Algorithms and Data Structures

COMP261
Pathfinding 1: Dijkstra's Algorithm

Yi Mei

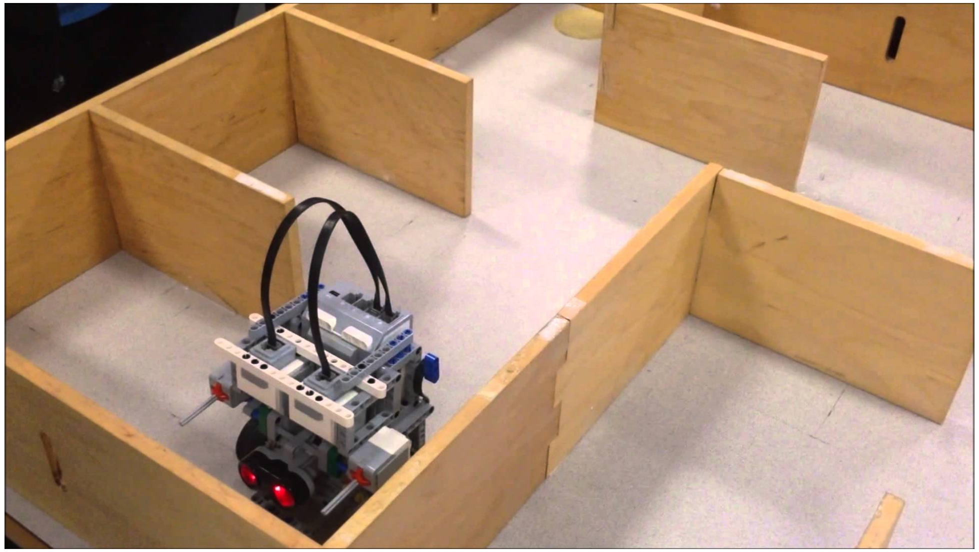
yi.mei@ecs.vuw.ac.nz

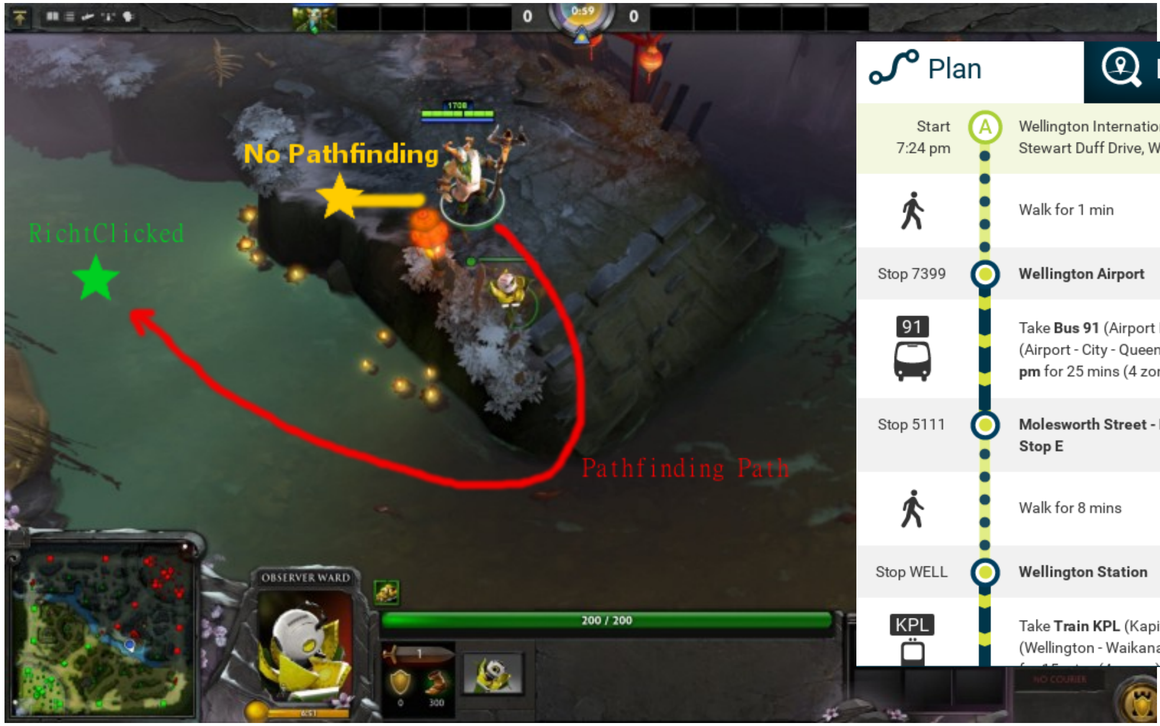
Outline

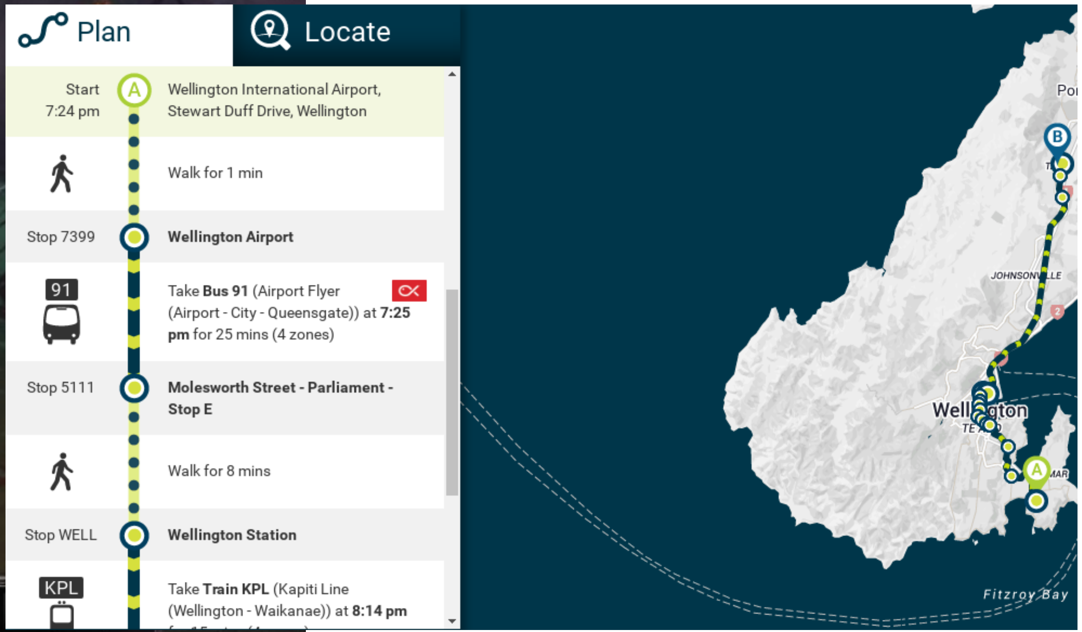
- Path finding
- Graph search technique for path finding
- Dijkstra's algorithm

Path Finding

- Robotics
- Video games
- Journey planner

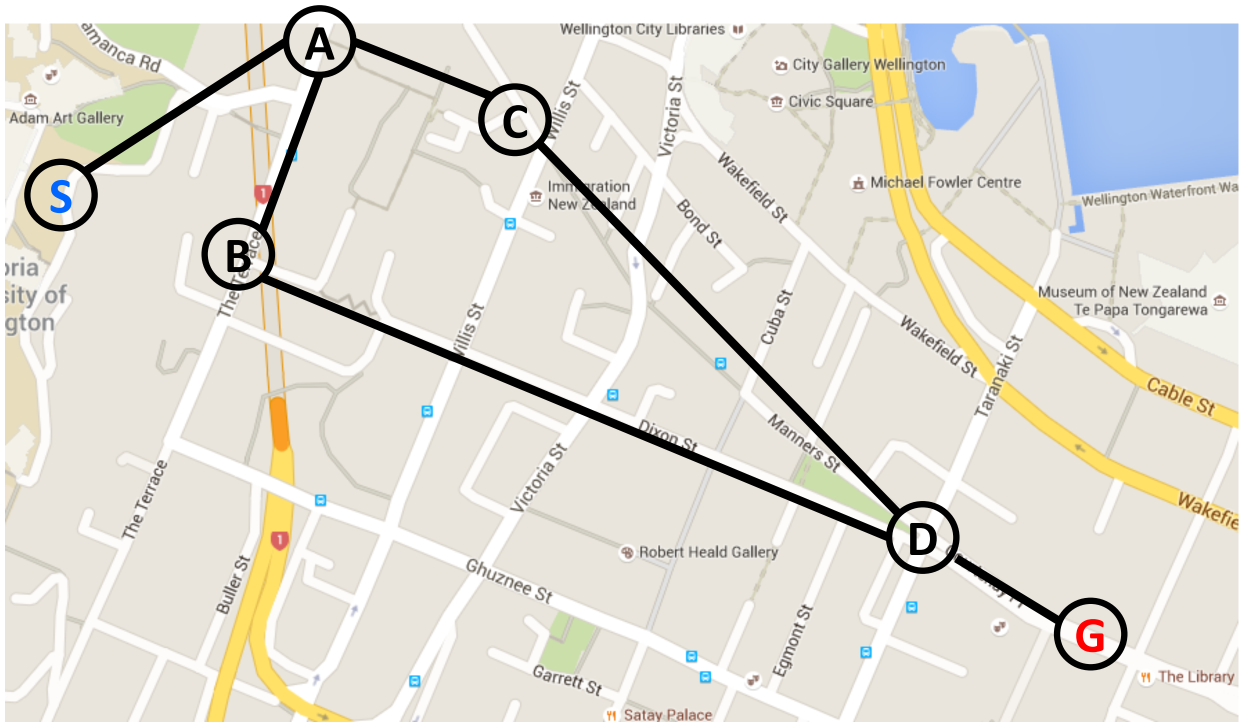


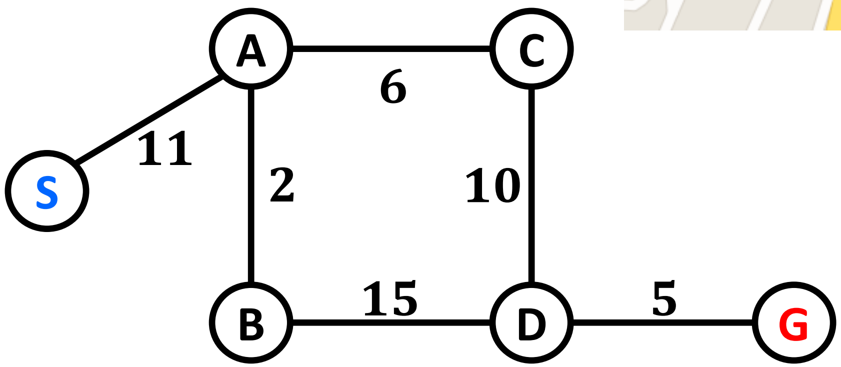




Path Finding

- Model environment as graph
- From VUW to Courtney Pl?





Path Finding

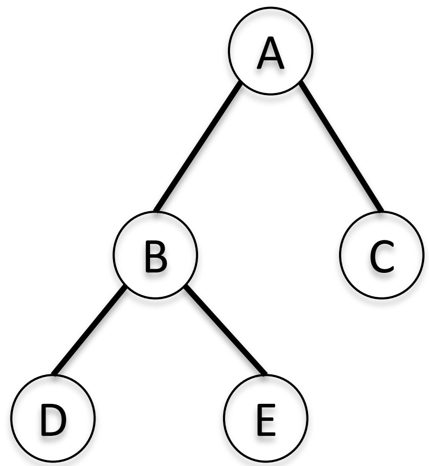
- Given a graph with a set of nodes and a set of edges, each edge is undirected and has a cost (e.g. length/travel time)
- Find the least-cost (e.g. shortest/fastest) path(s)
  - From one node to another (1-to-1)
  - From one node to all other nodes (1-to-all)
  - For all node pairs (all-to-all)

- Graph search
  - Depth-First Search (DFS)
  - Breadth-First Search (BFS)
  - …

Graph Search

- Keep a "**fringe**" (all the leaf nodes of the search tree: frontier)
  - **Data structure** for the fringe: stack/queue/priority queue
- Start the fringe with one node (root)
- **Repeat until** stopping criteria are met:
  - Choose a node from the fringe to visit (visit a leaf node);
  - Add its neighbours to the fringe, and remove the node from fringe (expand the leaf node and grow the search tree);

- Example: DFS to visit all nodes
  - Use stack as fringe (last-in-first-out)
  - Step 1: fringe = {A}, visited = {}
  - Step 2: visit A; fringe = {B,C}, visited = {A}
  - Step 3: visit C; fringe = {B}, visited = {A,C}
  - Step 4: visit B; fringe = {D,E}, visited = {A,C,B}
  - Step 5: visit E; fringe = {D}, visited = {A,C,B,E}
  - Step 6: visit D; fringe = {}, visited = {A,C,B,E,D}



- What data structure for fringe for BFS?

Dijkstra's Algorithm

- 1-to-all path finding (from one node to all other nodes)
- Also for 1-to-1 (by early stopping)

- Use the graph search technique
  - Start the fringe with the start node (`start`)
  - Always expand the unvisited node that are "**most** likely to be in the shortest path" first
  - It has the **minimal cost from the start node**
  - **Expand**: visit, remove from the fringe, and add its neighbours into the fringe

- For this purpose, a data structure `SearchNode`, containing
  - `Node`: the node in the graph it represents
  - `Cost`: the minimal cost from the start node found so far
  - `Prev`: the previous node in the shortest path found so far

  - Denoted as `<Cost, Node, Prev>`

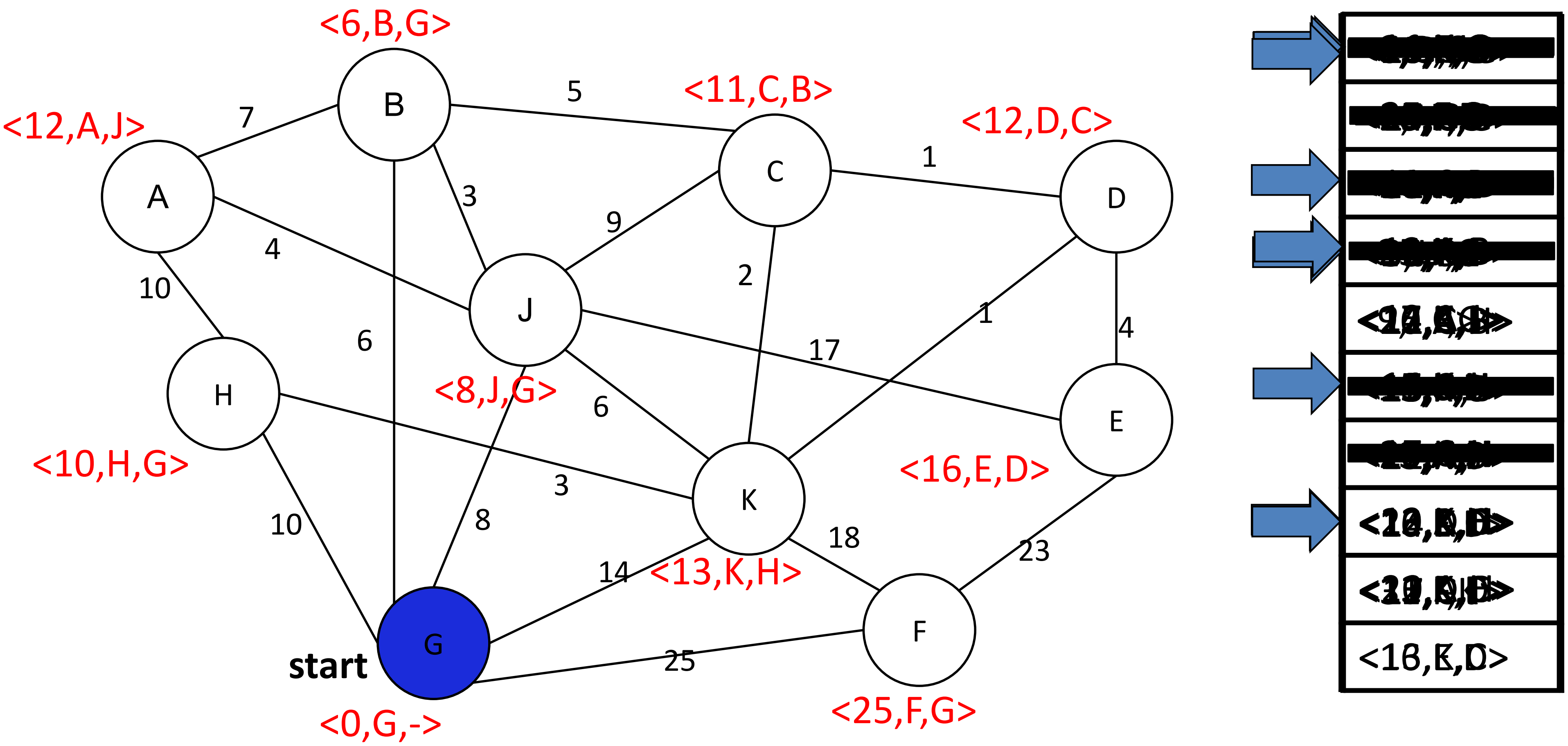
Dijkstra's Algorithm

Input: A weighted graph and a `start` node
Output: Shortest paths from `start` to all other nodes

Initially all the nodes are *unvisited*, and the fringe has a single element
`<0, start, null>`;

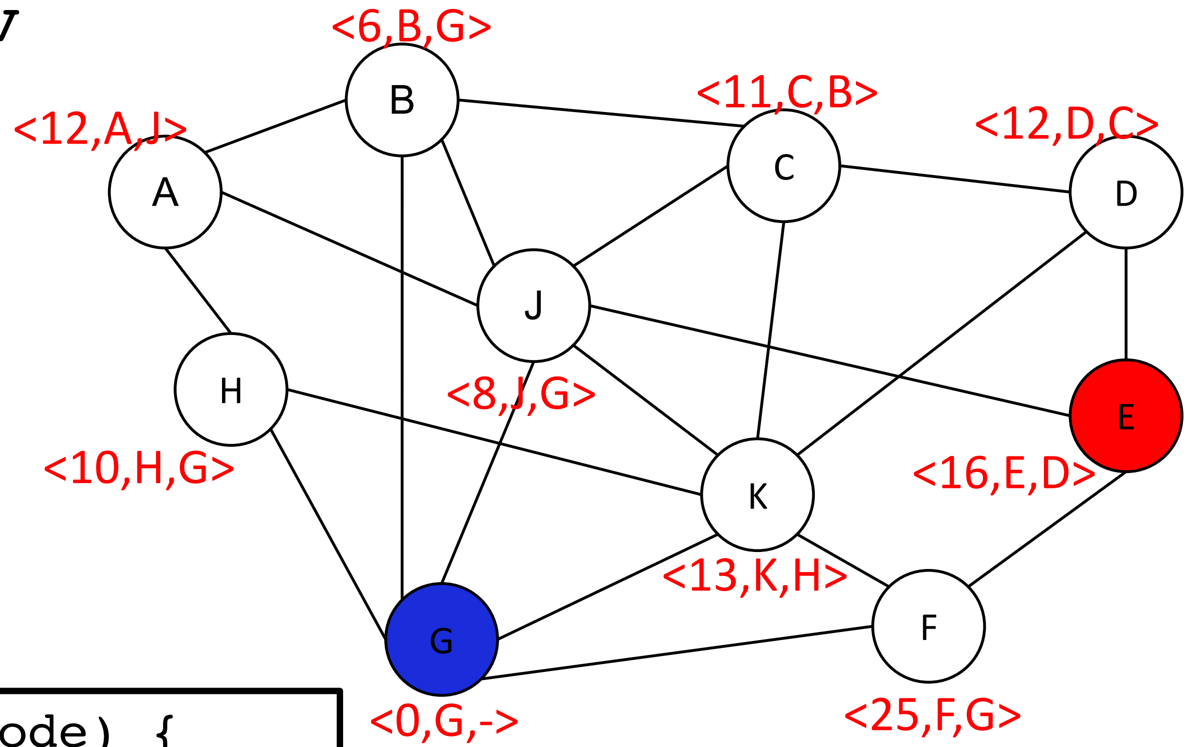
```
While (the fringe is not empty) {
    Expand <cost*, node*, prev*> from the fringe, where cost* is the minimal cost among
all the elements in the fringe;

    if (node* is unvisited) {
        Set node* as visited;
        Set node*.prev = prev*;

        for (edge = (node*, neigh) outgoing from node*) {
            if (neigh is unvisited) {
                costToNeigh = cost* + edge.weight;
                add a new element <costToNeigh, neigh, node*> into the fringe;
            }
        }
    }
}
Obtain the shortest path based on the .prev fields;
```

Example of Dijkstra's Algorithm



Obtain Shortest Path

- We can obtain the shortest path from the start node to any other node from `prev`




```
getShortestPath(start, node) {
    path ← (node), curr ← node;
    while (not curr.prev = start) {
        curr ← curr.prev;
        path ← (curr, path);
    }
    return path;
}
```

Path

G	B	C	D	E

Correctness of Dijkstra's Algorithm

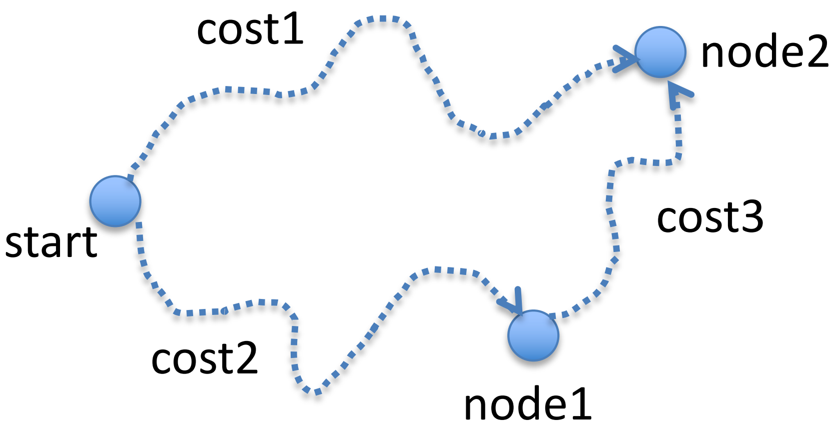
- **Theorem**: the path found for each node by Dijkstra's algorithm is the shortest path from the start node to the node

- **Proof**:
  - Each node is **visited only once**, when it is removed from the fringe
    - If future SearchNode contains the same node, it will be skipped
  - The minimal cost from the start node to a node is obtained when visiting the node (for the first time)

Correctness of Dijkstra’s Algorithm

- **Theorem**: the path found for each node by Dijkstra’s algorithm is the shortest path from the start node to the node

- **Proof (continued)**:
  - No shorter path from the start node to a node **before** it is visited.
    - Otherwise the node would have been visited earlier (Dijkstra’s algorithm always expand a node that is nearer the start node)
  - No shorter path from the start node to a node **after** it is visited.
    - Dijkstra’s algorithm always expand a node that is nearer the start node



If cost1 < cost2 + cost3, **then**
<cost1, node2, start> is visited **before**
<cost2+cost3, node2, node1>

If cost1 > cost2 + cost3, **then**
<cost1, node2, start> is visited **after**
<cost2+cost3, node2, node1>

Data Structure for Fringe

- Which data structure for the fringe should be used in Dijkstra's Algorithm?

- Most common operation to the fringe
  - Add an element `<cost, node, prev>` to the fringe
  - Visit the node `<cost*, node*, prev*>` with the **minimal cost among all the elements** in the fringe (and remove it from the fringe)

- **Priority queue** – treat cost as the priority
  - Efficient for getting the element with the best priority

1-to-1 Dijkstra's Algorithm

- If we want to find ONLY the shortest (least cost) path from the start node to a particular goal node, then we can do it faster
  - Stop once we find the shortest path to the goal node, no need to continue for all the other nodes

```
Input: A weighted graph, a start node, a goal node
Output: A shortest path from start to goal

Initially all the nodes are unvisited, and the fringe has a single element
<0, start, null>;
While (the fringe is not empty) {
    Expand <cost*, node*, prev*> from the fringe;

    if (node* is unvisited) {
        Set node* as visited, and set node*.prev = prev*;

        if (node* is the goal node) return;

        // add all the unvisited neighbours of node* into the fringe
        …
    }
}
```

Summary

- Path finding
  - 1-to-1 (Early-stop Dijkstra's algorithm)
  - 1-to-all (Dijkstra's algorithm)
  - All-to-all (not mentioned, but can also be solved by Dijkstra's algorithm)

- Dijkstra's algorithm
  - Graph search (fringe to store leaf nodes of search tree)
  - Priority queue for fringe
  - Always visit the node with minimal cost from the start node

- Can we do better?
  - Next lecture: A* search