CS310 - Advanced Data Structures and Algorithms

Spring 2014– Class 24

May 6, 2014

Shortest Paths With Negative Costs

- Dijkstra assumes no negative edge costs.
- In fact, it may not work with even a single negative edge. Why?
- A whole cycle that gives a negative cost makes it impossible to determine best paths for all cases.
- Why?
- It turns out a few negative edges can be tolerated and the shortest paths found using another algorithm known as the Bellman-Ford algorithm.

Bellman-Ford's algorithm

- Naturally it takes additional computation to compensate for the possibility of negative costs, so this algorithm does not perform as well as Dijkstra.
- But it is actually easier to implement, because it only uses a simple queue, not a priority queue.
- Every time a D is reduced, the target node is scheduled for future exploration by being put back on the queue, unless it's there already.

Bellman-Ford Pseudocode

```
 Given a directed graph G and source node S.
Set up a queue for a distance for each node, Di:
For each node i, make Di be INFINITY
Made Ds be 0
Enqueue S
Loop while queue is not empty
  Dequeue a node, v.
  Loop though nodes w adjacent to v.
      Dw = min(Dw, Dv + cvw)
      If v decreased w's weight and w is not in queue
          enqueue w.
```

Weiss' Implementation

The "scratch" field is used like this:

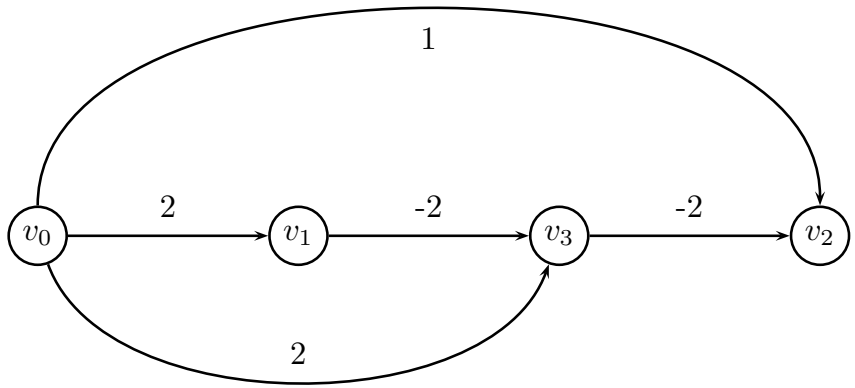
- scratch = 0: not yet seen, not on q
- scratch = 1: seen once as w or start node, on q
- scratch = 2: visited once as eyeball, not on q
- scratch = 3: visited once as eyeball, then later enqueued as w, on q
- scratch = 4: visited twice as eyeball, not on q
- ...
- scratch = odd number iff node is on q, and scratch/2 = number of times it has been an eyeball.

The rule is that it should be done before being an eyeball $|V|$ times, so if this number gets too high, it means a negative cycle was found in the graph.

Example

C_{ij}:

	V_0	V_1	V_2	V_3
V_0	∞	2	1	2
V_1	∞	∞	∞	-2
V_2	∞	∞	∞	∞
V_3	∞	∞	-2	∞



- Although locally it looks good get to V_2 from V_0 at cost 1, there is a better path $V_0 \rightarrow V_1 \rightarrow V_3 \rightarrow V_2$ because of the negative cost links.
- The algorithm needs to keep going whenever it sees a D being reduced, revisiting that seen node.

Example

- Notation: $V_0(1,0)$: status=scratch=1, D=0
- Init: $q = [V_0(1,0)]$
- Dequeue V_0, now $V_0(2,0)$, look at neighbors, enqueue them: $q = [V_1(1,2), V_2(1,1), V_3(1,2)]$
- Dequeue V_1, now $V_1(2,2)$, look at neighbors, just $V_3(1,2)$, but now it has better D=0 via V_1,
- so would enqueue V_3 except that it's on the queue, as $V_3(1,0) : q = [V_2(1,1), V_3(1,0)]$

Example

- Dequeue V_2, now $V_2(2, 1)$, look at neighbors, none, $q = [V_3(1, 0)]$
- Dequeue V_3, now $V_3(2, 0)$, look at neighbors, just V_2, has better D=-2 via V_3, enqueue V_2, $q = [V_2(3, -2)]$
- Dequeue V_2, now $V_2(4, -2)$, look at neighbors, none, q = [] so DONE.

Example

- Result: look for last $V_i(x, D)$'s:
  $V_0(D = 0), V_1(D = 2), V_2(D = -2), V_3(D = 0)$.
- An implementation with JGraphT would have a Map from V to DistInfo with status (instead of scratch), D, and prev for paths, as well as a Queue as in Weisss code on pg. 554.
- Runtime?

Summary on Shortest Path Algorithms

Type of Problem	Big-O (assuming $\|E\| \geq \|V\|$)	Algorithm
Unweighted	$O(\|E\|)$	BFS
Weighted, no negative	$O(\|E\|log\|V\|)$	Dijkstra
Weighted, negative	$O(\|E\| * \|V\|)$	Bellman-Ford
Weighted, acyclic	$O(\|E\|)$	Topological sort (DFS or by in-degrees)

Cycle Detection by DFS

- (not in Weiss): While doing DFS, visiting a current node, look for an outbound edge to an "ancestor" vertex, a vertex on path from start node to the current node.
- Clearly if we see an edge to an ancestor vertex that completes a cycle that started from that ancestor vertex, and follows down the tree to our current node.
- Notice when while a node is being visited, all the nodes visited until we backtrack to it must be its descendants.

Cycle Detection by DFS

- The harder part is to prove we catch all cycles this way.
- Proof that cycle touched by DFS will have a "back arc" to an ancestor vertex.
- Consider the first vertex on the cycle visited by the DFS, and the inbound cycle edge to that vertex.
- The source vertex of that edge is reachable from the start node, so will be visited by the DFS, and then that edge will be a back edge.

Transitive Closure of a Directed Graph

- The transitive closure of a graph G has an edge from A to B if there is a path in G from vertex A to vertex B, for all pairs of vertices.
- The famous algorithm for transitive closure is Warshall's algorithm (not in Weiss)
- We can first consider the first vertex V_1 as a stepping stone from A to B, looking for 2-edge paths from A to V_1 to B, and for any, fill in edges for them in the transitive closure, i.e. add the one edge $A \rightarrow B$, then use V_2 as a stepping stone, and so on.

Transitive Closure of a Directed Graph

- It is neatly expressed using adjacency matrices, but can also be implemented using JGraphT graphs, because we have easy access to all connections involving V_1, inbound and outbound.
- We just put them together in all pairs (inbound, outbound) and do addEdge on the resulting (in-neighbor, out-neighbor) pair.

Warshall's Algorithm for Transitive Closure

This takes Graph g and turns it into its own transitive closure.

```
 for (V v:  g.vertexSet) {
    // v now a stepping stone:
    // look at 2-edge paths with v in the middle
    for (E e1:  g.edgesOf(v))
        for (E e2:  g.edgesOf(v))
            if (g.getEdgeTarget(e1).equals(v) &&
            g.getEdgeSource(e2).equals(v))
            // uses v as stepping stone
                g.addEdge(g.getEdgeSource(e1),
                g.getEdgeTarget(e2));
}
```

Runtime

- This is potentially $O(|V| * |V|^2) = O(|V|^3)$, but usually much less because of sparsity.
- If dense graphs are in use, the adjacency matrix formulation becomes attractive, and the matrix version of Warshall's algorithm is a triple loop over $A[i][j] = A[i][j]||(A[i][k]\&\&A[k][j])$, so $O(|V|^3)$.
- Can be used to find all-pairs shortest paths.
- It is also possible to use DFS to compute the transitive closure, because DFS(v) explores everything reachable by paths from v. You have to do a DFS for each vertex in the graphs.

Undirected Graphs

- As in directed graphs, a path is a sequence of vertices with edges between neighbors in the sequence.
- The notion of adjacent vertex is also similarly one edge away from the given node.
- With undirected graphs, we can say the first and last vertices on a path are connected by the path.
- It's a symmetric relationship: a is connected to b iff b is connected to a.
- A graph is connected if every pair of vertices in it is connected.

Undirected Graphs

- A connected component of a graph is a maximal subset of nodes that themselves form a connected graph.
- A cycle needs to have at least 3 edges in an undirected graph, vs. 1 in a directed graph, where an edge can loop back to the same node it started from.
- A cycle in an undirected graph is a path with at least 3 edges where the first and last nodes are the same, and otherwise the nodes are all different.

Undirected Graphs

- An adjacency matrix can be used for an undirected graph, and it's symmetric, with 0's down the diagonal, since no edge loops back to the same node.
- However, for the same reasons we gave for digraphs, the workhorse implementation is the adjacency list representation, as in SimpleGraph1.java.

Minimum Spanning Tree

- Given a connected graph, we often want the cheapest way to connect all the nodes together, and this is obtainable by a minimum cost spanning tree.
- A spanning tree is a tree that contains all the nodes in the graph.
- The total cost is just the sum of the edge costs.
- The minimum spanning tree is the spanning tree whose sum of edge cost is minimal among all spanning trees.

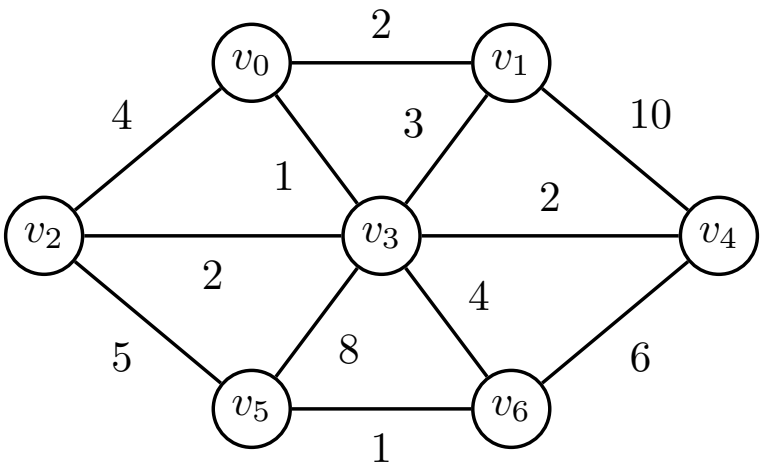
Minimum Spanning Tree

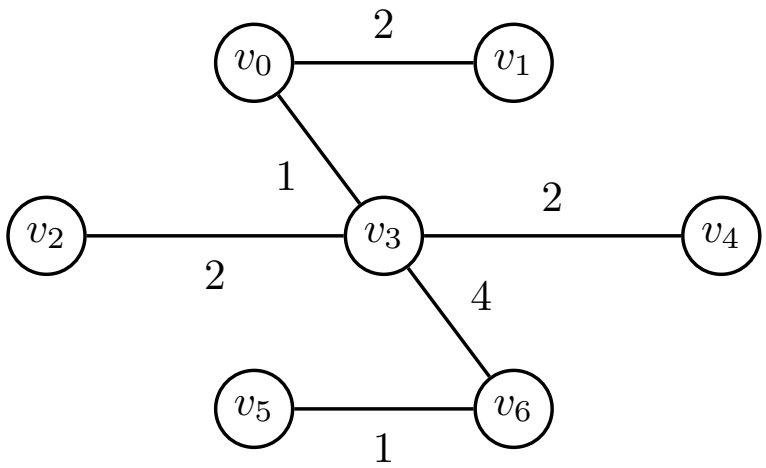
- The kind of tree involved here has no particular root node, and such trees are called free trees, because they are not tied down by a root.
- Weiss relates the min-cost spanning tree problem to the Steiner tree problem.
- If you want to see an example of a Steiner tree, follow this link: `http://www.cs.sunysb.edu/~algorith/files/steiner-tree.shtml`

Kruskal's algorithm

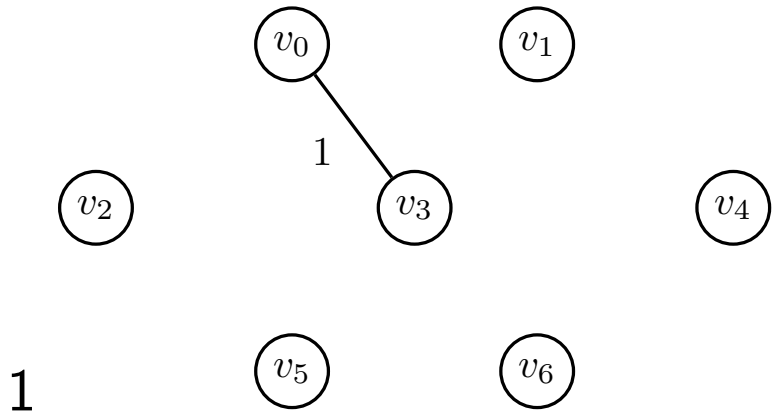
- This algorithm is a lot like Huffman's.
- Start with all the nodes separate and then stick them together incrementally, using a greedy approach that turns out to give you the optimal solution.
- Set up a partition, a set of sets of nodes, starting with one node in each set.
- Then find the minimal edge to join two sets, and use that as a tree edge, and join the sets.

Example

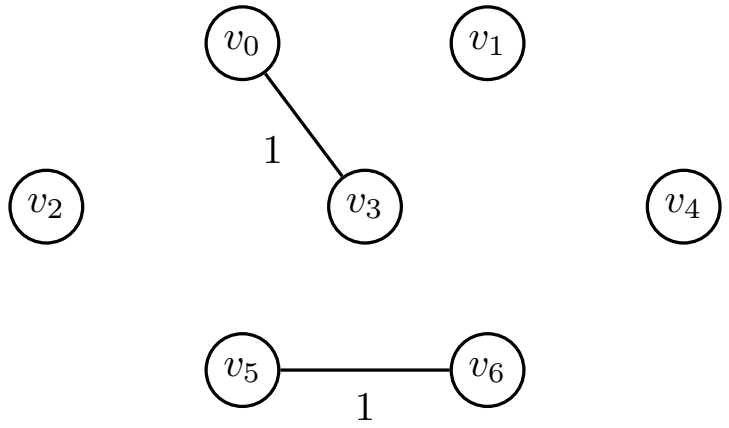




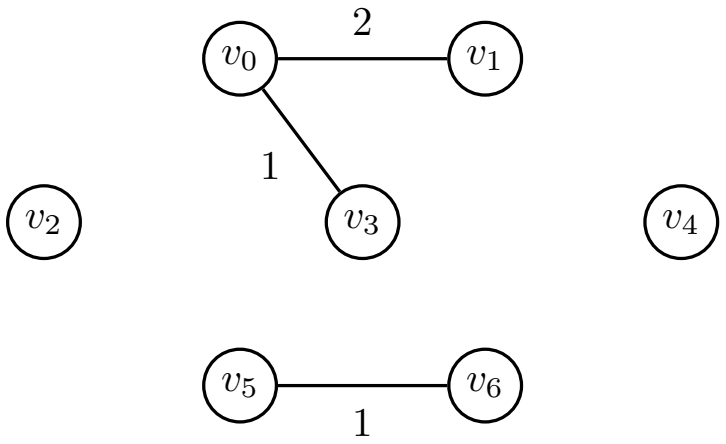
Example



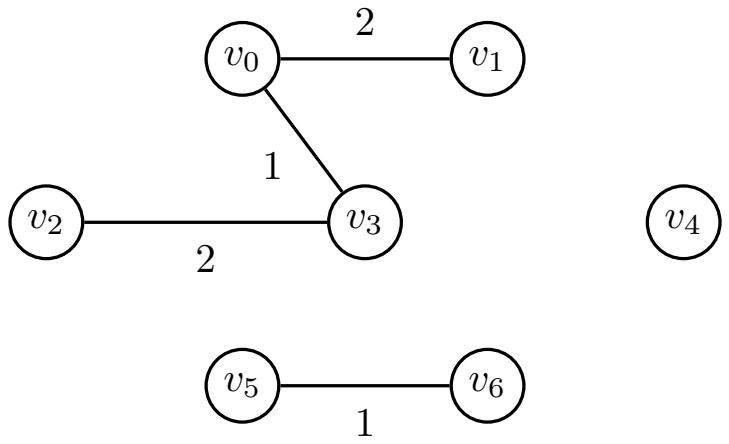
1



2



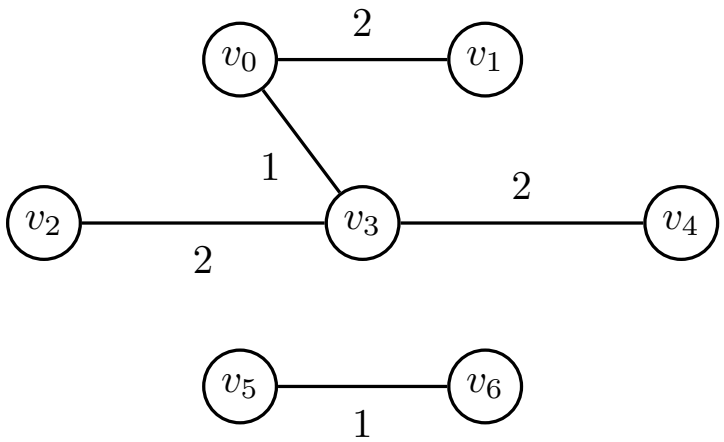
3



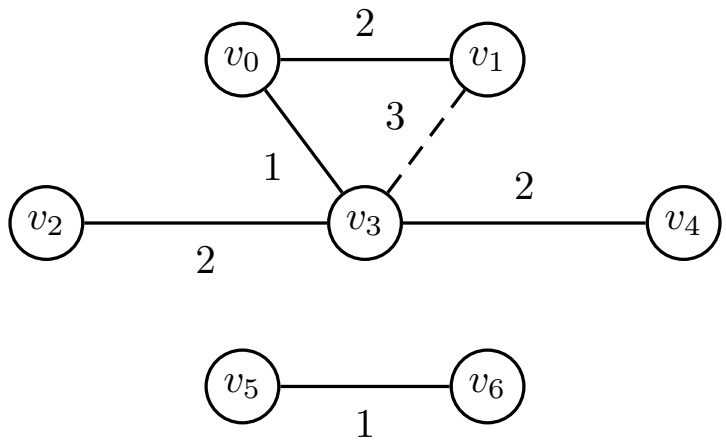
4

Example

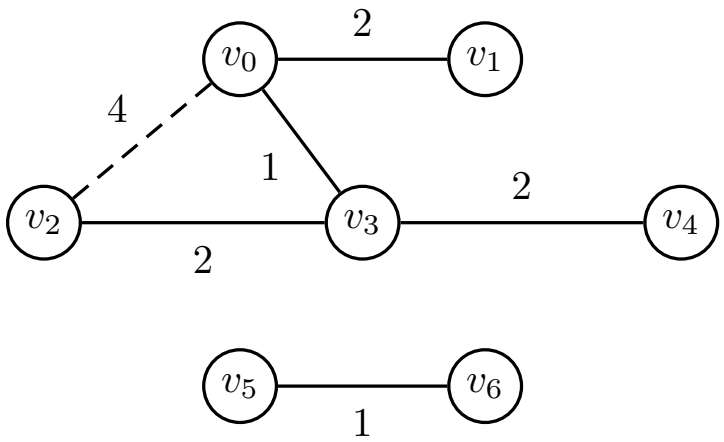
5



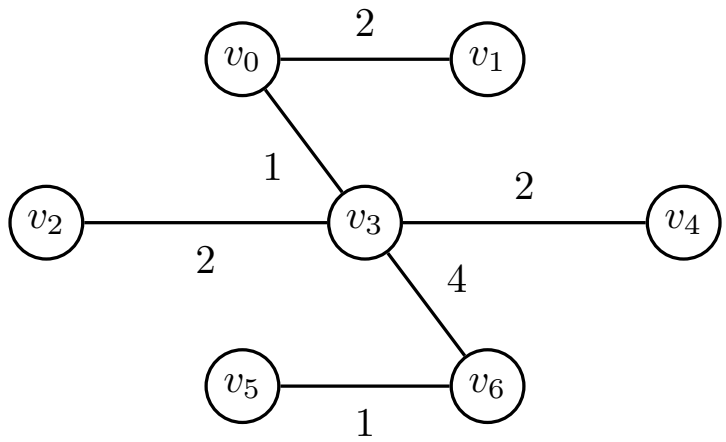
6



7



8



Kruskal's Algorithm

- Note that several edges are left unprocessed in the edge list: these are the high-cost edges we were hoping to avoid using.
- Implementation: There are fancy data structures for partitions.
- Come back to this chapter if you ever need to do this with good performance.
- In cases that dont need the very best performance, a Map from vertex to partition number will do the job of holding the partition.

Running Example

Here are the partition numbers for the first few steps:

Node	V_0	V_1	V_2	V_3	V_4	V_5	V_6	
Set	0	1	2	3	4	5	6	
V_0-V_3	0	1	2	0	4	5	6	(3s turned into 0s)
V_5-V_6	0	1	2	0	4	5	5	(6s turned into 5s)
V_0-V_1	0	0	2	0	4	5	5	
V_2-V_3	0	0	0	0	4	5	5	

- and so on. Edges are used in cost order.
- If an edge has both to and from vertices with the same partition number, it is skipped.
- The resulting tree is defined by the edges used.

Prim's algorithm for MST

Quite similar to Kruskal:

```
 Initialize tree V=v s.t.  v is an arbitrary node.
Initialize E={}.
Repeat until all nodes are in T:
    Choose an edge e=(v,w) with minimum
    weight such that v is in T and w is not.
    Add w to T, add e to E.
Output T=(V,E)
```