COMPSCI220 – Algorithms and data structures

Ulrich Speidel
ulrich@cs.auckland.ac.nz
A bit about myself

• I'm a physicist by training and didn't "become" a computer scientist until my PhD

• Have taught in this department since 2000

• Have been involved in a wide variety of courses ranging from application development, data communication, Internet programming and introductory programming to computer architecture

• I'm quite an approachable person & you can find me in my office (303S.594) ☺️
Why I teach this course

• My research is in a variety of fields, including applied ones. What we do here mixes into most of these.

• E.g.: I have an interest in variable-length codes in data communication. Such codes are often visualised as trees, and digraphs turn up all over the place in coding and communication – even in web programming.

• E.g.: One of the most important algorithms I've developed works in $O(n \log n)$ as opposed to its predecessor's $O(n^2)$ – thanks to a complex but highly efficient data structure.

• E.g., algorithms such as Dijkstra's or Floyd's which you'll meet in part 3 of 220 are of fundamental importance in data networks – but not just there.

• I don't like theory for its own sake. I like it because it ever so often really helps me in the practical side of what I do. And because of that, I ever so often feel obliged to give something back.
Communication with me

• Don't be shy to approach me, but try and make life easier for all of us

• Put "COMPSCI220" in the subject of your e-mail on a question that relates to COMPSCI220

• Try to be as concise as possible in your question. It doesn't need to be as short as a text message. It helps if you refer to slide numbers, assignment questions, etc.

• I may anonymise your question and cc the rest of the class into my answer if I think it's of interest to others. Check and keep your e-mail – it'll often already contain the answer you're looking for.

• Note that around assignment due dates, tests and exams, I often get lots of questions by e-mail. Checking your past class e-mail is often a faster way to get an answer!

• Get in early and don't wait to the last minute if you need help.
Course overview

• This part of the course has 11 lectures

• If there is demand, I'll give an extra lecture / tutorial / review at the end
Preliminaries

• The following material has probably been covered by Nevil Brownlee

• I’m including my old slides here just in case we need to look back to a concept or in case something we need wasn’t covered
Graphs

A graph $G$ consists of two sets $V$ and $E$.

- $V$ is the set of vertices (also called nodes). A vertex (not "vertice"!) is often graphically displayed as a point or junction point (like the seven little circles on the right).

- $E$ is the set of edges. Each edge connects two vertices $u$ and $v$ in $V$. Edges have no direction – they equally go from $u$ to $v$ and from $v$ to $u$. We denote an edge as $(u,v)$ with the understanding that $(u,v)=(v,u)$. The graph on the right contains eight edges (the black lines).

- If two vertices $u$ and $v$ are connected by an edge, they are said to be adjacent. We also say that $u$ is a neighbour of $v$.

- It is not uncommon to number the vertices or otherwise label them uniquely. This is then called a labelled graph.
Graphs - remarks

• We also write $G(V, E)$ for the graph itself, or $E(G)$ for the edges, or $V(G)$ for the vertices.

• Note: not every vertex of a graph needs to have an edge attached; not every pair of vertices needs to be connected by an edge, and it's perfectly OK to have a graph with many vertices but no edges at all!
Order and size of a graph

- The **order** of a graph $G$ is the number of its vertices: $|V(G)|$

- The **size** of a graph $G$ is the number of its edges: $|E(G)|$

- Remember that $|X|$ denotes the number of elements of a set $X$.

In this example:

$|V(G)| = 7$

$|E(G)| = 8$
Real-life examples of graphs

- Computer networks
- Street networks (with two-way streets only)
- Airline route networks (as long as the planes fly both ways on each route)
How many edges can a graph have?

- Observation 1: At most one edge per pair of different vertices.

- Observation 2: There are \(|V(G)| (|V(G)| - 1)\) possible such vertex pairs \((u,v)\), but there can be only one such edge for \((u,v)\) and \((v,u)\).

- So we need to divide this number by two:

\[ 0 \leq |E(G)| \leq |V(G)| (|V(G)| - 1)/2. \]
Sparse and dense graphs

• If $|E(G)|$ approaches the high end of the scale $0 \leq |E(G)| \leq |V(G)| (|V(G)| - 1)/2$, we say that the graph is **dense**

• If $|E(G)|$ approaches the low end of the scale $0 \leq |E(G)| \leq |V(G)| (|V(G)| - 1)/2$, we say that the graph is **sparse**

• Knowing whether a graph that we are dealing with is dense or sparse makes a difference in how it is best stored when space efficiency is important.

• There is no defined boundary between when we would call a graph sparse or dense – but the criterion of storage could help us decide if we wanted such a criterion.
Sparse and dense graphs

A sparse graph (2 out of 21 possible edges)

Generally more efficient to store edges by recording between which vertices they occur

A dense graph (16 out of 21 possible edges)

Generally more efficient to store edges by listing vertex pairs and recording between which pairs there are edges
The degree of a vertex in a graph

• The degree of a vertex $v$ is the number of edges that terminate in this vertex

Node 1: degree 5  
Node 2: degree 4  
Node 3: degree 5  
Node 4: degree 4  
Node 5: degree 4  
Node 6: degree 5  
Node 7: degree 5

• Observation 1: The number of edges is the sum of the degrees of all vertices divide by 2 (Why? Every edge has two endpoints!)

• Observation 2: In sparse graphs, the nodes tend to have a lower degree than in dense graphs.
A sense of direction: di(rected) graphs

- In a **directed graph** (or **digraph** for short), we replace the set \( E \) of edges by a set \( E \) of **arcs**.

- The difference between an edge and an arc is that an arc has a direction: an edge runs between two vertices \( u \) and \( v \), an arc runs from a vertex \( u \) to a vertex \( v \).

- The textbook draws arcs as arrows – the fact that we draw it curved allows us to have arcs between two vertices in both directions. We use double-headed arrows here to indicate that two vertices are connected by arcs in both directions (LLDS\(^1\)).

- This also requires us to extend and amend our previous definitions and terminology a little.

\(^1\) Lazy Lecturer Drawing Syndrome
There goes the neighbourhood...

- In a directed graph, we now have two sorts of neighbours:
  - A neighbour \( v \) of \( u \) is an **out-neighbour** of \( u \) if there is an arc from \( u \) to \( v \).
  - A neighbour \( v \) of \( u \) is an **in-neighbour** of \( u \) if there is an arc from \( v \) to \( u \).

- Similarly, there are two sorts of degrees now: the **out-degree** (number of out-neighbours) and the **in-degree** (number of in-neighbours).

- Obviously, now \((u,v)\neq(v,u)\).

- Size of a digraph = number of arcs

E.g.:
- 1 is an out-neighbour of 5
- 2 is an in-neighbour of 6
- 2 is an in-neighbour and out-neighbour of 4 (and vice versa)
- 7 has in-degree 0 and out-degree 3
- 4 has in-degree 3 and out-degree 3
- Size is 10
Sources and sinks...

- A vertex with in-degree 0 is called a **source** (all arcs "flow out" from the vertex).
- A vertex with out-degree 0 is called a **sink** (all arcs "flow into" the vertex).

1 and 6 are sinks
5 and 7 are sources
The other nodes are neither sinks nor sources
Going for a walk...

• A **walk** is a sequence of vertices $v_0, v_1, \ldots, v_l$, such that $(v_i, v_{i+1})$ is an arc in $E$ for $0 \leq i < l$.

• A walk can pass by the same vertex twice, i.e., $v_i = v_j$ is possible even for $i \neq j$.

• The **length** of the walk is $l$. This is the number of arcs involved.

E.g., 4 5 6 4 3 1 2 3 1 7 2 is a walk.
...but are we sticking to a path?

- A **path** is a walk in which no vertex is repeated.

- A **cycle** is a walk of length 3 or more on a graph or any walk on a digraph where \( v_0 = v_i \), i.e., a walk that ends in the same vertex that it started in and that does not simply go down one arc and comes back along an arc in the opposite direction.

We'll encounter cycles again later.

E.g., 4 5 6 4 3 1 2 3 1 7 2 is a walk but not a path: We visit 4, 3, 1, and 2 more than once.
E.g., 4 5 6 2 3 1 is a path.
E.g., 2 3 1 2 is a cycle.
E.g., 3 1 3 is a cycle in the digraph but not a cycle in its underlying graph (obtained by removing the arrowheads, see slide 26).
Subgraphs and subdigraphs

• A sub(di)graph $G'(V',E')$ of a (di)graph $G(V,E)$ is a (di)graph for which $V' \subseteq V$ and $E' \subseteq E$.

• Note that $G'(V',E')$ still has to be a (di)graph in its own right, i.e., all edges/arcs in $E'$ must be associated with vertices in $V'$.

• If $V'=V$, i.e., if it $G'$ contains all vertices of $G$, the $G'$ is called a spanning sub(di)graph.
Exercise: Subdigraph or not?

\[ G \]

\[ G' \]
Exercise: Subdigraph or not?

$G$

$G'$
Exercise: Spanning subdigraph or not?

$G$

$G'$
Induced sub(di)graphs

• A sub(di)graph $G'$ of $G$ based on a subset of vertices $V' \subseteq V$ and the edges/arcs in $E' \subseteq E$ is referred to as the induced sub(di)graph of $V'$ if the following condition applies:

• For all $u, v \in V'$, if $(u, v)$ is an edge/arc in $E$ then $(u, v)$ is an edge/arc in $E'$.

• In plain English: it's the sub(di)graph consisting of all vertices in $V$ that are also in $V'$ and all the edges/arcs in $E$ that connect them.
Induced subdigraph

E.g. $V' = \{1, 2, 3, 4, 6\}$
Reverse digraph

- The reverse digraph $G_r$ of a digraph $G$ is constructed by reversing the direction of all arcs in $G$.
Underlying graph of a digraph

- The **underlying graph** $G'$ of a digraph $G$ is constructed by converting all arcs in $G$ into edges.

![Graph $G$](image1)

![Graph $G'$](image2)
Storing a (di)graph on a computer

• Computers don't know (di)graphs – they know just bits and bundles of bits called "data structures". To store a (di)graph, we need to find an appropriate data structure.

• Choice of two main data structures for general (di)graphs of order $n$: an adjacency matrix or an adjacency list.

• The adjacency matrix is an $n$-by-$n$ matrix whose rows represent vertices that an arc/edge may originate from and whose columns represent vertices an arc/edge may end at. Its entry $(i, j)$ in row $i$ and column $j$ is 0 (false) if there is no arc/edge from vertex $i$ to vertex $j$, and 1 (true) if there is such an arc/edge.

• The adjacency list is a list with $n$ entries $L_1, L_2, \ldots, L_n$. Each of these entries $L_i$ is a sequence of between 0 and $n-1$ numbers. The sequence $L_i$ list the vertices that are (out-)neighbours of vertex $i$.

• Note: Textbook error in 2nd ed. Definition 4.22 on p. 88: number the sequences from 1 to $n$ not 0 to $n-1$, as shown on this slide
Adjacency matrix example

\[ G = \begin{bmatrix}
0 & 1 & 1 & 0 & 0 & 0 & 1 \\
0 & 0 & 1 & 0 & 0 & 1 & 0 \\
1 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 1 & 0 & 1 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 1 & 0 \\
0 & 1 & 0 & 1 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 & 0 & 0 & 0 \\
\end{bmatrix} \]

Arc from vertex 2 to 3

Arcs to out-neighbours of vertex 6 (vertices 2 and 4)
Adjacency list example

\[ L_1 = 2,3,7 \]
\[ L_2 = 3,6 \]
\[ L_3 = 1 \]
\[ G = L_4 = 3,5 \]
\[ L_5 = 6 \]
\[ L_6 = 2,4 \]
\[ L_7 = 2 \]

Arc from vertex 2 to 3

Arcs to out-neighbours of vertex 6 (vertices 2 and 4)
Adjacency list or matrix?

• Whether it is better to use an adjacency matrix or an adjacency list to implement a (di)graph in a computer program depends on three factors governing storage requirement and performance:

• The order of the (di)graph: the storage space we require for an adjacency matrix is $O(n^2)$.

• The size of the (di)graph: the storage space we require for an adjacency list is $O(n^2 \log n)$. However, in most practical cases, the graphs are sparse. E.g., if the in-/out-degree is limited, the space requirement is only $O(n \log n)$.

• What we want to do with the graph (operations). This requires a bit more discussion.
Elementary graph operations

• Does the arc \((u,v)\) exist?
• What is the outdegree of vertex \(u\)?
• What is the indegree of vertex \(u\)?
• Add an arc between two vertices \(u\) and \(v\).
• Delete the arc between vertices \(u\) and \(v\).
• Add a node to the (di)graph.
• Delete a node from the (di)graph.
Does the arc \((u,v)\) exist?

• In the adjacency matrix, we need to find the entry in row \(u\), column \(v\). This operation is equivalent to accessing an element in an array: \(\Theta(1)\).

• In the adjacency list, the requires looking up the list (\(\Theta(1)\)) and then checking whether \(v\) is on that list. If \(d\) is the outdegree of vertex \(u\), this will take \(\Theta(d)\) in the worst case. The overall worst case scenario is thus \(\Theta(d)\).

• The matrix wins!
What is the outdegree of vertex $u$?

• In the adjacency matrix, we need to scan row $u$, and count the number of 1's we find. This operation is $\Theta(n)$.

• In the adjacency list, this requires looking up the size of the sequence $L_u$, which takes $\Theta(1)$ (if the sequence is properly implemented).

• The list wins!

• Note: This problem is equivalent to finding all the out-neighbours of $u$. 
What is the indegree of vertex \( u \)?

- In the adjacency matrix: Essentially the same operation as for the outdegree, except that we scan a column for \( u \) rather than a row: \( \Theta(n) \).

- In the adjacency list, this requires checking for all \( v \neq u \) whether \( u \) is in the sequence \( L_v \). which takes \( \Theta(n + |E|) \). Note: need to check each \( L_v \) even if the sequence is empty.

- The matrix wins (at least for dense (di)graphs)!
Adding an arc from $u$ to $v$

• For the adjacency matrix: change value of matrix element $(u,v)$. This is $\Theta(1)$.

• For the adjacency list: Insert $v$ into list $u$. This is also $\Theta(1)$.

• Matrix and list perform equally well here.
Deleting an arc from $u$ to $v$

- This problem is pretty much identical to the that of answering the question whether the arc exists.

- In the adjacency matrix, rather than returning the value of $(u, v)$, we set it to 0 (false). So we have $\Theta(1)$.

- In the adjacency list, we delete the appropriate list entry, and operation of constant time once we have located the entry in the list. $\Theta(d)$.

- Matrix wins!
Adding a vertex to the (di)graph

• In the adjacency matrix: Need to add $2n+1$ entries (one row with $n$ entries at the bottom and then one column with $n+1$ entries at the end): $\Theta(n)$.

• In the adjacency list: Add one entry: $\Theta(1)$.

• List wins!
Deleting a vertex from the (di)graph

• Note that the vertex may have arcs attached!

• Matrix: Need to remove a row and a column, so need to shift up to \((n-1)\) elements up and up to \((n-1)\) elements to the left: \(\Theta(n^2)\).

• List: Need to look at all \(n\) entries and need to check their sequences for the presence of the node that needs to be removed. The total number of sequence entries is the number of arcs, \(m\), so the total time complexity here is \(\Theta(n+m)\).

• In sufficiently sparse graphs, the list wins, but as \(m \leq 2n(n-1)\) for digraphs and \(m \leq n(n-1)\) for graphs, this isn't necessarily always so!
Conclusion here

• Use adjacency matrix for small, dense (di)graphs for which we wish to test for the existence of arcs, find the indegree of vertices, and/or delete arcs.

• Use adjacency list for large, sparse (di)graphs for which we need to compute outdegree and add and/or delete vertices.
Lectures this week

• Lecture 1: Graph traversal – general approach, depth-first search (DFS), breadth-first search (BFS)

• Lecture 2: Priority-first search (PFS), directed acyclic graphs (DAGs), topological sorting

• Lecture 3: (Di)Graph connectivity, strongly connected components
Lecture 1

• Graph traversal – general approach, depth-first search (DFS), breadth-first search (BFS)
Graph traversal basics

• Common problem: Need to write a program to visit all vertices of a graph or digraph to perform some operation with them, and need to visit them following existing edges/arcs (i.e., we can't simply go row-by-row in an adjacency matrix or list).

• As far as possible, we want to avoid visiting vertices more than once

• Classical example: "Travelling Salesman Problem" (visit all cities in a country using the road network)

• Many algorithms exist to do this, but they all follow a common underlying algorithm, which we will discuss first.
Fundamental graph traversal algorithm

Start: "colour" all vertices "white".

Select one of the white vertices in the (di)graph and colour it grey.

Select one of the grey vertices in the (di)graph.

Does the vertex have a white (out-)neighbour?

Yes

Colour the (out-)neighbour grey.

No

Colour the vertex black

Are there any grey vertices left?

Yes

Are there any white vertices left?

No

No

Done!

Note: pseudocode for this algorithm is in the textbook in Figure 5.1. The inner loop (on the left) here corresponds to the visit algorithm, the outer loop (that surrounds this box) to traverse.
Notes on the fundamental algorithm

• The use of three "colours" means that we do not visit vertices twice: we always continue work at grey vertices first and only their white out-neighbours are coloured grey.

• Only once these grey vertices are "used up", we continue with the next white node to generate more. This allows us to deal with graphs that do not have a path from the starting vertex to each other vertex.

• Note that the fundamental algorithm says nothing about how the next grey or white node is chosen. This choice is made in actual implementations and is quite important!

• One typically implements colours as some sort of state property for a vertex object (e.g., with integer values). White: unvisited; grey: visited but outgoing arcs/edges not followed; black: visited and outgoing arcs/edges followed.

• Note that it is possible for a grey vertex to have only grey out-neighbours if these have been visited from other vertices!
Search forests

• Using the fundamental algorithm, the inner loop (which colours vertices black and its out-neighbours grey) implicitly generates a tree.

• In this tree, the out-neighbours that are coloured grey become the descendants of the vertex (node) that is coloured black, which is their ancestor in the tree.

• The outer loop (that colours a white node grey) corresponds to starting a new tree.

• This way, the algorithm produces a set of disjoint trees called a search forest.

• Note that different versions of the algorithm may produce different forests!
Arc, arc...

- In a search forest $F$ of a graph $G$, we can find four different kinds of arcs:
- Tree arc: an arc in $G$ that connects a vertex in $G$ to one of its immediate descendants in the tree of $F$ that the vertex belongs to, i.e., if the arc belongs to the tree
- Forward arc: an arc that does not belong to a tree in $F$ but that connects a vertex to one of its descendants in the tree
- Back arcs: an arc that does not belong to a tree in $F$ but that connects a vertex to one of its ancestors in the tree
- Cross arcs: arcs that fall into neither of the above categories
- Examples: will be provided for breadth-first and depth-first search soon
Depth-first search algorithm (DFS)

• DFS is a specific implementation of our fundamental graph traversal algorithm (also known as depth-first traversal)

• It specifies that we select the next grey vertex to pick as the youngest remaining grey vertex.

• The following slides are best viewed online as they trace the steps of the algorithm (the starting nodes for the outer loop of our traversal algorithm are always the nodes at the top, from left to right).
Depth-first traversal
Depth-first traversal
Depth-first traversal
Depth-first traversal
Depth-first traversal
Depth-first traversal
Depth-first traversal
Depth-first traversal
Depth-first traversal
Depth-first traversal
Depth-first traversal
Depth-first traversal
Depth-first traversal
Depth-first traversal
Depth-first traversal
Depth-first traversal
Depth-first traversal
Depth-first traversal
Depth-first traversal
Depth-first traversal
Depth-first traversal
Depth-first traversal
Depth-first traversal
Depth-first traversal
Depth-first traversal
Depth-first traversal
Depth-first traversal
Depth-first traversal
Depth-first traversal
Depth-first traversal
Depth-first traversal
Depth-first traversal
Depth-first traversal
Depth-first traversal
Depth-first traversal
Depth-first traversal
Depth-first traversal
Depth-first traversal
Depth-first traversal
Depth-first traversal
Depth-first traversal
Depth-first traversal
Depth-first traversal
Depth-first traversal
Depth-first traversal
Depth-first traversal
Depth-first traversal
Depth-first traversal
Depth-first traversal
Depth-first traversal – tree arcs
Depth-first traversal – back arc
Depth-first traversal – forward arcs
Depth-first traversal – cross arc
Breadth-first search algorithm (BFS)

• BFS is also a specific implementation of our fundamental graph traversal algorithm (and also known as breadth-first traversal)

• It specifies that we select the next grey vertex to pick as the oldest remaining grey vertex.

• The following slides are again best viewed online as they trace the steps of the algorithm (the starting nodes for the outer loop of our traversal algorithm are always the nodes at the top, from left to right).
Breadth-first traversal
Breadth-first traversal
Breadth-first traversal
Breadth-first traversal
Breadth-first traversal
Breadth-first traversal
Breadth-first traversal
Breadth-first traversal
Breadth-first traversal
Breadth-first traversal
Breadth-first traversal
Breadth-first traversal
Breadth-first traversal
Breadth-first traversal
Breadth-first traversal
Breadth-first traversal
Breadth-first traversal
Breadth-first traversal
Breadth-first traversal
Breadth-first traversal
Breadth-first traversal
Breadth-first traversal
Breadth-first traversal
Breadth-first traversal
Breadth-first traversal
Breadth-first traversal
Breadth-first traversal
Breadth-first traversal
Breadth-first traversal
Breadth-first traversal
Breadth-first traversal
Breadth-first traversal
Breadth-first traversal
Breadth-first traversal
Breadth-first traversal
Breadth-first traversal – tree arcs
Breadth-first traversal – back arc
Breadth-first traversal – cross arcs

Note: Breadth-first traversal does not produce forward arcs!
Depth-first traversal and stacks

• A depth-first traversal operates like a stack (last-in-first-out, LIFO)

• Vertices are pushed onto the stack when we colour them grey

• We always operate on the vertex that is on top of the stack

• If the vertex on top of the stack has an (out-)neighbour that is still white, we push that (out-)neighbour onto the stack (we colour it grey)

• If the vertex on top of the stack has no more white (out-)neighbours, we pop the vertex off the stack (we colour it black)

• Depth-first traversals are usually implemented via a recursive method/function call (see textbook). By the way: When you see a stack mentioned anywhere, the one word that should pop into your head is "recursive"...
Breadth-first traversal and queues

- A breadth-first traversal operates like a queue (first-in-first-out, FIFO)
- Vertices are added to the end of the queue when we colour them grey
- We always operate on the vertex that is at the front of the queue (the vertex that has been grey for the longest time)
- If the vertex at the front of the queue has an (out-)neighbour that is still white, we add that (out-)neighbour to the end of the queue (we colour it grey)
- If the vertex at the front of the queue has no more white (out-)neighbours, we take the vertex out of the queue (we colour it black)
Basic theorem on search forests

• For a (di)graph $G$, and vertices $u, v \in V(G)$:

• If $T_1$ and $T_2$ are two distinct trees in the search forest $F$ for $G$ for some algorithm that follows our basic traversal algorithm, and $T_1$ is traversed before $T_2$, then there are no arcs/edges from $T_1$ to $T_2$.

• If $u$ is visited before $v$ and there is a path from $u$ to $v$, then $u$ and $v$ belong to the same tree in $F$.

• If $u$ and $v$ belong to the same tree $T$ in $F$, then any path from $u$ to $v$ must have all its vertices in $T$. 
Basic theorems on DFS

• A (recursive) DFS from a vertex $v$ (inner loop visit algorithm) terminates only once all vertices reachable from $v$ via a path of white vertices have been visited.

• If $u$ and $v$ are in $F$ and $u$ is visited (coloured grey) before $v$, then

  a) if $u$ is an ancestor of $v$, $u$ is coloured black after $v$ is coloured black.

  b) if $u$ is not an ancestor of $v$, $u$ is coloured black before $v$ is coloured grey (or black, for that matter)
Basic theorems on BFS

• Theorems 5.6, 5.7 and 5.8

• When we visit the vertex $v$ at the front of the queue with distance $d(v)$ from the root:

  a) any tree arcs from $v$ lead to a white vertex with distance $d(v)+1$ from the root (for graphs: any tree edges lead to a vertex with distance $d(v)+1$ or distance $d(v)-1$).
  b) any back arcs from $v$ lead to a black vertex with distance less than $d(v)$ from the root
  c) any cross arcs either lead to a black vertex with distance up to $d(v)$ from the root, or to a grey vertex with distance $d(v)$ or $d(v)-1$.
  For graphs, the difference in distance that a cross edge bridges is 0, 1, or -1.
  d) In BFS, there are no forward arcs.
Time complexity of DFS and BFS

• With adjacency matrix: Need to find out-neighbours for \( n \) vertices. With \( \Theta(n) \) per vertex, this means \( \Theta(n^2) \).

• With adjacency list: \( \Theta(n+m) \).

• No difference between BFS and DFS here as time needed to choose the next grey/white node is constant
Application examples of DFS and BFS

• Listing all files in a directory and its subdirectories: if we explore a subdirectory as soon as we see it, it's DFS, if we first list all other files and subdirectories in the directory before we come back to explore the directory, it's BFS

• "Depth" and "breadth" are used because DFS and BFS are often used in conjunction with digraphs that are rooted trees (such as a directory structure on a computer)

• Often the decision between using DFS and BFS in these cases is based on when we expect to "find" what we have been looking for in our "search".
Lecture 2

• Priority-first search, DAGs, topological sorting
Priority-first search (PFS)

• Next grey vertex is selected according to a priority value

• Typically, this is an integer, such that the grey vertex with the lowest value is selected first

• Priority value is assigned (often computed) no later than when the vertex turns grey

• In simple PFS, the priority value of a vertex does not change. In advanced PFS, the priority value of a vertex may be updated again later.

• BFS and DFS are really just special cases of simple PFS. In BFS, the priority values are the order in which the vertices turn grey (1, 2, 3, ...). In DFS, the priority values are the negative order in which the vertices turn grey (-1, -2, -3, ...).
Time complexity of PFS

- General time complexity is not very good: Need to find minimum priority value each time we need to select the next grey node

- If we search up to $n$ vertices (say, in an array) for the minimum priority value, we need $\Omega(n^2)$

- This can be improved on a little by using a priority queue ADT (e.g., a heap) for the grey vertices, but it's still slower than pure DFS or BFS

- Use PFS when there is an advantage in processing high priority vertices first (e.g., when this allows us to remove other vertices or edges from the (di)graph to be traversed, thereby reducing the search space)
DAGS – Directed Acyclic Graphs

• What's the difference between the following two graphs?
DAGS – Directed Acyclic Graphs

• What's the difference between the following two graphs?
DAGS – Directed Acyclic Graphs

• What's the difference between the following two graphs?
DAGS – Directed Acyclic Graphs

• Digraphs can have cycles - such as the one on the left on the previous slide

• A digraph without a cycle such as the digraph on the right of the previous slides is called acyclic.

• We call such a graph a DAG – a directed acyclic graph
DAGS – Properties

• Any walk on a DAG $G$ is limited in length

• A DAG must have at least one source and one sink (why?)

• If $G$ is a DAG, then it is possible to find a topological order of the vertices. A topological order is a numbering of the vertices such that an arc $(u,v)$ in the digraph means that $u$ has a smaller number than $v$.

• Only DAGs have a topological ordering

• Just because a DAG has no cycles does not mean that the underlying graph has none. E.g., all the DAGs in the examples here have cycles in their underlying graphs!
Topological order – Example 1
Topological order – Example 2
Topological order – Example 3
Topological orders

• There is often more than one topological order for a digraph (see examples on previous slides)

• A topological order is also called a topological sort or a linear order.

• Why "linear" order? Because you can order the vertices in a line and make all arcs point the same way!
Topological order – to linear

These two digraphs are exactly the same!

In linear sort order, all arcs run from left to right!
Creating a topological order on a DAG

• Algorithm:

1. Initialise a counter

2. Find a source on the digraph (a vertex with outwards arcs only). If we cannot find a source, the digraph is not a DAG.

3. Assign the value of the counter to this source vertex.

4. Increment the counter

5. "Delete" the vertex and its outward arcs from the digraph and continue working with the remaining digraph. (Note: This cannot create a cycle)

6. If there are still vertices left, continue at step 2
Is it a DAG?

• A simple way to test whether a digraph is a DAG or not is to run a DFS on it.

• If DFS yields a back arc, then the digraph has a cycle and is not a DAG.

• DFS also gives us a topological order if we number the vertices in the opposite order in which they are being popped off the stack.
Topological order by DFS

• Starting at the node A:
  - push A, push C, push D,
  - push E, pop E, pop D,
  - push F, push G, pop G,
  - pop F, pop C, pop A,
  - push B, pop B

• So, we get B=1, A=2, C=3, F=4, G=5, D=6, E=7 (as in example 2)
Lecture 3

• Graph connectivity; strongly connected components
Connectivity

• A graph is **connected** if for each vertex $u \in V(G)$, there is a path to any other vertex $v \in V(G)$.
Connected components of a graph

• Any graph (note not digraph) $G$ consists of one or more subgraphs $G_i$ such that:
  – each $G_i$ is connected, and
  – if there is a path between $u, v \in V(G)$, then $u$ and $v$ are in the same subgraph $G_i$.

• The subgraphs $G_i$ are called the connected components of $G$
Connected components

Note: The connected components (indicated by the red ellipses) are the trees of the search forest we obtain if we run DFS or BFS on the graph (so, nothing new, really!)
Connectivity and digraphs

• Problem

There is no path from 1 and 4 to 2, 3, and 5.

Connected or not?

Answer: *weakly connected*
Connectivity and digraphs

• A digraph $G$ whose underlying graph is not connected is also not connected.

• A digraph $G$ whose underlying graph is connected but for which a pair of vertices $u, v$ exists such that there is no path from $u$ to $v$ is said to be weakly connected.

• A digraph $G$ for which each vertex $u$ has a path to each other vertex $v$ is said to be strongly connected.
Strongly connected components of digraphs

- The strongly connected components of a digraph $G$ are strongly connected subdigraphs, but their union does not yield $G$ (unlike for graphs).

- Rather, the union of the subsets of the vertices in the strongly connected subdigraphs forms $V$ (the set of all vertices in $G$).

- In other words: some arcs do not belong to any of the strongly connected subdigraphs, so they are missing from the union.

- This applies in particular to any arcs starting or terminating in sources or sinks, and any arcs that are responsible for weakly connecting different strongly connected components of $G$. 
Finding the strongly connected components in a digraph

• Trickier than for a graph

• Consider this: within a strongly connected component, there is a path from any vertex in the component to any outgoing arc that connects the component weakly to another component

• Similarly, from any incoming arc that weakly connects from another component, there is a path to any vertex in the strongly connected component – e.g., a DFS starting at a vertex in the strongly connected component would find all these vertices.

• So we can (conceptually) treat a strongly connected component like a single vertex
Treating strongly connected components like single vertices
Treating strongly connected components like single vertices
Finding the strongly connected components in a digraph

• Some preliminary observations

• Consider a digraph $G$ and its reverse digraph $G_r$. Then the strongly connected components are the same subsets of vertices in both digraphs.

• Now consider "shrinking" the connected components down to a single vertex in both digraphs. This gives us DAGs $H$ from $G$ and $H_r$ from $G_r$. 
Treating strongly connected components like single vertices
Treating strongly connected components like single vertices

$H_r$
Finding the strongly connected components in a digraph

• Run a DFS on $G_r$. Having separate trees in the resulting search forest you get means that there is no path from vertices in one tree to any of its successor trees (there may be back arcs but we'll get to that).

• As strongly connected components require such a path, the vertices of each strongly connected component must reside within the same tree. Different strongly connected components can reside in different trees, but a strongly connected component cannot overlap two trees.

• However, there could be more than one strongly connected component in a tree. How do we find out?
Finding the strongly connected components in a digraph

• Here's the trick: As you run the DFS on $G_r$, number the nodes in the order they turn black: i.e., in a graph with $n$ vertices, the first vertex to turn black is numbered 1, the next 2, and so on.

• Note that this makes the vertex numbered $n$ the root of the last tree of the search forest for $G_r$. The numbers in that tree are all higher than those in other trees of the search forest.

• More generally, the root in each tree of the search forest has the highest number in that tree, and the vertices in that tree have numbers between the number of the root and the number of the root of the previous tree in the search forest.
Finding the strongly connected components in a digraph

• Now we run DFS on $G$ by using the white node with the highest number as root whenever we start a new tree in the search forest for $G$.

• This means we start with the root of the last tree in the search forest of $G_r$.

• Because no tree in $G_r$ has an incoming arc from a predecessor tree, there is no arc from that tree in $G$ to the predecessor tree (remember that the arcs in both digraphs are the same, just in opposite directions!)

• So our DFS on $G$ will not leave the respective trees from the DFS on $G_r$.

• If the DFS on $G$ does not reach every vertex in the current tree from $G_r$, then the tree contains more than one strongly connected component.

• In this case, we again start the next tree with the white node with the highest number.
Step 1: DFS on $G_r$

Vertices numbered in order of completion (turning black)
Step 2: DFS on $G$

Trees in thick black with bold roots

First tree root: vertex 10
Second tree root: vertex 8
Third tree root: vertex 2
Algorithm postmortem

• Think about the DFS on $G_r$ as cutting the weak links in $G$ that run in forward direction

• The DFS on $G$ then cuts the weak links that run backwards

• Note: The description of the algorithm here differs a little from the textbook, but the algorithm itself is the same!
Lectures this week

• Lecture 4: Cycles, girth, graph colouring
• Lecture 5: Matching on bipartite graphs
• Lecture 6: Weighted digraphs and single-source shortest paths (Dijkstra’s algorithm)
Lecture 4

• Cycles, girth, graph colouring
Cycles and girth of a graph

- Recap: A **cycle** is a walk of length 3 or more on a graph (or length 2 or more on a digraph) where \( v_0 = v_\ell \), i.e., a walk that ends in the same vertex that it started in and that does not simply go down an undirected edge and comes back in the opposite direction.

- In a digraph, we talk about *directed cycles*.

- The length of the shortest cycle in a graph is called the graph's *girth*.

- NOTE: In the case of a digraph, the girth is determined based on the underlying graph, i.e., we do not require a walk on the digraph itself. We may refer to the length of the smallest directed cycle of a digraph as its *directed girth*. 
Examples: graphs and girth

Girth: 3
Directed girth: 3

Girth: 4
Directed girth: undefined

Girth: undefined
Directed girth: undefined

Girth: 3
Directed girth: 4
Finding the girth of a graph

- Simple algorithm based on BFS:
  - Perform BFS $|V|$ times, starting at each vertex $v \in V$ in turn.
  - If during a BFS search, we encounter a grey (rather than white) neighbour while exploring the edges that start at grey vertex $x$, we continue to the end of the current level (i.e., explore the remaining vertices at this distance from the starting vertex) and then stop.
  - For each edge $(x,y)$ at this level for which both $x$ and $y$ are grey and for which $v$ is the lowest common ancestor in the search tree, we compute the distances $d(x)$ and $d(y)$ to $v$.
  - The length of the cycle involving $x$, $y$ and $v$ is then $d(x) + d(y) + 1$
  - The minimum of these lengths at the level is the smallest cycle that involves $v$.
  - The smallest cycle among all possible start vertices $v$ is the girth.
Example: finding the smallest cycle involving vertex 1
Example: finding the smallest cycle involving vertex 1

Queue: 1
Example: finding the smallest cycle involving vertex 1

Queue: 1 2
Example: finding the smallest cycle involving vertex 1

Queue: 1 2 6
Example: finding the smallest cycle involving vertex 1

Queue: 2 6 8
Example: finding the smallest cycle involving vertex 1

Queue: 6 8 3
Example: finding the smallest cycle involving vertex 1

Queue: 6 8 3 5
Example: finding the smallest cycle involving vertex 1

Queue: 8 3 5 7
Example: finding the smallest cycle involving vertex 1

Queue: 8 3 5 7

7 is already grey!
Example: finding the smallest cycle involving vertex 1

Queue: 3 5 7 (stopped)
Example: finding the smallest cycle involving vertex 1

Queue: 3 5 7 (stopped)
Example: finding the smallest cycle involving vertex 1

Queue: 3 5 7 (stopped)
Example: finding the smallest cycle involving vertex 1

Queue: 3 5 7 (stopped)

Length of smallest cycle with vertex 1:
\[ d(x) + d(y) + 1 = 4 \]
$k$-colourings

• A graph $G$ is said to have a $k$-colouring if its set of vertices $V(G)$ can be divided into $k$ disjoint subsets such that every edge in $E(G)$ connects two vertices from different subsets.

• This is a problem that turns up e.g., for map makers (cartographers): colour a map of countries with $k$ colours such that no two adjacent countries have the same colour.

• If a graph has a $k$-colouring, then it also has a $(k+1)$-colouring. The reverse does not apply!
This graph has a 4-colouring

... and a 3-colouring...

... but no 2-colouring!

$k$-colourings
Bi-partite graphs

- Bi-partite graphs are graphs that have a 2-colouring

- This is equivalent to saying that the graph has no odd-length cycles: every edge crosses from one of the colour to the other, and to get back to a vertex of the same colour, we need a multiple number of switches to the other colour and back.
Example: bipartite graphs

bipartite

not bipartite

bipartite

not bipartite
Example: bipartite graphs

bipartite graphs

bipartite

not bipartite

bipartite

not bipartite
Lecture 5

• Matchings on bipartite graphs
Matchings

- A **matching** $M$ is a set of pairwise non-adjacent edges in a graph:
Matchings

• A matching is a set of pairwise non-adjacent edges in a graph:
Matchings

• A matching is a set of pairwise non-adjacent edges in a graph:
Matchings

• A matching is a set of pairwise non-adjacent edges in a graph:
Matchings

• A matching is a set of pairwise non-adjacent edges in a graph:
Why matchings are important

• Matchings are used whenever we need to assign members of a set to each other as exclusive pairs based on suitability criteria

• The members of the set are the vertices and the edges indicate potentially suitable pairings

• Matchings are often but not always used in conjunction with bipartite graphs
A small car ferry across a river can take two vehicles at a time with a combined weight of 4,000 kg. On one side, the following cars are waiting for a trip across:

- **BMW** 2,300 kg
- **Holden** 1,850 kg
- **Fiat** 1,650 kg
- **Jeep** 2,400 kg
- **Mazda** 1,400 kg
- **Toyota** 2,200 kg
Matchings example: Car ferry

- Edges indicate cars that can be put on the ferry together:
Matchings example: Car ferry

- A matching indicates a possible way in which to get the cars across: in this case, we require four ferry trips: two with a pair of cars and two with a single car each.
Matchings example: Car ferry

- In this matching, we can get all cars across in three ferry trips:
Matchings example: Car ferry

• Our first matchings example matched only two pairs of cars. Yet it is a *maximal matching*, because it is not a subset of any other matching.

• Our second example is a *maximum matching*, because we cannot find a matching with more edges (in this case, because there are only six vertices, but there are other reasons why this may happen as well)

• So, a maximum matching is always a maximal matching, but a maximal matching is not necessarily a maximum matching.

• Note: A maximum matching in an arbitrary graph G can be found in polynomial time. For a polynomial time algorithm see section 5.9 in the textbook.
Notes on finding maximum matchings

• A maximum matching in an arbitrary graph \( G \) can be found in polynomial time
Graph diameter

• The **diameter** of a graph or strongly connected digraph $G$ is the length (in number of edges) of the longest shortest path in $G$.

• This is equivalent to the maximum distance $d(u,v)$ between any two vertices $u$ and $v$ in the (di)graph.
Lecture 6

• Weighted digraphs and single-source shortest paths (Dijkstra's algorithm)
Weighted (di)graphs

- In many applications, arcs (or edges) in (di)graphs carry a cost. In this case, we talk of a **weighted** (di)graph and the cost of an edge is also called its **weight**. (The term **distance** is also sometimes used, in which case its is usually distinguished from our existing definition of distance we by referring to the latter as "distance in hops" or "distance in edges/arcs")

- Example: distances, travel times or fares in a transport network, usage charges or latencies (signal travel times) in a communication network

- We can model this by means of a cost function \( c(x,y) \) that returns the cost of the arc (edge) from vertex \( x \) to vertex \( y \).
Example: Weighted graph

Source of distances: Google Maps
A note on the weight function $c(x,y)$

• Strictly speaking, the weight function $c(x,y)$ is only defined if there is actually an edge (or arc) from $x$ to $y$.

• However, for practical reasons, it's often necessary to attach a value to $c(x,y)$ in implementations even when there is no edge or arc from $x$ to $y$.

• In this case, we usually assign weight of $+\infty$.

• Can store weights either in adjacency matrix format (distance matrix) or in adjacency list format.

• In the case of an adjacency matrix, we use the weights instead of the 1's and 0's (although we may sometimes – confusingly – use 0 to indicate that a weight is infinite, i.e., that there is no edge/arc).

• In the case of an adjacency list, each storage of an out-neighbour is paired with the weight to that out-neighbour.
## Distance matrix representation

<table>
<thead>
<tr>
<th></th>
<th>Auckland</th>
<th>Gisborne</th>
<th>Hamilton</th>
<th>Napier</th>
<th>New Plymouth</th>
<th>Palmerston North</th>
<th>Rotorua</th>
<th>Taupo</th>
<th>Wellington</th>
<th>Whanganui</th>
</tr>
</thead>
<tbody>
<tr>
<td>Auckland</td>
<td>0</td>
<td>∞</td>
<td>127</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
</tr>
<tr>
<td>Gisborne</td>
<td>∞</td>
<td>0</td>
<td>378</td>
<td>215</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
</tr>
<tr>
<td>Hamilton</td>
<td>127</td>
<td>378</td>
<td>0</td>
<td>∞</td>
<td>239</td>
<td>387</td>
<td>106</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
</tr>
<tr>
<td>Napier</td>
<td>∞</td>
<td>215</td>
<td>∞</td>
<td>0</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>141</td>
<td>319</td>
<td>∞</td>
</tr>
<tr>
<td>New Plymouth</td>
<td>∞</td>
<td>∞</td>
<td>239</td>
<td>∞</td>
<td>0</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>159</td>
</tr>
<tr>
<td>Palmerston North</td>
<td>∞</td>
<td>∞</td>
<td>387</td>
<td>∞</td>
<td>∞</td>
<td>0</td>
<td>242</td>
<td>145</td>
<td>73</td>
<td>∞</td>
</tr>
<tr>
<td>Rotorua</td>
<td>∞</td>
<td>∞</td>
<td>106</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>0</td>
<td>81</td>
<td>∞</td>
<td>∞</td>
</tr>
<tr>
<td>Taupo</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>141</td>
<td>∞</td>
<td>242</td>
<td>81</td>
<td>0</td>
<td>∞</td>
<td>∞</td>
</tr>
<tr>
<td>Wellington</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>319</td>
<td>∞</td>
<td>145</td>
<td>∞</td>
<td>∞</td>
<td>0</td>
</tr>
<tr>
<td>Whanganui</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>159</td>
<td>73</td>
<td>∞</td>
<td>∞</td>
<td>208</td>
</tr>
</tbody>
</table>
Adjacency list representation

Auckland: Hamilton, 127
Gisborne: Hamilton, 378, Napier, 215
Hamilton: Auckland, 127, Gisborne 378, New Plymouth, 239, Palmerston North, 387, Rotorua, 106
Napier: Gisborne, 215, Taupo, 141, Wellington, 319
New Plymouth: Hamilton, 239, Whanganui, 159
Palmerston North: Hamilton, 387, Taupo, 242, Whanganui, 73, Wellington, 145
Rotorua: Hamilton, 106, Taupo, 81
Taupo: Napier, 141, Palmerston North, 242, Rotorua, 81
Wellington: Napier, 319, Palmerston North, 145, Whanganui, 208
Whanganui: New Plymouth, 159, Palmerston North, 73, Wellington, 208

-> basically, we need the same number of extra storage spaces as there are objects, so the fundamental complexity does not change!
The negative weight issue

• The vast majority of weighted graph problems that occur in practice have no negative weights attached (i.e., following an edge/arc costs rather than gains us something)

• However, exceptions exist where the weight of an edge/arc can be negative.

• E.g., consider going on a holiday tripping around the world. Between some cities, you may need to buy a train/ferry ticket or an airfare (=cost, positive weight), but between others you might actually have the opportunity to earn money by working as crew on a ship, train, or plane (=gain, negative weight).
Classical algorithms on weighted graphs

• Dijkstra (pronounced "Dyke‐stra"): Used to find the cost to each destination vertex from a single source ("single source shortest path" - SSSP). Cannot handle negative weights.

• Bellman‐Ford: solves SSSP as well, slower than Dijkstra but can handle negative weights

• Floyd: solves all‐pairs shortest path (APSP) problem – minimal cost between any given pair of vertices
Dijkstra's algorithm

Start: "colour" all vertices "white" and colour the source vertex "black". Set the distance cost $d(s,x)$ of all vertices to the respective $c(s,x)$.

Are there any white vertices left?

Yes

Loop through all white vertices $x$ and set their $d(s,x)$ to the lower of $d(s,x)$ and $d(s,u) + c(u,x)$.

No

Select the white vertex $u$ with the lowest $d(s,x)$ and colour it black.

Done!
Dijkstra at work

Source of distances: Google Maps
Dijkstra at work

Source of distances: Google Maps
Dijkstra at work

Source of distances: Google Maps
Dijkstra at work

Source of distances: Google Maps
Dijkstra at work

Source of distances: Google Maps
Dijkstra at work

Source of distances: Google Maps
Dijkstra at work

Source of distances: Google Maps
Dijkstra at work

Source of distances: Google Maps
Dijkstra at work

Auckland

659 km

Wellington

Taupo

Rotorua

Whanganui

Palmerston North

Napier

New Plymouth

Gisborne

127 km

378 km

106 km

338 km

73 km

242 km

145 km

319 km

208 km

159 km

127 km

366 km

525 km

366 km

514 km

455 km

215 km

Source of distances: Google Maps
Notes on Dijkstra

- Note that the only vertices that have their total cost updated in each iteration of the loop are those that are adjacent to the vertex that just turned black.

- Black vertices have the final minimal total cost and are never updated.

- White vertices may have their total cost updated more than once.

- Update process and selection of white vertex with lowest total cost can also be implemented as a priority queue – in which case Dijkstra becomes a simple PFS: The white vertices that have a total cost of less than infinity can be regarded as grey, and the total cost is simply the priority key. See the Dijkstra2 algorithm in the textbook.
Dijkstra analysis: $\Theta(n^2)$

- **Start**: "colour" all vertices "white" and colour the source vertex "black". Set the distance cost $d(s,x)$ of all vertices to the respective $c(s,x)$.

- **Requirements**:
  - Loop runs $n-1$ times – one white vertex turns black in each iteration.
  - Requires an internal loop that runs $n$ times. Lookup of $c(s,x)$ is $\Theta(1)$ in adjacency matrix format or $\Theta(n)$ in adjacency list format. Total is $\Theta(n^2)$.

- **Loop**:
  - Loop through all white vertices $x$ and set their $d(s,x)$ to the lower of $d(s,x)$ and $d(s,u) + c(u,x)$.
  - Select the white vertex $u$ with the lowest $d(s,x)$ and colour it black.

- **Conclusion**:
  - These items both need loops that run $\Theta(n^2)$ times each across all iterations of the outer loop.
  - Done!
Dijkstra analysis II: $\Theta((n+m) \log n)$

- So what if we run Dijkstra using PFS with a priority queue?
- Presume we implement the queue using a binary heap.
- Then we need: $n$ operations to insert / delete a vertex with the minimum key to / from the queue, at a cost of $O(\log n)$ each: $O(n \log n)$
- We also need an update (decrease) of the priority value for up to $m$ edges, at a cost of $O(\log n)$ each: $O(m \log n)$
- Total cost in this case is thus $O((n+m) \log n)$. Do remember though that $m$ is $O(n^2)$. So this isn't necessarily an improvement unless the graph is sparse.
Lectures this week

• Lecture 7: Single-source shortest paths (Bellman-Ford algorithm)

• Lecture 8: All-pairs shortest paths (Floyd's algorithm)

• Lecture 9: Minimum spanning trees
Lecture 7

• Single-source shortest paths (Bellman-Ford algorithm)
Bellman-Ford

Start: Set the distance cost $d(s, x)$ of all vertices to $+\infty$, except for $d(s, s)$, which is set to 0. Set a counter $i$ to 0.

Is $i < n$?

Yes

Increment $i$

No

Loop through all edges $(x, y)$ and set $d(s, y)$ to the lower of $d(s, y)$ and $d(s, x) + c(x, y)$.

Done!
Bellman-Ford at work

Note: numbers in parentheses give one possible order in which edges are processed.

Source of distances: Google Maps
Bellman-Ford at work

Source of distances: Google Maps
Bellman-Ford at work

Source of distances: Google Maps

\[ i = 1 \]
Bellman-Ford at work

Source of distances: Google Maps
Bellman-Ford at work

Source of distances: Google Maps

i > 2

No change!
Bellman-Ford notes

• The order in which we iterate over the edges in the (inner) loop does matter: e.g., for $i=1$ we got distances for Taupo and Napier because the edges (Rotorua, Taupo) and (Gisborne, Napier) were processed after (Hamilton, Rotorua) and (Hamilton- Gisborne)

• There is no hard and fast rule on which order to choose, however

• Note that Bellman-Ford converged rather quickly to the result – after only 3 iterations out of a possible 9. Can stop processing if no distance has changed after an iteration.

• Note also that while Bellman-Ford can handle negative edge weights, it cannot handle negative weight cycles (which mean that there is no solution to the minimum weight path problem anyway!)

• If we can have negative edge weights, then we must also check after running the algorithm as to whether there are any such negative weight cycles.
Bellman-Ford analysis

Start: Set the distance cost $d(s,x)$ of all vertices to $+\infty$, except for $d(s,s)$, which is set to 0. Set a counter $i$ to 0.

Requirements:
- Requires a loop that runs $n$ times: $\Theta(n)$
- Outer loop runs $n$ times: $\Theta(n)$
- Inner loop runs $m$ times (adjacency list): $\Theta(m)$
  or $n^2$ times (adjacency/distance matrix): $\Theta(n^2)$

Loop through all edges $(x,y)$ and set $d(s,y)$ to the lower of $d(s,y)$ and $d(s,x) + c(x,y)$.

Increment $i$.

Is $i < n$?
- Yes
- Done!
- No

Requires a loop that runs $n$ times: $\Theta(n)$

Outer loop runs $n$ times: $\Theta(n)$

Inner loop runs $m$ times (adjacency list): $\Theta(m)$
  or $n^2$ times (adjacency/distance matrix): $\Theta(n^2)$
Bellman-Ford analysis

• For sparse graphs and adjacency lists: $\Theta(n)\Theta(m) = \Theta(nm)$

• For dense graphs we have $\Theta(m)$=O($n^2$), so Bellman-Ford is $\Theta(n)O(n^2) = O(n^3)$.

• With an adjacency matrix: $\Theta(n^3)$.

• Conclusion: Dijkstra is faster but doesn't give the right answers when we have negative weight edges/arcs
Lecture 8

• All-pairs shortest paths (Floyd’s algorithm)
All-Pairs-Shortest-Path

- Remember the distance matrix?

<table>
<thead>
<tr>
<th></th>
<th>Auckland</th>
<th>Gisborne</th>
<th>Hamilton</th>
<th>Napier</th>
<th>New Plymouth</th>
<th>Palmerston North</th>
<th>Rotorua</th>
<th>Taupo</th>
<th>Wellington</th>
<th>Whanganui</th>
</tr>
</thead>
<tbody>
<tr>
<td>Auckland</td>
<td>0</td>
<td>∞</td>
<td>127</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
</tr>
<tr>
<td>Gisborne</td>
<td>∞</td>
<td>0</td>
<td>378</td>
<td>215</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
</tr>
<tr>
<td>Hamilton</td>
<td>127</td>
<td>378</td>
<td>0</td>
<td>∞</td>
<td>239</td>
<td>387</td>
<td>106</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
</tr>
<tr>
<td>Napier</td>
<td>∞</td>
<td>215</td>
<td>∞</td>
<td>0</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>141</td>
<td>319</td>
<td>∞</td>
</tr>
<tr>
<td>New Plymouth</td>
<td>∞</td>
<td>∞</td>
<td>239</td>
<td>∞</td>
<td>0</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>159</td>
</tr>
<tr>
<td>Palmerston North</td>
<td>∞</td>
<td>∞</td>
<td>387</td>
<td>∞</td>
<td>∞</td>
<td>0</td>
<td>∞</td>
<td>242</td>
<td>145</td>
<td>73</td>
</tr>
<tr>
<td>Rotorua</td>
<td>∞</td>
<td>∞</td>
<td>106</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>0</td>
<td>81</td>
<td>∞</td>
<td>∞</td>
</tr>
<tr>
<td>Taupo</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>141</td>
<td>∞</td>
<td>242</td>
<td>81</td>
<td>0</td>
<td>∞</td>
<td>∞</td>
</tr>
<tr>
<td>Wellington</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>319</td>
<td>∞</td>
<td>145</td>
<td>∞</td>
<td>∞</td>
<td>0</td>
<td>208</td>
</tr>
<tr>
<td>Whanganui</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>159</td>
<td>73</td>
<td>∞</td>
<td>∞</td>
<td>208</td>
<td>0</td>
</tr>
</tbody>
</table>

- Our graph is clearly connected. So how can we produce a matrix that has the actual lowest costs rather than just ∞?
Options for All-Pairs-Shortest-Path

• Run Dijkstra's algorithm starting at each of the $n$ vertices: $\Theta(n)$ for iterating through the vertices, and $\Theta(n^2)$ for each Dijkstra run. Total: $\Theta(n^3)$.

• Use the Bellman-Ford algorithm $n$ times: $\Theta(n^2m)$ at best, $\Theta(n^4)$ at worst!

• Alternative: Use Floyd's algorithm
Floyd's algorithm

Start: Set the distance cost \( d(x,y) \) of all pairs of vertices \( x \) and \( y \) to \( +\infty \), except for \( d(s,s) \), which is set to 0.

1. Set a counter \( i = 0 \)
2. \( i < n? \)
   - Yes: Set a counter \( j = 0 \)
     - \( j < n? \)
       - Yes: Set a counter \( k = 0 \)
         - \( k < n? \)
           - Yes: Let \( x \) be the \( i \)'th, \( y \) the \( j \)'th and \( z \) the \( k \)'th vertex in \( V \). Set \( d(y,z) \) to the lower of \( d(y,z) \) and \( d(y,x) + d(x,z) \).
           - No: Increment \( k \)
         - No: Increment \( j \)
     - No: Increment \( j \)
   - No: Increment \( i \)

Done!
Notes on Floyd's algorithm

• Essentially just three nested loops: $\Theta(n^3)$. The outer loop specifies the potential "via" vertex, the inner two loops the end points of the path.

• Generally faster than a Bellman-Ford algorithm repeated $n$ times.

• Not necessarily faster than Dijkstra but handles negative edge weights correctly.

• See simulator under "Resources" on Cecil (a normal HTML file, download and open in a browser)
Lectures this week

• Lecture 5: Single-source shortest paths (Bellman-Ford algorithm)

• Lecture 6: All-pairs shortest paths (Floyd's algorithm)

• Lecture 7: Minimum spanning trees
Lecture 9

• Minimum spanning trees
Spanning trees

• A **spanning tree** of a graph $G$ is a connected acyclic graph $T$ that is a spanning subgraph of $G$, such that $V(T)=V(G)$ and $E(T)\subseteq E(G)$.

• That is, we can walk from any vertex of $G$ to any other vertex of $G$ along exactly one path whose edges are part of the tree $T$.

• Finding a spanning tree is easy: Run DFS or BFS from any of the vertices. If $G$ is connected, there will be only one search tree and this will be a spanning tree.
Spanning trees – yes or no?
Minimum spanning trees

- If $G$ is a weighted graph (i.e., we have a cost function $c(u,v)$ for the cost (weight) of edge $(u,v)$), we can compute the total weight of a spanning tree as the sum of the weights of its edges.

- A **minimum spanning tree (MST)** $T$ of $G$ is a spanning tree with the lowest possible weight among all spanning trees of $G$.

- Problem: The number of possible spanning trees can be huge, so how do we find a minimum spanning tree?

- Two algorithms: Prim's algorithm and Kruskal's algorithm
Note: Minimum spanning trees are not unique.
Prim's algorithm

• Prim's algorithm is essentially the same as Dijkstra's. We simply pick one of the vertices in $G$ as the source.

• The differences between Dijkstra's algorithm and Prim's are:
  – Instead of selecting the white vertex with the shortest distance to the source, we select the white vertex with the shortest distance to a black vertex (black vertices are part of the partial MST)
  – Instead of the distance to the source in each vertex, we record the shortest distance to a black vertex (we do that by “relaxing” the edges to the white neighbours each time we turn a vertex black)
  – We also store the search tree (by recording the edge through which we have last updated each vertex)

• Whenever we update the distance from the source to a vertex, we also update the information about the edge that we use to reach this vertex.
Prim's algorithm

Start: "colour" all vertices "white" and colour the source vertex $s$ "black". Set the distance cost $d(x)$ of all vertices to the respective $c(s,x)$. Record the parent of all vertices $x$ as $p(x)=$null, except for the neighbours of $s$, for which $p(x)=s$.

Are there any white vertices left?

Loop through all white neighbours $y$ of $u$ and set their $d(y)$ to the lower of $d(y)$ and $c(u,y)$. If you set it to $c(u,y)$, then also set $p(y)=u$.

Select the white vertex $u$ with the lowest $d(u)$ to a black vertex $x$ and colour it black.

The resulting minimum spanning tree is given by the black vertices $x$ and the edges $(p(x),x)$. 

Done!
Prim’s algorithm at work

Source of distances: Google Maps
Prim’s algorithm at work

Source of distances: Google Maps
Prim’s algorithm at work

Source of distances: Google Maps
Prim’s algorithm at work

Source of distances: Google Maps
Prim’s algorithm at work

Source of distances: Google Maps
Prim’s algorithm at work

Source of distances: Google Maps
Prim’s algorithm at work

Source of distances: Google Maps
Prim’s algorithm at work
Prim’s algorithm at work

Source of distances: Google Maps
Prim’s algorithm at work

Source of distances: Google Maps
Analysis: Prim's algorithm

- Prim's algorithm is essentially the same as Dijkstra's

- The extra update of the vertex property (parent / immediate predecessor of vertex in search tree) can be done in $O(1)$, the time complexity is the same as Dijkstra's: $O((m+n) \log n)$. 
Kruskal's algorithm

Sort edges in increasing order of weight (cost) such that $e_i = (u_i, v_i)$ is the $i$-th "cheapest" edge. Mark all edges as white. Set a counter $i=0$.

Does $e_i$ connect two subgraphs that are currently not connected to each other by black edges??

Yes

Colour $e_i$ black

No

Remove $e_i$

$i \leq m$?

Yes

$i \geq m$?

Yes

Done!

No

Increment $i$
Kruskal's algorithm
Kruskal's algorithm
Kruskal's algorithm

\[ i = 0 \]
Kruskal's algorithm

\[ i = 1 \]
Kruskal's algorithm

\[ i = 2 \]
Kruskal's algorithm

\[ i = 3 \]
Kruskal's algorithm

\[ i = 4 \]
Kruskal's algorithm

\[ i = 5 \]
Kruskal's algorithm

\[ i = 6 \]
Kruskal's algorithm

\[ i = 7 \]
Kruskal's algorithm

\[ i = 8 \]
Kruskal's algorithm

\[ i = 9 \]
Kruskal's algorithm

$i = 10$
Kruskal's algorithm

\[ i = 11 \]
Kruskal's algorithm

\[ i = 12 \]
Kruskal's algorithm

\[ i = 13 \]
Kruskal's algorithm

$i=14$
Kruskal's algorithm

\[ i = 15 \]
Kruskal's algorithm

\( i = 16 \)
Analysis: Kruskal's algorithm

- Time complexity is dominated by having to sort the edges by weight: $O(m \log n)$

- Kruskal's algorithm is a little harder to program but actually easier to do by hand
Lectures this week

• Lecture 10: Review of harder graph problems

• Lecture 11: Overflow
Lecture 10

• Review of some harder graph problems
Harder graph problems

• Actively researched area

• Perhaps you might be able to find a solution?

• General problem: complexity of known solutions isn't polynomial – meaning the time it takes to solve the problem will quickly outgrow available resources
Example 1

• Finding the longest path between two nodes in a digraph

• How would you tackle this problem?
Example 2

• Finding a $k$-colouring for $k > 2$

• This problem is often encountered by cartographers: How do you colour countries on a map in $k$ colours such that no two adjacent countries have the same colour?
Example 3

• The "Traveling Salesperson Problem" (also known as "Travelling Salesman Problem") - TSP

• Find a minimum-weight path that passes through all vertices in a graph

• Idea: Salesperson has to visit all cities in a country – what's the shortest route to do this by?

• One of the most famous problems in computing!
Example 4

• Finding a *Hamiltonian cycle* (a cycle that passes through all vertices of a graph)

• Similar to TSP but with different constraints: no minimum weight criterion but must return to origin & not visit a vertex twice
Example 5

• Finding the largest *independent set* or smallest *vertex cover* of a graph

• An independent set is a set of vertices such that no two vertices from the set share a common edge

• A vertex cover is a set of vertices such that each vertex in the graph is adjacent to a vertex from the cover
Approaches to hard graph problems

• May need to check all possible combinations

• Sometimes, we can discount some combinations in the search

• In some cases, it may be possible to *prove* that no polynomial solution exists
A quiet reminder...

• ...that your PeerWise 2 Assignment is due at 5 pm on June 7!
...and, finally:

All the best for your exam!