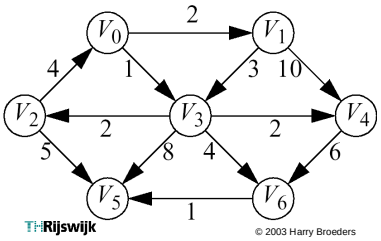
Graph

Definities

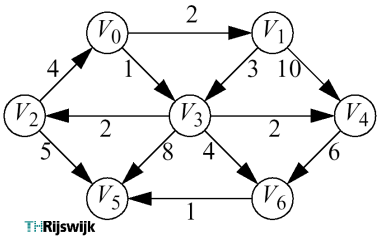
- Graph bestaat uit
  - verzameling **vertices** (nodes) en
  - verzameling **edges** (arcs).
- Edge **verbind** twee virtices.
- In een **directed graph** (digraph) hebben de edges een richting.
- Vertex *v* is **adjacent** to *w* als er een edge is van *v* naar *w*.
- Edge heeft **cost** (of weight).



Graph

Definities

- **Path** is een rijtje vertices verbonden door edges.
- **Path length** = aantal edges in path.
- **Weighted path length** = som van de costs van de edges in path.
- **Cycle** = path van vertix naar zichzelf met lengte > 0.
- **DAG** = **directed acyclic graph** = directed graph zonder cycles.



Graph

Toepassingen

- Computernetwerk (Internet).
- Openbaar vervoer.
- Telefoonnetwerk.

Voor de meeste practische toepassingen geldt dat het aantal edges veel kleiner is dan alle mogelijke verbindingen = **sparce graph**.

Belangrijk algoritme: **bepalen shortest path**.

Graph

Datastructuur

- Matrix |V| x |V|.
  - Elk element m[*a*][*b*] bevat de cost van de edge van *a* naar *b*. Cost = oneindig als edge niet bestaat.
  - Benodigde geheugen **O(|V|²)**
  - Niet geschikt voor sparce graphs.
- Adjacency list.
  - Maak voor elke vertex een lijstje met adjacent vertices en de bijbehorende costs.
  - Elke Vertex bevat een **list< pair<Vertex*, int> > adjacent**
  - Benodigde geheugen **O(|E|)**
  - Gebruik een **map<string, Vertex*>** zodat een Vertex op naam gezocht kan worden.
  - Sla in elke Vertex ook de naam op en enkele variabelen nodig bij het berekenen van het **shortest path**.

Vertex

```
class Vertex {
private:
   typedef list< pair<Vertex*, int> > AList;
   string name;
   AList adjacent;
// gebruikt door shortest-path algoritmes
   Vertex* prev;
   int dist;
   unsigned int scratch;
public:
   Vertex(const string& name);
   void addEdge(Vertex* v, int cost);
// gebruikt door shortest-path algoritmes
   typedef AList::const_iterator const_iterator;
   const_iterator begin() const;
   const_iterator end() const;
   void reset();
   int distance() const;
   void setDistance(int d);
   unsigned int getScratch() const;
   void scratchInc();
   void scratchDec();
   void setPrev(Vertex* v);
   void printPath() const;
};
```


Vertex

```
const int INFINITY=INT_MAX;

Vertex::Vertex(const string& name):
                                    name(name) {
   reset();
}

void Vertex::reset() {
   dist=INFINITY;
   prev=0;
   scratch=0;
}

void Vertex::addEdge(Vertex* v, int cost) {
   adjacent.push_back(make_pair(v, cost));
}

void Vertex::printPath() const {
   if (prev) {
      prev->printPath();
      cout<<" to ";
   }
   cout<<name;
}
```


Graph

```
class Graph {
public:
   Graph();
   ~Graph();
   void addEdge(const string& sourceName,
                const string& destName, int cost);
   void printPath(const string& destName)
                                        const;
   void unweighted(const string& startName);
   void dijkstra(const string& startName);
   void negative(const string& startName);
   void acyclic(const string& startName);
private:
   Vertex* getVertex(const string& n);
   Vertex* findVertex(const string& n) const;
   void reset();
   typedef map<string, Vertex*> VMap;
   VMap vm;
   Graph(const Graph& rhs);
   Graph& operator=(const Graph& rhs);
};
```


Graph

```
Vertex* Graph::getVertex(const string& n) {
   VMap::const_iterator itr(vm.find(n));
   if (itr==vm.end()) {
      Vertex* newv(new Vertex(n));
      vm[n]=newv;
      return newv;
   }
   return itr->second;
}

void Graph::addEdge(const string& source,
                    const string& dest, int cost) {
   Vertex* v(getVertex(source));
   Vertex* w(getVertex(dest));
   v->addEdge(w, cost);
}

Graph::~Graph() {
   for (VMap::const_iterator itr(vm.begin());
                                itr!=vm.end(); ++itr)
      delete itr->second;
}
```


Graph

```
Vertex* Graph::findVertex(const string& n)
                                        const {
   VMap::const_iterator itr(vm.find(n));
   if (itr==vm.end())
      throw runtime_error(n +
                  " is not a vertex in this graph");
   return itr->second;
}

void Graph::printPath(const string& dn) const {
   Vertex* dest(findVertex(dn));
   if (dest->distance()==INFINITY)
      cout<<dn<<" is unreachable";
   else {
      cout<<"(Cost is: "<<dest->distance()<<") ";
      dest->printPath();
   }
   cout<<endl;
}

void Graph::reset() {
   for (VMap::const_iterator itr(vm.begin());
                                itr!=vm.end(); ++itr)
      itr->second->reset();
}
```

Graph

Verschillen met Weiss

- int cost in plaats van double.
- pair<Vertex*, int> in plaats van struct Edge. Zie fig 15.6.
- class Vertex in plaats van struct Vertex.
  - Met memberfuncties en iterator interface om adjacent vertices te benaderen.
  - Zie fig 15.7.
- list< pair<Vertex*, int> > in plaats van vector<Edge>.
- Graph::findVertex toegevoegd in plaats van code steeds te herhalen.
- itr->second in plaats van (*itr).second

THRijswijk

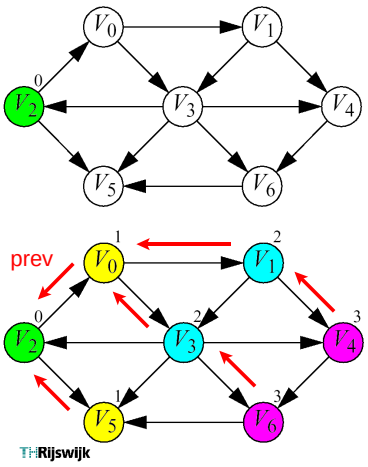
Shortest-Path

Unweighted

- single source algoritme
  - uitgaande van een start vertex wordt het kortste pad naar alle andere vertices berekend.
  - (geldt voor alle SP algoritmen)
- voor elke vertex:
  - dist=INFINITY
- voor start vertex:
  - dist=0
- voor alle vertices met dist==0:
  - voor alle adjacent vertices:
    - if dist==INFINITY dist=1
- voor alle vertices met dist==1:
  - voor alle adjacent vertices:
    - if dist==INFINITY dist=2
- enz...

THRijswijk

USP



THRijswijk

Shortest-Path

Unweighted

- **breadth-first search O(|E|)**
- eerste keer dat dist een waarde krijgt is dit meteen de kortste afstand. Je kunt dan meteen prev pointer goed zetten.
- Volgorde van de te bezoeken vertices kan als volgt bepaald worden:
  - zet start vertex in een queue.
  - zolang queue niet leeg:
    - haal vertex uit queue
    - N=dist
    - voor alle adjacent vertices:
      - als dest==INFINITY.
        - dest=N+1.
        - zet vertex in de queue.

THRijswijk

Graph

```
void Graph::unweighted(const string& name) {
   reset();
   Vertex* start(findVertex(name));
   start->setDistance(0);
   queue<Vertex*> q;
   q.push(start);
   while (!q.empty()) {
      Vertex* v(q.front());
      q.pop();
      for (Vertex::const_iterator i(v->begin());
                                    i!=v->end(); ++i) {
         Vertex* w(i->first);
         if (w->distance()==INFINITY) {
            w->setDistance(v->distance()+1);
            w->setPrev(v);
            q.push(w);
         }
      }
   }
}
```

THRijswijk