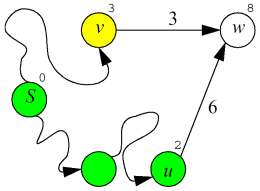
Shortest-Path

Positive-Weighted

- Dijkstra's algoritme
- Als vertex *w* adjacent to vertex *v* is:
  - $dist_w = dist_v + cost_{v,w}$ als deze waarde kleiner is dan de huidige waarde van $dist_w$
- Het is nu niet meer mogelijk om de dist van elke vertex maar 1x aan te passen.



THRijswijk

166

Shortest-Path

Positive-Weighted

- Volgorde van de te bezoeken vertices kan als volgt bepaald worden:
  - Zet start vertex in een priority_queue die min dist geeft.
  - Zolang queue niet leeg:
    - Haal vertex *v* uit priority_queue
    - Voor alle adjacent vertices *w*:
      - Als $dist_w > dist_v + cost_{v,w}$
        - $dist_w = dist_v + cost_{v,w}$
        - Zet vertex *w* in de priority_queue.

```
class GreaterDistance {
public:
   bool operator()(const Vertex* l,
                   const Vertex* r) {
      return l->distance()>r->distance();
   }
};
```

THRijswijk

167

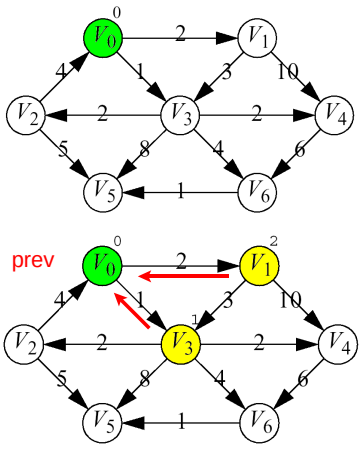
Graph

```
void Graph::dijkstra(const string& name) {
   reset();
   Vertex* start(findVertex(name));
   start->setDistance(0);
   priority_queue<Vertex*, vector<Vertex*>,
                  GreaterDistance> pq;
   pq.push(start);
   while (!pq.empty()) {
      Vertex* v(pq.top());
      pq.pop();
      for (Vertex::const_iterator i(v->begin());
           i!=v->end(); ++i) {
         Vertex* w(i->first);
         int cvw(i->second);
         if (cvw<0)
            throw runtime_error("negative cost!");
         if (w->distance()>v->distance()+cvw) {
            w->setDistance(v->distance()+cvw);
            w->setPrev(v);
            pq.push(w);
         }
      }
   }
}
```

THRijswijk

168

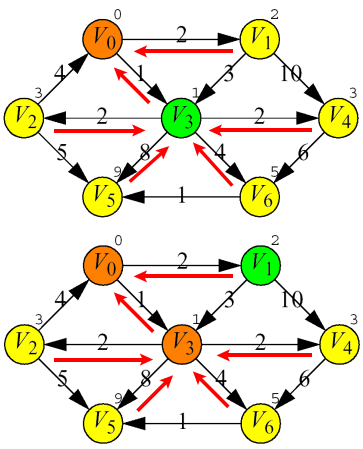
PWSP



THRijswijk

169

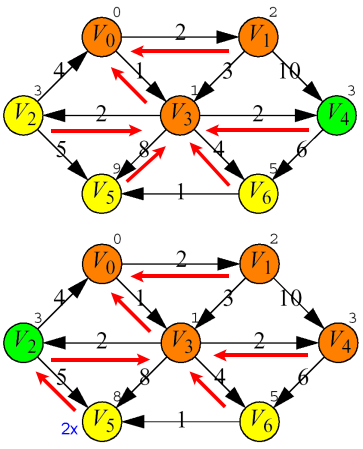
PWSP



THRijswijk

170

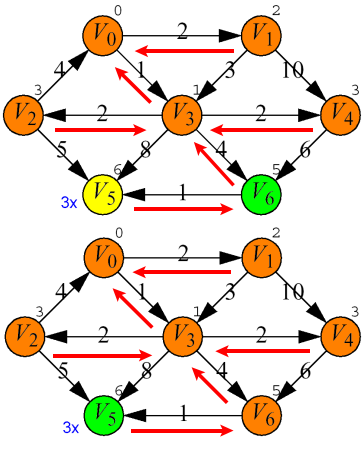
PWSP



THRijswijk

171

PWSP



THRijswijk

172

Shortest-Path

Positive-Weighted

- Optimalisatie:
  - Een Vertex* die al in de priority_queue staat hoeft daar niet nogmaals in te worden geplaatst.
  - De variabele scratch wordt gebruikt om aan te geven of de Vertex in de priority_queue staat (scratch==1).
  - Weiss: maakt scratch = 1 als Vertex uit de priority_queue wordt gehaald. Als je deze Vertex voor de tweede keer uit de queue haalt en scratch == 1 hoef je niets te doen.
  - Weiss heeft een overbodige manier om te kijken of het algoritme klaar is (alle vertices gehad).
- Dijkstra: **O(|E|•log|V|)**

THRijswijk

173

Graph

```
void Graph::dijkstra(const string& name) {
   reset();
   Vertex* start(findVertex(name));
   start->setDistance(0);
   priority_queue<Vertex*, vector<Vertex*>,
                  GreaterDistance> pq;
   pq.push(start);
   while (!pq.empty()) {
      Vertex* v(pq.top());
      pq.pop();
      for (Vertex::const_iterator i(v->begin());
           i!=v->end(); ++i) {
         Vertex* w(i->first);
         int cvw(i->second);
         if (cvw<0)
            throw runtime_error("negative cost!");
         if (w->distance()>v->distance()+cvw) {
            w->setDistance(v->distance()+cvw);
            w->setPrev(v);
            if (w->getScratch()==0) {
               w->scratchInc();
               pq.push(w);
            }
         }
      }
   }
}
```

THRijswijk

174

Shortest-Path

Negative-Weighted

- Bellman-Ford algoritme
- Het is nu niet meer mogelijk om elke vertex maar 1x te "bezoeken".
- Een negative-cost cycle moet herkend worden.
- Telkens als dist van een vertex wordt aangepast moet deze vertex opnieuw "bezocht" worden.
- Als een vertex |V| x bezocht is, is er negative-cost cycle aanwezig.

- Bellman-Ford: **O(|E|•|V|)**

THRijswijk

Shortest-Path

Negative-Weighted

- Zet start vertex in een queue.
  - Zolang queue niet leeg:
    - Haal vertex *v* uit queue.
    - Als *v* |V| keer in de queue heeft gestaan
      - rapporteer negative-cost cycle en stop
    - Voor alle adjacent vertices *w*:
      - Als $dist_w > dist_v + cost_{v,w}$
        - $dist_w = dist_v + cost_{v,w}$
        - Als *w* niet in de queue staat:
          - Zet vertex *w* in de queue.

- Slim gebruik van scratch:
  - bij push: ++scratch
  - bij pop: ++scratch
  - scratch/2 ==> aantal "bezoeken"
  - scratch%2==1 ==> staat in de queue

THRijswijk

Graph

```
void Graph::negative(const string& name) {
   reset();
   Vertex* start(findVertex(name));
   start->setDistance(0);
   queue<Vertex*> q;
   q.push(start); start->scratchInc();
   while (!q.empty()) {
      Vertex* v(q.front());
      q.pop(); v->scratchInc();
      if (v->getScratch()/2>vm.size())
         throw runtime_error("Negative cycle!");
      for (Vertex::const_iterator i(v->begin());
                                    i!=v->end(); ++i) {
         Vertex* w(i->first);
         int cvw(i->second);
         if (w->distance()>v->distance()+cvw) {
            w->setDistance(v->distance()+cvw);
            w->setPrev(v);
            if (w->getScratch()%2==0) {
               q.push(w); w->scratchInc();
            }
            w->scratchInc(); w->scratchInc();
         }
      }
   }
}
```

THRijswijk

Shortest-Path

Acyclic

- Geen cycles.
- Gebruik topological sort.
  - $E_{v,w}$ => $V < W$
  - bereken de indegree van elke vertex (aantal binnenkomende edges)
  - Herhaal tot alle vertices in de queue staan:
    - Zoek vertex *v* met indegree==0
    - plaats v in een queue
    - Voor alle adjacent vertices *w*:
      - --indegree *w*
  - De vertices in de queue zijn nu topological sorted.
- Bij het zoeken naar het kortste pad worden vertices in topological order "bezocht":
  - Elke vertex hoeft maar 1x bezocht te worden.
- Acyclic: **O(|E|)**

THRijswijk

Graph

```
void Graph::acyclic(const string& name) {
   reset();
   Vertex* start(findVertex(name));
   start->setDistance(0);
   queue<Vertex*> q;
   for (VMap::const_iterator itr(vm.begin());
                                 itr!=vm.end(); ++itr) {
      Vertex* v(itr->second);
      for (Vertex::const_iterator i(v->begin());
                                    i!=v->end(); ++i)
         i->first->scratchInc();
   }
   for (VMap::const_iterator itr(vm.begin());
                                 itr!=vm.end(); ++itr) {
      Vertex* v(itr->second);
      if (v->getScratch()==0)
         q.push(v);
   }
// Zie volgende sheet ==>
```

THRijswijk

Graph

```
// vervolg:
   VMap::size_type iterations;
   for (iterations=0; !q.empty(); ++iterations) {
      Vertex* v(q.front());
      q.pop();
      for (Vertex::const_iterator i(v->begin());
                                    i!=v->end(); ++i) {
         Vertex* w(i->first);
         int cvw(i->second);
         w->scratchDec();
         if (w->getScratch()==0) {
            q.push(w);
         }
         if (v->distance()==INFINITY)
            continue;
         if (w->distance()>v->distance()+cvw) {
            w->setDistance(v->distance()+cvw);
            w->setPrev(v);
         }
      }
   }
   if (iterations!=vm.size())
      throw runtime_error("Graph has a cycle!");
}
```

THRijswijk

Shortest-Path

Samenvatting

- Breath-first
  - Unweighted
  - O(|E|)

- Acyclic
  - Weighted, no cycles
  - O(|E|)

- Dijkstra
  - Weighted, no negative edges
  - O(|E|•log|V|)

- Bellman-Ford
  - Weighted
  - O(|E|•|V|)

THRijswijk