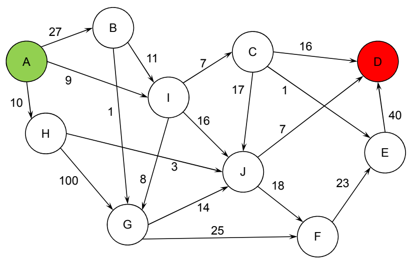
Network Flow
Longest Paths
Backtracking DFS

Victoria University of Wellington
Te Whare Wānanga o te Ūpoko o te Ika a Māui
CAPITAL CITY UNIVERSITY

Network Flow

- What is the maximum total flow/traffic/capacity possible from node A to node D?



Maximum Flow problem

- Given
  - a directed weighted graph
    ie, edges labeled with positive numbers = “capacity”
  - a source node and a sink node
- Find
  - a positive flow on each edge that maximises the total flow out of the source such that
    - flow on any edge is at most the capacity of the edge
    - net flow into any node = net flow out of the node
      except for source and sink nodes.
- Some solutions:
  - Ford–Fulkerson Algorithm
  - Edmonds–Karp algorithm (= Ford-Fulkerson but using BFS)
  - Dinitz blocking flow algorithm
  - Push-relabel maximum flow algorithm (and variations)

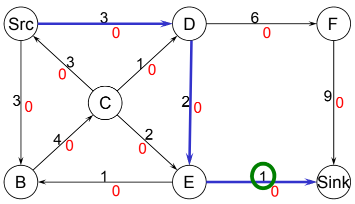
Edmonds–Karp / Ford-Fulkerson

- Key idea:
  - Each edge $u \rightarrow v$ has a capacity cap(u, v)
  - For every edge $u \rightarrow v$, where there is no edge $v \rightarrow u$ then add a phantom edge $v \rightarrow u$ with capacity 0
  - Add a flow (=0) for each edge (and phantom edge):
    - flow(u,v) = – flow(v,u)
    - for each edge (including all the phantom edges):
      - Remaining-capacity(edge) = capacity(edge) – flow(edge)
  - Repeatedly
    - find a path from source to sink with non-zero remaining capacity on each edge of the path
    - find minCap, the smallest remaining-capacity edge along the path
    - add minCap to the flow of each edge on the path

Example

- (From Wikipedia)

  black = capacity, red = flow

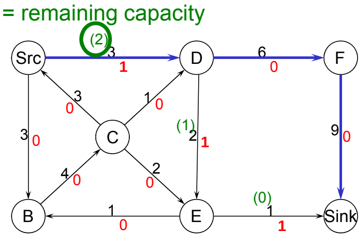


Example

- (From Wikipedia)

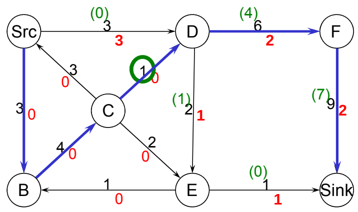
  black = capacity, red = flow,

  (green) = remaining capacity



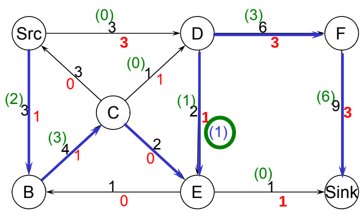
Example

- (From Wikipedia)
  black = capacity, red = flow,
  (green) = remaining capacity



Example

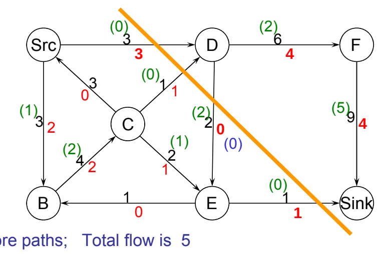
- (From Wikipedia)
  black = capacity, red = flow,
  (green) = remaining capacity
  (blue) = remaining capacity on phantom edge (cancels flow on real edge)



Flow on phantom edge can reverse a wrong decision earlier.

Example

- (From Wikipedia)
  black = capacity, red = flow,
  (green) = remaining capacity
  (blue) = remaining capacity on phantom edge (cancels flow on real edge)



No more paths; Total flow is 5

Minimum cut = capacity 5.

Edmonds-Karp Algorithm

```
flow[u,v] ← 0  for all nodes u, v
totalFlow  ← 0
repeat
    path ← BreadthFirstSearch(source, sink)
    pathFlow ← minRemainingCapacity(path)
    if pathFlow = 0   break
    totalFlow  +=  pathFlow
    for each  u – v in path
        flow[u, v]  +=  pathFlow
        flow[v, u]  –=  pathFlow
return totalFlow
```

Finds shortest path (counting # edges only)

Finds minimum remaining capacity of edges on the path

BreadthFirstSearch (flow)

```
for each  node,
    initialise parent(node)  ← null      // records  path and "visited"
queue.enqueue (source)
while queue is not empty
    node  ←  queue.dequeue
    for each  neighbour node
        if  parent(neighbour) = null && remainingCap(node, neighbour) > 0
            parent(neighbour)  ← node
            if  neighbour = sink   return  makePath(sink)
            else  queue.enqueue(neighbour)
return  null   //  failed to find a path.
```

remaining capacity on the edge from node to neighbour.

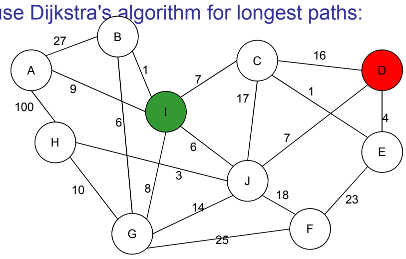
Constructs path back from sink to source, using parent links.

Network Flow Algorithms.

- Ford – Fulkerson
  - repeatedly find "augmenting" paths.
  - doesn't specify how to find the next path to add.
  - there are cases where it is very slow, or even doesn't terminate!
- Edmonds – Karp
  - = Ford – Fulkerson, but using Breadth First search to find next path
  - $O(NE^2)$ (#nodes x #edges^2)
- Push-relabel
  - pushes flow out from source, like fluid flow.
  - $O(N^2E)$
- Push-relabel with dynamic trees
  - $O(NE \log (N^2 / E))$
- Orlin's algorithm (2012)
  - $O(NE)$

Longest Paths: Backtracking

- Can't use Dijkstra's algorithm for longest paths:



- Need to look ahead to make right decision
  ⇒ need to backtrack

Longest Paths: Backtracking DFS

```
Initialise:   longestpath ← null, maxLength ← 0,   ∀ nodes: unvisit(node )
    visit(start )
    recDFS(start, 0, null)

    recDFS (node, pathLength, path):
        for each  edge  from node
            neighbour ← edge.other,  newLength ← pathLength +edge.length
            if neighbour = goal then
                if  newLength > maxLength  then
                    maxLength ← newLength
                    longestPath = path
            else if not visited(neighbour )  then
                visit(neighbour )
                recDFS(neighbour, newLength, append(neighbour, path))
                unvisit(neighbour )
```

Longest Paths: Backtracking

```
Initialise:    fringe ← new Stack / Queue / PriorityQueue
               longestpath ← null, maxLength ← 0,
push ⟨start, null, 0⟩ onto fringe

while  fringe  not empty
    ⟨node, path , pathLength⟩  ← pop fringe
    for each  edge  from node
        neighbour ← edge.other,
        newLength ← pathLength +edge.length
        if neighbour = goal then
            if  newLength > maxLength  then
                maxLength ← newLength
                longestPath = path
        else if  neighbour not on path  then
            push ⟨neighbour, cons(neighbour, path ) , newLength ⟩
                 onto fringe
```

Use path to avoid cycles

Options for General Graph Search

- Is the graph explicit or implicit
  - can be hard to record visited with implicit graphs
- keep track of paths or not
  - If only want to find a node, then don't record path
- visit only once or backtrack
  - visit only once only considers first path to a node
    - ⇒ search constructs a tree
  - backtrack considers all paths to a node
    - ⇒ search constructs a DAG (Directed acyclic Graph)
    - **Note**: if you don't care about paths, no point in backtracking
- DFS, BFS, PFS(priority first search)
  - If using PFS, what is the priority?
    - local: next edge, node value, estimate of "promise"
    - path cost: cost so far, or cost so far plus estimate to goal.