Algorithms and Datastructures

Graphs and Breadth-First Search
September 12, 2023

iCIS | Software Science
Radboud University

Graphs

Representing graphs

Breadth-First Search

Outline

Graphs

Representing graphs

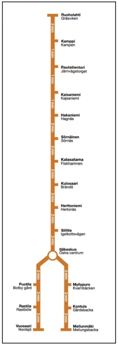
Breadth-First Search

Graphs are everywhere

Graphs are useful and interesting to computer scientists

- Many different problems in practice can be encoded as graphs
- For many problems there are efficient algorithms to solve them (content of lectures 2-7)
- But there also are problems for which an efficient algorithm might not exist! (see the course on “Complexity”)
- Large amount of applications!

Graphs are in Helsinki

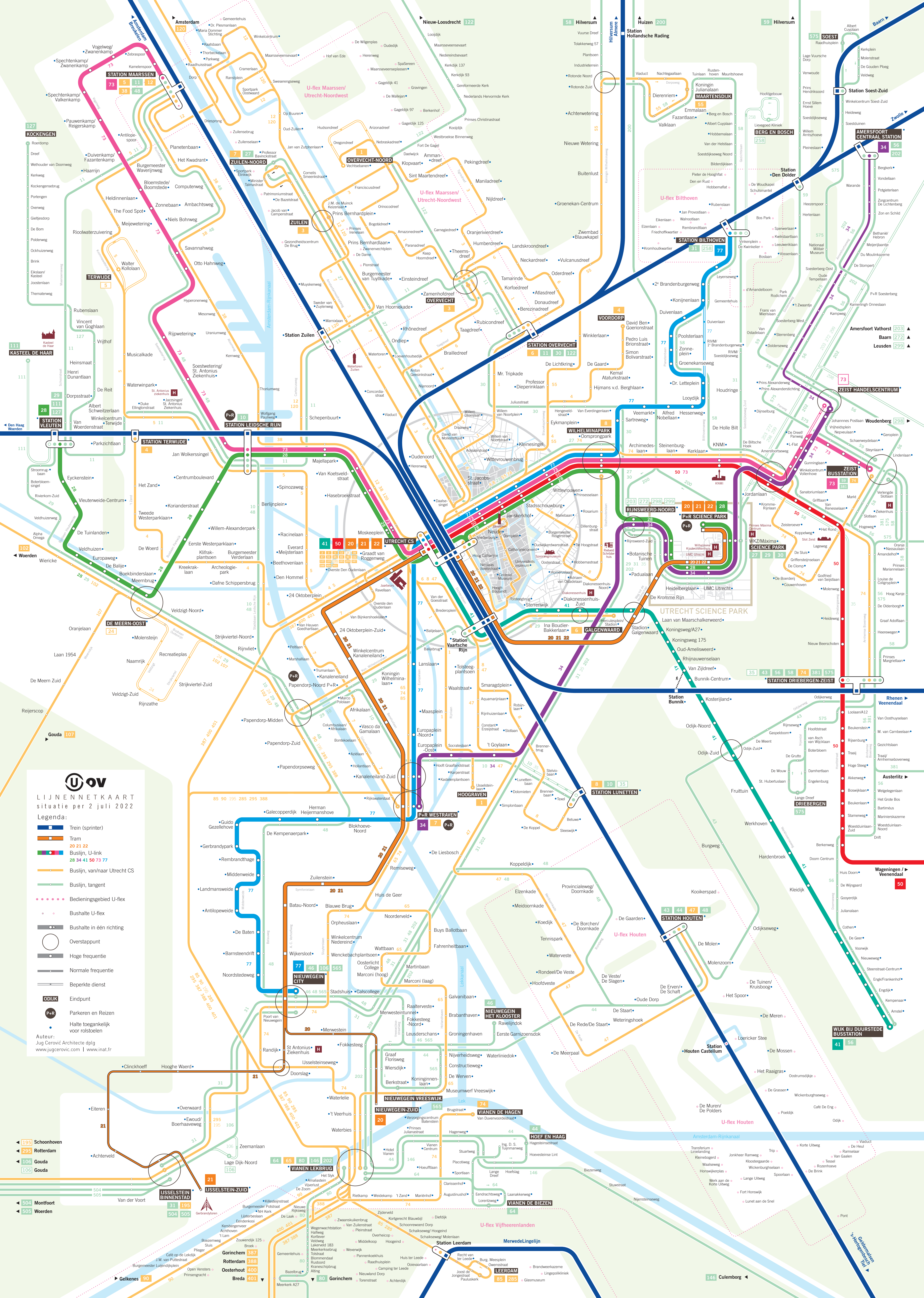


Question: what is the shortest route from Hakaniemi to Puotila?

Source: https://www.hel.fi/helsinki/en/maps-and-transport/transport/metro/

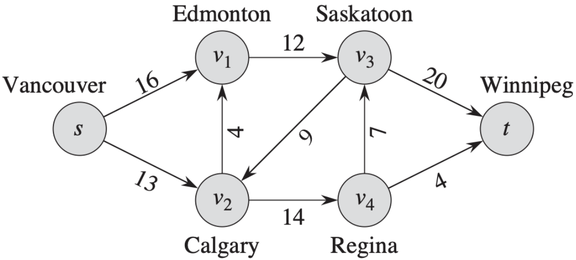
Graphs are also in Utrecht

Source: https://www.u-ov.info/reizen/kaarten-en-plattegronden


U OV
LIJNENNETKAART
situatie per 2 juli 2022
Legenda:
Trein (sprinter)
Tram
20 21 22
Buslijn, U-link
28 34 41 50 73 77
Buslijn, van/naar Utrecht CS
Buslijn, tangent
Bedieningsgebied U-flex
Bushalte U-flex
Bushalte in één richting
Overstappunt
Hoge frequentie
Normale frequentie
Beperkte dienst
ODIJK Eindpunt
Parkeren en Reizen
Halte toegankelijk voor rolstoelen
Auteur:
Jug Cerović Architecte dplg
www.jugcerovic.com | www.inat.fr
UTRECHT CS
UTRECHT SCIENCE PARK
STATION MAARSSEN
STATION ZUILEN
STATION TERWIJDE
STATION LEIDSCHE RIJN
STATION VLEUTEN
STATION OVERVECHT
STATION BILTHOVEN
STATION LUNETTEN
STATION HOUTEN
STATION DRIEBERGEN-ZEIST
AMERSFOORT CENTRAAL STATION
ZEIST HANDELSCENTRUM
ZEIST BUSSTATION
NIEUWEGEIN CITY
IJSSELSTEIN-ZUID
VIANEN LEKBRUG
U-flex Maarssen/Utrecht-Noordoost
U-flex Maarssen/Utrecht-Noordwest
U-flex Bilthoven
U-flex Houten
U-flex Vijfheerenlanden
Amsterdam-Rijnkanaal
MerwedeLingelijn
KOCKENGEN
KASTEEL DE HAAR
DE MEERN-OOST
P+R WESTRAVEN
P+R SCIENCE PARK
GALGENWAARD
WIJK BIJ DUURSTEDE BUSSTATION
DRIEBERGEN
VIANEN DE HAGEN
HOEF EN HAAG
VIANEN DE BIEZEN
LEERDAM
Station Leerdam
Gorinchem
Culemborg
Hilversum
Amsterdam
Woerden
Gouda
Rhenen Veenendaal
Wageningen / Veenendaal

Graphs are in Canada

We have cities, roads between them, and every road has a capacity.



Question: how much can we bring from the factory in Vancouver to the warehouse in Winnipeg?

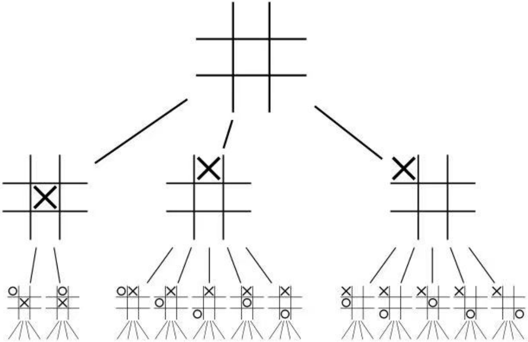
Source: Figure 26.1 in Cormen, Thomas H., et al. *Introduction to algorithms*. MIT press, 2022.

Graphs are in your social networks

Question: which friends should you recommend to people?

Source: https://medium.com/analytics-vidhya/social-network-analytics-f082f4e21b16

Graphs are in your games



Question: is there a winning strategy? Can I still win from this position?

Source: https://en.wikipedia.org/wiki/Breadth-first_search

Graphs are in your electrical grids

Below: number of miles of electrical line needed to make a power line between two cities.

	Ash.	Ast.	B.	C.	C.L.	E.	N.	P.	Sal.	Sea.
Ashland	-	374	200	223	108	178	252	285	240	356
Astoria	374	-	255	166	433	199	135	95	136	17
Bend	200	255	-	128	277	128	180	160	131	247
Corvallis	223	166	128	-	430	47	52	84	40	155
Crater Lake	108	433	277	430	-	453	478	344	389	423
Eugene	178	199	128	47	453	-	91	110	64	181
Newport	252	135	180	52	478	91	-	114	83	117
Portland	285	95	160	84	344	110	114	-	47	78
Salem	240	136	131	40	389	64	83	47	-	118
Seaside	356	17	247	155	423	181	117	78	118	-

Question: which power grid requires the least amount of new line?

Source: https://mathbooks.unl.edu/Contemporary/sec-graph-tree.html

And graphs are in many other places!

Other applications of graphs:

- Airline scheduling: given a flight schedule, can we execute it with at most k planes?[1]
- What is the shortest route to visit every Dutch monument?[2]
- Timetable scheduling[3]
- Network design (telecommunication networks)[4]

and more...

[1] Kleinberg, Jon, and Eva Tardos. *Algorithm design*. Pearson Education India, 2006.
[2] https://cqm.nl/uploads/media/613b23fe72d33/nrc-20210910-monumentenroute.pdf
[3] Burke, E. K., D. G. Elliman, and R. Weare. "A university timetabling system based on graph colouring and constraint manipulation." *Journal of research on computing in education* 27.1 (1994): 1-18.
[4] Korte, Bernhard, and Jens Vygen. "Combinatorial Optimization." (2017).

Structure of the first part of the course

- **Search algorithms**: breadth-first search (lecture 2), depth-first search (lecture 3)
- **Shortest path algorithms**: Dijkstra's algorithm (lecture 4)
- **Flow algorithms**: Ford-Fulkerson (lecture 5), Edmonds-Karp (lecture 6)
- **Greedy algorithms**. In particular, algorithms for **minimal spanning trees**: Kruskal's algorithm, Prim's algorithm (lecture 7)

Outline

Graphs

Representing graphs

Breadth-First Search

Basic terminology

Definition
A **graph** G consists of

- a set V of **vertices** (also called **nodes**)
- a set $E \subseteq V \times V$ of **edges**

Basic terminology

Definition
A **graph** G consists of

- a set V of **vertices** (also called **nodes**)
- a set $E \subseteq V \times V$ of **edges**

More terminology:

- Given $v, w \in V$, we write $v \to w$ if there is an edge from v to w.
- Formally, this means $(v, w) \in E$.
- If we have an edge e from v to w, then we say v is the **source** of e and w is the **target** of e
- We can have $v \to v$.
- If we have $v \to w$, then v and w are **adjacent**

Different varieties of graphs

- **Directed graphs**: the definition we saw on the previous slide

Different varieties of graphs

- **Directed graphs**: the definition we saw on the previous slide
- **Undirected graphs**: if $v \to w$, then also $w \to v$
  Often self-loops are not allowed in undirected graphs
  Edges can be represented as **sets** of 2 vertices

Different varieties of graphs

- **Directed graphs**: the definition we saw on the previous slide
- **Undirected graphs**: if $v \rightarrow w$, then also $w \rightarrow v$
  Often self-loops are not allowed in undirected graphs
  Edges can be represented as **sets** of 2 vertices
- **Weighted**: every edge e has a weight $c(e) \in \mathbb{R}$

Different varieties of graphs

- **Directed graphs**: the definition we saw on the previous slide
- **Undirected graphs**: if $v \rightarrow w$, then also $w \rightarrow v$
  Often self-loops are not allowed in undirected graphs
  Edges can be represented as **sets** of 2 vertices
- **Weighted**: every edge e has a weight $c(e) \in \mathbb{R}$
- **Dense**: $|E| \approx |V|^2$

Different varieties of graphs

- **Directed graphs**: the definition we saw on the previous slide
- **Undirected graphs**: if $v \to w$, then also $w \to v$
  Often self-loops are not allowed in undirected graphs
  Edges can be represented as **sets** of 2 vertices
- **Weighted**: every edge e has a weight $c(e) \in \mathbb{R}$
- **Dense**: $|E| \approx |V|^2$
- **Sparse**: $|E| \ll |V|^2$

Different varieties of graphs

- **Directed graphs**: the definition we saw on the previous slide
- **Undirected graphs**: if $v \to w$, then also $w \to v$
  Often self-loops are not allowed in undirected graphs
  Edges can be represented as **sets** of 2 vertices
- **Weighted**: every edge e has a weight $c(e) \in \mathbb{R}$
- **Dense**: $|E| \approx |V|^2$
- **Sparse**: $|E| \ll |V|^2$
- **Directed acyclic graphs**: we will see them later

Different varieties of graphs

- **Directed graphs**: the definition we saw on the previous slide
- **Undirected graphs**: if $v \to w$, then also $w \to v$
  Often self-loops are not allowed in undirected graphs
  Edges can be represented as **sets** of 2 vertices
- **Weighted**: every edge e has a weight $c(e) \in \mathbb{R}$
- **Dense**: $|E| \approx |V|^2$
- **Sparse**: $|E| \ll |V|^2$
- **Directed acyclic graphs**: we will see them later
- **Multigraphs**: there could be multiple edges between two vertices

The number of edges

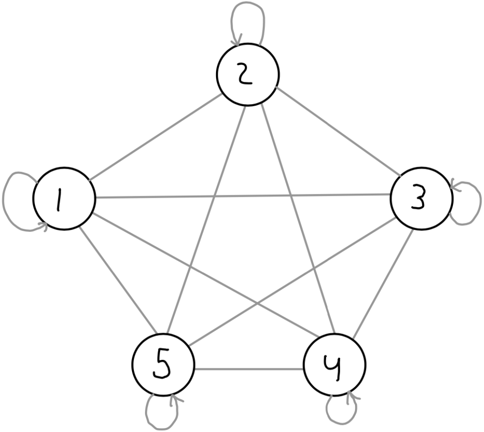
Let $G = (V, E)$ be a graph with $|V|$ vertices.
Can you give an upper bound for the number of edges in G?

The number of edges

Let $G = (V, E)$ be a graph with $|V|$ vertices.
Can you give an upper bound for the number of edges in G?
The number of edges is in $\mathcal{O}(|V|^2)$.
So: $|E| \in \mathcal{O}(|V|^2)$.

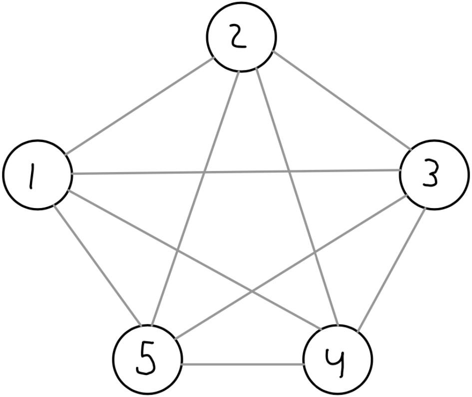
The number of edges (directed graphs)

Directed graphs: at most n^2 edges



The number of edges (undirected graphs)

Undirected graphs: at most $\binom{n}{2} = \frac{n^2-n}{2}$ edges



Interface for graphs

On graphs, we have the following operations

- Get the set of vertices (**vertex**)
- Given two vertices, is there an edge between them? (**edge**)
- Given a vertex, get all the vertices adjacent to it (**adjacent**)

Interface for graphs

On graphs, we have the following operations

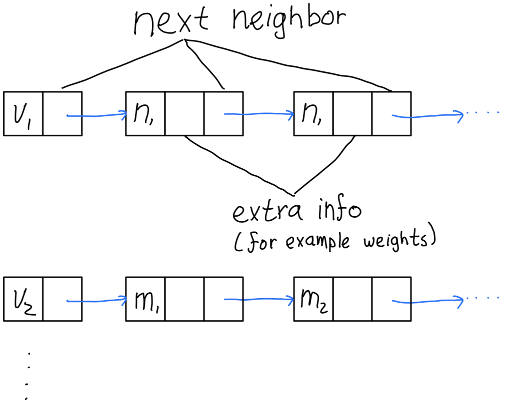
- Get the set of vertices (**vertex**)
- Given two vertices, is there an edge between them? (**edge**)
- Given a vertex, get all the vertices adjacent to it (**adjacent**)

We could also consider other operations such as adding/removing vertices, adding/removing edges, and so on

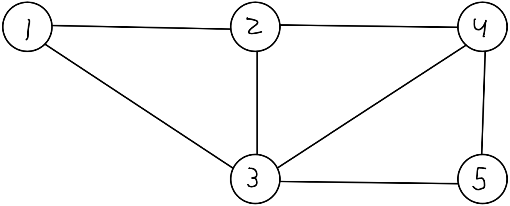
Representation 1: Adjacency lists

Idea:

For each vertex v, store a list of vertices adjacent to v



Example



The adjacency list of this graph:

1	2, 3
2	1, 3, 4
3	1, 2, 4, 5
4	2, 3, 5
5	3, 4

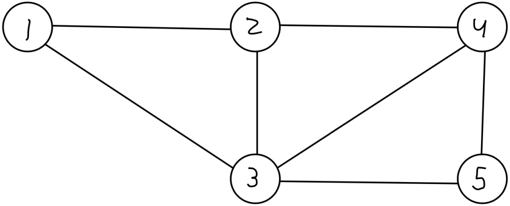
Representation 2: Adjacency matrices

Idea:

> Label the vertices are $1, \ldots, n$. We store a matrix such that position (i, j) is 1 if we have an edge from i to j and a 0 otherwise.

Note: you can also take the weight of edges into account by storing the weight instead of just 0 or 1.

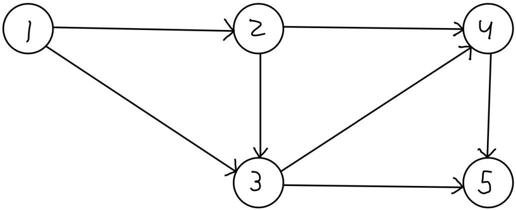
Example



The adjacency matrix of this graph:

	1	2	3	4	5
1	0	1	1	0	0
2	1	0	1	1	0
3	1	1	0	1	1
4	0	1	1	0	1
5	0	0	1	1	0

Example



The adjacency matrix of this graph:

	1	2	3	4	5
1	0	1	1	0	0
2	0	0	1	1	0
3	0	0	0	1	1
4	0	0	0	0	1
5	0	0	0	0	0

Complexity of the operations

	Adjacency list	Adjacency matrix
edge	$\mathcal{O}(\lvert V \rvert)$	$\mathcal{O}(1)$
adjacent	$\mathcal{O}(\lvert V \rvert)$	$\mathcal{O}(\lvert V \rvert)$
Space complexity	$\mathcal{O}(\lvert V \rvert + \lvert E \rvert)$	$\mathcal{O}(\lvert V \rvert^2)$

Quiz time

Which representation of graphs would you use in the following cases?

- The public transport network of the EU
  Vertices: bus/train/tram/subway stops
  Edges: bus/train/tram/subway lines

Quiz time

Which representation of graphs would you use in the following cases?

- The public transport network of the EU
  Vertices: bus/train/tram/subway stops
  Edges: bus/train/tram/subway lines
- Electrical grids
  Vertices: villages/towns/cities in the Netherlands
  Weight on edges: length of power line to connect them

Quiz time

Which representation of graphs would you use in the following cases?

- The public transport network of the EU
  Vertices: bus/train/tram/subway stops
  Edges: bus/train/tram/subway lines
- Electrical grids
  Vertices: villages/towns/cities in the Netherlands
  Weight on edges: length of power line to connect them
- Social networks
  Vertices: people
  Edges: there is an edge between people if they are friends

Quiz time

Which representation of graphs would you use in the following cases?

- The public transport network of the EU
  Vertices: bus/train/tram/subway stops
  Edges: bus/train/tram/subway lines
- Electrical grids
  Vertices: villages/towns/cities in the Netherlands
  Weight on edges: length of power line to connect them
- Social networks
  Vertices: people
  Edges: there is an edge between people if they are friends

Rule of thumb:

- If the graph is **dense**, adjacency matrices are "better"
- If the graph is **sparse**, adjacency lists are "better"

Quiz time

Which representation of graphs would you use in the following cases?

- The public transport network of the EU
  Vertices: bus/train/tram/subway stops
  Edges: bus/train/tram/subway lines
- Electrical grids
  Vertices: villages/towns/cities in the Netherlands
  Weight on edges: length of power line to connect them
- Social networks
  Vertices: people
  Edges: there is an edge between people if they are friends

Rule of thumb:

- If the graph is **dense**, adjacency matrices are "better"
- If the graph is **sparse**, adjacency lists are "better"

Outline

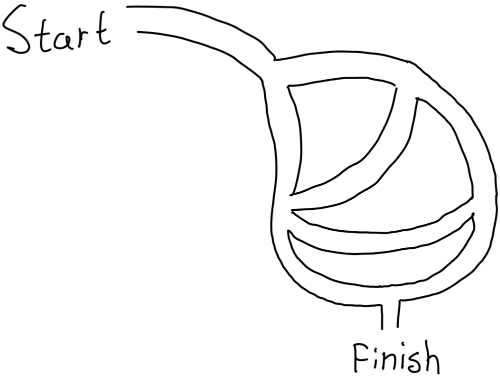
Graphs

Representing graphs

Breadth-First Search

Problem setting: motivation

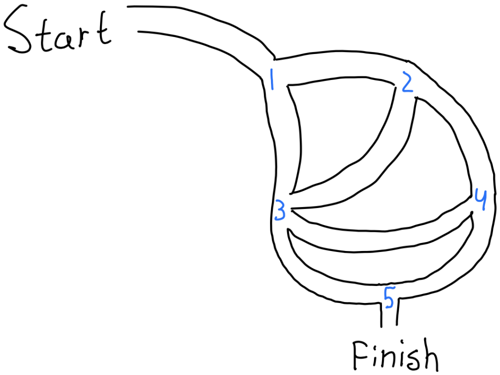
Look at the following maze



Question: can you go from “Start” to “Finish”?

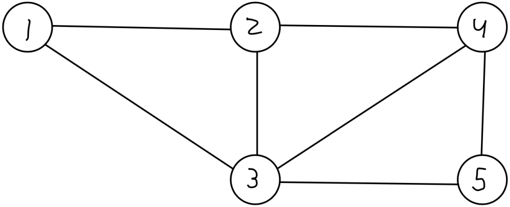
It is a graph problem!

We label all the crossings



It is a graph problem!

And we turn it into a graph!



Question: can we reach the vertex 5 from the vertex 1?

Problem setting: reachability

General problem:
given a graph G and a vertex v, return the list of vertices that can be reached from v.

You can also consider modifications. For example, can we reach a vertex for which a certain property holds?

Intermezzo: connected graphs

Let G be a graph

- A **path** from v to w is a list of edges: $v \to v_1 \to \ldots \to v_n \to w$
- Note: paths can be empty

Intermezzo: connected graphs

Let G be a graph

- A **path** from v to w is a list of edges: $v \to v_1 \to \ldots \to v_n \to w$
- Note: paths can be empty
- A graph is called **connected** if for all vertices v and w there is a path $v \to w$
- The **connected component** of v is the list of all vertices w for which there is a path from v to w

Intermezzo: connected graphs

Let G be a graph

- A **path** from v to w is a list of edges: $v \to v_1 \to \ldots \to v_n \to w$
- Note: paths can be empty
- A graph is called **connected** if for all vertices v and w there is a path $v \to w$
- The **connected component** of v is the list of all vertices w for which there is a path from v to w

The problem can also be formulated as follows:
given a graph G and a vertex v, determine the connected component of v.

Graph Searching

In the upcoming lectures, we shall consider three search algorithms for graphs

- Breadth-first search
- Depth-first search
- Dijkstra's algorithm

These algorithms follow the same idea.

Graph Searching: Idea

We divide the graph into three parts

- **Explored** vertices: these are vertices that we already visited
- **The frontier**: vertices that are adjacent to one of the explored vertices
- **Undiscovered vertices**: all other vertices

Discovered vertices: either explored or in the frontier

Graph Searching: Basic Algorithm

The graph searching algorithms that we discuss, work as follows:

- At every step, we pick a vertex from the frontier
- We label that vertex as an explored
- All undiscovered vertices adjacent to that vertex, are put in the frontier
- We continue this process until there are no vertices left in the frontier

Graph Searching: Basic Algorithm

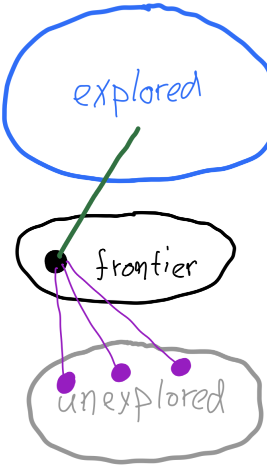
The graph searching algorithms that we discuss, work as follows:

- At every step, we pick a vertex from the frontier
- We label that vertex as an explored
- All undiscovered vertices adjacent to that vertex, are put in the frontier
- We continue this process until there are no vertices left in the frontier

Differences between search algorithms:

- Different ways of picking vertices from the frontier
- Exploring vertices might require some additional steps (see Lecture 4)

Graph Searching



Necessary data structure: queues

For breadth first search, we represent the frontier with a **queue**.
For queues, we have the following operations:

- Return the empty queue
- Determine whether the queue is empty
- Enqueue: add an element to the back of the queue
- Dequeue: return and remove the front element from the queue

Necessary data structure: queues

For breadth first search, we represent the frontier with a **queue**.
For queues, we have the following operations:

- Return the empty queue
- Determine whether the queue is empty
- Enqueue: add an element to the back of the queue
- Dequeue: return and remove the front element from the queue

For example:

Enqueue 3 to $[1, 2]$ gives $[1, 2, 3]$

If we dequeue $[1, 2, 3]$, we get 1 and the queue becomes $[2, 3]$

Necessary data structure: queues

For breadth first search, we represent the frontier with a **queue.**
For queues, we have the following operations:

- Return the empty queue
- Determine whether the queue is empty
- Enqueue: add an element to the back of the queue
- Dequeue: return and remove the front element from the queue

For example:

Enqueue 3 to $[1, 2]$ gives $[1, 2, 3]$

If we dequeue $[1, 2, 3]$, we get 1 and the queue becomes $[2, 3]$

For implementation: see prerecorded lecture by Frits Vaandrager!

Breadth-first search: the data

We maintain:

- A queue Q: next vertices to explore
- An array explored: whether a vertex is already explored
- An array predecessor: the previous vertex

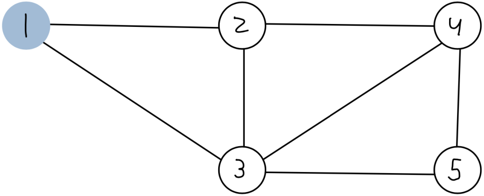
Breadth-first search: the algorithm

```
enum State := { UNDISCOVERED, DISCOVERED }

void bfs(G, v)
    // initialize
    for each u in vertex(G) unequal v
        explored[u] := UNDISCOVERED
        predecessor[u] := null
    explored[v] := DISCOVERED
    predecessor[v] := null
    Q := emptyQueue
    enqueue(Q, v)
    // main loop
    while (!isEmpty(Q))
        u := dequeue(Q)
        for each w in adjacent(u)
            if (explored[w] == UNDISCOVERED)
                explored[w] := DISCOVERED
                predecessor[w] := u
                enqueue(Q, w)
```

An example

Initialize the algorithm

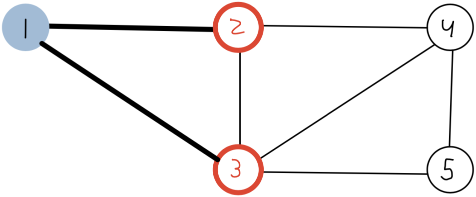


$$Q = [] \qquad \begin{bmatrix} 1 & \textbf{DISCOVERED} \\ 2 & \textbf{UNDISCOVERED} \\ 3 & \textbf{UNDISCOVERED} \\ 4 & \textbf{UNDISCOVERED} \\ 5 & \textbf{UNDISCOVERED} \end{bmatrix}$$

Filled blue: dequeued, **thick red**: in queue, **thick path**: predecessor

An example

Add the neighbors of 1 to the queue

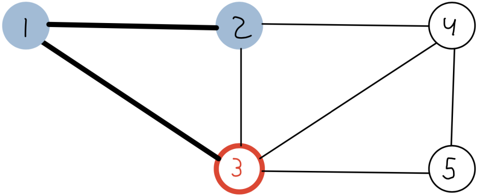


$$Q = [2, 3] \quad \begin{bmatrix} 1 & \textbf{DISCOVERED} \\ 2 & \textbf{DISCOVERED} \\ 3 & \textbf{DISCOVERED} \\ 4 & \textbf{UNDISCOVERED} \\ 5 & \textbf{UNDISCOVERED} \end{bmatrix}$$

Filled blue: discovered, **thick red**: in queue, **thick path**: predecessor

An example

2 is explored

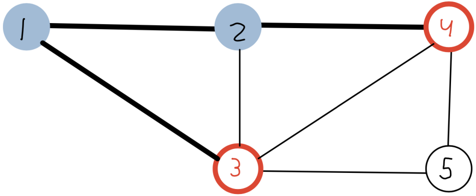


$$Q = [3] \qquad \begin{bmatrix} 1 & \textbf{DISCOVERED} \\ 2 & \textbf{DISCOVERED} \\ 3 & \textbf{DISCOVERED} \\ 4 & \textbf{UNDISCOVERED} \\ 5 & \textbf{UNDISCOVERED} \end{bmatrix}$$

Filled blue: discovered, **thick red**: in queue, **thick path**: predecessor

An example

Add the neighbors of 2 to the queue



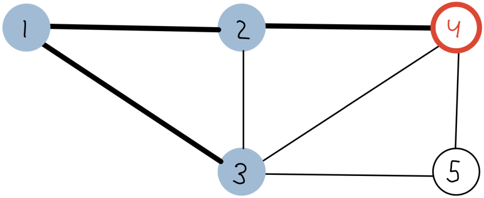
$Q = [3, 4]$

$$\begin{bmatrix} 1 & \textbf{DISCOVERED} \\ 2 & \textbf{DISCOVERED} \\ 3 & \textbf{DISCOVERED} \\ 4 & \textbf{DISCOVERED} \\ 5 & \textbf{UNDISCOVERED} \end{bmatrix}$$

Filled blue: discovered, **thick red**: in queue, **thick path**: predecessor

An example

Explore 3

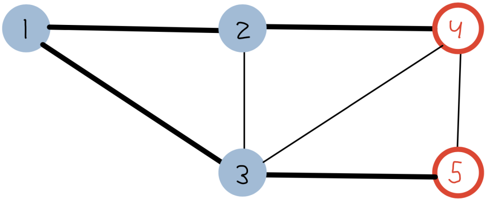


$$Q = [4] \qquad \begin{bmatrix} 1 & \textbf{DISCOVERED} \\ 2 & \textbf{DISCOVERED} \\ 3 & \textbf{DISCOVERED} \\ 4 & \textbf{DISCOVERED} \\ 5 & \textbf{UNDISCOVERED} \end{bmatrix}$$

Filled blue: discovered, **thick red**: in queue, **thick path**: predecessor

An example

Add the neighbors of 3 to the queue

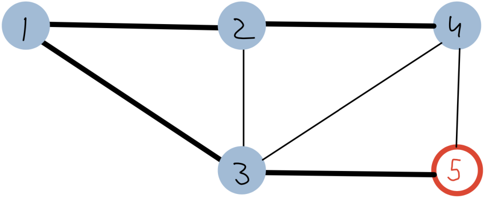


$$Q = [4, 5] \qquad \begin{bmatrix} 1 & \textbf{DISCOVERED} \\ 2 & \textbf{DISCOVERED} \\ 3 & \textbf{DISCOVERED} \\ 4 & \textbf{DISCOVERED} \\ 5 & \textbf{DISCOVERED} \end{bmatrix}$$

Filled blue: discovered, **thick red**: in queue, **thick path**: predecessor

An example

Explore 4

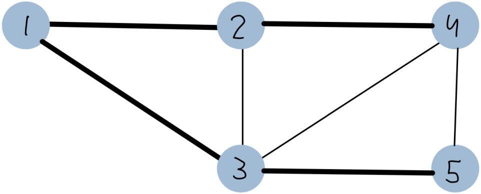


$$Q = [5] \qquad \begin{bmatrix} 1 & \textbf{DISCOVERED} \\ 2 & \textbf{DISCOVERED} \\ 3 & \textbf{DISCOVERED} \\ 4 & \textbf{DISCOVERED} \\ 5 & \textbf{DISCOVERED} \end{bmatrix}$$

Filled blue: discovered, **thick red**: in queue, **thick path**: predecessor

An example

Explore 5



$Q = []$

$$\begin{bmatrix} 1 & \textbf{DISCOVERED} \\ 2 & \textbf{DISCOVERED} \\ 3 & \textbf{DISCOVERED} \\ 4 & \textbf{DISCOVERED} \\ 5 & \textbf{DISCOVERED} \end{bmatrix}$$

Filled blue: discovered, **thick red**: in queue, **thick path**: predecessor

Quality of the algorithm

We argue that this algorithm is "good" by proving:
Functional correctness:

- If **predecessor**[w] $\neq$ **null**, then there is a path from v to w
- If there is a non-empty path from v to w and $v \neq w$, then **predecessor**[w] $\neq$ **null**

Efficiency:

- the algorithm runs in $\mathcal{O}(|V| + |E|)$

Proving correctness

- For this course, both an informal and a formal proof of correctness is accepted.
- Both techniques are presented in the slides

Proving correctness: informally I

We show:

If **predecessor**[w] $\neq$ **null**, then there is a path from v to w

The proof:

- Since **predecessor**[w] $\neq$ **null**, at some point w was discovered by breadth-first search
- As such, it suffices to show that there is a path from v to every discovered vertex u $\neq$ v
- This holds vacuously in the initialization phase
- In the loop: we only set a vertex to discovered if it is adjancent to some vertex that is already discovered

Proving correctness: informally II

We show:

If there is a non-empty path from v to w and $v \neq w$, then **predecessor**[w] $\neq$ **null**

The proof:

- Let p be the path from v to w
- Write p as v, v_1, ..., v_n, w
- For each i, the vertex v_i will be discovered and added to the frontier
- As such, at some point, w will be discovered, and w will be assigned a predecessor

Proving correctness: main technique

Let us start with some observations:

- If a vertex is in the queue, then it has been discovered
- If a vertex is discovered, then it has a predecessor
- If a vertex has a predecessor, then there is a path from v to that vertex

Proving correctness: main technique

Let us start with some observations:

- If a vertex is in the queue, then it has been discovered
- If a vertex is discovered, then it has a predecessor
- If a vertex has a predecessor, then there is a path from v to that vertex

These properties hold

- **before** the while-loop
- **during** the while-loop
- **after** the while-loop

We need techniques to prove these observations.

Proving correctness: loop invariants

To prove the correctness of the code, we use **loop invariants**.

- We choose a property called **invariant**
- We show that the invariant holds before we enter the loop
- We show that if the invariant holds at some iteration, then it also holds after it
- Result: the invariant holds after the loop
- Show that correctness follows from the invariant

More on loop invariants in the course Semantics and Correctness

Functional Correctness I

We show:

If **predecessor**[w] $\neq$ **null**, then there is a path from v to w

For the loop invariant, we pick the following

- for every w, if predecessor[w] $\neq$ **null**, then there is a path from v to w

Functional Correctness I

We show:

If **predecessor**[w] $\neq$ **null**, then there is a path from v to w

For the loop invariant, we pick the following

- for every w, if predecessor[w] $\neq$ **null**, then there is a path from v to w
- if w is in Q and w $\neq$ v, then predecessor[w] $\neq$ **null**

The last one is needed to make the proof work.

Functional Correctness I: Initialization

```
for each u in vertex(G) unequal v
    explored[u] := UNDISCOVERED
    predecessor[u] := null
explored[v] := DISCOVERED
predecessor[v] := null
Q := emptyQueue
enqueue(Q, v)
```

After this phase:

- For no vertex w we have predecessor[w] $\neq$ **null**
- Only v is in Q.

So: the invariant holds before we enter the loop

Functional Correctness I: the loop

```
while (!isEmpty(Q))
    u := dequeue(Q)
    for each w in adjacent(u)
        if (explored[w] == UNDISCOVERED)
            explored[w] := DISCOVERED
            predecessor[w] := u
            enqueue(Q, w)
```

If predecessor[w] $\neq$ **null**, then there is a path from v to w

Case 1: u $=$ v

- If u $=$ v, then we only give a predecessor to vertices adjacent to v

Case 2: u $\neq$ v

- Then predecessor(u) $\neq$ **null**, so there is a path from v to u
- We only give a predecessor to vertices w adjacent to u
- Hence, predecessor[w] $\neq$ **null**, then there is a path from v to w

Functional Correctness I: the loop

```
while (!isEmpty(Q))
    u := dequeue(Q)
    for each w in adjacent(u)
        if (explored[w] == UNDISCOVERED)
            explored[w] := DISCOVERED
            predecessor[w] := u
            enqueue(Q, w)
```

If w is in Q and w $\neq$ v, then predecessor[w] $\neq$ **null**

Note:

- It holds for vertices that are already in the queue.
- If we add a vertex to the queue, then we also set its predecessor

Functional Correctness II

We show:

If there is a non-empty path from v to w and $v \neq w$, then
predecessor[w] $\neq$ **null**

Let p be a non-empty path from v to w.
For the loop invariant, we pick the following

- either w has a predecessor or there is a vertex in p that is in the queue.

Functional Correctness II: why is this fine?

The loop invariant is

> either w has predecessor or there is a vertex in *p* that is in the queue.

After the loop, the queue is empty.
So, we must be in the first case: w has a predecessor

Functional Correctness II: initialization

```
for each u in vertex(G) unequal v
    explored[u] := UNDISCOVERED
    predecessor[u] := null
explored[v] := DISCOVERED
predecessor[v] := null
Q := emptyQueue
enqueue(Q, v)
```

The invariant holds after the initialization, because v is in the queue

Functional Correctness II: the loop

```
while (!isEmpty(Q))
    u := dequeue(Q)
    for each w in adjacent(u)
        if (explored[w] == UNDISCOVERED)
            explored[w] := DISCOVERED
            predecessor[w] := u
            enqueue(Q, w)
```

There are three cases for u

- It is w
- u $\neq$ w and u is in p
- u $\neq$ w and u is not in p

Functional Correctness II: the loop, case 1

```
while (!isEmpty(Q))
    u := dequeue(Q)
    for each w in adjacent(u)
        if (explored[w] == UNDISCOVERED)
            explored[w] := DISCOVERED
            predecessor[w] := u
            enqueue(Q, w)
```

Case 1: u = w

- This means that w was in the queue.
- So, it has a predecessor

Functional Correctness II: the loop, case 2

```
while (!isEmpty(Q))
    u := dequeue(Q)
    for each w in adjacent(u)
        if (explored[w] == UNDISCOVERED)
            explored[w] := DISCOVERED
            predecessor[w] := u
            enqueue(Q, w)
```

Case 2: u $\neq$ w and u is in p

- Since u $\neq$ w, there is a successor w' of w in p
- This w' gets added to the queue

Functional Correctness II: the loop, case 3

```
while (!isEmpty(Q))
    u := dequeue(Q)
    for each w in adjacent(u)
        if (explored[w] == UNDISCOVERED)
            explored[w] := DISCOVERED
            predecessor[w] := u
            enqueue(Q, w)
```

Case 3: u $\neq$ w and u is not in *p*

- The loop invariant holds at the beginning of the loop
- if w has a predecessor, then it still has one
- If there is a vertex in *p* in the queue, then it still is there

The complexity

The algorithm has two phases:

- Initialization: $\mathcal{O}(|V|)$
- The main loop: $\mathcal{O}(|V| + |E|)$

In total: $\mathcal{O}(|V| + |E|)$

Complexity of initialization

```
for each u in vertex(G) unequal v
    explored[u] := UNDISCOVERED
    predecessor[u] := null
```

This runs in $\mathcal{O}(|V|)$.

```
explored[v] := DISCOVERED
predecessor[v] := null
Q := emptyQueue
enqueue(Q, v)
```

This runs in $\mathcal{O}(1)$.

Complexity of the main loop

```
while (!isEmpty(Q))
    u := dequeue(Q)
    for each w in adjacent(u)
        if (explored[w] == UNDISCOVERED)
            explored[w] := DISCOVERED
            predecessor[w] := u
            enqueue(Q, w)
```

Note:

- For each vertex: there is a dequeue
- For each neighbor of that vertex: there is one iteration of the for-loop

In total: $\mathcal{O}(|V| + |E|)$

Additional Properties of Breadth-First Search

Vertices are considered in the following order:

- First we consider the vertices w for which there is a path with 1 edge from v to w
- Then we consider the vertices w for which there is a path with 2 edges from v to w
- and so on

So, the path from v to w given by BFS is the shortest path (with regard to the number of edges).
In addition, we can find all connected components using BFS by running it on every vertex.

Conclusion

Main lessons of today:

- Graphs are useful data structures
- Definition of graphs, basic terminology (vertex, edge, adjacent)
- Breadth-first search

Important tools for analysing code:

- Loop invariants can be used to prove properties about code
- Determine complexity by counting the amount of iterations

Reading material: Chapter 7 and 8.1, 8.2, 8.3 in Roughgarden