Data Structures (Not UML)

Pseudo-code Syntax, Graphs

Amandine Decker & Marie Cousin

M1 TAL/SC 2023–2024

Université de Lorraine, LORIA

Class Organisation

Third Lab :

- Reminder : optional exercise TD3 (due December 8th) ;
- hand back on a sheet of paper (December 8th), send it by e-mail, or hand back on Arche ;
- Correction of the third lab (except bonus exercise) is available on Arche ;
- Points and feedback on the optional exercise of TD2 are on Arche as well.

How to find the Arche repository :

- on your Arche top bar (blue-ish), click on "Home" ("Accueil") ;
- then "LORRAINE MANAGEMENT" (purple) ;
- and search for "Data Structure" ;
- Full name of the course is "Data Structures - Beginners", password is "Nox".

Syntax

Structures we have seen so far

Structure	Accessibility	Iterable ?
Queue	First element only (FIFO)	No iteration
Stack	Last element only (LIFO)	No iteration
List	Any element by ID	Iteration over elements or by IDs
Dict.	Any element by key	Iteration without order
Tree	Access root, left child, right child	No iteration (traversal algorithms)

Loops (1)

For loop

In a *for loop*, we create a variable that will take several values one after the other, *i.e.*, we will iterate over some elements and the variable will take one value at a time. The loop is thus executed a known definite number of times.

While loop

The *loop condition* is a condition that is true or false, it may use some variable used elsewhere in the algorithm. The instructions 1, 2, etc. will be executed as long as the *loop condition* remains satisfied (i.e., true).

Loops (1)

For loop

```
for i in range(0,5) do
    print(i)
end for
```

- The variable *i* is created by the loop and can be used inside it ;
- It is also incremented by the loop, we do not need to change it by hand.

While loop

```
i ← 0
while i < 5 do
    print(i)
    i ← i + 1
end while
```

- The variable *i* is NOT created by the loop, we need to create it by hand to use it ;
- It is NOT incremented by the loop, we must change it by hand if we want it to change.

Loops (2)

Nested loops

When you use a loop inside another loop, they are NOT executed in parallel. The *inside* loop will be executed at every step of the *outside* loop.

```
for i in range(0, 3) do
    for j in range(0, 5) do
        print(j)
    end for
end for
```

This algorithm will print 0, then 1, then 2, then 3, then 4 (j loop) and repeat this three times in total (because of i loop).

Loops (2)

Nested loops

When you use a loop inside another loop, they are NOT executed in parallel. The *inside* loop will be executed at every step of the *outside* loop.

```
L ← [0, 1, 2]
while !isEmpty(L) do
    for x in L do
        print(x)
    end for
end while
```

This algorithm will print 0, then 1, then 2 (For loop) and repeat this an infinite number of time because the condition of the While loop will remain *True*.

Return

Return instruction

This instruction returns *result*, and ends the algorithm.
(Note : If you encounter a ***return*** *in the middle of an algorithm,* ***THE FOLLOWING INSTRUCTIONS WILL NOT BE EXECUTED !****)*

```
if condition then
    print("condition True")
    return result
else
    print("condition False")
end if
print("If block finished")
```

- If *condition* is *True*, the algorithm will print "*condition True*", return *result*, and stop ;
- If *condition* is *False*, the algorithm will print "*condition False*", get out of the If block and print "*If block finished*".

Note : If you want to return more than one thing, use comas !

```
return result1, result2, result3
```

Require

Require instruction

This instruction requires the user to give a certain input to your algorithm. The input is stored in a variable that can be used in the algorithm.

- *Comment your instruction to explicit the input you want. For example, if your algorithm requires a list, you can write* "`Require:` `L` – *We need a list as input*" ;
- *If you need to require several objects, separate the variables with a comma.*

Require

Require instruction

This instruction requires the user to give a certain input to your algorithm. The input is stored in a variable that can be used in the algorithm.

- *Comment your instruction to explicit the input you want. For example, if your algorithm requires a list, you can write* "`Require: L` – *We need a list as input*" ;
- *If you need to require several objects, separate the variables with a comma.*

DO NOT :

- Create an arbitrary object :

1: $L \leftarrow [1, 2, 3, 4, 5, 6]$ `-- Use Require: L instead`

Require

Require instruction

This instruction requires the user to give a certain input to your algorithm. The input is stored in a variable that can be used in the algorithm.

- *Comment your instruction to explicit the input you want. For example, if your algorithm requires a list, you can write* "`Require: L` – *We need a list as input*" ;
- *If you need to require several objects, separate the variables with a comma.*

DO NOT :

- Re-use your variable to store something else / Overwrite your variable :

Require: L -- *We require a list L*

1: $L \leftarrow []$ -- *This erases the values given in the input*

Require

Require instruction

This instruction requires the user to give a certain input to your algorithm. The input is stored in a variable that can be used in the algorithm.

- *Comment your instruction to explicit the input you want. For example, if your algorithm requires a list, you can write* "`Require: L` – *We need a list as input*" ;
- *If you need to require several objects, separate the variables with a comma.*

DO NOT :

- Use Require AND create the object afterwards :

  Require: L *-- We require a list L*

  1: $L \leftarrow []$ *-- This erases the values given in the input, use a comment to say that L must be a list*

Require

Require instruction

This instruction requires the user to give a certain input to your algorithm. The input is stored in a variable that can be used in the algorithm.

- *Comment your instruction to explicit the input you want. For example, if your algorithm requires a list, you can write* "`Require:` `L` – *We need a list as input*" ;
- *If you need to require several objects, separate the variables with a comma.*

Difference between **Require** and **Creating a new variable** for later :

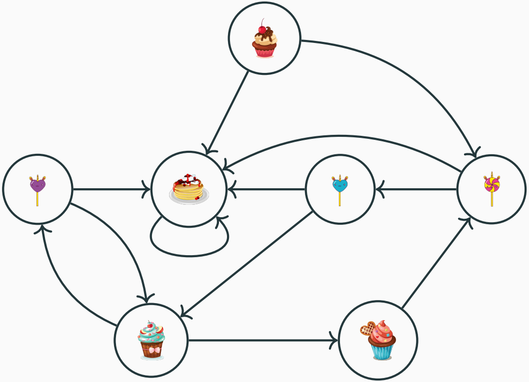
Require: L -- *We require a list of integers L*

1: $L\_even \leftarrow []$ -- *We create an empty list to store all the even numbers of L*

Graphs

Graphs : Idea

"I want to find all the desserts !"



- Nox is in front of a maze where each room contains a dessert ;
- Some rooms have a door to other rooms ;
- We have a graph of desserts !

Graphs : Definition

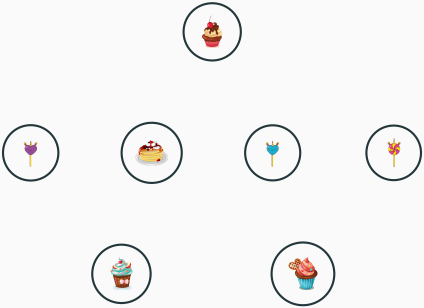
Graph

A Graph is a data structure composed of nodes and edges.

Graphs : Definition

Graph

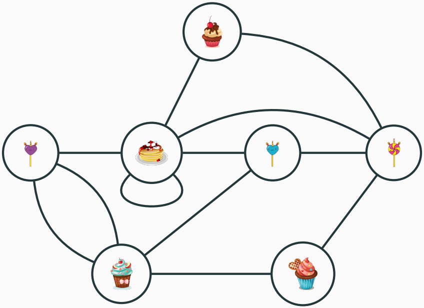
A Graph is a data structure composed of nodes and edges.



Graph

A Graph is a data structure composed of nodes and edges.

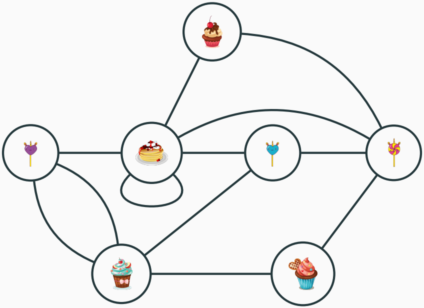
- The nodes can be connected by edges ;



Graph

A Graph is a data structure composed of nodes and edges.

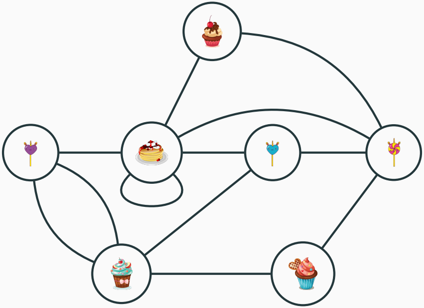
- The nodes can be connected by edges ;
- The edges are either ALL directed or ALL un-directed :



Graph

A Graph is a data structure composed of nodes and edges.

- The nodes can be connected by edges ;
- The edges are either ALL directed or ALL un-directed :
  - If they are un-directed, both direction work ;

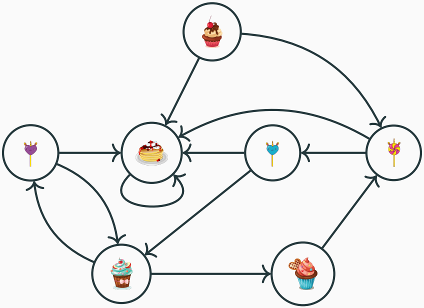


Graphs : Definition

Graph

A Graph is a data structure composed of nodes and edges.

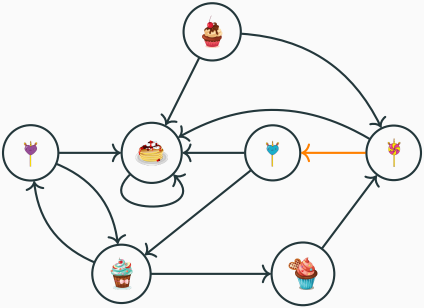
- The nodes can be connected by edges ;
- The edges are either ALL directed or ALL un-directed :
  - If they are un-directed, both direction work ;
  - If they are directed, only the indicated direction(s) work ;



Graph

A Graph is a data structure composed of nodes and edges.

- The nodes can be connected by edges ;
- The edges are either ALL directed or ALL un-directed :
  - If they are un-directed, both direction work ;
  - If they are directed, only the indicated direction(s) work ;

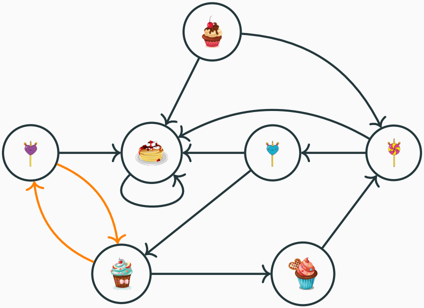


Graphs : Definition

Graph

A Graph is a data structure composed of nodes and edges.

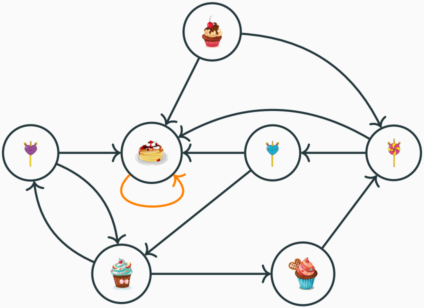
- The nodes can be connected by edges ;
- The edges are either ALL directed or ALL un-directed :
  - If they are un-directed, both direction work ;
  - If they are directed, only the indicated direction(s) work ;



Graph

A Graph is a data structure composed of nodes and edges.

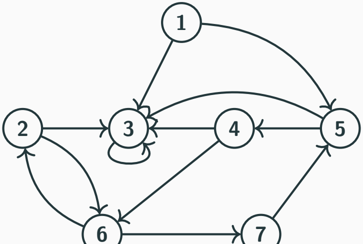
- The nodes can be connected by edges ;
- The edges are either ALL directed or ALL un-directed :
  - If they are un-directed, both direction work ;
  - If they are directed, only the indicated direction(s) work ;
- A node can be connected to itself.



Graphs : Definition

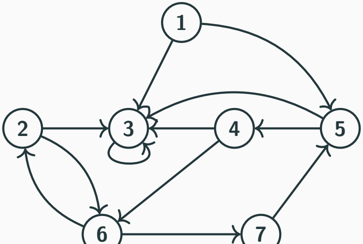
Properties

- The size of a graph is its number of nodes ;
- Each node has a set of directly accessible nodes : the nodes linked to them by an edge ;
- We say that a node B is accessible from a node A if there exist a path (*i.e.*, a sequence of edges) going from node A to node B (A and B can be the same node) ;
- The length of a path between two nodes is the number of edges used to go from one to the other.



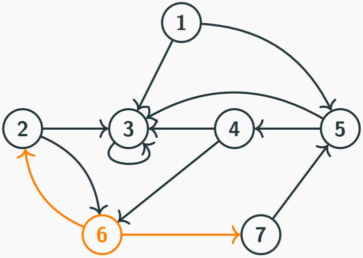
Properties

- The size of a graph is its number of nodes ; → *7 here*
- Each node has a set of directly accessible nodes : the nodes linked to them by an edge ;
- We say that a node B is accessible from a node A if there exist a path (*i.e.*, a sequence of edges) going from node A to node B (A and B can be the same node) ;
- The length of a path between two nodes is the number of edges used to go from one to the other.



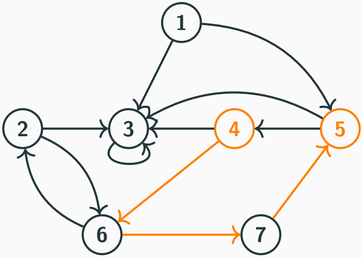
Properties

- The size of a graph is its number of nodes ;
- Each node has a set of directly accessible nodes : the nodes linked to them by an edge ; → *2 and 7 are directly accessible from 6*
- We say that a node B is accessible from a node A if there exist a path (*i.e.*, a sequence of edges) going from node A to node B (A and B can be the same node) ;
- The length of a path between two nodes is the number of edges used to go from one to the other.



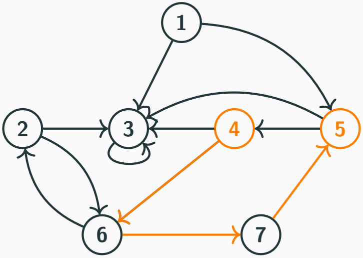
Properties

- The size of a graph is its number of nodes ;
- Each node has a set of directly accessible nodes : the nodes linked to them by an edge ;
- We say that a node B is accessible from a node A if there exist a path (*i.e.*, a sequence of edges) going from node A to node B (A and B can be the same node) ; $\rightarrow$ *5 is accessible from 4*
- The length of a path between two nodes is the number of edges used to go from one to the other.



Properties

- The size of a graph is its number of nodes ;
- Each node has a set of directly accessible nodes : the nodes linked to them by an edge ;
- We say that a node B is accessible from a node A if there exist a path (*i.e.*, a sequence of edges) going from node A to node B (A and B can be the same node) ;
- The length of a path between two nodes is the number of edges used to go from one to the other. → *The path from 4 to 5 is of length 3*



Graphs : No Formalism

Operations

- **Creating an edge** means adding an edge linking two existing nodes (possibly the same node to itself). If the graph is directed the direction of the new edge must be specified. $\rightarrow$ *Create the edge* $1 \rightarrow 3$*.*
- **Creating a node** means adding a new node to the graph. The edges coming from and to this new node can be specified (see *creating an edge*). $\rightarrow$ *Create the edge 8 with the edges* $2 \rightarrow 8$ *and* $8 \rightarrow 5$*.*
- **Removing an edge** means removing the edge from one node to another (possibly the same node to itself). If the graph is directed the direction of the deleted edge must be specified. $\rightarrow$ *Remove the edge* $5 \rightarrow 4$*.*
- **Removing a node** means removing an existing node from a graph and all the edges coming from or to this node. $\rightarrow$ *Remove the node* 3*.*

Graphs : Don't panic

Regarding graphs, we expect from you :

- To be able to draw a graph given instructions such as "The nodes 3, 7 and 12 and the edges $3 \rightarrow 12$, $12 \rightarrow 3$, $7 \rightarrow 12$, and $7 \rightarrow 3$"
- To be able to read a graph and find the nodes that are accessible to each other ;
- To be able to explain operations on a graph such as *adding* and *removing* edges and nodes ;
- To explain in English the principle of basic algorithms.

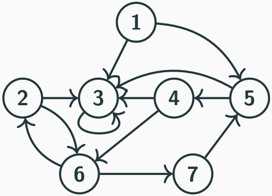
We do not expect from you :

- to read or write algorithms on graphs.

Graphs : Example

Question

How do you find all the nodes accessible from a given node X in a graph ?



Graphs : Example

Question

How do you find all the nodes accessible from a given node X in a graph ?

Solution

Intuition : All the nodes directly accessible from X are accessible, so are the nodes directly accessible from the ones we just found, etc.

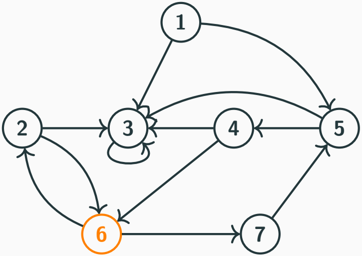
- We need to store two things : the accessible nodes, and the nodes we have already visited to avoid checking them several times ;
- All the nodes directly accessible from X are accessible : we store them as accessible nodes and X as checked nodes ;
- All the nodes directly accessible from the ones we have just found are accessible from X as well : For all the nodes in accessible nodes, if we have not checked them already ; add their direct neighbours to the accessible nodes and add the nodes we just checked to checked nodes ;
- We repeat this until all the accessible nodes have been checked.

Question

How do you find all the nodes accessible from a given node X in a graph ?

Let's try with X = 6 :

- We will need to store the accessible nodes and the already checked nodes
  - Accessible nodes : []
  - Checked nodes : []



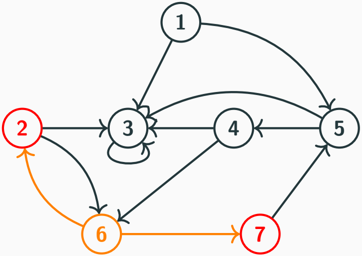
Graphs : Example

Question

How do you find all the nodes accessible from a given node X in a graph ?

Let's try with X = 6 :

- We look for all the nodes directly accessible from 6 (2, 7) :
    - Accessible nodes : [2, 7]
    - Checked nodes : [6]

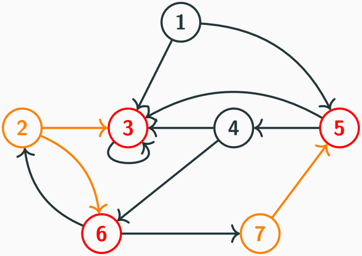


Question

How do you find all the nodes accessible from a given node X in a graph ?

Let's try with X = 6 :

- Neither 2 nor 7 have been checked yet so we look for all the nodes directly accessible from 2 (3, 6) and from 7 (5) :
  - Accessible nodes : [2, 3, 5, 6, 7]
  - Checked nodes : [2, 6, 7]

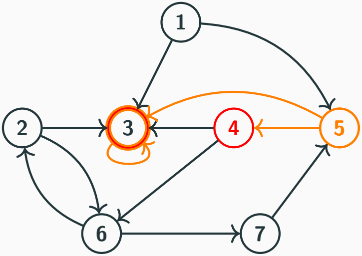


Question

How do you find all the nodes accessible from a given node X in a graph ?

Let's try with X = 6 :

- In our accessible nodes, only 3 and 5 have not been checked yet so we look for all the nodes directly accessible from 3 (3) and from 5 (3, 4) :
  - Accessible nodes : [2, **3**, 4, 5, 6, 7]
  - Checked nodes : [2, 3, 5, 6, 7]



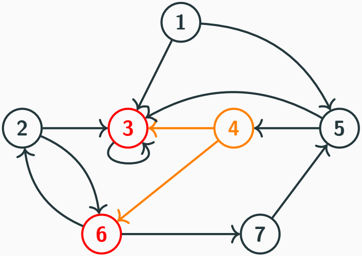
Graphs : Example

Question

How do you find all the nodes accessible from a given node X in a graph ?

Let's try with X = 6 :

- In our accessible nodes, only 4 has not been checked yet so we look for all the nodes directly accessible from 4 (6) :
  - Accessible nodes : [2, **3**, 4, 5, **6**, 7]
  - Checked nodes : [2, 3, 4, 5, 6, 7]



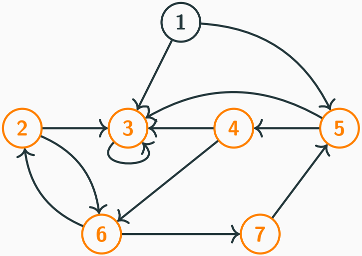
Graphs : Example

Question

How do you find all the nodes accessible from a given node X in a graph ?

Let's try with X = 6 :

- All of our accessible nodes have been checked so this is our final answer :
  - All the nodes accessible from 6 are [2, 3, 4, 5, 6, 7]



TD2 Optional Exercise

TD2 Optional Exercise : General Feedback

- Read the question carefully, most of the time it indicates the input and output structures you should use !
- If the algorithm must take an input, use Require (do not define an arbitrary structure) ;
- *Remove* does not exist for lists ;
- Modifying a list while iterating over it (“`for x in L do:`”) can create problems in some programming languages ;
- Comment your algorithm and write a short description (you can get points even when your algorithm does not work perfectly if the idea is good and well explained) ;
- Please use English when required.

Summary

Summary

- Graphs are complex structures ;

  $\rightarrow$ not linear like trees ;
- They are constituted of Nodes and Edges ;
  - *Nodes can also be called Vertices or sometimes States for specific types of graphs ;*
  - *Edges can also be called Lines or Arcs ;*
- Nodes are accessible to each other ;
- A graph has a size ;
- A certain path between two nodes has a length ;
- No formalism, only descriptions in English, drawings, etc.