University of Central Lancashire
UCLan

CO2412 Computational Thinking

Tutorial 3.1 discussion
Shortest paths
Travelling salesman

Where opportunity creates success

Updated structure of the module

Upon successful completion of this module, a student will be able to:

1) Use methods including logic and probability to reason about algorithms and data structures;
2) Compare, select, and justify algorithms and data structures for a given problem;
3) Analyse the computational complexity of problems and the efficiency of algorithms;
4) Use a range of notations to represent and analyse problems;
5) Implement and test algorithms and data structures.

program analysis

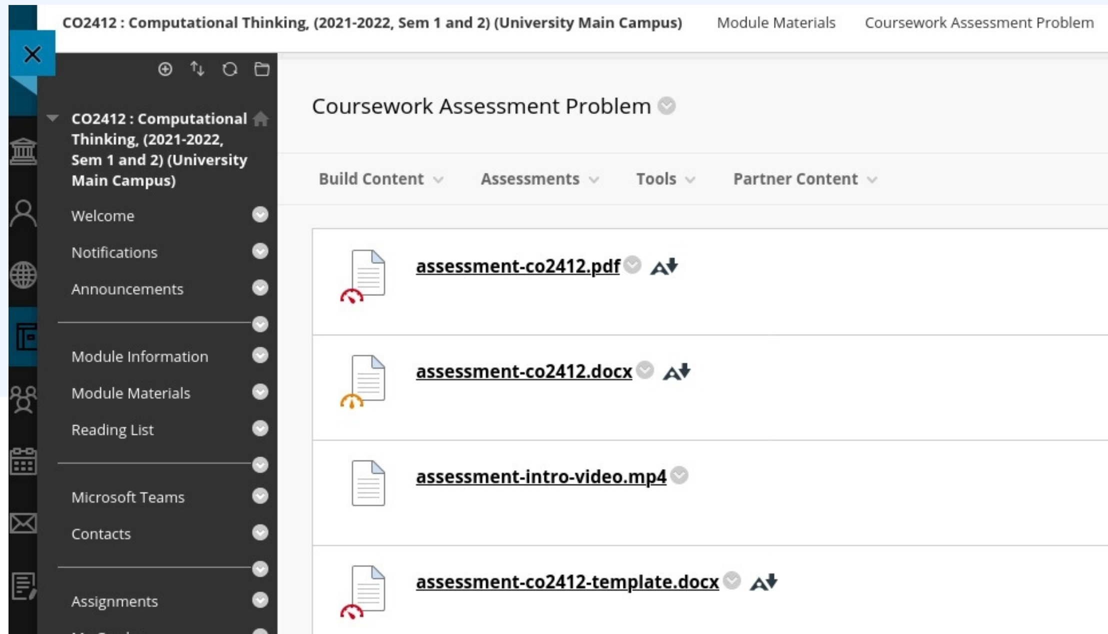
algorithm design

graphs and trees

logic and complexity

randomness and probability

Assessment update: Template document



Sections of this document:

- Cover Sheet (this page)
- Part A: Exploration and Discussion (max. 4 pages)
- Part B: Selection and Design (max. 4 pages)
- Part C: Asymptotic Efficiency (max. 4 pages)
- Part D: Correctness (max. 4 pages)
- Part F: Validation (max. 2 pages)
- Part G: Performance Measurement (max. 2 pages)
- Part H: Documentation (max. 2 pages)
- References (literature list containing all the cited academic literature)

(Part E, Implementation, is not included. It should consist of the code solving the problem. An archive – in zip, tar.bz2, tar.gz or any other widespread format – containing the implementation should be submitted as a separate file. That archive should also include the code used to validate the implementation and to measure its performance.)

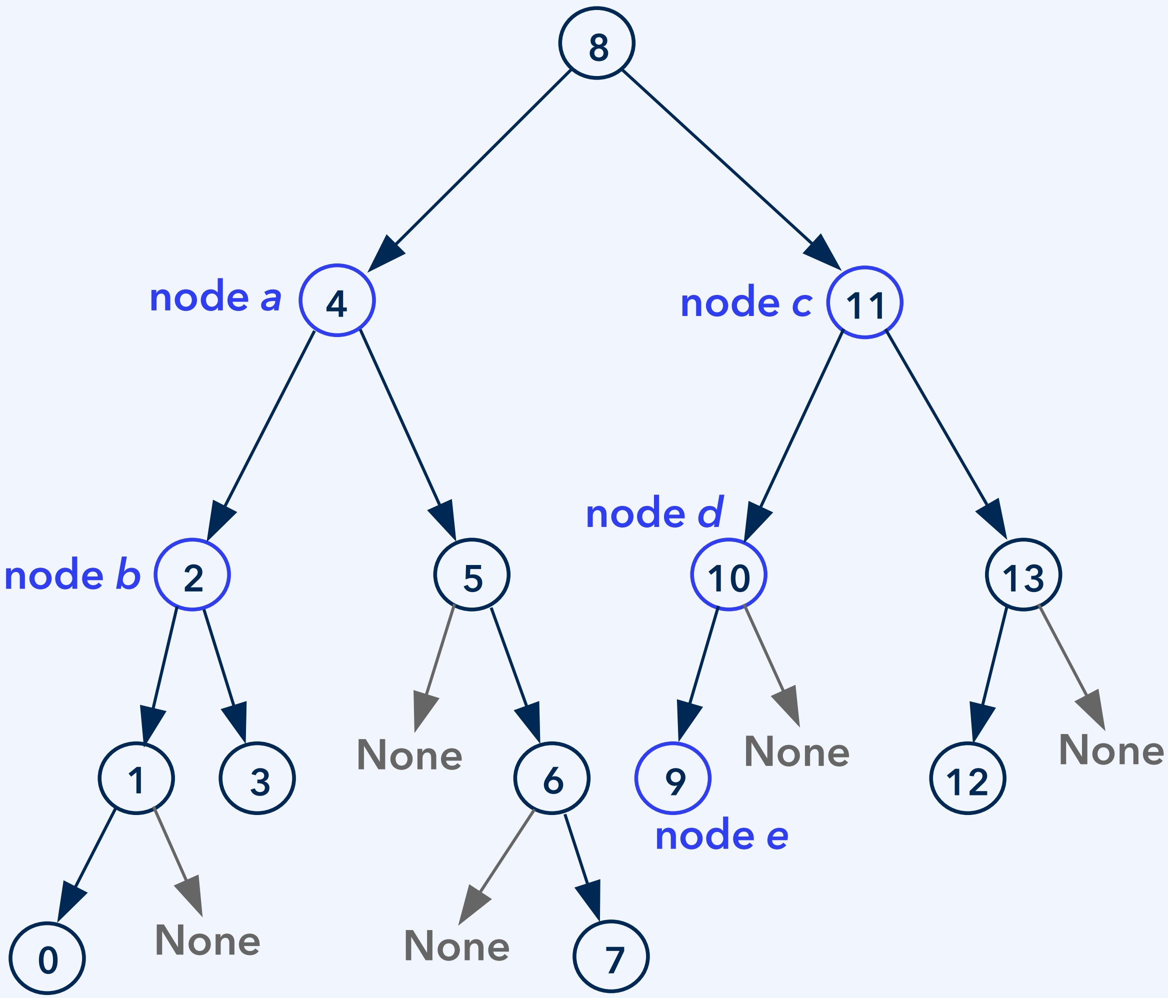
(You are welcome to submit documents that are shorter than the indicated maximum length. The maximum length is by no means to be understood as a recommended length.)

Tutorial 3.1 discussion

CO2412

18th January 2022

Deleting an element from a binary search tree



Example task

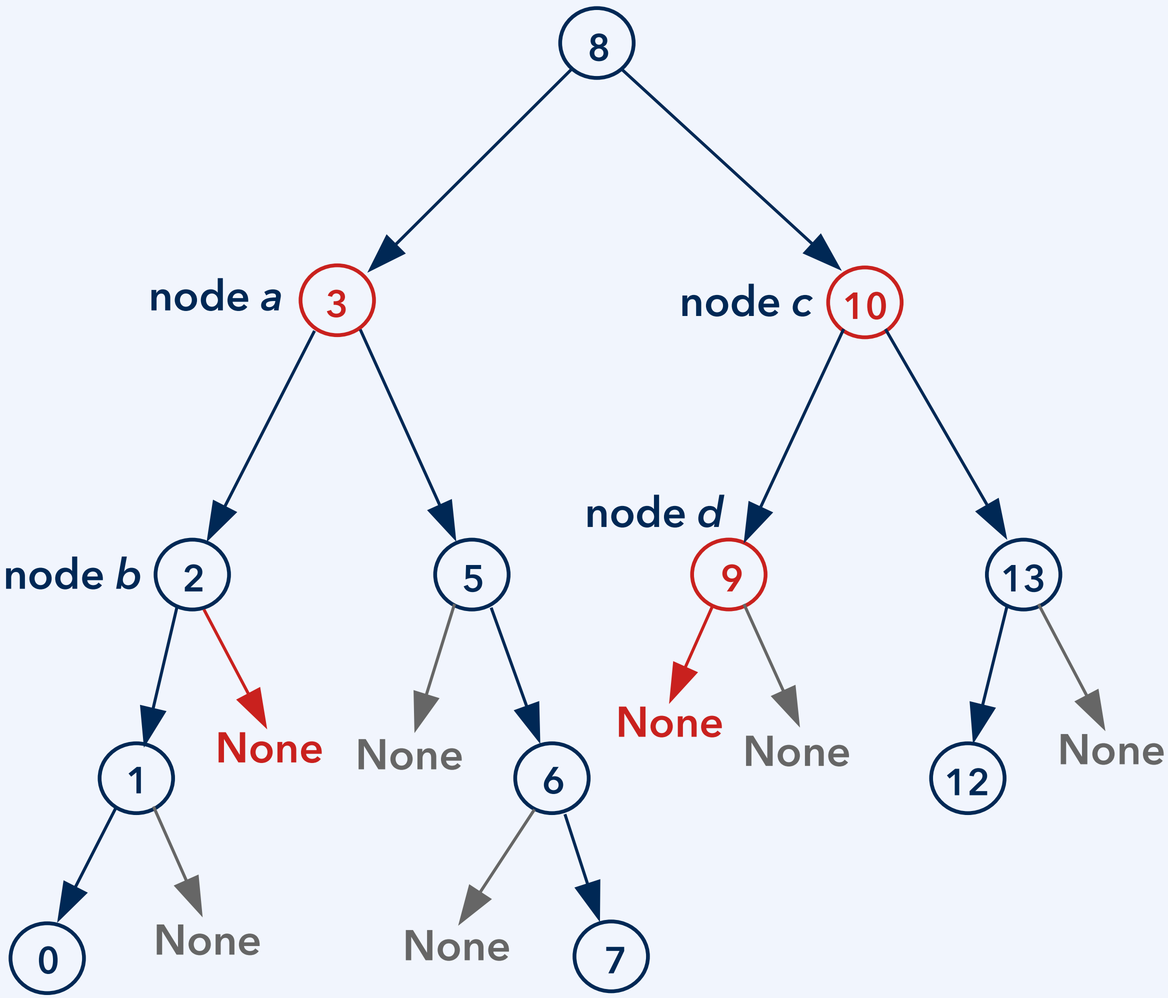
Delete 4 from the tree:

- Replace the label of node *a*, which is initially 4, with 3

Delete 11 from the tree:

- Replace the label of node *c* with 10

Deleting an element from a binary search tree



Example task

Delete 4 from the tree:

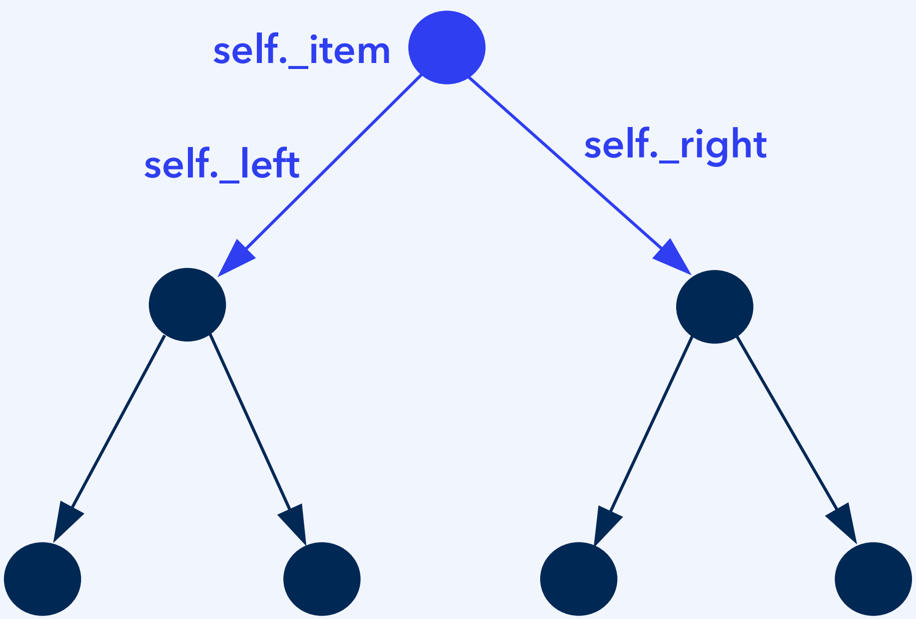
- Replace the label of node *a*, which is initially 4, with 3
- Then delete 3 from the subtree/node *b*

Delete 11 from the tree:

- Replace the label of node *c* with 10
- Then delete 10 from node *d*, by writing 9 to node *d* and erasing node *e*

Deleting an element from a binary search tree

See the Jupyter Notebook **bst-with-deletion**.



Method delete(self, value):

Is value smaller than self._item?

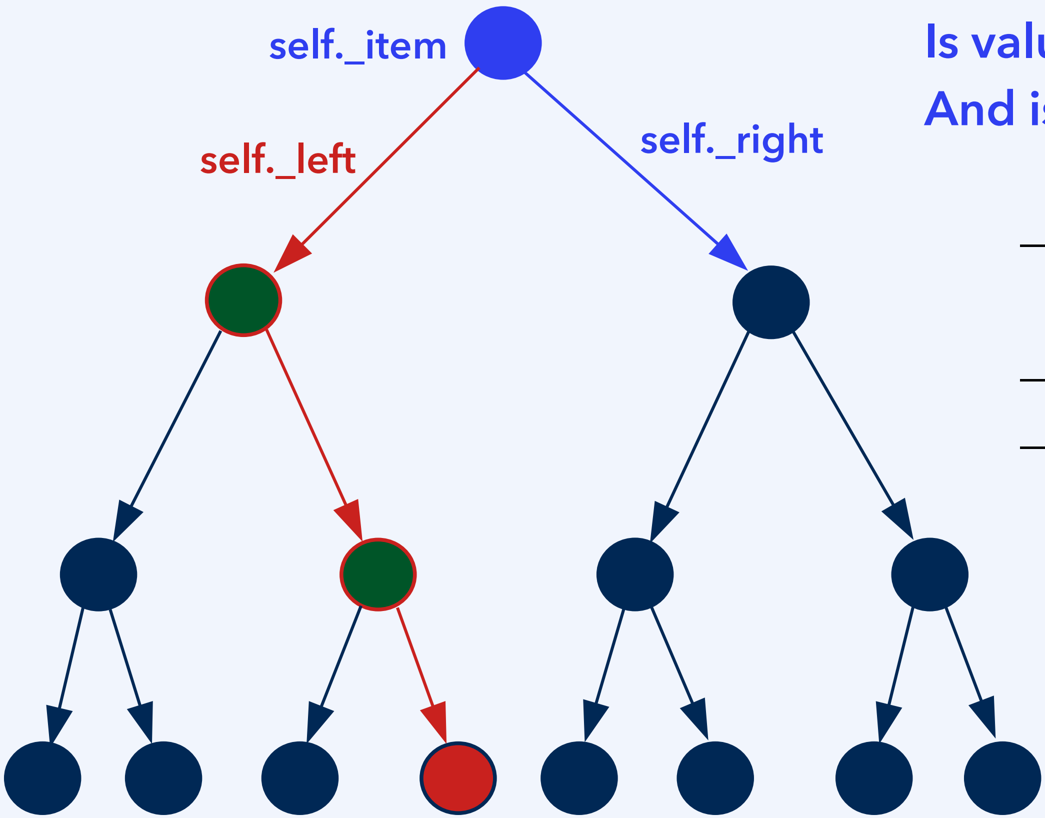
- If self._left is None, return
- self._left.delete(value)
- If self._left._item is now None, detach via self._left = None

Is value greater than self._item?

- If self._right is None, return
- self._right.delete(value)
- If self._right._item is now None, detach via self._right = None

Deleting an element from a binary search tree

See the Jupyter Notebook **bst-with-deletion**.

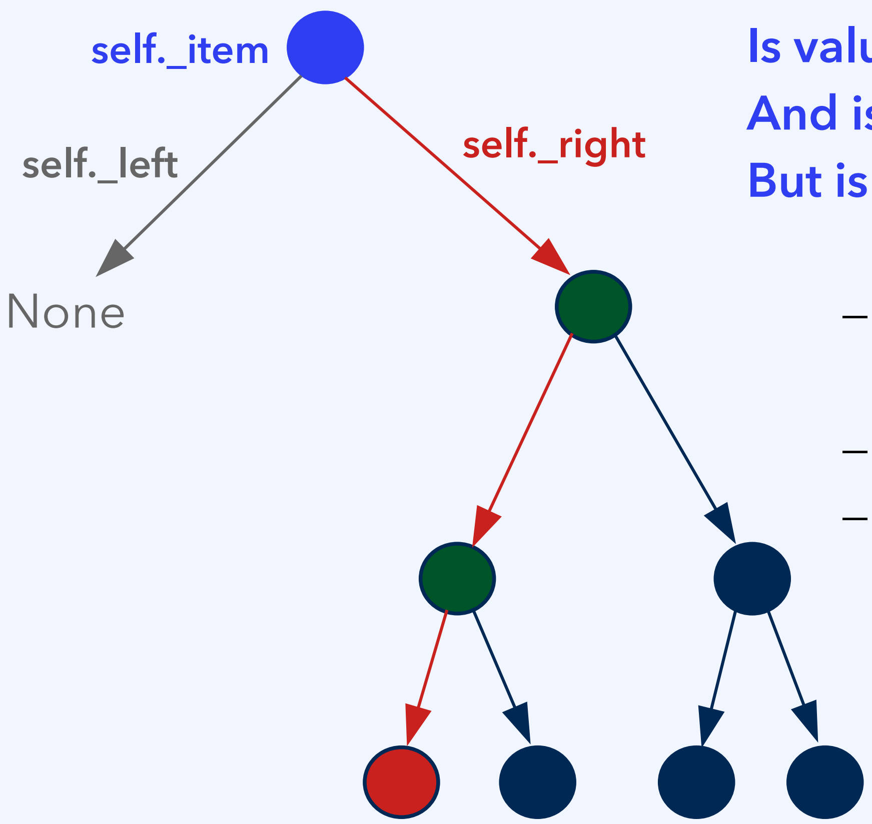


Is value the same as self._item?
And is self._left not None?

- Find the greatest element x from the left branch, set self._item = x
- Now, self._left.delete(x)
- If self._left._item is now None, detach via self._left = None

Deleting an element from a binary search tree

See the Jupyter Notebook **bst-with-deletion**.

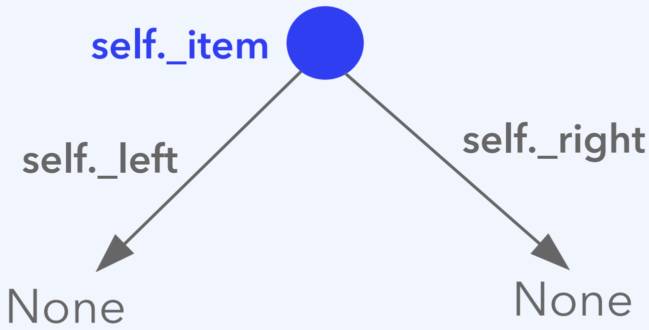


Is value the same as self._item?
And is self._left None?
But is self._right not None?

- Find the smallst element x from the right branch, set self._item = x
- Now, self._right.delete(x)
- If self._right._item is now None, detach via self._right = None

Deleting an element from a binary search tree

See the Jupyter Notebook **bst-with-deletion**.



Is value the same as self._item?
And is self a leaf?

- Delete value by setting self._item = None

University of Central Lancashire
UCLan

Shortest paths

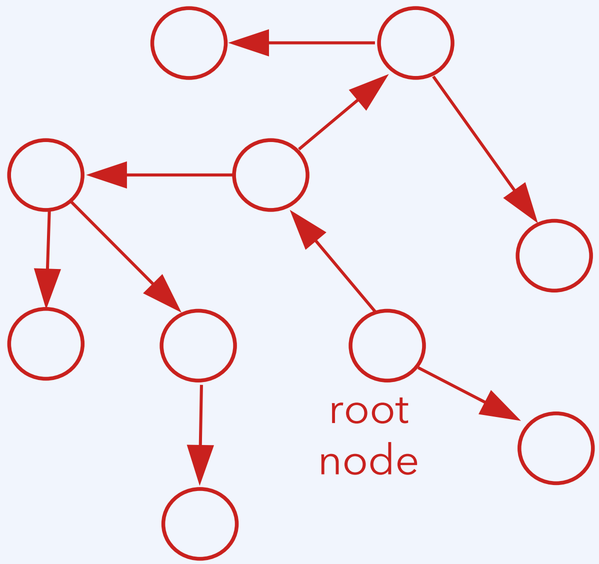
CO2412

18th January 2022

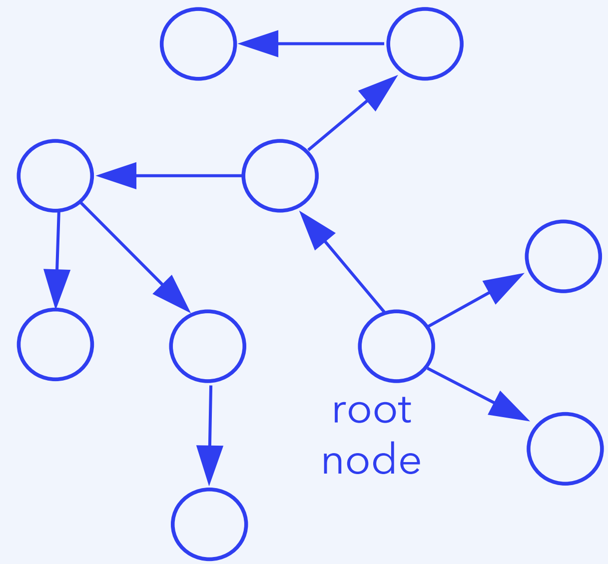
Graph traversal and spanning trees

A graph that is not a tree can be reduced to a tree by *eliminating edges*. Such a tree is called a **spanning tree** if it **covers all nodes**. When needed, this is often done by DFS or BFS, retaining only the edges followed for visiting nodes.

DFS spanning tree



BFS spanning tree

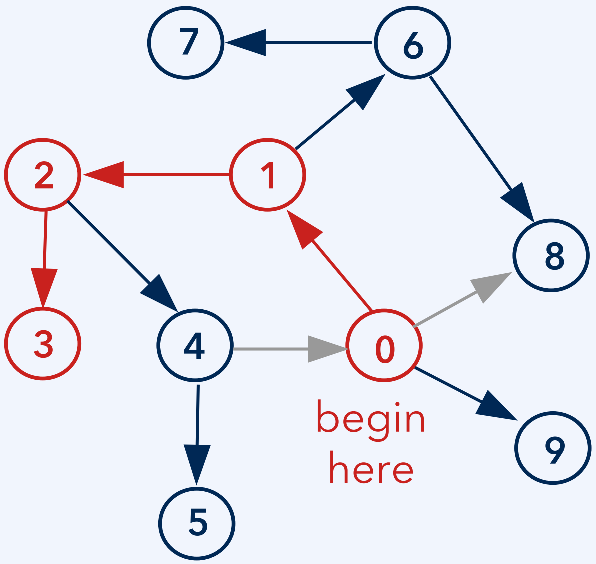


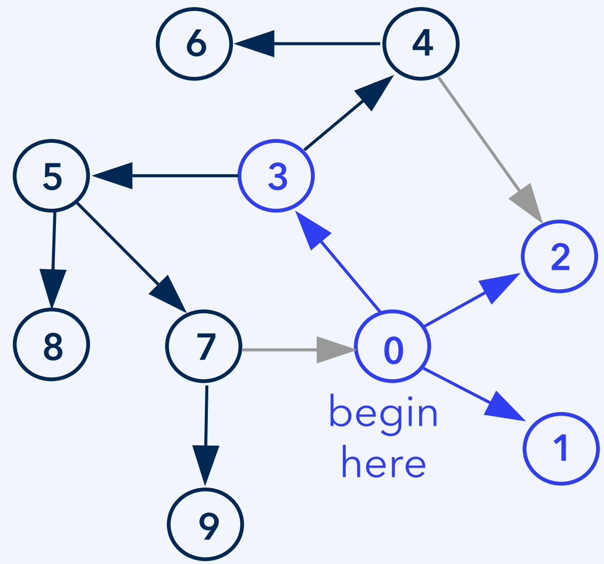
This construction is only feasible *if there are paths to all nodes from the root*.

Graph traversal and spanning trees

Traversal of trees and graphs: Depth-first search and breadth-first search

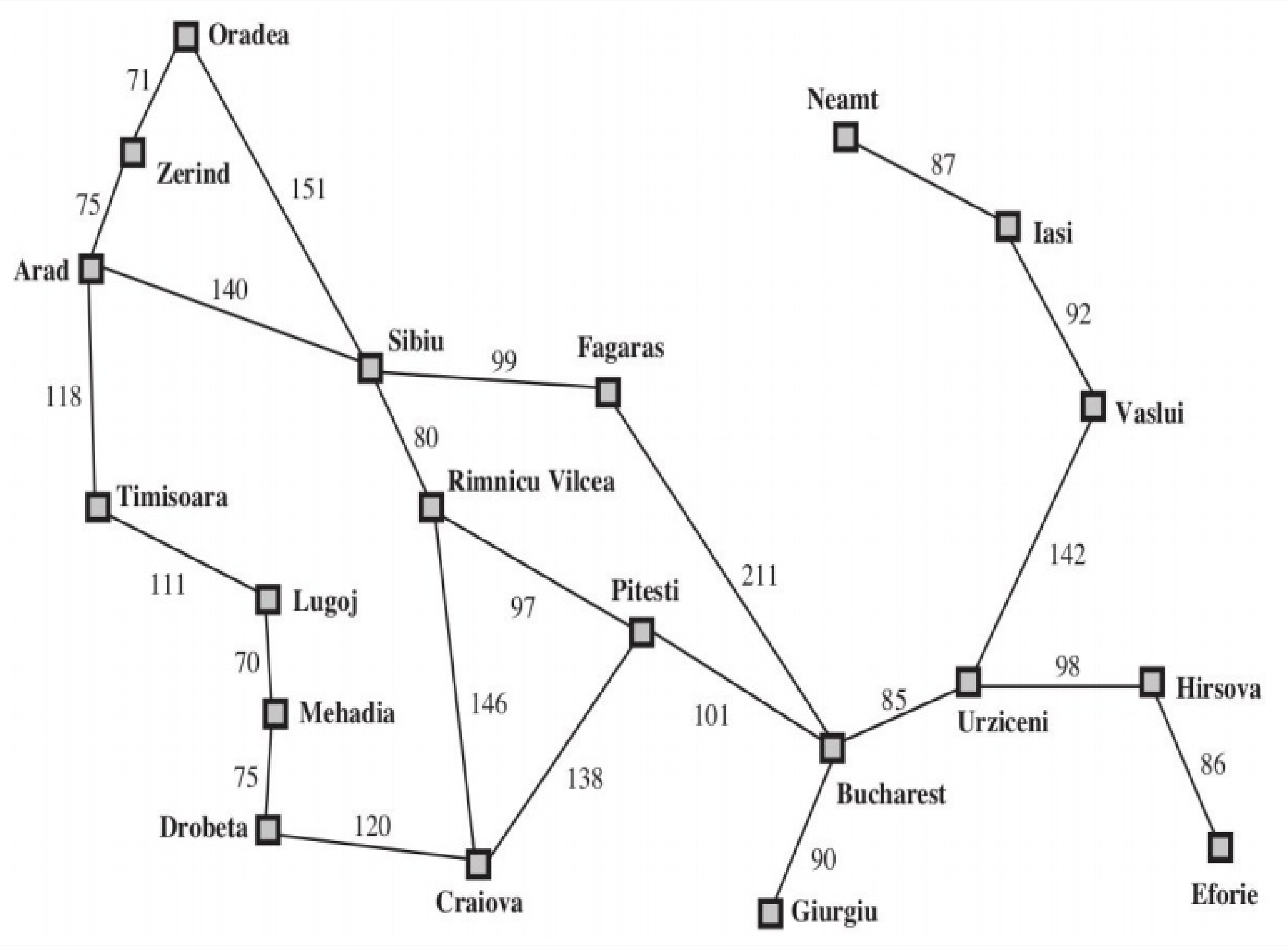
DFS always proceeds from the most recently detected node (LIFO).
BFS always proceeds from the node that was detected earliest (FIFO).



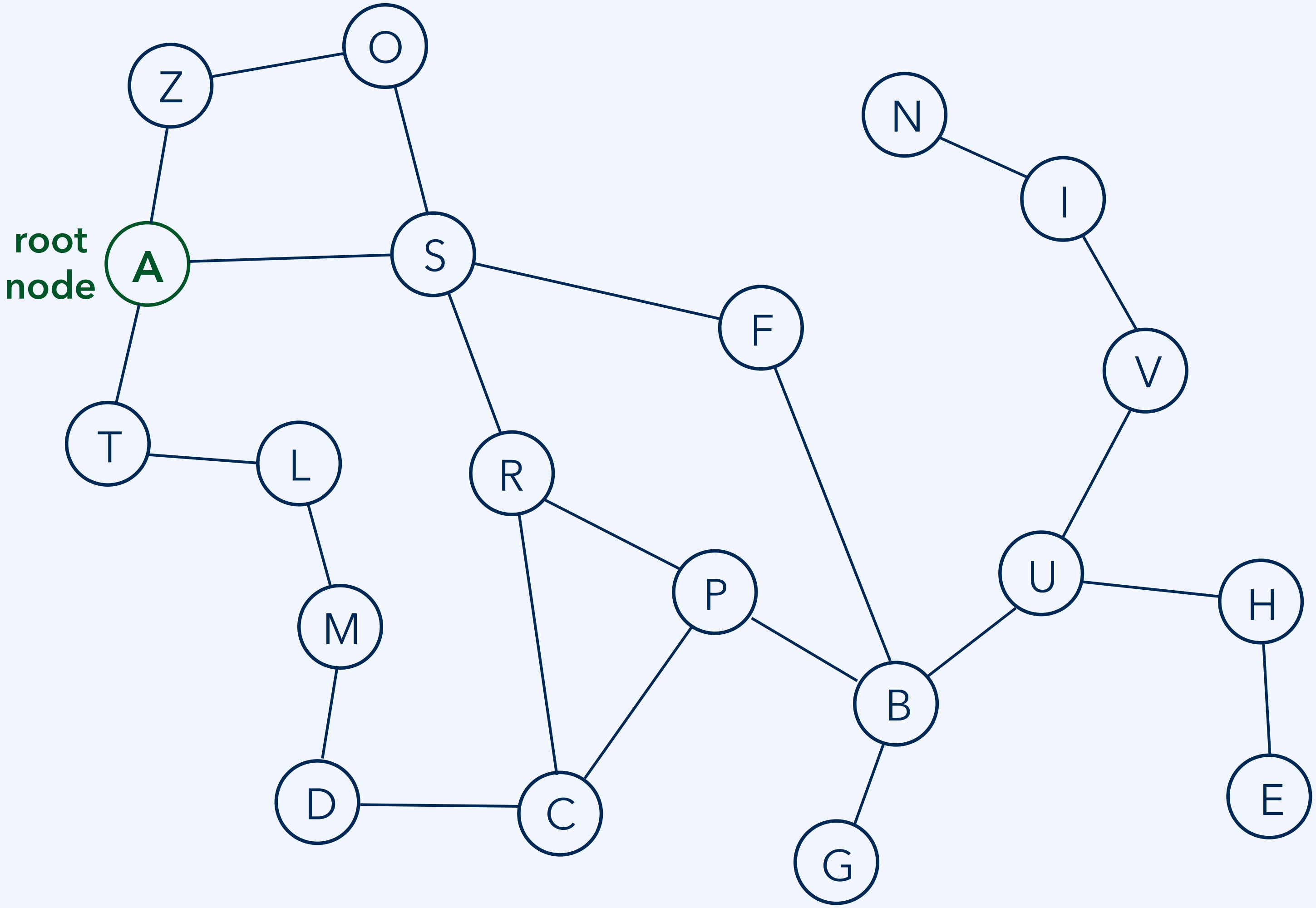


Note: Only elements *to which there is a path from the initial node* can be found.

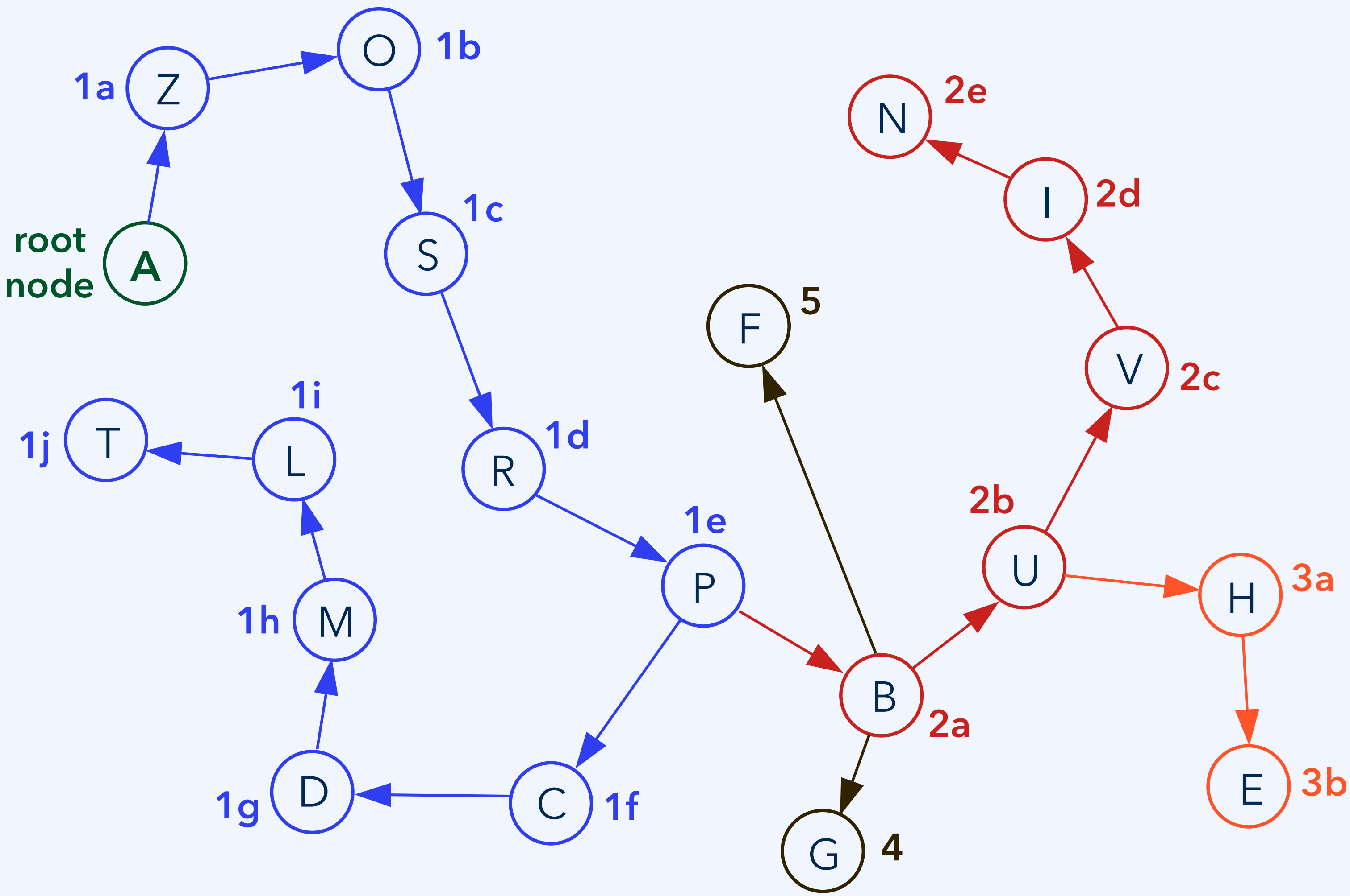
DFS and BFS spanning trees



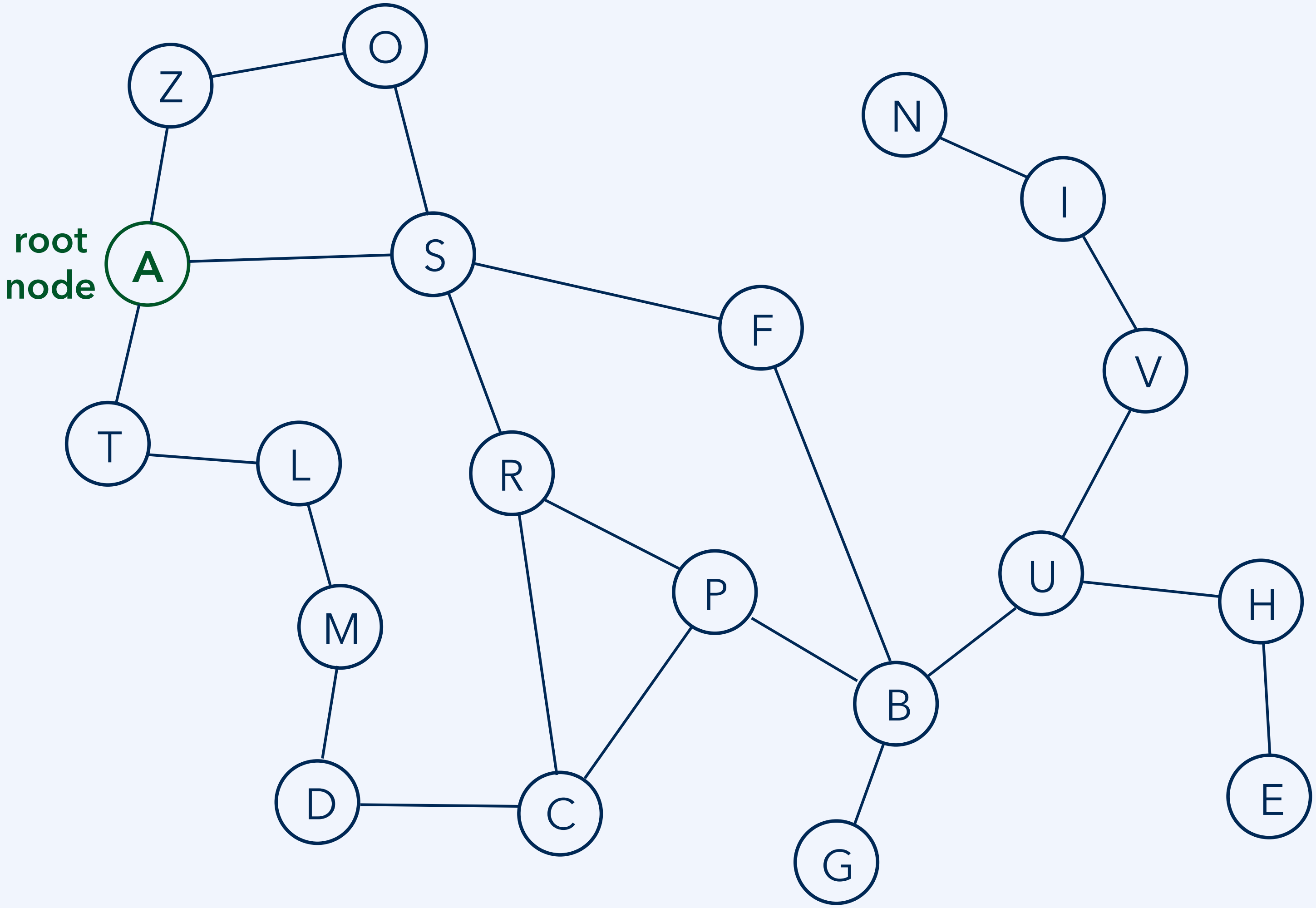
DFS and BFS spanning trees



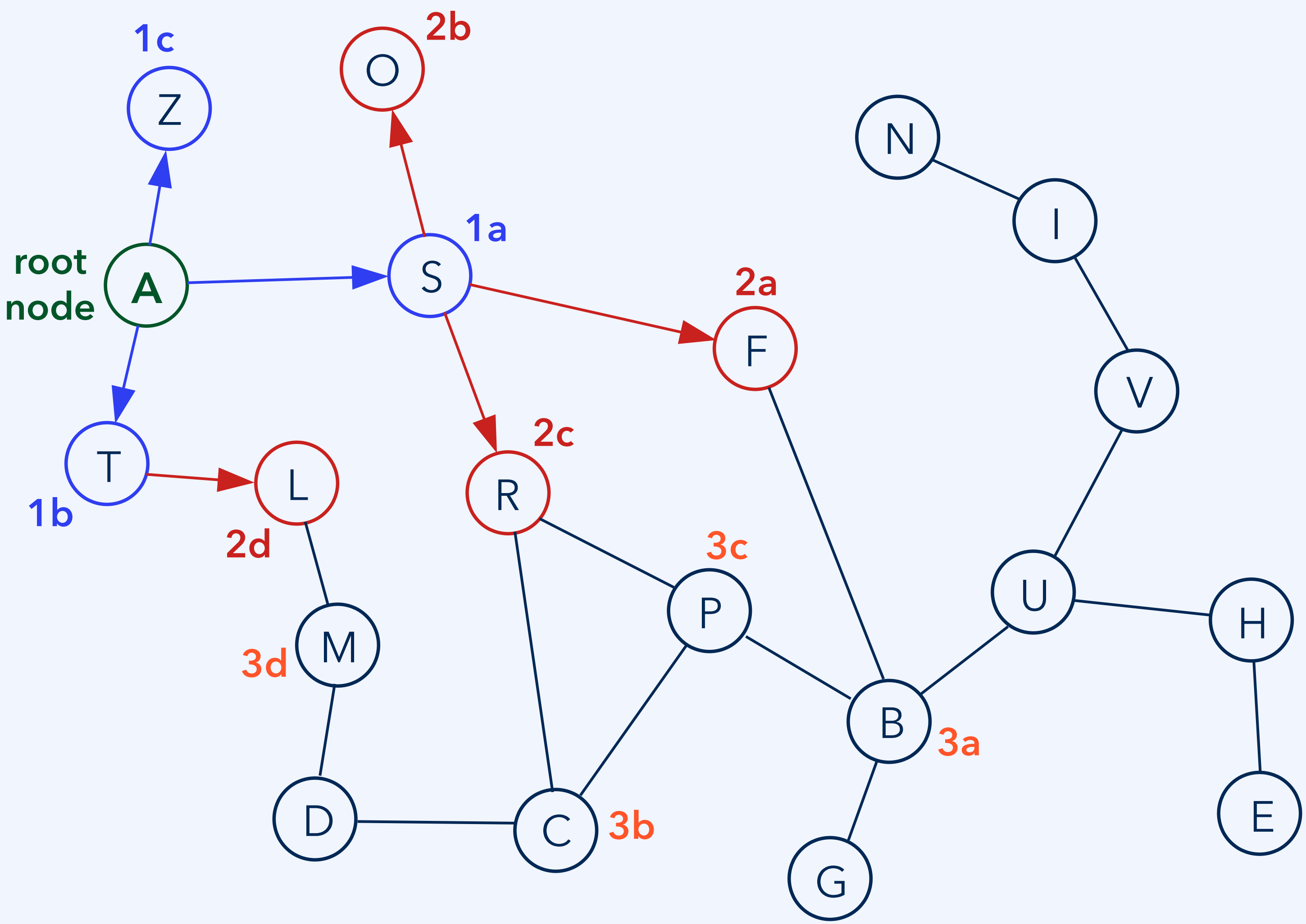
DFS spanning tree (also, “depth-first tree”)



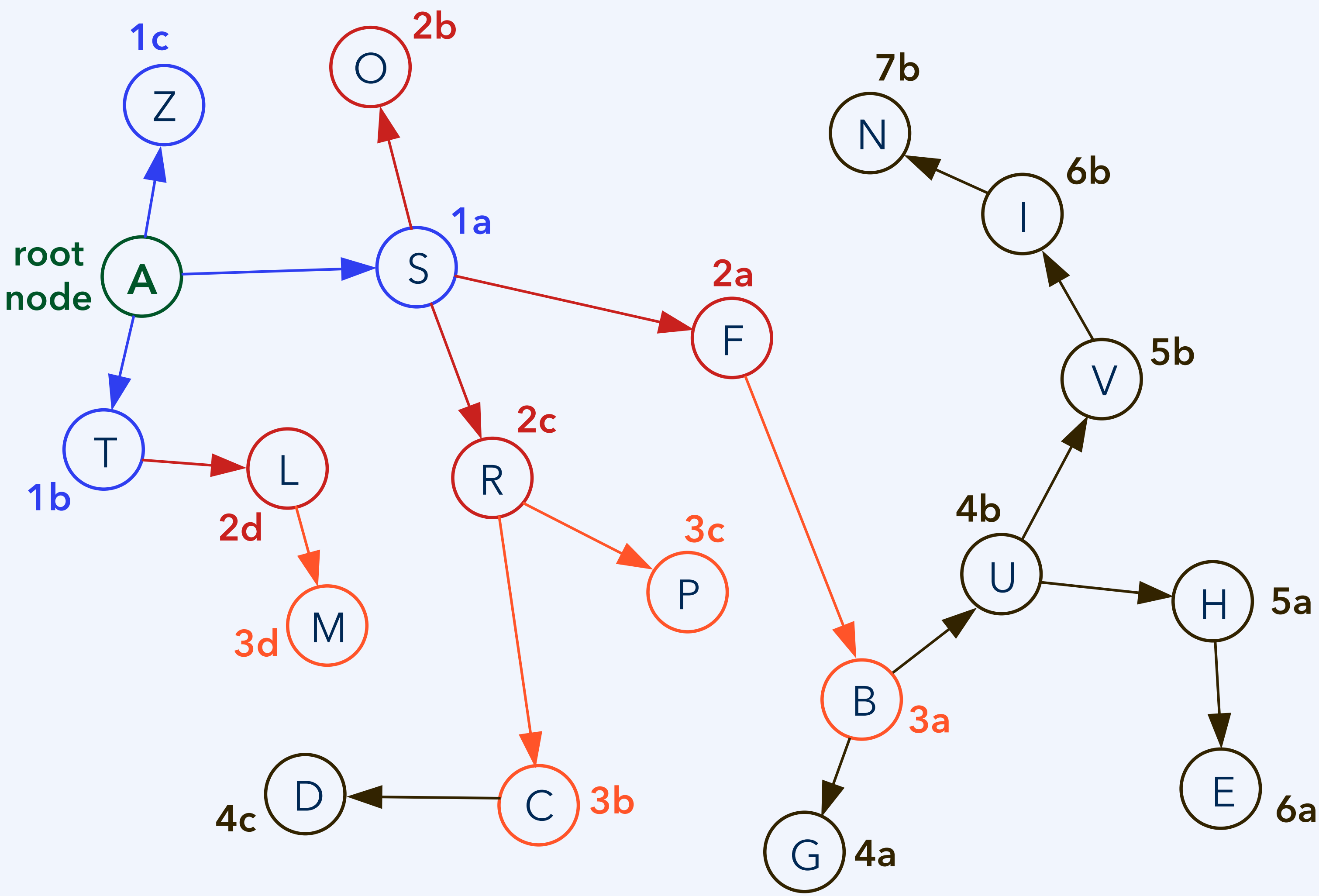
BFS spanning tree (also, "breadth-first tree")



BFS spanning tree (also, "breadth-first tree")



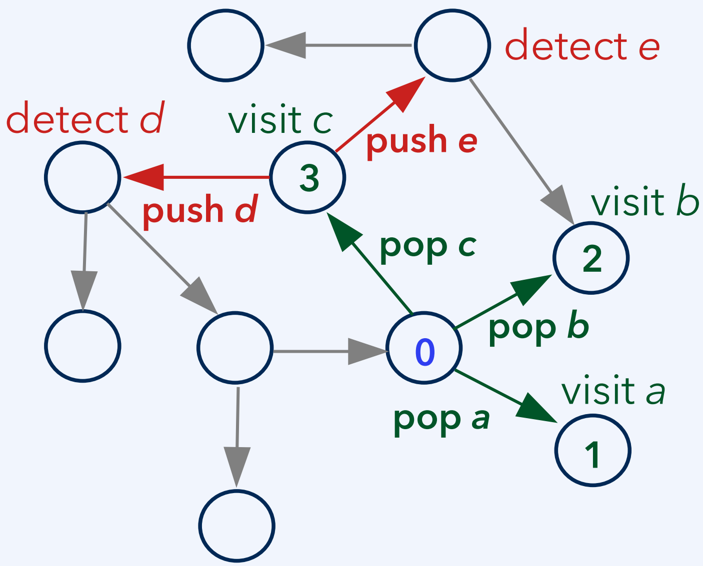
Shortest paths & distances from A to all other nodes



Time efficiency: Shortest paths (unweighted graphs)

Unweighted directed graph with ***n* nodes** and **e edges**, where $e \leq n^2$.

breadth-first search (BFS)



Traversal algorithm, one iteration:

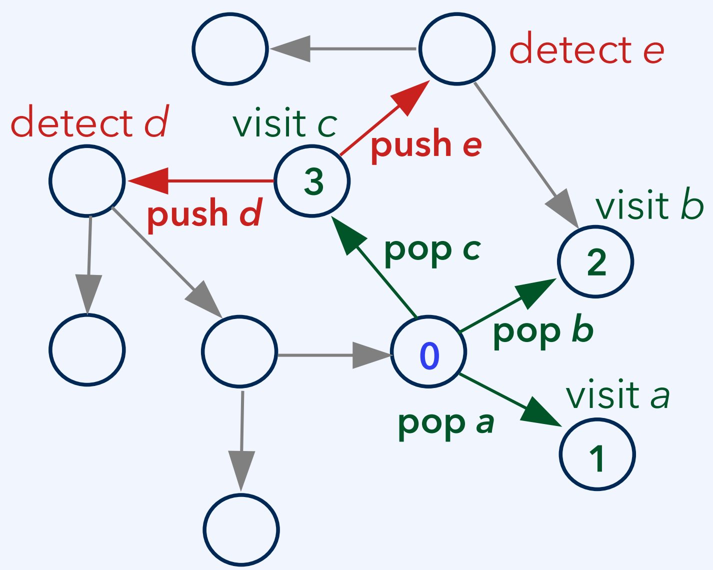
- **visit** present node
- detect nodes that can be reached directly from here (if undetected so far), **push** them to a **FIFO queue**
- **pop** node from **FIFO queue** of detected nodes and proceed there

O(1) time per edge, assuming a linked list is used for the queue and a list-like data structure (not an adjacency matrix) is used for adjacency/indicence data. (With an adjacency matrix, O(n) time per node is required to find the edges.)

Time efficiency: Shortest paths (unweighted graphs)

Unweighted directed graph with ***n* nodes** and **e edges**, where $e \leq n^2$.

breadth-first search (BFS)



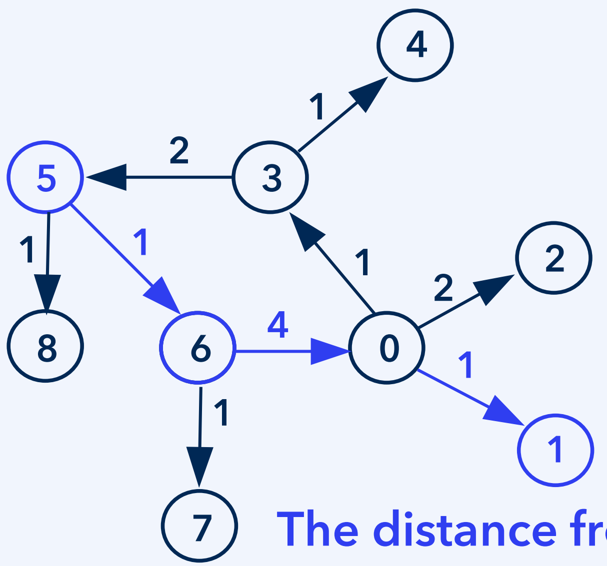
Traversal algorithm, one iteration:

- **visit** present node
- detect nodes that can be reached directly from here (if undetected so far), **push** them to a **FIFO queue**
- **pop** node from **FIFO queue** of detected nodes and proceed there

Overall **O(e) time**, where e is the number of edges, or $\mathbf{O(n^2)}$ in the worst case. For BFS beyond this use case, it is $O(n + e)$, which is usually the same as O(e). It also generally requires $O(n^2)$ time if an adjacency matrix is used.

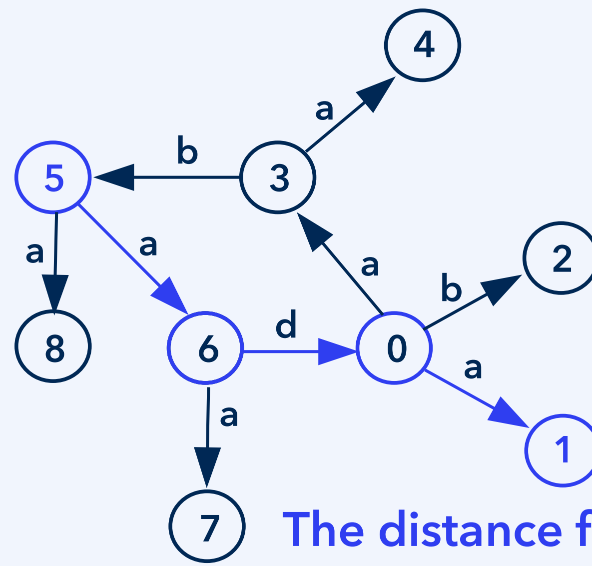
From unweighted to weighted graphs

weighted graph



The distance from node 5 to node 1 is 6 (from 1+4+1).

unweighted graph



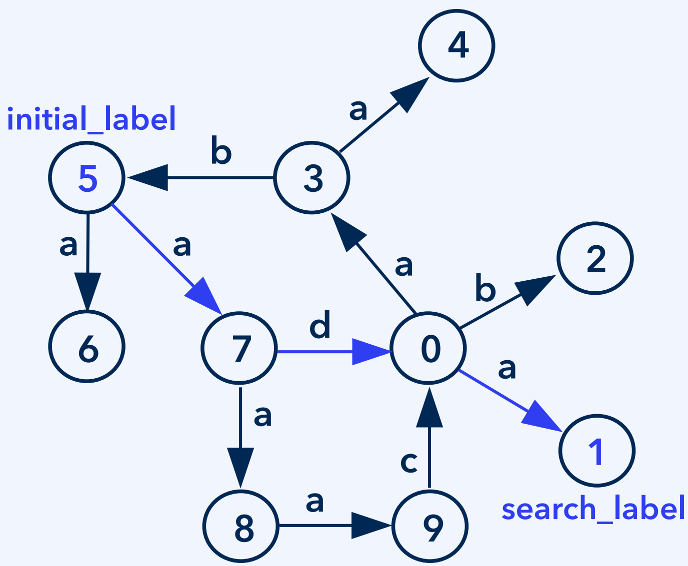
The distance from node 5 to node 1 is 3 (three edges).

In unweighted graphs, the distance between nodes is the number of edges.

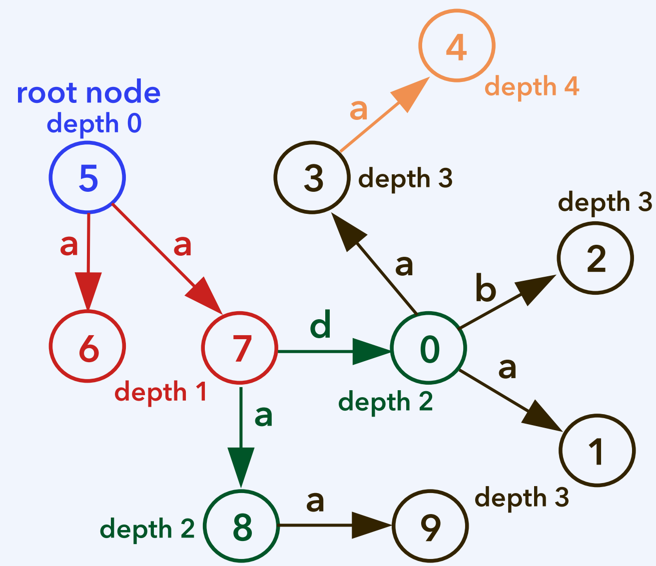
In weighted graphs, distances between nodes are obtained from edge labels.

From unweighted to weighted graphs

BFS starting from node 5



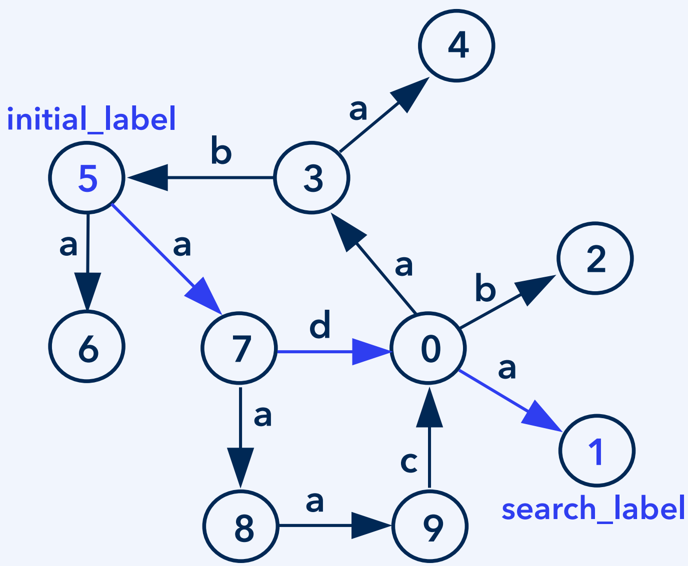
BFS spanning tree with the root at node 5



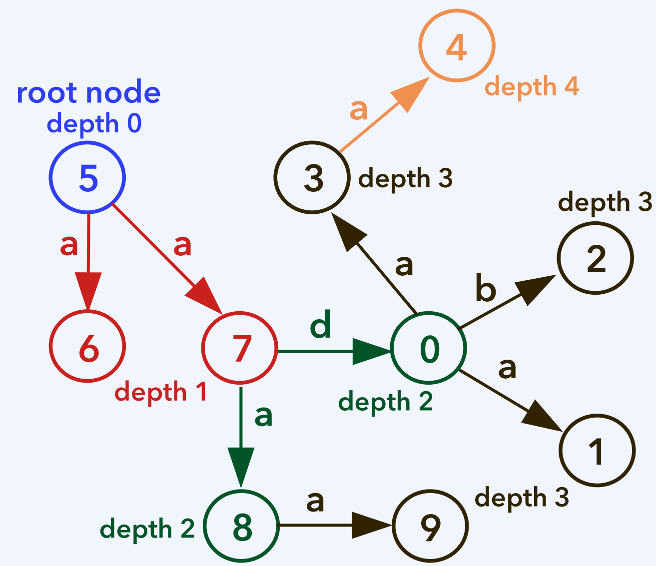
For unweighted graphs, the shortest paths and distances from one node to all other nodes can be computed by breadth-first search (BFS).

From unweighted to weighted graphs

BFS starting from node 5



BFS spanning tree with the root at node 5

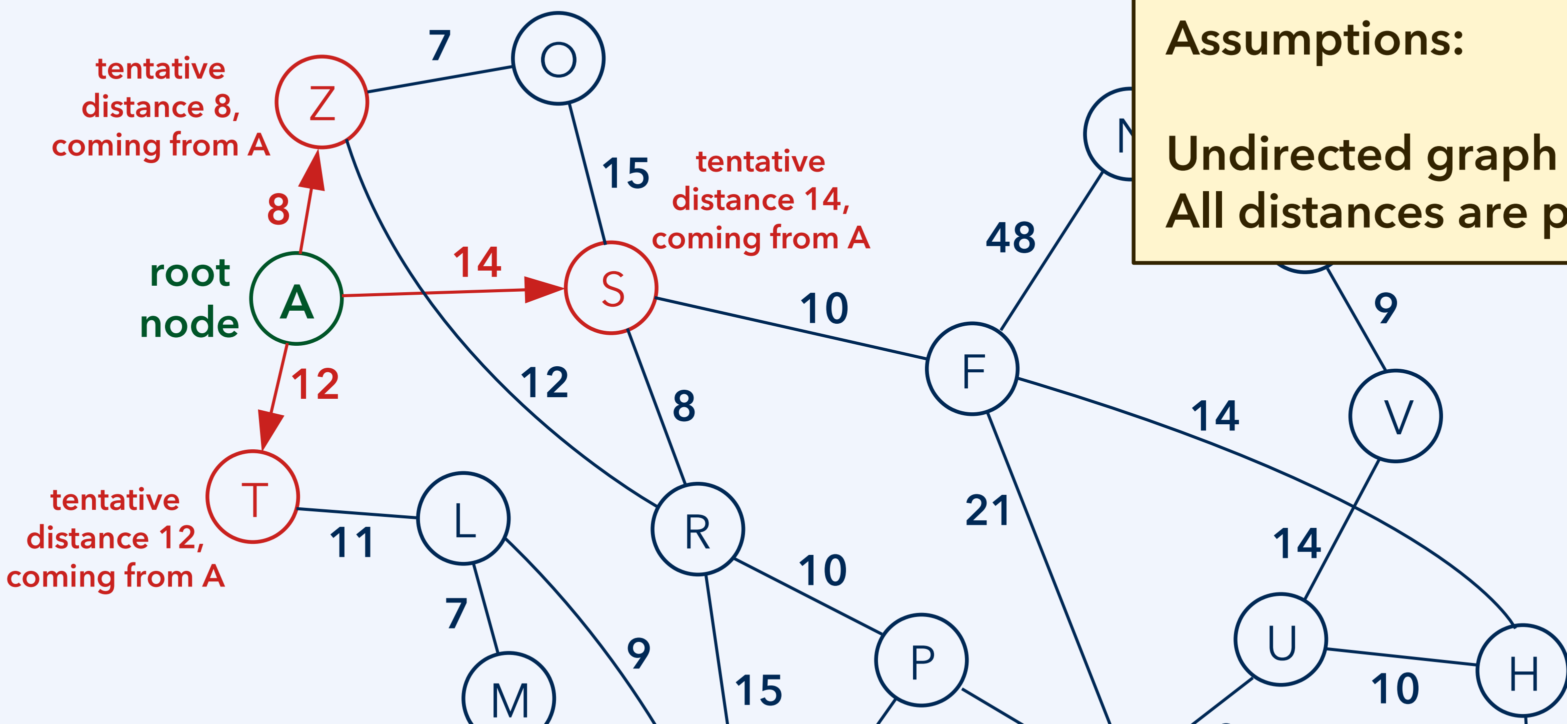


Features of the algorithm:

It is greedy. The spanning tree is constructed node by node, until complete.
Every time a node is added to the tree, we are sure to know the shortest path.

University of Central Lancashire
UCLan

Weighted graphs: Dijkstra's algorithm



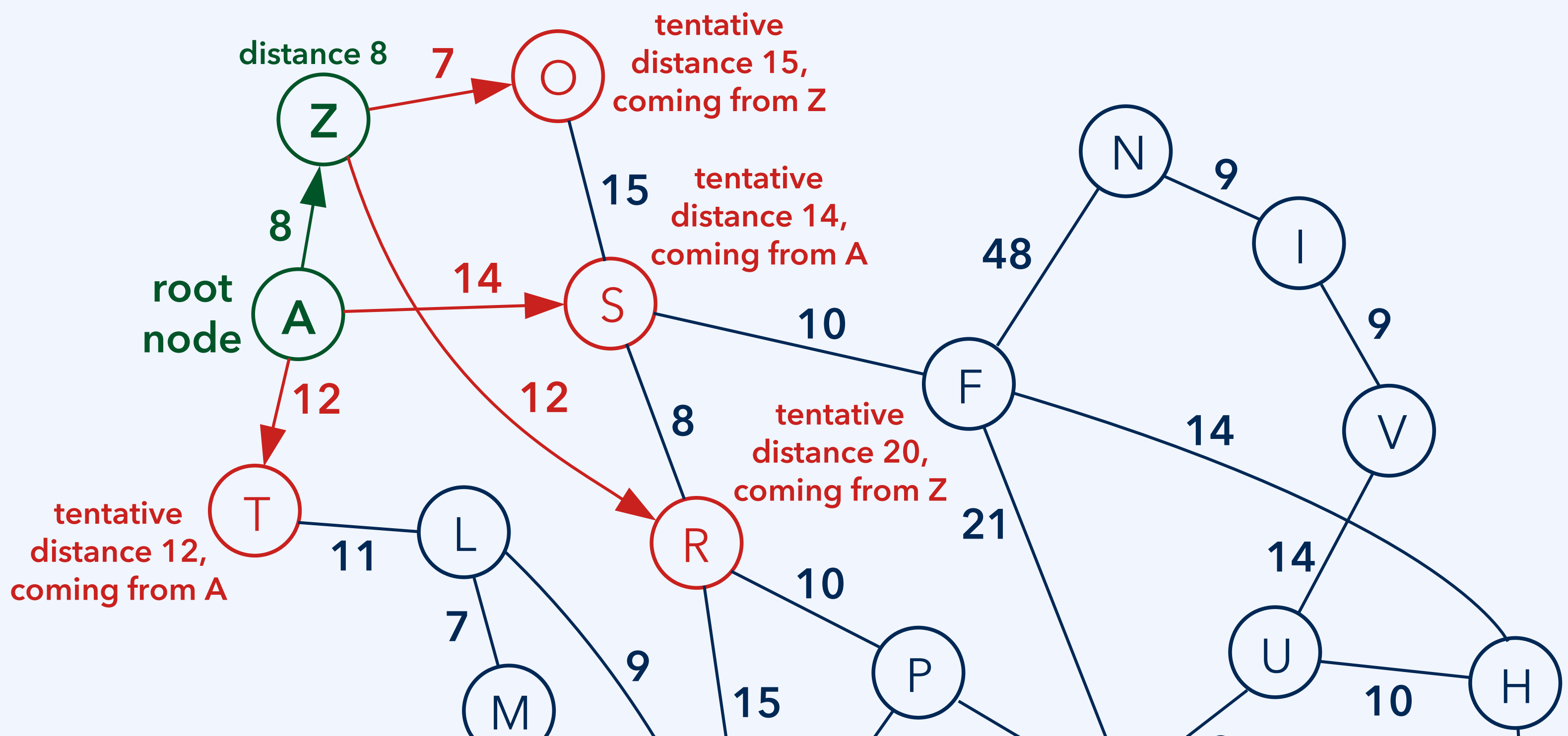
Assumptions:

Undirected graph
All distances are positive

Features of the algorithm:

It is greedy. The spanning tree is constructed node by node, until complete.
Every time a node is added to the tree, we are sure to know the shortest path.

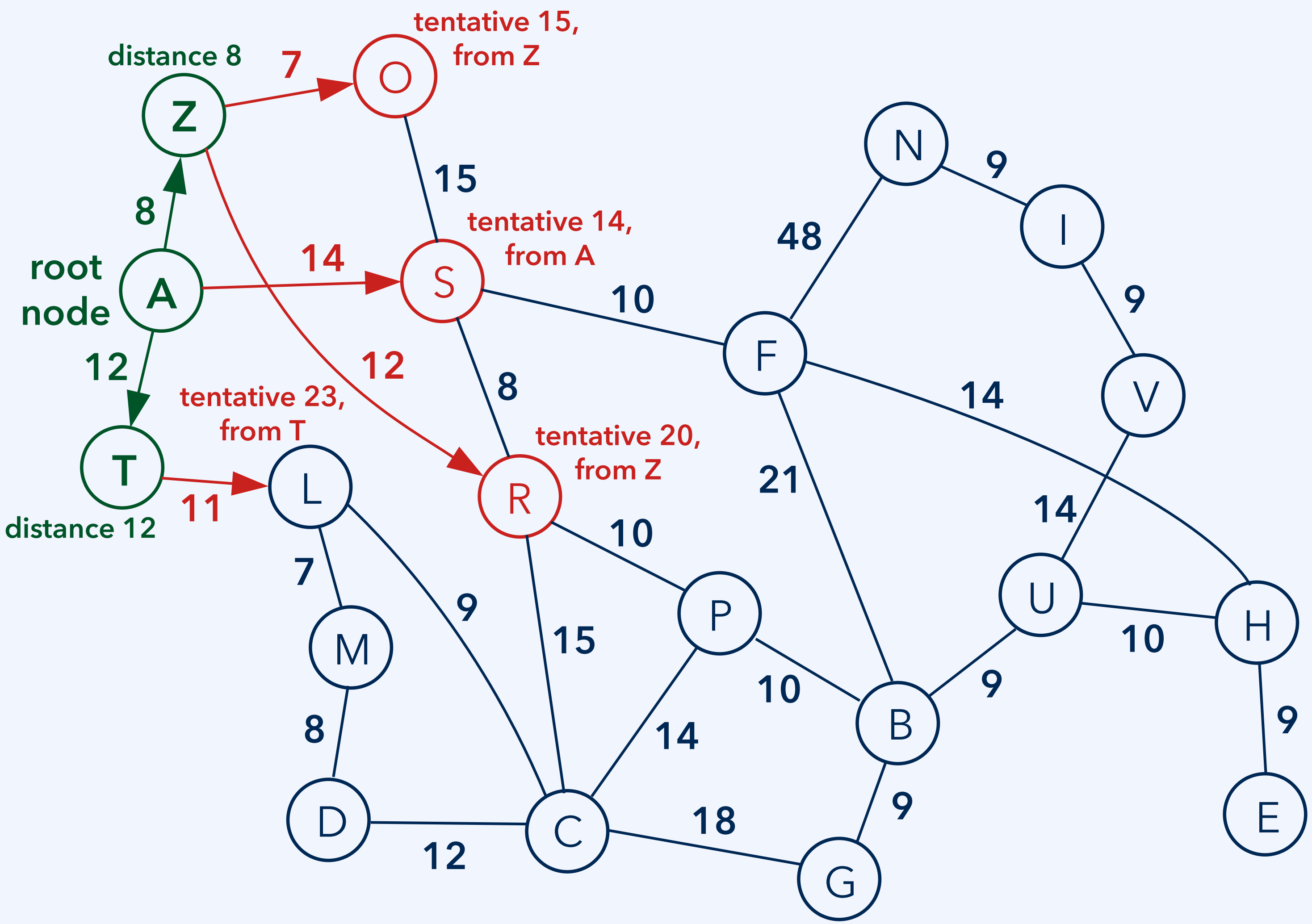
Weighted graphs: Dijkstra's algorithm



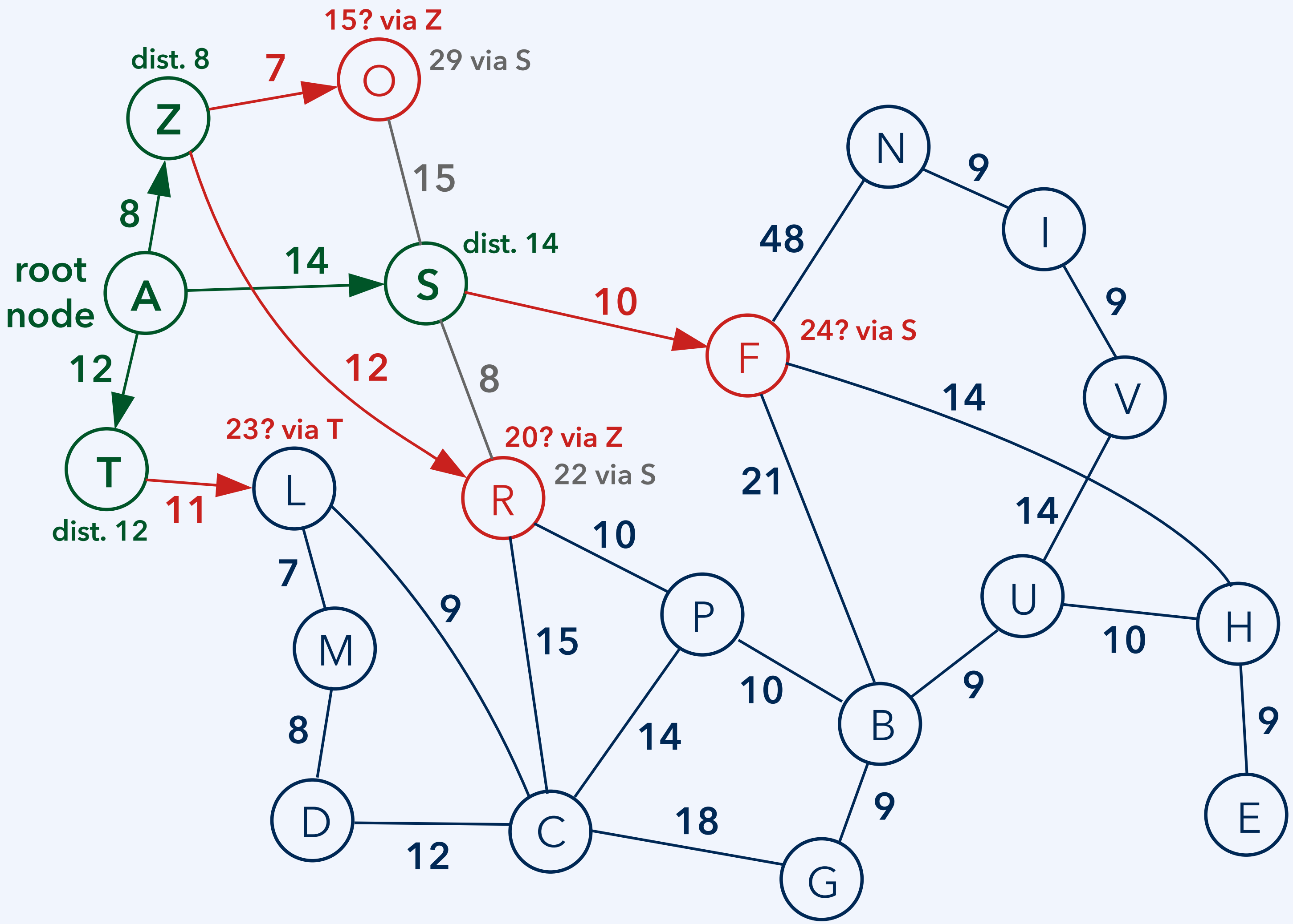
Features of the algorithm:

It is greedy. The spanning tree is constructed node by node, until complete.
Every time a node is added to the tree, we are sure to know the shortest path.

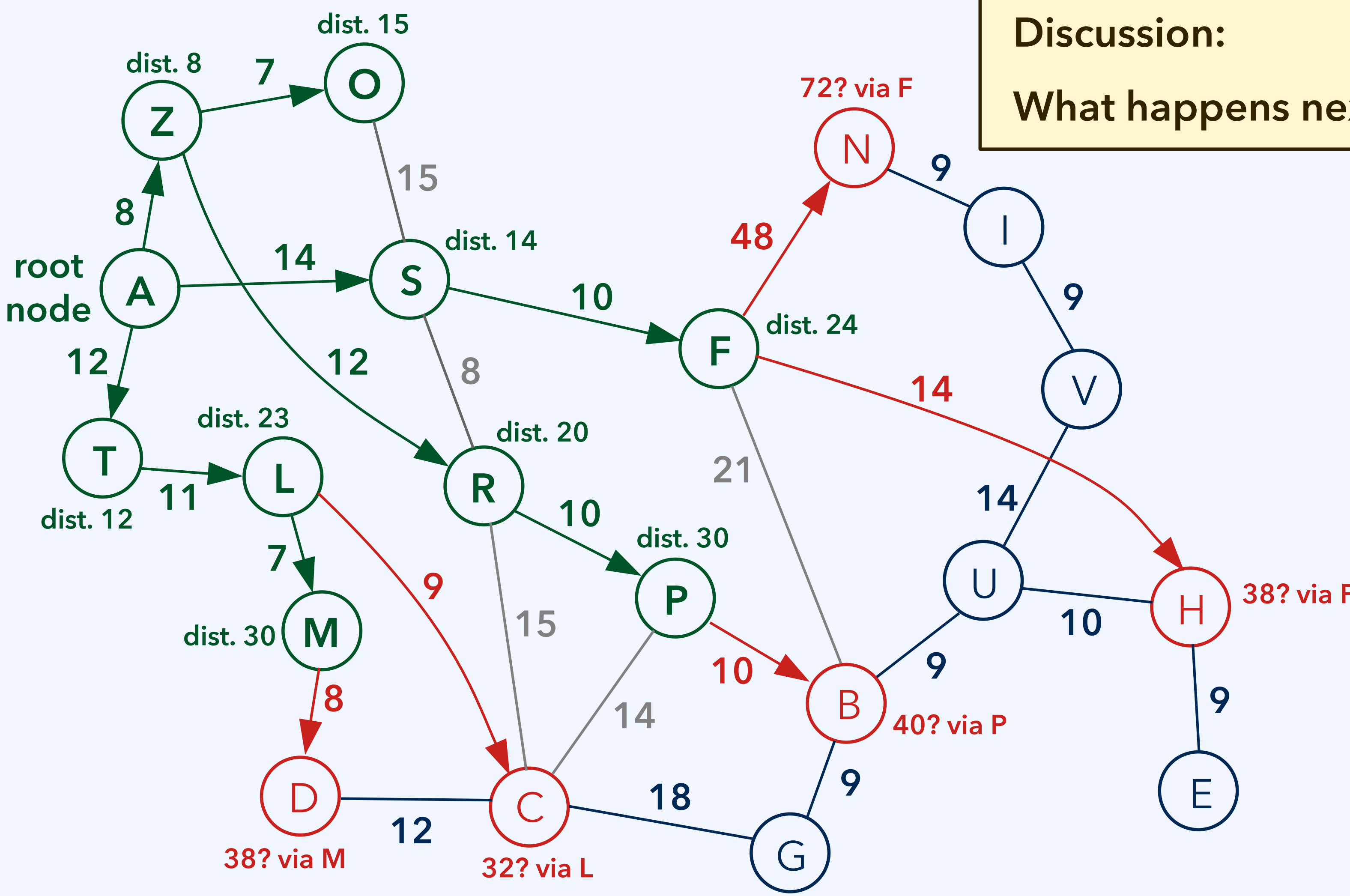
Weighted graphs: Dijkstra's algorithm



Weighted graphs: Dijkstra’s algorithm



Weighted graphs: Dijkstra's algorithm



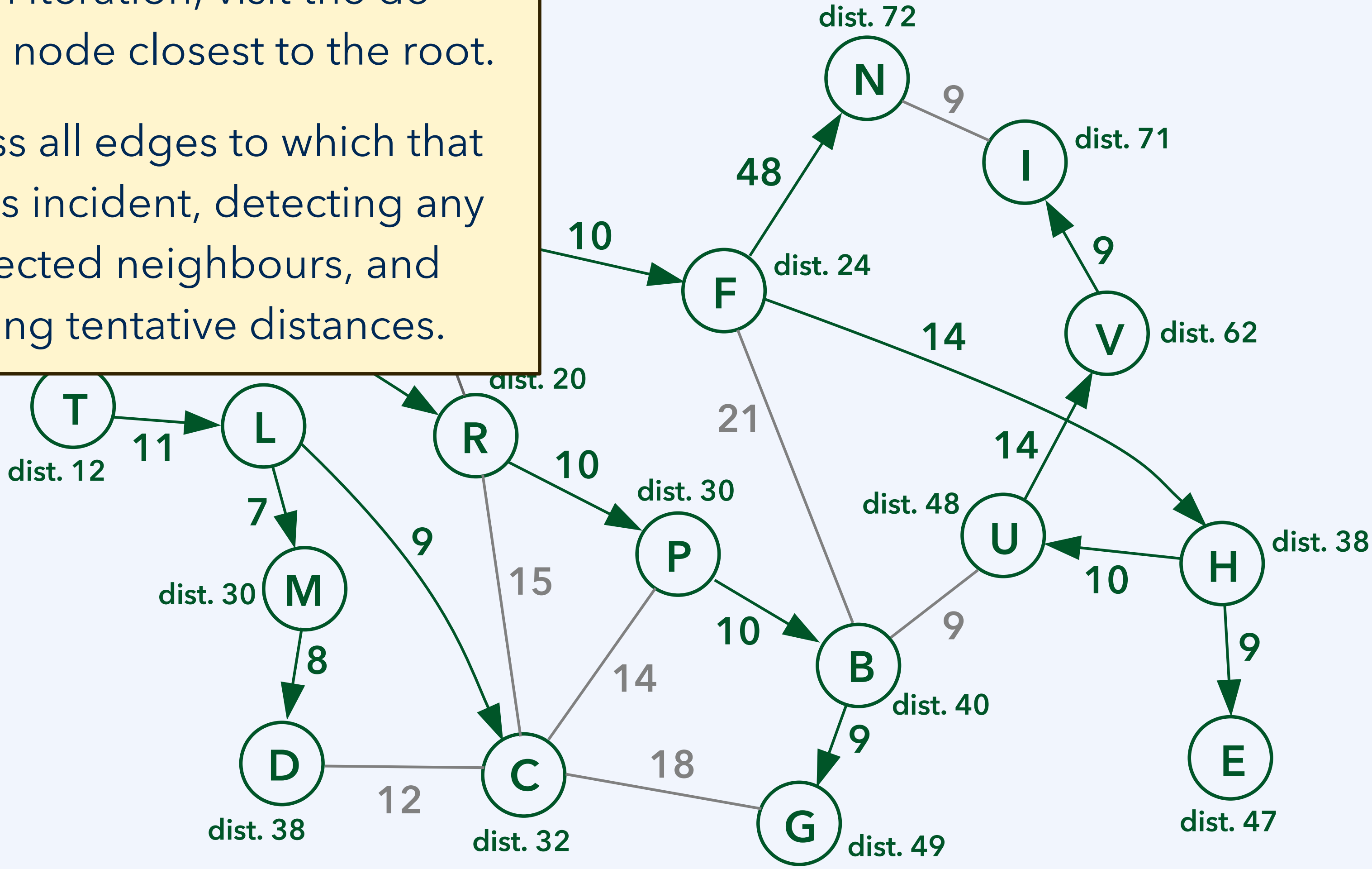
Discussion:

What happens next?

Weighted graphs: Dijkstra's algorithm

In each iteration, visit the detected node closest to the root.

Process all edges to which that node is incident, detecting any undetected neighbours, and updating tentative distances.

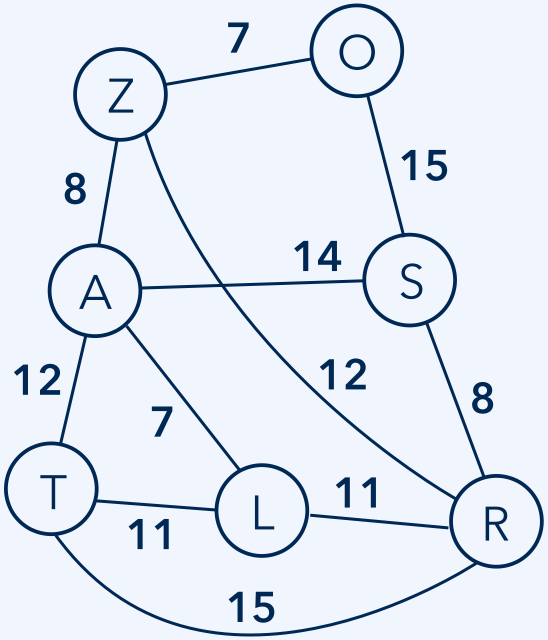


Travelling salesman

CO2412

18th January 2022

The travelling salesman problem (TSP)



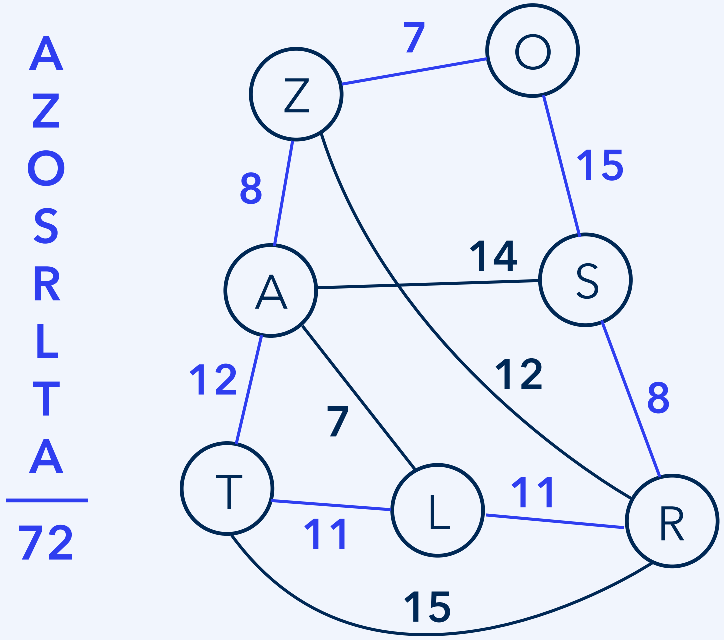
Scenario:

A travelling salesman needs to visit all the cities, by a path that ends at the same city where it starts (a **cycle**).

No city may be visited twice. Every city must be visited exactly once. (Except for returning to the start.)

The total travel distance, that is, the total length of the path, must be as short as possible.

The travelling salesman problem (TSP)



Scenario:

A travelling salesman needs to visit all the cities, by a path that ends at the same city where it starts (a **cycle**).

No city may be visited twice. Every city must be visited exactly once. (Except for returning to the start.)

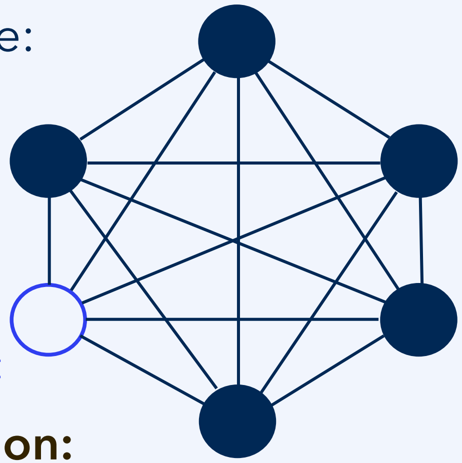
The total travel distance, that is, the total length of the path, must be as short as possible.

Discussion:

The cycle highlighted above has the length 8+7+15+8+11+11+12 = 72. Find an alternative route. How long is it?

The travelling salesman problem (TSP)

Example:
$n = 6$



Discussion:

How many cycles covering all nodes are there in a **complete graph** with n nodes, that is a graph where every node is adjacent to every other node?

How long would it then take to solve the TSP by a **brute force** algorithm?

Scenario:

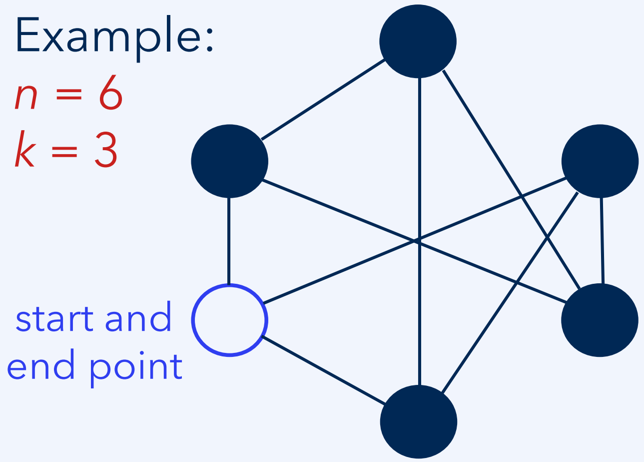
A travelling salesman needs to visit all the cities, by a path that ends at the same city where it starts (a **cycle**).

No city may be visited twice. Every city must be visited exactly once. (Except for returning to the start.)

The total travel distance, that is, the total length of the path, must be as short as possible.

The travelling salesman problem (TSP)

Example:
$n = 6$
$k = 3$



How many cycles covering all nodes might there be at most in a **graph** of n nodes, having maximum **degree** k, that is a graph where every node is adjacent to at most k other nodes?

How long would it then take to solve the TSP by a **brute force** algorithm?

Scenario:

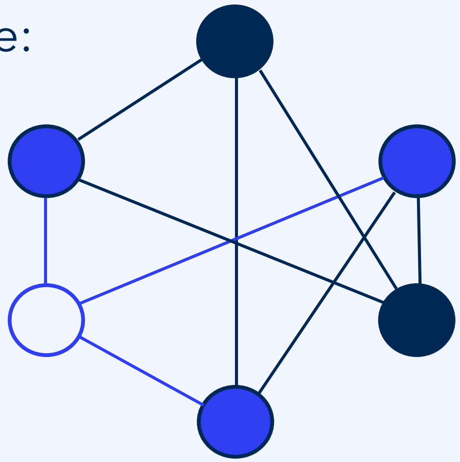
A travelling salesman needs to visit all the cities, by a path that ends at the same city where it starts (a **cycle**).

No city may be visited twice. Every city must be visited exactly once. (Except for returning to the start.)

The total travel distance, that is, the total length of the path, must be as short as possible.

The travelling salesman problem (TSP)

Example:
$n = 6$
$k = 3$



How many cycles covering all nodes might there be at most in a **graph** of n nodes, having maximum **degree** k, that is a graph where every node is adjacent to at most k other nodes?

How long would it then take to solve the TSP by a **brute force** algorithm?

Discussion:

The initial node is given.

For the next node there are at most k options. The same (in the worst case) for the node after that, and in each following step, at least as an upper bound. We need to visit n–1 nodes other than the initial node.

Upper bound: $k \cdot k \cdot \ldots \cdot k = k^{n-1}$ paths, with O(n) time per path to construct and compute the length of a path.

$O(n \cdot k^{n-1})$ time, or in slight abuse of notation $k^{O(n)}$ time, "**exponential time**."

University of Central Lancashire
UCLan

CO2412 Computational Thinking

Tutorial 3.1 discussion
Shortest paths
Travelling salesman

Where opportunity creates success