Algorithms and Data Structures

COMP261
Pathfinding 2: A* Search

Yi Mei

yi.mei@ecs.vuw.ac.nz

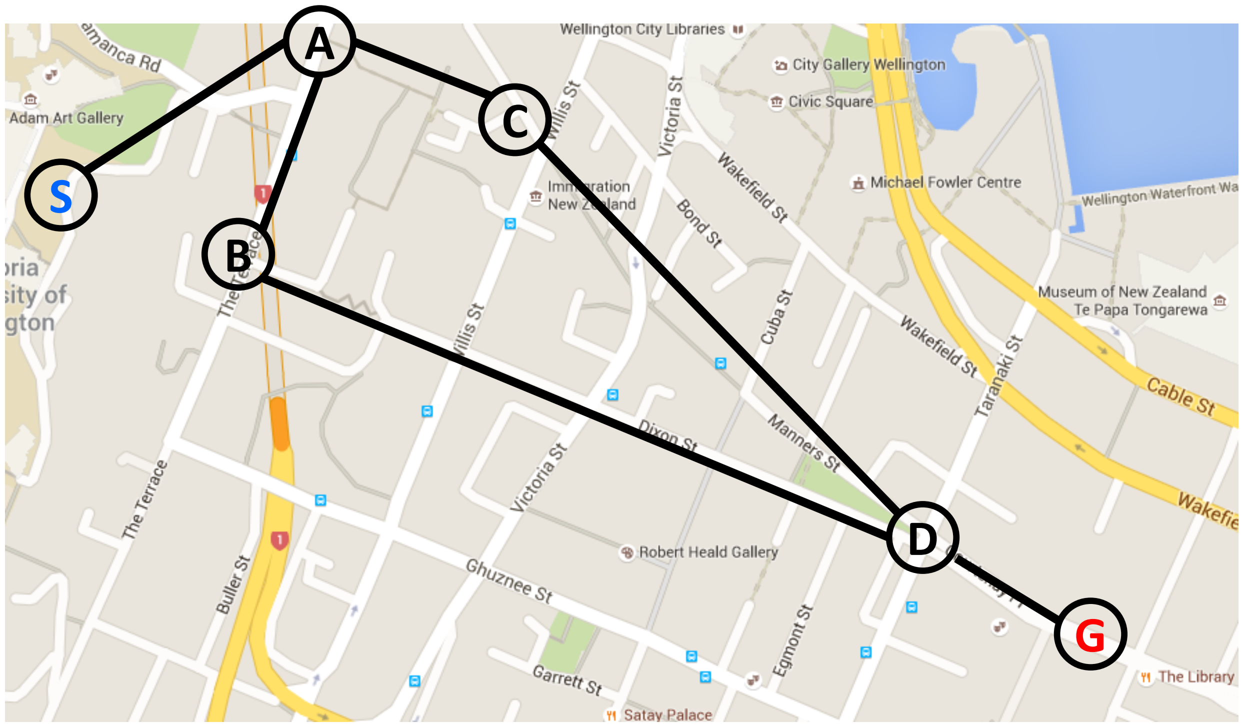
Outline

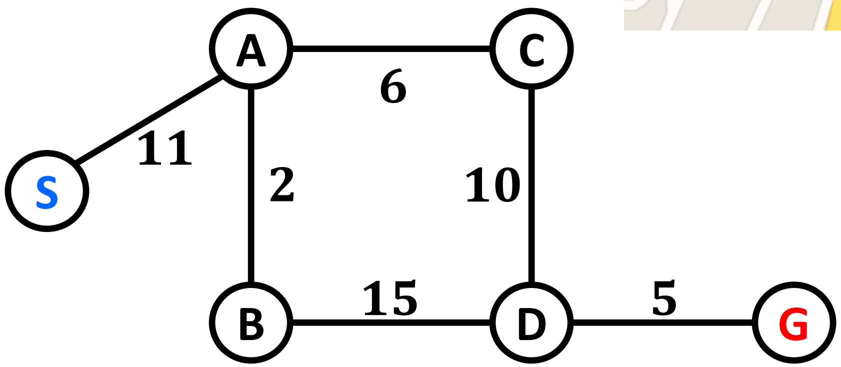
- A* Search
  - Heuristic function to estimate cost to the goal node

- Condition for A* search to be correct
  - Admissible heuristic function
  - Consistent/Monotonic heuristic function

1-to-1 Path Finding

- Find the least-cost path from a start node to a goal node



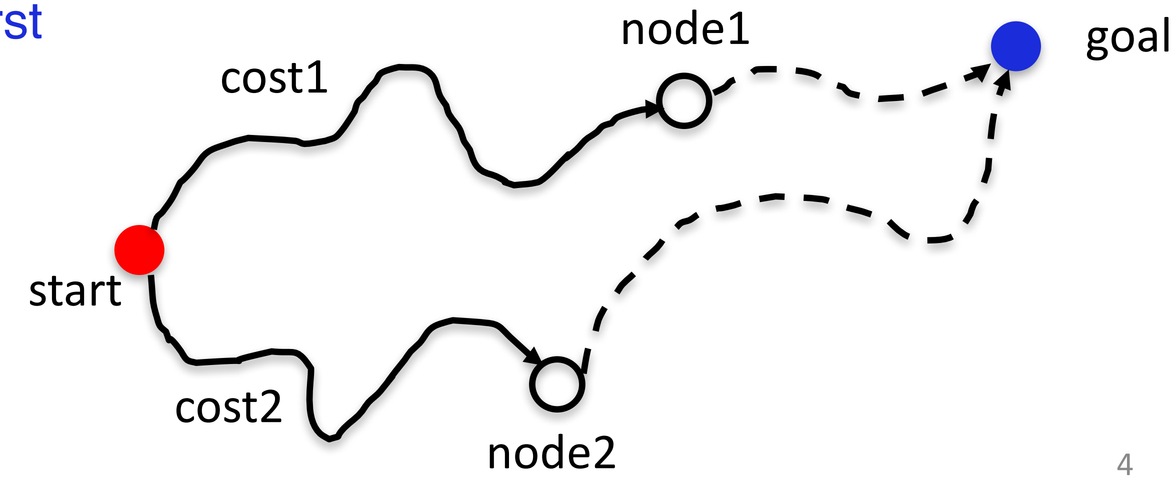


Dijkstra’s Algorithm

- Use the graph search technique
  – Start the fringe with the start node
  – Always expand the unvisited node that **has the minimal cost from the start node**
  – Lack of information towards the goal node

- Example:
  – cost1 > cost2, so Dijkstra’s Algorithm will expand node2 first
  – However, node1 is much closer to the goal node than node2
    - cost(node1, goal) << cost(node2, goal)
    - cost1+cost(node1,goal) < cost2+cost(node2,goal)
  – Should expand node1 first

- A* Search
  – **f(node) = g(node) + h(node)**
  – **g(node)**: cost from start
  – **h(node)**: *estimated* cost to goal



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;

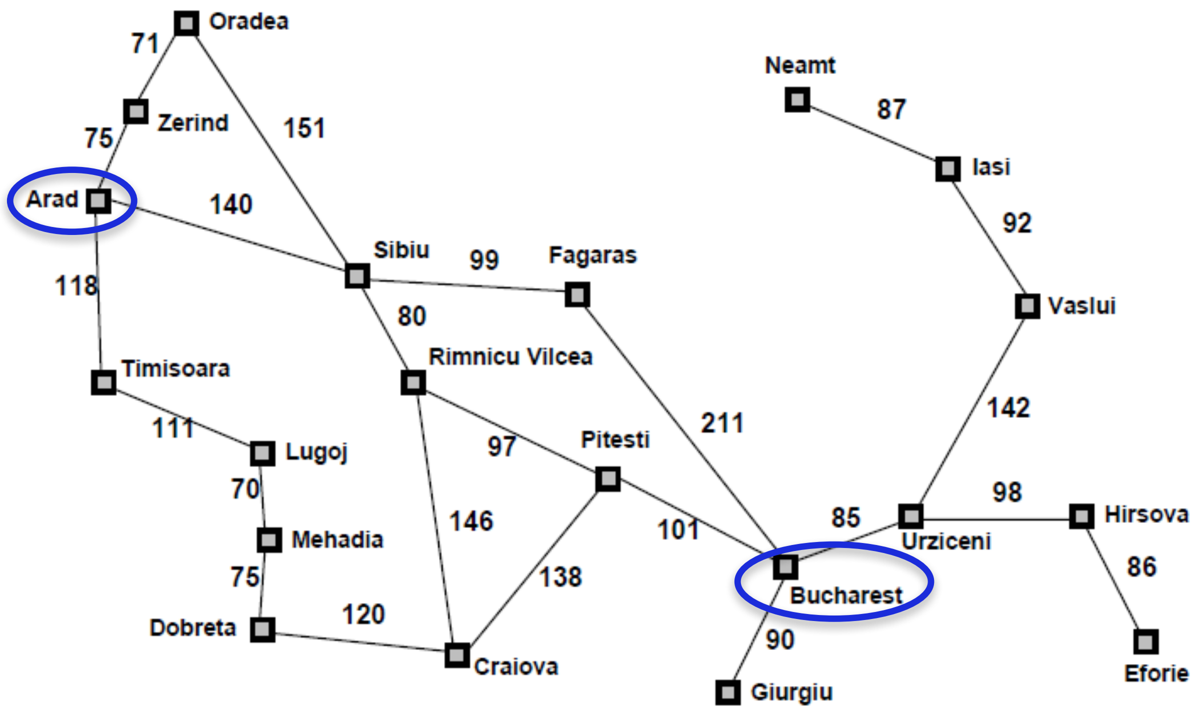
```
    if (node* is unvisited) {
        Set node* as visited, and set node*.prev = prev*;

        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>



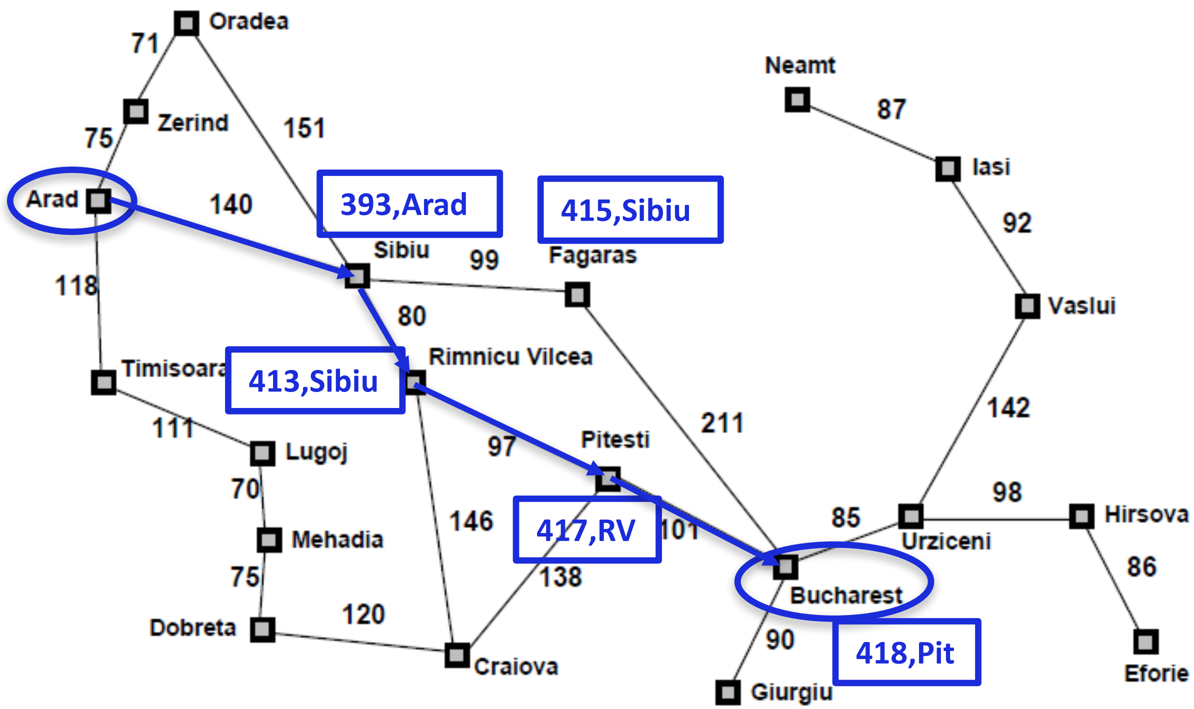
Estimated cost to Bucharest

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

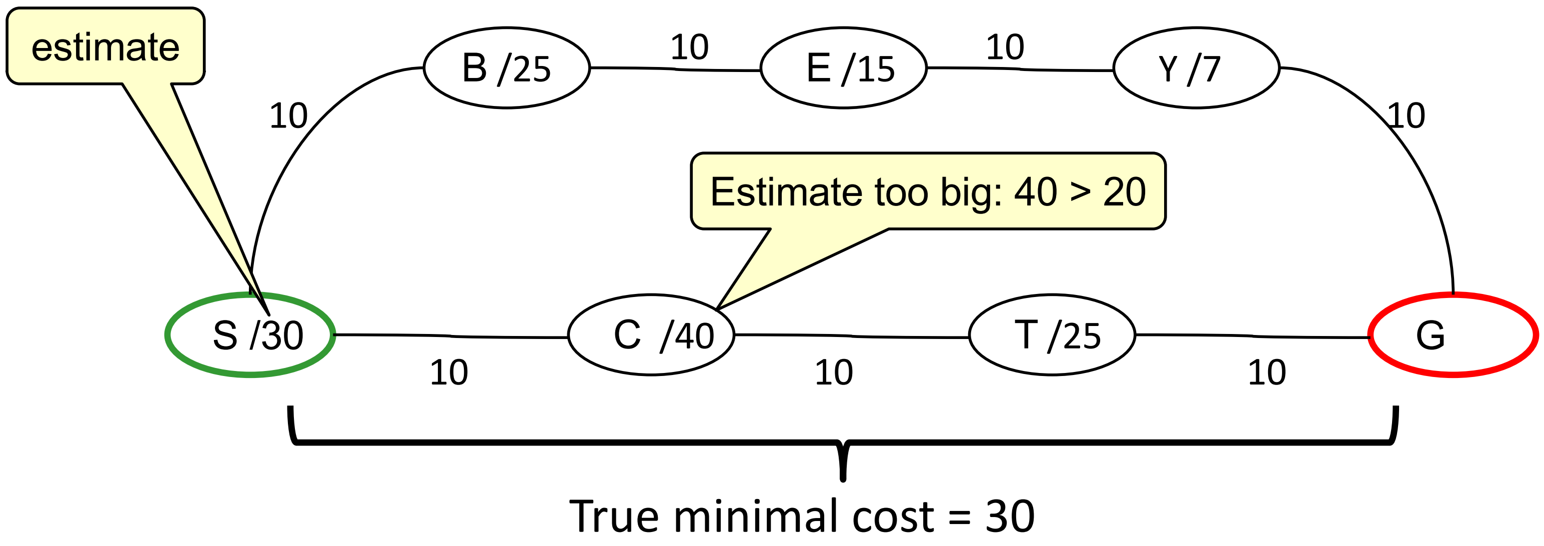
- The path found for by A* Search is the shortest path from the start node to the goal node if the following conditions are satisfied.
  1. The estimated cost to goal is never greater than the true cost (admissible heuristic)
  2. For each node, the first expand always has the minimal cost from start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



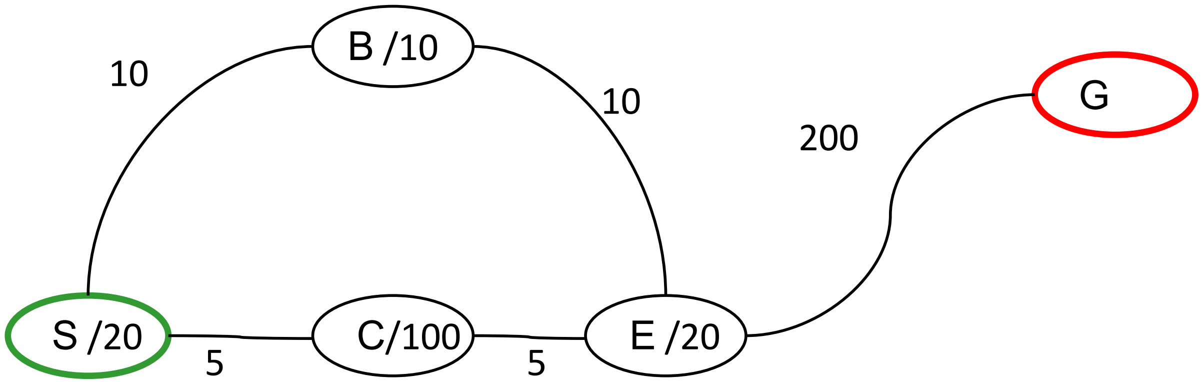
Consistent/Monotonic Heuristic

- 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>

Heuristics for A* Search

- The heuristic function has to be admissible and consistent/monotonic to make A* search be able to obtain optimal solution
  - Admissible: can stop the algorithm after visiting the goal node
  - Consistent/Monotonic: no need to revisit the same node

- But how to design admissible and consistent/monotonic heuristic function?
  - `h=0` for all nodes (1-to-1 Dijkstra's algorithm)
  - If cost is distance: straight-line distance
  - If cost is time: straight-line distance over speed limit
  - …

- Why `h=0` is admissible and consistent?

Summary

- A* Search is more effective than Dijkstra's algorithm for 1-to-1 pathfinding
- Many real-world applications
  - not just paths: e.g. search for optimal loading of a truck
  - any optimisation problem where build up a solution as a series of steps.
  - works with implicit graphs (AI planning, COMP307)

- Conditions for success
  - Admissible heuristic: never overestimate
  - Consistent/Monotonic heuristic: f=g+h is monotonically non-descreasing
  - The key is to design heuristic function to meet the conditions