Welcome to Programming Fundamentals 3

University of Luxembourg

2025/2026

Bachelor in Computer Sciences (BICS)
Semester 3

Lecture - Week #14

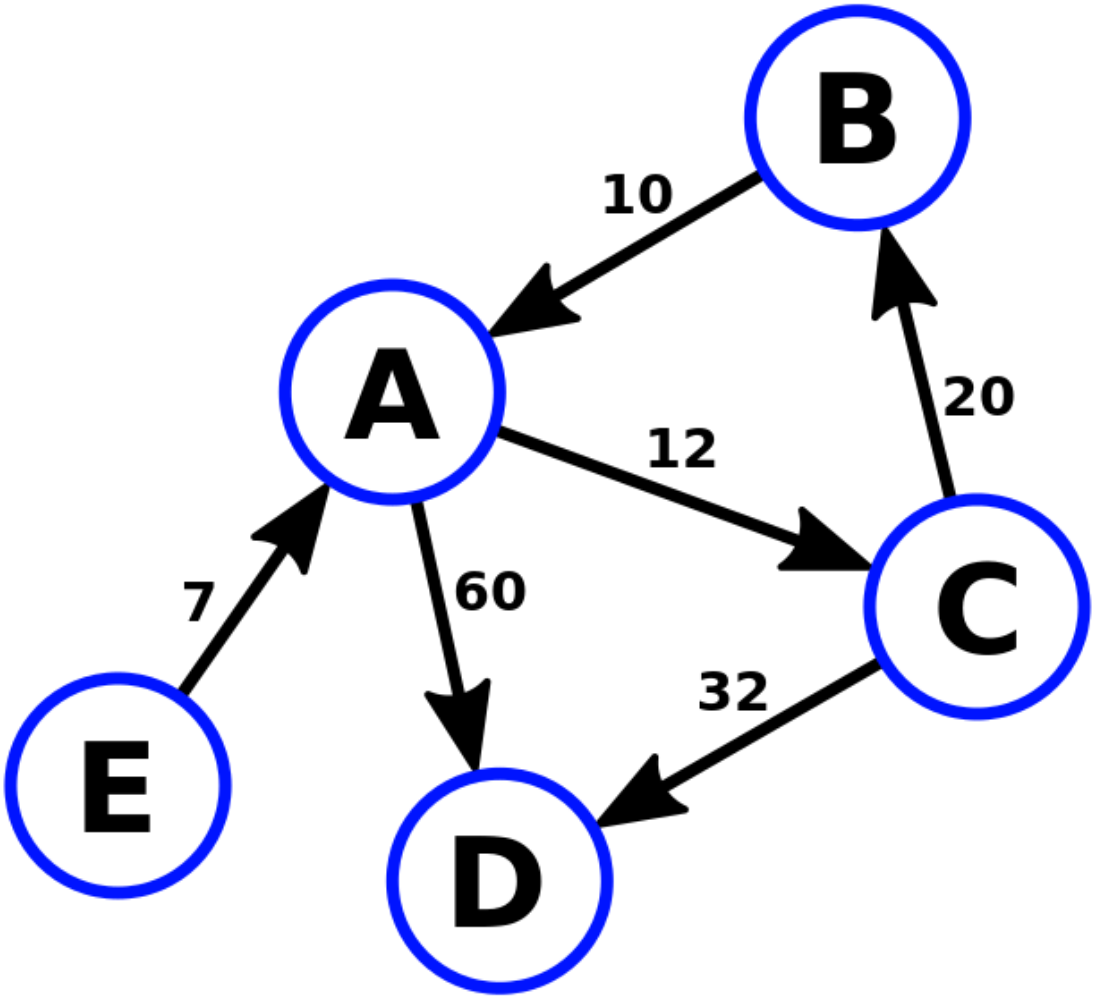
Dr. Afonso DELERUE ARRIAGA afonso.delerue@uni.lu

Important: Please remember to register your attendance in Moodle!

Graphs

- A graph is a data structure that consists of a **set of nodes** (also known as *vertices*) and a **set of edges** (sometimes called *links*), where each edge connects two nodes, indicating a relationship between them.
- Graphs are often visualized as a network of nodes connected by edges.
- Nodes can represent various entities such as people, cities, web pages, or any other abstract concept.
- Edges represent the connections or relationships between these entities.

- Graphs can be:
  - **undirected**, meaning that edges have no direction. If there is an edge between node A and node B, it implies that there is a relationship between A and B in both directions.
  - **directed**, meaning edges have a direction associated with them. If there is an edge from node A to node B, it implies a relationship from A to B but not necessarily from B to A.
  - **weighted**, meaning each edge is labelled with a numerical value or weight, which represents some additional information about the relationship between the nodes connected by the edge. These weights can represent distances for example.
  - **acyclic** if the graph does not contain any loops.



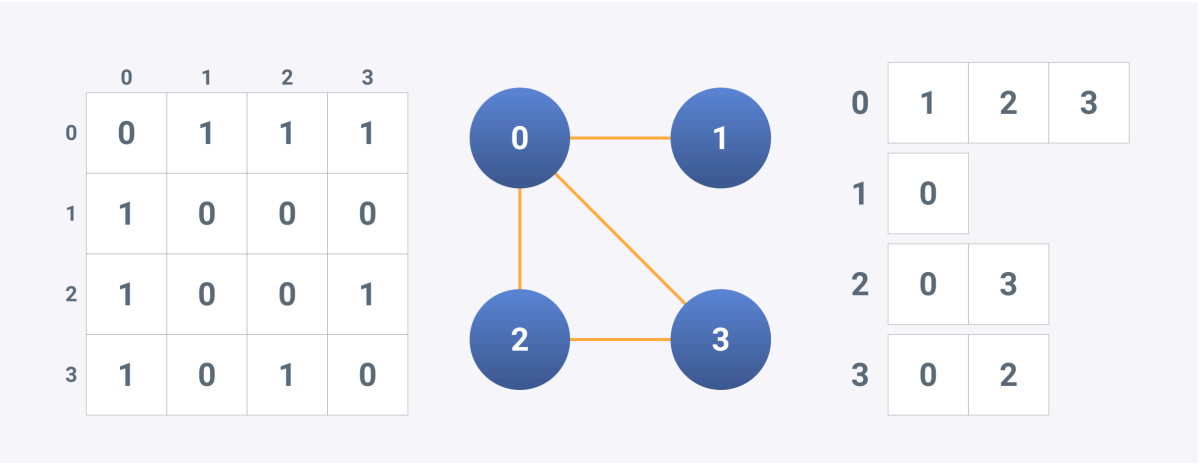
Most common graph algorithms

- Breadth-first search (**BFS**)
  - Searching engine crawlers
  - Searching on social networks
  - Finding available neighbour nodes in peer-to-peer networks
- Depth-first search (**DFS**)
  - Finding a path between two nodes
  - Solving mazes
- Shortest paths (**Dijkstra** algorithm)
  - Finding directions to travel from one location to another (mapping software, flight search engines, etc.)
  - Network package routing

Representation of graphs

In the imperative world, there are 2 common representations of graphs:

- Adjacency matrix
- Adjacency lists



- Each has it's strengths and weaknesses.
- In general, adjacency lists perform better except for very dense graphs.
  - Adjacency matrix are faster in removing or querying a single edge.
  - Storing a graph, adding or removing a node is faster with adjacency lists.

Graphs in Haskell

- Tree-based data structures are easy to deal with in Haskell.
- Working with graph-like structures is much less obvious.
  - Graph algorithms are usually easier explained in an imperitive style.
  - Many graph algorithms require the ability to update the state of nodes or edges during the algorithm's execution. In an immutable language like Haskell, this means creating a new data structure each time an update is needed, which can be time-consuming and memory-intensive.
  - "Node marking" strategy reflects an inherently imperative style.

Multiple approaches

1. Passing around the state.
2. Make use of ST or IO monads, which allow for safe mutation within a limited scope.
3. **Inductive graphs and functional algorithms** ← This is the scope of this lecture!

Inductive graphs

This lecture draws inspiration from the work of Martin Erwig on inductive graphs M. Erwig, "Inductive Graphs and Functional Graph Algorithms", Journal of Functional Programming, 2001. To enhance clarity and facilitate understanding, the code examples provided in this lecture have been simplified. For optimal performance, it is recommended to refer to the Haskell Functional Graph Library available on Github.

Graph constructors

- A graph is a set of nodes connected by edges.
- Nodes have indices (and optionally a label).

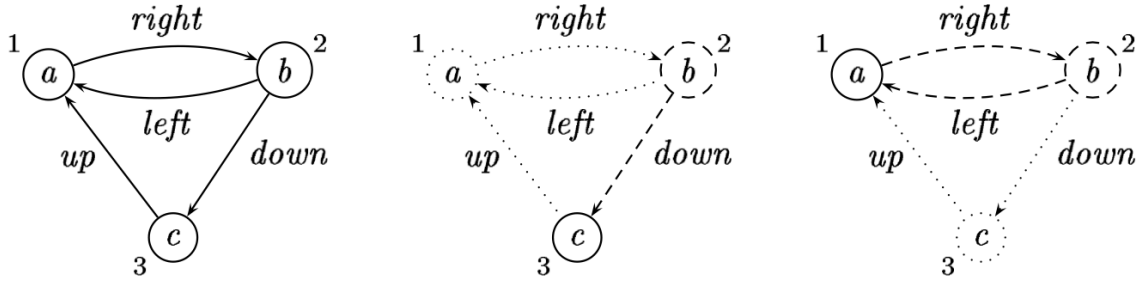
- Edges either have directions or are two-ways, resulting in *directed* and *undirected* graphs, respectively.
- Edges are optionally labelled.

```
type Node = Int
type Adj b = [(b,Node)]
type Context a b = (Adj b, Node, a, Adj b)

data Graph a b = Empty | Context a b :& Graph a b
    deriving (Show)
```

- Node have indices of type `Int`.
- List of adjacent nodes, `b` is the type of label of the edge (e.g. `String` or `Int`)
- A Context contains:
  1. a list of predecessors (of all the nodes already in the graph, this is the list of those pointing to the new node; `b` is the type of the label of the edge).
  2. the index of the new node.
  3. the label of the new node (type `a`, which can be `()` if nodes are unlabelled).
  4. a list of successors (of all the nodes already in the graph, this is the list of those that the new node points to; `b` is the type of the label of the edge).

- We can build the graph by inserting nodes contexts into the graph.




```
g :: Graph Char String
g =  ([("left",2),("up",3)],1,'a',[("right",2)]) :&
     ([],2,'b',[("down",3)])                     :&
     ([],3,'c',[])                               :& Empty
```

- There are multiple ways to represent the same graph, depending on the order node contexts are added. For example, we could have represented the graph as follows:

```
g' = ([("down",2)],3,'c',[("up",1)])    :&
     ([("right",1)],2,'b',[("left",1)]) :&
     ([],1,'a',[])                      :& Empty
```

- In fact, we can choose an arbitrary order of node insertions for building a graph.
- Consistency checks can be integrated into the insert functions, e.g.
  1. Check that node index is unique.
  2. Check that predecessors and sucessors are already in the graph.

Basic functions and folding graphs

```
succ :: Context a b -> [Node]
succ (_,_,_,s) = map snd s

gmap :: (Context a b -> Context c d) -> Graph a b -> Graph c d
gmap f Empty = Empty
gmap f (ctxt :& g) = f ctxt :& gmap f g

gfoldr :: (Context a b -> c -> c) -> c -> Graph a b -> c
gfoldr f acc Empty = acc
gfoldr f acc (ctxt :& g) = f ctxt (gfoldr f acc g)

gfoldl :: (c -> Context a b -> c) -> c -> Graph a b -> c
gfoldl f acc Empty = acc
gfoldl f acc (ctxt :& g) = gfoldl f (f acc ctxt) g
```

How to reason about graphs?

- Suppose there is a function `match` that decomposes a graph on a particular node

  ```
  match :: Node -> Graph a b -> (Maybe (Context a b), Graph a b)
  ```

- Given a `Node` and a `Graph`, if the node is part of the graph, the function `match` return a new graph where all traces of the node in question have been removed, and a new `Context` to add the node back to the graph with a complete list of **predecessors** and **successors** nodes.
- Type `Maybe` is used to account for the case in which the node to decompose from the graph is not part of the graph.
- Before discussing how to implement `match` efficiently, let's us first see how this inductive view of graphs allows us to describe a selection of graph algorithms often used in courses on algorithms and data structures.

Depth-First Search (DFS)

```
-- depth-first search (DFS)
dfs :: [Node] -> Graph a b -> [Node]
dfs [] _     = []
dfs _ Empty  = []
dfs (v:vs) g = case match v g of
                    (Just c,g')  -> v : dfs (succ c ++ vs) g'
                    (Nothing,_)  -> dfs vs g
```

Breadth-first search (BFS)

Exercise! 

```
-- breadth-first search (BFS)
bfs :: [Node] -> Graph a b -> [Node]
```

```
bfs [] _     = []
bfs _ Empty  = []
bfs (v:vs) g = case match v g of
                    (Just c,g')  -> v : bfs (vs ++ succ c) g'
                    (Nothing,_)  -> bfs vs g
```

How to actually implement `match`?

Naive approach:

1. Check all updates until node to decompose is found
2. On the way, prune node from predecessors and successors lists
3. At the same time, accumulate an update to node's predecessor and successor list
4. Also accumulate pruned contexts
5. When node is found, returned the updated context of the node + the pruned graph

In [ ]:
```haskell
match :: Node -> Graph a b -> (Maybe (Context a b), Graph a b)
match n g = decompose n g Empty [] []

decompose :: Node -> Graph a b -> Graph a b -> Adj b -> Adj b -> (Maybe (Contex
decompose _ Empty accG accP accS = (Nothing, accG)
decompose node ((p, n, l, s) :& g) accG accP accS
    | n == node =
        let ctx = (p ++ accP, n, l, s ++ accS)
        in (Just ctx, gfoldl (flip (:&)) g accG)
            -- the untouched graph `g` is the initial accumulator
            -- last context added to `accG` is first to be added back
            -- thus, pruned graph `accG` must be fold from left
    | otherwise =
        let (matchingSuccs, nonMatchingSuccs) = partition ((== node) . snd) s
            (matchingPreds, nonMatchingPreds) = partition ((== node) . snd) p
            accP' = case matchingSuccs of
                []  -> accP
                ((b,_):_) -> (b,n) : accP -- preserve the label b
            accS' = case matchingPreds of
                []  -> accS
                ((b,_):_) -> (b,n) : accS
            accG' = (nonMatchingPreds, n, l, nonMatchingSuccs) :& accG
        in decompose node g accG' accP' accS'
```

How to improve the implementation?

- The implementation of `match` defined here is simple but inefficient. It's described as such for educational purposes only.
- Time-complexity is `O(N*D)`, where `N` is the number of nodes and `D` is the maximum degree of the graph
- A simple optimization for `match` would be to store predecessors and successors nodes in a AVL tree, instead of a list, reducing the steps to `O(N * log N)`.
- FGL library uses more efficient Integer Maps, which bring down the asymptotic complexity of `match` to O(1).

How to implement Dijkstra's algorithm?

- Dijkstra shortest path algorithm can be implemented similarly to BFS algorithm, but instead of keeping nodes to visit in a queues, we keep them in a priority queue (implemented as a *heap* for example).
- The priority queue keeps track of visited nodes, the known smallest cost to get there and the predecessor node that leads to it (with the known smallest cost).
- The algorithm expands the search like in BFS, starting with the highest priority node (i.e. the next node with the smallest cost).
- For a visual step-by-step explanation, check out the video on the topic on the Youtube channel Computerphile. If you are willing to go the extra mile 🏃, try to implement if yourself, in Haskell, using a library that implements MinHeap.

Exercises 🥷🥷🥷

- Determining if a graph has a solution.

Implement a function `solveGraph` accepting 3 arguments in the given order:

`start` - The initial node of the directed graph `end` - The destination node of the directed graph `arcs` - A directed graph represented by a list/array of directed edges

and returns a boolean value depending on whether the destination node can be reached from the initial node by traversing zero or more directed edges. That means that if the start and end nodes are identical then the end node is trivially considered to be reachable - return `True` in this case. Also, if the start and end nodes are distinct and either node does not appear in arcs then you should return `False` since there is no sequence of directed edges that you may traverse to reach the end node from the start node.

https://www.codewars.com/kata/53223653a191940f2b000877

In [ ]:
```haskell
type Node = Char
type Arc  = (Node, Node)

solveGraph :: Node -> Node -> [Arc] -> Bool
solveGraph s e arcs = e `elem` bfs [s] arcs

bfs :: [Node] -> [Arc] -> [Node]
bfs [] _ = []
bfs vs [] = vs
bfs (v:vs) arcs = v : bfs (vs ++ neighbors) prunedArcs
  where
    neighbors = [b | (a,b) <- arcs, a==v]
    prunedArcs = [(a,b) | (a,b) <- arcs, a/=v]

-- try with import Data.List (partition), at home
```


The End