09 B: Graph Algorithms II

CS1102S: Data Structures and Algorithms

Martin Henz

March 19, 2010

Generated on Thursday 18th March, 2010, 00:20

1. Review: Graphs, Shortest Path
2. Unweighted Shortest Paths
3. Dijkstra's Algorithm
4. Correctness and Complexity of Dijkstra's Algorithm

Graph, Vertices, Edges

Graph

A *graph* $G = (V, E)$ consists of a set of *vertices*, V, and a set of *edges*, E.

Edge

Each *edge* is a pair (v, w), where $v, w \in V$.

Directed graph

If the pairs are ordered, then the graph is *directed*.

Weight

Sometimes the edges have a third component, knows as either a *weight* or a *cost*. Such graphs are called *weighted graphs*.

Paths

Path

A *path* in a graph is a sequence of vertices $w_1, w_2, w_3, \ldots, w_N$ such that $(w_i, w_{i+1}) \in E$ for $1 \leq i < N$. It is said to lead from w_1 ot w_N.

The Shortest Path Problem

Input

Weighted graph: associated with each edge (v_i, v_j) is a cost $c_{i,j}$ to traverse the edge.

Weighted path length

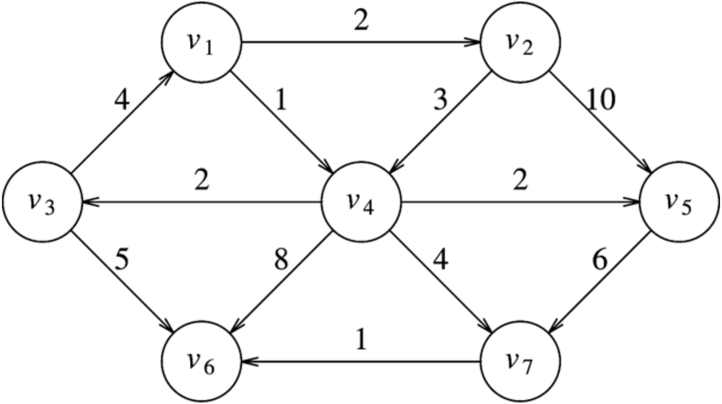
Cost of path $v_1 v_2 \cdots v_N$ is $\sum_{i=1}^{N-1} c_{i,i+1}$.

Single-Source Shortest-Path Problem

Problem

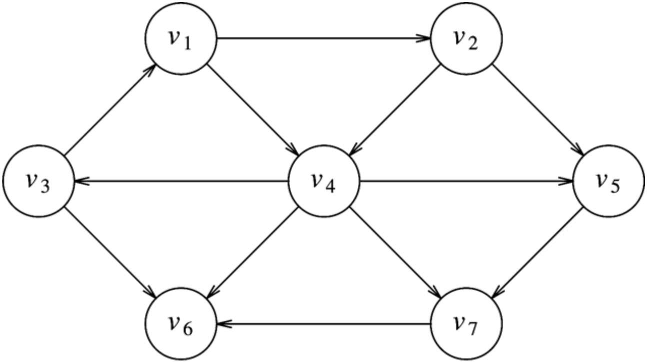
Given as input a weighted graph, $G = (V, E)$, and a distinguished vertex, s, find the shortest weighted path from s to every other vertex in G.

Example



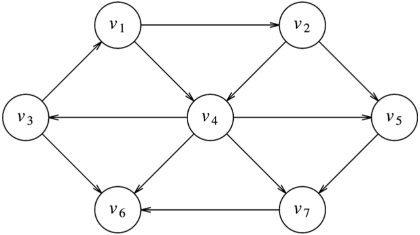
Shortest path from v_1 to v_6
has a cost of 6 and goes from v_1 to v_4 to v_7 to v_6.

Unweighted Shortest Paths: Example



Find the shortest path from v_3 to all other vertices

Idea



Level-order traversal

Start with s (distance 0) and proceed in phases currDist, each time going through all vertices. If vertex is "known" and has distance currDist, set the distance of its neighbors to currDist$+$1.

Implementation

```
void unweighted( Vertex s )
{
    for each Vertex v
    {
        v.dist = INFINITY;
        v.known = false;
    }

    s.dist = 0;

    for( int currDist = 0; currDist < NUM_VERTICES; currDist++ )
        for each Vertex v
            if( !v.known && v.dist == currDist )
            {
                v.known = true;
                for each Vertex w adjacent to v
                    if( w.dist == INFINITY )
                    {
                        w.dist = currDist + 1;
                        w.path = v;
                    }
            }
}
```

Inefficiency

Careless loop

In each phase, we go through all vertices. We can remember the "known" vertices in a data structure.

Suitable data structure

Queue: will contain the vertices in order of increasing distance

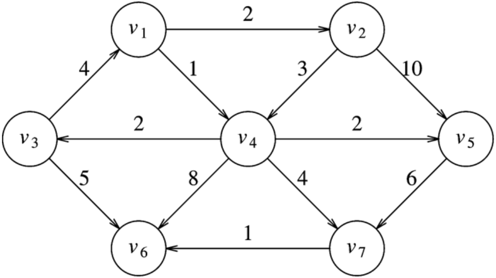
Implementation

```
void unweighted( Vertex s )
{
    Queue<Vertex> q = new Queue<Vertex>( );

    for each Vertex v
        v.dist = INFINITY;

    s.dist = 0;
    q.enqueue( s );

    while( !q.isEmpty( ) )
    {
        Vertex v = q.dequeue( );

        for each Vertex w adjacent to v
            if( w.dist == INFINITY )
            {
                w.dist = v.dist + 1;
                w.path = v;
                q.enqueue( w );
            }
    }
}
```

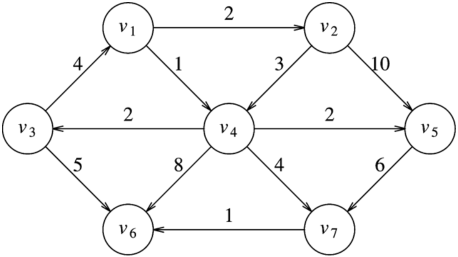
Dijkstra’s Algorithm: Idea

Idea

Treat nodes in the order of shortest distance



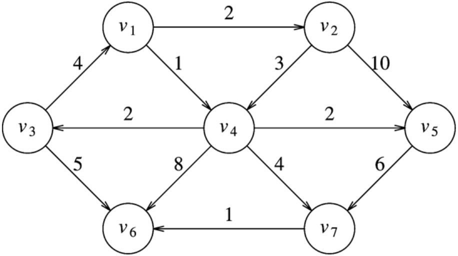
Example



Initial configuration:

v	*known*	d_v	p_v
v_1	F	0	0
v_2	F	∞	0
v_3	F	∞	0
v_4	F	∞	0
v_5	F	∞	0
v_6	F	∞	0
v_7	F	∞	0

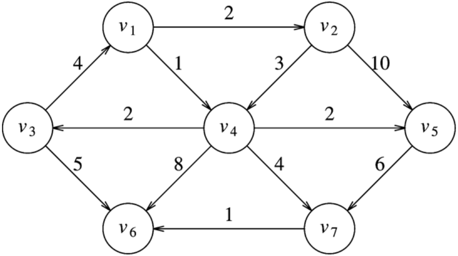
Example



After v_1 is declared known:

v	*known*	d_v	p_v
v_1	T	0	0
v_2	F	2	v_1
v_3	F	∞	0
v_4	F	1	v_1
v_5	F	∞	0
v_6	F	∞	0
v_7	F	∞	0

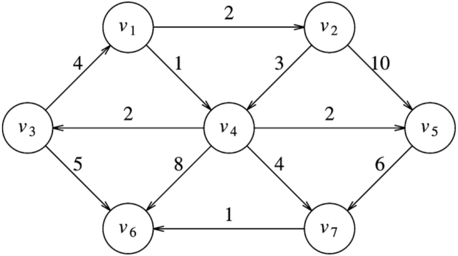
Example



After v_4 is declared known:

v	*known*	d_v	p_v
v_1	T	0	0
v_2	F	2	v_1
v_3	F	3	v_4
v_4	T	1	v_1
v_5	F	3	v_4
v_6	F	9	v_4
v_7	F	5	v_4

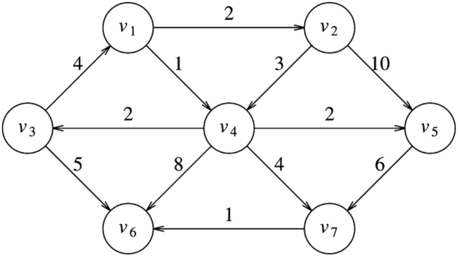
Example



After v_2 is declared known:

v	*known*	d_v	p_v
v_1	T	0	0
v_2	T	2	v_1
v_3	F	3	v_4
v_4	T	1	v_1
v_5	F	3	v_4
v_6	F	9	v_4
v_7	F	5	v_4

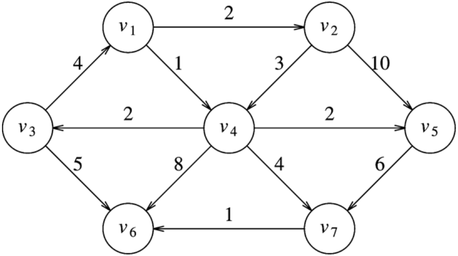
Example



After v_5 and then v_3 are declared known:

v	*known*	d_v	p_v
v_1	T	0	0
v_2	T	2	v_1
v_3	T	3	v_4
v_4	T	1	v_1
v_5	T	3	v_4
v_6	F	8	v_3
v_7	F	5	v_4

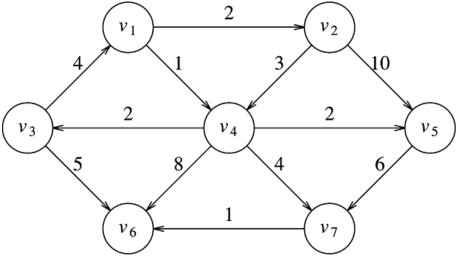
Example



After v_7 is declared known:

v	*known*	d_v	p_v
v_1	T	0	0
v_2	T	2	v_1
v_3	T	3	v_4
v_4	T	1	v_1
v_5	T	3	v_4
v_6	F	6	v_7
v_7	T	5	v_4

Example



After v_6 is declared known:

v	*known*	d_v	p_v
v_1	T	0	0
v_2	T	2	v_1
v_3	T	3	v_4
v_4	T	1	v_1
v_5	T	3	v_4
v_6	T	6	v_7
v_7	T	5	v_4

Pseudocode for Dijkstra's Algorithm

```
void dijkstra( Vertex s )
{
    for each Vertex v
    {
        v.dist = INFINITY;
        v.known = false;
    }

    s.dist = 0;

    for( ; ; )
    {
        Vertex v = smallest unknown distance vertex;
        if( v == NOT_A_VERTEX )
            break;
        v.known = true;

        for each Vertex w adjacent to v
            if( !w.known )
                if( v.dist + cvw < w.dist )
                {
                    // Update w
                    decrease( w.dist to v.dist + cvw );
                    w.path = v;
                }
    }
```

Shortest Subpath

Lemma

Any subpath of a shortest path must also be a shortest path.

Proof

By contradiction: Assume that a subpath $p = v_i \cdots v_j$ of the shortest path $q = v_1 \cdots p \cdots v_k$ is not a shortest path. Then there is a shorter path p' from v_i to v_j. Plug p' into q to get $q' = v_1 \cdots p' \cdots v_k$ shorter than q!

Definition of Shortest Distance

Notation

We use the notation

$$\delta(v, w)$$

to denote the length of the shortest path from v to w.

Distance

We call $\delta(v, w)$ the *distance* between v and w.

dist is a Relaxation

Lemma

At any point in time and for any vertex v, we have

$$v\,.\,dist \geq \delta(s, v)$$

Proof Idea

We show this by proving that whenever we set $v\,.\,\mathrm{dist}$ to a finite value, there exists a path of that length.

dist is a Relaxation

Lemma

At any point in time and for any vertex v, we have

$$v.dist \geq \delta(s, v)$$

Proof

By induction over the number of iterations of outer loop.

Start: claim holds for s (distance 0) and all other vertices (distance ∞)

Hypothesis: claim holds for previous iterations.

Induction step: Updates are done such that an edge weight is added to a previously computed dist value

Order of Adding Vertices

Observation

Vertices are becoming known in order of increasing `dist` values.

Observation

Once a vertex becomes known, its `dist` value does not change.

Correctness of Dijkstra's Algorithm

```
void dijkstra( Vertex s )
{
    for each Vertex v
    {
        v.dist = INFINITY;
        v.known = false;
    }

    s.dist = 0;

    for( ; ; )
    {
        Vertex v = smallest unknown distance vertex;
        if( v == NOT_A_VERTEX )
            break;
        v.known = true;

        for each Vertex w adjacent to v
            if( !w.known )
                if( v.dist + cvw < w.dist )
                {
                    // Update w
                    decrease( w.dist to v.dist + cvw );
                    w.path = v;
                }
    }
```

Main Correctness Lemma

Lemma

When we set v.known = **true** then v. dist $= \delta(s, v)$.

Correctness of Dijkstra's algorithm

Each iteration through the outer loop makes one vertex known. At the end every vertex v is known and thus, according to the lemma, its dist value is $\delta(s, v)$.

Proving the Main Lemma

Proof by contradiction

Assume that there is a vertex v for which the claim does not hold.
Therefore, using relaxation property, when we set v.known = true then v.dist $> \delta(s, v)$.
Then, there must be a vertex u for which this is the case for the *first time* in the algorithm.

Situation

Analysis

Situation just before u.known = true: The value u.dist reflects the length of the path given by u.path.

The real shortest path

The real shortest path from s to u is shorter than u.dist. Consider the real shortest path $s \cdots u$.

Situation

Jumping into "unknown"

Since u.known still false, and s.known=true, there must be a pair of two neighboring vertices r and t in the real shortest path such that r.known=true and t.known=false.

First pair

There must be a first pair x and y where this is the case.

Analysis of x and y

Processing of x

We have processed x, but not yet y. Since y's dist value is decreased by decrease(...), we know that

$$y.dist \le x.dist + c_{x,y}$$

Using hypothesis

Since x becomes known earlier than u, we have

$$x.dist = \delta(s, x)$$

Shortest subpath

$s \cdots xy$ is subpath of shortest path, thus

$$\delta(s, y) = \delta(s, x) + c_{x,y} = x.dist + c_{x,y}$$

Recap

We have

$$y.\mathit{dist} \leq x.\mathit{dist} + c_{x,y}$$

and

$$\delta(s, y) = \delta(s, x) + c_{x,y} = x.\mathit{dist} + c_{x,y}$$

and thus: $y.\text{dist} \leq \delta(s, y)$
and therefore $y.\text{dist} = \delta(s, y)$

Finale

Since $y\,.\,\text{dist} = \delta(s, y)$, we have $y \neq u$.

Edge costs between y and u are non-negative, thus

$$\delta(s, y) \leq \delta(s, u)$$

and thus

$$y\,.\,dist = \delta(s, y) \leq \delta(s, u) < u\,.\,dist$$

If $y\,.\,\text{dist} < u\,.\,\text{dist}$, and since vertices become known in order of increasing distance, y would have become known before u, which contradicts the assumption that u is next vertex to become known!