CS 1114:
Graphs and Blobs

Prof. Graeme Bailey

http://cs1114.cs.cornell.edu

(notes modified from Noah Snavely, Spring 2009)

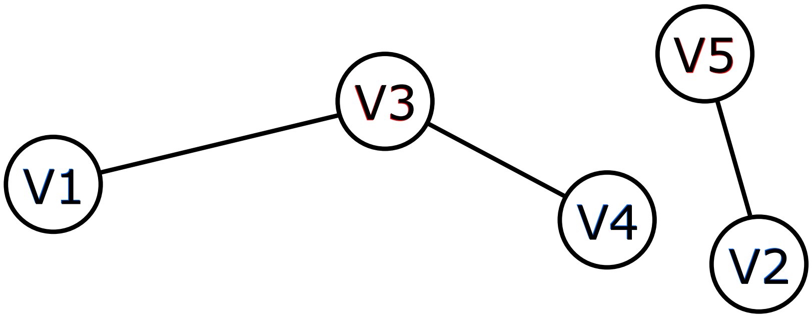
Cornell University
Computer Science

Some major graph problems

- Graph colouring
  - Ensuring that radio stations don’t clash
- Graph connectivity
  - How fragile is the internet?
- Graph cycles
  - Helping FedEx/UPS/DHL plan a route
- Planarity testing
  - Connecting computer chips on a motherboard
- Graph isomorphism
  - Is a chemical structure already known?

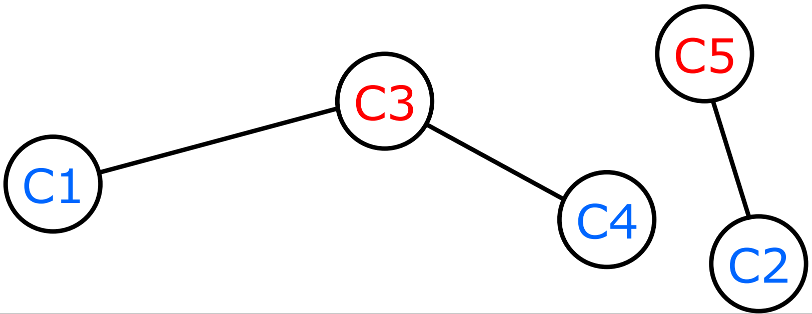
Graph colouring problem

- Given a graph and a set of colours {1,…,k}, assign each vertex a colour
- Adjacent vertices have different colours

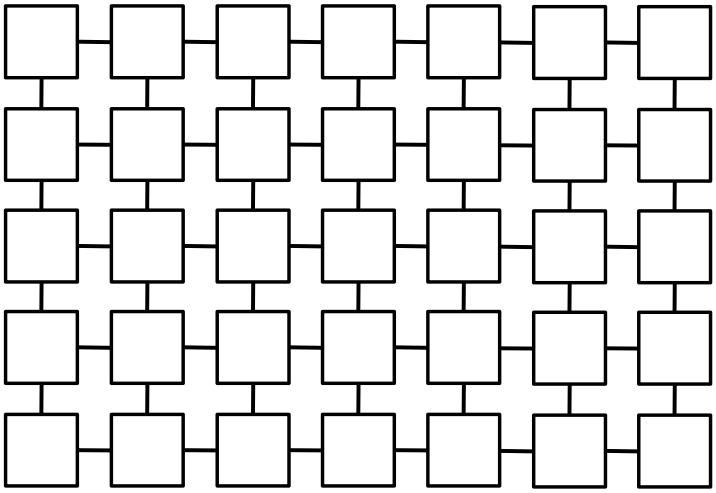


Radio frequencies via colouring

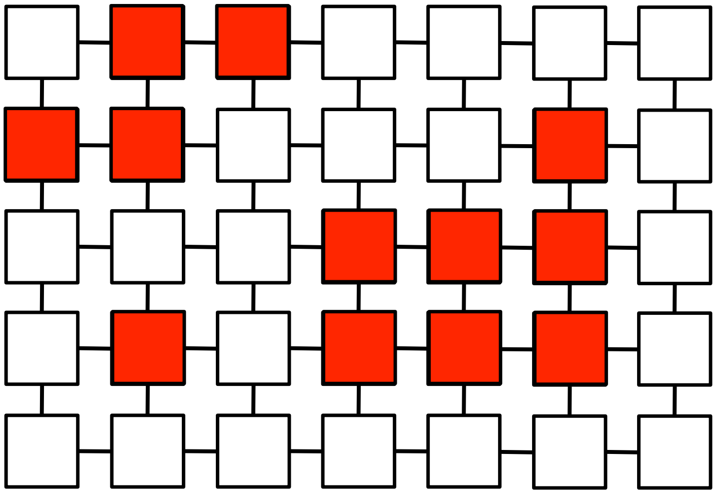
- How can we assign frequencies to a set of radio stations so that there are no clashes?
- Make a graph where each station is a vertex
  - Put an edge between two stations that clash
    - I.e., if their signal areas overlap
  - Any colouring is a non-clashing assignment of frequencies
    - Can you prove this? What about vice-versa?



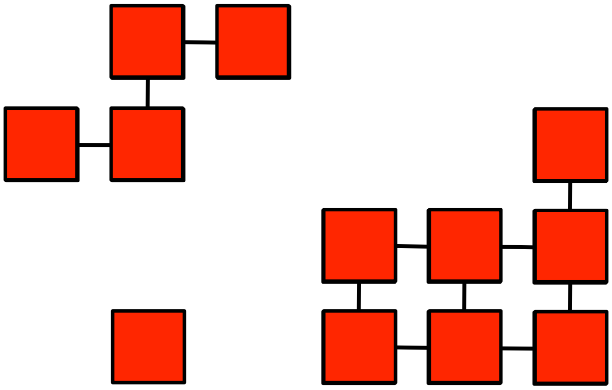
Images as graphs



Images as graphs



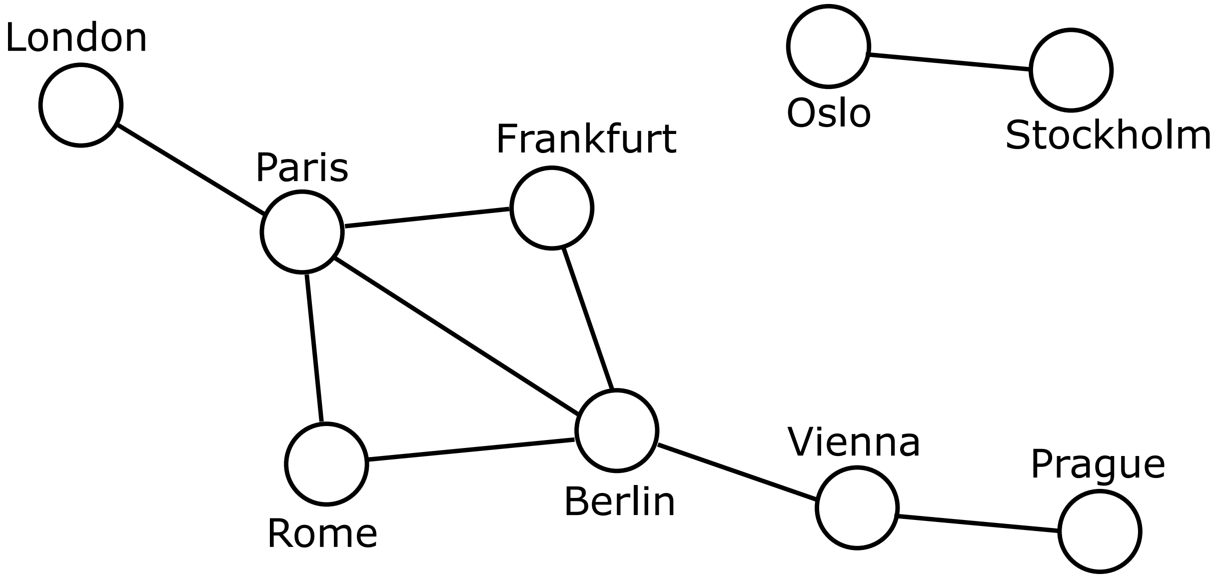
Images as graphs



Graphs and paths

- Can you get from vertex V to vertex W?
  - Is there a route from one city to another?
- More precisely, is there a sequence of vertices $\{V, V_1, V_2, \ldots, V_k, W\}$ such that every adjacent pair has an edge between them? *(Sometimes we care about directed edges.)*
  - This is called a **path**
  - A **cycle** (or **loop**) is a path from V to V
  - A path is **simple** if no vertex appears twice
    - *though sometimes we define simple loops*

European rail links (simplified)



- Can we get from London to Prague on the train?
- How about London to Stockholm?

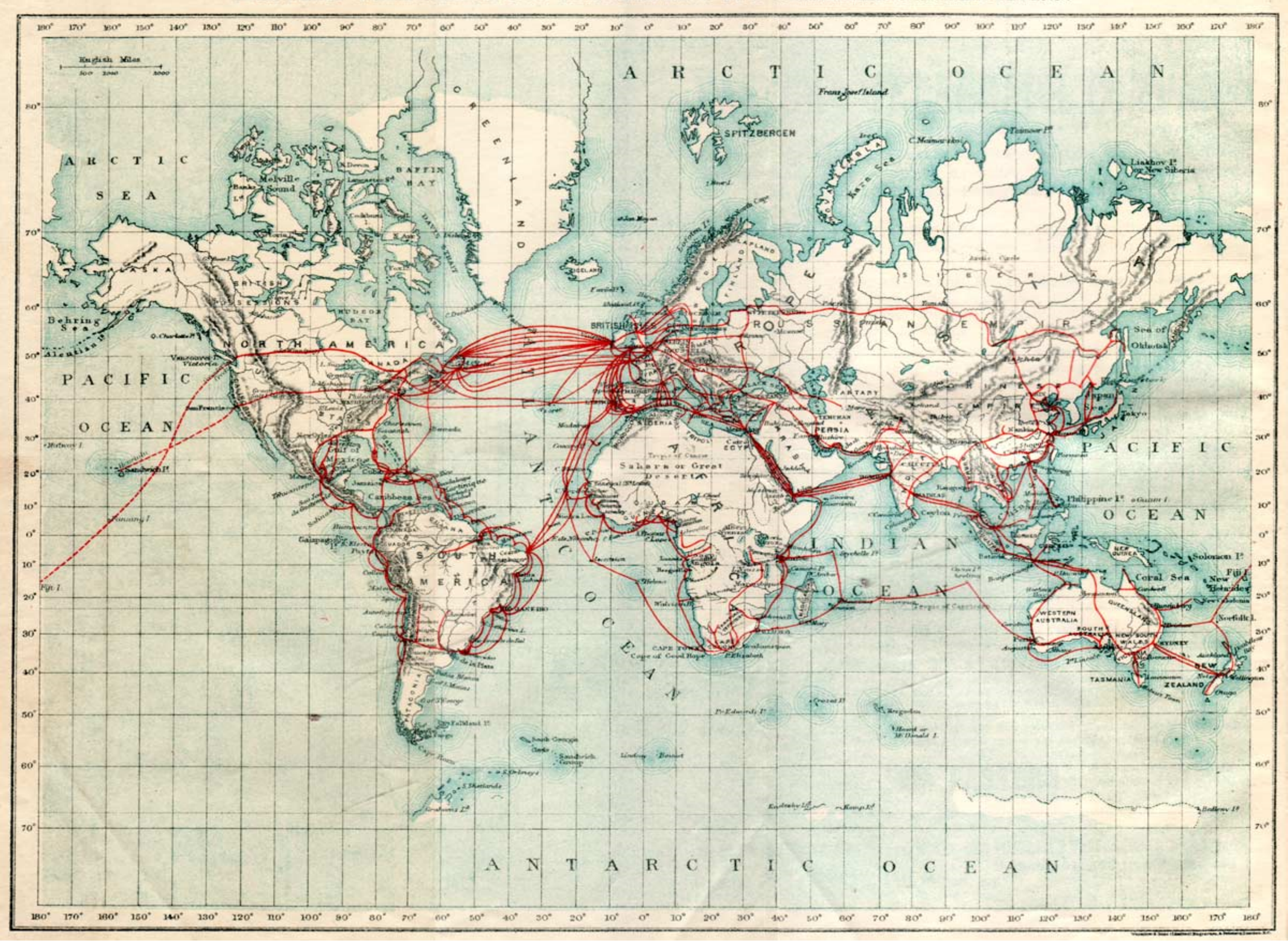
Graph connectivity

- For any pair of nodes, is there a path between them?
  - Basic idea of the Internet: you can get from any computer to any other computer
  - This pair of nodes is called *connected*
  - A graph is connected if all nodes are connected

- Related question: remove an arbitrary node (or edge), how connected is the graph?
  - Is the Internet intact if any 1 computer fails?
  - Or any 1 edge between computers?

"Eastern Telegraph Co. and its General Connections" (1901)

ChrisHarrison.net



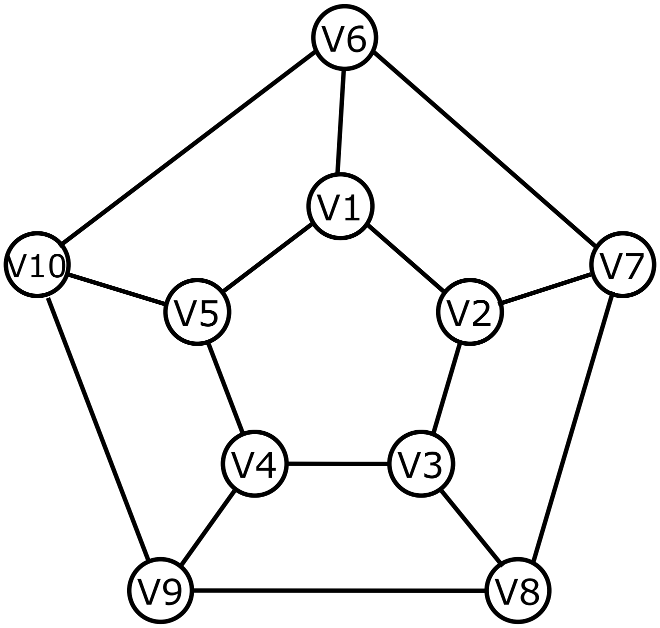
Hamiltonian & Eulerian cycles

- Two questions that are useful for problems such as mailman delivery routes

  - Hamiltonian cycle:
    - A cycle that visits each *vertex* exactly once (except the start and end)
  - Eulerian cycle:
    - A cycle that uses each *edge* exactly once

  - Sometimes we look for Hamiltonian or Eulerian *paths*

Hamiltonian & Eulerian cycles



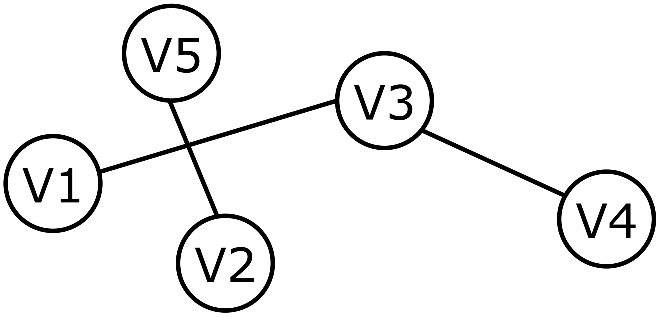
visits vertices

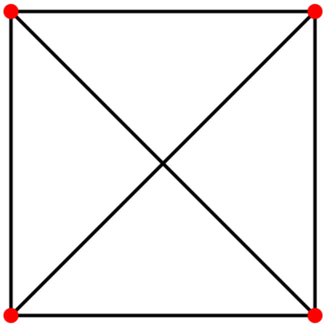
visits edges

- ♦♦ Is it easier to tell if a graph has a Hamiltonian cycle or an Eulerian cycle?

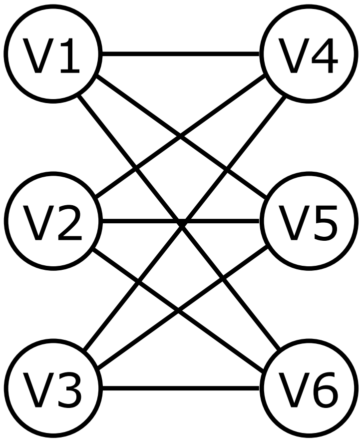
Planarity testing

- A graph is planar if you can draw it without the edges crossing
  - It's OK to move the edges or vertices around, as long as edges connect the same vertices





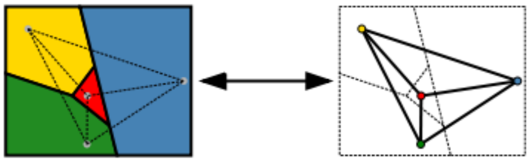
◆ Is this graph planar?



◆◆ *Can you prove it?*

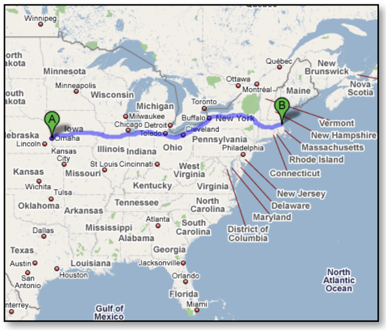
Four-colour theorem

- Any planar graph can be coloured using no more than 4 colours



“Small world” phenomenon
(Six degrees of separation)

- How close together are nodes in a graph (e.g., what’s the average number of hops connecting pairs of nodes?)



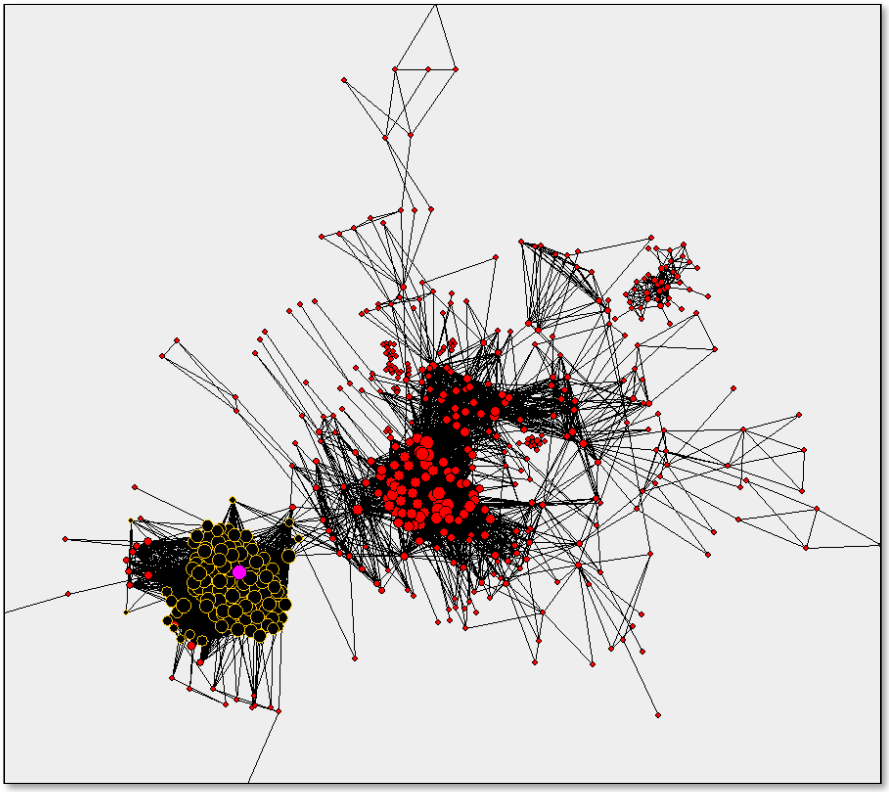
- Milgram’s small world experiment:
    - Send postcard to random person A in Omaha; task is to get it to a random person B in Boston
    - If A knows B, send directly
    - Otherwise, A sends to someone A knows who is most likely to know B
    - People are separated by 5.5 links on average

Graph of Flickr images ……

Flickr images of the Pantheon, Rome (built 126 AD)

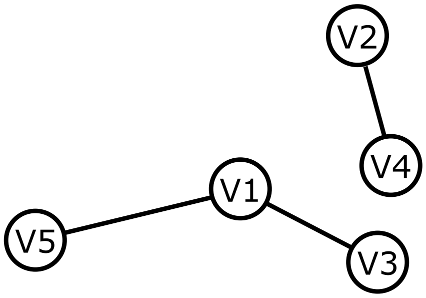
Images are matched using visual features

Image graph of the Pantheon



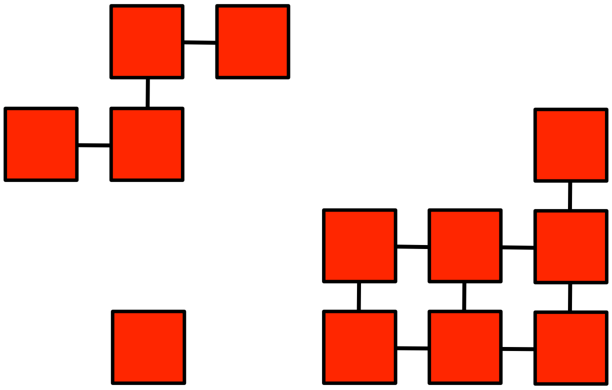
Connected components

- Even if all nodes are not connected, there will be subsets that are all connected
  - Connected components

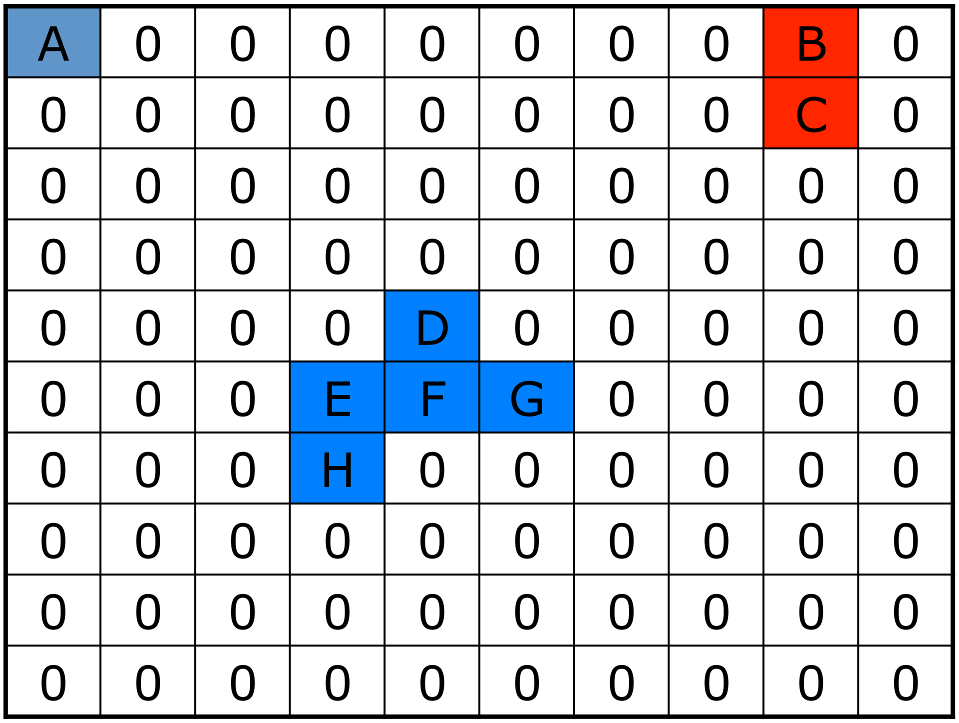


  - Component 1: { V1, V3, V5 }
  - Component 2: { V2, V4 }

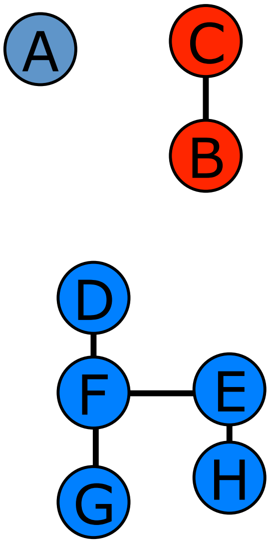
Blobs are components!



Blobs are components!



A	0	0	0	0	0	0	0	B	0
0	0	0	0	0	0	0	0	C	0
0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0
0	0	0	0	D	0	0	0	0	0
0	0	0	E	F	G	0	0	0	0
0	0	0	H	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0



Finding components *(blobs)*

- For each vertex we visit, we colour its neighbours and remember that we need to visit them at some point (e.g., put them in a *todo* list):
  - **While** there are any *uncoloured* vertices, select one, adding it to the (empty) *todo* list and *colouring* it uniquely
    - **While** the *todo* list is not empty
      - remove a vertex V from the *todo* list to visit
      - add the *uncoloured* neighbors of this V to the *todo* list and colour them with the same colour
  - Repeat until all vertices are coloured
- This is also called *graph traversal*

1	0	0	0	0	0	0	0	1	0
0	0	0	0	0	0	0	0	1	0
0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0
0	0	0	0	1	0	0	0	0	0
0	0	0	1	1	1	0	0	0	0
0	0	0	1	1	1	0	0	0	0
0	0	0	1	1	1	0	0	0	0
0	0	0	1	1	1	0	0	0	0
0	0	0	0	0	0	0	0	0	0

Coloring a component

1	0	0	0	0	0	0	0	1	0
0	0	0	0	0	0	0	0	1	0
0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0
0	0	0	0	**A**	0	0	0	0	0
0	0	0	B	C	D	0	0	0	0
0	0	0	E	F	G	0	0	0	0
0	0	0	H	I	J	0	0	0	0
0	0	0	K	L	M	0	0	0	0
0	0	0	0	0	0	0	0	0	0

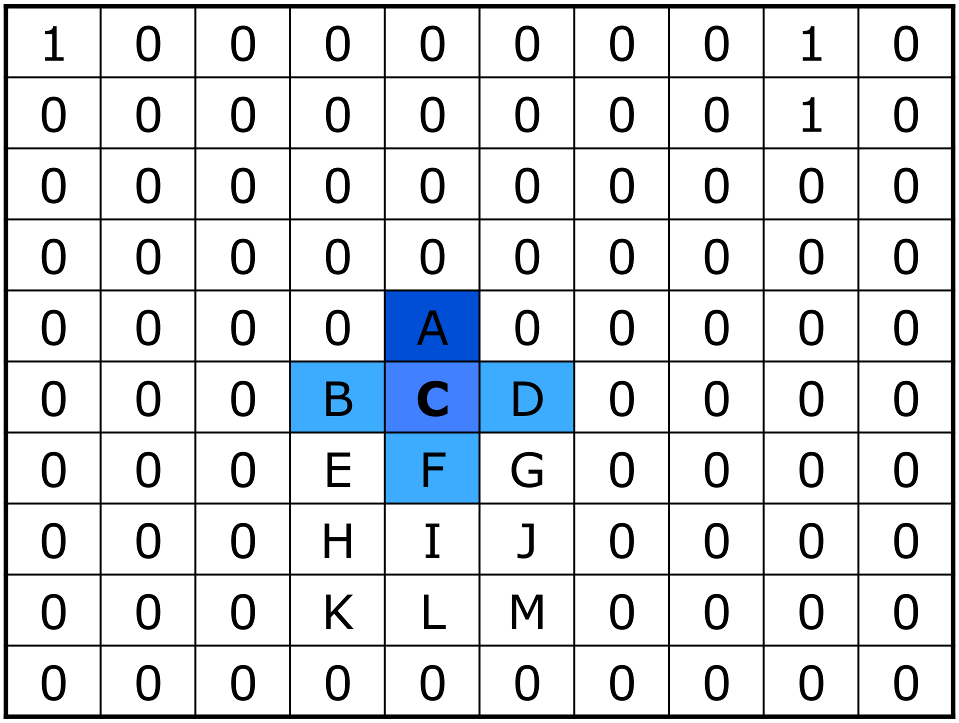
Current node: A
Todo List: [ ]

1	0	0	0	0	0	0	0	1	0
0	0	0	0	0	0	0	0	1	0
0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0
0	0	0	0	**A**	0	0	0	0	0
0	0	0	B	C	D	0	0	0	0
0	0	0	E	F	G	0	0	0	0
0	0	0	H	I	J	0	0	0	0
0	0	0	K	L	M	0	0	0	0
0	0	0	0	0	0	0	0	0	0

Current node: A
Todo List: [C] ← *Done with A, choose next from Todo List*

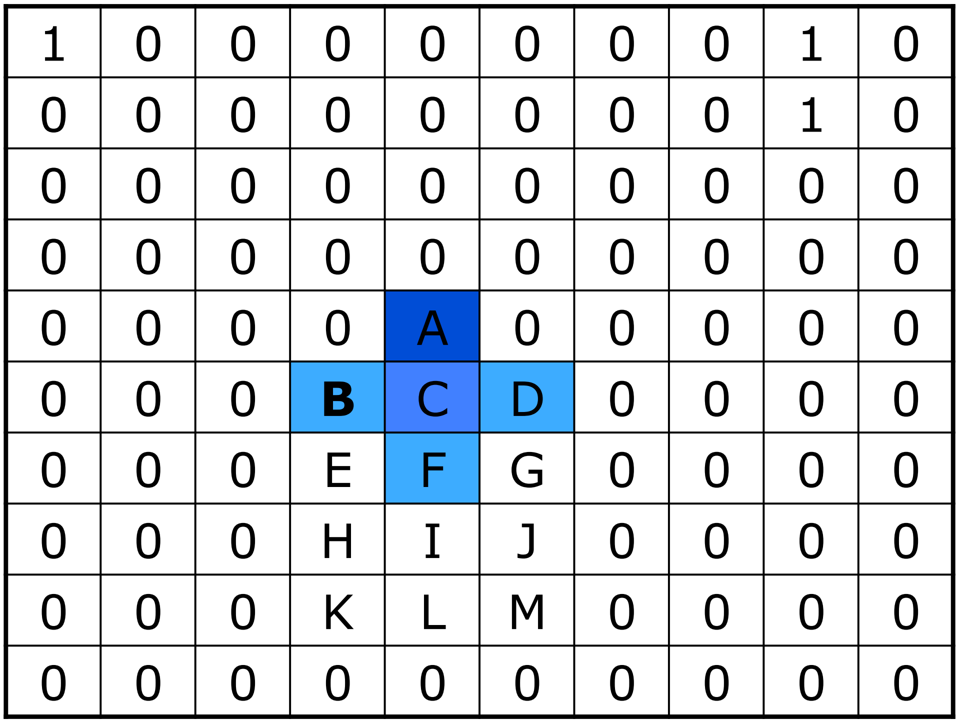
1	0	0	0	0	0	0	0	1	0
0	0	0	0	0	0	0	0	1	0
0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0
0	0	0	0	A	0	0	0	0	0
0	0	0	B	**C**	D	0	0	0	0
0	0	0	E	F	G	0	0	0	0
0	0	0	H	I	J	0	0	0	0
0	0	0	K	L	M	0	0	0	0
0	0	0	0	0	0	0	0	0	0

Current node: C
Todo List: [ ]



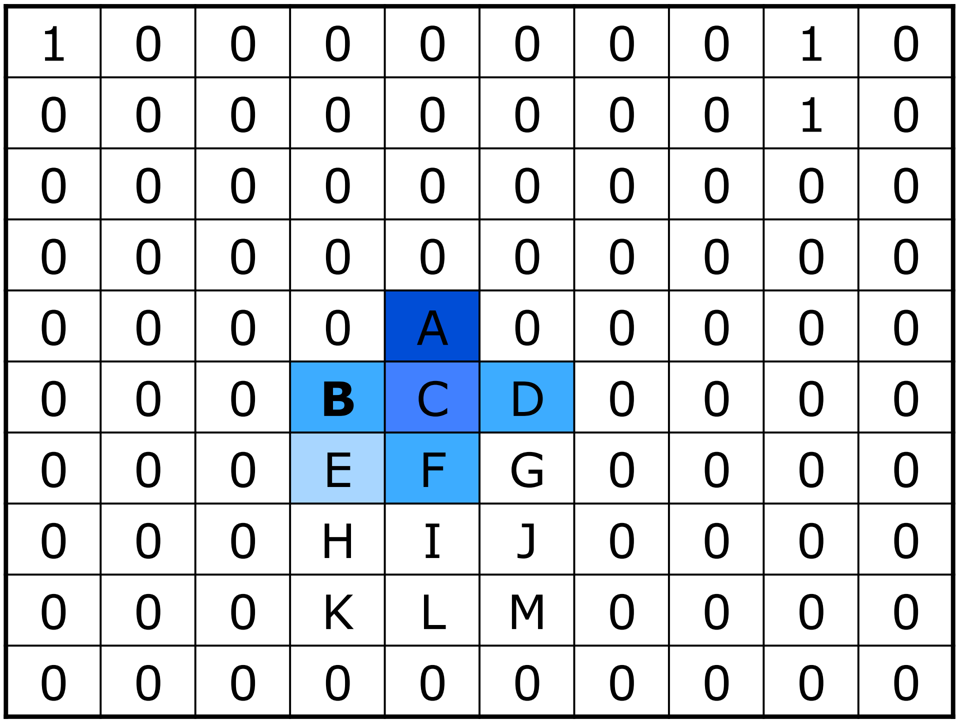
1	0	0	0	0	0	0	0	1	0
0	0	0	0	0	0	0	0	1	0
0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0
0	0	0	0	A	0	0	0	0	0
0	0	0	B	**C**	D	0	0	0	0
0	0	0	E	F	G	0	0	0	0
0	0	0	H	I	J	0	0	0	0
0	0	0	K	L	M	0	0	0	0
0	0	0	0	0	0	0	0	0	0

Current node: C
Todo List: [B, F, D]



1	0	0	0	0	0	0	0	1	0
0	0	0	0	0	0	0	0	1	0
0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0
0	0	0	0	A	0	0	0	0	0
0	0	0	**B**	C	D	0	0	0	0
0	0	0	E	F	G	0	0	0	0
0	0	0	H	I	J	0	0	0	0
0	0	0	K	L	M	0	0	0	0
0	0	0	0	0	0	0	0	0	0

Current node: B
Todo List: [F, D]



1	0	0	0	0	0	0	0	1	0
0	0	0	0	0	0	0	0	1	0
0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0
0	0	0	0	A	0	0	0	0	0
0	0	0	**B**	C	D	0	0	0	0
0	0	0	E	F	G	0	0	0	0
0	0	0	H	I	J	0	0	0	0
0	0	0	K	L	M	0	0	0	0
0	0	0	0	0	0	0	0	0	0

Current node: B
Todo List: [F, D, E]

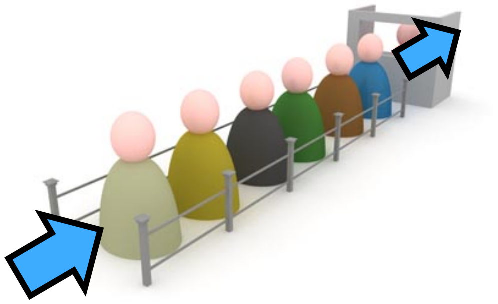
Stacks and Queues

- One way to implement the *todo* list is as a *stack*
  - LIFO: Last In First Out
  - The newest task is the one you'll do next
  - Think of a pile of trays in a cafeteria
    - Trays at the bottom can stay there a while…
- The alternative is a *queue*
  - FIFO: First In First Out
  - The oldest task is the one you'll do next
  - Think of a line of (well-mannered) people
    - First come, first served

Stacks

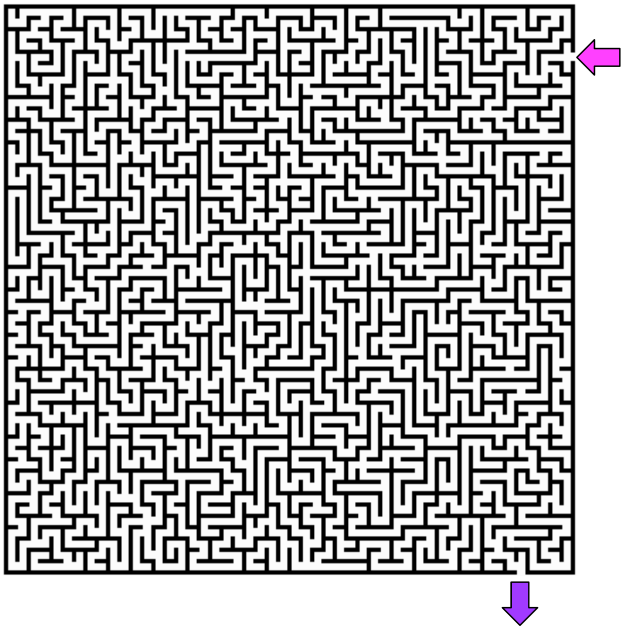
- Two operations:
- Push: add something to the top of the stack
- Pop: remove the thing on top of the stack

Queue



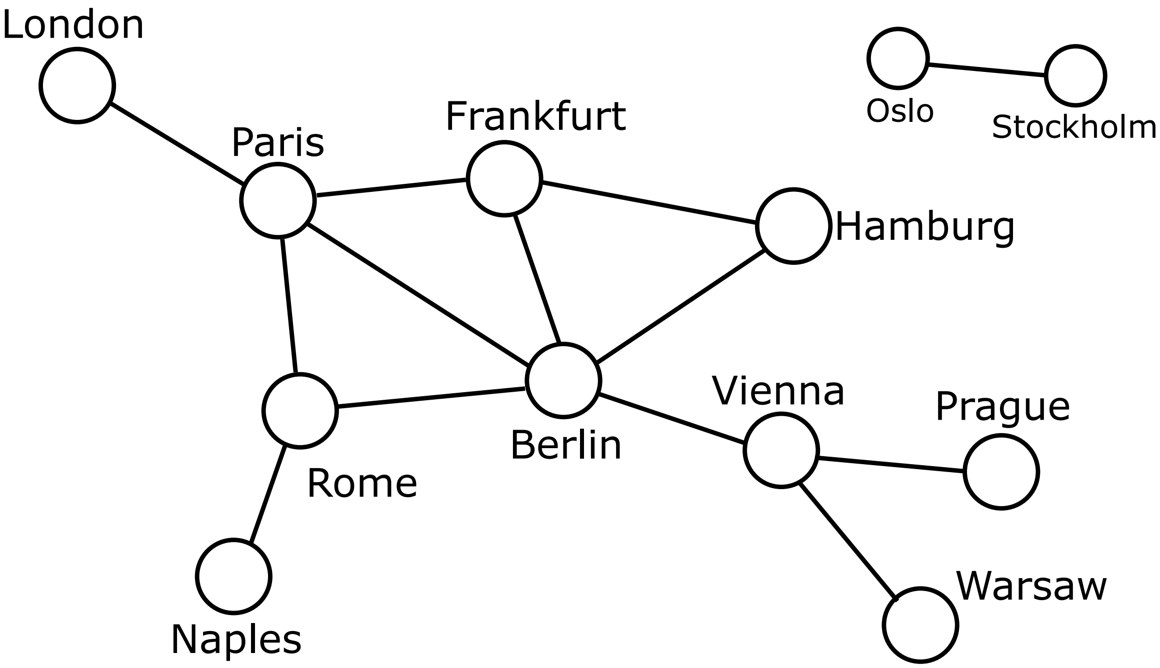
- Two operations:
- Enqueue: add something to the end of the queue
- Dequeue: remove something from the front of the queue

Graph traversal

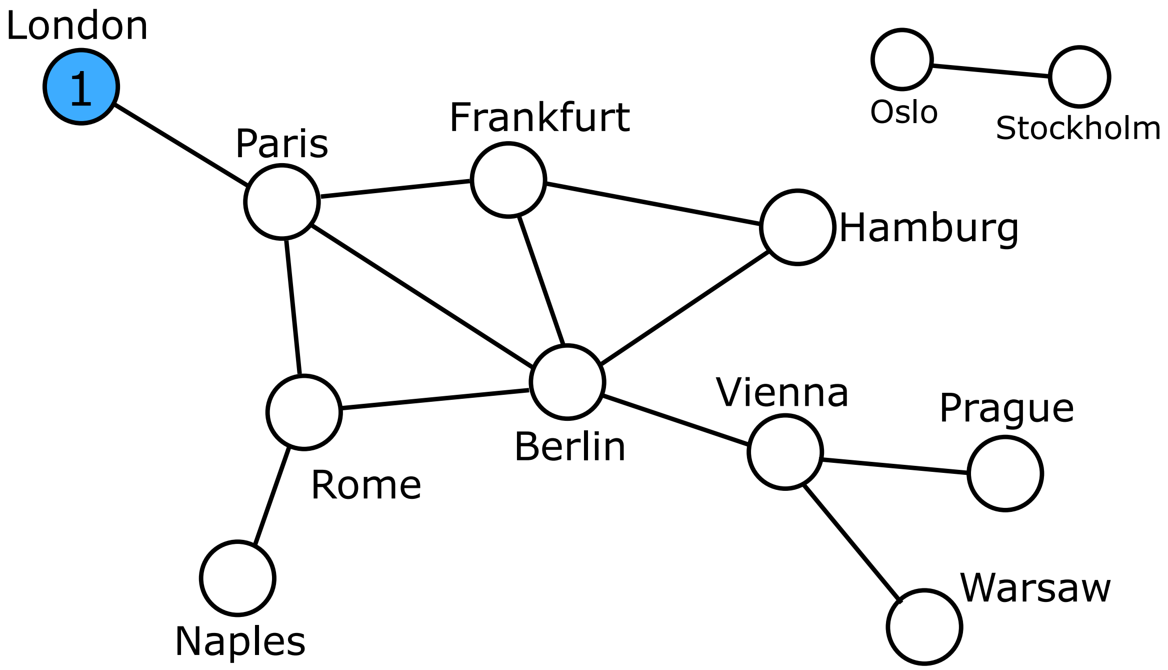


- Suppose you’re in a maze
- What strategy can you use to find the exit?

Graph traversal

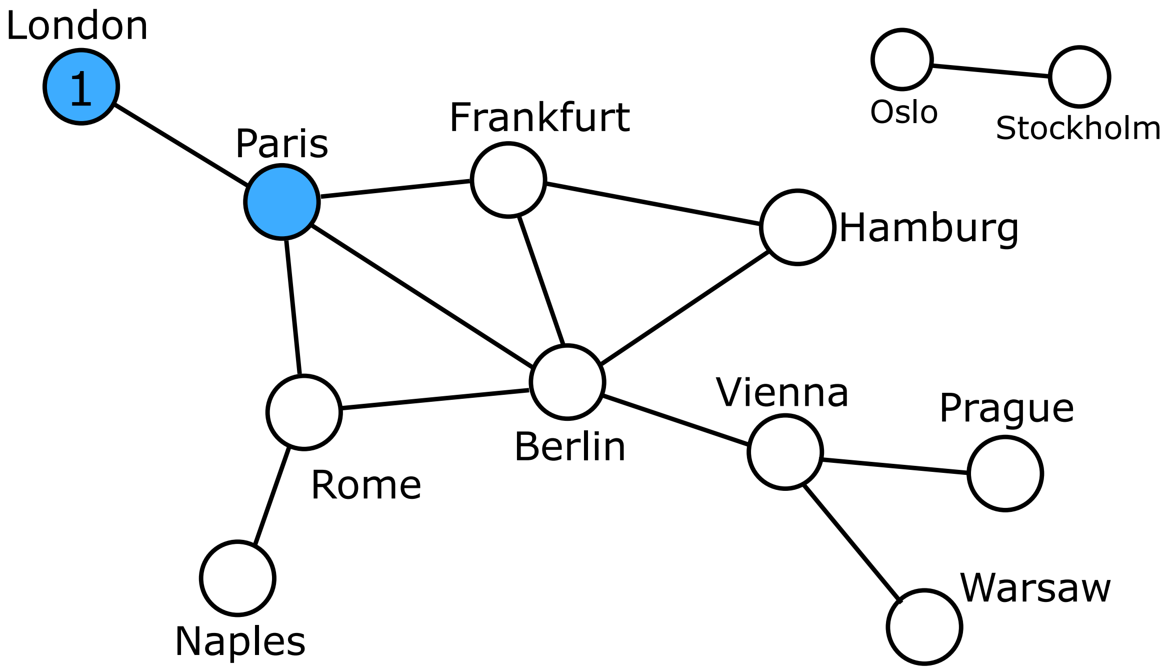


Graph traversal (stack)



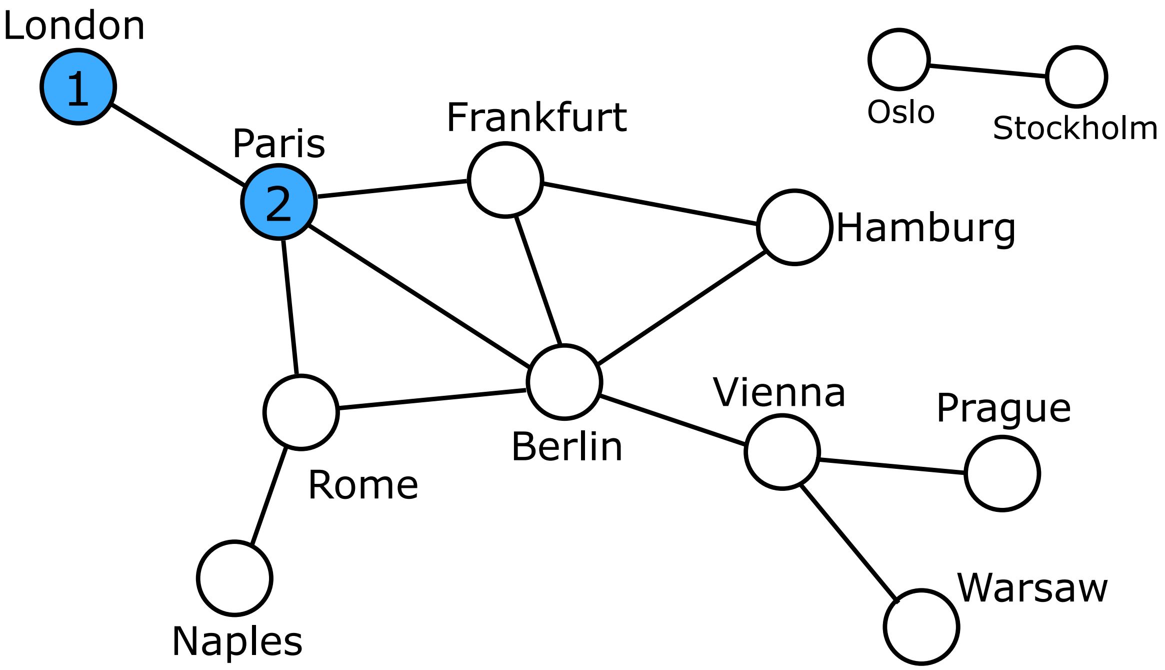
Current node: London
Todo list: [ ]

Graph traversal (stack)



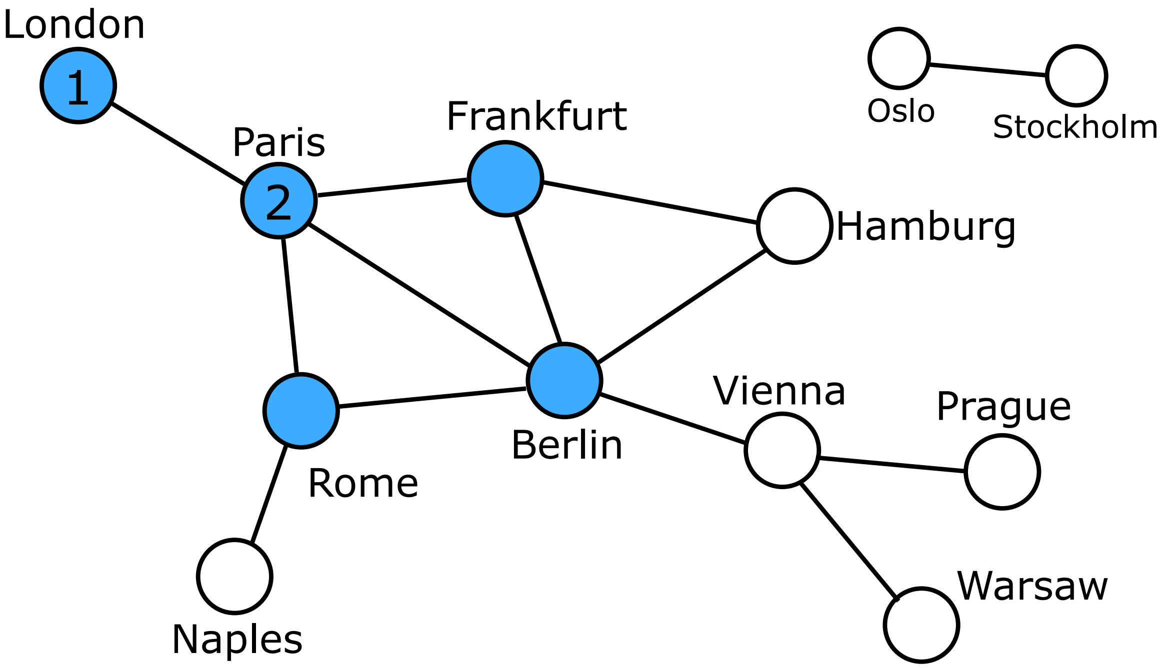
Current node: London
Todo list: [Paris]

Graph traversal (stack)



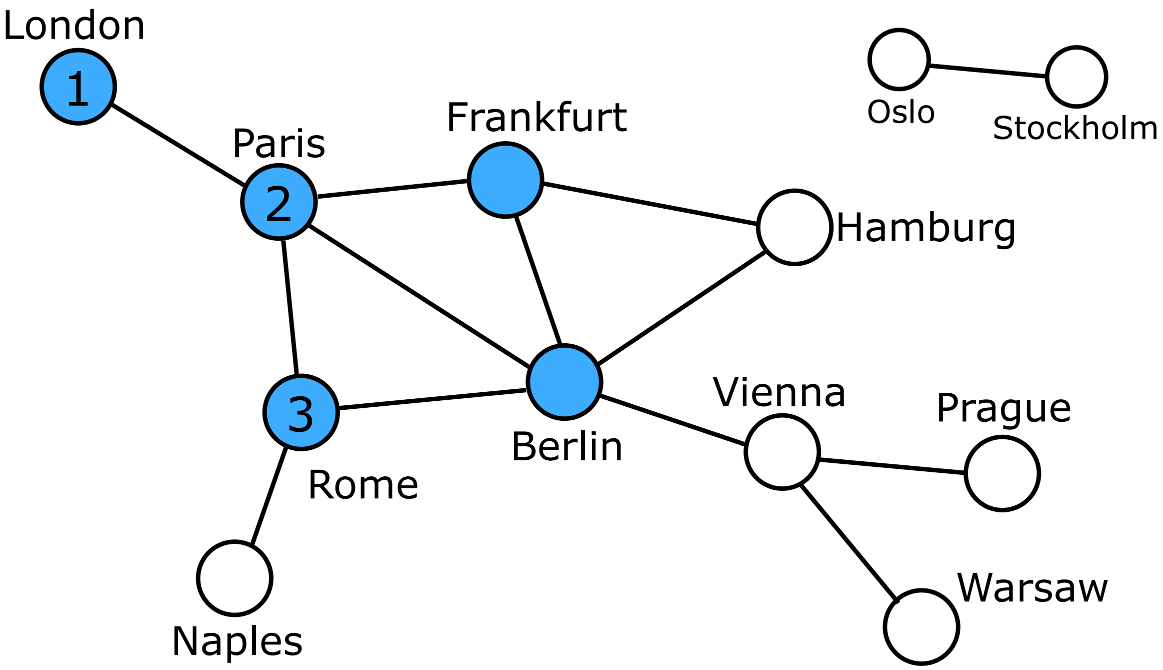
Current node: Paris
Todo list: [ ]

Graph traversal (stack)



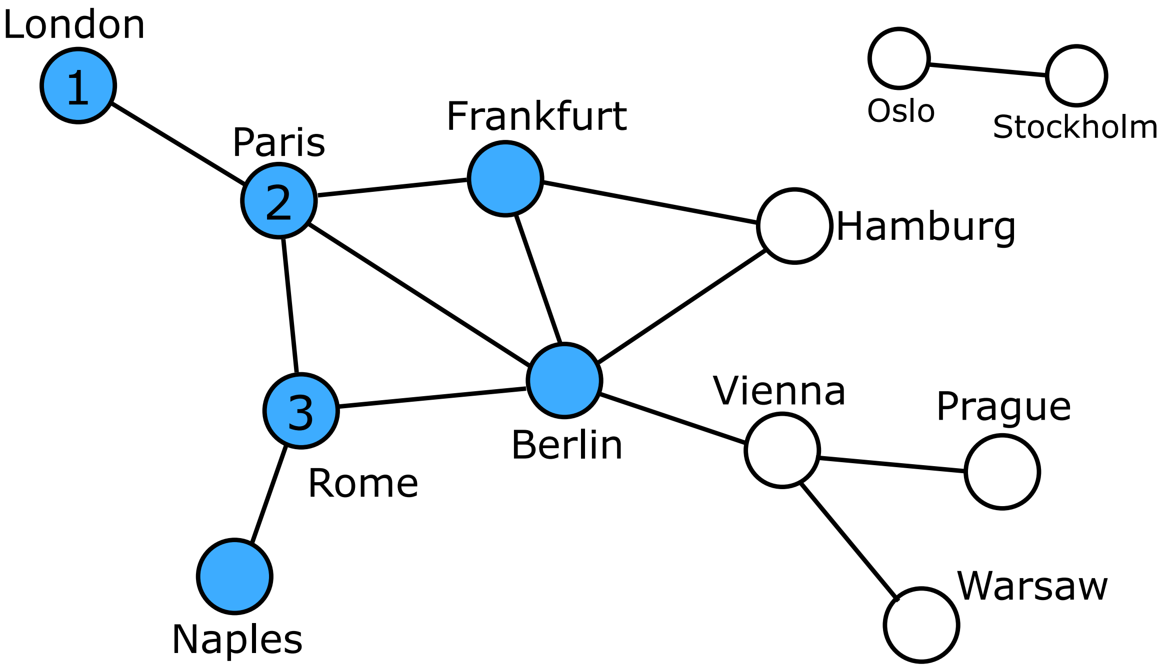
Current node: Paris
Todo list: [Frankfurt, Berlin, Rome]

Graph traversal (stack)



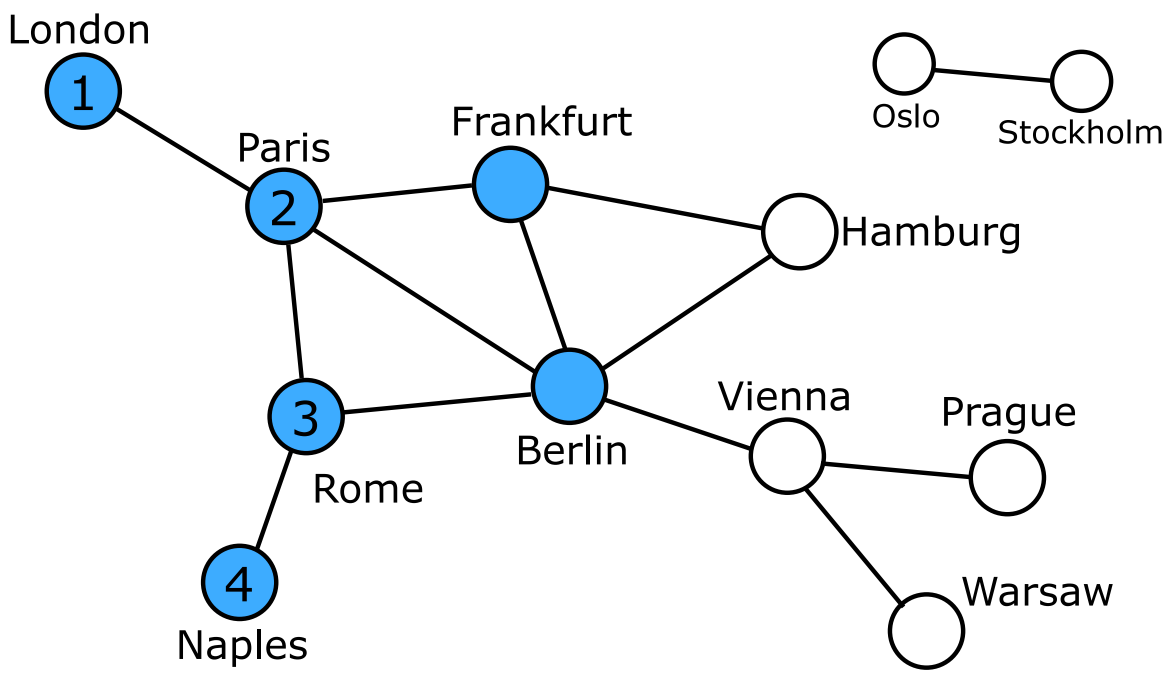
Current node: Rome
Todo list: [Frankfurt, Berlin]

Graph traversal (stack)



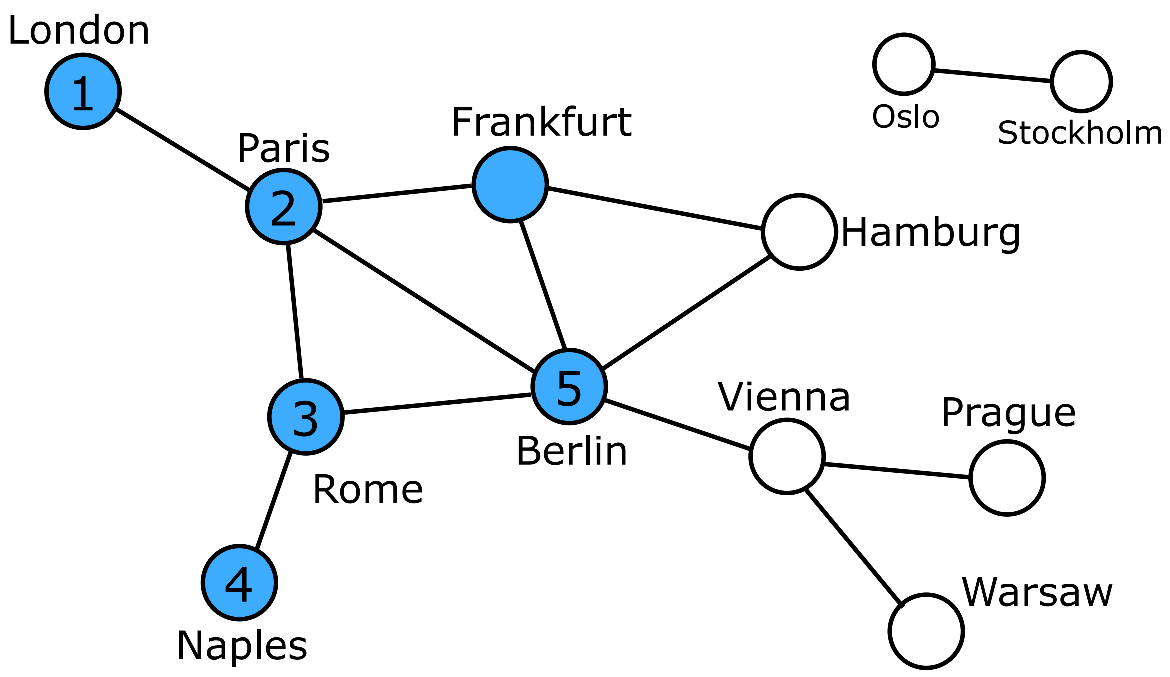
Current node: Rome
Todo list: [Frankfurt, Berlin, Naples]

Graph traversal (stack)



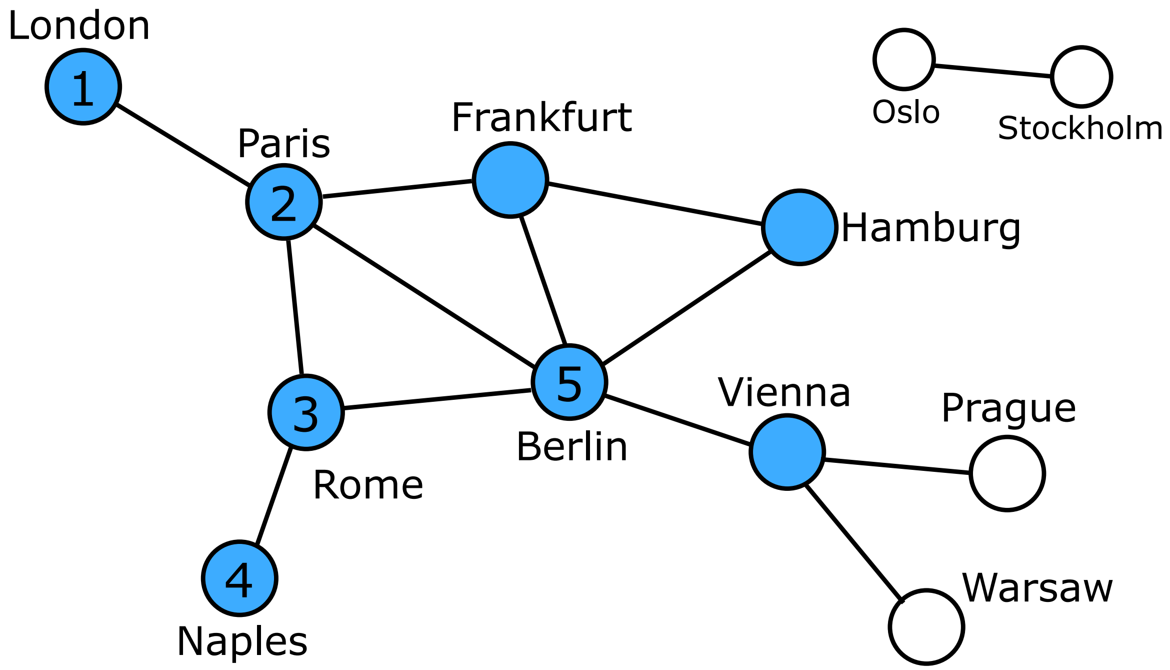
Current node: Naples
Todo list: [Frankfurt, Berlin]

Graph traversal (stack)



Current node: Berlin
Todo list: [Frankfurt]

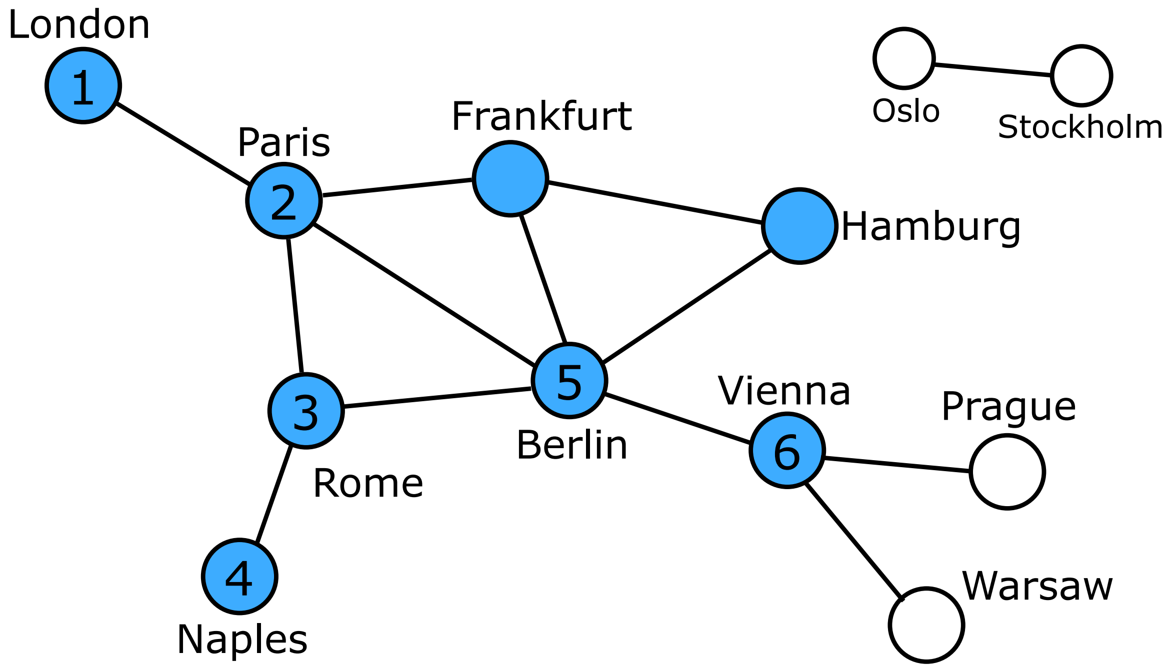
Graph traversal (stack)



Current node: Berlin

Todo list: [Frankfurt, Hamburg, Vienna]

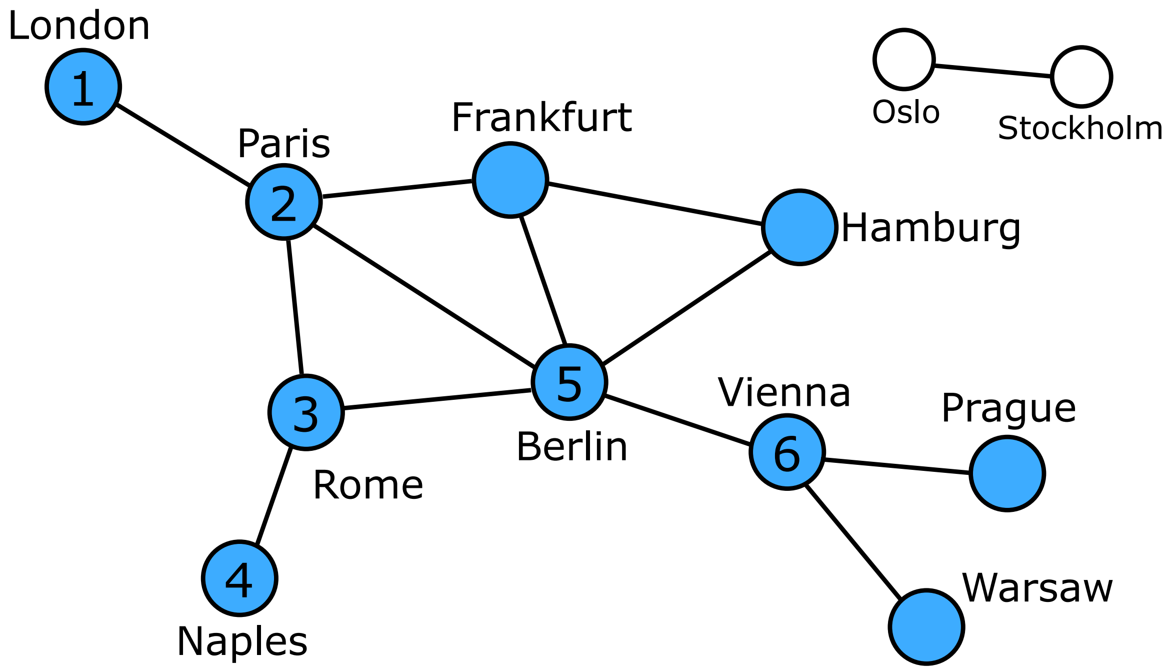
Graph traversal (stack)



Current node: Vienna

Todo list: [Frankfurt, Hamburg]

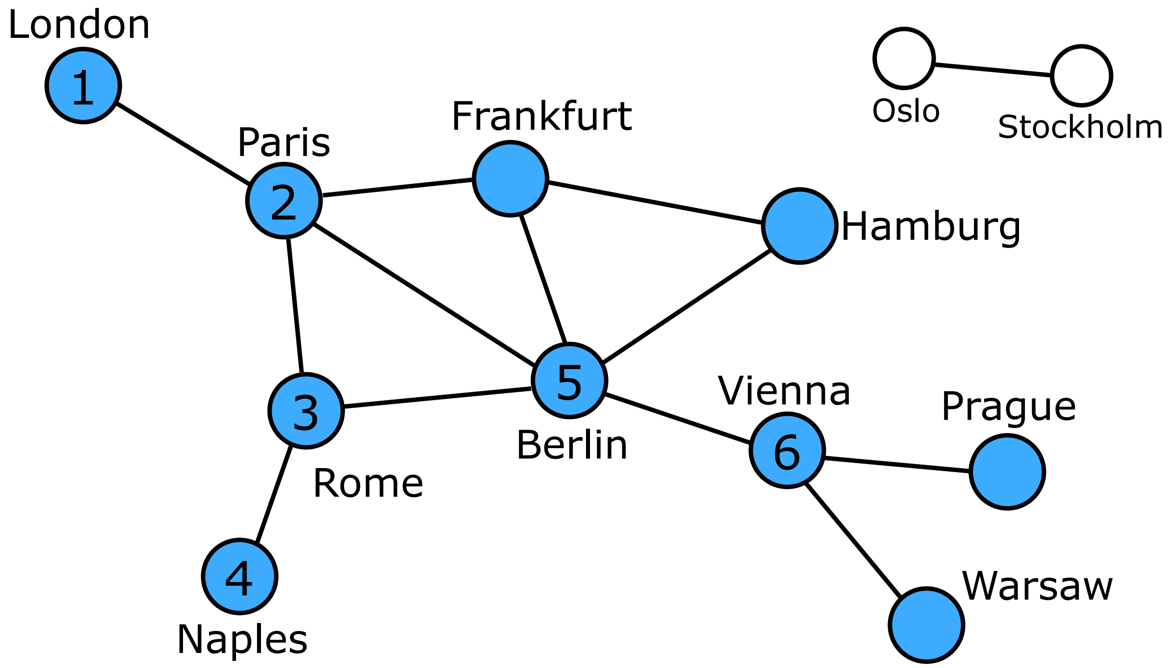
Graph traversal (stack)



Current node: Vienna

Todo list: [Frankfurt, Hamburg, Prague, Warsaw]

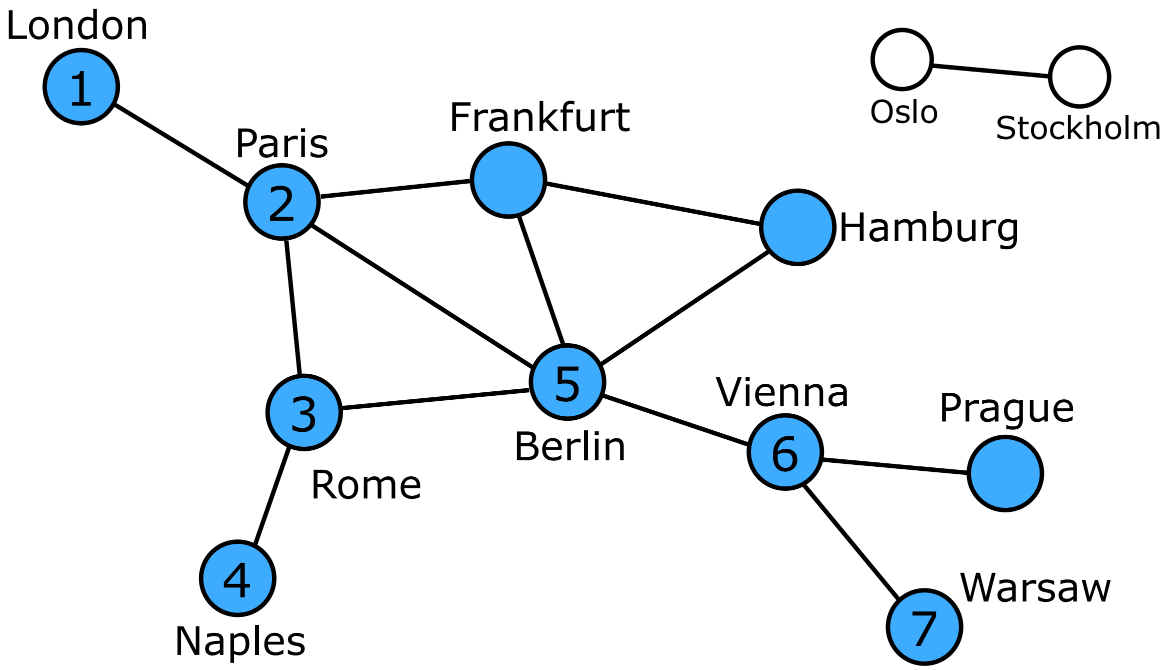
Graph traversal (stack)



Current node: Vienna

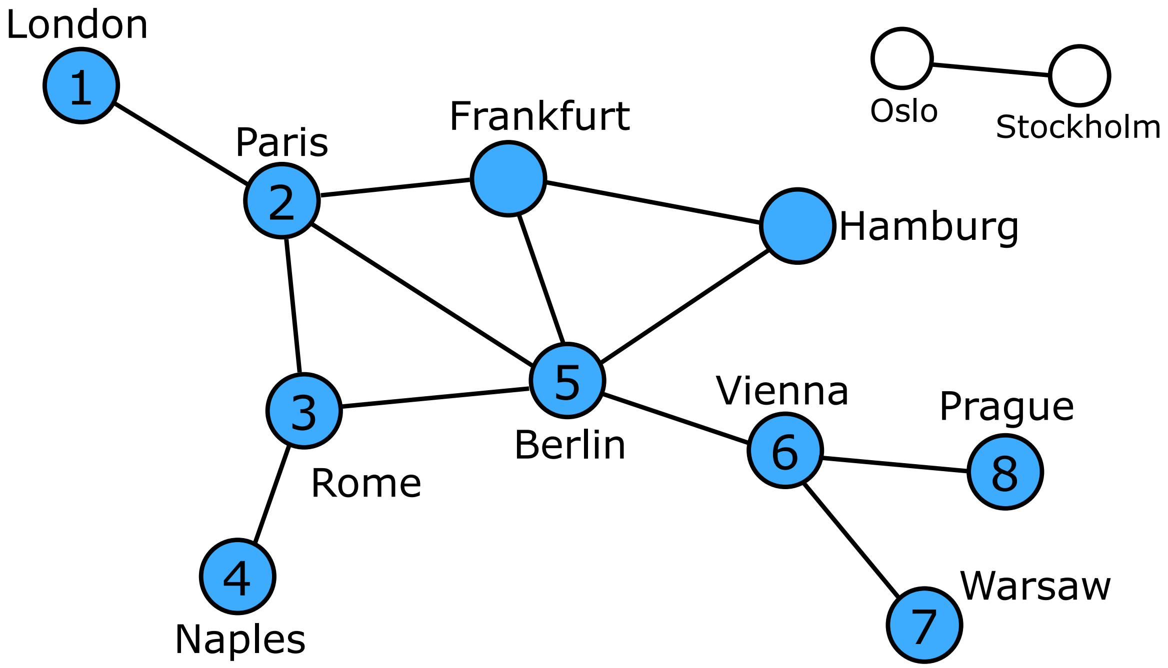
Todo list: [Frankfurt, Hamburg, Prague, Warsaw]

Graph traversal (stack)



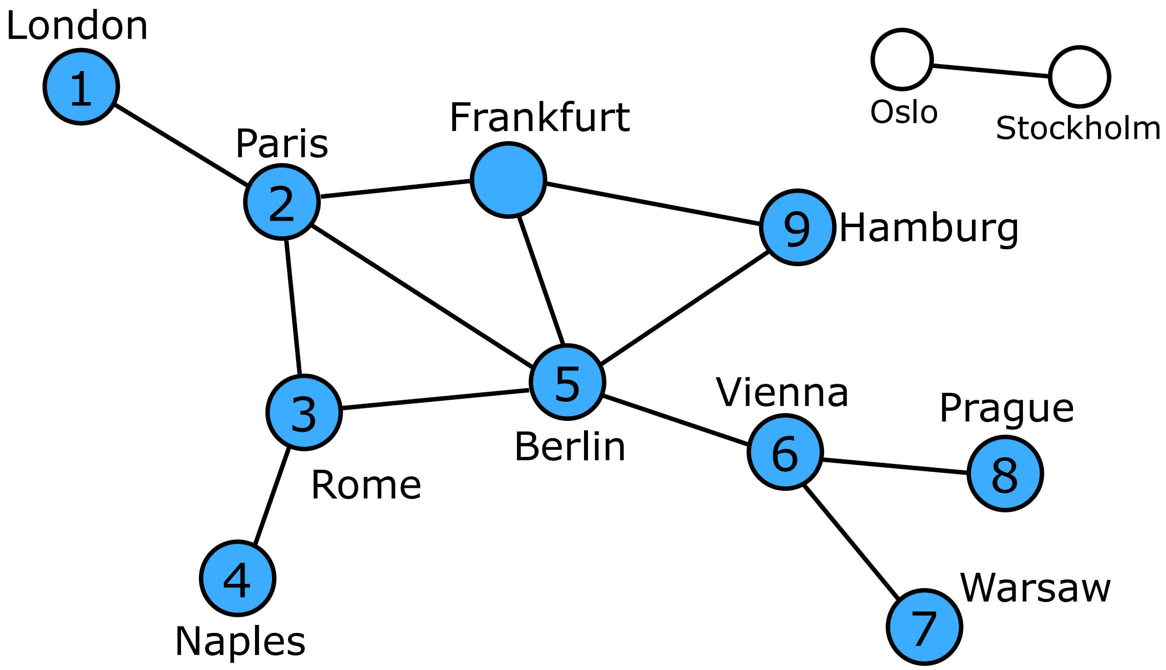
Current node: Warsaw
Todo list: [Frankfurt, Hamburg, Prague]

Graph traversal (stack)



Current node: Prague
Todo list: [Frankfurt, Hamburg]

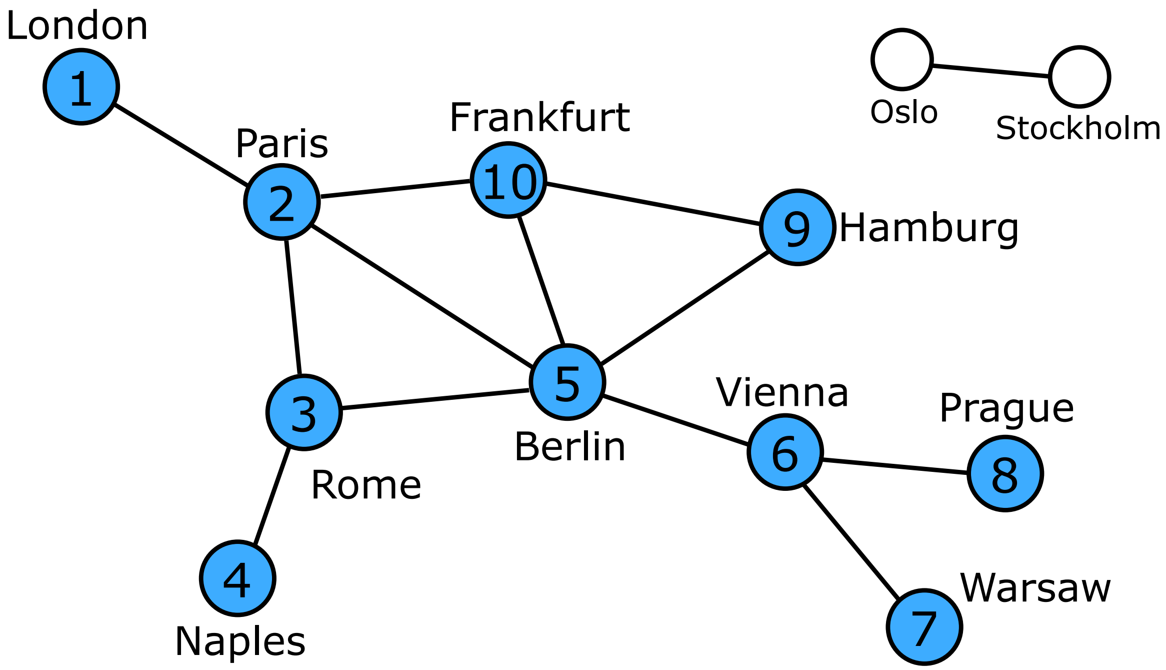
Graph traversal (stack)



Current node: Hamburg

Todo list: [Frankfurt]

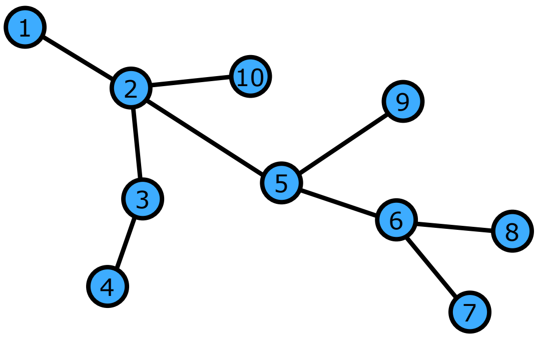
Graph traversal (stack)



Current node: Frankfurt

Todo list: [ ]

Depth-first search (DFS)



- Call the starting node the *root*
- We traverse paths all the way until we get to a dead-end, then backtrack (until we find an unexplored path)

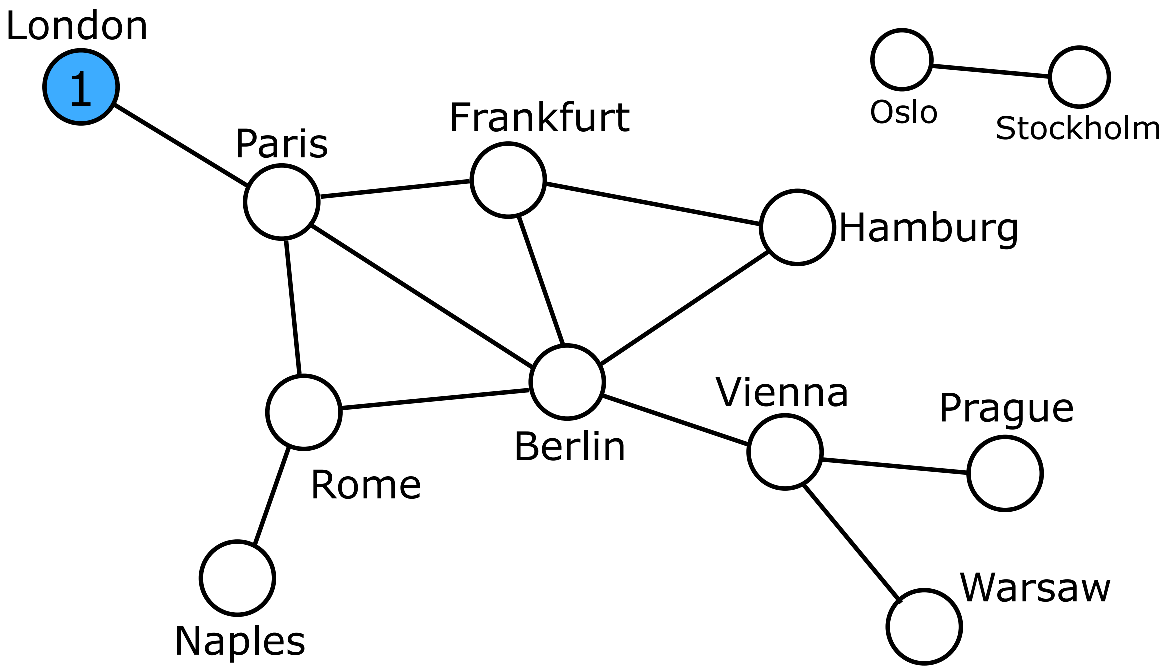
Another strategy

1. Explore all the cities that are one hop away from the root
2. Explore all cities that are two hops away from the root
3. Explore all cities that are three hops away from the root

   …

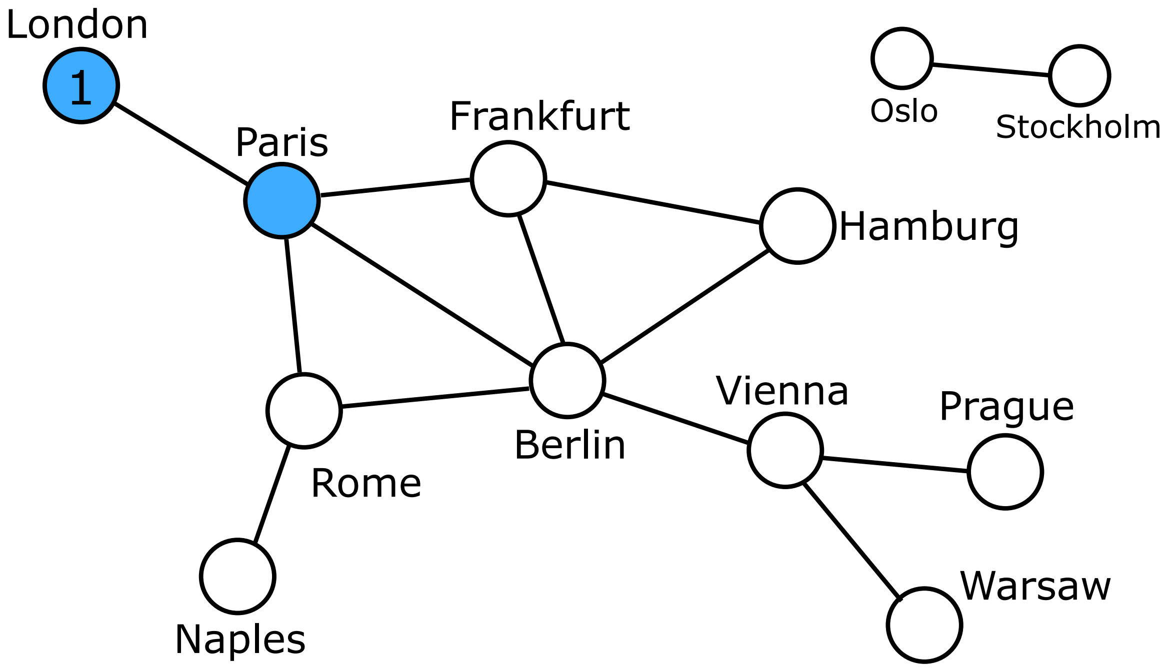
- This corresponds to using a *queue*

Graph traversal (queue)



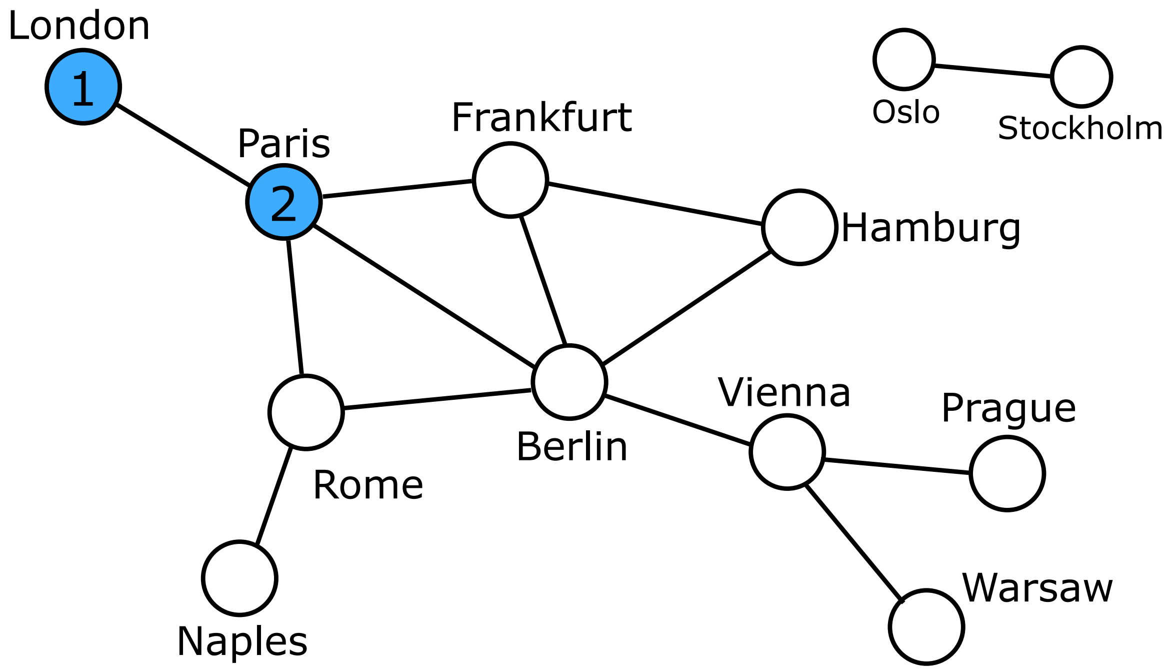
Current node: London
Todo list: [ ]

Graph traversal (queue)



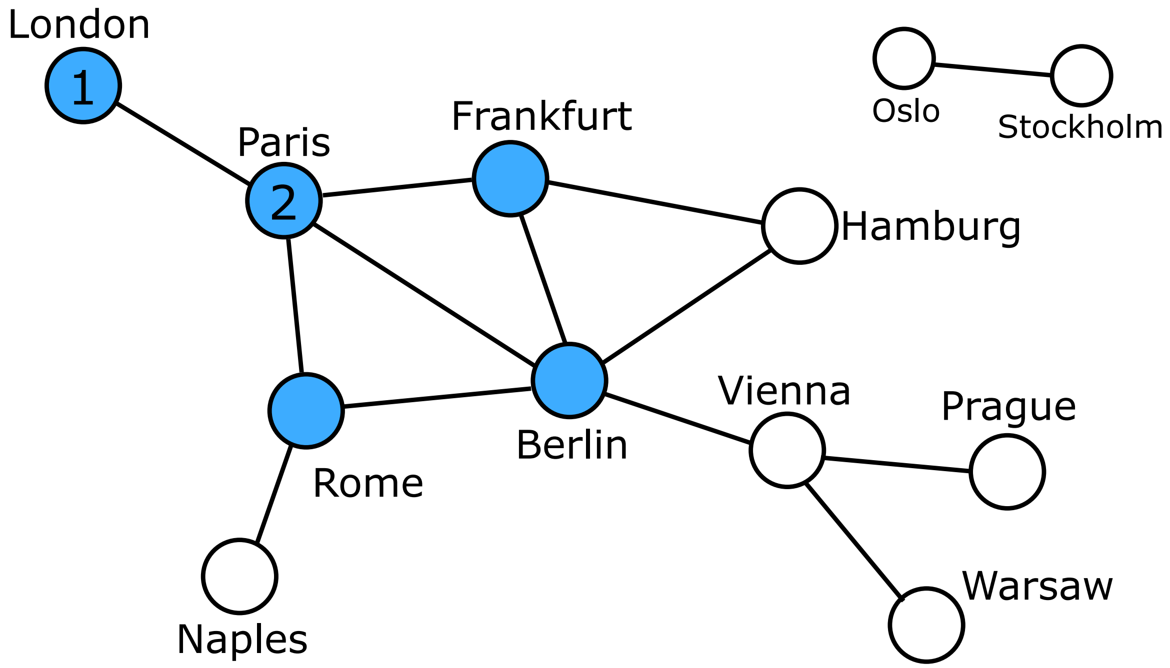
Current node: London
Todo list: [Paris]

Graph traversal (queue)



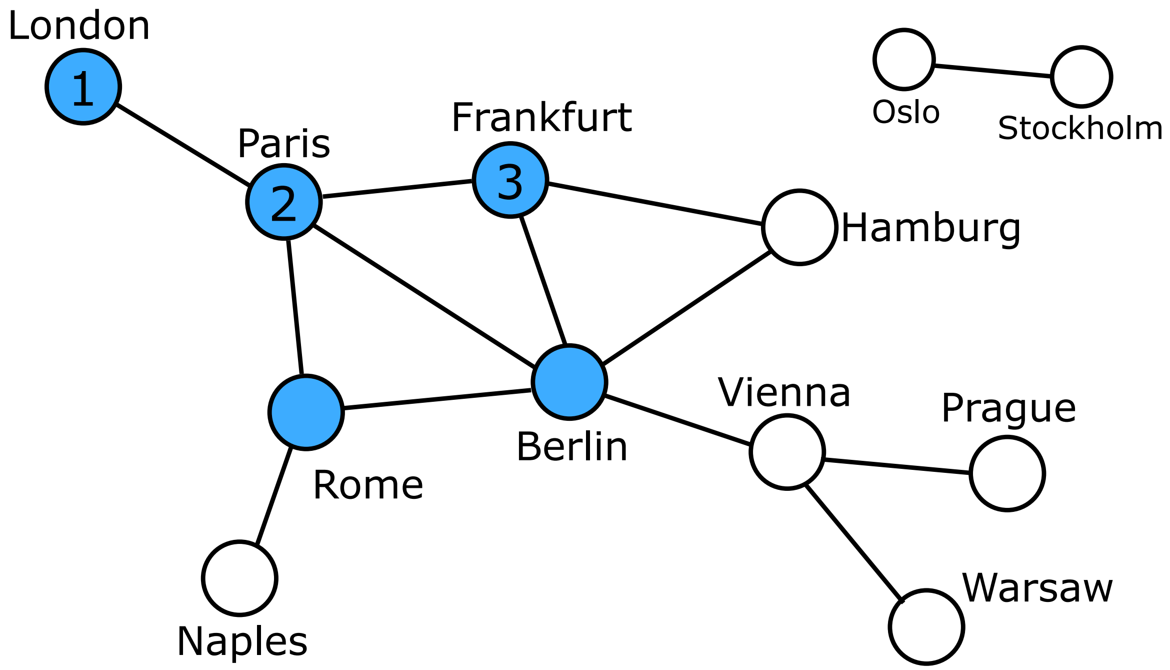
Current node: Paris
Todo list: [ ]

Graph traversal (queue)



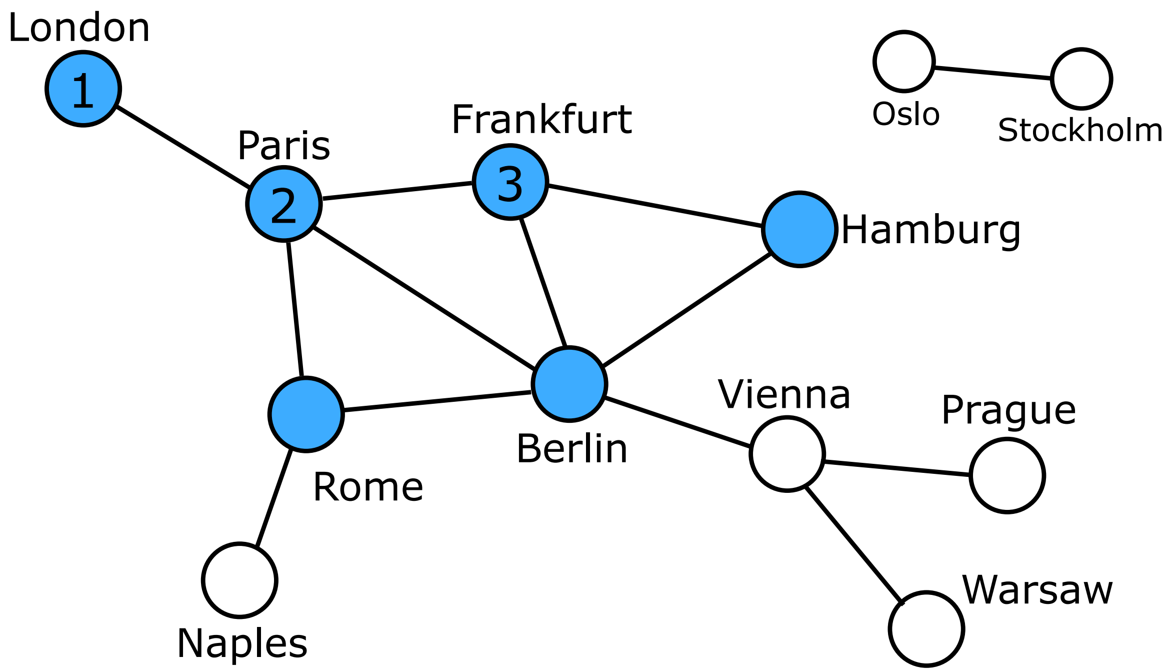
Current node: Paris
Todo list: [Frankfurt, Berlin, Rome]

Graph traversal (queue)



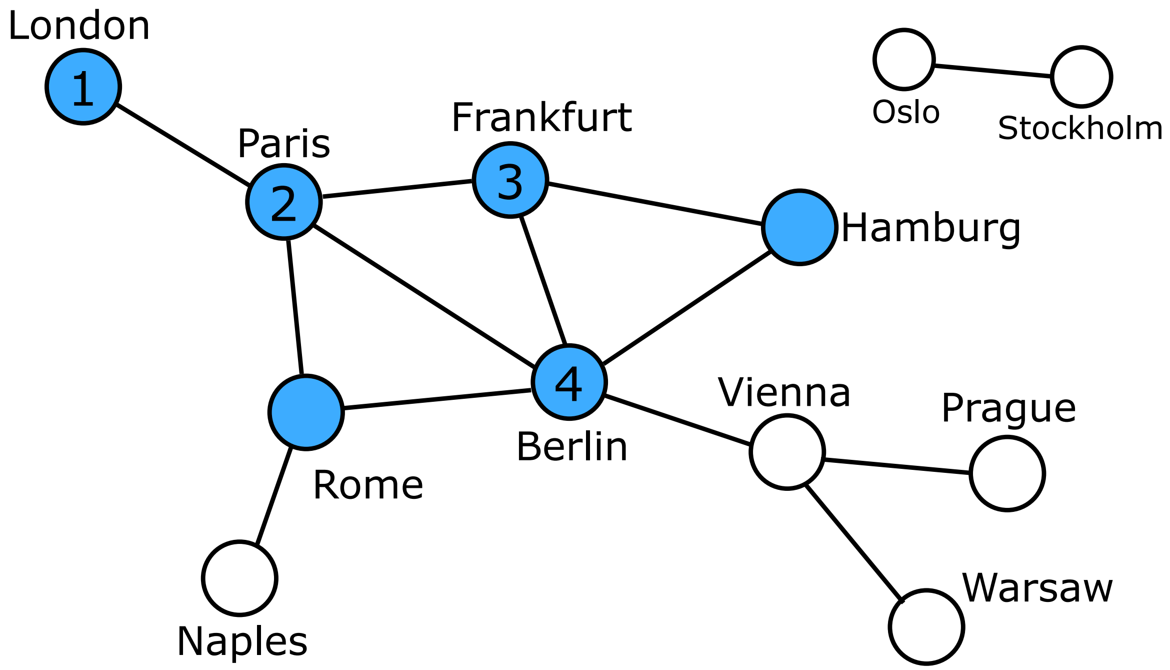
Current node: Frankfurt
Todo list: [Berlin, Rome]

Graph traversal (queue)



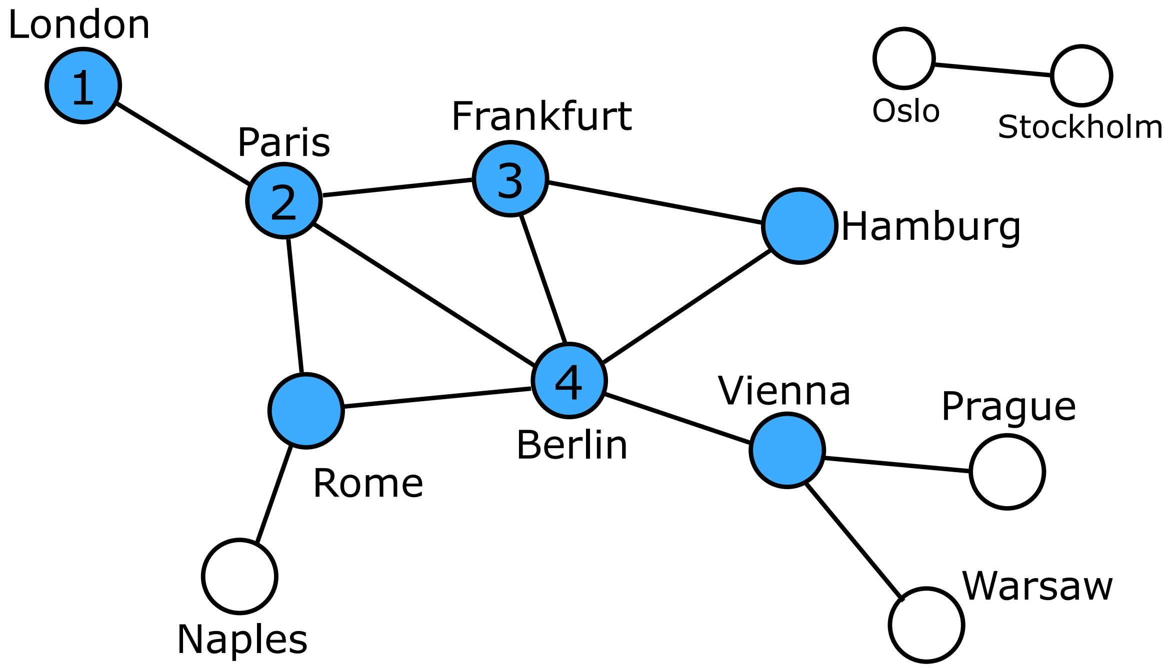
Current node: Frankfurt
Todo list: [Berlin, Rome, Hamburg]

Graph traversal (queue)



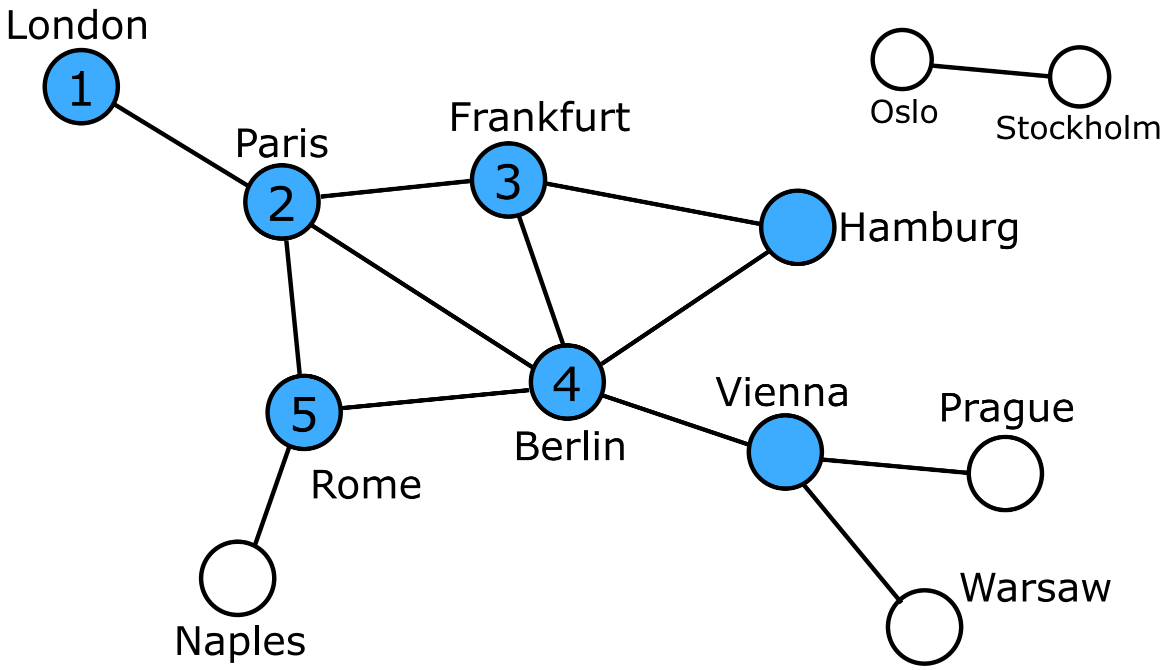
Current node: Berlin
Todo list: [Rome, Hamburg]

Graph traversal (queue)



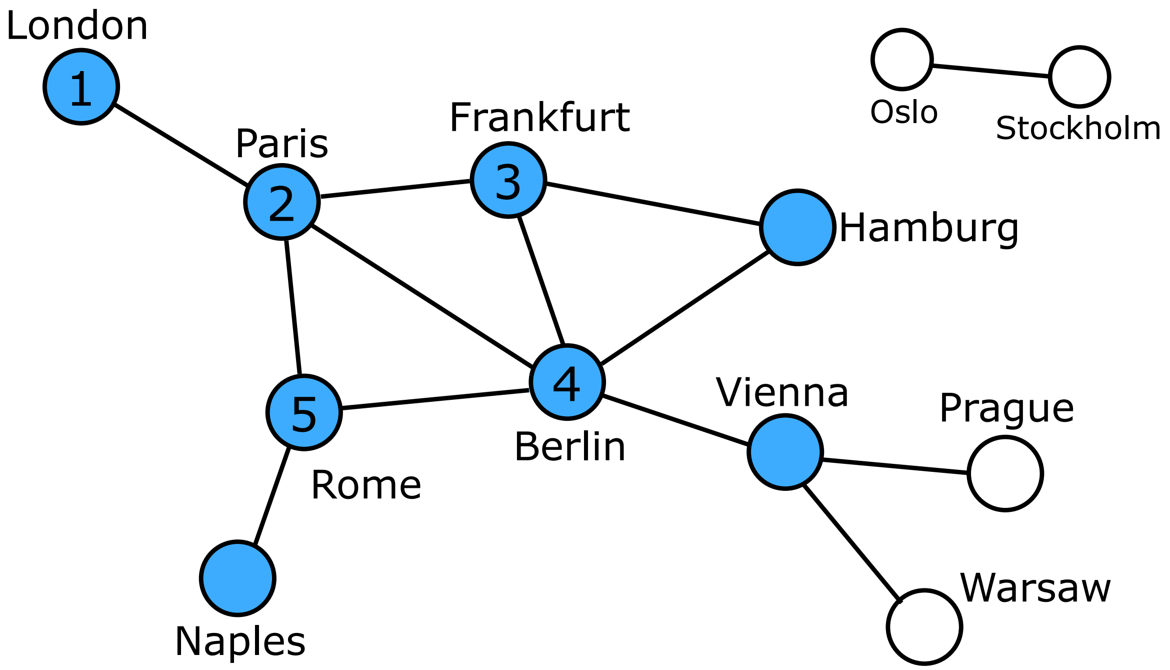
Current node: Berlin
Todo list: [Rome, Hamburg, Vienna]

Graph traversal (queue)



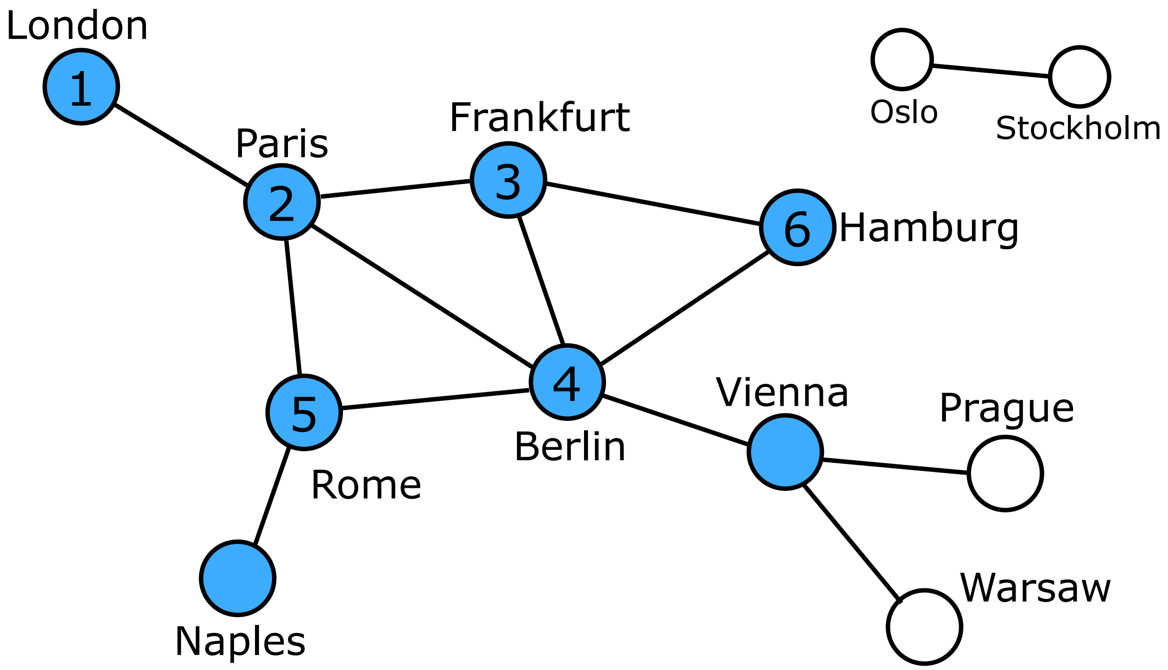
Current node: Rome
Todo list: [Hamburg, Vienna]

Graph traversal (queue)



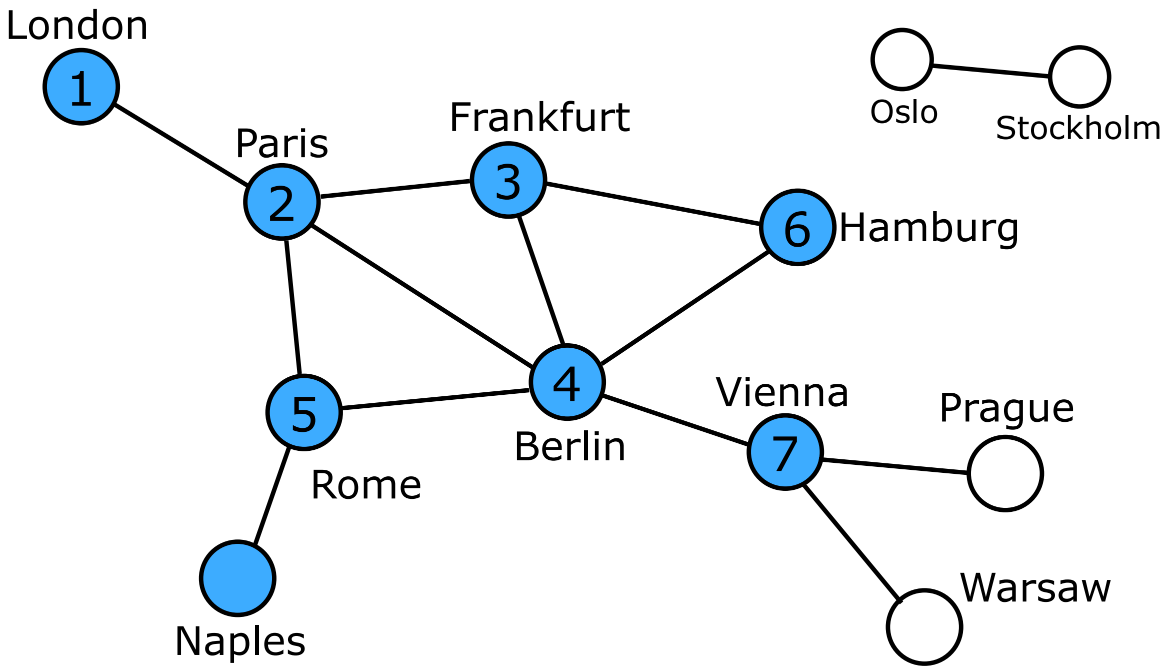
Current node: Rome
Todo list: [Hamburg, Vienna, Naples]

Graph traversal (queue)



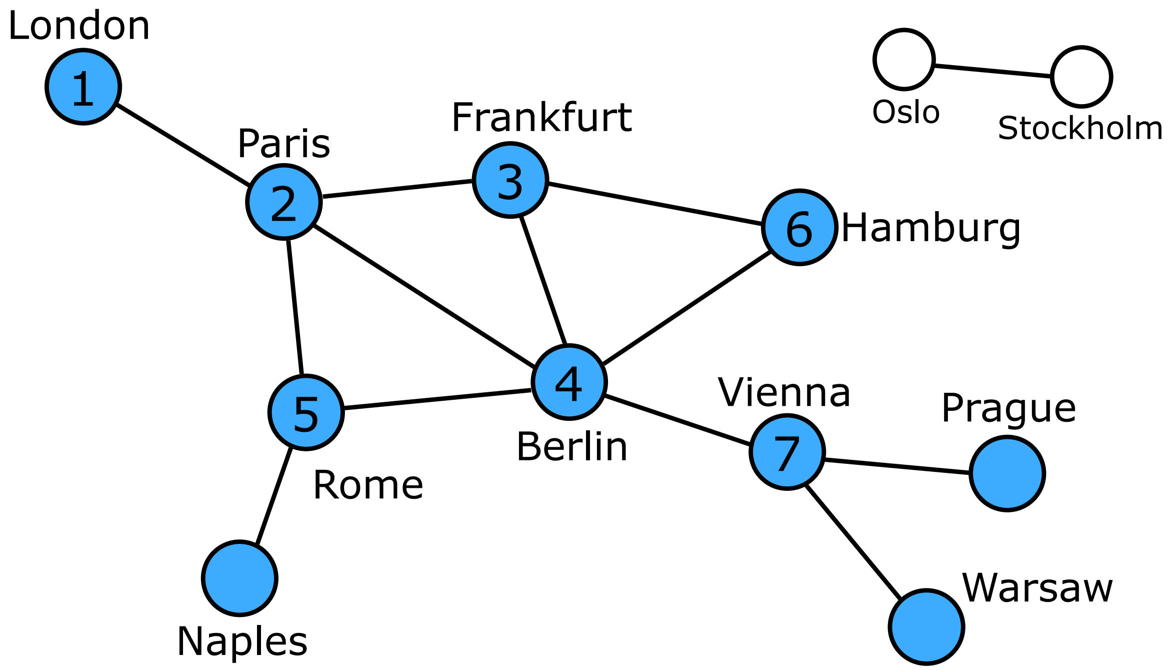
Current node: Hamburg
Todo list: [Vienna, Naples]

Graph traversal (queue)



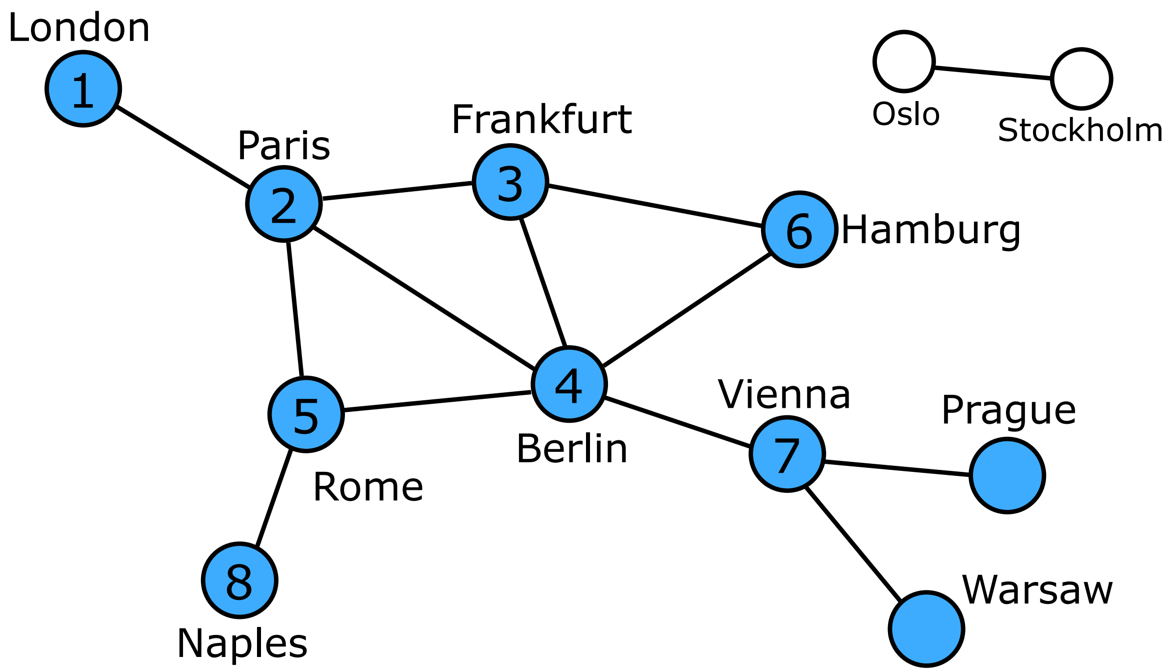
Current node: Vienna
Todo list: [Naples]

Graph traversal (queue)



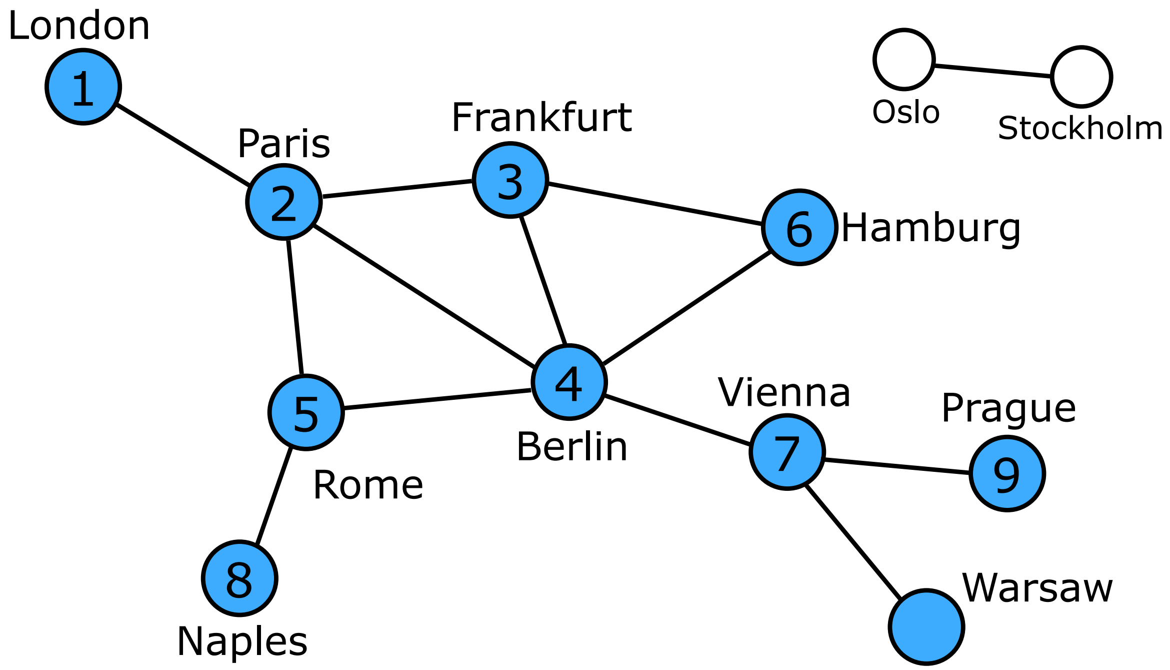
Current node: Vienna
Todo list: [Naples, Prague, Warsaw]

Graph traversal (queue)



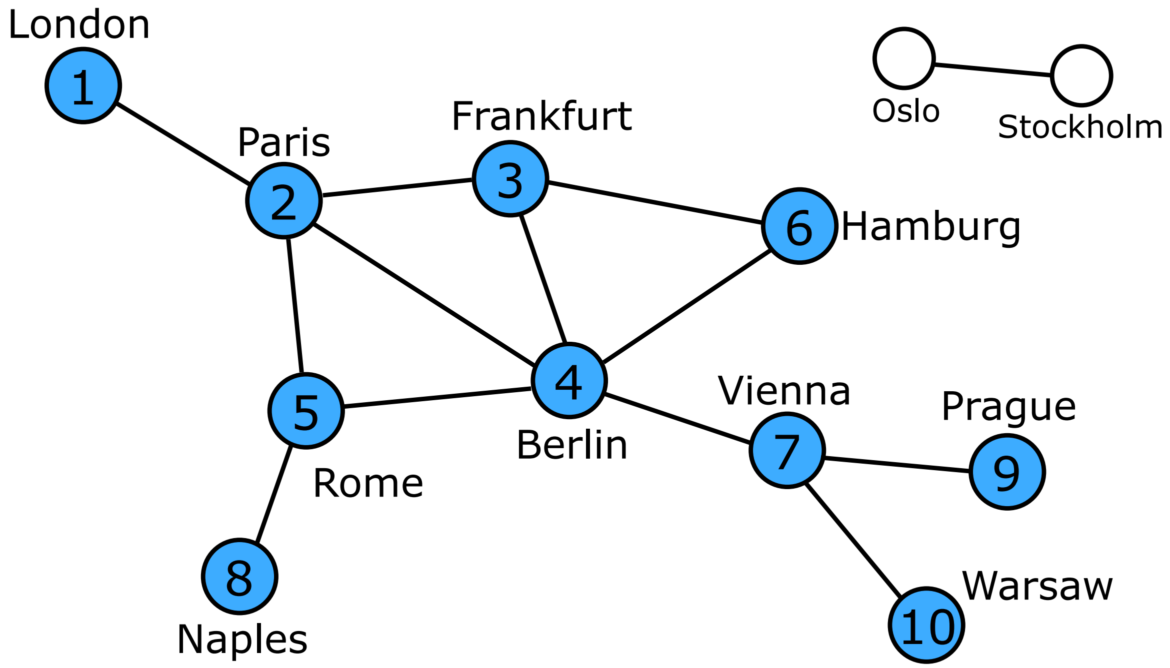
Current node: Naples
Todo list: [Prague, Warsaw]

Graph traversal (queue)



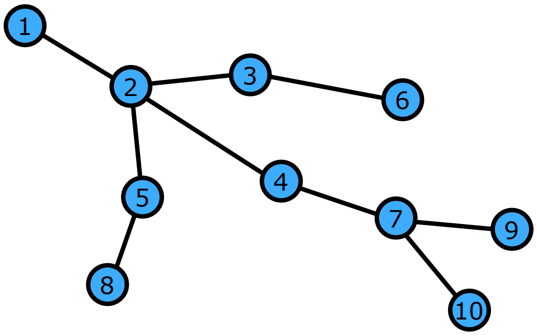
Current node: Prague
Todo list: [Warsaw]

Graph traversal (queue)



Current node: Warsaw
Todo list: [ ]

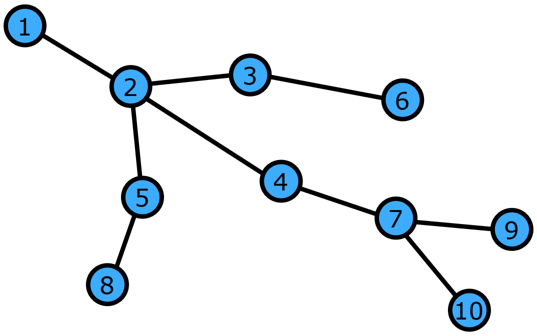
Breadth-first search (BFS)



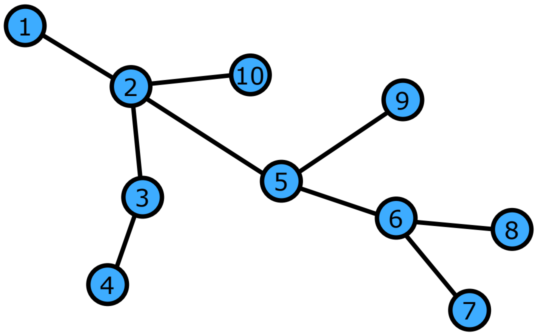
- We visit all the vertices at the same level (same distance to the root) before moving on to the next level

---

BFS vs. DFS

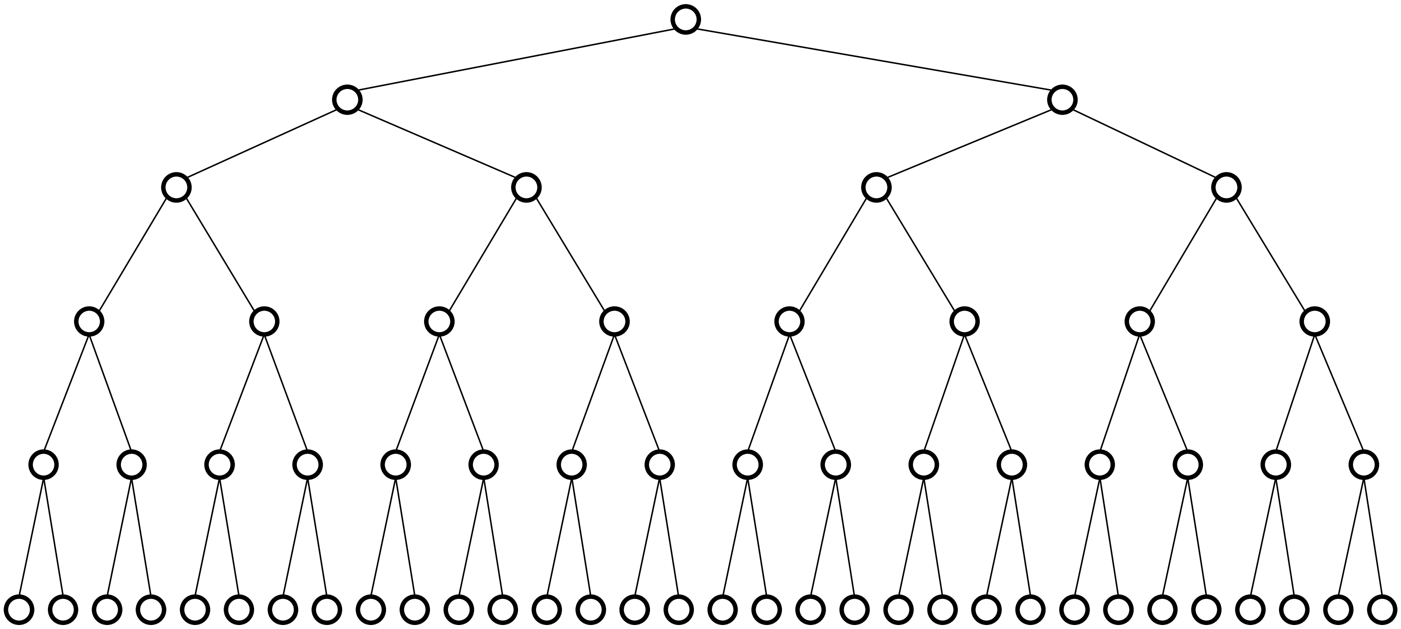


Breadth-first (queue)



Depth-first (stack)

BFS vs. DFS



(tree = graph with no cycles)