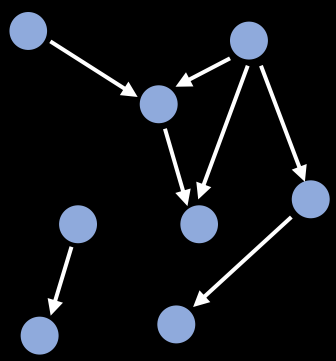
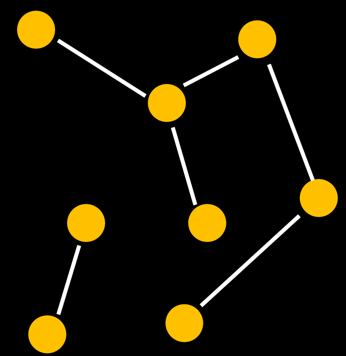
directed graphs



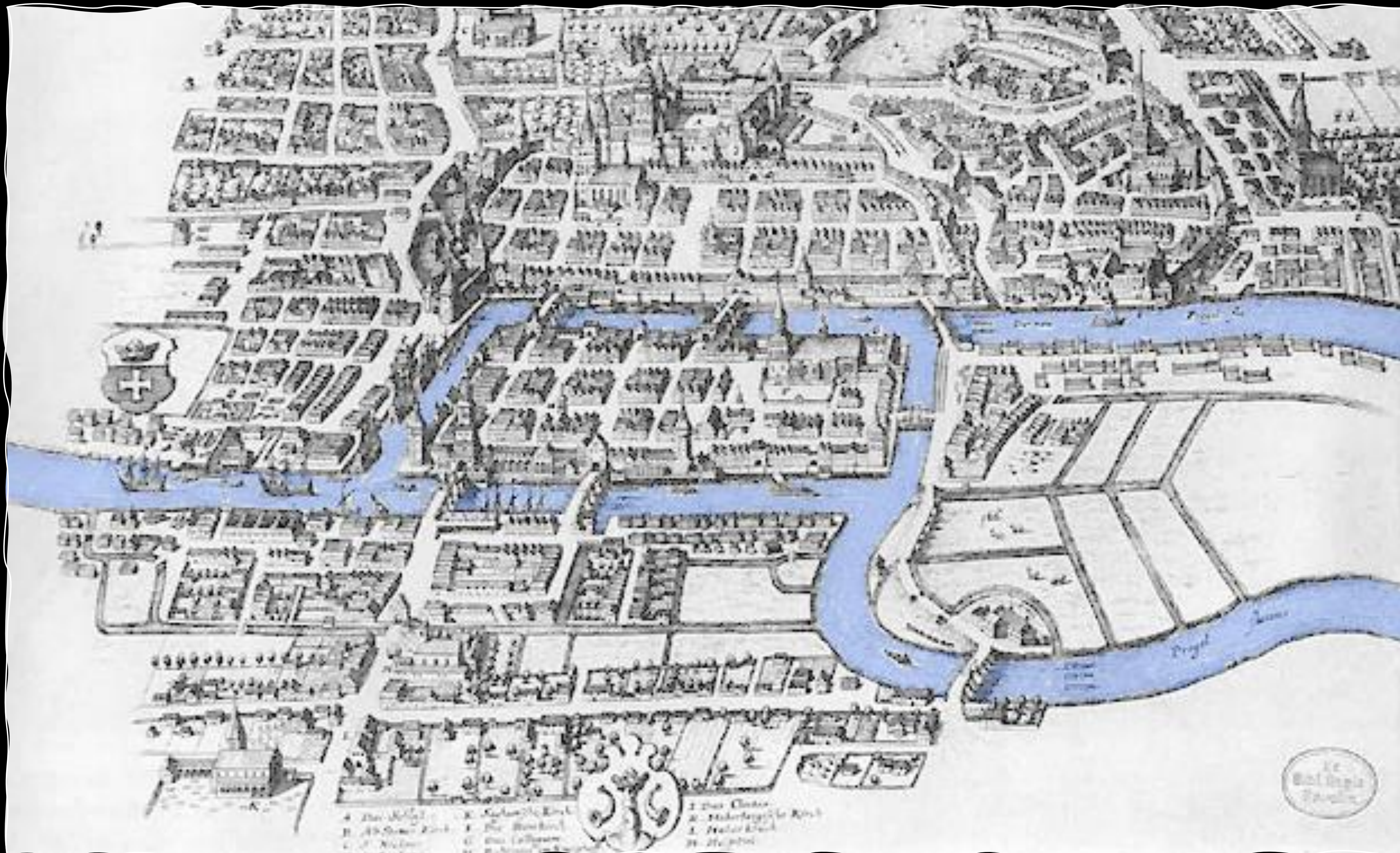
$E \subseteq V \times V$

undirected graphs

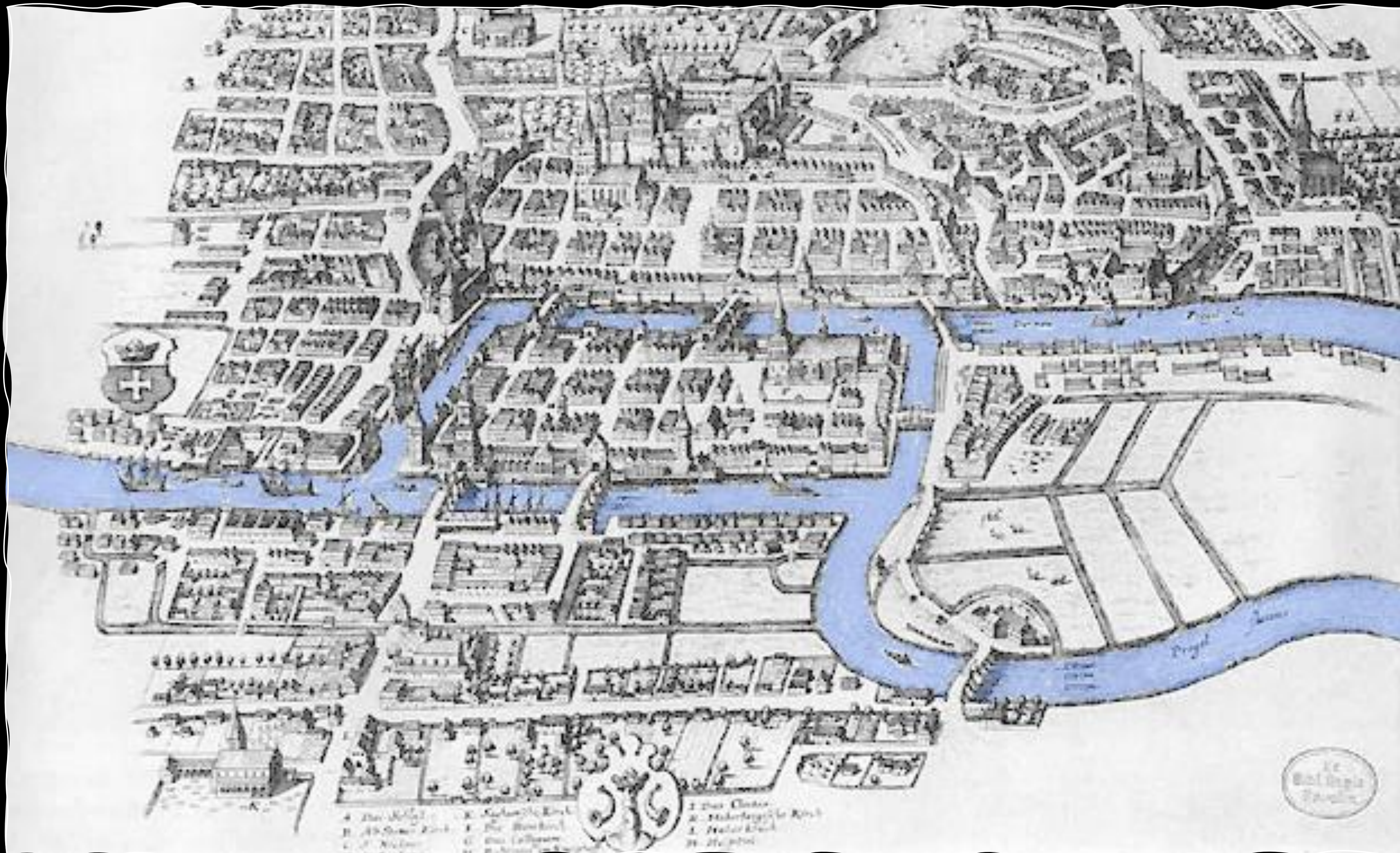


$E \subseteq$ subsets of V of size 2

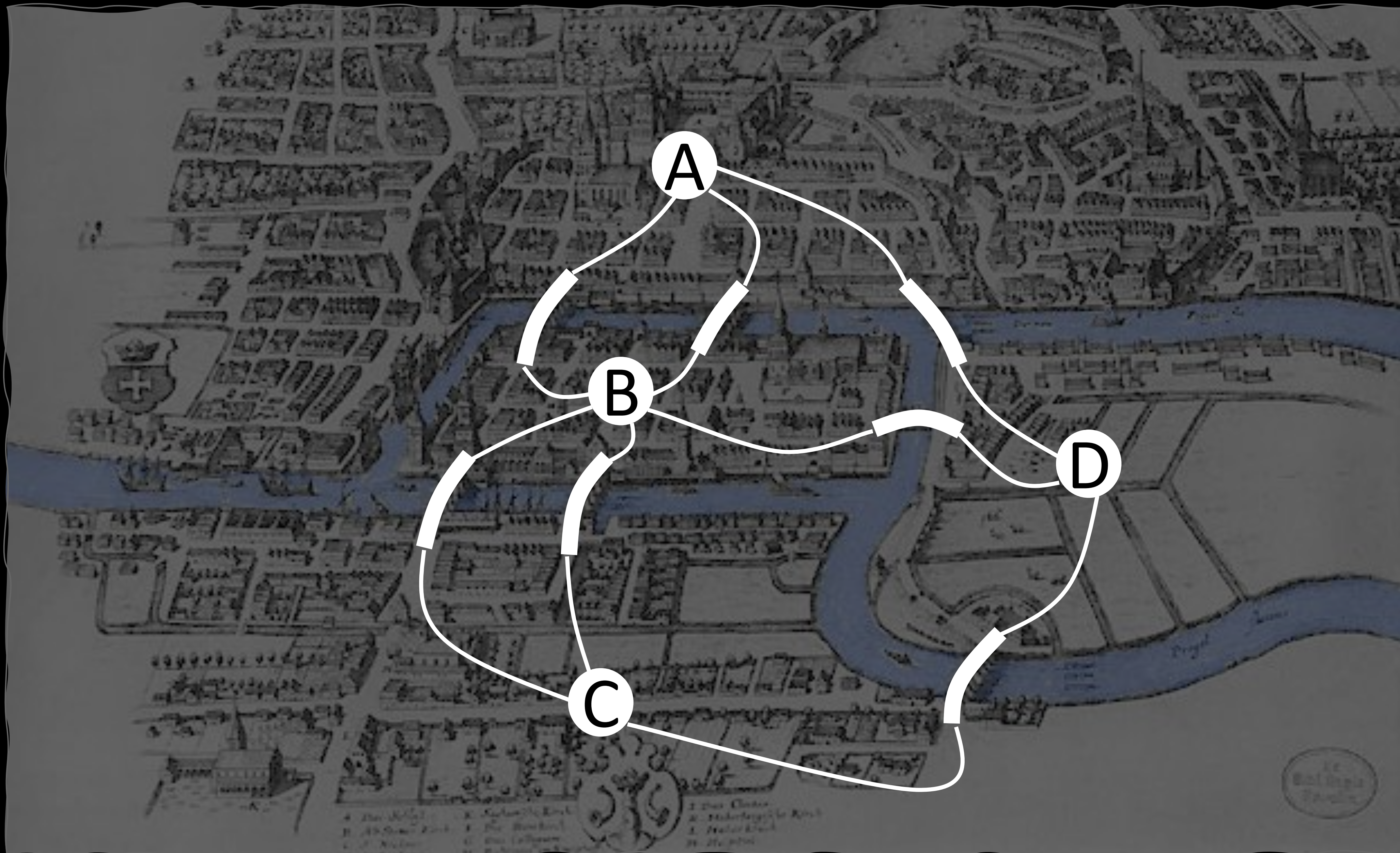
Konigsberga



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

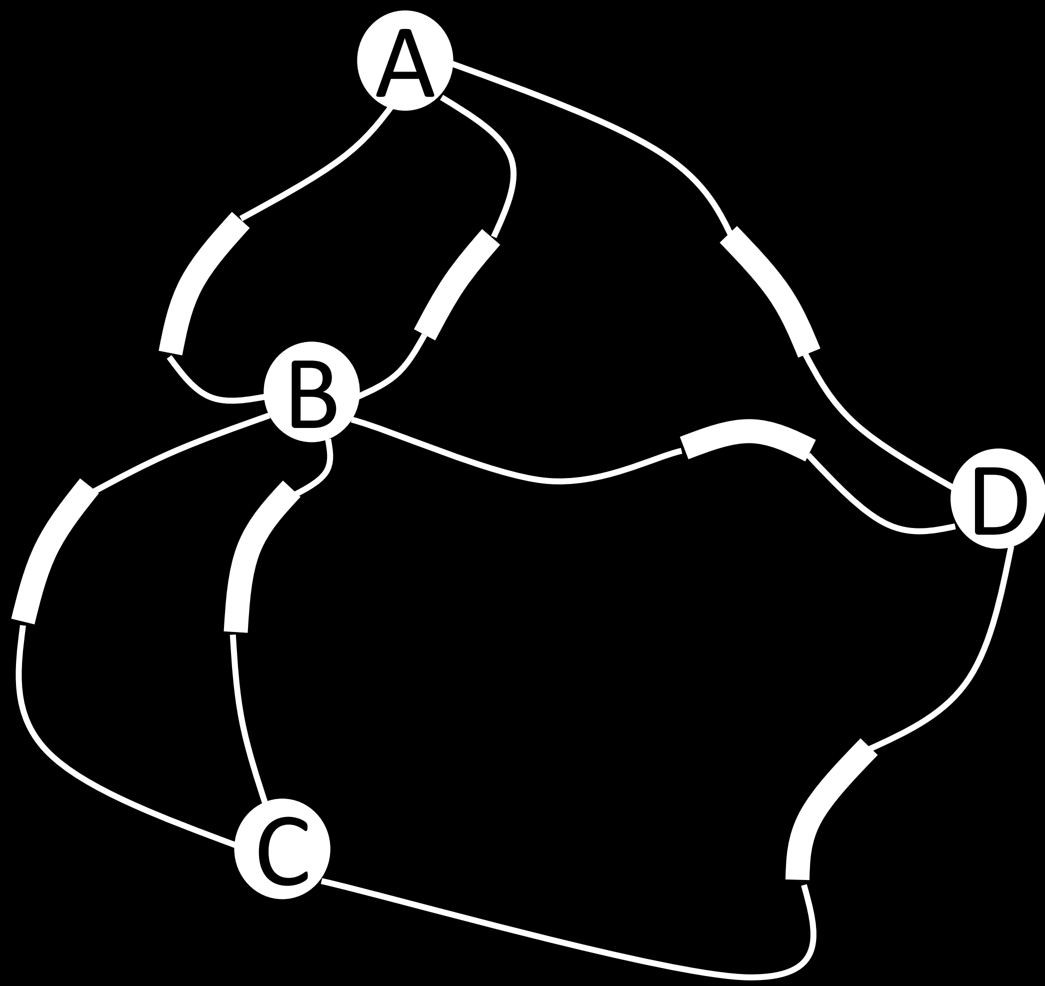


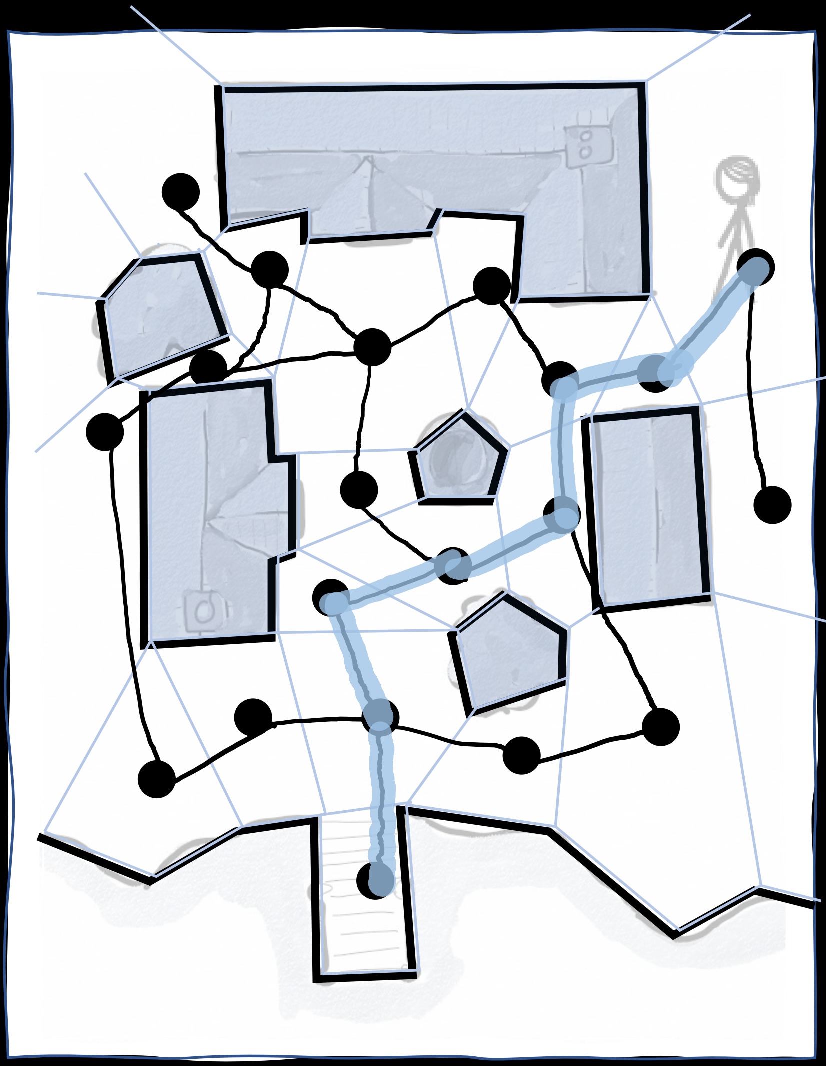
“Can I go for a stroll around the city on a route that crosses each bridge exactly once?”



"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)

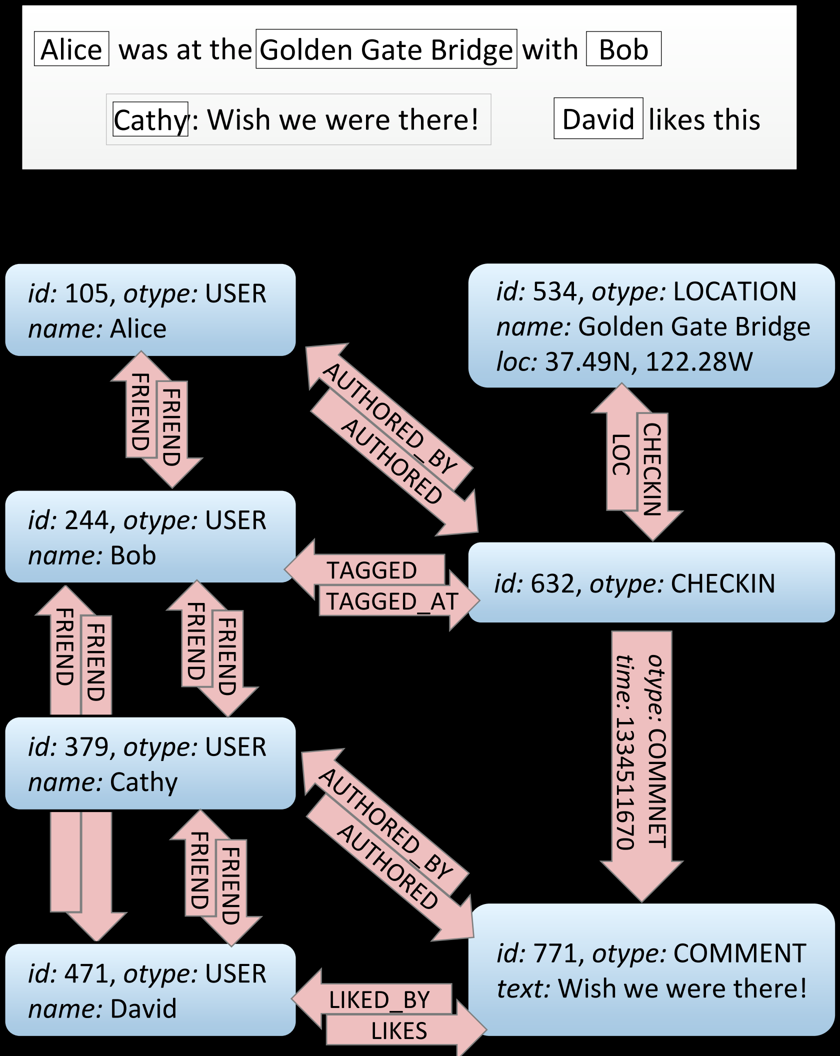
Dwarf Fortress

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?

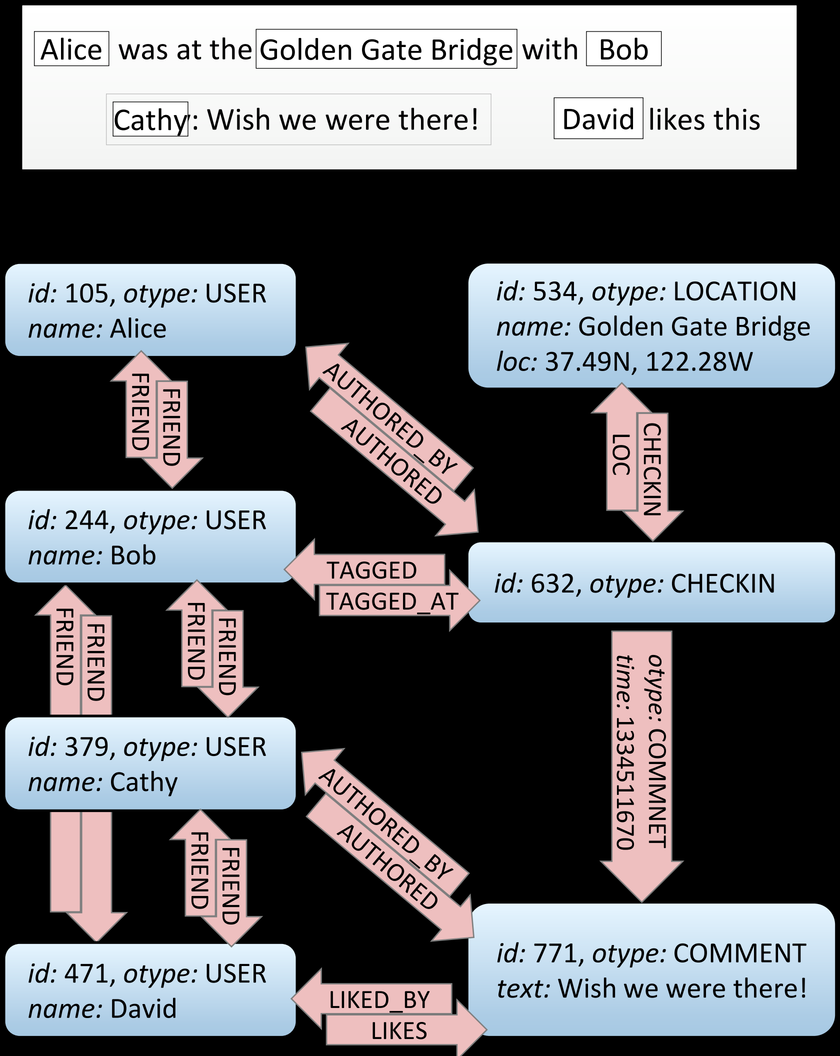
A: Yeah, the base algorithm is only part of it. We use A*, which is fast of course, but it’s not good enough by itself.

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



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



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

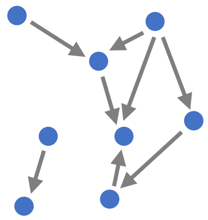
Right from the beginning, and all through the course, we stress that the programmer's task is not just to write down a program, but that his main task is to give a formal proof that the program he proposes meets the equally formal functional specification.

Edsger Dijkstra (1930—2002)
On the cruelty of really teaching computer science, 1988

Graph notation

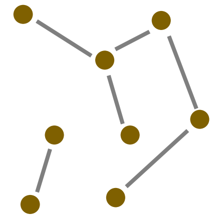
A graph consists of a set of vertices V, and a set of edges E.

directed graphs

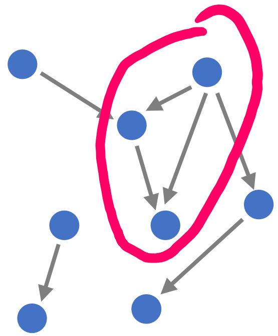


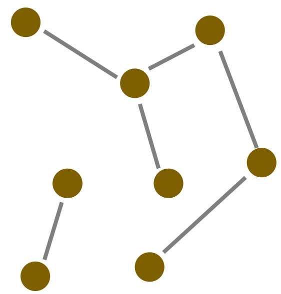
$v_1 \rightarrow v_2$ is how we write
the edge from v_1 to v_2

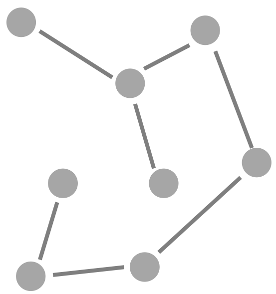
undirected graphs



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



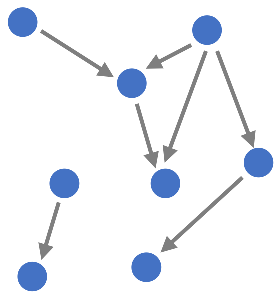


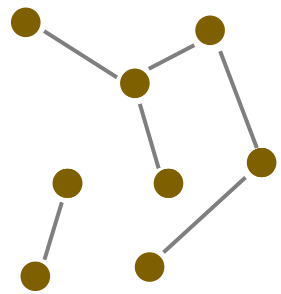


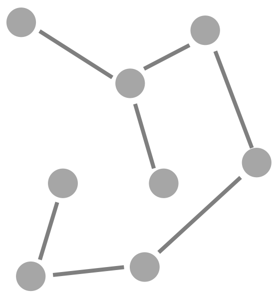
Which of these two graphs is a tree, which a forest?

- 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)





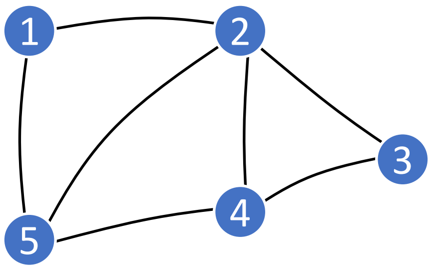


What’s wrong with my definitions for *path* and *cycle*?

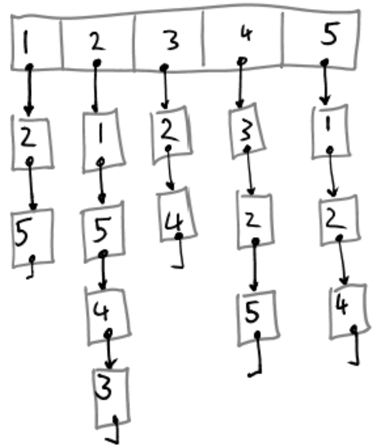
- A *directed acyclic graph* (DAG) is a directed graph without any cycles

- 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

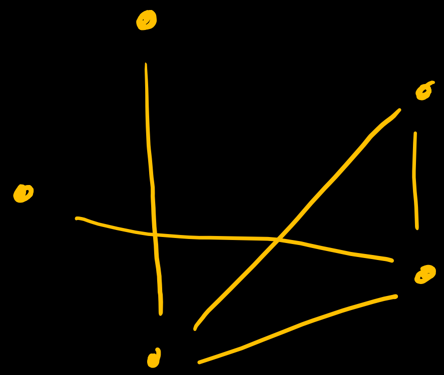
	1	2	3	4	5
1	0	1	0	0	1
2	1	0	1	1	1
3	0	1	0	1	0
4	0	1	1	0	1
5	1	1	0	1	0

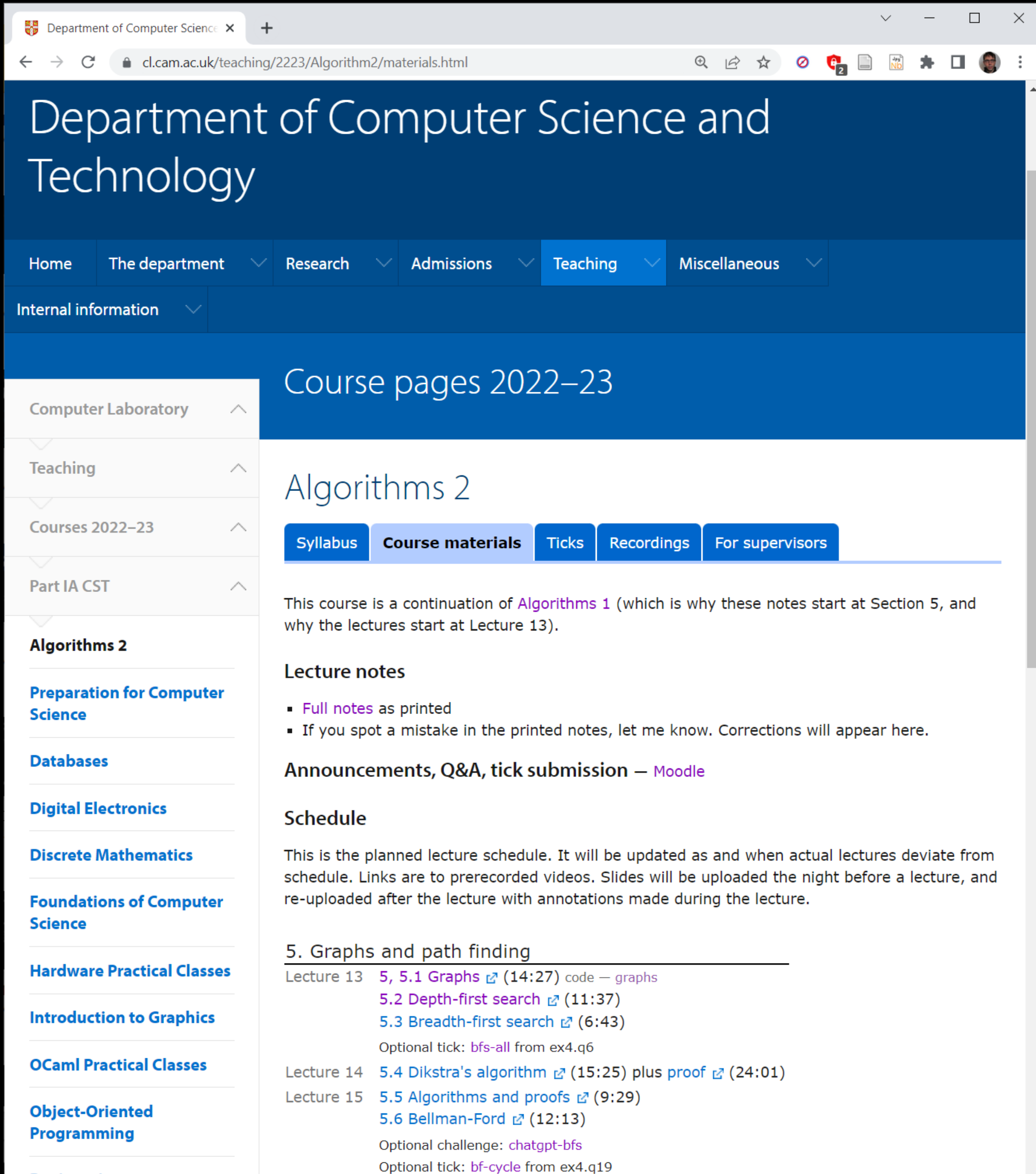
```
np.array([[0,1,0,0,1],
          [1,0,1,1,1],
          [0,1,0,1,0],
          [0,1,1,0,1],
          [1,1,0,1,0]])
```

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?



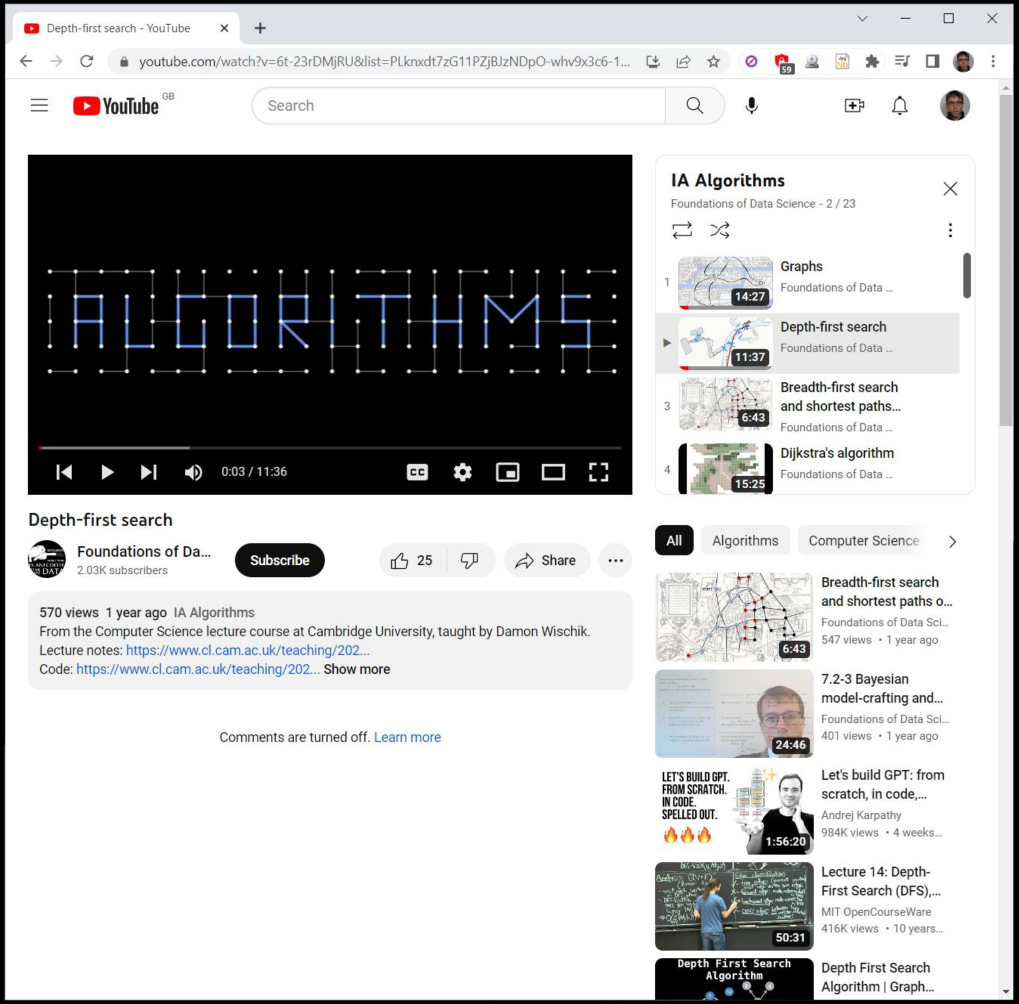


- lecture notes
- example sheets
- slides
- ticks
- recordings

How to learn effectively

PASSIVE LEARNING	ACTIVE LEARNING	REFLECTIVE LEARNING
▪ attend lectures	▪ copy out the lecturer's hand-writing	▪ mini-exercises and example sheets
▪ read code snippets, watch animations, see examples	▪ annotate printed code snippets and examples (using page numbers)	▪ optional ticks
▪ read notes, watch videos		▪ skeptical reading

Pre-recorded videos



Consent to recordings of live lectures

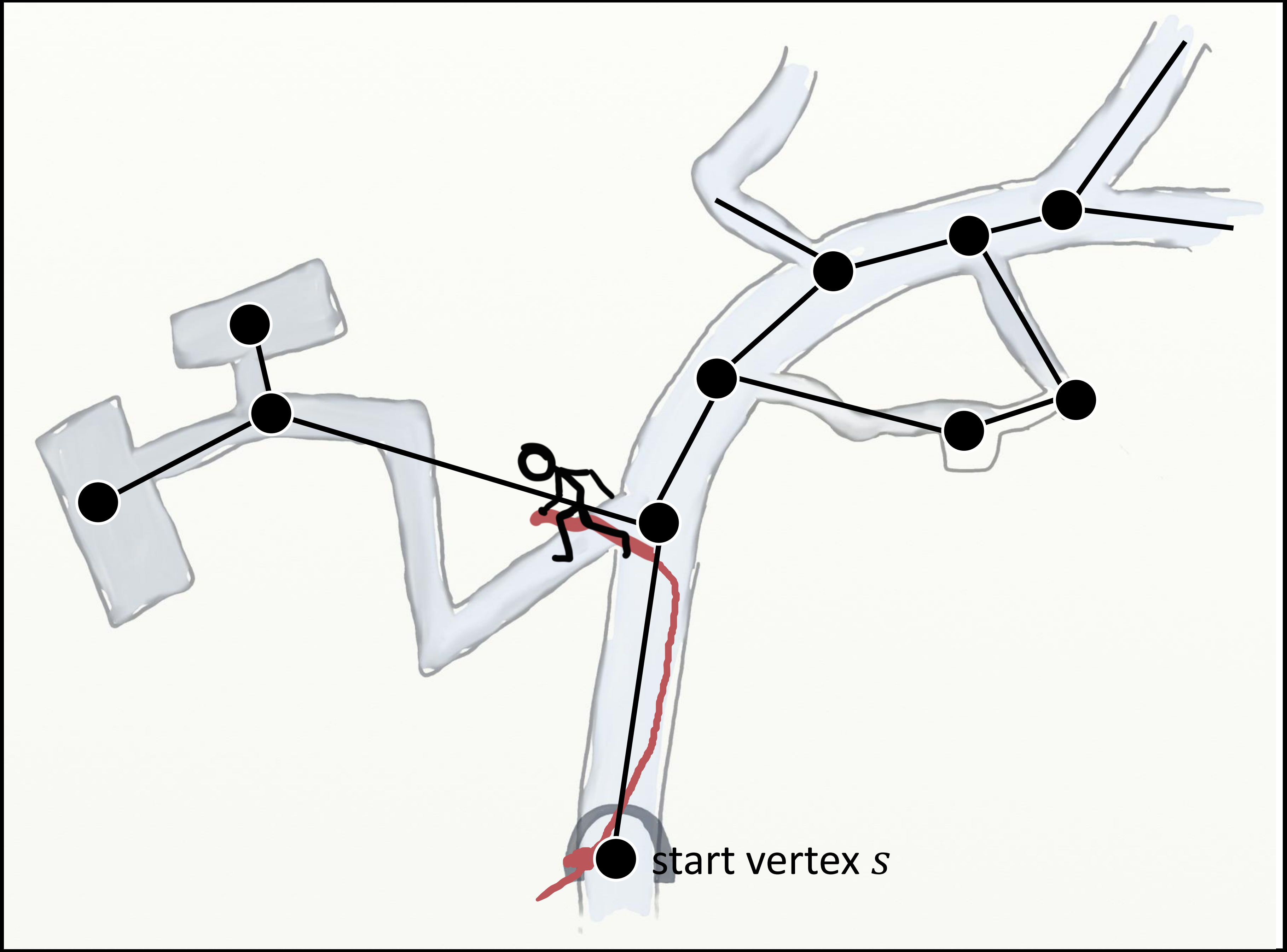
https://www.educationalpolicy.admin.cam.ac.uk/supporting-students/policy-recordings/recordings-student-information-sheet

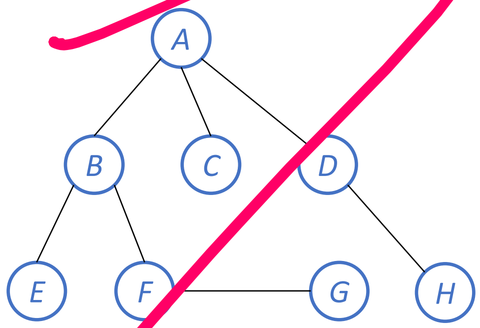
For any teaching session where your contribution is mandatory or expected, we must seek your consent to be recorded.

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

SECTION 5.2

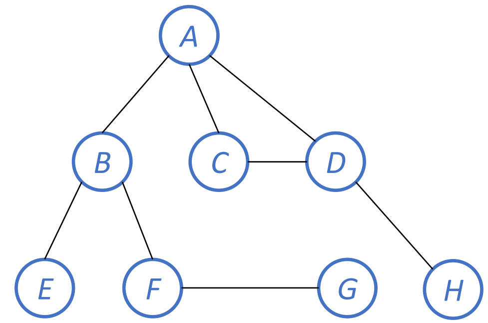
Depth-first search


start vertex s



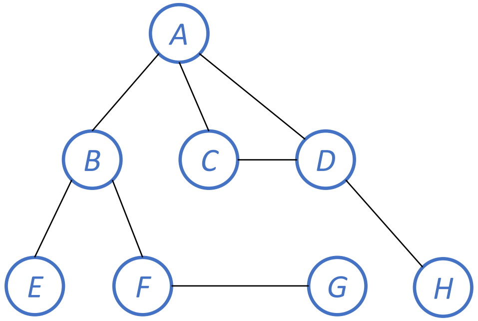

```
def visit(v):
    print("visiting", v)
    for w in v.neighbours:
        visit(w)

visit(A)
```

visiting A
visiting B
visiting A
visiting B
...
RecursionError:
maximum recursion depth exceeded




```
def visit_tree(v, v_parent):
    print("visiting", v, "from", v_parent)
    for w in v.neighbours:
        if w != v_parent:
            visit_tree(w, v)

visit_tree(D, None)
```

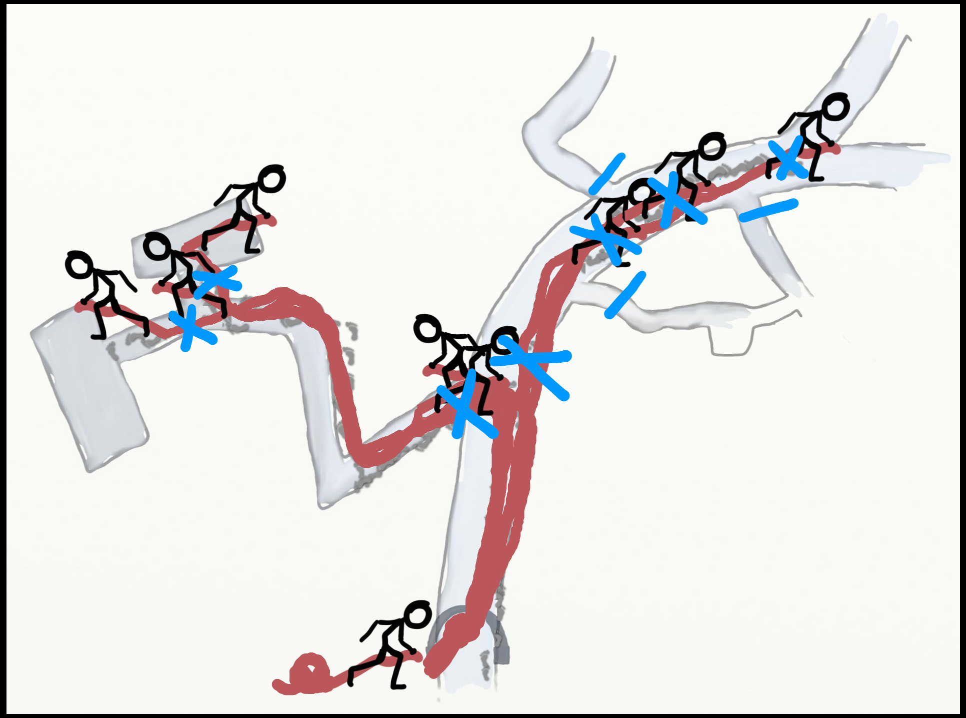
visiting D from None
visiting C from D
visiting A from C
visiting D from A
...
RecursionError:
maximum recursion depth exceeded



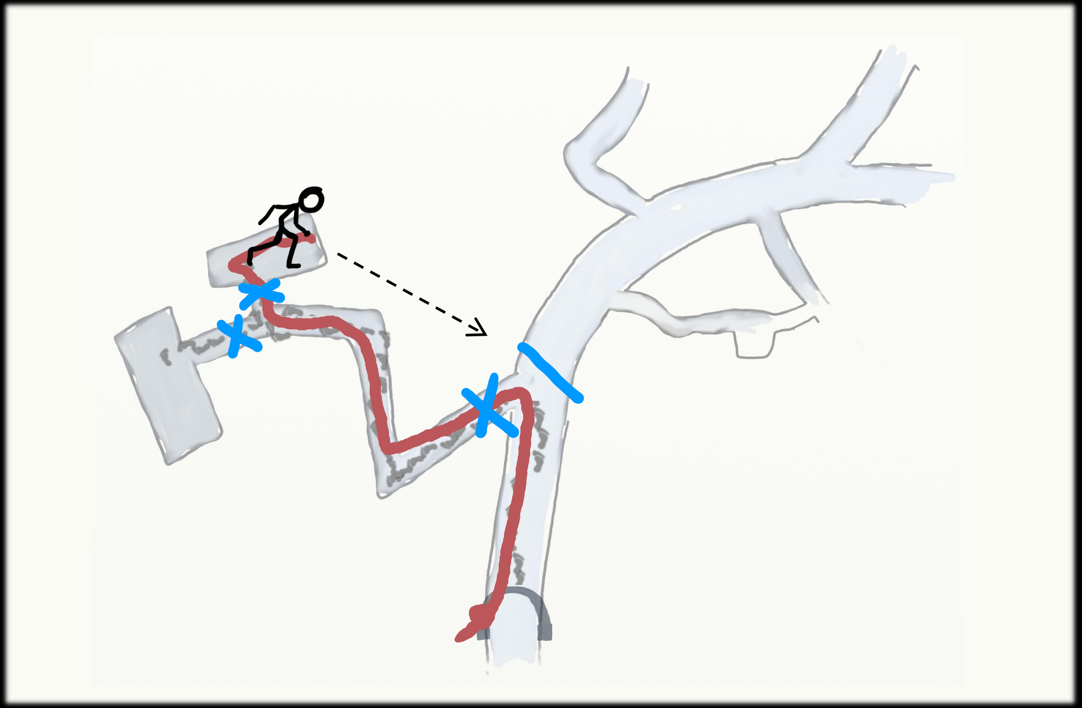

```python
# visit all vertices reachable from s
def dfs_recurse(g, s):
    for v in g.vertices:
        v.visited = False
    visit(s)

def visit(v):
    v.visited = True
    for w in v.neighbours:
        if not w.visited:
            visit(w)
```

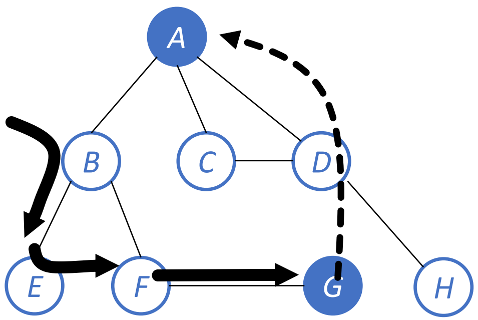
```
dfs_recurse(g, D):
| visit(D):
| | neighbours = [H, C, A]
| | visit(H):
| | | neighbours = [D]
| | | don't visit D
| | | return from visit(H)
| | visit(C)
| | | neighbours = [D, A]
| | | don't visit D
| | | visit(A):
| | | | ...
```

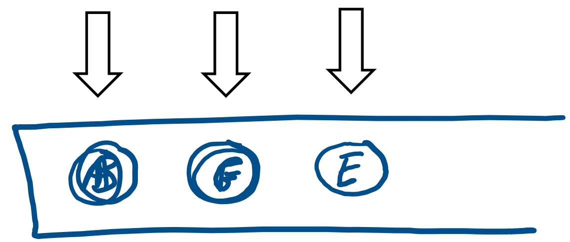
Ariadne's thread ...



but why not just teleport?







```
# visit all vertices reachable from s
def dfs(g, s):
    for v in g.vertices:
        v.seen = False
    toexplore = Stack([s])
    s.seen = True

    while not toexplore.is_empty():
        v = toexplore.popright()
        for w in v.neighbours:
            if not w.seen:
                toexplore.pushright(w)
                w.seen = True
```

Analysis of running time for stack-based dfs

```
# visit all vertices reachable from s
def dfs(g, s):
    for v in g.vertices:
        v.seen = False
    toexplore = Stack([s])
    s.seen = True

    while not toexplore.is_empty():
        v = toexplore.popright()
        for w in v.neighbours:
            if not w.seen:
                toexplore.pushright(w)
                w.seen = True
```

$O(V)$

$O(1)$

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

run for every edge out of every vertex we visit, so $O(E)$

total $O(V+E)$

Analysis of running time for recursive dfs

```python
# visit all vertices reachable from s
def dfs_recurse(g, s):
    for v in g.vertices:
        v.visited = False
    visit(s)

def visit(v):
    v.visited = True
    for w in v.neighbours:
        if not w.visited:
            visit(w)
```

$O(V)$

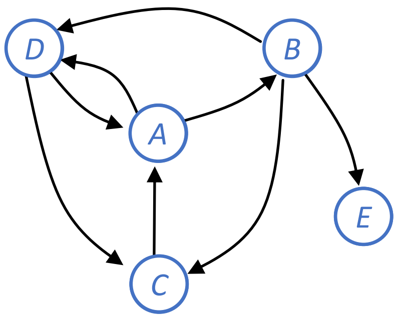
run at most once per vertex, so $O(V)$

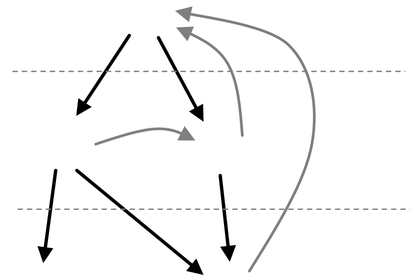
$O(E)$

Total: $O(V+E)$

SECTION 5.2

Breadth-first search / finding shortest path

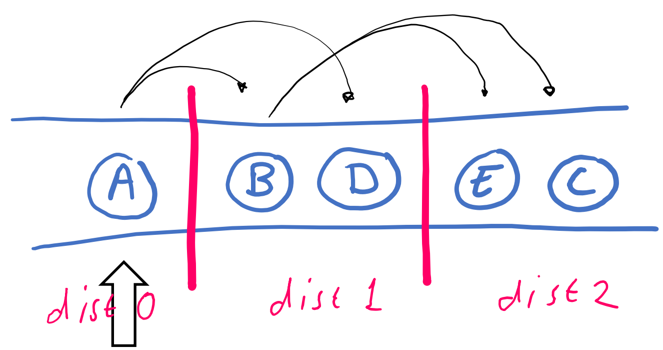




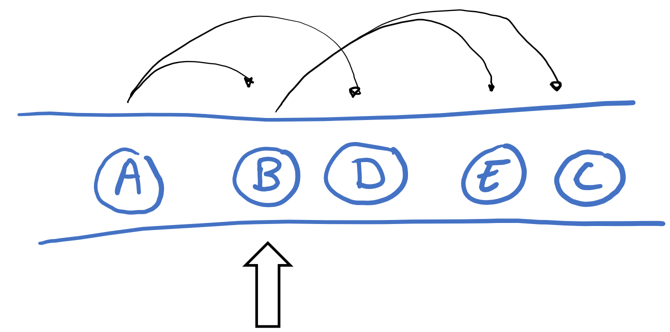
distance from A = 0

distance from A = 1

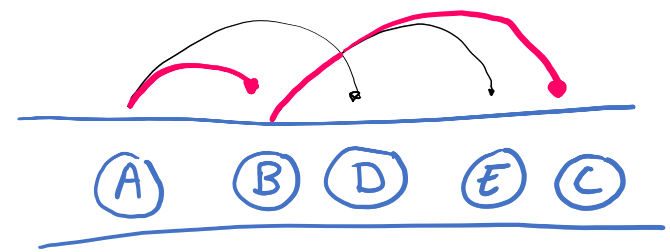
distance from A = 2

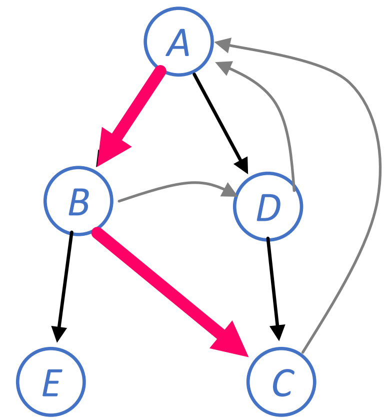


```python
# Visit all the vertices in g reachable from start vertex s
def bfs(g, s):
    for v in g.vertices:
        v.seen = False
    toexplore = Queue([s])
    s.seen = True

    while not toexplore.is_empty():
        v = toexplore.popleft()
        for w in v.neighbours:
            if not w.seen:
                toexplore.pushright(w)
                w.seen = True
```



```
# Find a path from s to t, if one exists
def bfs_path(g, s, t):
    for v in g.vertices:
        (v.seen, v.come_from) = (False, None)
...
    while not toexplore.is_empty():
        v = toexplore.popleft()
        for w in v.neighbours:
            if not w.seen:
                toexplore.pushright(w)
                (w.seen, w.come_from) = (True, v)
...
    if t.come_from has not been set:
        there is no path from s to t
    else:
        reconstruct the path from s to t,
        working backwards
```





Analysis of running time for bfs

```
# Visit all the vertices in g reachable from start vertex s
def bfs(g, s):
    for v in g.vertices:
        v.seen = False
    toexplore = Queue([s])
    s.seen = True

    while not toexplore.is_empty():
        v = toexplore.popleft()
        for w in v.neighbours:
            if not w.seen:
                toexplore.pushright(w)
                w.seen = True
```

O(V+E)

same as for dfs

Analysis of running time for stack-based dfs

```
# visit all vertices reachable from s
def dfs(g, s):
    for v in g.vertices:
        v.seen = False
    to_explore = Stack([s])
    s.seen = True

    while not to_explore.is_empty():
        v = toexplore.popright()
        for w in v.neighbours:
            if not w.seen:
                toexplore.pushright(w)
                w.seen = True
```

Lines 3–4: O(V)

Lines 5–6: O(1)

Line 8–9: at most once per vertex, so O(V)

Lines 10–13: run for every edge out of every vertex we visit, so O(E)

total O(V+E)

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 Dikstra'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

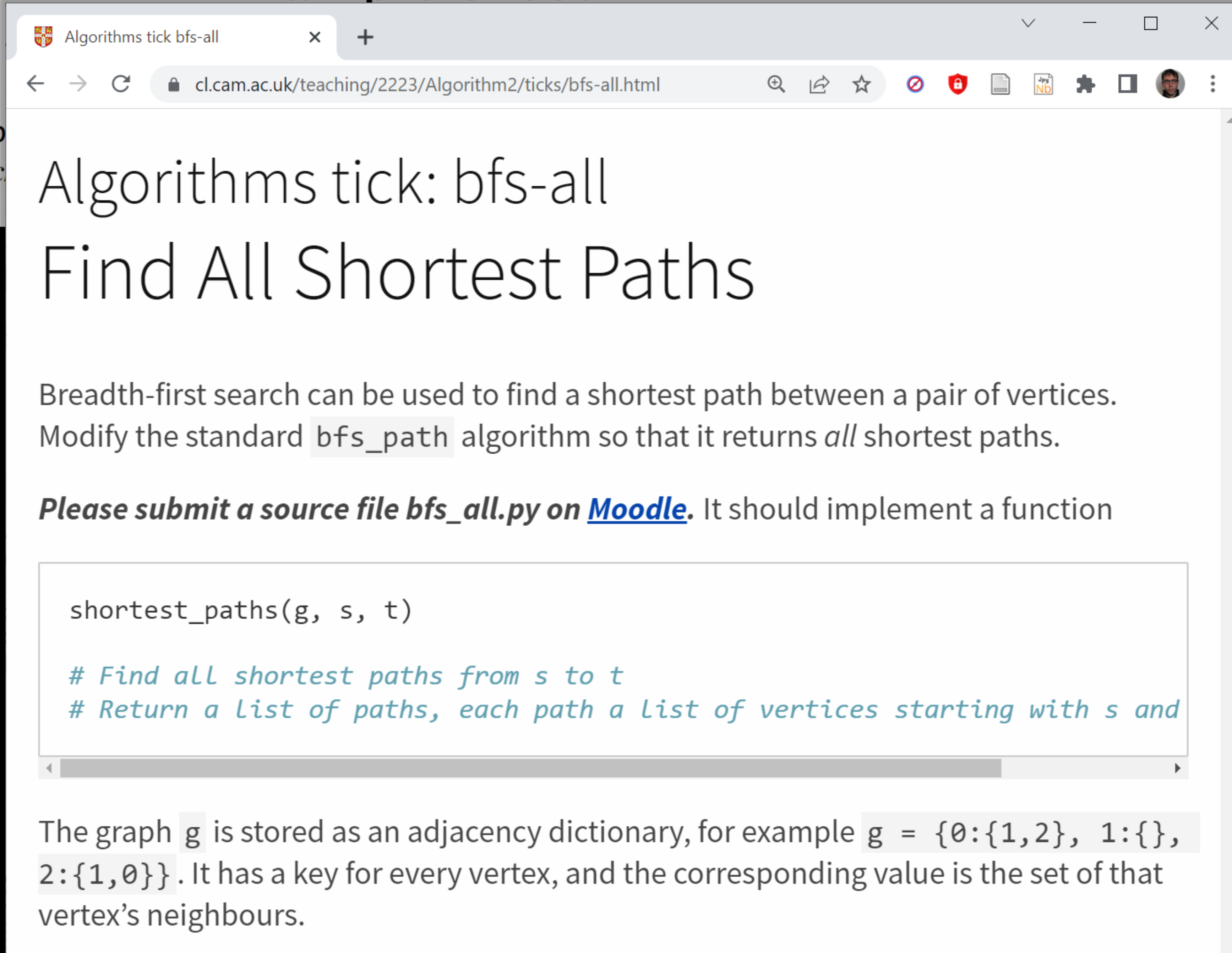
Lecture 16 5.7 Dynamic programming (13:06)
5.8 Johnson's algorithm (13:43)
Example sheet 4 [pdf]

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 b … *rse*
website, for you to chec



Algorithms tick bfs-all

cl.cam.ac.uk/teaching/2223/Algorithm2/ticks/bfs-all.html

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.

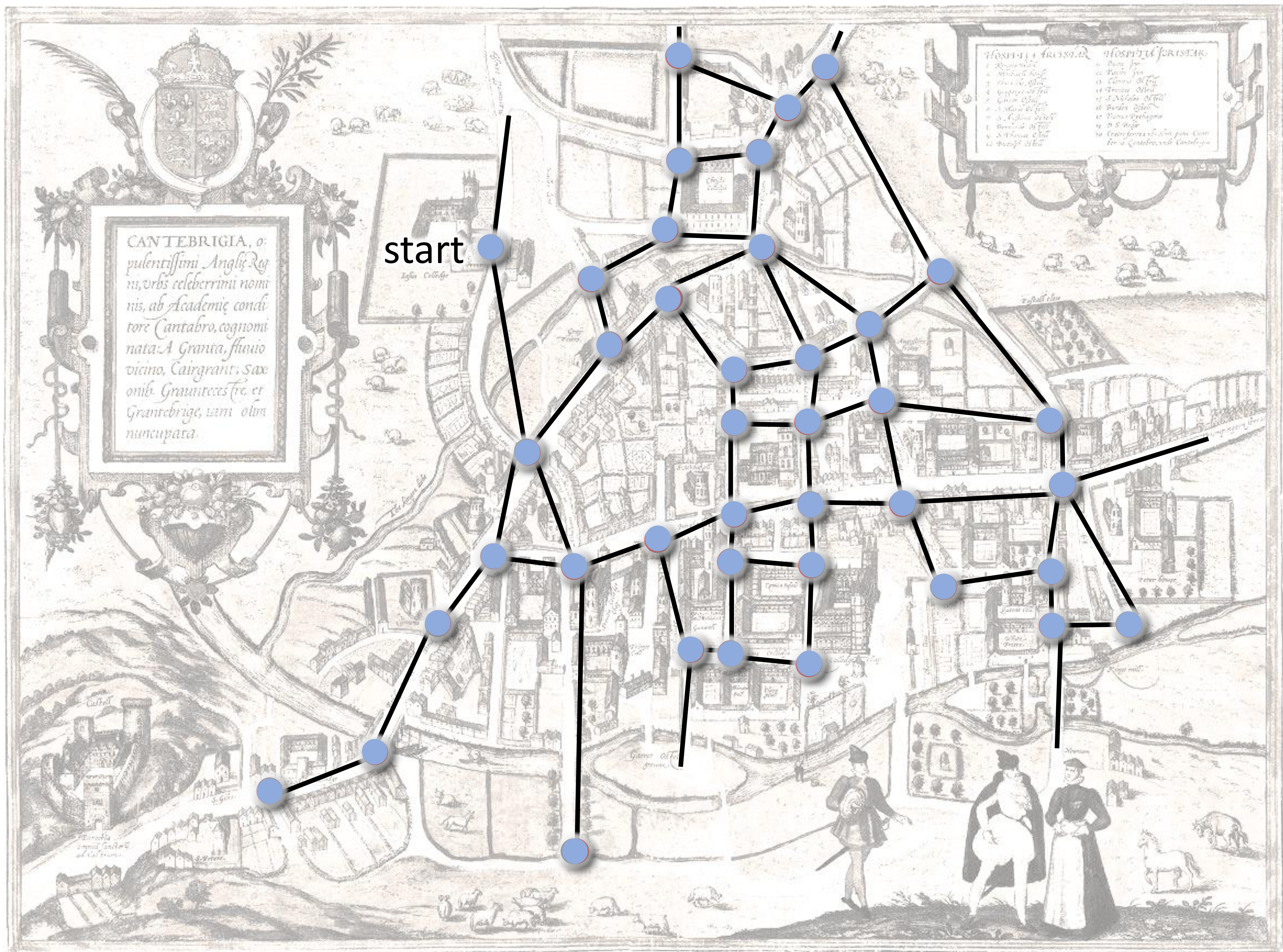
Please submit a source file bfs_all.py on Moodle. It should implement a function

```
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 <u>proof of correctness</u>.



not yet seen

distance [redacted]

visited