Minimum Cost Search (Dijkstra's Algorithm)

(Some material from Artificial Intelligence: Foundations of Computational Agents, Poole and Mackworth, 2010.)

Path Nodes

PathNode
`+state: Node` `+parent: PathNode` `+path_cost: Number`
`+PathNode(State: Node, parent: PathNode,` `step_cost: Number)` *path_cost = step_cost + parent.path_cost*

- For our purposes:
  - A "Node" corresponds to a state in the problem. There may be infinitely many Nodes.
  - A "PathNode" is a data type that represents a partial solution and the associated cost.
- (This is not standard terminology. Our textbook does not discuss PathNodes; in Russel & Norvig, they are just called "Nodes".)

Depth First Search (With PathNodes!)

```
procedure DepthFirstSearch(G,S,goal)
    Inputs
        G: graph with nodes N and arcs A
        S: set of start nodes
        goal: Boolean function of states
        cost: The cost function for arcs
    Output
        path from a member of S to a node for which goal is true
        or ⊥ if there are no solution paths
    Local
        Frontier: a stack of PathNodes
        Explored: set of nodes that have been expanded

    Frontier ← Empty Stack
    Frontier.push(PathNode(s, None, 0)) for s ∈ S
    Explored ← {}
    while (Frontier is not empty)
        Pop pNode from Frontier
        Explored ← Explored ∪ {pNode.state}
        if ( goal(pNode.state)) then
            return The path represented by pNode
        For all {⟨s_k,s⟩ : ⟨s_k,s⟩ ∈ A ∧ s ∉ Frontier ∧ s ∉ Explored}
            Frontier.push(PathNode(s, pNode, cost(⟨s_k,s⟩))
    return ⊥
```

Lowest Cost Search (Dijkstra's Algorithm)

```
procedure LowestCostSearch(G,S,goal)
    Inputs
        G: graph with nodes N and arcs A
        S: set of start nodes
        goal: Boolean function of states
        cost: The cost function for arcs
    Output
        path from a member of S to a node for which goal is true
        or ⊥ if there are no solution paths
    Local
        Frontier: a Priority Queue of PathNodes ordered by cost
        Explored: set of nodes that have been expanded

    Frontier ← Empty Stack
    Frontier.enqueue(PathNode(s, None, 0)) for s ∈ S
    Explored ← {}
    while (Frontier is not empty)
        Pop pNode from Frontier
        Explored ← Explored ∪ {pNode.state}
        if ( goal(pNode.state)) then
            return The path represented by pNode
        For all {⟨s_k,s⟩ : ⟨s_k,s⟩ ∈ A ∧ s ∉ Frontier ∧ s ∉ Explored}
            Frontier.enqueue(PathNode(s, pNode, cost(⟨s_k,s⟩)
    return ⊥
```

Lowest Cost Search (Dijkstra's Algorithm)

```
procedure LowestCostSearch(G,S,goal)
    Inputs
        G: graph with nodes N and arcs A
        S: set of start nodes
        goal: Boolean function of states
        cost: The cost function for arcs
    Output
        path from a member of S to a node for which goal is true
        or ⊥ if there are no solution paths
    Local
        Frontier: a Priority Queue of PathNodes ordered by cost
        Explored: set of nodes that have been expanded

    Frontier ← Empty Stack
    Frontier.enqueue(PathNode(s, None, 0)) for s ∈ S
    Explored ← {}
    while (Frontier is not empty)
        Pop pNode from Frontier
        Explored ← Explored ∪ {pNode.state}
        if ( goal(pNode.state)) then
            return The path represented by pNode
        For all {⟨s_k,s⟩ : ⟨s_k,s⟩ ∈ A ∧ s ∉ Frontier ∧ s ∉ Explored}
            Frontier.enqueue(PathNode(s, pNode, cost(⟨s_k,s⟩))
    return ⊥
```

Missing detail: if *s* is already in the frontier, then it's PathNode should be replaced if the new node would have a lower path cost.