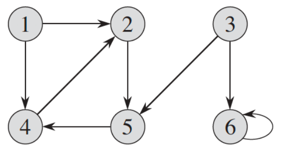


Graphs 2
Lecture 20

COSC 242 – Algorithms and Data Structures

Today’s outline

1. DFS, BFS, stacks, and queues
2. Topological sort
3. Path finding

Today’s outline

1. DFS, BFS, stacks, and queues
2. Topological sort
3. Path finding

DFS, BFS, stacks, and queues

BFS keeps track of the vertices in the graph using a <u>queue</u>. BFS adds new vertices (the children of the current vertices) to the back of a queue.

DFS on the other hand, uses a <u>stack</u>. In the previous algorithm the stack was implicit - we used the system stack, but we could just as well have used an explicit stack (*S* in the algorithm on the next page).

But, as can be seen on the following pages, using an explicit stack often results in more complicated code.

Recursive DFS

```
procedure DFS(G)
1:   for each vertex $u \in G.V$    // Initialise all vertices
2:       u.colour = WHITE
3:       u.π = NIL
4:   time = NIL                     // Initialise time (global variable)
5:   for each vertex $u \in G.V$
6:       if u.colour == WHITE
7:           DFS-Visit(G, u)        // Visit each white vertex in Graph

procedure DFS-Visit(G, u)
1:   time = time + 1                // Increment global time value
2:   u.d = time                     // Vertex discovered time
3:   u.colour = GREY
4:   for each $v \in G.Adj[u]$      // For each vertex v adjacent to vertex u
5:       if v.colour == WHITE       // if white, then it’s unexplored and we need to visit
6:               v.π = u
7:               $\boxed{\texttt{DFS-Visit(G, v)}}$     Recursive call
8:   u.colour = BLACK               // Explored all adjacent vertices, make self black
9:   time = time + 1                // Increment time (global)
10:  u.f = time                     // Set finish time for self
```

DFS with explicit stack

```
procedure DFS-Visit(G, u)
    time = time + 1;
    u.d = time
    u.colour = GREY
    Push(S, u)
    while not Empty(S)
        u = Top(S)                    // Get top element without removing from stack
        push = false;                 // Boolean tracks if we ’pushed
        for each $v \in G.Adj[u]$     // For each vertex v adjacent to vertex u
            if v.colour == WHITE
                v.$\pi$ = u
                push = true           // Pushed onto the stack
                break
        if push                       // Adjacent white vertices to be explored?
            v.colour = GREY; time = time + 1; v.d = time; Push(S, v)
        else                          // All adjacent vertices explored, colour self black
            u = Pop(S); u.colour = BLACK; time = time +1; u.f = time
end procedure
```

Today’s outline

1. DFS, BFS, stacks, and queues
2. Topological sort
3. Path finding

Application of DFS: Topological sort

A Directed Acyclic Graph, or DAG, is a directed graph with no cycles*.

A DAG is used to model a project's dependencies.

A topological sort turns a DAG into a horizontal list, where later items depend on earlier items.

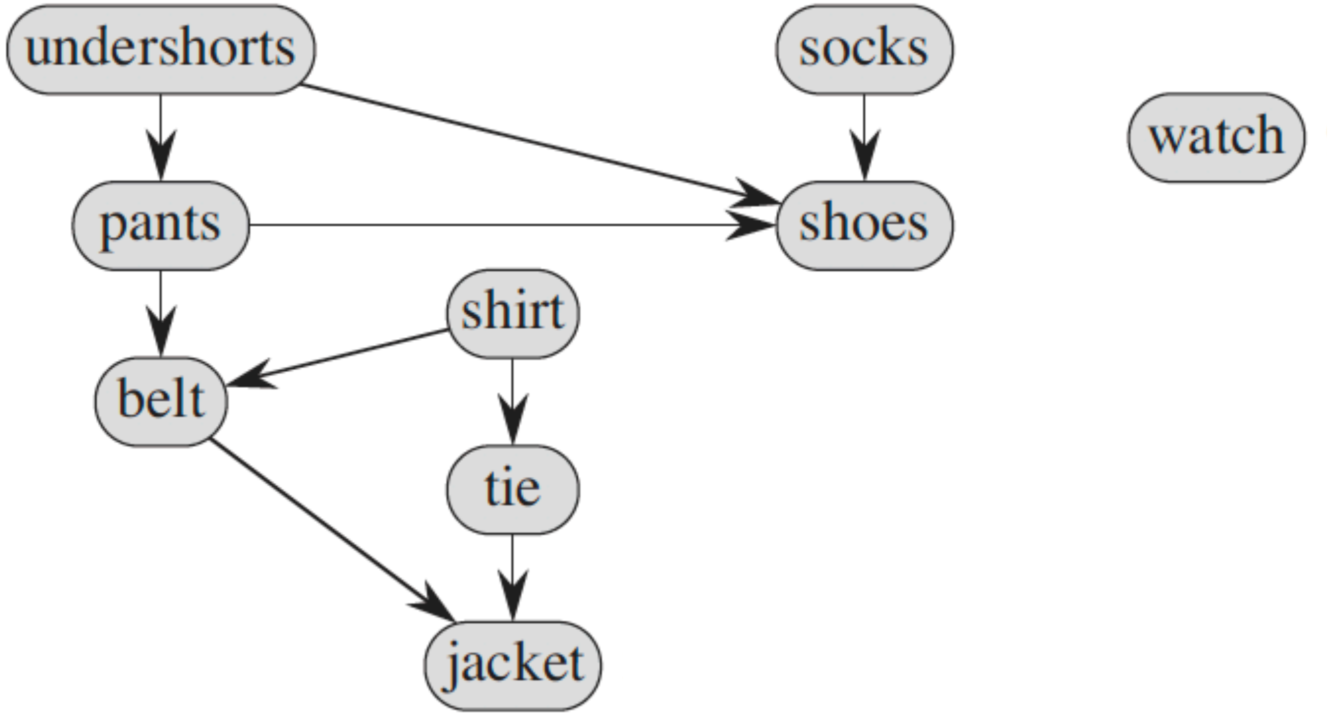
We can produce a topological sort using DFS – as each vertex is finished (coloured black and *u.f* is set), add it to the front of a list.

** A cycle is a non-empty path where the first and last vertices are the only repeated vertices.*

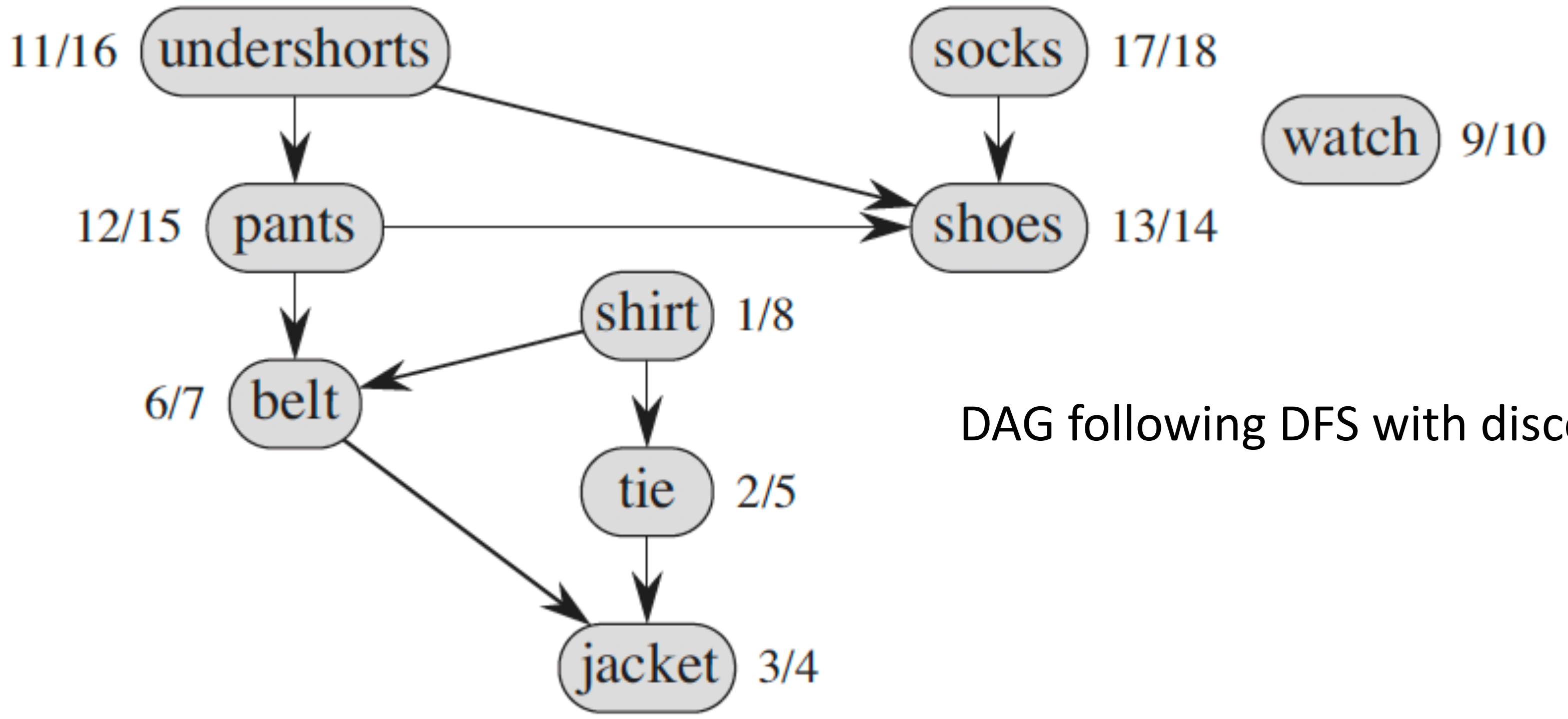
Topological sort

```
**procedure** Topological-Sort(G)
1:   call DFS(G) to compute finishing times v.f for each vertex v
2:   as each vertex is finished, insert onto front of a linked list
3:   return linked list
```

Example: Topological sort

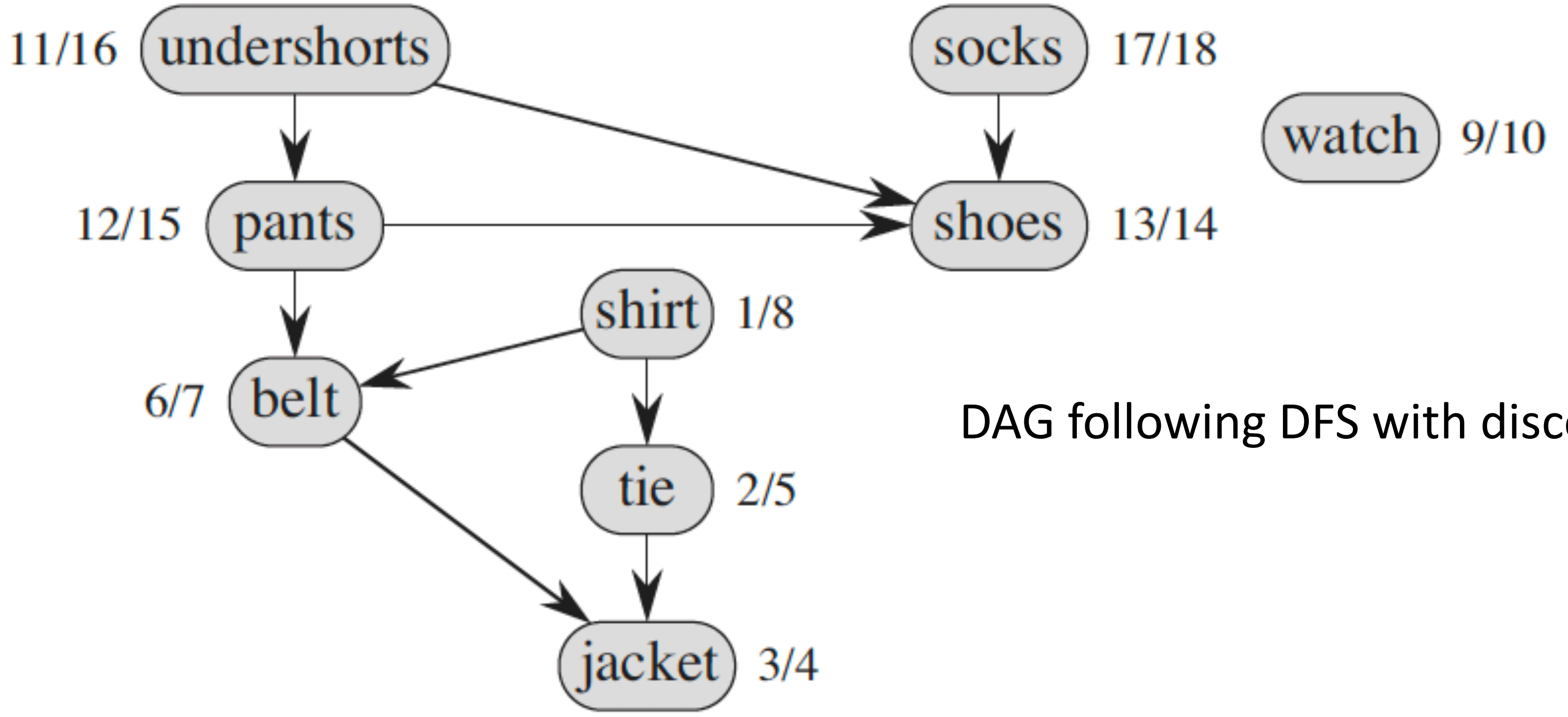


Example: Topological sort



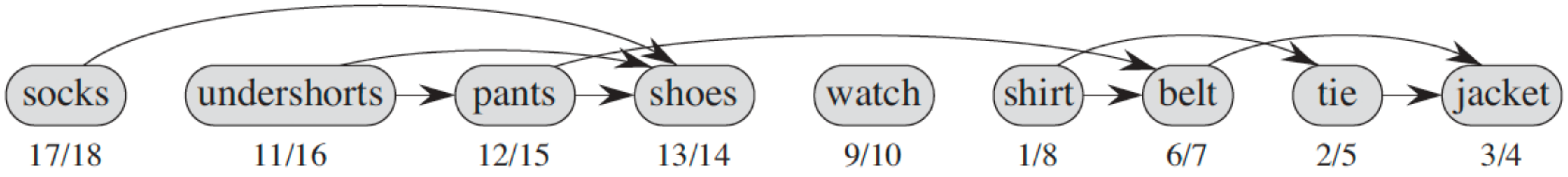
DAG following DFS with discover/finish times

Example: Topological sort

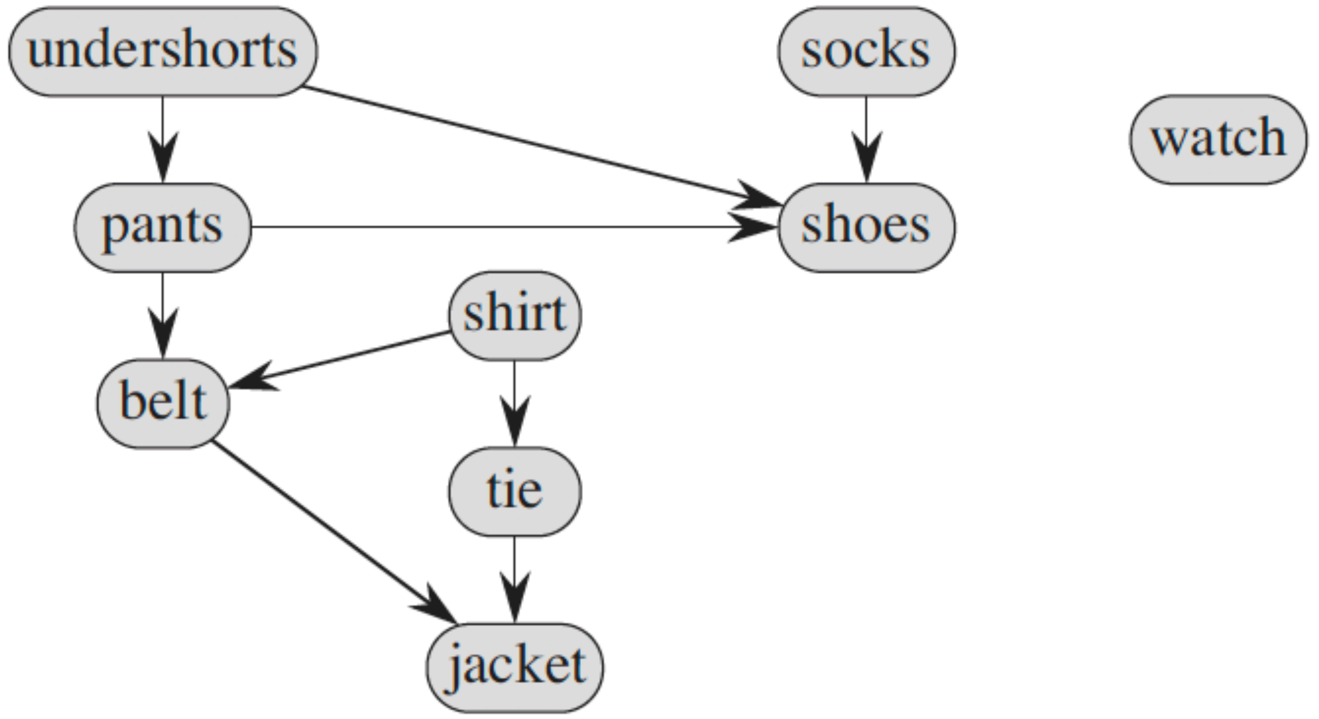


DAG following DFS with discover/finish times

List sorted by decreasing order of finish times. Edges retained.



Class challenge: Topological sort



Try it yourself. Start at any vertex. Your discover/finish times may be different, but you will arrive at the same correct dependency output.

Why does this work?

The key insight is to recognise what finish times tell us about a vertex. A vertex gets marked finished when it has no more adjacent vertices to explore.

Therefore, if we arrived at that vertex by way of an edge, it must be at the end of the dependency chain. These vertices will be marked with earlier finish times relative to their ancestors.

Similarly, a vertex with adjacent vertices will be finished later, as we need to explore these potentially long paths. These vertices will be marked with later finish times relative to their descendants.

Today’s outline

1. DFS, BFS, stacks, and queues
2. Topological sort
3. Path finding

Application of DFS: Path Finding

Maps are often represented as graphs. Every location on the graph is a vertex, and edges represent the fact that you can get from vertex a to vertex b in a single ”step”.

This is how Google Maps represents locations on the map. Google Maps probably has many billions of locations (perhaps trillions).

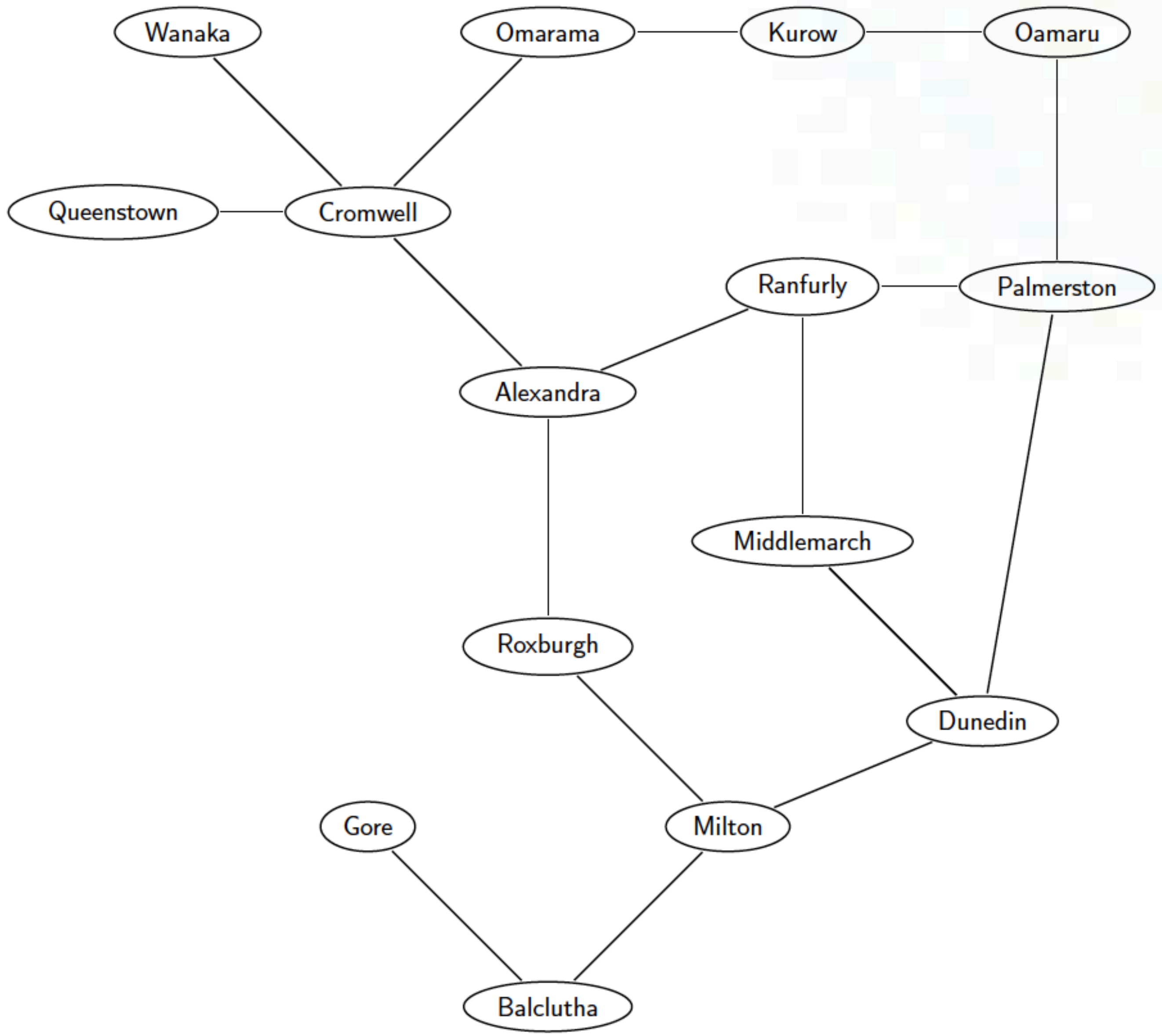
Application of DFS: Path Finding

You can use DFS to find a path between two locations:

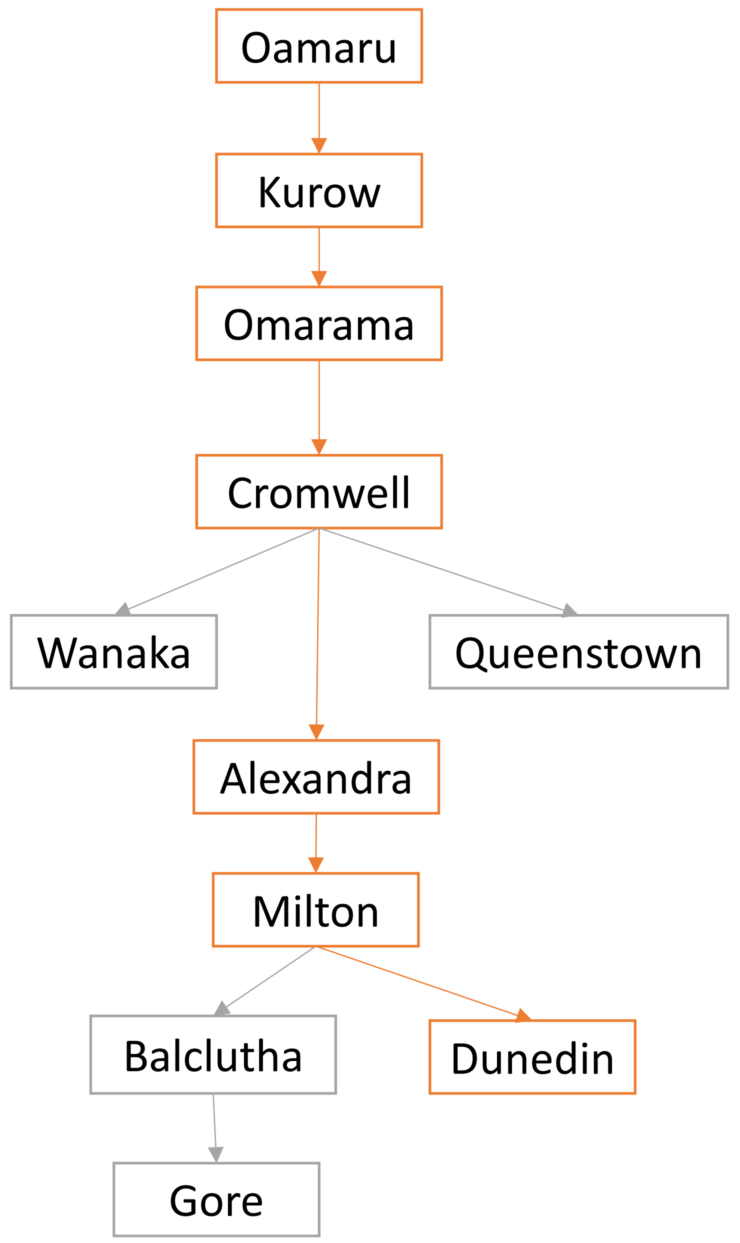
1. call DFS starting at the destination
2. build the DFS tree as you go,
3. halt if you reach the required source (which will be a leaf in the tree). Note, there may be no path.
4. then backtrack up the tree from the source to the leaf, and that gives a path between the two locations.

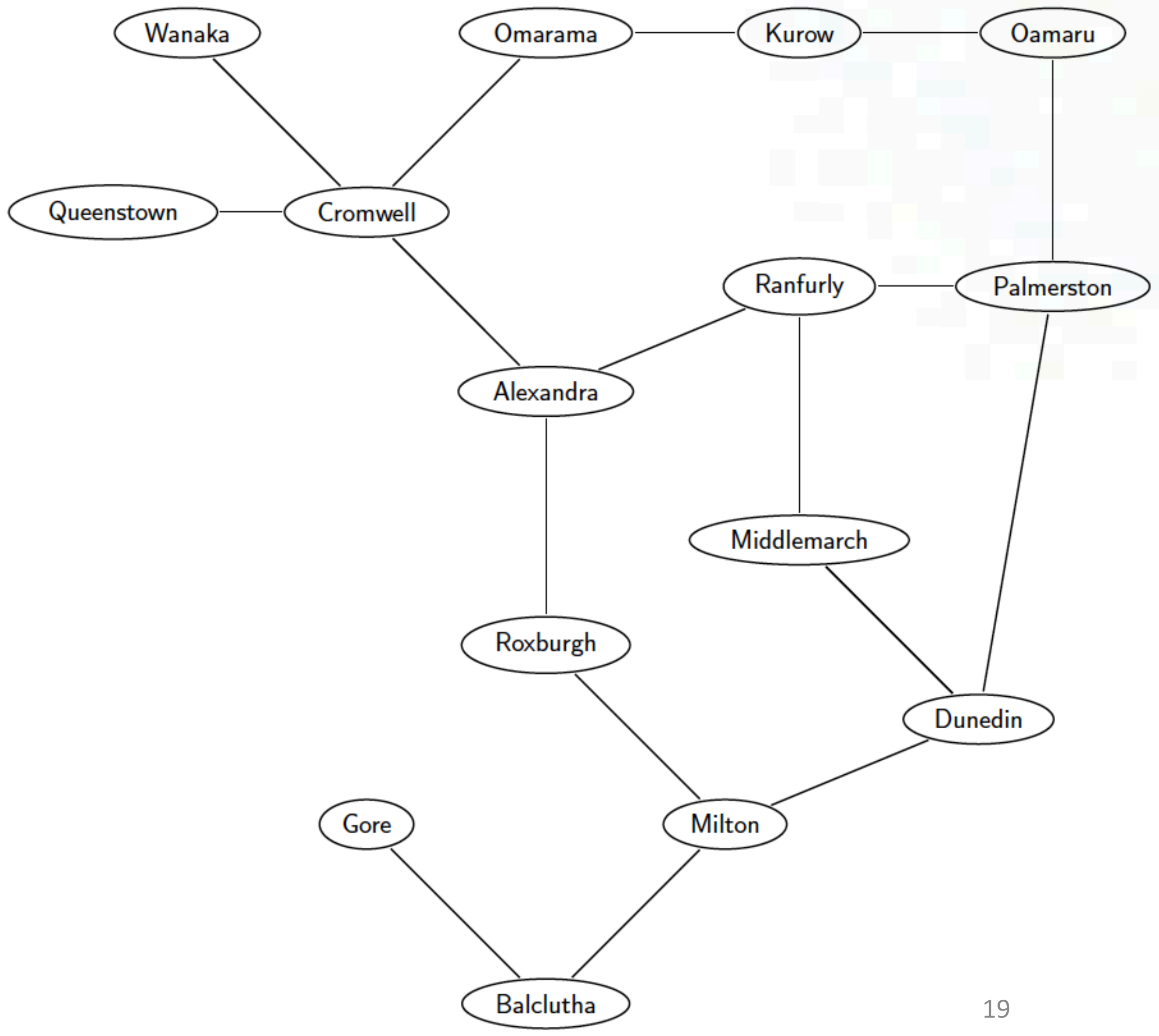
This path may not be the shortest one - we'll look at shortest path algorithms in a later lecture.

Path finding: Dunedin to Oamaru

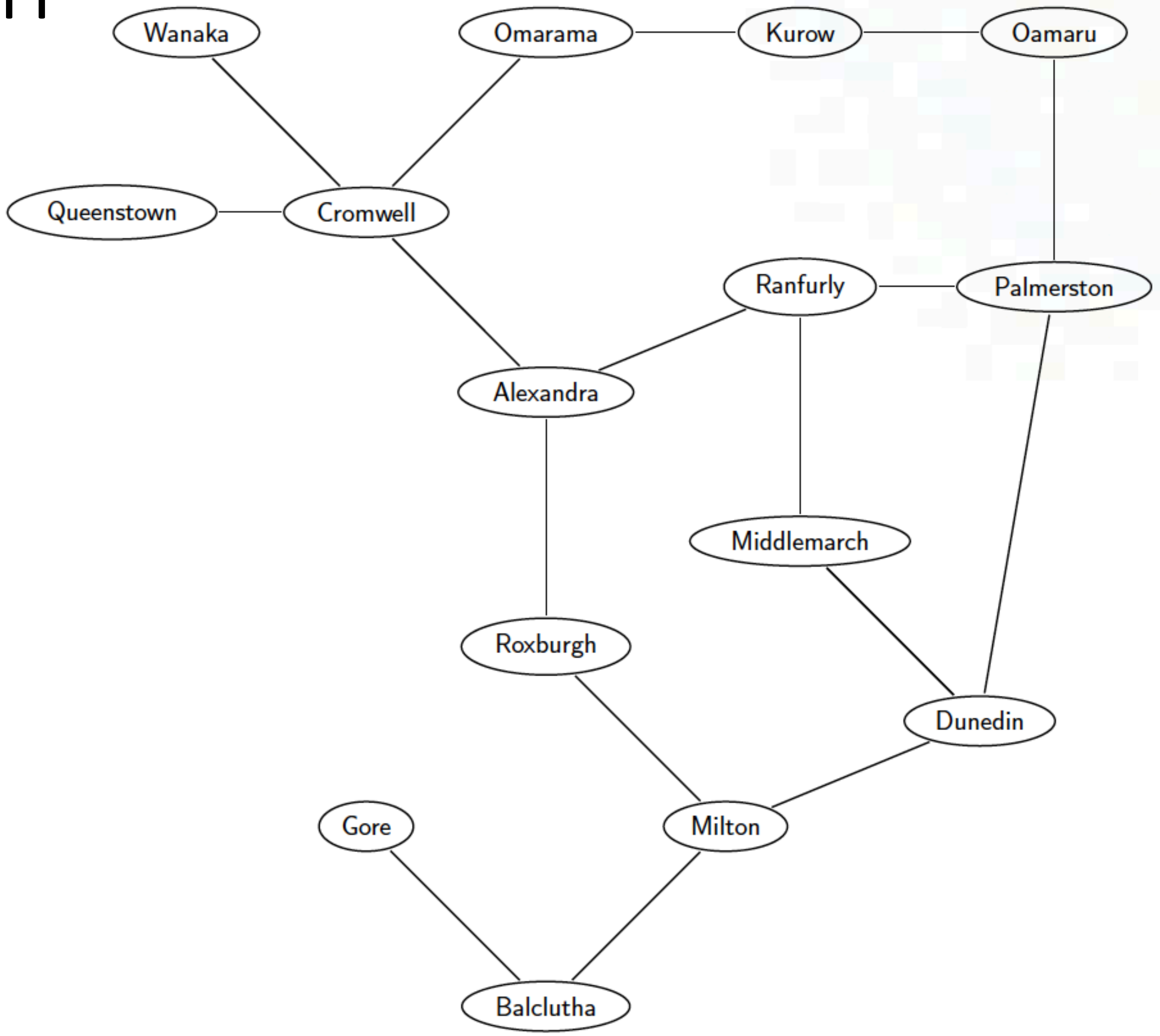


Path finding: Dunedin to Oamaru





Class challenge: Find path from Queenstown to Middlemarch

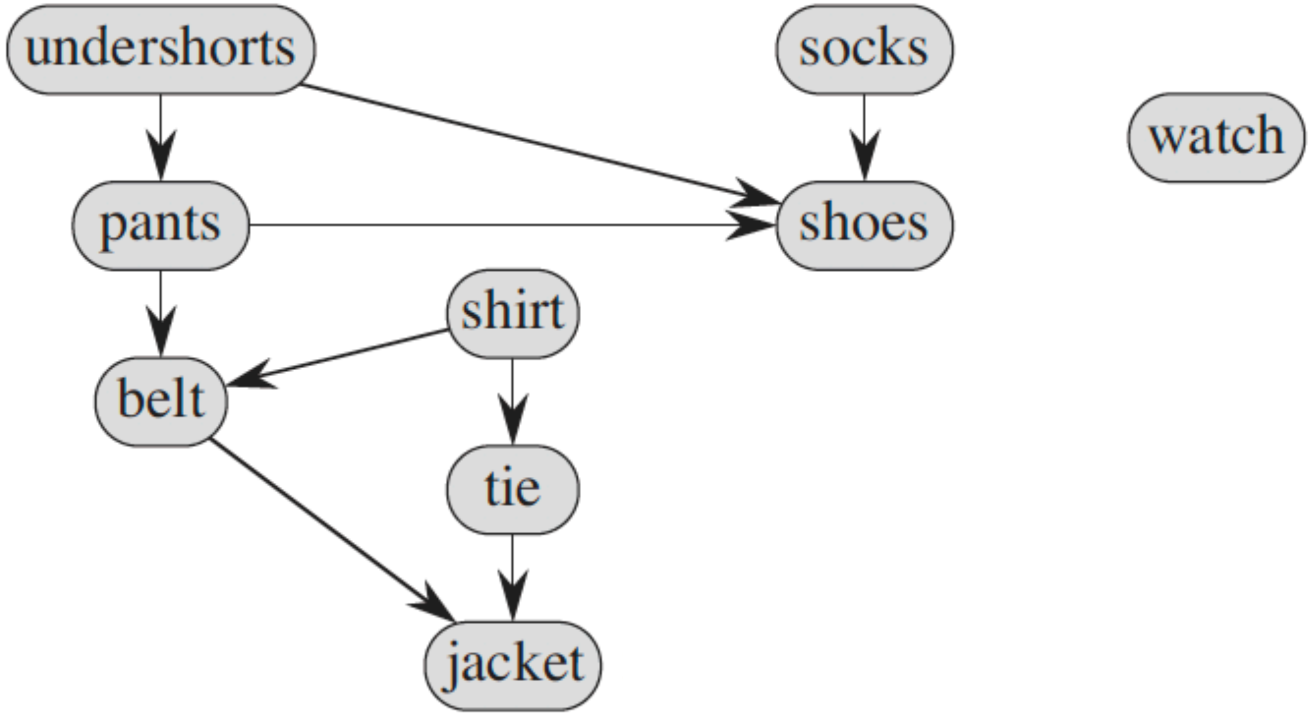


Suggested reading

Depth first search and topological search are discussed in sections 22.3 and 22.4 of the textbook, respectively.

Solutions

Class challenge: Topological sort



Try it yourself. Start at any vertex. Your discover/finish times may be different, but you will arrive at the same correct dependency output.

Image attributions