

# COMP202

## Complexity of Algorithms

### Graphs

[See Chapters 6 and 7 in Goodrich and Tamassia.]

# Learning Outcomes

At the conclusion of this set, and the next set, of lecture notes students will

1. Be familiar with some basic graph theory terminology (have received a review of it);
2. Understand and be able to use *Dijkstra's algorithm* for finding shortest paths in (weighted) graphs and digraphs (something that they have seen before in COMP 108);
3. Comprehend the *Maximum Flow Problem*, the classical Ford-Fulkerson (augmenting path) algorithm for finding maximum flows in directed graphs (flow networks), and be familiar with the Max Flow/Min Cut theorem.

# Connectivity information

*Connectivity information* can be defined by many kinds of relationships that exist between pairs of objects.

For example, connectivity information is present in city maps, where the objects are roads, and also in the routing tables for the Internet, where the objects are computers.

# Connectivity information

*Connectivity information* can be defined by many kinds of relationships that exist between pairs of objects.

For example, connectivity information is present in city maps, where the objects are roads, and also in the routing tables for the Internet, where the objects are computers.

Connectivity information is also present in the parent-child relationships defined by a binary tree, where the objects are tree nodes.

# Connectivity information

*Connectivity information* can be defined by many kinds of relationships that exist between pairs of objects.

For example, connectivity information is present in city maps, where the objects are roads, and also in the routing tables for the Internet, where the objects are computers.

Connectivity information is also present in the parent-child relationships defined by a binary tree, where the objects are tree nodes.

*Graphs* are one way in which connectivity information can be stored, expressed, and utilized.

# Graphs

A *graph* is a set of objects, called *vertices* (or *nodes*), together with a collection of pair-wise connections, called *edges*, between them.

# Graphs

A *graph* is a set of objects, called *vertices* (or *nodes*), together with a collection of pair-wise connections, called *edges*, between them.

Graphs have applications in a number of different domains, including

- ▶ *mapping* (geographic information systems);
- ▶ *transportation* (road and flight networks);
- ▶ *electrical engineering* (circuit design);
- ▶ *process scheduling* (job makespans and assembly-line scheduling); and
- ▶ *computer networking* (connectivity of networks).

## Graphs (cont.)

More formally, a graph  $G = (V, E)$ , is a set,  $V$ , of *vertices* and a collection,  $E$ , of pairs of vertices from  $V$ , called *edges*.

## Graphs (cont.)

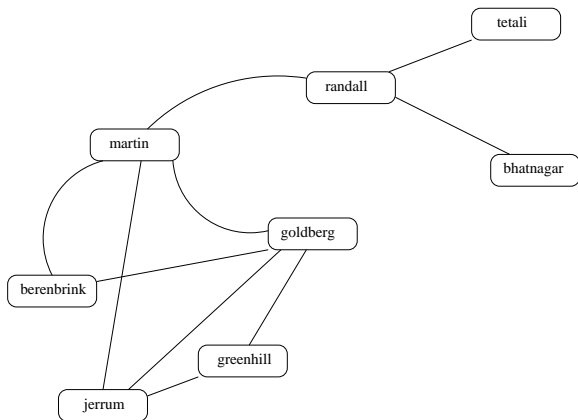
More formally, a graph  $G = (V, E)$ , is a set,  $V$ , of *vertices* and a collection,  $E$ , of pairs of vertices from  $V$ , called *edges*.

Edges in a graph are either *directed* or *undirected*.

- ▶ An edge  $(u, v)$  is said to be *directed* from  $u$  to  $v$  if the pair  $(u, v)$  is ordered. If all edges of a graph are directed, we usually refer to  $G$  as a *digraph*.
- ▶ An edge  $(u, v)$  is said to be *undirected* if the pair  $(u, v)$  is unordered. Typically, undirected edges are written as  $\{u, v\}$  (using braces instead of parentheses).

# Graph - Example

Graph of co-authorship



# Graph Terminology

Two vertices are said to be *adjacent* if they are end-points of the same edge.

An edge is said to be *incident* to a vertex if the vertex is one of its end-points.

An *outgoing edge* of a vertex is a directed edge whose origin is that vertex.

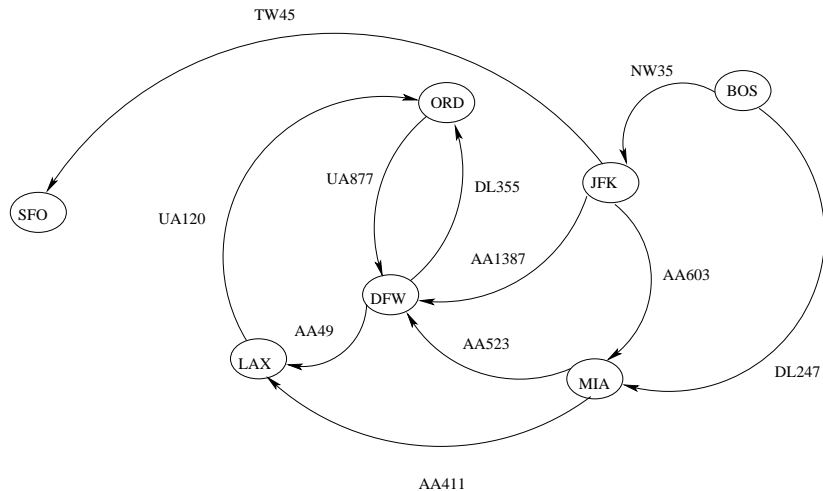
An *incoming edge* of a vertex is a directed edge whose destination is that vertex.

## Graph terminology (cont.)

The *degree* of a vertex  $v$ , denoted  $\text{deg}(v)$ , is the number of edges incident to  $v$ .

In a directed graph, the *in-degree* (*out-degree*) of a vertex  $v$  is the number of *incoming* (*outgoing*) edges of  $v$ , and is denoted by  $\text{indeg}(v)$  ( $\text{outdeg}(v)$ ).

# Digraph - Example



## Graphs (cont.)

We have the following two elementary results about graphs.

**Theorem:** If  $G$  is an undirected graph with  $m$  edges then

$$\sum_{v \in V} \text{deg}(v) = 2m.$$

**Theorem:** If  $G$  is a directed graph with  $m$  edges, then

$$\sum_{v \in V} \text{indeg}(v) = \sum_{v \in V} \text{outdeg}(v) = m.$$

## Graphs (cont.)

An undirected graph  $G$  is said to be *simple* if there is at most one edge between each pair of vertices  $u$  and  $v$ .

A digraph is *simple* if there is at most one directed edge from  $u$  to  $v$  for every pair of distinct vertices  $u$  and  $v$ .

**Theorem:** Let  $G$  be a *simple* graph with  $n$  vertices and  $m$  edges.

- ▶ If  $G$  is *undirected*, then  $m \leq \frac{n(n-1)}{2}$ .
- ▶ If  $G$  is *directed*, then  $m \leq n(n-1)$ .

## More graph terminology

A *walk* in a graph is a sequence of alternating vertices and edges, starting at a vertex and ending at a vertex.

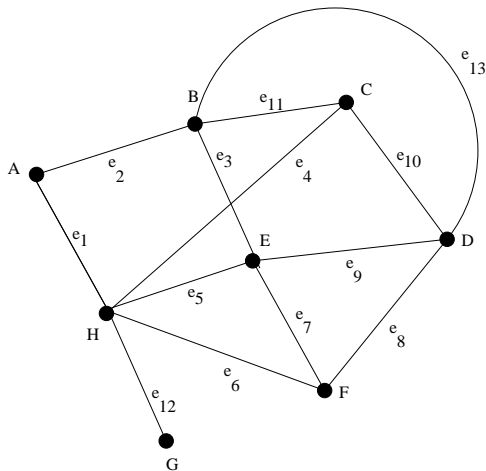
A *path* is a walk where each vertex in the walk is *distinct*.

A *circuit* is a walk with the same start and end vertex.

A *cycle* is a circuit where each vertex in the circuit is *distinct* (except for first and last vertex).

A *directed walk* is a walk in which all edges are directed and are traversed along their direction. Directed paths, circuits, and cycles are defined similarly.

## An example



$A, e_2, B, e_{13}, D, e_{10}, C, e_{11}, B, e_3, E$  is a *walk* in this graph.

$G, e_{12}, H, e_5, E, e_9, D, e_{13}, B$  is a *path* (and also a walk) joining  $G$  and  $B$ .

$B, e_{11}, C, e_{10}, D, e_8, F, e_7, E, e_5, H, e_1, A, e_2, B$  is a *cycle*.

## A small note...

If  $G$  is a simple graph, then giving the sequence of vertices is sufficient to describe a walk, path, circuit, or cycle (as then the edges are implied).

For example, the path joining  $G$  and  $G$  on the previous slide could be (more compactly) represented as

$G, H, E, D, B.$

Similarly the cycle could be written as

$B, C, D, F, E, H, A, B.$

## Still more terminology

A *subgraph* of a graph  $G$  is a graph  $H$  whose vertices and edges are *subsets* of the vertices and edges of  $G$ .

A *spanning subgraph* of  $G$  is a subgraph of  $G$  that contains all the vertices of  $G$ .

A graph is *connected* if, for any two distinct vertices, there is a path between them.

If a graph  $G$  is not connected, its maximal connected subgraphs are called the *connected components* of  $G$ .

## Graphs (cont.)

A *forest* is a graph without cycles.

A *tree* is a *connected forest*, i.e. a connected graph without cycles.

A tree with a distinguished node (*root*) is called a *rooted tree*, otherwise it is called a *free tree* (or, often, simply a *tree*).

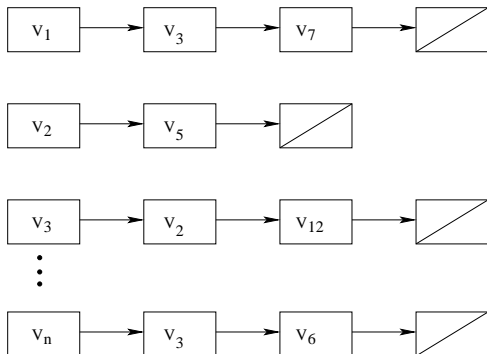
A *spanning tree* of a graph is a spanning subgraph that is a free tree.

## Graphs (cont.)

Let  $G$  be an undirected graph with  $n$  vertices and  $m$  edges. We have the following observations:

- ▶ If  $G$  is *connected*, then  $m \geq n - 1$ .
- ▶ If  $G$  is a tree then,  $m = n - 1$ .
- ▶ If  $G$  is a forest, then  $m \leq n - 1$ .

# Representing Graphs - Adjacency List (linked lists)



# Representing Graphs - Adjacency Matrix

Here we represent the structure of the graph with a  $\{0, 1\}$  matrix, the ones signifying that there is an edge present between the two vertices.

	$V_1$	$V_2$	$V_3$	$V_4$	$V_5$	$V_6$	$V_7$
$V_1$	0	1	1	0	1	0	0
$V_2$	1	0	1	0	1	0	0
$V_3$	1	1	0	0	0	0	1
$V_4$	0	0	0	0	1	1	0
$V_5$	1	1	0	1	0	0	1
$V_6$	0	0	0	1	0	0	0
$V_7$	0	0	1	0	1	0	0

# Digraphs

A *digraph* is a graph whose edges are all directed.

A fundamental issue with directed graphs is the notion of *reachability*, which deals with determining where we can get to in a directed graph.

Given two vertices  $u$  and  $v$  of a digraph  $G$ , we say that  $u$  reaches  $v$  (or  $v$  is reachable from  $u$ ) if  $G$  has a *directed path* from  $u$  to  $v$ .

## Digraphs (cont.)

A digraph  $G$  is *strongly connected* if, for any two distinct vertices  $u$  and  $v$ , we have that  $u$  reaches  $v$ , and  $v$  reaches  $u$ .

A *directed cycle* of  $G$  is a cycle where all the edges are traversed according to their respective directions.

A digraph is *acyclic* if it has no directed cycles.

# Graph search methods

Two common methods for exploring graphs are the *Depth-First Search* (DFS) and *Breadth-First Search* (BFS) methods.

# Graph search methods

Two common methods for exploring graphs are the *Depth-First Search* (DFS) and *Breadth-First Search* (BFS) methods.

As a very brief reminder, DFS starts at a vertex, chooses an edge from that vertex and “walks out as far as possible”, finding new vertices until it encounters one it has already seen, then it backs up (as little as possible) to find other un-encountered vertices.

# Graph search methods

Two common methods for exploring graphs are the *Depth-First Search* (DFS) and *Breadth-First Search* (BFS) methods.

As a very brief reminder, DFS starts at a vertex, chooses an edge from that vertex and “walks out as far as possible”, finding new vertices until it encounters one it has already seen, then it backs up (as little as possible) to find other un-encountered vertices.

In contrast to the DFS method, the Breadth First Search algorithm starts at a vertex and first explores the entire neighborhood of that vertex before moving onto another vertex.

## Graph search methods (cont.)

Therefore, the DFS method generates “long, skinny” search trees, while the BFS method generates “short, bushy” ones.

The running time of BFS and DFS is the same, namely  $O(n + m)$ .

## Graph search methods (cont.)

Therefore, the DFS method generates “long, skinny” search trees, while the BFS method generates “short, bushy” ones.

The running time of BFS and DFS is the same, namely  $O(n + m)$ .

Generally speaking, we may utilize a *stack* data structure to control a DFS search, while a *queue* data structure may be used for a BFS search.

## Graph search methods (cont.)

Therefore, the DFS method generates “long, skinny” search trees, while the BFS method generates “short, bushy” ones.

The running time of BFS and DFS is the same, namely  $O(n + m)$ .

Generally speaking, we may utilize a *stack* data structure to control a DFS search, while a *queue* data structure may be used for a BFS search.

We will not go into great detail about these search methods here. Any book on (graph) algorithms should contain more detailed descriptions of these procedures.

# Applications of the search methods

BFS and DFS can be used to answer a variety of questions about graphs including:

- ▶ Testing whether  $G$  is connected.
- ▶ Computing the connected components of  $G$ .
- ▶ Finding a spanning forest of  $G$  (or spanning tree if  $G$  is connected).
- ▶ Searching for a cycle in  $G$ , or reporting that  $G$  is acyclic.
- ▶ Given a start vertex  $x$  of  $G$ , computing, for every vertex  $v$  of  $G$ , a path with the minimum number of edges between  $x$  and  $v$ , or reporting that no such path exists (BFS).
- ▶ Testing for *strong connectivity* of digraphs. (Is there a directed path from  $u$  to  $v$ , for all  $u$  and  $v$  in  $D$ ?)

# Weighted Graphs

A *weighted graph* is a graph that has a numerical label  $w(e)$  associated with each edge  $e$ , called the *weight* of  $e$ .

Alternatively, we might sometimes consider graphs having weights on the *vertices*, or on both the vertices and edges.

# Single-Source Shortest Paths

Often in the case of weighted graphs, we want to consider the following problem:

For some fixed vertex  $v$ , find a *shortest path* from  $v$  to all other vertices  $u \neq v$  in  $G$  (viewing weights on edges as distances between adjacent vertices).

# Single-Source Shortest Paths

Often in the case of weighted graphs, we want to consider the following problem:

For some fixed vertex  $v$ , find a *shortest path* from  $v$  to all other vertices  $u \neq v$  in  $G$  (viewing weights on edges as distances between adjacent vertices).

The *length* (or *weight*) of a path  $P$  is the *sum of the weights of the edges*  $e_1, \dots, e_k$  of  $P$ , i.e.  $w(P) = \sum_{i=1}^k w(e_i)$ .

# Single-Source Shortest Paths

Often in the case of weighted graphs, we want to consider the following problem:

For some fixed vertex  $v$ , find a *shortest path* from  $v$  to all other vertices  $u \neq v$  in  $G$  (viewing weights on edges as distances between adjacent vertices).

The *length* (or *weight*) of a path  $P$  is the *sum of the weights of the edges*  $e_1, \dots, e_k$  of  $P$ , i.e.  $w(P) = \sum_{i=1}^k w(e_i)$ .

The *distance* from a vertex  $u$  to a vertex  $v$  in  $G$ , denoted  $d(u, v)$  is the value of a *minimum length path* (also called a *shortest path*) from  $u$  to  $v$ .

# Single-Source Shortest Paths

Often in the case of weighted graphs, we want to consider the following problem:

For some fixed vertex  $v$ , find a *shortest path* from  $v$  to all other vertices  $u \neq v$  in  $G$  (viewing weights on edges as distances between adjacent vertices).

The *length* (or *weight*) of a path  $P$  is the *sum of the weights of the edges*  $e_1, \dots, e_k$  of  $P$ , i.e.  $w(P) = \sum_{i=1}^k w(e_i)$ .

The *distance* from a vertex  $u$  to a vertex  $v$  in  $G$ , denoted  $d(u, v)$  is the value of a *minimum length path* (also called a *shortest path*) from  $u$  to  $v$ .

This problem is known as the *Single-Source Shortest Path* problem (SSSP for short).

# Greedy Approach to SSSP

There is an interesting approach for solving the SSSP based on the *greedy method*.

# Greedy Approach to SSSP

There is an interesting approach for solving the SSSP based on the *greedy method*.

The main idea in applying the greedy method to SSSP is to perform a “weighted” Breadth First Search.

One algorithm using this design pattern is known as *Dijkstra's algorithm*.

Dijkstra's algorithm can be used in both directed and undirected graphs, with one requirement...

# Dijkstra's algorithm

We assume that all edges in the graph have *non-negative weights*. (This is a requirement for Dijkstra's algorithm to work correctly.)

# Dijkstra's algorithm

We assume that all edges in the graph have *non-negative weights*. (This is a requirement for Dijkstra's algorithm to work correctly.)

Let  $v$  be a *source vertex* and let  $D[u]$  represent the *temporary distance* in  $G$  from  $v$  to  $u$ . Initially we take  $D[v] = 0$  and  $D[u] = +\infty$  for all  $u \neq v$ .

# Dijkstra's algorithm

We assume that all edges in the graph have *non-negative weights*. (This is a requirement for Dijkstra's algorithm to work correctly.)

Let  $v$  be a *source vertex* and let  $D[u]$  represent the *temporary distance* in  $G$  from  $v$  to  $u$ . Initially we take  $D[v] = 0$  and  $D[u] = +\infty$  for all  $u \neq v$ .

At the start of the algorithm, all entries in the array  $D$  are temporary, but after each round of the algorithm one entry in  $D$  becomes *fixed*.

## Edge Relaxation

Assume that  $C$  is a set of vertices for which entries in  $D$  are fixed (i.e. shortest distances between  $v$  and all  $w \in C$  have been found).

Suppose that entry  $D[u]$  was the one fixed in the *most recent round*.

## Edge Relaxation

Assume that  $C$  is a set of vertices for which entries in  $D$  are fixed (i.e. shortest distances between  $v$  and all  $w \in C$  have been found).

Suppose that entry  $D[u]$  was the one fixed in the *most recent round*.

For any vertex  $z$  for which  $D[z]$  is temporary, perform the *edge relaxation*:

- ▶ If  $D[u] + w(\{u, z\}) < D[z]$  then  $D[z] \leftarrow D[u] + w(\{u, z\})$ .

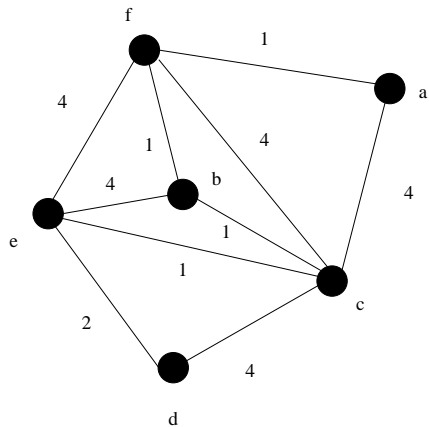
In other words, if we find a better path from  $v$  to  $z$  (through the vertex  $u$ ), we update  $D[z]$  to this smaller value.

## Fixing the next entry of $D$

When the edge relaxation step is completed for all temporarily labeled vertices we then

- ▶ Fix one entry in  $G$  (among vertices still outside  $C$ ) with the smallest weight currently available, and add that new vertex to  $C$ .
- ▶ Proceed to the next stage of edge relaxation based on this extended set of vertices  $C$ .

# Dijkstra's algorithm



D	[	a	b	c	d	e	f	]	C
		$\infty$	$\infty$	$\infty$	0	$\infty$	$\infty$		$\phi$
		$\infty$	$\infty$	4	<b>0</b>	2	$\infty$		{d}
		$\infty$	6	3	<b>0</b>	<b>2</b>	6		{d, e}
		7	4	<b>3</b>	<b>0</b>	<b>2</b>	6		{c, d, e}
		7	<b>4</b>	<b>3</b>	<b>0</b>	<b>2</b>	5		{b, c, d, e}
		6	4	<b>3</b>	<b>0</b>	<b>2</b>	<b>5</b>		{b, c, d, e, f}
		<b>6</b>	<b>4</b>	<b>3</b>	<b>0</b>	<b>2</b>	<b>5</b>		{a, b, c, d, e, f}

# Dijkstra's algorithm pseudo-code

DIJKSTRA( $G, v$ )

- 1  $D[v] \leftarrow 0$
- 2 **for** each  $u \neq v$  **do**
- 3      $D[u] \leftarrow +\infty$
- 4 Let  $Q$  be a priority queue (heap) having all vertices of  $G$  using the  $D$  labels as keys.
- 5 **while** NOTEMPTY( $Q$ ) **do**
- 6      $u \leftarrow$  REMOVEMIN( $Q$ )
- 7     **for** each  $z$  s.t.  $(u, z) \in E$  **do**
- 8         **if**  $D[u] + w(\{u, z\}) < D[z]$
- 9             **then**  $D[z] \leftarrow D[u] + w(\{u, z\})$
- 10              $key(z) \leftarrow D[z]$
- $\triangleright$  ( $z$  might “bubble up” in the heap.)
- 11 **return**  $D$

# Complexity of Dijkstra's algorithm

Let  $G = (V, E)$  where  $|V| = n$  and  $|E| = m$ .

Construction of the initial heap for the set of  $n$  vertices takes time  $O(n \log n)$ .

# Complexity of Dijkstra's algorithm

Let  $G = (V, E)$  where  $|V| = n$  and  $|E| = m$ .

Construction of the initial heap for the set of  $n$  vertices takes time  $O(n \log n)$ .

In each step of the “while” loop, getting the minimum entry from the heap requires  $O(\log n)$  time (to update the heap), and then  $O(\deg(u) \log n)$  time to perform the edge relaxation steps (and update the heap as necessary).

# Complexity of Dijkstra's algorithm

Let  $G = (V, E)$  where  $|V| = n$  and  $|E| = m$ .

Construction of the initial heap for the set of  $n$  vertices takes time  $O(n \log n)$ .

In each step of the “while” loop, getting the minimum entry from the heap requires  $O(\log n)$  time (to update the heap), and then  $O(\deg(u) \log n)$  time to perform the edge relaxation steps (and update the heap as necessary).

So the overall running time of the “while” loop is

$$\sum_{u \in V(G)} (1 + \deg(u)) \log n = O((n + m) \log n).$$

## Complexity of Dijkstra's algorithm (cont.)

Thus we've proven the following result:

**Theorem:** Given a weighted graph with  $n$  vertices and  $m$  edges, each with non-negative weight, Dijkstra's algorithm solves the SSSP in  $O(m \log n)$  time.

(Note: This uses the fact that  $n \leq m$  for a connected graph.)

## Path enumeration - adding predecessor

So far we have seen an implementation of Dijkstra's algorithm which gives us the *length* of the shortest path.

There is no information about the path itself, i.e. what the sequence of edges is that forms the shortest path.

## Path enumeration - adding predecessor

So far we have seen an implementation of Dijkstra's algorithm which gives us the *length* of the shortest path.

There is no information about the path itself, i.e. what the sequence of edges is that forms the shortest path.

We can remedy this by adding a predecessor field to our table for each vertex.

## Path enumeration - adding predecessor

So far we have seen an implementation of Dijkstra's algorithm which gives us the *length* of the shortest path.

There is no information about the path itself, i.e. what the sequence of edges is that forms the shortest path.

We can remedy this by adding a predecessor field to our table for each vertex.

The predecessor,  $P[start]$ , of the *start* vertex is *always*  $\emptyset$ .

## Path enumeration - adding predecessor

So far we have seen an implementation of Dijkstra's algorithm which gives us the *length* of the shortest path.

There is no information about the path itself, i.e. what the sequence of edges is that forms the shortest path.

We can remedy this by adding a predecessor field to our table for each vertex.

The predecessor,  $P[start]$ , of the *start* vertex is *always*  $\emptyset$ .

If  $D[u] + w(\{u, z\}) < D[z]$  then we update the predecessor of  $z$ , namely we set  $P[z] = u$ .

Once an entry becomes fixed for a vertex  $v$ , then  $D[v]$  and  $P[v]$  don't change from this point on.

## What about negative weights?

If the graph  $G$  contains some edges with negative weights, then Dijkstra's algorithm may not give the correct results.

## What about negative weights?

If the graph  $G$  contains some edges with negative weights, then Dijkstra's algorithm may not give the correct results.

Other algorithms, such the Bellman-Ford algorithm may be used if there are negative weights (but in this case, the graph must be directed).

The Floyd-Warshall algorithm is a dynamic programming procedure to compute shortest paths between *all* pairs of vertices in a directed graph.

## Other problems on weighted graphs

We are also often interested in such things as

- ▶ minimum spanning trees (e.g. Kruskal's algorithm or Prim's algorithm);
- ▶ maximum weight independent sets (where the weights are on vertices). This is a difficult problem for general graphs. For paths and trees (and some other classes of graphs), there are efficient algorithms.
- ▶ maximum flows (we will see this next);
- ▶ maximum weight matchings in graphs (especially in *bipartite graphs*, in this case very much related to maximum flows);
- ▶ minimum cost flows.