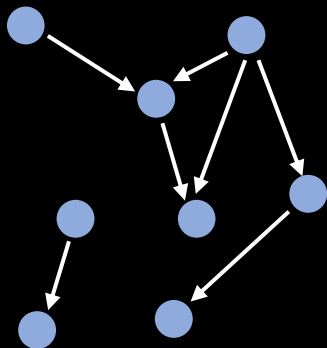
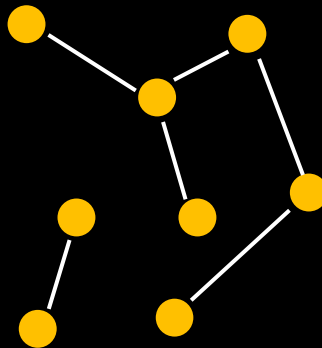


### directed graphs



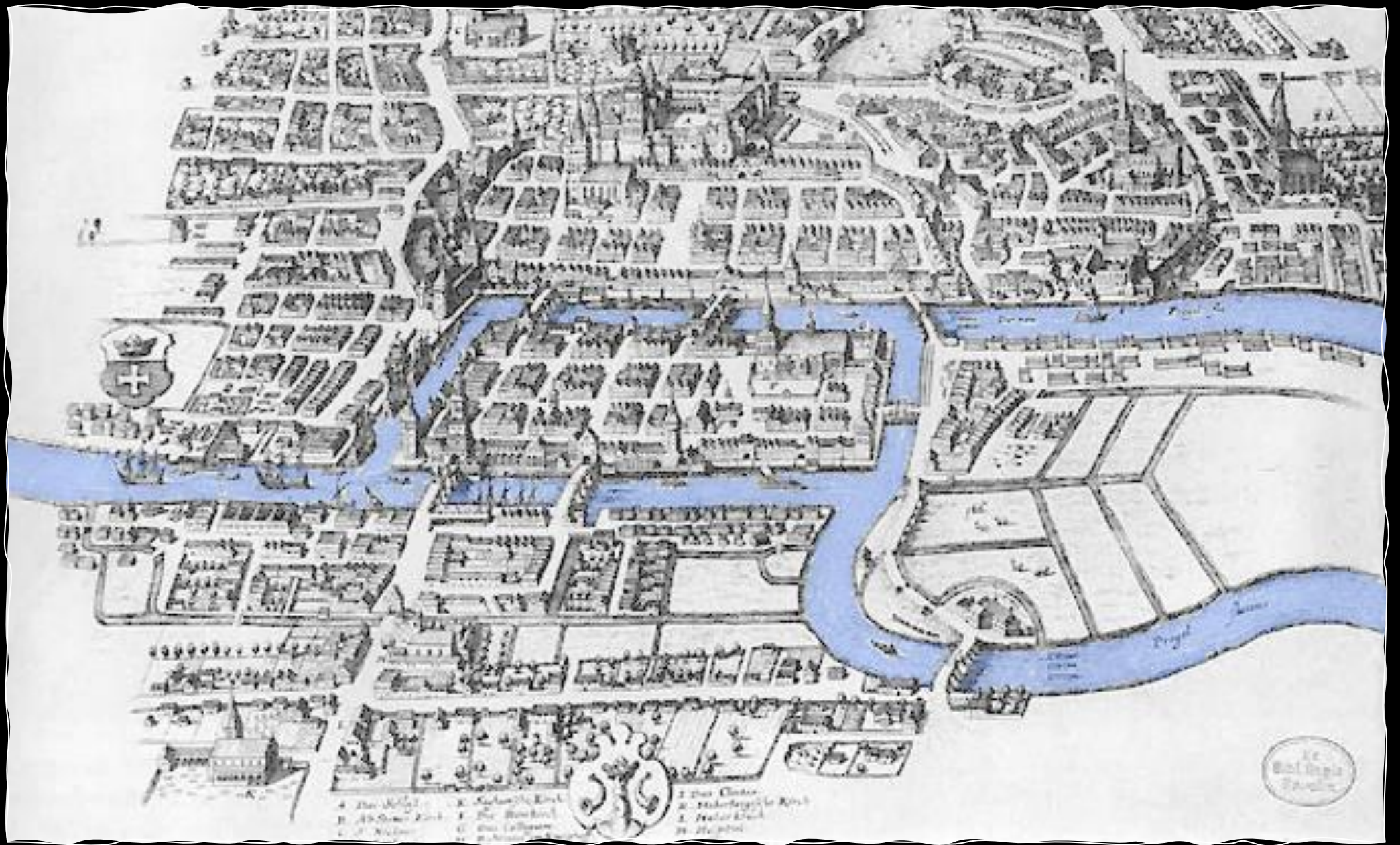
$$E \subseteq V \times V$$

### undirected graphs

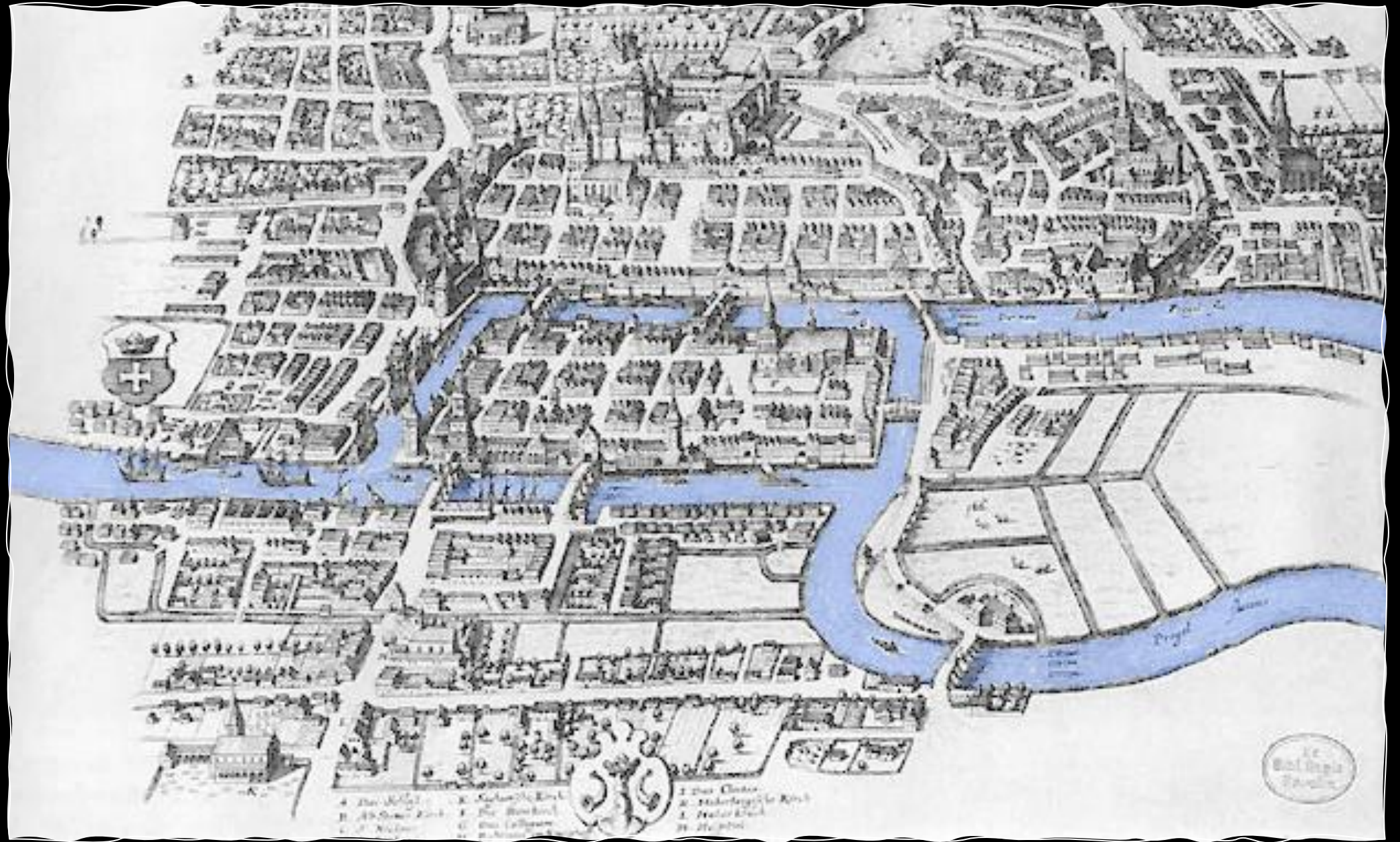


$$E \subseteq \text{subsets of } V \text{ of size } 2$$

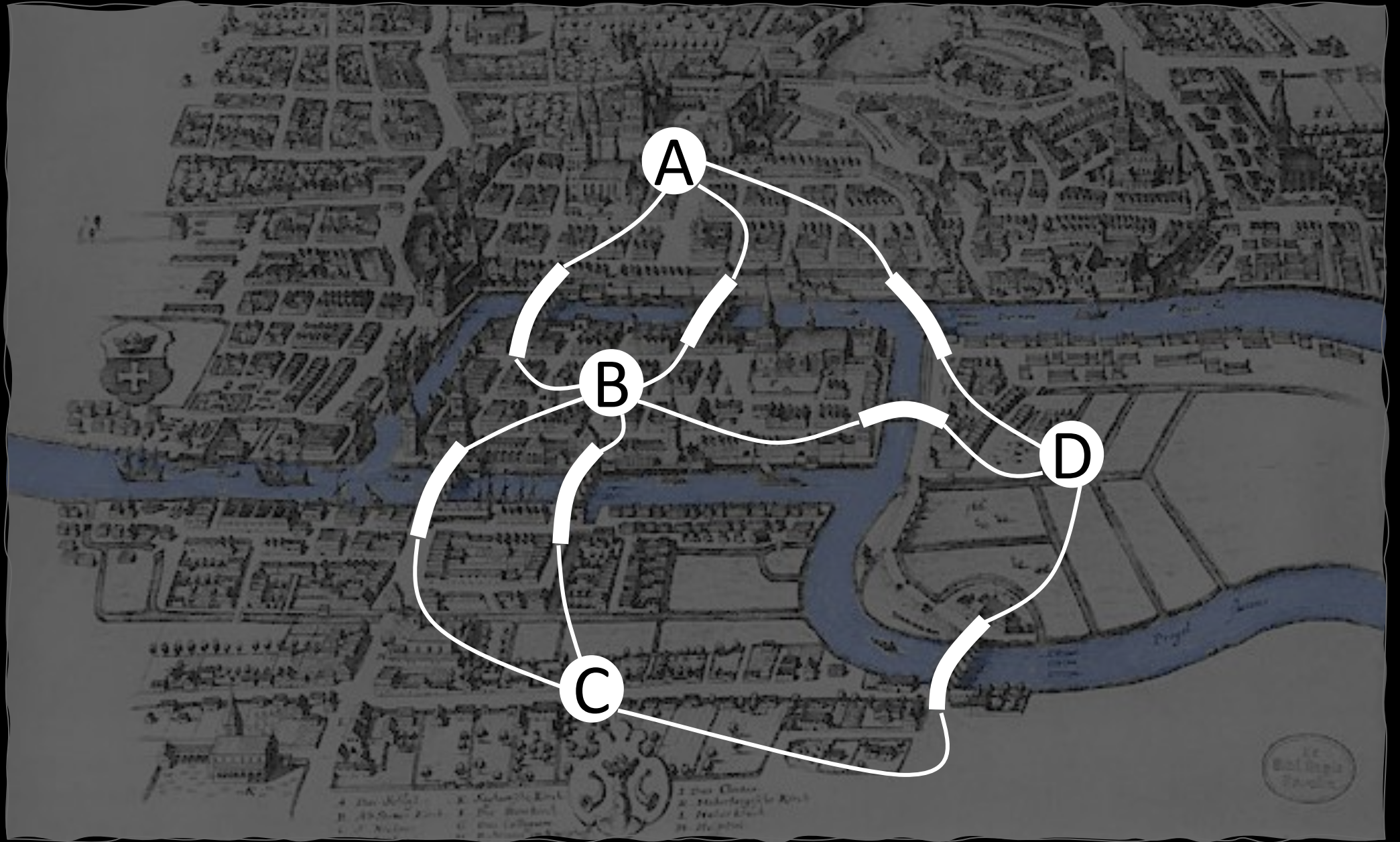
# KONIGSBERGA



“Can I go for a stroll around the city on a route that crosses each bridge exactly once?”

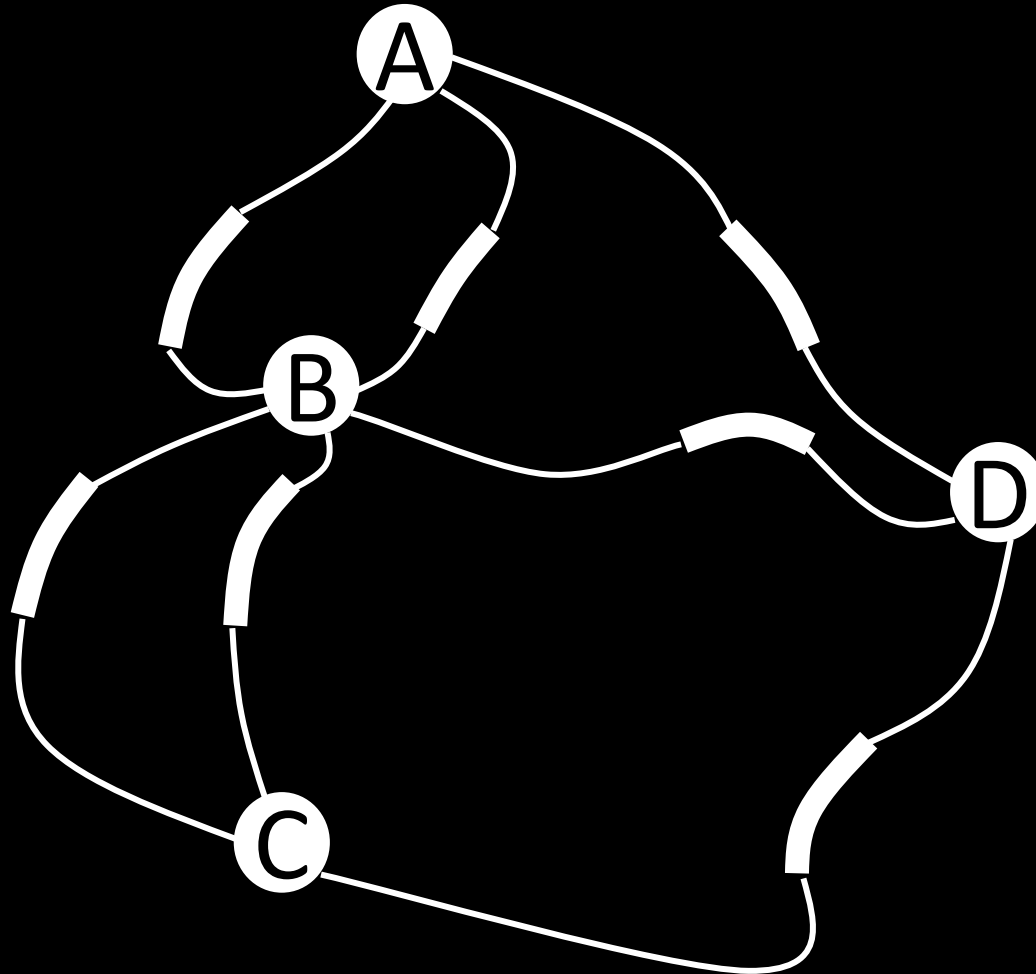


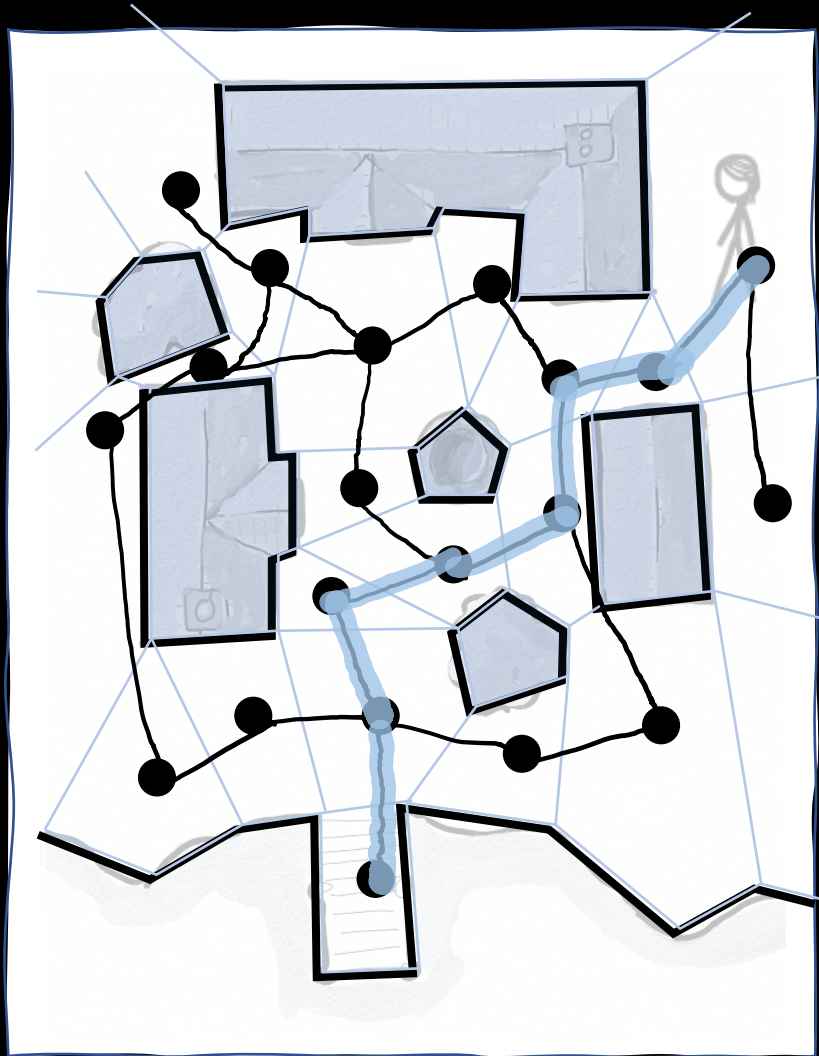
“Can I go for a stroll around the city on a route that crosses each bridge exactly once?”



“Is there a path in which every edge appears exactly once?”

$g = \{A: [B, B, D],$   
 $B: [A, A, C, C, D],$   
 $C: [B, B, D],$   
 $D: [A, B, C]\}$





# PATH-FINDING ALGORITHMS

How should this game agent navigate to the jetty?

1. Draw polygon boundaries around obstacles
2. Divide free space into convex polygons
3. Create a graph, with edges between adjacent polygons
4. Find a path on the graph
5. Draw this path in 2D coordinates on the map (easy, since we've used convex polygons)

# Dwarf Fortress



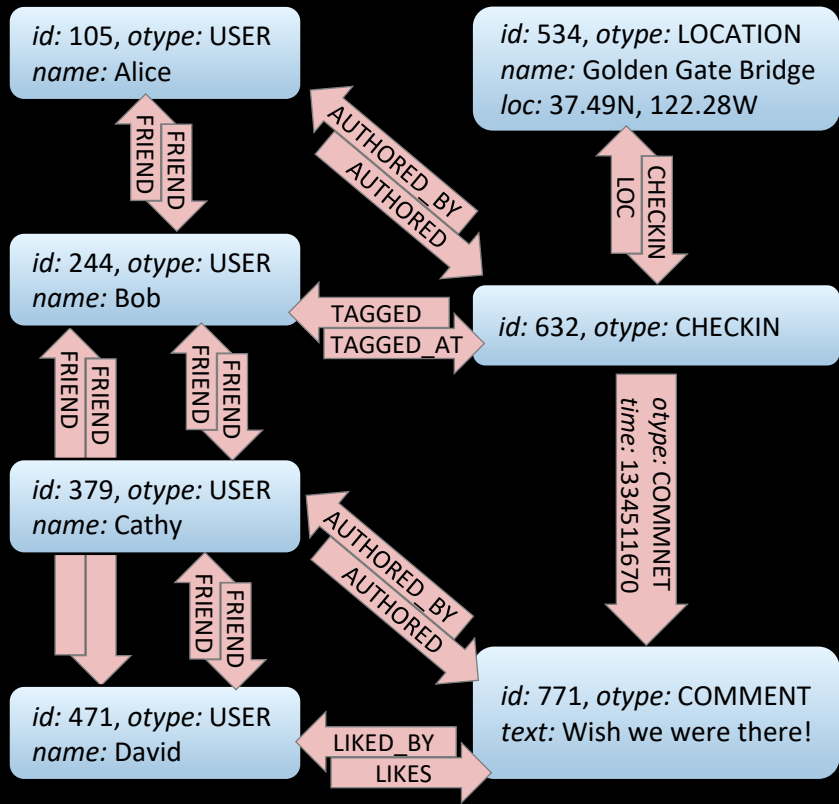
Q: I've seen other games similar to Dwarf Fortress die on their pathfinding algorithms. What do you use and how do you keep it efficient?

A: Yeah, the base algorithm is only part of it. We use A\*, which is fast of course, but it's not good enough by itself.

Generally, people have used approaches that add various larger structures on top of the map to cut corners. But we can't take advantage of these innovations since our map changes so much.

*Interview with Tarn Adams (developer) by Ryan Donovan from the StackOverflow blog, Dec 2021*

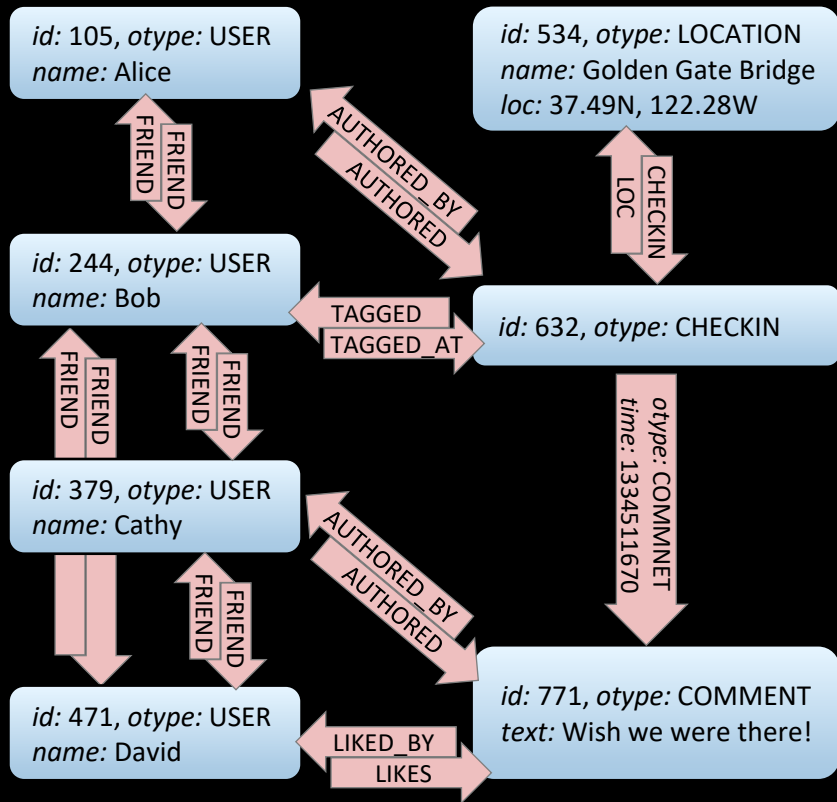
Alice was at the Golden Gate Bridge with Bob  
 Cathy: Wish we were there! David likes this



Q. Why did Facebook choose to make CHECKIN a vertex, rather than a USER→LOCATION edge?



Alice was at the Golden Gate Bridge with Bob  
 Cathy: Wish we were there! David likes this



Q. What algorithmic questions we might ask about this graph?

# What this course is about

- Clever algorithms
- Performance analysis
- What we can model with graphs
- Proving correctness



Right from the beginning, and all through the course, we stress that the programmer's task is not just to write down a program, but that his main task is to give a formal proof that the program he proposes meets the equally formal functional specification.

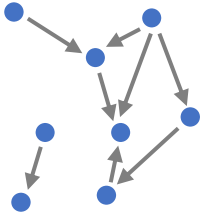
Edsger Dijkstra (1930—2002)

*On the cruelty of really teaching computer science, 1988*

# Graph notation

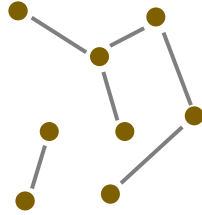
A graph consists of a set of vertices  $V$ , and a set of edges  $E$ .

## directed graphs

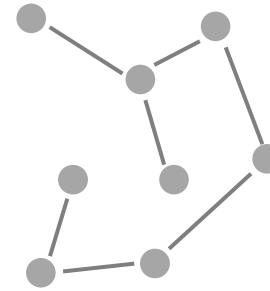
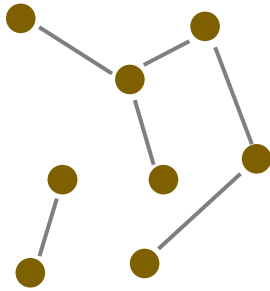
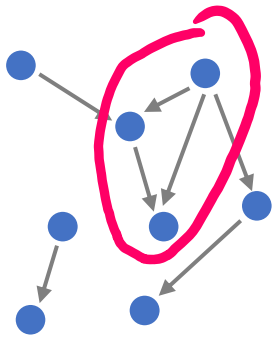


$v_1 \rightarrow v_2$  is how we write  
the edge from  $v_1$  to  $v_2$

## undirected graphs



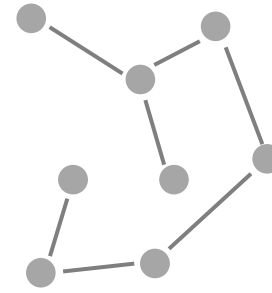
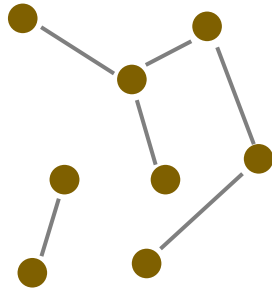
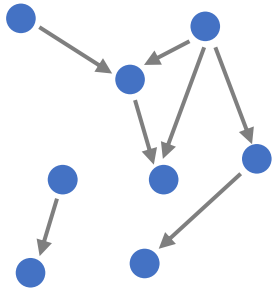
$v_1 \leftrightarrow v_2$  is how we write  
the edge between  $v_1$  and  $v_2$



Which of these two graphs is a tree, which a forest?

- A *directed acyclic graph* (DAG) is a directed graph without any cycles

- A *forest* is an undirected acyclic graph
- A *tree* is a connected forest
- (An undirected graph is *connected* if for every pair of vertices there is an edge between them)

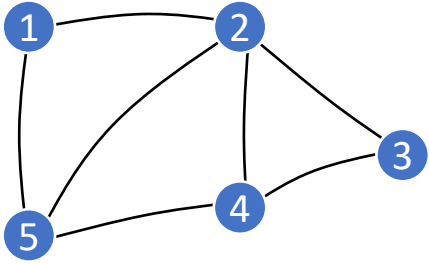


What's wrong with my definitions for *path* and *cycle*?

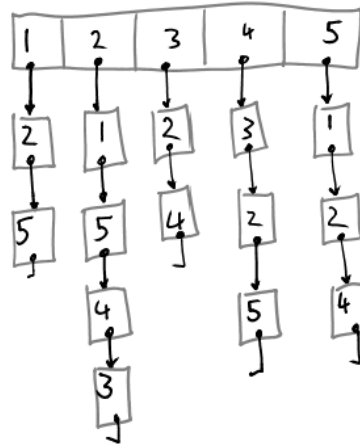
- A *directed acyclic graph* (DAG) is a directed graph without any cycles

- A *forest* is an undirected acyclic graph
- A *tree* is a connected forest
- (An undirected graph is *connected* if for every pair of vertices there is an edge between them)

# How we can store graphs, in computer code



## Array of adjacency lists



```

{1: [2,5],
 2: [1,5,4,3],
 3: [2,4],
 4: [3,2,5],
 5: [1,2,4]
}
  
```

Storage:

$$O(|V| + |E|)$$

## Adjacency matrix

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

```

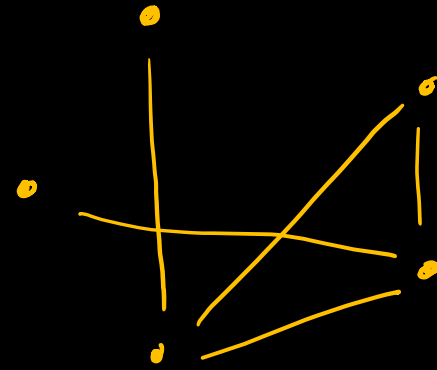
np.array([[0,1,0,0,1],
          [1,0,1,1,1],
          [0,1,0,1,0],
          [0,1,1,0,1],
          [1,1,0,1,0]])
  
```

Storage:

$$O(|V|^2)$$

## Mini-exercise

- What is the largest possible number of edges in an undirected graph with  $V$  vertices?
- and in a directed graph?
- What's the smallest possible number of edges in a tree with  $V$  vertices?





- Computer Laboratory
- Teaching
- Courses 2022–23
- Part IA CST
- Algorithms 2**
- Preparation for Computer Science
- Databases
- Digital Electronics
- Discrete Mathematics
- Foundations of Computer Science
- Hardware Practical Classes
- Introduction to Graphics
- OCaml Practical Classes
- Object-Oriented Programming
- Registration

## Course pages 2022–23

### Algorithms 2

- Syllabus
- Course materials**
- Ticks
- Recordings
- For supervisors

This course is a continuation of [Algorithms 1](#) (which is why these notes start at Section 5, and why the lectures start at Lecture 13).

#### Lecture notes

- Full notes as printed
- If you spot a mistake in the printed notes, let me know. Corrections will appear here.

Announcements, Q&A, tick submission – [Moodle](#)

#### Schedule

This is the planned lecture schedule. It will be updated as and when actual lectures deviate from schedule. Links are to prerecorded videos. Slides will be uploaded the night before a lecture, and re-uploaded after the lecture with annotations made during the lecture.

#### 5. Graphs and path finding

- Lecture 13 [5, 5.1 Graphs](#) (14:27) [code](#) — [graphs](#)  
[5.2 Depth-first search](#) (11:37)  
[5.3 Breadth-first search](#) (6:43)  
Optional tick: [bfs-all](#) from ex4.q6
- Lecture 14 [5.4 Dijkstra's algorithm](#) (15:25) plus [proof](#) (24:01)
- Lecture 15 [5.5 Algorithms and proofs](#) (9:29)  
[5.6 Bellman-Ford](#) (12:13)  
Optional challenge: [chatgpt-bfs](#)  
Optional tick: [bf-cycle](#) from ex4.q19

❖ lecture notes

❖ example sheets

❖ slides

❖ ticks

❖ recordings

# How to learn effectively

## PASSIVE LEARNING

- attend lectures
- read code snippets, watch animations, see examples
- read notes, watch videos

## ACTIVE LEARNING

- copy out the lecturer's hand-writing
- annotate printed code snippets and examples (using page numbers)

## REFLECTIVE LEARNING

- mini-exercises and example sheets
- optional ticks
- skeptical reading

# Pre-recorded videos

# Consent to recordings of live lectures

<https://www.educationalpolicy.admin.cam.ac.uk/supporting-students/policy-recordings/recordings-student-information-sheet>

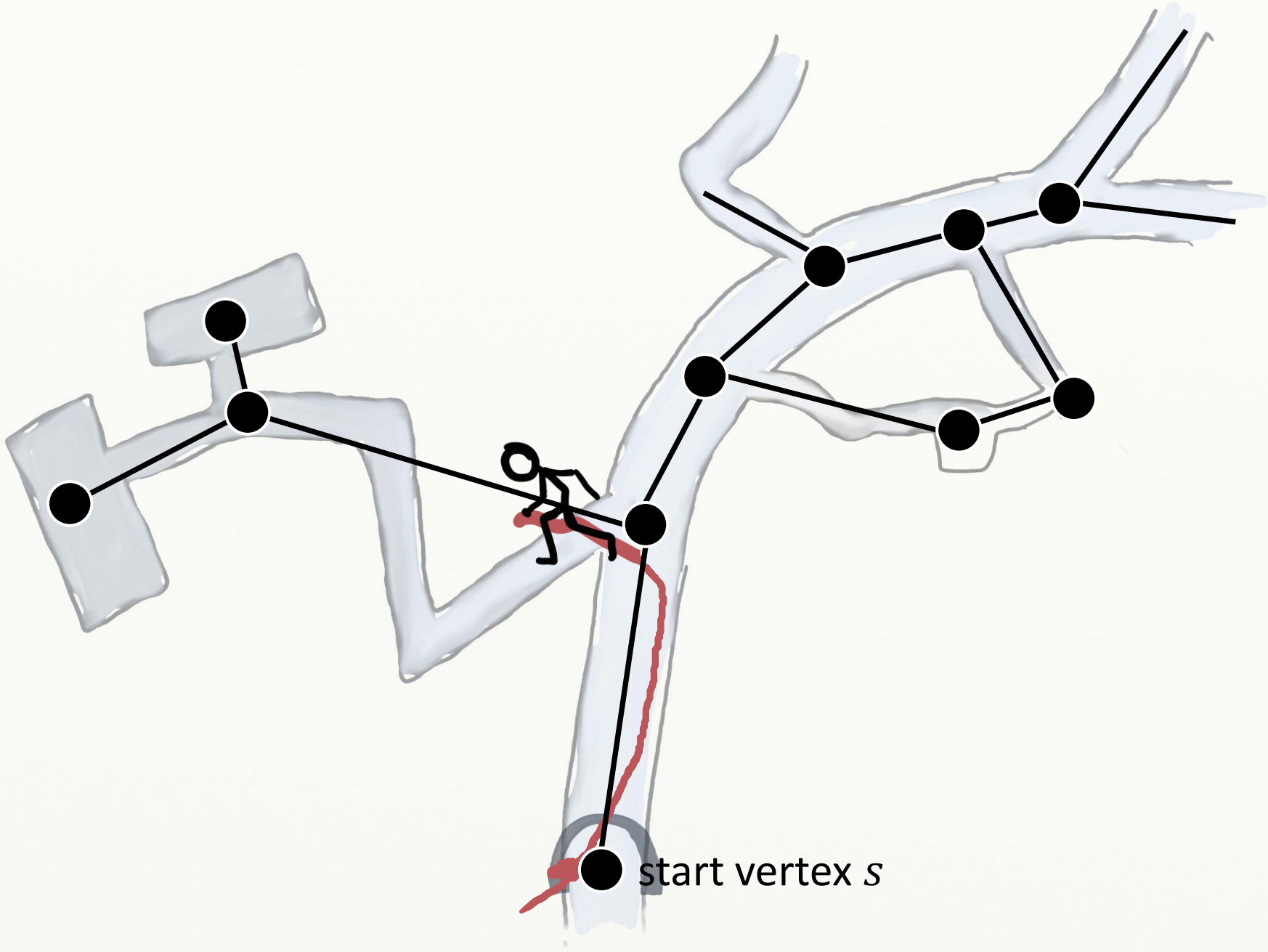
For any teaching session where your contribution is mandatory or expected, we must seek your consent to be recorded.

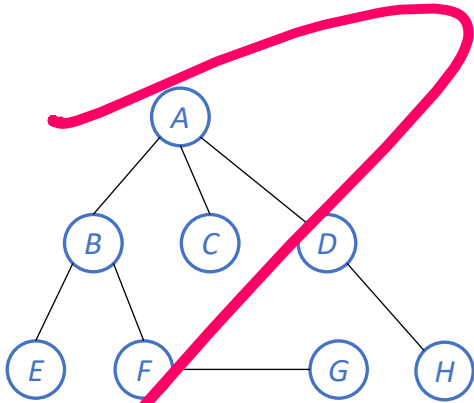
**You are not obliged to give this consent, and you have the right to withdraw your consent after it has been given.**

The screenshot shows a YouTube video player for a video titled "Depth-first search" from the "IA Algorithms" playlist. The video player shows the word "ALGORITHMS" in a stylized font on a grid background. Below the player, the video title "Depth-first search" and the channel name "Foundations of Data Science" are visible. The video has 570 views and was uploaded 1 year ago. The description mentions it is from a lecture course at Cambridge University, taught by Damon Wischik, and provides links to lecture notes and code. To the right of the video player, there is a playlist titled "IA Algorithms" with four items: "Graphs" (14:27), "Depth-first search" (11:37), "Breadth-first search and shortest paths..." (6:43), and "Dijkstra's algorithm" (15:25). Below the playlist, there are several recommended videos, including "Breadth-first search and shortest paths o...", "7.2-3 Bayesian model-crafting and...", "Let's build GPT: from scratch, in code,..." by Andrej Karpathy, and "Lecture 14: Depth-First Search (DFS)..." by MIT OpenCourseWare.

SECTION 5.2

# Depth-first search





```

1 def visit(v):
2     print("visiting", v)
3     for w in v.neighbours:
4         visit(w)
5
6 visit(A)

```

```

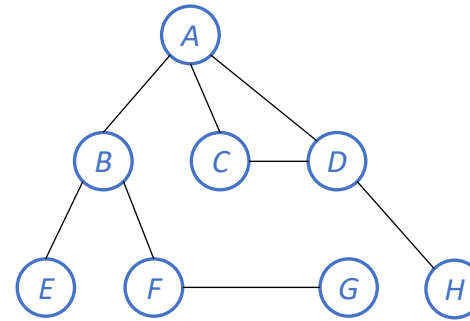
visiting A
visiting B
visiting A
visiting B
...

```

```

RecursionError:
maximum recursion depth exceeded

```



```

1 def visit_tree(v, v_parent):
2     print("visiting", v, "from", v_parent)
3     for w in v.neighbours:
4         if w != v_parent:
5             visit_tree(w, v)
6
7
8 visit_tree(D, None)

```

```

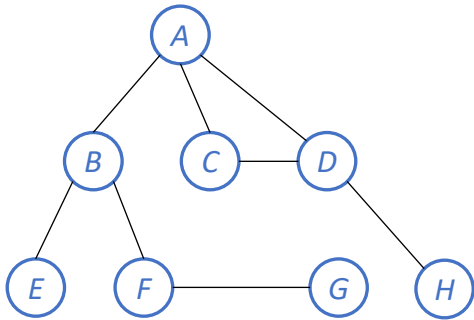
visiting D from None
visiting C from D
visiting A from C
visiting D from A
...

```

```

RecursionError:
maximum recursion depth exceeded

```



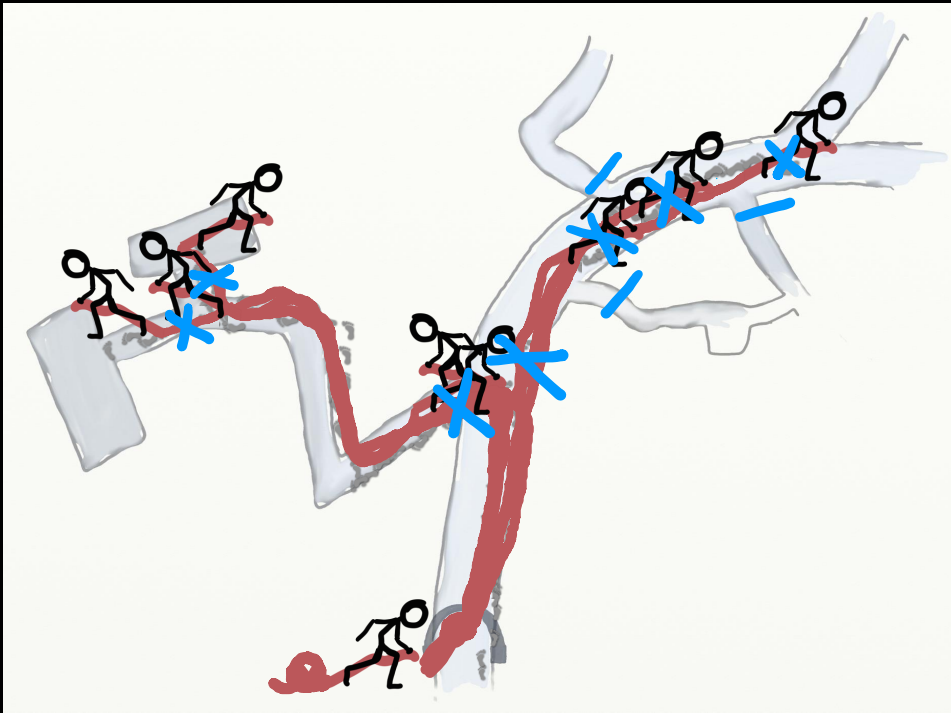
```

1 # visit all vertices reachable from s
2 def dfs_recurse(g, s):
3     for v in g.vertices:
4         v.visited = False
5     visit(s)
6
7 def visit(v):
8     v.visited = True
9     for w in v.neighbours:
10        if not w.visited:
11            visit(w)
  
```

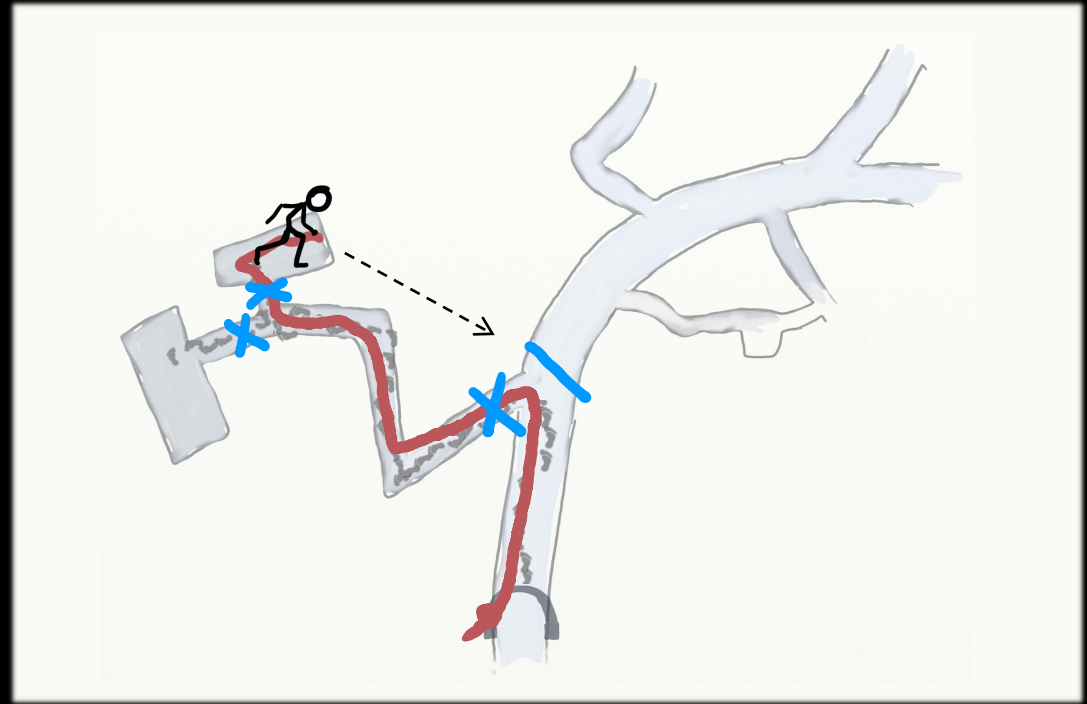
```

dfs_recurse(g, D):
  visit(D):
    neighbours = [H, C, A]
  visit(H):
    neighbours = [D]
    don't visit D
    return from visit(H)
  visit(C)
    neighbours = [D, A]
    don't visit D
  visit(A):
    | ...
  
```

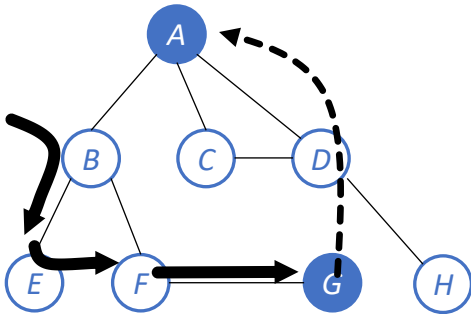
Ariadne's thread ...



but why not just teleport?



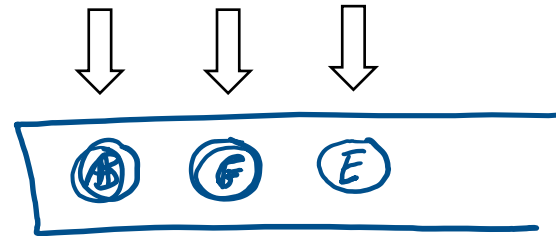




```

1 # visit all vertices reachable from s
2 def dfs(g, s):
3     for v in g.vertices:
4         v.seen = False
5     toexplore = Stack([s])
6     s.seen = True
7
8     while not toexplore.is_empty():
9         v = toexplore.popright()
10        for w in v.neighbours:
11            if not w.seen:
12                toexplore.pushright(w)
13                w.seen = True

```



## Analysis of running time for stack-based dfs

1 *# visit all vertices reachable from s*

2 `def dfs(g, s):`

3     `for v in g.vertices:`

4         `v.seen = False`

5     `toexplore = Stack([s])`

6     `s.seen = True`

7  
8     `while not toexplore.is_empty():`

9         `v = toexplore.popright()`

10         `for w in v.neighbours:`

11             `if not w.seen:`

12                 `toexplore.pushright(w)`

13                 `w.seen = True`

} —  $O(V)$

} —  $O(1)$

} — at most once per vertex, so  $O(V)$

} — run for every edge we visit, so  $O(E)$

total  $O(V+E)$

## Analysis of running time for recursive dfs

```

1 # visit all vertices reachable from s
2 def dfs_recurse(g, s):
3     for v in g.vertices:
4         v.visited = False
5     visit(s)

```

}  $O(V)$

```

7 def visit(v):
8     v.visited = True
9     for w in v.neighbours:
10         if not w.visited:
11             visit(w)

```

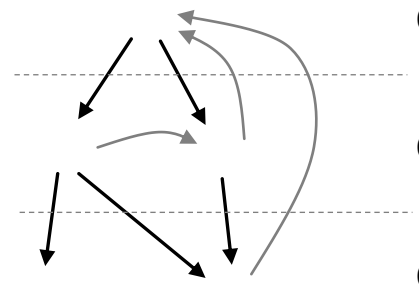
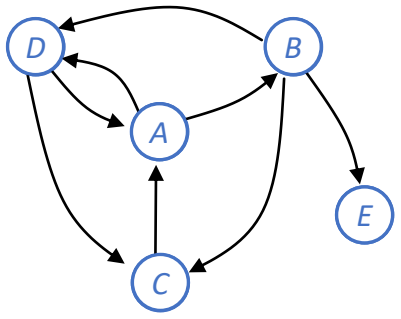
— run at most once per vertex, so  $O(V)$

}  $O(E)$

Total:  $O(V+E)$

SECTION 5.2

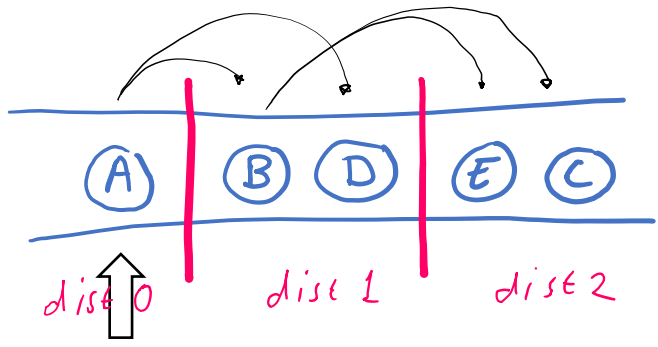
# Breadth-first search / finding shortest path



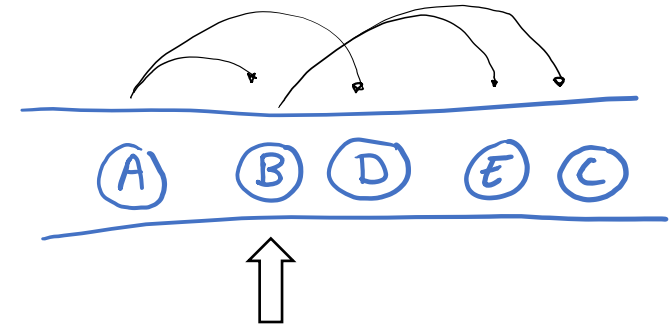
distance from A = 0

distance from A = 1

distance from A = 2



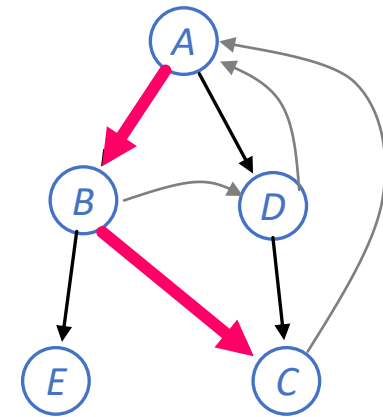
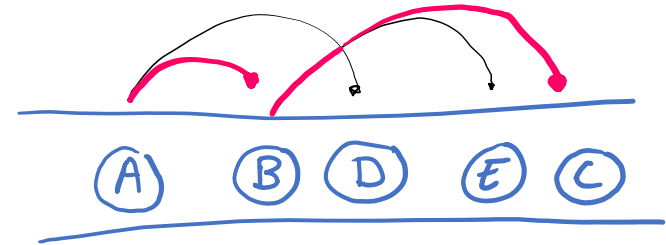
```
1 # Visit all the vertices in g reachable from start vertex s
2 def bfs(g, s):
3     for v in g.vertices:
4         v.seen = False
5     toexplore = Queue([s])
6     s.seen = True
7
8     while not toexplore.is_empty():
9         v = toexplore.popleft()
10        for w in v.neighbours:
11            if not w.seen:
12                toexplore.pushright(w)
13                w.seen = True
```



```

1 # Find a path from s to t, if one exists
2 def bfs_path(g, s, t):
3     for v in g.vertices:
4         (v.seen, v.come_from) = (False, None)
5     ...
10    while not toexplore.is_empty():
11        v = toexplore.popleft()
12        for w in v.neighbours:
13            if not w.seen:
14                toexplore.pushright(w)
15                (w.seen, w.come_from) = (True, v)
16        ...
19    if t.come_from has not been set:
20        there is no path from s to t
21    else:
22        reconstruct the path from s to t,
23        working backwards

```



## Analysis of running time for stack-based dfs

```
1 # visit all vertices reachable from s
2 def dfs(g, s):
3     for v in g.vertices:
4         v.seen = False
5     to_explore = Stack([s])
6     s.seen = True
7
8     while not to_explore.is_empty():
9         v = to_explore.popright()
10        for w in v.neighbours:
11            if not w.seen:
12                to_explore.pushright(w)
13                w.seen = True
```

}  $O(V)$

}  $O(1)$

} at most once per vertex, so  $O(V)$

} run for every edge  $wv$  of every vertex  $w$  we visit, so  $O(E)$

total  $O(V+E)$

## Analysis of running time for bfs

```
1 # Visit all the vertices in g reachable from start vertex s
2 def bfs(g, s):
3     for v in g.vertices:
4         v.seen = False
5     to_explore = Queue([s])
6     s.seen = True
7
8     while not to_explore.is_empty():
9         v = to_explore.popleft()
10        for w in v.neighbours:
11            if not w.seen:
12                to_explore.pushright(w)
13                w.seen = True
```

$O(V+E)$

same as for dfs



## Schedule

This is the planned lecture schedule. It will be updated as and when actual lectures deviate from schedule. Links are to prerecorded videos. Slides will be uploaded the night before a lecture, and re-uploaded after the lecture with annotations made during the lecture.

### 5. Graphs and path finding

---

Lecture 13 [5, 5.1 Graphs](#) (14:27) code — [graphs](#)

[5.2 Depth-first search](#) (11:37)

[5.3 Breadth-first search](#) (6:43)

Optional tick: [bfs-all](#) from ex4.q6

Lecture 14 [5.4 Dijkstra's algorithm](#) (15:25) plus [proof](#) (24:01)

Lecture 15 [5.5 Algorithms and proofs](#) (9:29)

[5.6 Bellman-Ford](#) (12:13)

Optional challenge: [chatgpt-bfs](#)

Optional tick: [bf-cycle](#) from ex4.q19

Lecture 16 [5.7 Dynamic programming](#) (13:06)

[5.8 Johnson's algorithm](#) (13:43)

Example sheet 4 [\[pdf\]](#)

### 6. Graphs and subgraphs

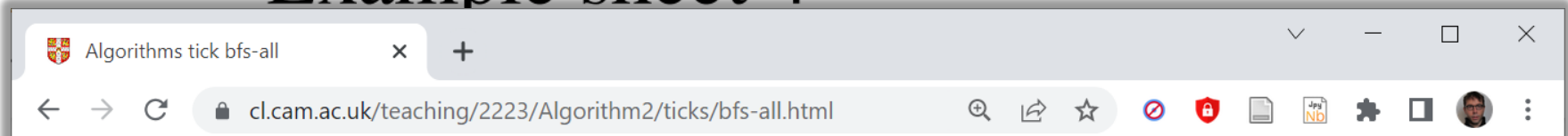
---

Lecture 17 [6.1 Flow networks](#) (9:31) code — [subgraphs](#)

[6.2 Ford-Fulkerson algorithm](#) (21:55)

# Example sheet 4

**Question 6.** Modify `bfs_path` on the website, for you to check



## Algorithms tick: bfs-all

# Find All Shortest Paths

Breadth-first search can be used to find a shortest path between a pair of vertices. Modify the standard `bfs_path` algorithm so that it returns *all* shortest paths.

**Please submit a source file `bfs_all.py` on [Moodle](#).** It should implement a function

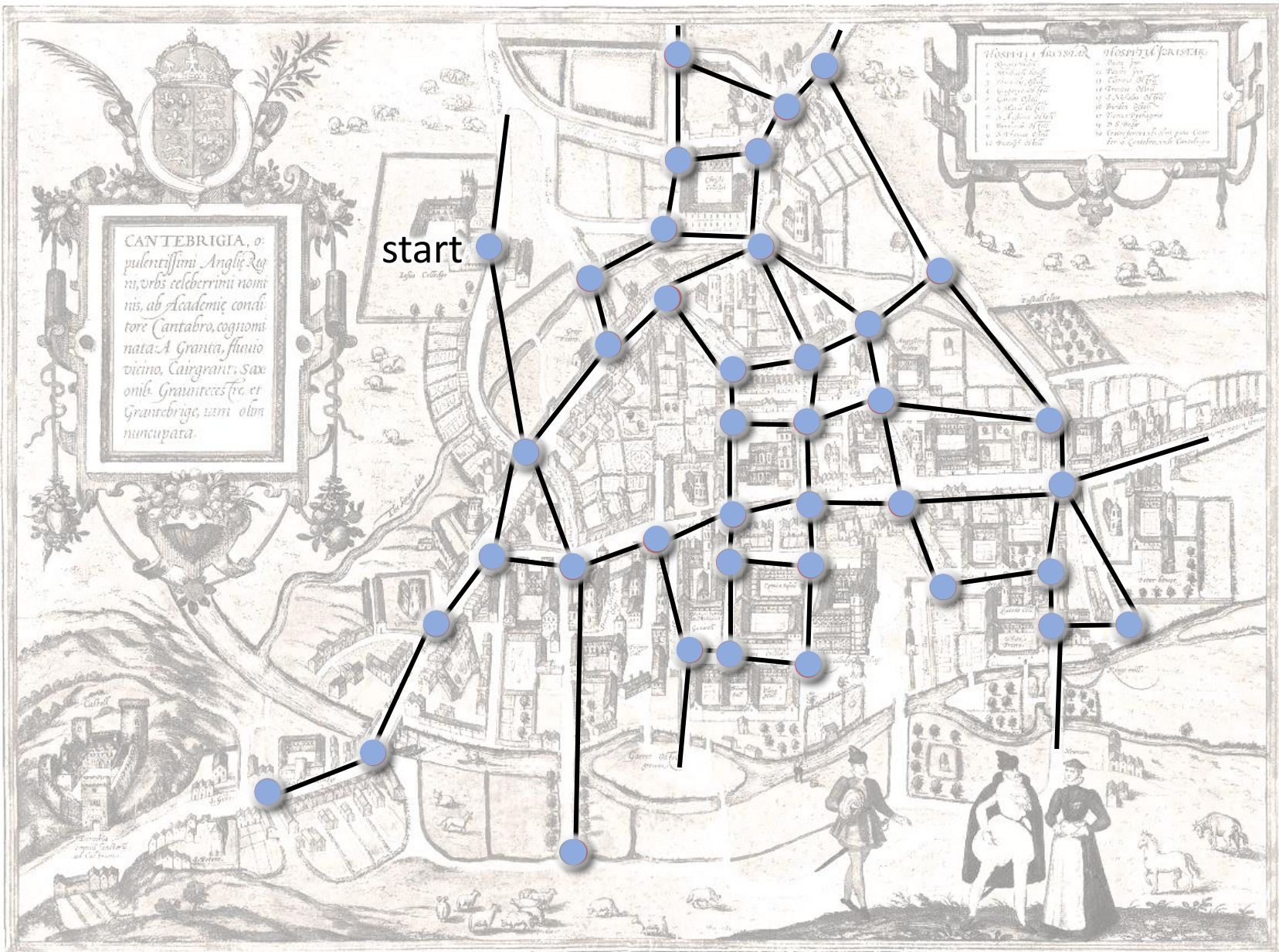
```
shortest_paths(g, s, t)

# Find all shortest paths from s to t
# Return a list of paths, each path a list of vertices starting with s and
```

The graph `g` is stored as an adjacency dictionary, for example `g = {0:{1,2}, 1:{}, 2:{1,0}}`. It has a key for every vertex, and the corresponding value is the set of that vertex's neighbours.

**EXERCISE:** Read the notes / watch the video for section 5.3, to familiarize yourself with Dijkstra's algorithm.

We will spend Monday's lecture going through the proof of correctness.



CANTEBRIGIA, o-  
pulentissimi Angliæ Reg-  
ni, urbs celeberrimi nomi-  
nis, ab Academiæ condit-  
ore Cantabro, cognomi-  
nata. A Grantæ, fluvio  
vicino, Cairgranti, Sax-  
onib. Grantæces fræ, et  
Grantæbrige, iam olim  
nuncupata.

HOSPITIA BRASSERIE	HOSPITIA SERRARIE
1. Sancti Petri	1. Sancti Petri
2. Sancti Pauli	2. Sancti Pauli
3. Sancti Martini	3. Sancti Martini
4. Sancti Michaelis	4. Sancti Michaelis
5. Sancti Johannis	5. Sancti Johannis
6. Sancti Andree	6. Sancti Andree
7. Sancti Thomæ	7. Sancti Thomæ
8. Sancti Augustini	8. Sancti Augustini
9. Sancti Eboracensis	9. Sancti Eboracensis
10. Sancti Nicolai	10. Sancti Nicolai
11. Sancti Gervasii	11. Sancti Gervasii
12. Sancti Juliani	12. Sancti Juliani
13. Sancti Petri ad Vinetum	13. Sancti Petri ad Vinetum
14. Sancti Petri ad Martini	14. Sancti Petri ad Martini
15. Sancti Petri ad Sanctum Paulum	15. Sancti Petri ad Sanctum Paulum
16. Sancti Petri ad Sanctum Martinum	16. Sancti Petri ad Sanctum Martinum
17. Sancti Petri ad Sanctum Augustinum	17. Sancti Petri ad Sanctum Augustinum
18. Sancti Petri ad Sanctum Thomam	18. Sancti Petri ad Sanctum Thomam
19. Sancti Petri ad Sanctum Andream	19. Sancti Petri ad Sanctum Andream
20. Sancti Petri ad Sanctum Johannem	20. Sancti Petri ad Sanctum Johannem

- not yet seen
- distance
- visited