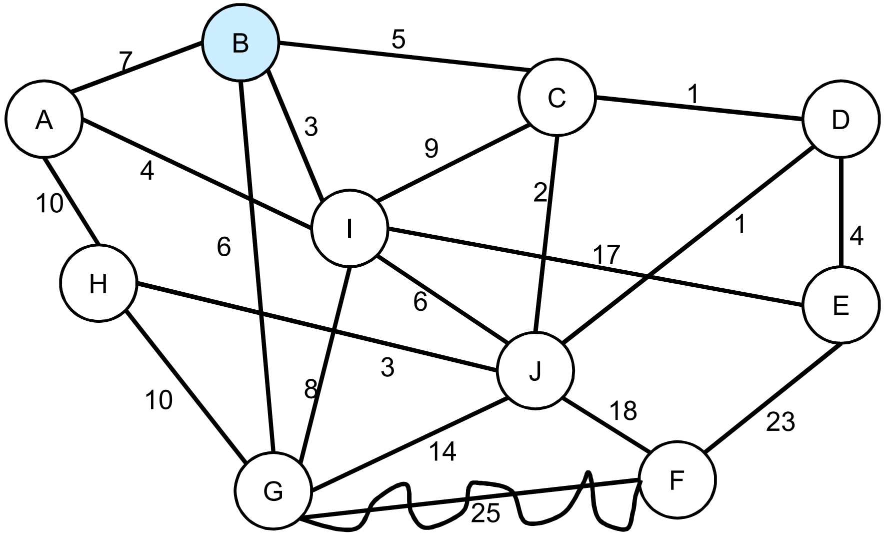
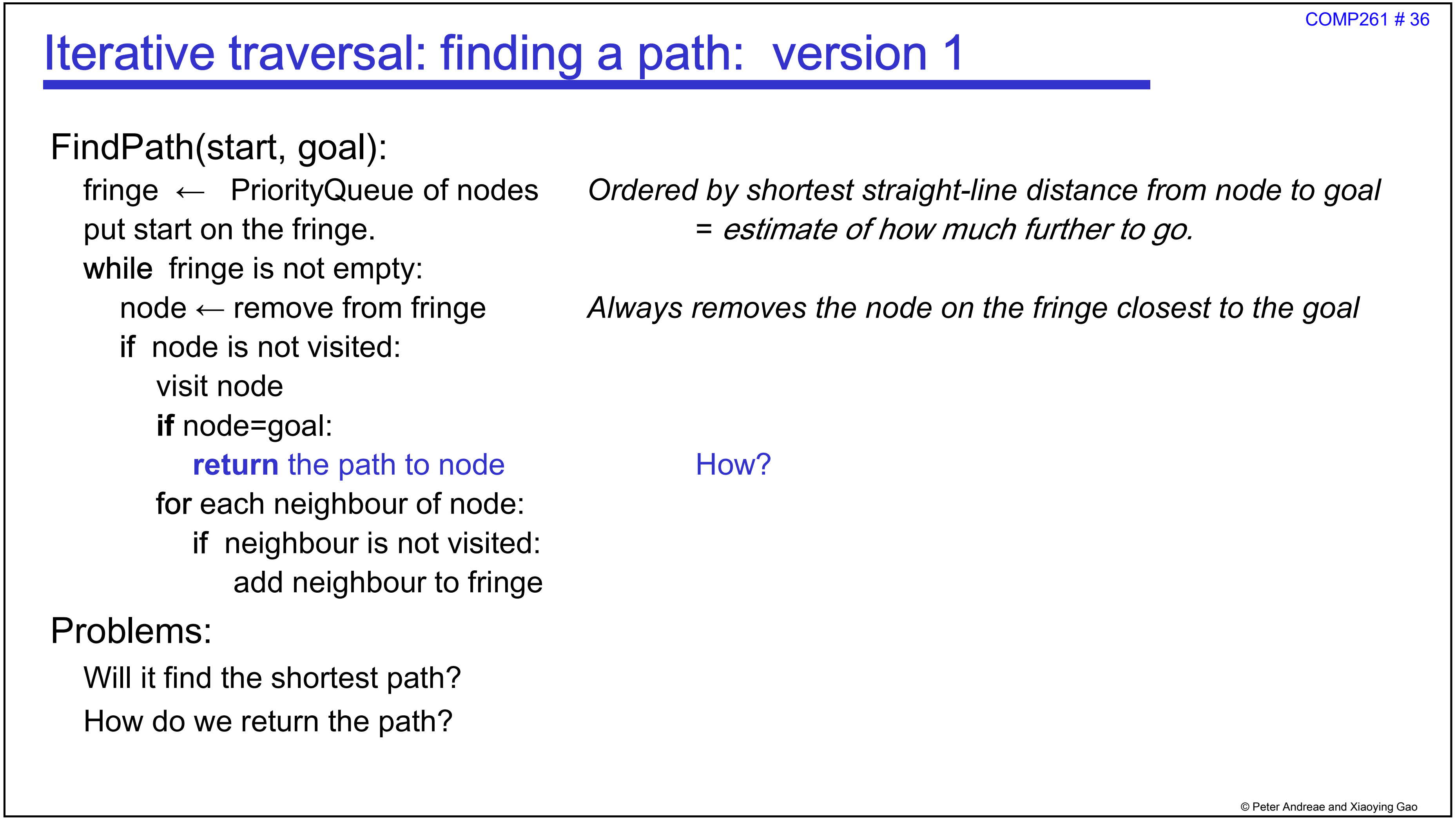
Admin

- Assign 2 is out
- Tutorial:
  - Graph representation
  - Construct Path by backpointers
  - Path finding
- Term test marking is finished, hopefully will be handed back tomorrow.

Finding a path:

- Suppose we want to find a path from start to a goal?
- Assume graph is of physical places,
  - each node has a location.
  - each edge has the actual path length
- Which order should we choose?
  - DFS?
  - BFS?
  - ??





Iterative traversal: finding a path: version 1

```
FindPath(start, goal):
   fringe ←  PriorityQueue of nodes        Ordered by shortest straight-line distance from node to goal
   put start on the fringe.                          = estimate of how much further to go.
   while  fringe is not empty:
      node ← remove from fringe            Always removes the node on the fringe closest to the goal
      if  node is not visited:
         visit node
         if node=goal:
            return the path to node                  How?
         for each neighbour of node:
            if  neighbour is not visited:
               add neighbour to fringe
```

Problems:

Will it find the shortest path?

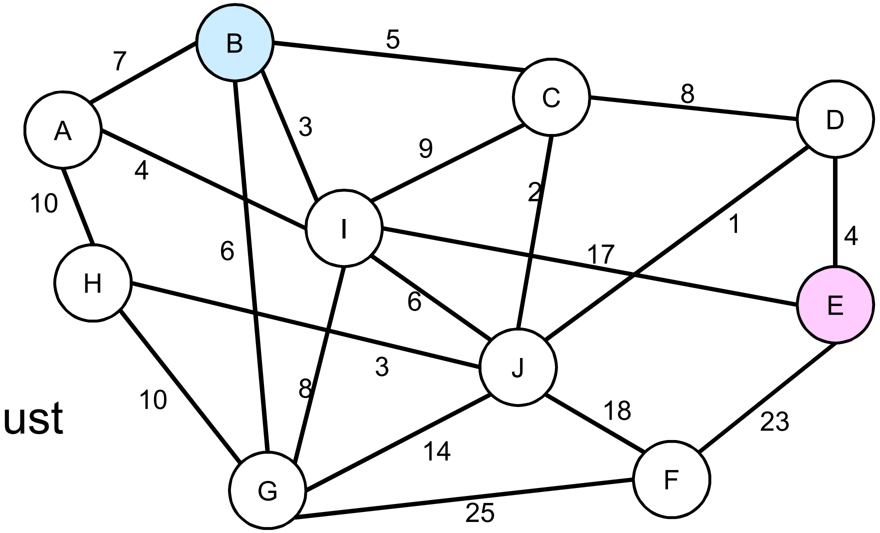
How do we return the path?

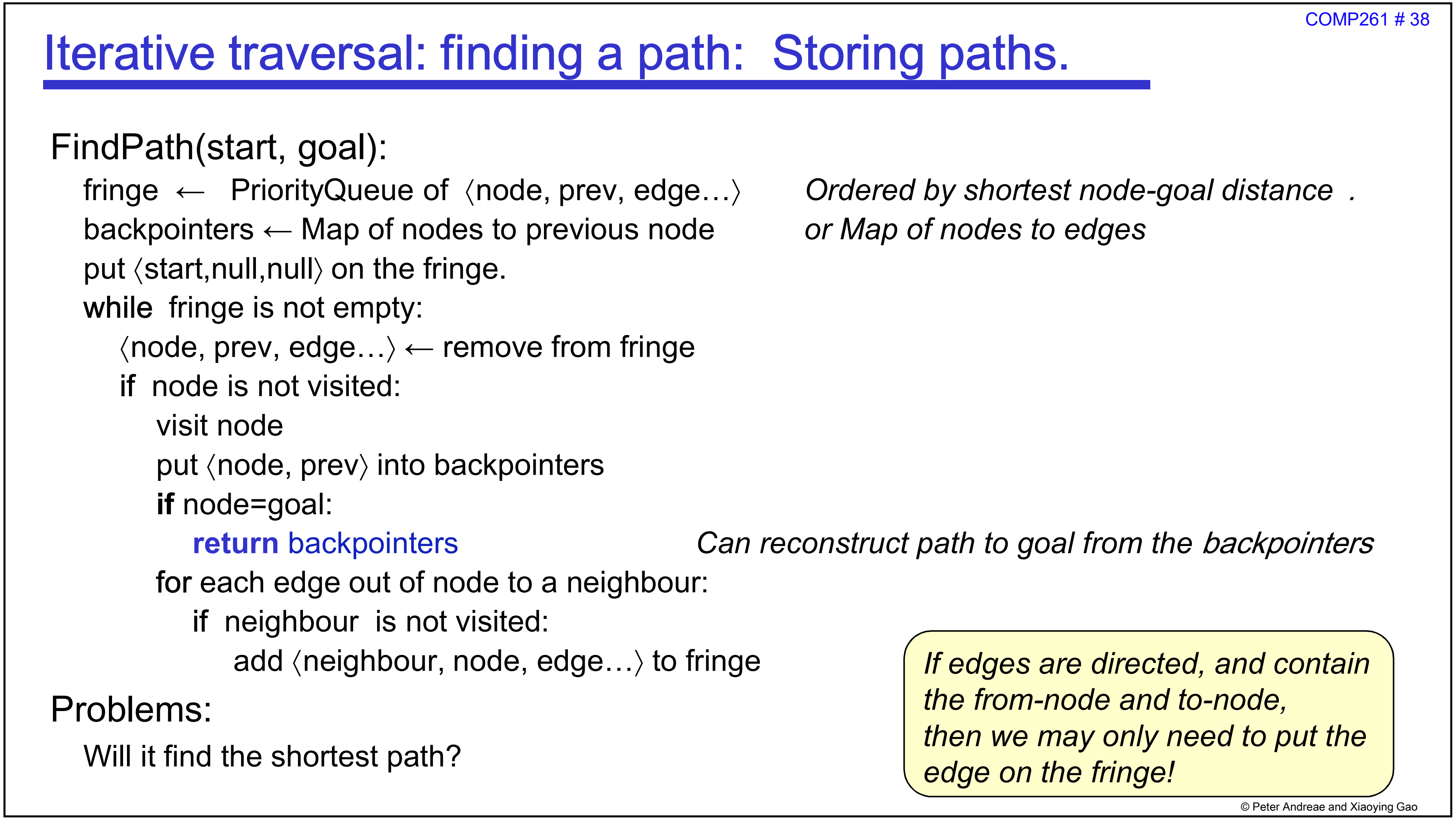
Iterative search, keeping track of the path

- When we visit a node, we need to record how we got to it ("backpointers")

- Use a Map from node to previous node
- But how do we know where we came from when we take the node off the fringe?

- The fringe needs to contain more than just the node:
  - the node,
  - the node we came from,
  - …. the edge we came along
  - …. other information to help decide





COMP261 # 38

Iterative traversal: finding a path: Storing paths.

FindPath(start, goal):

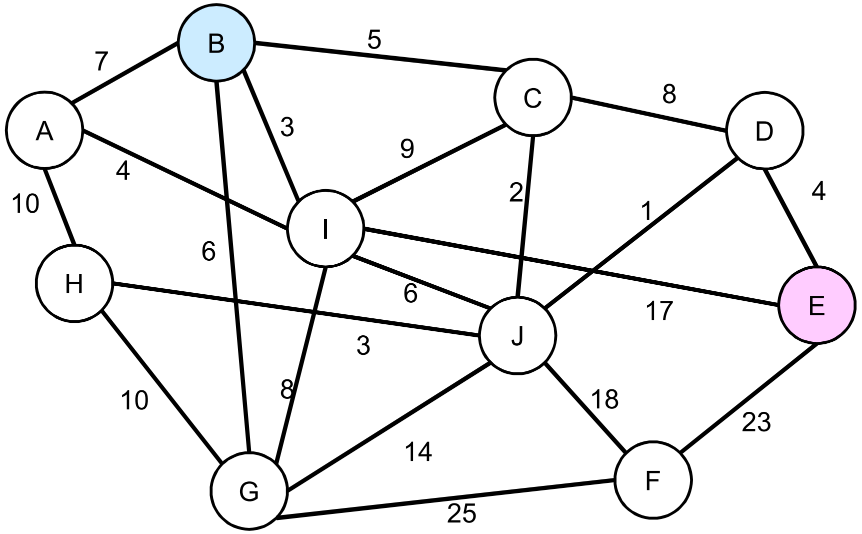
```
fringe ←  PriorityQueue of ⟨node, prev, edge…⟩        Ordered by shortest node-goal distance .
backpointers ← Map of nodes to previous node          or Map of nodes to edges
put ⟨start,null,null⟩ on the fringe.
while fringe is not empty:
   ⟨node, prev, edge…⟩ ← remove from fringe
   if node is not visited:
      visit node
      put ⟨node, prev⟩ into backpointers
      if node=goal:
         return backpointers                Can reconstruct path to goal from the backpointers
      for each edge out of node to a neighbour:
         if neighbour is not visited:
            add ⟨neighbour, node, edge…⟩ to fringe
```

Problems:

Will it find the shortest path?

If edges are directed, and contain the from-node and to-node, then we may only need to put the edge on the fringe!

Paths from BackPointers



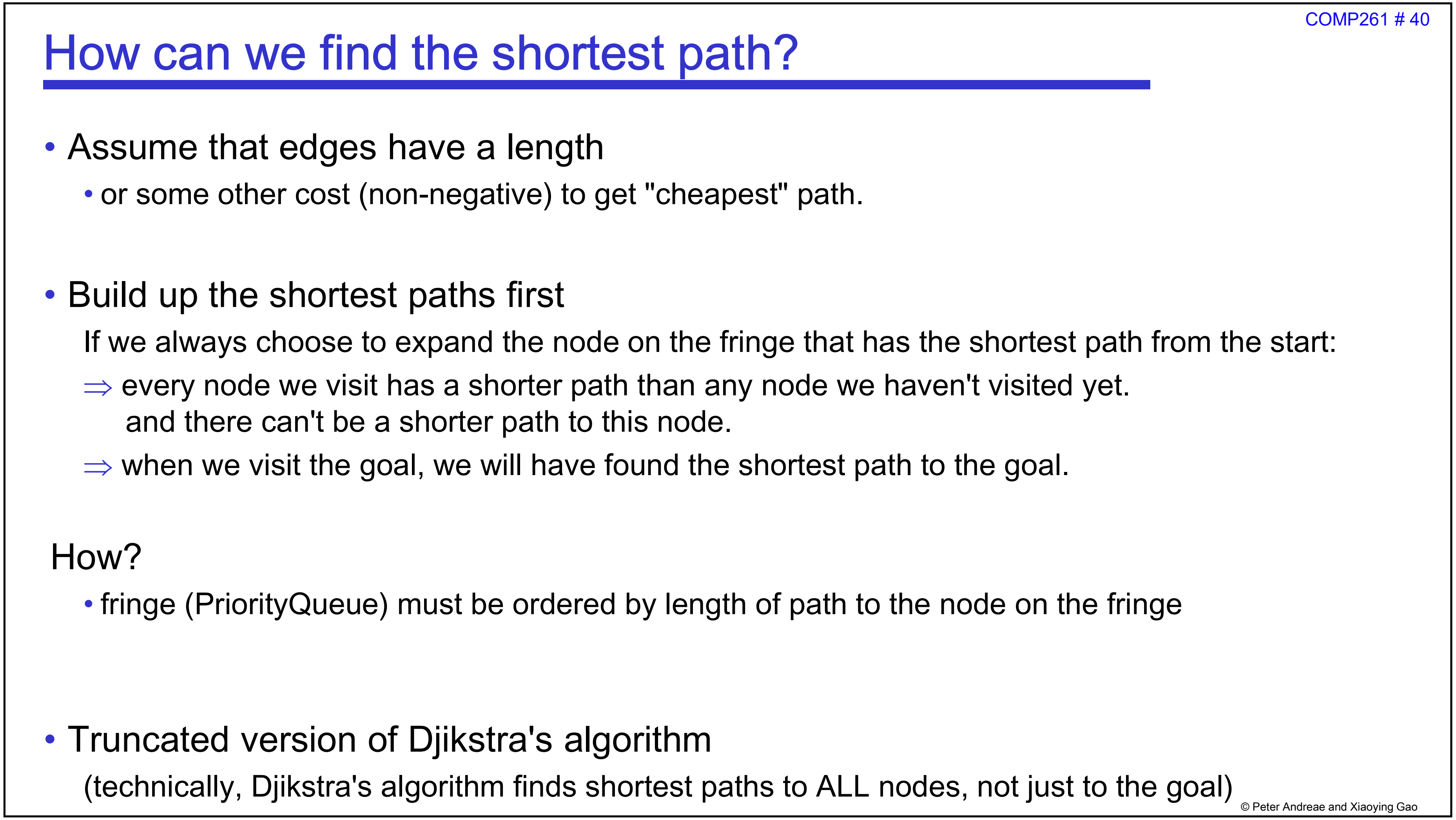
- Backpointers:

```
ReconstructPath(start, goal, backpointers)
   path ← List of nodes
   add goal to path
   node ← goal
   while node ≠ start
      node ← backpointers.get(node)
      add node to path
   reverse path
```

Map:node→prev

```
ReconstructPath(start, goal, backpointers)
   path ← List of edges
   node ← goal
   do
      edge ← backpointers.get(node)
      add edge to path
      node ← edge.from
   until node = start
```

Map:node→edge



COMP261 # 40

How can we find the shortest path?

- Assume that edges have a length
  - or some other cost (non-negative) to get "cheapest" path.

- Build up the shortest paths first

  If we always choose to expand the node on the fringe that has the shortest path from the start:

  ⇒ every node we visit has a shorter path than any node we haven't visited yet.
  and there can't be a shorter path to this node.

  ⇒ when we visit the goal, we will have found the shortest path to the goal.

How?

- fringe (PriorityQueue) must be ordered by length of path to the node on the fringe

- Truncated version of Djikstra's algorithm

  (technically, Djikstra's algorithm finds shortest paths to ALL nodes, not just to the goal)

Finding the Shortest Path (Djikstra)

```
FindShortestPath(start, goal):
   fringe ←  PriorityQueue of ⟨node, edge, length-to-node⟩      Ordered by shortest length-to-node
   backpointers ← Map of nodes to edges
   put ⟨start, null, 0⟩ on the fringe.                           .
   while  fringe is not empty:
      ⟨node, edge, length-to-node⟩ ←  remove from fringe
      if  node is not visited:
         visit node
         put ⟨node, edge⟩ into backpointers
         if node=goal:
            return ReconstructPath(start, goal, backpointers)
         for each edge out of node to a neighbour:
            if  neighbour  is not visited:
               length-to-neighbour ←  length-to-node +  edge.length
               add ⟨neighbour, edge,  length-to-neighbour⟩ to fringe
```

Note: we check if a node is the goal when we remove from the fringe, not when we add it to the fringe.

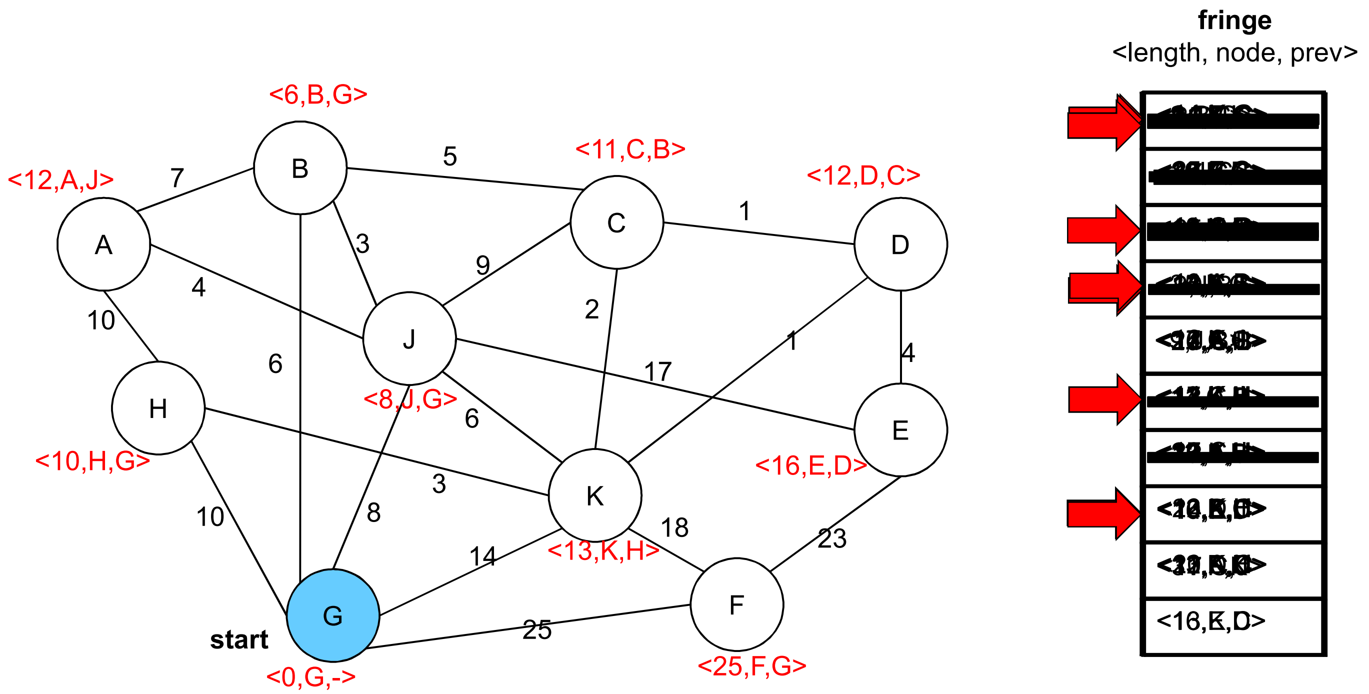
Finding All Shortest Paths: Djikstra's Algorithm

```
FindShortestPaths (start, goal):
   fringe ←  PriorityQueue of ⟨node, edge, length-to-node⟩        Ordered by shortest length-to-node
   backpointers ← Map of nodes to edges
   put ⟨start, null, 0⟩ on the fringe.                            .
   while  fringe is not empty:
      ⟨node, edge, length-to-node⟩ ←  remove from fringe
      if  node is not visited:
         visit node
         put ⟨node, edge⟩ into backpointers


         for each edge out of node to a neighbour:
            if  neighbour  is not visited:
               length-to-neighbour ← length-to-node +  edge.length
               add ⟨neighbour, edge,  length-to-neighbour⟩ to fringe
   return backpointers
```

Djikstra keeps going until all nodes visited.
Backpointers = all shortest paths!

Example of Dijkstra’s Algorithm



Visited: ●

Backpointers: <dist, node, prev>

What's the cost of Dijkstra's algorithm?

If a graph has N nodes and E edges:

Identify the most expensive line:

```
while  fringe is not empty:
       :
    for each edge out of node to a neighbour:
       :
          add ⟨neighbour, edge,  length-to-neighbour⟩ to fringe
```

How many times might we do that line?
What is the cost of that line?

Problem with Djikstra's Algorithm

- If we want all shortest paths: Djikstra is best.
  - Greedy: never backtracks and every iteration adds a path to the answer
- If we want the shortest path to a goal: Djikstra is wasteful:
  - spends time building paths to useless nodes, not on the way to the goal:
- Need to combine:
  - length of path to here, AND
  - estimate of remaining dist
    - Biases the choice towards nodes that are on the way to the goal.