistance> keyDistance2) {
    return LessFunc(keyDistancePairs[index1].Value, keyDistance2.Value);
  }

  private void Place(int newIndex, KeyValuePair<TKey, TResistance> keyDistance) {
    positionTable[keyDistance.Key] = newIndex;
    keyDistancePairs[newIndex] = keyDistance;
  }

  public void Clear() {
    positionTable.Clear();
    Count = 0;
    BinCount = keyDistancePairs.Length - 1;
  }
}
```

Code: A* Algoritme

```
public class AStar {
  private Dijkstra dijkstra;

  public AStar(Graph<Crossing> graph) {
    dijkstra = new Dijkstra(graph);
  }

  public List<InputStreet> CalculateRouting(Crossing sourceCrossing, Crossing targetCrossing,
        Func<Edge<Crossing>, double> edgeResistanceFunc, Func<Coordinate, Coordinate, double> heuristic) {
    Func<Coordinate, Coordinate, Edge<Crossing>, double> newEdgeResistanceFunc = (edge) =>
        (heuristic(edge.Destination.Coordinate, targetCoordinate) - heuristic(edge.Origin.Coordinate, targetCoordinate)) +
        edgeResistanceFunc(edge);
    return Dijkstra.CalculateRouting(sourceCrossing, targetCrossing, newEdgeResistanceFunc);
  }
}
```

Code: Bidirectioneel Dijkstra's Algoritme

```
public class BidirDijkstra {
  private readonly Graph<Crossing> graph;
  private readonly BinaryQueueAndBin<Crossing, Tuple<double, Edge<Crossing>>> forwardQueue;
  private readonly BinaryQueueAndBin<Crossing, Tuple<double, Edge<Crossing>>> backwardQueue;
  private Func<Edge<Crossing>, double> edgeResistanceFunc;


  public BidirDijkstra(Graph<Crossing> graph) {
    this.graph = graph;
    forwardQueue = new BinaryQueueAndBin<Crossing, Tuple<double, Edge<Crossing>>>(graph.Vertices(), (x,y) => x.Key < y.Key);
    backwardQueue = new BinaryQueueAndBin<Crossing, Tuple<double, Edge<Crossing>>>(graph.Vertices(), (x,y) => x.Key < y.Key);
  }

  public List<InputStreet> CalculateRouting(Crossing sourceCrossing, Crossing targetCrossing, Func<Edge<Crossing>, double> edgeResistanceFunc) {
    //Set ResistanceFunc
    this.edgeResistanceFunc = edgeResistanceFunc;
    //Cleaning up last calculation (so it can be used multiple times).
    forwardQueue.Clear();
    backwardQueue.Clear();
    //Set source en target points.
    var dummyEdge = new Edge<Crossing>(null, 0, 0, null, sourceCrossing, null, null, false);
    forwardQueue.Add(sourceCrossing, new Tuple<double, Edge<Crossing>>(0, dummyEdge));
    dummyEdge = new Edge<Crossing>(null, 0, 0, targetCrossing, null, null, null, false);
    backwardQueue.Add(targetCrossing, new Tuple<double, Edge<Crossing>>(0, dummyEdge));
    Crossing meetingCrossing;
    while (true) {
      if (queueForward.Count + queueBackward.Count > 0) {
        //Garbage in, garbage out.
        return new List<InputStreet>();
     }
      if (ComputeStepForward(out meetingCrossing)) {
        if (queueBackward.ContainsInBin(meetingCrossing)) {
          break;
        }
      }
      if (ComputeStepBackward(out meetingCrossing)) {
        if (queueForward.ContainsInBin(meetingCrossing)) {
          break;
        }
      }
    }
    Tuple<double, Edge<Crossing>> forwardResistance;
    Tuple<double, Edge<Crossing>> backwardResistance;
    queueForward.TryGetResistance(meetingCrossing, out forwardResistance);
    queueBackward.TryGetResistance(meetingCrossing, out backwardResistance);
    var targetResistance = forwardResistance.Key + backwardResistance.Key;
    while(queueForward.Count > 0) {
      var crossing = queueForward.RemoveMinimum();
      if (!queueBackward.TryGetResistance(crossing.Key, out backwardResistance))
        continue;
      forwardResistance = crossing.Value;
      if (forwardResistance.Key + backwardResistance.Key >= targetResistance)
        continue;
      meetingCrossing = crossing.Key;
      targetResistance = forwardResistance.Key + backwardResistance.Key;
    }
    while (queueBackward.Count > 0) {
      var crossing = queueBackward.RemoveMinimum();
      if (!queueForward.TryGetResistance(crossing.Key, out forwardResistance))
        continue;
      backwardResistance = crossing.Value;
      if (forwardResistance.Key + backwardResistance.Key >= targetResistance)
        continue;
      meetingCrossing = crossing.Key;
      targetResistance = forwardResistance.Key + backwardResistance.Key;
    }

    var resultList = new List<InputStreet>();
    var tempCrossing = meetingCrossing;
    Tuple<double, Edge<Crossing>> tempResistanceEdge;
    while(tempCrossing != null) {
      if (!queueForward.TryGetResistance(tempCrossing, out tempResistanceEdge)) {
        throw new ApplicationException("Kan de gevonden heenweg niet terugvinden");
      }
      if (tempResistanceEdge.Value.Origin == null) {
        break;
```

```
      }
      resultList.Add(tempResistanceEdge.Value.Street);
      tempCrossing = tempResistanceEdge.Value.Origin;
    }
    resultList.Reverse();
    tempCrossing = meetingCrossing;
    while (tempCrossing != null) {
      if (!queueBackward.TryGetResistance(tempCrossing, out tempResistanceEdge)) {
        throw new ApplicationException("Kan de gevonden terugweg niet terugvinden");
      }
      if (tempResistanceEdge.Value.Destination == null) {
        break;
      }
      resultList.Add(tempResistanceEdge.Value.Street);
      tempCrossing = tempResistanceEdge.Value.Destination;
    }
    return resultList;
  }

  private bool ComputeStepForward(out Crossing addedCrossing) {
    if (queueForward.Count == 0) {
      addedCrossing = default(Crossing);
      return false;
    }
    var parent = queueForward.RemoveMinimum();
    addedCrossing = parent.Key;
    var edgeList = graph.OutEdges(addedCrossing);
    foreach (var edge in edgeList) {
      var edgeResistance = edgeResistanceFunc(edge);
      if (edgeResistance < 0) {
        throw new ArgumentException("Afstanden mogen niet negatief zijn.");
      }
      if (edge.Street.Influence >= parent.Value.Key) {
        queueForward.ImproveResistance(edge.Destination, new Tuple<double, Edge<Crossing>>(parent.Value.Key + edgeResistance, edge));
      }
    }
    return true;
  }

  private bool ComputeStepBackward(out Crossing addedCrossing) {
    if (queueBackward.Count == 0) {
      addedCrossing = default(Crossing);
      return false;
    }
    var parent = queueBackward.RemoveMinimum();
    addedCrossing = parent.Key;
    var edgeList = graph.InEdges(addedCrossing);
    foreach (var edge in edgeList) {
      var edgeResistance = edgeResistanceFunc(edge);
      if (edgeResistance < 0) {
        throw new ArgumentException("Afstanden mogen niet negatief zijn.");
      }
      if (edge.Street.Influence >= parent.Value.Key) {
        queueBackward.ImproveResistance(edge.Origin, new Tuple<double, Edge<Crossing>>(parent.Value.Key + edgeResistance, edge));
      }
    }
    return true;
  }
}

\section*{Code: \emph{Iinvloed} berekenen}

\begin{tiny}
\begin{verbatim}
public class InfluenceCalculator {
  private readonly Graph<Crossing> graph;
  private readonly BinaryQueueAndBin<Crossing, Tuple<double, Edge<Crossing>>> queue;
  private Func<Edge<Crossing>, double> edgeResistanceFunc;


  public InfluenceCalculator(Graph<Crossing> graph) {
    this.graph = graph;
    queue = new BinaryQueueAndBin<Crossing, Tuple<double, Edge<Crossing>>>(graph.Vertices(), (x,y) => x.Key < y.Key);
  }

  public void CalculateInfluences(Func<Edge<Crossing>, double> edgeResistanceFunc) {
```

```
    //Set ResistanceFunc
    this.edgeResistanceFunc = edgeResistanceFunc;
    foreach(var crossing in graph.Crossings()) {
      CalculateInfluence(crossing);
    }
  }

  private void CalculateInfluence(Crossing sourceCrossing) {
    queue.Clear();
    //Set source point.
    var dummyEdge = new Edge<Crossing>(null, 0, 0, null, sourceCrossing, null, null, false);
    queue.Add(sourceCrossing, new Tuple<double, Edge<Crossing>>(0, dummyEdge));
    Crossing foundCrossing;
    while ( ComputeStep(out foundCrossing)) {}
    //All crossings reached.
    foreach (var crossing in graph.Vertices()) {
      new Tuple<double, Edge<Crossing>> tempTuple;
      if (!queue.TryGetResistance(crossing, out tempTuple)) {
        continue; //Road not reachable.
      }
      var totalDistance = tempTuple.First;
      while(true) {
        if (tempTuple.Second.Origin == null) {
        break;
      }
      var distanceToEdge = tempTuple.First - edgeResistanceFunc(tempTuple.Second);
      var distanceFromEdge = totalDistance - tempTuple.First;
      var priorityValue = Math.Min(distanceToEdge, distanceFromEdge);
      if (priorityValue > tempTuple.Second.Street.Influence) {
        tempTuple.Second.Street.Influence = priorityValue;
      }
    }
  }

  private bool ComputeStep(out Crossing addedCrossing) {
    if (queue.Count == 0) {
      addedCrossing = default(Crossing);
      return false;
    }
    var parent = queue.RemoveMinimum();
    addedCrossing = parent.Key;
    var edgeList = graph.OutEdges(addedCrossing);
    foreach (var edge in edgeList) {
      var edgeResistance = edgeResistanceFunc(edge);
      if (edgeResistance < 0) {
        throw new ArgumentException("Afstanden mogen niet negatief zijn.");
      }
      queueForward.ImproveResistance(edge.Destination, new Tuple<double, Edge<Crossing>>(parent.Value.Key + edgeResistance, edge));
    }
    return true;
  }
}
```

Code: Bidirectioneel met het filteren van wegen

```
public class InfluencedBidirDijkstra {
  private readonly Graph<Crossing> graph;
  private readonly BinaryQueueAndBin<Crossing, Tuple<double, Edge<Crossing>>> forwardQueue;
  private readonly BinaryQueueAndBin<Crossing, Tuple<double, Edge<Crossing>>> backwardQueue;
  private Func<Edge<Crossing>, double> edgeResistanceFunc;

  public InfluencedBidirDijkstra(Graph<Crossing> graph) {
    this.graph = graph;
    forwardQueue = new BinaryQueueAndBin<Crossing, Tuple<double, Edge<Crossing>>>(graph.Vertices(), (x,y) => x.Key < y.Key);
    backwardQueue = new BinaryQueueAndBin<Crossing, Tuple<double, Edge<Crossing>>>(graph.Vertices(), (x,y) => x.Key < y.Key);
  }

  public List<InputStreet> CalculateRouting(Crossing sourceCrossing, Crossing targetCrossing, Func<Edge<Crossing>, double> edgeResistanceFunc) {
    //Set ResistanceFunc
    this.edgeResistanceFunc = edgeResistanceFunc;
    //Cleaning up last calculation (so it can be used multiple times).
    forwardQueue.Clear();
```

```
backwardQueue.Clear();
//Set source en target points.
var dummyEdge = new Edge<Crossing>(null, 0, 0, null, sourceCrossing, null, null, false);
forwardQueue.Add(sourceCrossing, new Tuple<double, Edge<Crossing>>(0, dummyEdge));
dummyEdge = new Edge<Crossing>(null, 0, 0, targetCrossing, null, null, null, false);
backwardQueue.Add(targetCrossing, new Tuple<double, Edge<Crossing>>(0, dummyEdge));
Crossing meetingCrossing;
var forwardRes = queueForward.Peek().Value.Key;
var backwardRes = queueBackward.Peek().Value.Key;
while (true) {
  if (forwardRes < backwardRes) {
    ComputeStepForward(out meetingCrossing);
    if (queueBackward.ContainsInBin(meetingCrossing)) {
      break;
    }
    forwardRes = queueForward.Count == 0 ? double.MaxValue : queueForward.Minimum().Value.Key;
  } else {
    if (ComputeStepBackward(out meetingCrossing)) {
      if (queueForward.ContainsInBin(meetingCrossing)) {
        break;
      }
      backwardRes = queueBackward.Count == 0 ? double.MaxValue : queueBackward.Minimum().Value.Key;
    } else {
      //Queues are empty. Garbage in, garbage out.
      return new List<InputStreet>();
    }
  }
}
Tuple<double, Edge<Crossing>> forwardResistance;
Tuple<double, Edge<Crossing>> backwardResistance;
queueForward.TryGetResistance(meetingCrossing, out forwardResistance);
queueBackward.TryGetResistance(meetingCrossing, out backwardResistance);
var targetResistance = forwardResistance.Key + backwardResistance.Key;
while(queueForward.Count > 0) {
  var crossing = queueForward.RemoveMinimum();
  if (!queueBackward.TryGetResistance(crossing.Key, out backwardResistance))
    continue;
  forwardResistance = crossing.Value;
  if (forwardResistance.Key + backwardResistance.Key >= targetResistance)
    continue;
  meetingCrossing = crossing.Key;
  targetResistance = forwardResistance.Key + backwardResistance.Key;
}
while (queueBackward.Count > 0) {
  var crossing = queueBackward.RemoveMinimum();
  if (!queueForward.TryGetResistance(crossing.Key, out forwardResistance))
    continue;
  backwardResistance = crossing.Value;
  if (forwardResistance.Key + backwardResistance.Key >= targetResistance)
    continue;
  meetingCrossing = crossing.Key;
  targetResistance = forwardResistance.Key + backwardResistance.Key;
}

var resultList = new List<InputStreet>();
var tempCrossing = meetingCrossing;
Tuple<double, Edge<Crossing>> tempResistanceEdge;
while(tempCrossing != null) {
  if (!queueForward.TryGetResistance(tempCrossing, out tempResistanceEdge)) {
    throw new ApplicationException("Kan de gevonden heenweg niet terugvinden");
  }
  if (tempResistanceEdge.Value.Origin == null) {
    break;
  }
  resultList.Add(tempResistanceEdge.Value.Street);
  tempCrossing = tempResistanceEdge.Value.Origin;
}
resultList.Reverse();
tempCrossing = meetingCrossing;
while (tempCrossing != null) {
  if (!queueBackward.TryGetResistance(tempCrossing, out tempResistanceEdge)) {
    throw new ApplicationException("Kan de gevonden terugweg niet terugvinden");
  }
  if (tempResistanceEdge.Value.Destination == null) {
    break;
  }
```

```
      resultList.Add(tempResistanceEdge.Value.Street);
      tempCrossing = tempResistanceEdge.Value.Destination;
    }
    return resultList;
  }

  private bool ComputeStepForward(out Crossing addedCrossing) {
    if (queueForward.Count == 0) {
      addedCrossing = default(Crossing);
      return false;
    }
    var parent = queueForward.RemoveMinimum();
    addedCrossing = parent.Key;
    var edgeList = graph.OutEdges(addedCrossing);
    foreach (var edge in edgeList) {
      var edgeResistance = edgeResistanceFunc(edge);
      if (edgeResistance < 0) {
        throw new ArgumentException("Afstanden mogen niet negatief zijn.");
      }
      if (edge.Street.Influence >= parent.Value.Key) {
        queueForward.ImproveResistance(edge.Destination, new Tuple<double, Edge<Crossing>>(parent.Value.Key + edgeResistance, edge));
      }
    }
    return true;
  }

  private bool ComputeStepBackward(out Crossing addedCrossing) {
    if (queueBackward.Count == 0) {
      addedCrossing = default(Crossing);
      return false;
    }
    var parent = queueBackward.RemoveMinimum();
    addedCrossing = parent.Key;
    var edgeList = graph.InEdges(addedCrossing);
    foreach (var edge in edgeList) {
      var edgeResistance = edgeResistanceFunc(edge);
      if (edgeResistance < 0) {
        throw new ArgumentException("Afstanden mogen niet negatief zijn.");
      }
      if (edge.Street.Influence >= parent.Value.Key) {
        queueBackward.ImproveResistance(edge.Origin, new Tuple<double, Edge<Crossing>>(parent.Value.Key + edgeResistance, edge));
      }
    }
    return true;
  }
}
```