09 A: Graph Algorithms I

CS1102S: Data Structures and Algorithms

Martin Henz

March 17, 2010

Generated on Tuesday 16th March, 2010, 22:51

1. Definitions
   - Graphs
   - Representation of Graphs
2. Topological Sort
3. Shortest-Path Algorithms

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.

Path length

The *length* of a path is the number of edges on the path, namely $N - 1$.

Path with no edges

For any vertex v, a path with no edges always leads from v to v itself.

Loops

Loop

An edge (v, v) from a vertex to itself is called a *loop*.

Usually no loops

The graphs we consider here do not have loops.

Simple path

A *simple path* is a path such that all vertices are distinct, except that the first and last could be the same.

Cyclic and Acyclic Graphs

Cycle in a directed graph

A *cycle* in a directed graph is a path of length at least 1 such that $w_1 = w_N$; the cycle is *simple* if the path is simple.

Cycle in an undirected graph

A *cycle* in an undirected graph is a path of length at least 1 such that $w_1 = w_N$ and all edges are distinct.

Directed acyclic graph

A directed graph is acyclic if it has no cycles; it is called a *DAG*.

Connected Graphs

Connected undirected graph

An undirected graph is *connected* if there is a path from every vertex to every other vertex.

Strongly connected directed graph

A directed graph is *strongly connected* if there is a path from every vertex to every other vertex.

Weakly connected directed graph

A directed graph that is not strongly connected is called *weakly connected* if its undirected version is connected.

Adjacency Matrix

Adjacency matrix

A directed graph can be represented using a two-dimensional boolean array A by setting $A[u][v]$ to true if and only if $(u, v) \in E$.
A weighted directed graph can be represented using a two-dimensional array A whose type is the same as the weight type, by A by setting $A[u][v]$ to the weight of the corresponding edge, using an unambiguous default value if there is no edge.

Space requirement

The space requirement of an adjacency matrix is $\Theta(|V|^2)$. This works well if $|E| = \Theta(|V|^2)$. Such graphs are called *dense*.

Adjacency List

Sparse graphs

Graphs that are not *dense* are called *sparse*.

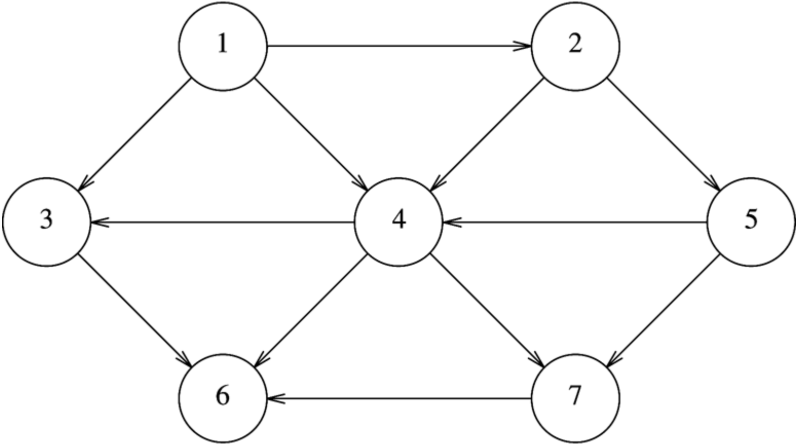
Example

Streets running east-west/north south.

Adjacency list

A directed graph can be represented using an array of lists, containing, for each vertex, all adjacent vertices.

Example Graph



Adjacency List

1	2, 4, 3
2	4, 5
3	6
4	6, 7, 3
5	4, 7
6	(empty)
7	6

The Problem

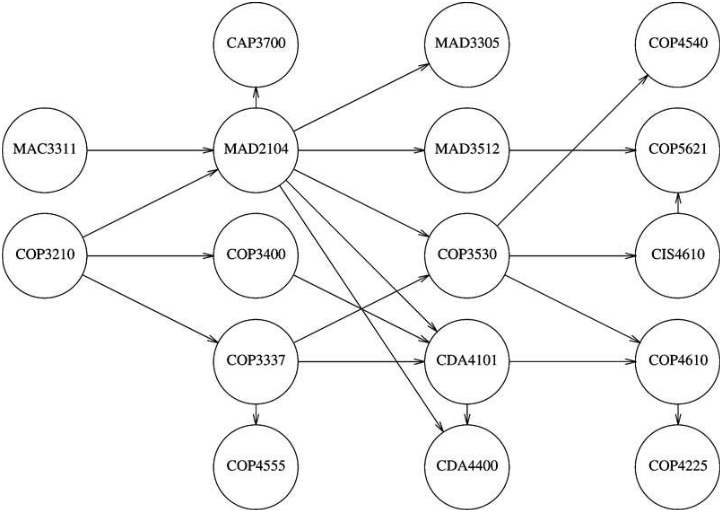
Topological sort

A *topological sort* is an ordering of vertices in a directed acyclic graph, such that if there is a path from v_i to v_j, then v_j appears *after* v_i in the ordering.

Example: In what order to take modules?

Module prerequisites can be represented by edges in a directed graph

Example


CAP3700
MAD3305
COP4540
MAC3311
MAD2104
MAD3512
COP5621
COP3210
COP3400
COP3530
CIS4610
COP3337
CDA4101
COP4610
COP4555
CDA4400
COP4225

A Simple Algorithm

Idea

Find any vertex with no incoming edges. Print the vertex, remove its edges, and find the next vertex with no incoming edges.

Indegree

The *indegree* of a vertex v is the number of edges of the form (u, v).

Implementation

```
void topsort( ) throws CycleFoundException
{
    for( int counter = 0; counter < NUM_VERTICES; counter++ )
    {
        Vertex v = findNewVertexOfIndegreeZero( );
        if( v == null )
            throw new CycleFoundException( );
        v.topNum = counter;
        for each Vertex w adjacent to v
            w.indegree--;
    }
}
```

Runtime Analysis

Setup

Initially, we compute the indegree of each vertex.

Quadratic runtime

Each call of findNewVertexOfIndegreeZero requires $O(|V|)$ time. There are $|V|$ calls, thus overall $O(|V|^2)$.

A Smarter Algorithm

Observation

After each iteration, we visit every vertex; quite wasteful.

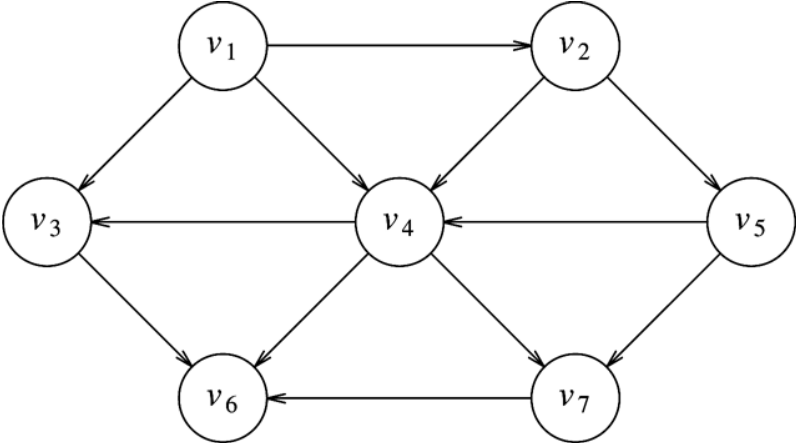
Idea

Learn from previous operations by keeping track of the indegrees and the vertices with indegree 0.

Implementation

```
void topsort( ) throws CycleFoundException
{
    Queue<Vertex> q = new Queue<Vertex>( );
    int counter = 0;

    for each Vertex v
        if( v.indegree == 0 )
            q.enqueue( v );

    while( !q.isEmpty( ) )
    {
        Vertex v = q.dequeue( );
        v.topNum = ++counter;  // Assign next number

        for each Vertex w adjacent to v
            if( --w.indegree == 0 )
                q.enqueue( w );
    }

    if( counter != NUM_VERTICES )
        throw new CycleFoundException( );
}
```

Example



Example

	Indegree Before Dequeue #						
Vertex	1	2	3	4	5	6	7
v_1	0	0	0	0	0	0	0
v_2	1	0	0	0	0	0	0
v_3	2	1	1	1	0	0	0
v_4	3	2	1	0	0	0	0
v_5	1	1	0	0	0	0	0
v_6	3	3	3	3	2	1	0
v_7	2	2	2	1	0	0	0
Enqueue	v_1	v_2	v_5	v_4	v_3, v_7		v_6
Dequeue	v_1	v_2	v_5	v_4	v_3	v_7	v_6

1. Definitions
2. Topological Sort
3. Shortest-Path Algorithms
   - The Problem and Some Variants
   - Unweighted Shortest Paths
   - Dijkstra's Algorithm

The 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 $\sum_{i=1}^{N-1} c_{i,i+1}$.

Unweighted path length

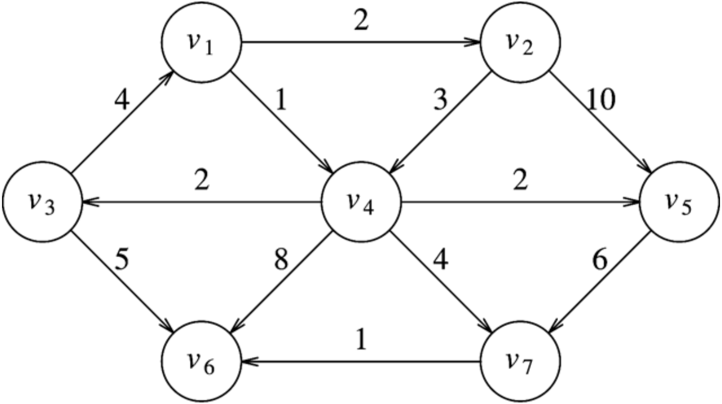
Length of path $v_1 v_2 \cdots v_N$ is $N - 1$.

Single-Source Shortest-Path Problem

Problem

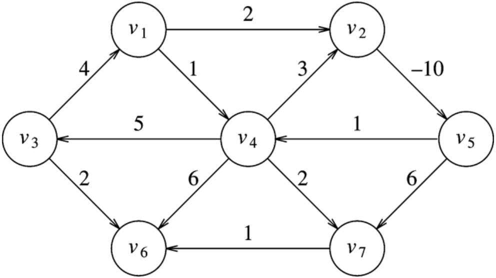
Given as input a weighted graph, $G = (V, E)$, and a distinguished vertex, s, find the shortest weighted path from s to every other vertex in G.

Example



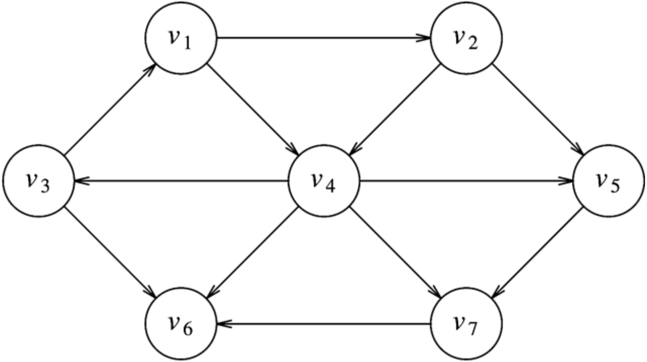
Shortest path from v_1 to v_6
has a cost of 6 and goes from v_1 to v_4 to v_7 to v_6.

Negative Cost Cycles



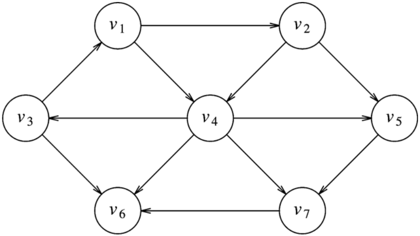
path from v_5 to v_4 has cost 1
but a shorter path exists:
v_5 to v_4 to v_2 to v_5 to v_4
This is a *negative cost cycle*

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

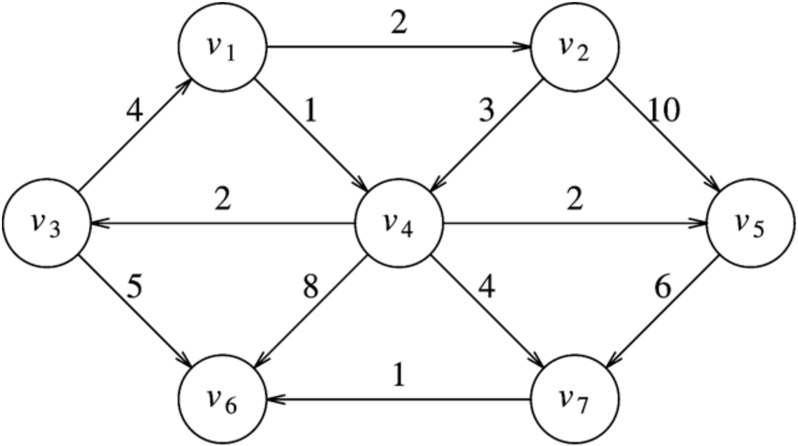
Idea

Similar to level-order traversal; treat nodes in the order of shortest distance

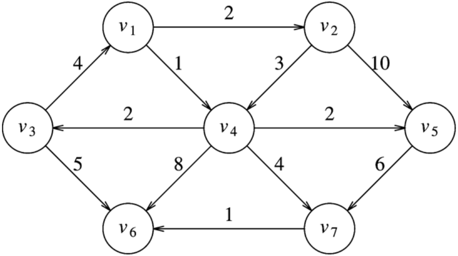
Greedy algorithm

Dijkstra's algorithm is an example of a class of algorithms that exploit that a local property is at the same time a global property

Example



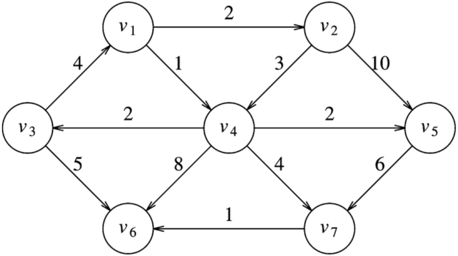
Example



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

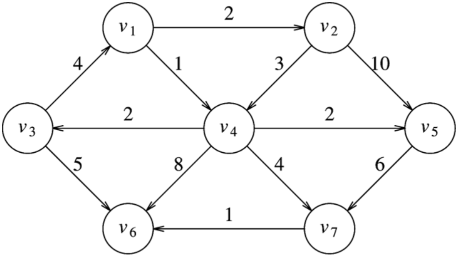
Example



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

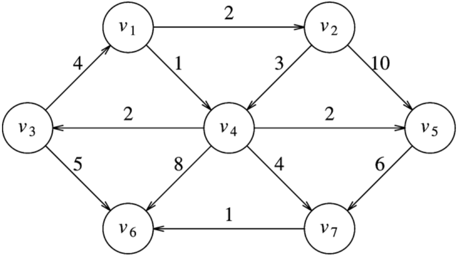
Example



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

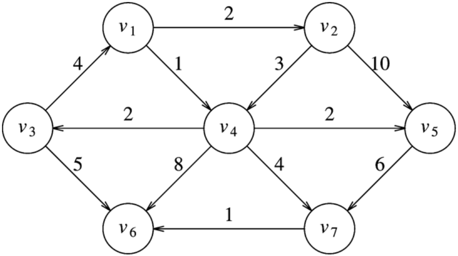
Example



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

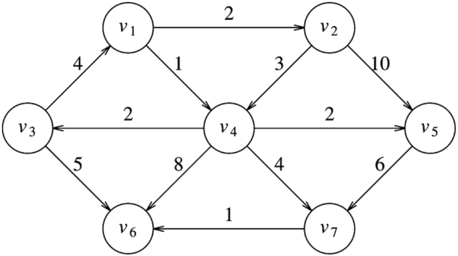
Example



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

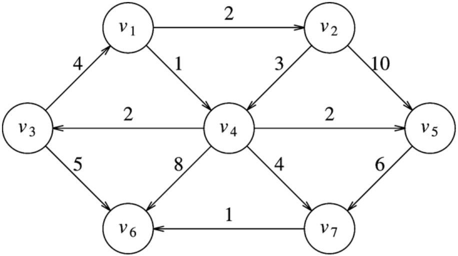
Example



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

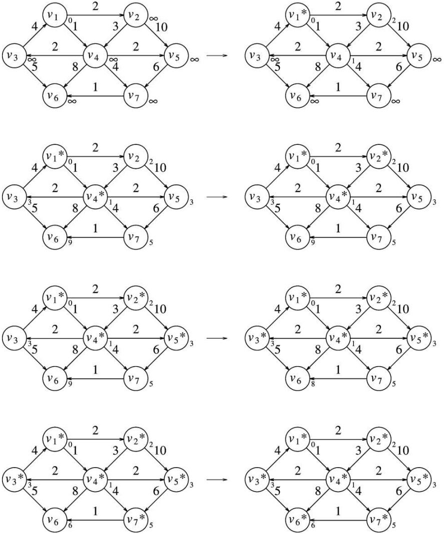
Example



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

Summary: Stages of Dijkstra's Algorithm



Data Structure for Vertices

```
class Vertex
{
    public List     adj;      // Adjacency list
    public boolean  known;
    public DistType dist;     // DistType is probably int
    public Vertex   path;
        // Other fields and methods as needed
}
```

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;
                }
    }
}
```

Complexity

Naive implementation

Scan all vertices sequentially to find unknown vertex with minimum d_v: $O(|V|)$. Thus overall: $O(|V|^2)$

Priority queue for unknown vertices

Runtime can be reduced to $O(|E| \log |V|)$.

This Week

- Thursday tutorials: Midterm 2 and lab tasks 2
- Friday Lecture: Graph Algorithms II
    - Minimum spanning tree
    - Euler paths