Algorithms and Data Structures

COMP261
Tutorial Week 3

Yi Mei

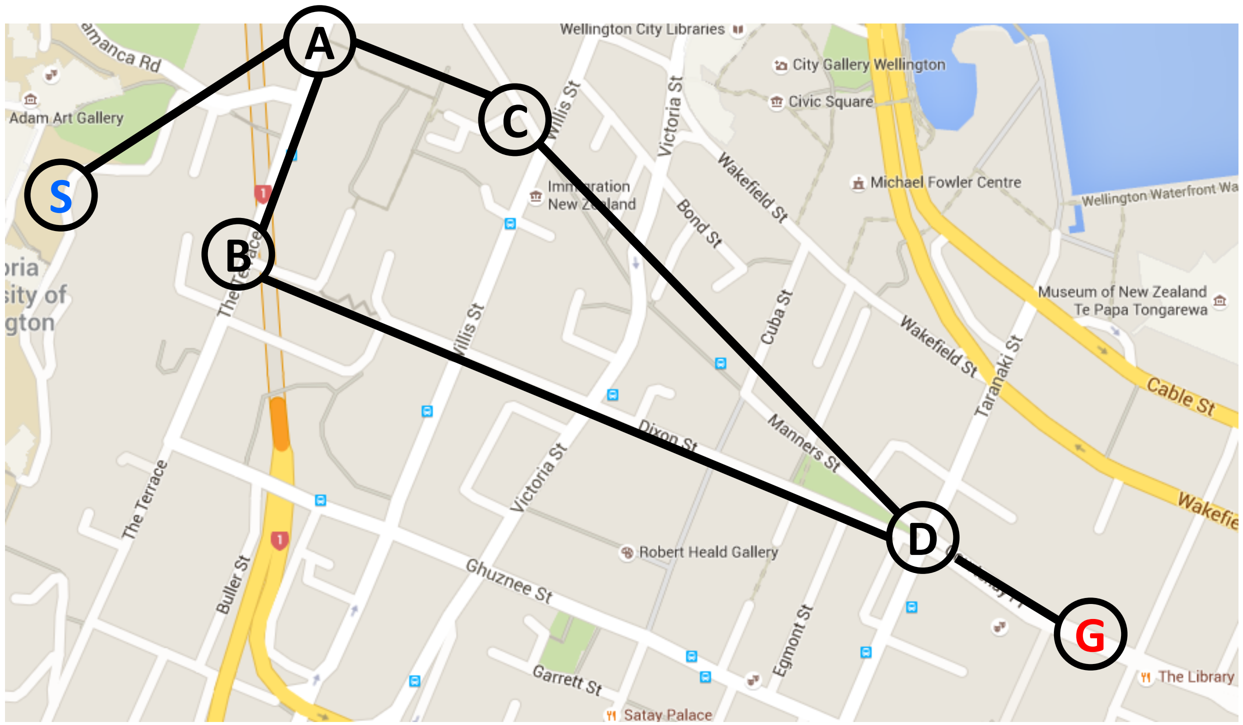
yi.mei@ecs.vuw.ac.nz

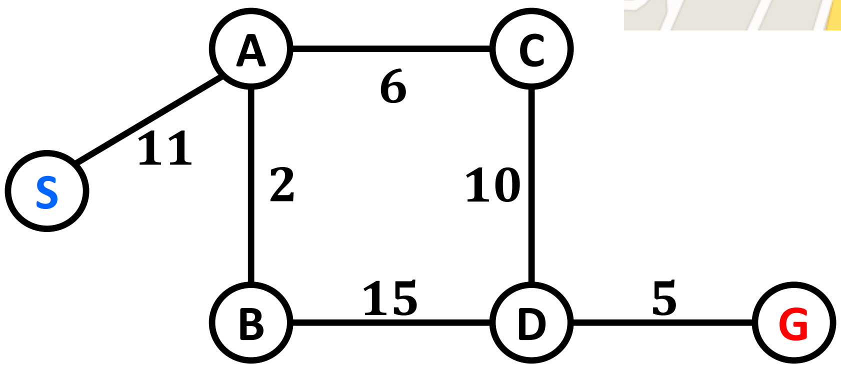
Outline

- Dijkstra's Algorithm
  - Early stopping
- A* Search
  - 1-to-1 mapping
  - Heuristic function
  - Conditions for success: admissible and consistent

Path Finding

- In a connected weighted graph, find the least cost path from one node to another (or all others)





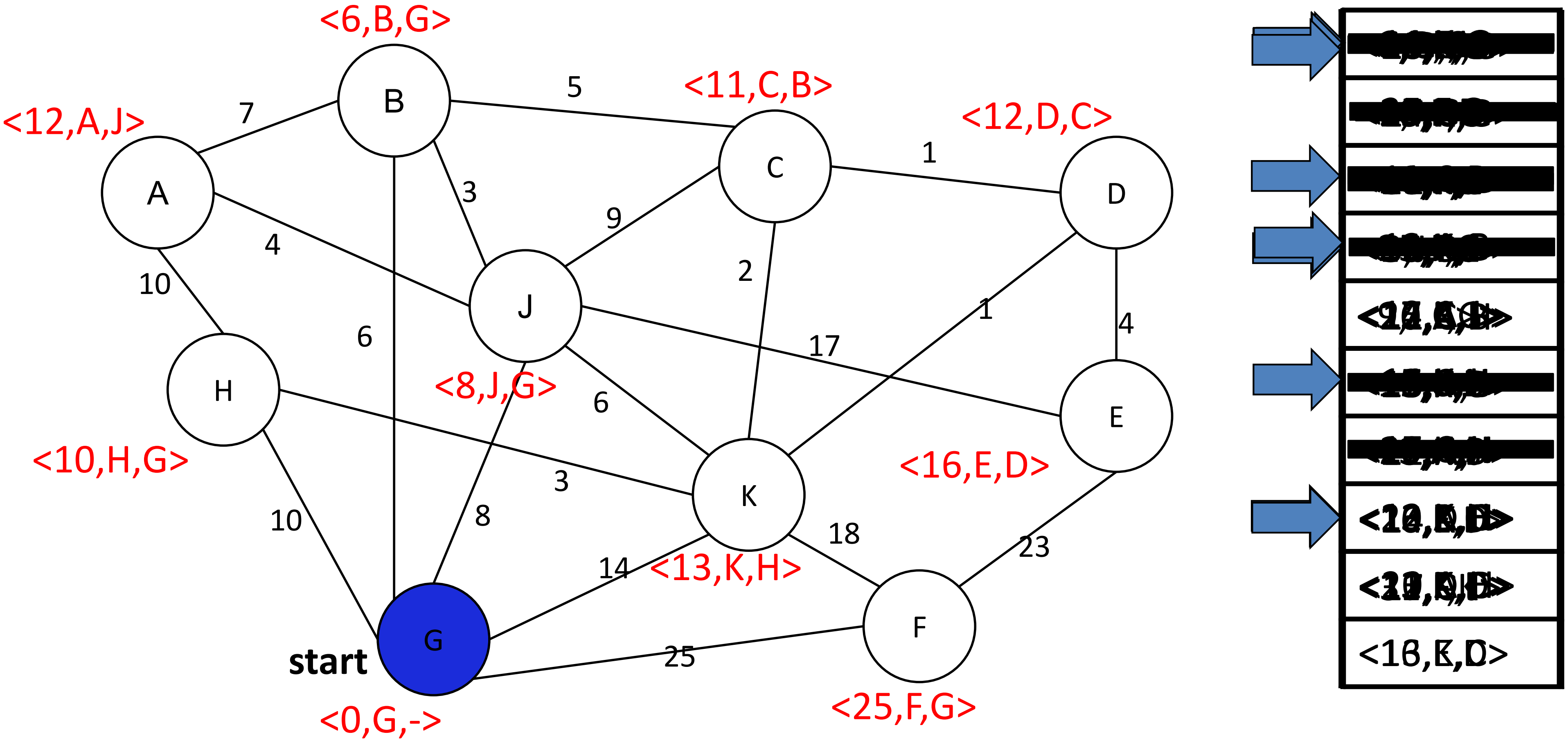
Dijkstra's Algorithm

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, and 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;
```

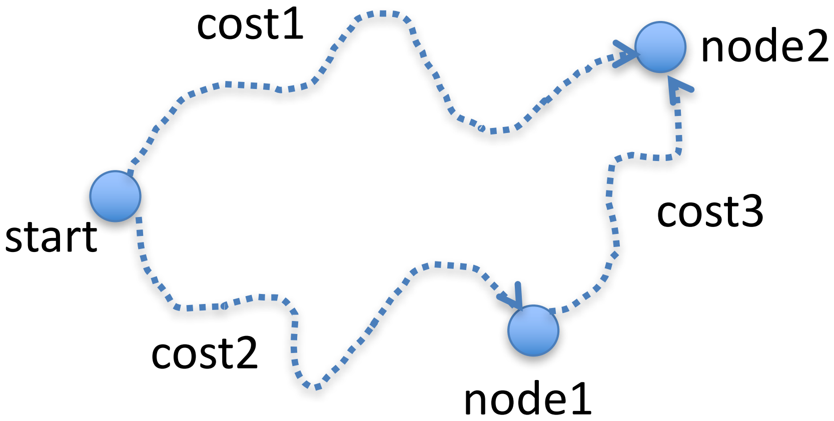
Example of Dijkstra's Algorithm



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**
  - 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>

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
        …
    }
}
```

A* Search

Input: A weighted graph, a `start` node, a `goal` node, the heuristic function `h()` for each node
Output: Shortest path from `start` to `goal`

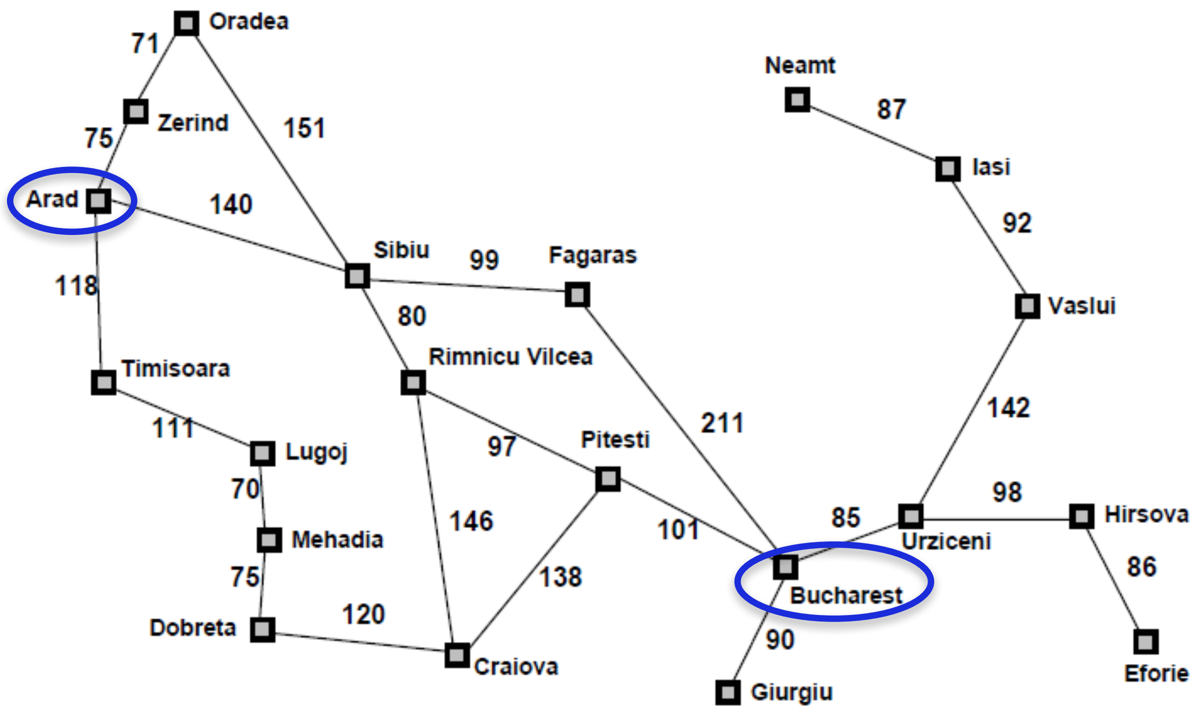
Initially all the nodes are *unvisited*, and the fringe has a single element `<start, null, 0, f(start)>`;

```
While (the fringe is not empty) {
    Expand <node*,prev*,g*,f*> from the fringe, where f* is minimal among all the elements in the fringe;
    if (node* is unvisited) {
        Set node* as visited, and set node*.prev = prev*;

        if (node* is the target node) break;

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

Example of A* Search

- Shortest path from Arad to Bucharest?

<Arad,null,0,366>



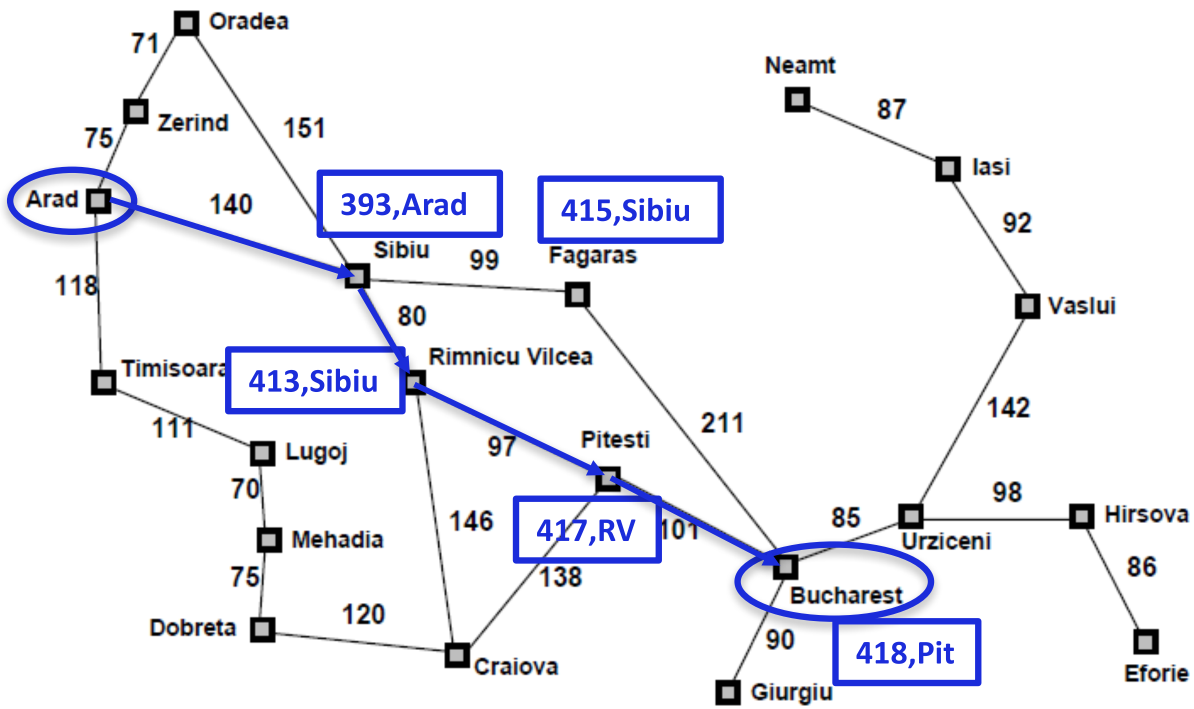
Estimated cost to Bucharest

Arad	366	Mehadia	241
Bucharest	0	Neamt	234
Craiova	160	Oradea	380
Dobreta	242	Pitesti	100
Eforie	161	Rimnicu Vilcea	193
Fagaras	176	Sibiu	253
Giurgiu	77	Timisoara	329
Hirsova	151	Urziceni	80
Iasi	226	Vaslui	199
Lugoj	244	Zerind	374

Example of A* Search

- Shortest path from Arad to Bucharest?

<Timisoara,Arad,118,447>
<Zerind,Arad,75,449>
<Craiova,RV,366,526>
<Bucharest,Fagaras,450,450>
<Bucharest,Pitesti,418,418>
<Craiova,Pitesti,455,615>



Estimated cost to Bucharest

Arad	366	Mehadia	241
Bucharest	0	Neamt	234
Craiova	160	Oradea	380
Dobreta	242	Pitesti	100
Eforie	161	Rimnicu Vilcea	193
Fagaras	176	Sibiu	253
Giurgiu	77	Timisoara	329
Hirsova	151	Urziceni	80
Iasi	226	Vaslui	199
Lugoj	244	Zerind	374

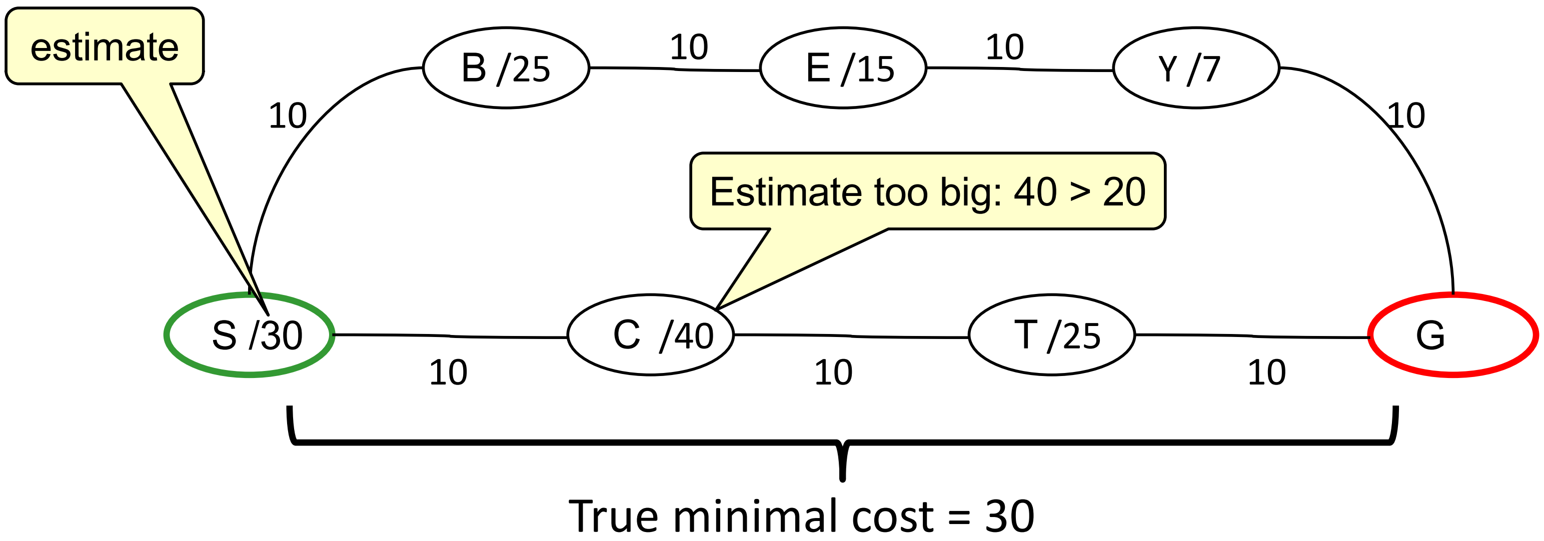
Correctness of A* Search

- The path found for by A* Search is the shortest path from the start node to the goal node if the following conditions are satisfied.
    1. The estimated cost to goal is never greater than the true cost (admissible heuristic)
    2. For each node, the first expand always has the minimal cost from start (consistent/monotonic heuristic)

- Both conditions are about the heuristic function

Admissible Heuristic

- A heuristic function is **admissible**, if it never overestimates the true cost to the goal node
  - If not, then the first visit may not have the minimum cost



<S,-,0,30>
<B,S,10,35>
<C,S,10,50>
<E,B,20,35>
<Y,E,30,37>
<G,Y,40,40>

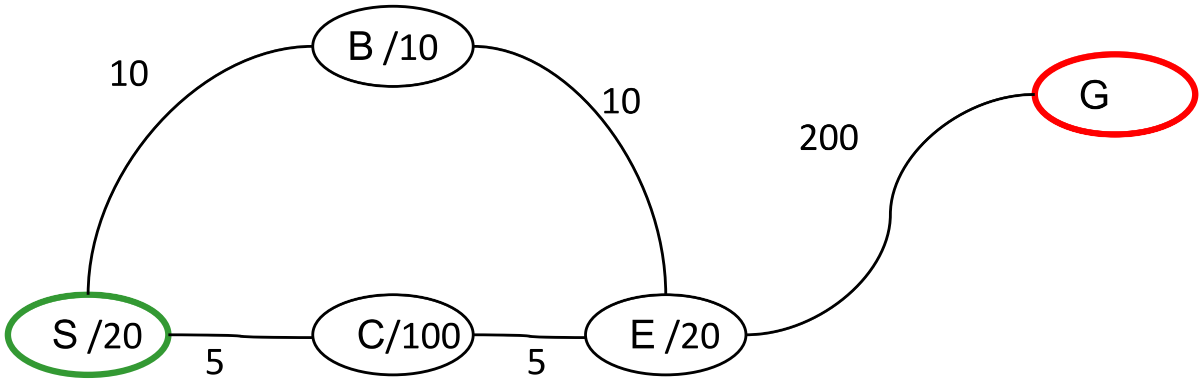
Consistent/Monotonic Heuristic

- To make sure no revisit will lead to better cost, a straightforward way is to make `f = g + h` monotonic (non-decreasing)
  - Whenever expanding `<g, f, node, prev>`, and adding its neighbours `<g', f', neigh, node>`, we always have `f <= f'`
  - We also have
    - `f = g + h(node)`
    - `g' = g + cost(node, neigh)`
    - `f' = g' + h(neigh)`
  - Therefore,
    - `g + h(node) <= g + cost(node, neigh) + h(neigh)`
    - `h(node) <= h(neigh) + cost(node, neigh)`
    - `h(node) – h(neigh) <= cost(node, neigh)`

- A heuristic function is consistent/monotonic if for any node and its neighbour:
    - `h(node) <= h(neigh) + cost(node, neigh)`
    - `h(node) – h(neigh) <= cost(node, neigh)`

Consistent/Monotonic Heuristic

- A counter example:
  - An admissible heuristic may not be consistent/monotonic
  - Revisit may lead to a better cost



All estimates smaller than the true cost

Admissible

h(C) = 100 > h(E) + 5

Not consistent

	<S,-,0,20>
	<B,S,10,20>
	<C,S,5,105>
	<E,B,20,40>
	<G,E,220,220>
C visited	<C,E,25,125>
E visited	<E,C,10,30>