Graphs and path finding
A directed graph is an ordered pair $g = (V, E)$ where $V$ is a set ("vertices") and $E$ is a relation on $V$ ("edges").
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]}
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)
How can we do path-finding at scale?

https://stackoverflow.blog/2021/12/31/700000-lines-of-code-20-years-and-one-developer-how-dwarf-fortress-is-built/
Cambridge Game Jam 2024
Build a Game in 48 Hours at Uni Of Cam!
9th - 11th February

Sign Up!  Join our Discord!

2 21 9 36
Days Hours Mins Secs
Until Game Jam!

In collaboration with: Sponsored by:

\(\text{CUCaTS}\)

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

❖ How fast will an epidemic of misinformation spread?
❖ At whom should I target my advertising?
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 $v_2$
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 a path between them.)

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 a path between them.)

What's wrong with my definitions for path and cycle?
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]
}
```

Memory: $O(V+E)$

Note: when we write this, we really mean $O(V + 1E)$, since $V$ and $E$ are sets.

```
Adjacency matrix
```

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

Memory: $O(V^2)$
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?
The next eight lectures

- Clever graph algorithms
- Performance analysis
- Proving correctness
- What we can model with graphs
what was printed out earlier: 2021 notes

what’s online, and will be ready to collect on Friday: 2024 notes
SECTION 5.2

Depth-first search
PROBLEM STATEMENT. Given a start vertex $s$, and given the $v \mapsto \text{neighbours}(v)$ function, list all the vertices of the graph.

How might we navigate a labyrinth?
```python
1 def visit(v):
   2    print("visiting", v)
   3    for w in v.neighbours:
   4       visit(w)

visit(A)

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

```

```python
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)

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
```
This algorithm needs to keep track of which vertices it's seen. It'd be cleaner to store this as a Set — but we haven't yet done performance analysis of Set operations, so in this code snippet I'm using a per-vertex attribute instead.
# 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)
Ariadne gave Theseus a ball of thread. She told him to tie one end at the entrance of the labyrinth, and to unroll the ball as he delved the branching paths. And to mark with chalk those passages he explored. After Theseus slew the Minotaur, he could follow the thread back to the entrance, where Ariadne was waiting.
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

```python
# 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+E) since we never visit a vertex more than once.
    O(V) since it looks at every edge from every vertex.

So total cost \( O(V+E) \). ```
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)
```

Cost: $O(V + E)$ for the same reason.

We don't try to work out the cost of a single call to `visit()`. Instead, we just look at aggregate costs.
SECTION 5.2

Breadth-first search / finding shortest path
PROBLEM STATEMENT. Given a start vertex $s$, and an end vertex $t$, find the shortest path from $s$ to $t$. 

![Diagram of a graph with vertices A, B, C, D, and E, showing distances from vertex A: distance from A = 0, distance from A = 1, distance from A = 2.]
# 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

Cost: $O(V + E)$

Same reasoning as for 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
# 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

Q. How might we find a shortest path from A to C?
3. Algorithm design

Lecture 05 [slides]  3.1 Dynamic programming
Lecture 06 [slides]  Dynamic programming examples
Lecture 07 [slides]  3.2 Greedy algorithms

First half of example sheet 2 [pdf]
Optional tick: huffman
Tick 2, due 19 Feb (TBC)

4. Data structures (intro)

Lecture 08 [slides.pre]  4.1 Memory and pointers
                      4.1–4.2 List, tree, stack, queue, dictionary
                      4.7 Hash tables
                      4.9 Priority queues

Rest of example sheet 2 [pdf]

5. Graphs and path finding

Lecture 09 [slides.pre]  5.1 Graphs (14:27)
                      5.2 Depth-first search (11:37)
                      5.3 Breadth-first search (6:43)
Lecture 10  5.4 Dijkstra’s algorithm (15:25) plus proof (24:01)
Lecture 11  5.5 Algorithms and proofs (9:29)
                      5.6 Bellman-Ford (12:13)
Lecture 12  5.7 Dynamic programming (13:06)
                      5.8 Johnson’s algorithm (13:43)

Example sheet 4 [pdf]
Optional tick: bfs-all from ex4.q6

6. Graphs and subgraphs

Lecture 13  6.1 Flow networks (9:31)
                  6.2 Ford-Fulkerson algorithm (21:55)
Lecture 14  6.3 Max-flow min-cut theorem (19:38)
Lecture 15  6.4 Matchings (11:01)
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 \texttt{bfs\_path} algorithm so that it returns all shortest paths.

\textit{Please submit a source file \texttt{bfs\_all.py} on Moodle.} It should implement a function

\begin{verbatim}
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
\end{verbatim}

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 5.4, to familiarize yourself with Dijkstra’s algorithm.

We will spend Friday’s lecture going through the proof of correctness.