SECTION 5.1

Depth-first search
Ariadne’s thread ... but why not just teleport?
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)
```
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
```
SECTION 5.2
Breadth-first search / finding shortest path
# Visit all the vertices in g reachable from start vertex s

```python
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
```
Breadth First Search

The key idea for all of these algorithms is that we keep track of an expanding ring called the *frontier*. On a grid, this process is sometimes called “flood fill”, but the same technique works for non-grids. **Start the animation** to see how the frontier expands:

https://www.redblobgames.com/pathfinding/a-star/introduction.html#breadth-first-search
# Find a path from s to t, if one exists

```python
def bfs_path(g, s, t):
    for v in g.vertices:
        (v.seen, v.come_from) = (False, None)
        ... # remaining code
    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)
        ... # remaining code
    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 stack-based dfs

```python
# 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 = to_explore.pop_right()
        for w in v.neighbours:
            if not w.seen:
                to_explore.push_right(w)
                w.seen = True
```

Analysis of running time for bfs

```python
# Visit all the vertices in g reachable from start vertex s
def bfs(g, s):
    for v in g.vertices:
        v.seen = False
    to_explore = Queue([s])
    s.seen = True
    while not to_explore.is_empty():
        v = to_explore.pop_left()
        for w in v.neighbours:
            if not w.seen:
                to_explore.push_right(w)
                w.seen = True
```

Analysis of running time:

- **dfs**
  - \(O(V + E)\) in the worst case, where \(V\) is the number of vertices and \(E\) is the number of edges. This is because we visit each vertex and each edge exactly once.

- **bfs**
  - \(O(V + E)\) in the worst case, similar to dfs, as it also visits each vertex and edge once.

These complexities hold true regardless of whether the graph is directed or undirected, as long as the graph is connected.
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 (6:13)
   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 (31:55)
Question 6. Modify bfs_all.py 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

```python
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}}
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.
In a graph where the edges have costs (e.g. travel time), we can find shortest paths by using a similar “grow the frontier” algorithm to bfs.
```python
def dijkstra(g, s):
    for v in g.vertices:
        v.distance = ∞
    s.distance = 0
    toexplore = PriorityQueue([s], sortkey = λv: v.distance)

    while not toexplore.is_empty():
        v = toexplore.popmin()
        # Assert: v.distance is distance(s to v)
        # Assert: v is never put back into toexplore
        for (w, edgecost) in v.neighbours:
            dist_w = v.distance + edgecost
            if dist_w < w.distance:
                w.distance = dist_w
                if w in toexplore:
                    toexplore.decreasekey(w)
                else:
                    toexplore.push(w)
```

<table>
<thead>
<tr>
<th>popped</th>
<th>toexplore</th>
</tr>
</thead>
<tbody>
<tr>
<td>{}</td>
<td>[s]</td>
</tr>
<tr>
<td>{s}</td>
<td>[b, a, d]</td>
</tr>
<tr>
<td>{s, b}</td>
<td>[c, a, d]</td>
</tr>
<tr>
<td>{s, b, c}</td>
<td>[d, a]</td>
</tr>
</tbody>
</table>
```python
def dijkstra(g, s):
    for v in g.vertices:
        v.distance = ∞
    s.distance = 0
toexplore = PriorityQueue([s], sortkey = λv: v.distance)

    while not toexplore.is_empty():
        v = toexplore.popmin()
        # Assert: v.distance is distance(s to v)
        # Assert: v is never put back into toexplore
        for (w, edgecost) in v.neighbours:
            dist_w = v.distance + edgecost
            if dist_w < w.distance:
                w.distance = dist_w
                if w in toexplore:
                    toexplore.decreasekey(w)
                else:
                    toexplore.push(w)
```

Total = \(O(v) + O(v) \times O(\text{popmin}) + O(e) \times O(\text{push key})\)
\[
= O(E + V \log V)
\]

where \(n\) = # items stored
\(n \leq V\) by line 10
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.

Programming is one of the most difficult branches of applied mathematics; the poorer mathematicians had better remain pure mathematicians.

Edsger Dijkstra (1930—2002)
*On the cruelty of really teaching computer science*, 1988
Problem statement
Given a directed graph in which each edge is labelled with a cost $\geq 0$, and a start vertex $s$, compute the distance from $s$ to every other vertex, where ...

$\text{cost}(u \rightarrow v)$ is the cost associated with edge $u \rightarrow v$

$\text{cost}(u \rightarrow \cdots \rightarrow v)$ is the sum of edge costs along the path $u \rightarrow \cdots \rightarrow v$

$\text{distance}(u \text{ to } v) = \begin{cases} 
\text{min cost of any path } u \rightarrow \cdots \rightarrow v, & \text{if one exists} \\
0, & \text{if } u = v \\
\infty, & \text{otherwise}
\end{cases}$
The “breakpoint” proof strategy

1. Decide on a property we want to be true at all times
2. Assume it’s true up to time $T - 1$
3. Show that it must therefore be true at time $T$
Assertion line 10.
A vertex \( v \), once popped, is never put back into the priority queue

Proof.

1. A vertex \( w \) is only pushed into the priority queue when we discover a path shorter than \( w . \text{distance} \).
2. Once \( v \) is popped, \( v . \text{distance} = \text{distance}(s \text{ to } v) \), so there can be no shorter path.

Hence \( v \) is never pushed back.
Theorem.

i. The algorithm terminates

ii. When it does, for every vertex \( v \), \( v\.distance = distance(s \text{ to } v) \)

iii. The two assertions never fail ✅