

Data Structures and Algorithms with Python

Heikki Peura

h.peura@imperial.ac.uk

Lecture 4

Last time

Data structures and object-oriented programming

- ▶ Data structure design affects efficiency — choose right structure
- ▶ Built-in Python structures are good but do not cover everything
- ▶ OOP allows abstraction and modularization of programs

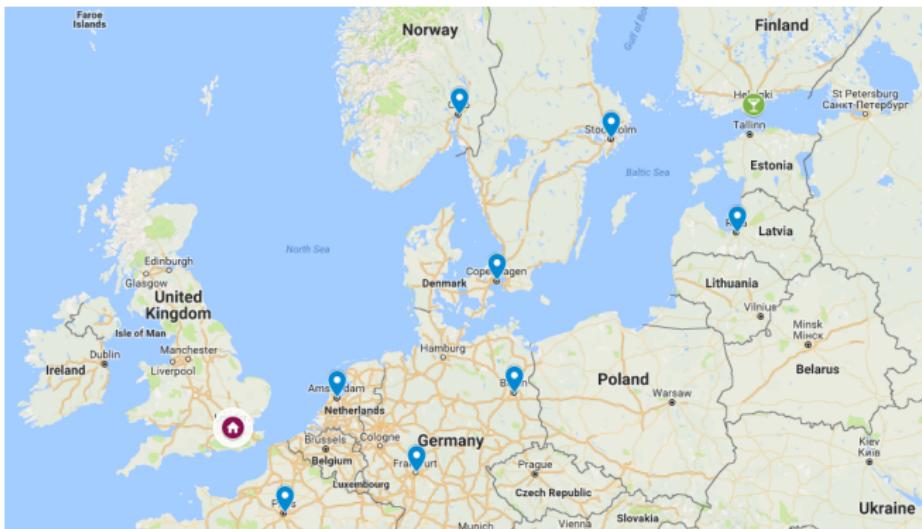
Plan for today:

- ▶ Graphs
- ▶ Graph search

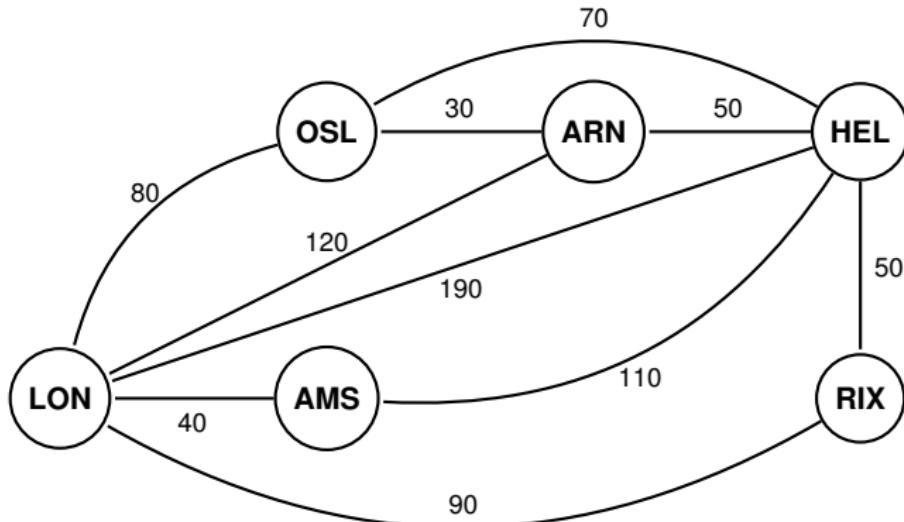
Graph search

Suppose you're travelling around Europe

- ▶ Know **flights** and their **prices**
- ▶ **Goal:** fewest stops between cities, cheapest price, ...?
- ▶ (Assume the price of flying from London to Helsinki via Stockholm is the sum of the two legs...)

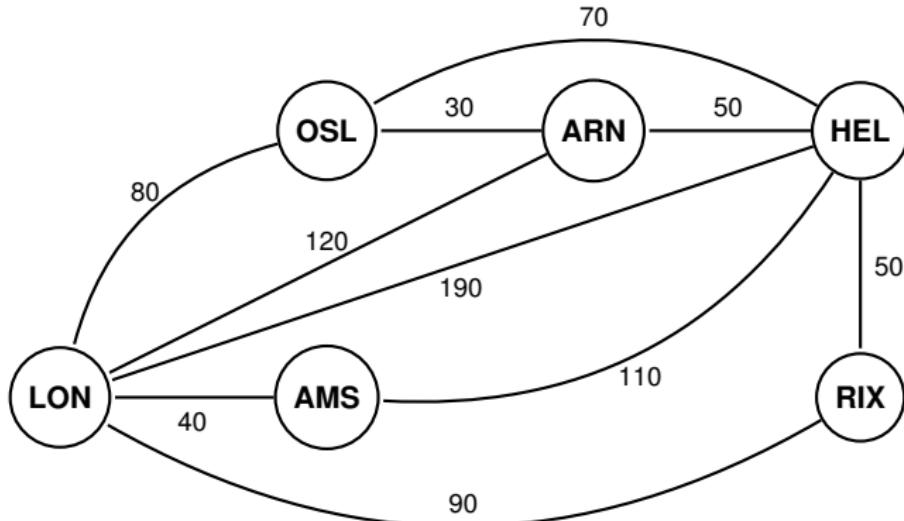


Defining graphs



A set of **nodes** (or vertices) connected by **edges**

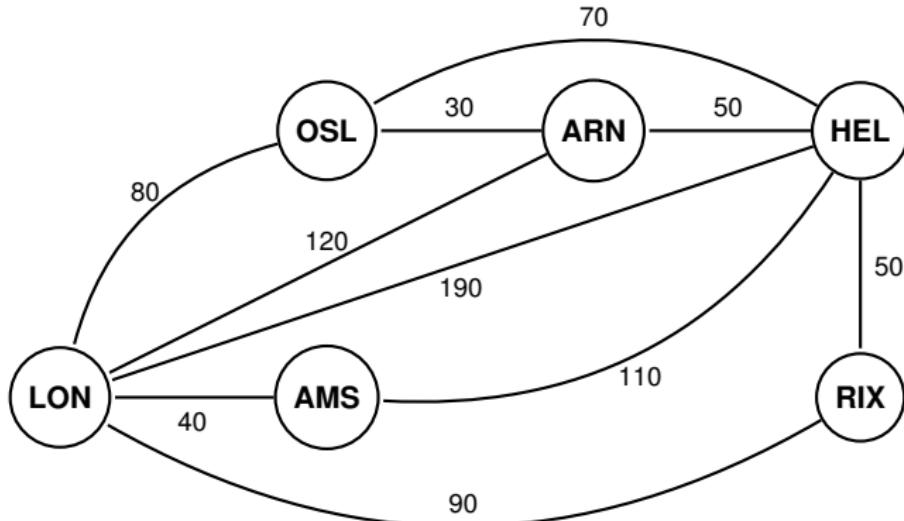
Defining graphs



A set of **nodes** (or vertices) connected by **edges**

Edges may or may not be directed -> **directed** or **undirected** graph
(Twitter vs Facebook?)

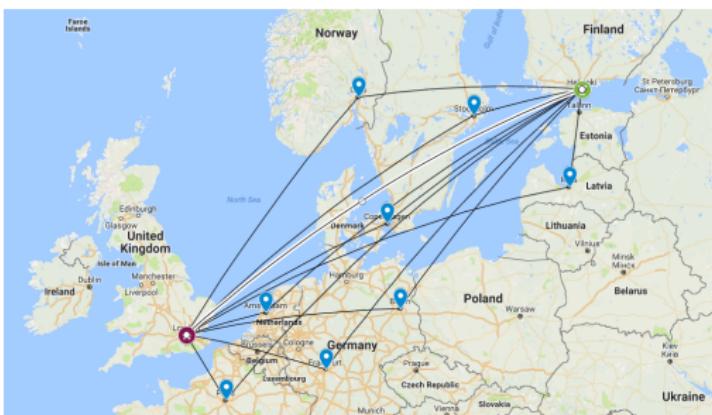
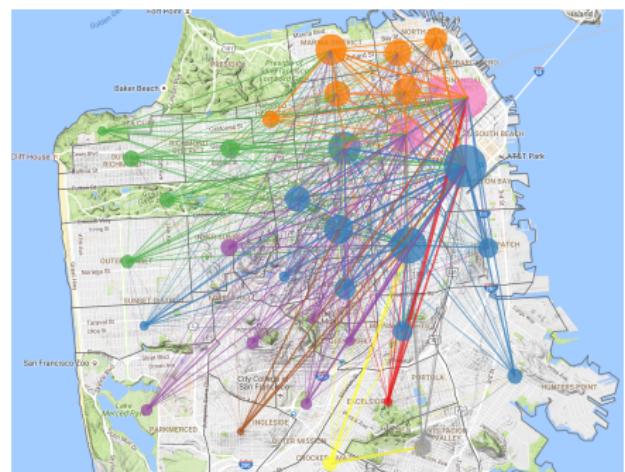
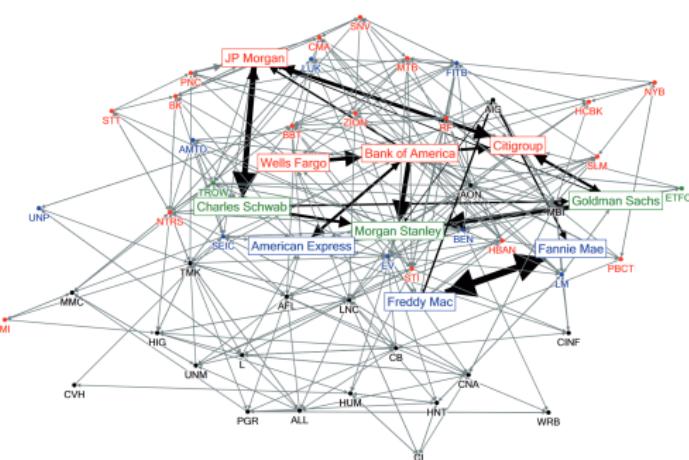
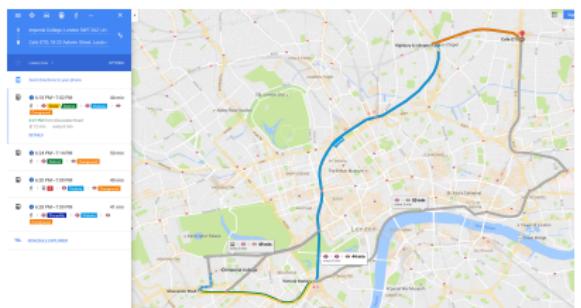
Defining graphs



A set of **nodes** (or vertices) connected by **edges**

Edges may or may not be directed -> **directed** or **undirected** graph
(Twitter vs Facebook?)

Edges may have **weights** (eg prices): weighted graph



Pics: Gmaps, Uber, Hautsch et al.

Graphs are everywhere

Represent situations with **interesting pairwise relationships**

Many practical applications, some not obvious

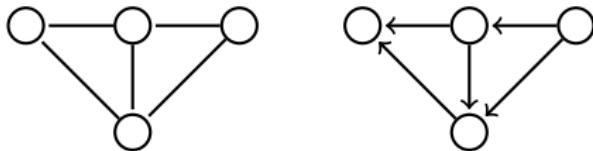
- ▶ Roads, social networks, the web, financial markets, complex projects etc

Need good algorithms!

Graph: collection of vertices and edges

Graph $G = (V, E)$: V = set of n **vertices**, E = set of m **edges**

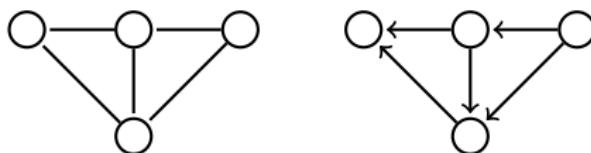
- ▶ Edges can be **directed** or **undirected**
- ▶ **Connected** graph: there is a **path** between any two vertices



Graph: collection of vertices and edges

Graph $G = (V, E)$: V = set of n vertices, E = set of m edges

- ▶ Edges can be **directed** or **undirected**
- ▶ **Connected** graph: there is a **path** between any two vertices

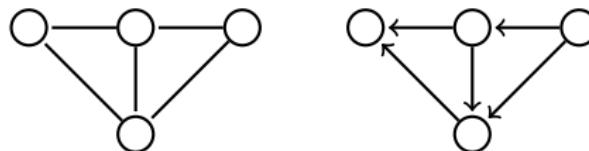


How many edges can there be in a connected graph with n vertices?

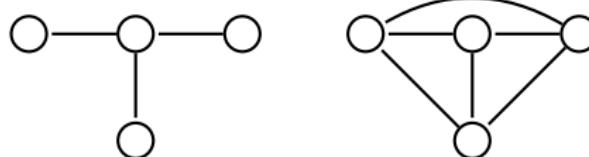
Graph: collection of vertices and edges

Graph $G = (V, E)$: V = set of n vertices, E = set of m edges

- ▶ Edges can be **directed** or **undirected**
- ▶ **Connected** graph: there is a **path** between any two vertices



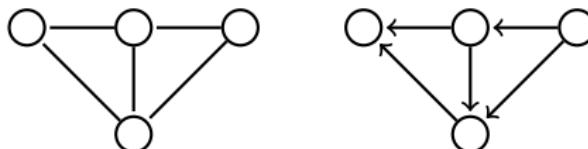
How many edges can there be in a connected graph with n vertices?



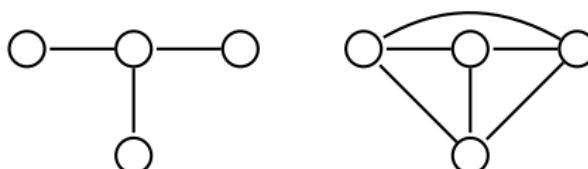
Graph: collection of vertices and edges

Graph $G = (V, E)$: V = set of n vertices, E = set of m edges

- ▶ Edges can be **directed** or **undirected**
- ▶ **Connected** graph: there is a **path** between any two vertices



How many edges can there be in a connected graph with n vertices?



Sparse vs. **dense** graphs:

- ▶ Between $n - 1$ vs $n(n - 1)/2$ edges

Graph: collection of vertices and edges

Graph $G = (V, E)$: V = set of n vertices, E = set of m edges

Graph: collection of vertices and edges

Graph $G = (V, E)$: V = set of n vertices, E = set of m edges

How to represent a graph?

- ▶ **Adjacency matrix**: size $n \times n$
 - ▶ $A_{vw} = 1$ if there is an edge between vertices v, w (unweighted)
 - ▶ n^2 space required

Graph: collection of vertices and edges

Graph $G = (V, E)$: V = set of n vertices, E = set of m edges

How to represent a graph?

- **Adjacency matrix:** size $n \times n$

- $A_{vw} = 1$ if there is an edge between vertices v, w (unweighted)
- n^2 space required

	<i>LON</i>	<i>OSL</i>	<i>ARN</i>	<i>HEL</i>	<i>AMS</i>	<i>RIX</i>
<i>LON</i>	0	1	1	1	1	1
<i>OSL</i>	1	0	1	1	0	0
<i>ARN</i>	1	1	0	1	0	0
<i>HEL</i>	1	1	1	0	1	1
<i>AMS</i>	1	1	0	1	0	0
<i>RIX</i>	1	0	0	1	0	0

Graph: adjacency lists

How to represent a graph?

- ▶ **Adjacency list:** connectivity represented using lists of edges and vertices
 - ▶ List of vertices (each vertex points to edges starting from it); list of edges (each edge points to its endpoints)
 - ▶ Linear space requirement in m, n

Graph: adjacency lists

How to represent a graph?

- ▶ **Adjacency list:** connectivity represented using lists of edges and vertices
 - ▶ List of vertices (each vertex points to edges starting from it); list of edges (each edge points to its endpoints)
 - ▶ Linear space requirement in m, n
- ▶ LON: {OSL,ARN,HEL,AMS,RIX}
- ▶ OSL: {LON,ARN,HEL}
- ▶ ARN: {LON,OSL,HEL}
- ▶ HEL: {LON,OSL,ARN,AMS,RIX}
- ▶ AMS: {LON,HEL}
- ▶ RIX: {LON,HEL}

Graph: adjacency lists

How to represent a graph?

- ▶ **Adjacency list:** connectivity represented using lists of edges and vertices
 - ▶ List of vertices (each vertex points to edges starting from it); list of edges (each edge points to its endpoints)
 - ▶ Linear space requirement in m, n
- ▶ LON: {OSL,ARN,HEL,AMS,RIX}
- ▶ OSL: {LON,ARN,HEL}
- ▶ ARN: {LON,OSL,HEL}
- ▶ HEL: {LON,OSL,ARN,AMS,RIX}
- ▶ AMS: {LON,HEL}
- ▶ RIX: {LON,HEL}

Adjacency matrices vs lists:

- ▶ Dense vs sparse graphs
- ▶ We'll be using adjacency lists

Searching a graph



Small worlds phenomenon:

- ▶ Six degrees of separation (popularized through play by Guare, 1990)
- ▶ Hollywood: Kevin Bacon numbers

Searching a graph

Graph search goal:

- ▶ Find everything findable from given start vertex
- ▶ Don't waste time finding the same thing many times

Searching a graph

Graph search goal:

- ▶ Find everything findable from given start vertex
- ▶ Don't waste time finding the same thing many times

Algorithm (given graph G , vertex s):

- ▶ Initially mark s explored, other nodes unexplored
- ▶ While there are unexplored vertices:
 - ▶ Choose an edge (u, v) with u explored and v unexplored
 - ▶ Mark v explored

Searching a graph

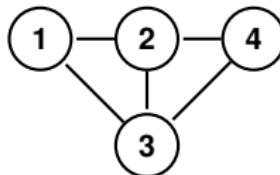
Graph search goal:

- ▶ Find everything findable from given start vertex
- ▶ Don't waste time finding the same thing many times

Algorithm (given graph G , vertex s):

- ▶ Initially mark s explored, other nodes unexplored
- ▶ While there are unexplored vertices:
 - ▶ Choose an edge (u, v) with u explored and v unexplored
 - ▶ Mark v explored

Claim: will explore every v with a path from s to v



Searching a graph

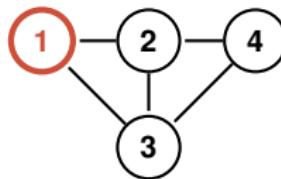
Graph search goal:

- ▶ Find everything findable from given start vertex
- ▶ Don't waste time finding the same thing many times

Algorithm (given graph G , vertex s):

- ▶ Initially mark s explored, other nodes unexplored
- ▶ While there are unexplored vertices:
 - ▶ Choose an edge (u, v) with u explored and v unexplored
 - ▶ Mark v explored

Claim: will explore every v with a path from s to v



Searching a graph

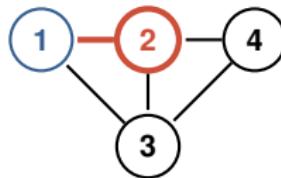
Graph search goal:

- ▶ Find everything findable from given start vertex
- ▶ Don't waste time finding the same thing many times

Algorithm (given graph G , vertex s):

- ▶ Initially mark s explored, other nodes unexplored
- ▶ While there are unexplored vertices:
 - ▶ Choose an edge (u, v) with u explored and v unexplored
 - ▶ Mark v explored

Claim: will explore every v with a path from s to v



Searching a graph

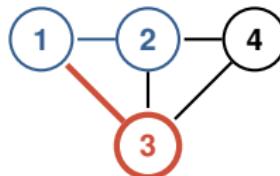
Graph search goal:

- ▶ Find everything findable from given start vertex
- ▶ Don't waste time finding the same thing many times

Algorithm (given graph G , vertex s):

- ▶ Initially mark s explored, other nodes unexplored
- ▶ While there are unexplored vertices:
 - ▶ Choose an edge (u, v) with u explored and v unexplored
 - ▶ Mark v explored

Claim: will explore every v with a path from s to v



Searching a graph

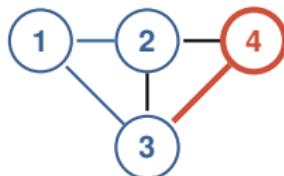
Graph search goal:

- ▶ Find everything findable from given start vertex
- ▶ Don't waste time finding the same thing many times

Algorithm (given graph G , vertex s):

- ▶ Initially mark s explored, other nodes unexplored
- ▶ While there are unexplored vertices:
 - ▶ Choose an edge (u, v) with u explored and v unexplored
 - ▶ Mark v explored

Claim: will explore every v with a path from s to v



How to choose which edge to explore?

Breadth-first search

- ▶ Explore nodes in layers
- ▶ Shortest paths
- ▶ Up next!

Depth-first search

- ▶ Explore aggressively like a maze
- ▶ Eg topological orderings
- ▶ Challenge problem...

Breadth-first search (BFS)

Algorithm: $\text{BFS}(\text{graph } G, \text{ start vertex } s)$

- ▶ **Initialize:** mark all nodes unexplored except s explored
- ▶ Use $Q =$ queue data structure, add s to Q

Breadth-first search (BFS)

Algorithm: BFS(graph G , start vertex s)

- ▶ **Initialize:** mark all nodes unexplored except s explored
- ▶ Use Q = queue data structure, add s to Q

What is a **queue**?

- ▶ First in, first out (just like in a cafe)
- ▶ Remove from front, insert to back in constant time $O(1)$
- ▶ Here we keep a queue of nodes to visit next

Breadth-first search (BFS)

Algorithm: BFS(graph G , start vertex s)

- ▶ **Initialize:** mark all nodes unexplored except s explored
- ▶ Use Q = queue data structure, add s to Q
- ▶ **Main loop:** While Q is not empty:

What is a **queue**?

- ▶ First in, first out (just like in a cafe)
- ▶ Remove from front, insert to back in constant time $O(1)$
- ▶ Here we keep a queue of nodes to visit next

Breadth-first search (BFS)

Algorithm: BFS(graph G , start vertex s)

- ▶ **Initialize:** mark all nodes unexplored except s explored
- ▶ Use Q = queue data structure, add s to Q
- ▶ **Main loop:** While Q is not empty:
 - ▶ Remove the first node of Q and call it v

What is a **queue**?

- ▶ First in, first out (just like in a cafe)
- ▶ Remove from front, insert to back in constant time $O(1)$
- ▶ Here we keep a queue of nodes to visit next

Breadth-first search (BFS)

Algorithm: BFS(graph G , start vertex s)

- ▶ **Initialize:** mark all nodes unexplored except s explored
- ▶ Use Q = queue data structure, add s to Q
- ▶ **Main loop:** While Q is not empty:
 - ▶ Remove the first node of Q and call it v
 - ▶ For each edge (v, w) : if w unexplored:
 - ▶ Mark w explored
 - ▶ Add w to Q

What is a **queue**?

- ▶ First in, first out (just like in a cafe)
- ▶ Remove from front, insert to back in constant time $O(1)$
- ▶ Here we keep a queue of nodes to visit next

BFS example

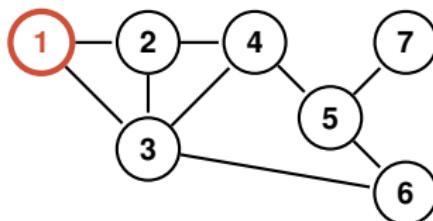
Algorithm: BFS(graph G , start vertex s)

- ▶ **Initialize:** mark all nodes unexplored except s explored
- ▶ Use Q = queue data structure, add s to Q
- ▶ **Main loop:** While Q is not empty:
 - ▶ Remove the first node of Q and call it v
 - ▶ For each edge (v, w) : if w unexplored:
 - ▶ Mark w explored
 - ▶ Add w to Q

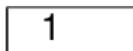
BFS example

Algorithm: BFS(graph G , start vertex s)

- ▶ **Initialize:** mark all nodes unexplored except s explored
- ▶ Use Q = queue data structure, add s to Q
- ▶ **Main loop:** While Q is not empty:
 - ▶ Remove the first node of Q and call it v
 - ▶ For each edge (v, w) : if w unexplored:
 - ▶ Mark w explored
 - ▶ Add w to Q



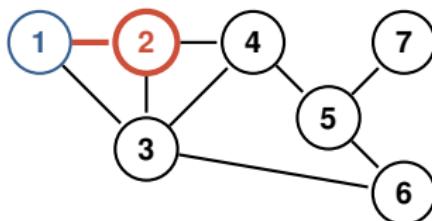
Queue of vertices:



BFS example

Algorithm: BFS(graph G , start vertex s)

- ▶ **Initialize:** mark all nodes unexplored except s explored
- ▶ Use Q = queue data structure, add s to Q
- ▶ **Main loop:** While Q is not empty:
 - ▶ Remove the first node of Q and call it v
 - ▶ For each edge (v, w) : if w unexplored:
 - ▶ Mark w explored
 - ▶ Add w to Q



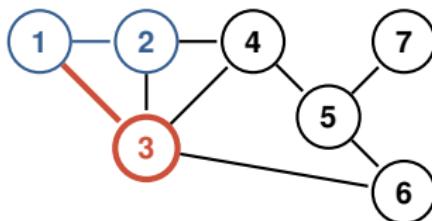
Queue of vertices:

2

BFS example

Algorithm: BFS(graph G , start vertex s)

- ▶ **Initialize:** mark all nodes unexplored except s explored
- ▶ Use Q = queue data structure, add s to Q
- ▶ **Main loop:** While Q is not empty:
 - ▶ Remove the first node of Q and call it v
 - ▶ For each edge (v, w) : if w unexplored:
 - ▶ Mark w explored
 - ▶ Add w to Q



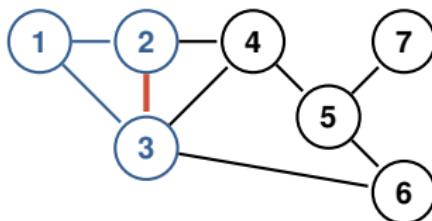
Queue of vertices:

2	3
---	---

BFS example

Algorithm: BFS(graph G , start vertex s)

- ▶ **Initialize:** mark all nodes unexplored except s explored
- ▶ Use Q = queue data structure, add s to Q
- ▶ **Main loop:** While Q is not empty:
 - ▶ Remove the first node of Q and call it v
 - ▶ For each edge (v, w) : if w unexplored:
 - ▶ Mark w explored
 - ▶ Add w to Q



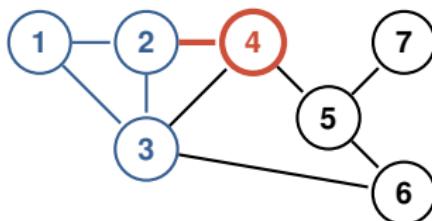
Queue of vertices:

3

BFS example

Algorithm: BFS(graph G , start vertex s)

- ▶ **Initialize:** mark all nodes unexplored except s explored
- ▶ Use Q = queue data structure, add s to Q
- ▶ **Main loop:** While Q is not empty:
 - ▶ Remove the first node of Q and call it v
 - ▶ For each edge (v, w) : if w unexplored:
 - ▶ Mark w explored
 - ▶ Add w to Q



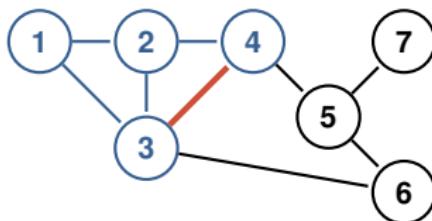
Queue of vertices:

3	4
---	---

BFS example

Algorithm: BFS(graph G , start vertex s)

- ▶ **Initialize:** mark all nodes unexplored except s explored
- ▶ Use Q = queue data structure, add s to Q
- ▶ **Main loop:** While Q is not empty:
 - ▶ Remove the first node of Q and call it v
 - ▶ For each edge (v, w) : if w unexplored:
 - ▶ Mark w explored
 - ▶ Add w to Q



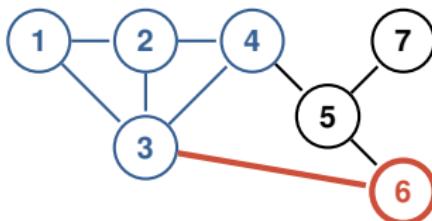
Queue of vertices:

4

BFS example

Algorithm: BFS(graph G , start vertex s)

- ▶ **Initialize:** mark all nodes unexplored except s explored
- ▶ Use Q = queue data structure, add s to Q
- ▶ **Main loop:** While Q is not empty:
 - ▶ Remove the first node of Q and call it v
 - ▶ For each edge (v, w) : if w unexplored:
 - ▶ Mark w explored
 - ▶ Add w to Q



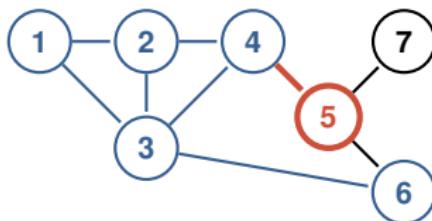
Queue of vertices:

4	6
---	---

BFS example

Algorithm: BFS(graph G , start vertex s)

- ▶ **Initialize:** mark all nodes unexplored except s explored
- ▶ Use Q = queue data structure, add s to Q
- ▶ **Main loop:** While Q is not empty:
 - ▶ Remove the first node of Q and call it v
 - ▶ For each edge (v, w) : if w unexplored:
 - ▶ Mark w explored
 - ▶ Add w to Q



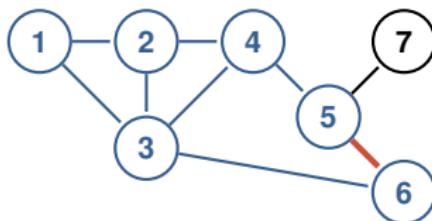
Queue of vertices:

6	5
---	---

BFS example

Algorithm: BFS(graph G , start vertex s)

- ▶ **Initialize:** mark all nodes unexplored except s explored
- ▶ Use Q = queue data structure, add s to Q
- ▶ **Main loop:** While Q is not empty:
 - ▶ Remove the first node of Q and call it v
 - ▶ For each edge (v, w) : if w unexplored:
 - ▶ Mark w explored
 - ▶ Add w to Q



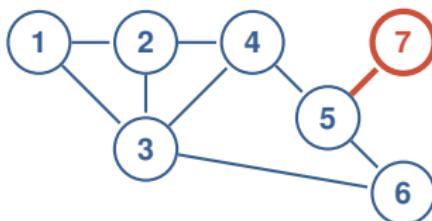
Queue of vertices:

5

BFS example

Algorithm: BFS(graph G , start vertex s)

- ▶ **Initialize:** mark all nodes unexplored except s explored
- ▶ Use Q = queue data structure, add s to Q
- ▶ **Main loop:** While Q is not empty:
 - ▶ Remove the first node of Q and call it v
 - ▶ For each edge (v, w) : if w unexplored:
 - ▶ Mark w explored
 - ▶ Add w to Q



Queue of vertices:



BFS running time

Correctness: At the end, any given vertex v is explored if and only if there is a path s to any other v

Running time: $O(n_s + m_s)$, the number of nodes and edges reachable from s

- ▶ Each node will be added
- ▶ Loop through each edge, but how many times?

BFS for shortest paths

Suppose we want to measure actors' degrees of separation from Kevin Bacon

- ▶ Want the **smallest distance** (fewest number of edges) from s to v

BFS for shortest paths

Suppose we want to measure actors' degrees of separation from Kevin Bacon

- ▶ Want the **smallest distance** (fewest number of edges) from s to v

We need some extra code: **initialize** $dist(v) = 0$ for $v = s$, ∞ otherwise

BFS for shortest paths

Suppose we want to measure actors' degrees of separation from Kevin Bacon

- ▶ Want the **smallest distance** (fewest number of edges) from s to v

We need some extra code: **initialize** $dist(v) = 0$ for $v = s$, ∞ otherwise

Main loop: when going through edge (v, w) : if w unexplored, set $dist(w) = dist(v) + 1$

BFS for shortest paths

Suppose we want to measure actors' degrees of separation from Kevin Bacon

- ▶ Want the **smallest distance** (fewest number of edges) from s to v

We need some extra code: **initialize** $dist(v) = 0$ for $v = s$, ∞ otherwise

Main loop: when going through edge (v, w) : if w unexplored, set $dist(w) = dist(v) + 1$

This will give the **shortest path!** **Why?**

BFS for connectivity

Connected components = the “pieces” of (undirected) graph G

Goal: compute all connected components

- ▶ Check if graph is disconnected
- ▶ Visualize graphs
- ▶ Clustering

Idea:

- ▶ If BFS finishes without exploring all nodes, restart BFS from unexplored node
- ▶ $O(m + n)$ time

Why study graph optimization?

Many problems revolve around **transitions from one system state to another**

- ▶ Graph search often useful
- ▶ Examples: travelling, scheduling, etc.

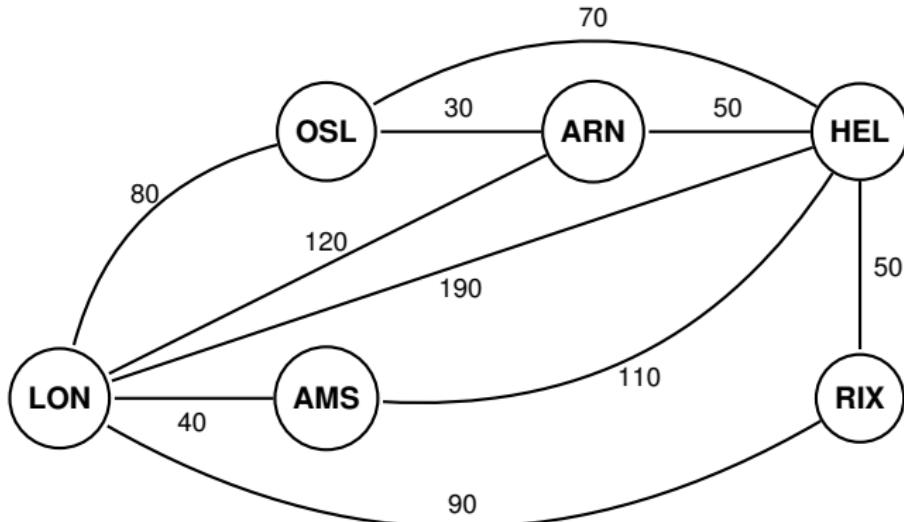
For **unweighted graphs**, many problems solved by **BFS** and **DFS**

- ▶ Linear time
- ▶ DFS is challenge problem in the workshop
- ▶ **Weighted graphs** require different methods — Dijkstra's algorithm, Bellman-Ford algorithm

Graph search

Suppose you're travelling around Europe

- ▶ Know flights and their prices
- ▶ Assume the price of flying from London to Helsinki via Stockholm is the sum of the two legs (not true!)
- ▶ Cheapest price?



Shortest paths

Input: (directed) graph $G = (V, E)$

- ▶ V : set of n vertices, E : set of m edges
- ▶ Each edge e has length (cost) $c_e \geq 0$
- ▶ Start from vertex s

Shortest paths

Input: (directed) graph $G = (V, E)$

- ▶ V : set of n vertices, E : set of m edges
- ▶ Each edge e has length (cost) $c_e \geq 0$
- ▶ Start from vertex s

Output: for each vertex v in V :

- ▶ $SP(v)$ = length of shortest $s - v$ path in G

Shortest paths

Input: (directed) graph $G = (V, E)$

- ▶ V : set of n vertices, E : set of m edges
- ▶ Each edge e has length (cost) $c_e \geq 0$
- ▶ Start from vertex s

Output: for each vertex v in V :

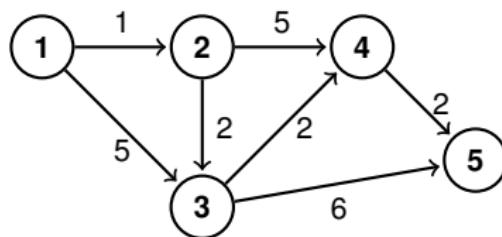
- ▶ $SP(v)$ = length of shortest $s - v$ path in G

Assume:

- ▶ That such paths exist (connected graph)
- ▶ $c_e \geq 0$ (important!!) – Dijkstra vs Bellman-Ford

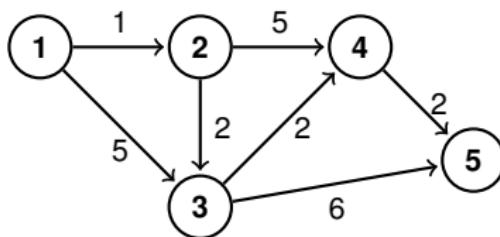
Example

What is the **shortest path** from $s=1$ to $v=1,2,3,4,5$?



Example

What is the **shortest path** from $s=1$ to $v=1,2,3,4,5$?



1. 0,1,5,6,11?
2. 0,1,3,6,8?
3. 0,1,5,5,8?
4. 0,1,3,5,7?

Why not just use BFS?

BFS already allows us to find the shortest paths?

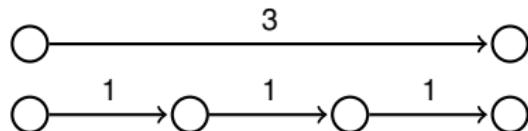
- ▶ Assuming that all edges have length 1...

Why not just use BFS?

BFS already allows us to find the shortest paths?

- ▶ Assuming that all edges have length 1...

Write each edge as a series of length one edges?

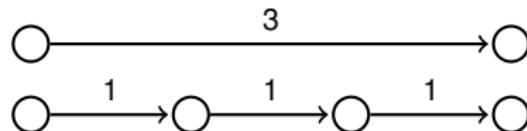


Why not just use BFS?

BFS already allows us to find the shortest paths?

- ▶ Assuming that all edges have length 1...

Write each edge as a series of length one edges?



- ▶ Graph becomes **impractical**...

Need a new algorithm: **Dijkstra's shortest-paths algorithm**

Dijkstra is extra...

Builds on BFS ideas

Running time $O(m \log n)$ – still blazingly fast

- ▶ With a clever use of **heap** data structure
- ▶ Essentially replace BFS queue with a **priority queue** ordered by distance “scores”

Workshop: graphs

After the break...

BFS, small world phenomenon and Kevin Bacon

Challenge: depth-first search (DFS)

Extra: Dijkstra...