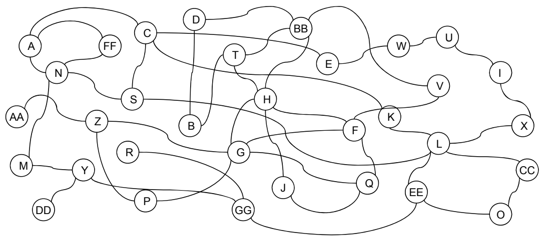
COMP261 Lecture 6

Dijkstra's Algorithm

Connectedness

- Is this graph connected or not?



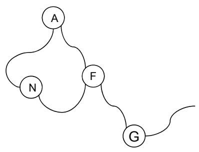
Connectedness: Recursive DFS

Traverse the graph from one node (eg, depth first search: DFS)
count nodes as you go

Check whether you got to all the nodes.

DFS of **trees** is easy, especially recursive version:
call DFS on the root node:

DFS (*node*):
count ++
for each *child* of *node*
DFS(*child*)



Graphs are more tricky
we might revisit a node and count it multiple times!
→ we need to record when we visit the first time and avoid revisiting!

Connectedness: Recursive DFS

Traverse the graph from one node (eg, depth first search),
mark nodes as you go
Check whether all the nodes have been marked.

DFS, recording when visited: (mark the nodes)

```
Initialise:  count ← 0,   for all nodes,  node.visited ← false
recDFS(start )
return  (count  = N)

recDFS (node ):
    if  not node.visited    then
        count ++,   node.visited ← true,
        for each  neighbour  of node
            if  not neighbour.visited
                recDFS(neighbour )
```

Connectedness: Recursive DFS

Traverse the graph from one node (eg, depth first search),
mark nodes as you go
Check whether all the nodes have been marked.

DFS, recording when visited: (explicit set of visited nodes)

```
Initialise:   count ← 0,   for all nodes,  visited ← { }
recDFS(start )
return  (count  = N)

recDFS (node ):
    if  node  not in visited   then
        count ++,  add node  to visited
        for each  neighbour   of node
            if  neighbour  not in visited
                recDFS(neighbour )
```

Connectedness: DFS

Using iteration and an explicit stack
Fringe is a stack of nodes that have been seen, but not visited.

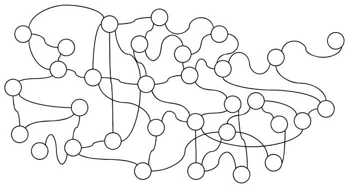
```
Initialise: count ← 0,   for all nodes,  node.visited ← false
push  start  onto fringe                          // fringe is a stack
repeat  until fringe  is empty:
    node  ← pop from fringe
    if  not node.visited  then
        node.visited  ← true,
        count ++
        for each  neighbour  of node
            if  not neighbour.visited
                push neighbour  onto fringe
return  (count = N)
```

A General Graph Search strategy:

- Start a "fringe" with one node
- Repeat:
  Choose a fringe node to visit; add its neighbours to fringe.

choice determines the algorithm and the result

- Stack: DFS
- Queue: ?



A General Graph Search strategy

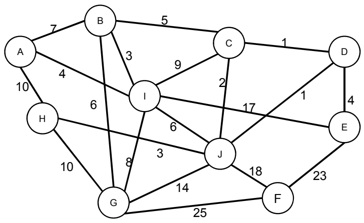
- Choices:
  - How is the fringe managed? stack? queue? priority queue?
    - If priority queue, what is the priority?
    - Can the fringe be pruned?
  - How are the neighbours determined?
  - When do you stop?
- DFS Connectedness:
  - fringe = stack.
  - neighbours = follow edges out of node
  - stop when all nodes visited.
- BFS connectedness:
  - fringe = queue.
  - neighbours = follow edges out of node
  - stop when all nodes visited.

Shortest paths problems

- Find the shortest path from start node to goal node
  "shortest path"
  - truncated Dijkstra's algorithm (Best-first search)
  - A*
- Find the shortest paths from start node to each other node
  "Single source shortest paths"
  - Dijkstra's algorithm
- Find the shortest paths between every pair of nodes
  "All pairs shortest paths"
  - Floyd-Warshall algorithm

Dijkstra's algorithm

- Idea: Grow the paths from the start node, always choosing the next node with the shortest path from the start



Dijkstra's algorithm

- Given: a graph with weighted edges.
- Initialise fringe to be a set containing start node
  start.pathlength ← 0
- Initialise path length of all other nodes to ∞
- Repeat until visited contains all nodes:
  - Choose an unvisited node from the fringe with minimum path length (ie, length from start to node)
  - Record the path to the current node
  - Add unvisited neighbours of current node to fringe
  - Add the current node to visited

More questions and design issues

- What to store for each node in the fringe
- How to store the paths (or find the paths)
- How to store the visited nodes
- How to represent the
  - graph
    - Node
    - Edge
  - Search node (element of fringe)

Refining Dijkstra's algorithm

- Given: a weighted graph of *N* nodes and a *start* node
  - nodes have an adjacency list of edges
  - nodes have a *visited* flag and *pathFrom* and *pathLength* fields
  - *fringe* is a priority queue of search nodes ⟨*length*, *node*, *from*⟩

Initialise: for all nodes *node.visited* ← false, *count* ← 0
  fringe.enqueue(⟨0, *start*, *null* ⟩),
Repeat until *fringe* is empty or *count* = N :
  ⟨*costToHere*, *node*, *from* ⟩ ← *fringe*.dequeue()
  If not *node.visited* **then**
    node.visited ← true, *node.pathFrom* ← *from*,
    count ++
    for each *edge* out of *node* to *neighbour*
      if not *neighbour.visited*
        costToNeighbour ← *costToHere* + *edge.weight*
        fringe.enqueue(⟨*costToNeighbour* , *neighbour*, *node* ⟩)

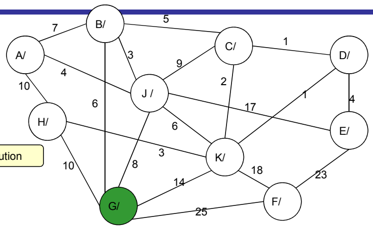
Illustrating Dijkstra's algorithm

nodes:
  label, visited,
  parent, pathlength

Solution

fringe nodes
  node,
  from,
  length

Candidate Solution



In	0																
nd	G																
fr	-																

Analysing Dijkstra's algorithm

- A node may be added to the fringe lots of times!
- What's the cost?
- How big could the fringe get?
- Finds shortest paths to every node

Note:
- All the paths are represented as a linked list from the node back to the start node
  - shared list structure.
- To print/follow path to a node, have to reverse the path.

Shortest path start → goal

- To find best path to a particular node:
  - Can use Djikstra's algorithm and stop when we get to the goal:

Initialise: for all nodes *visited* ← false, *count* ← 0
fringe.enqueue(⟨0, *start*, *null* ⟩),
Repeat until *fringe* is empty or *count* = N :
⟨*length, nd, from,* ⟩ ← *fringe*.dequeue()
If not *nd.visited* **then**
nd.visited ← true, *nd.pathFrom* ← *from*, *nd.pathLength* ← *length*
If *nd* = *goal* **then** exit
count ← *count* + 1
for each *edge* to *neighbour* out of *node*
if *neighbour.visited* = false
fringe.enqueue(⟨*length+edge.weight, neighbour, nd* ⟩)

Only check when dequeued.
Why?

Shortest path using Dijkstra's alg

- it explores lots of paths that aren't on the final path.
  ⇒ it is really doing a search
- dynamic programming
  - it never revisits nodes
  - it builds on optimal solutions to partial problems