Algorithmics and Complexity
Cours 1/7 : Graph Traversal

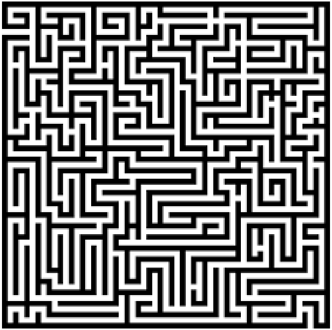
CentraleSupélec – Gif

ST2 – Gif

Plan

Plan

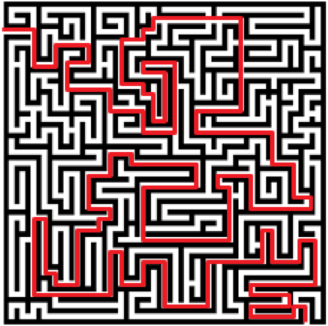
Find the exit out of a maze



Problems

- Computational model of this maze problem ?
- What characterises a solution to this problem ?

Find the exit out of a maze



Problems

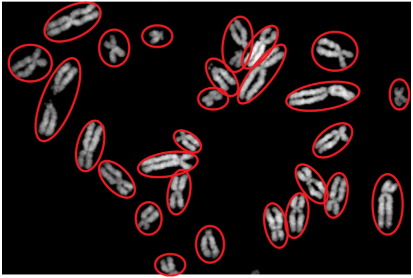
- Computational model of this maze problem ?
- What characterises a solution to this problem ?
- How to compute efficiently a solution in an efficient manner ?

Identify elements in a picture

Problems

- Computational model of this picture ?
- What characterises a chromosome ?

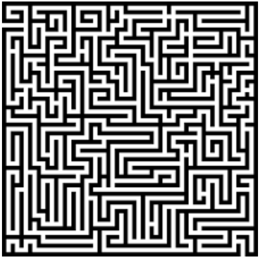
Identify elements in a picture



Problems

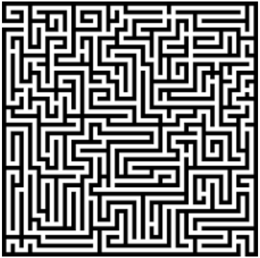
- Computational model of this picture ?
- What characterises a chromosome ?
- How to compute which parts of the image represent chromosomes ?

Mazes and chromosomes



What is the connection between finding a path in a maze and counting chromosomes ?

Mazes and chromosomes



What is the connection between finding a path in a maze and counting chromosomes ?

➜ Graphs, graph traversal and connectivity !

Graph

Data Structures

What you saw in your previous studies :

- ✓ Variables (often connected to representation types)
- ✓ Arrays (one dimensions or more)
- ✓ Lists, stacks, queues
- ✗ Objects
- ✗ Dictionnaries

Graph

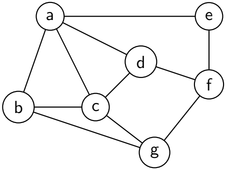
Graphs are another type of data structure

- ➜ Maybe the most frequently used in algorithmics !

What is a Graph ?

Graph

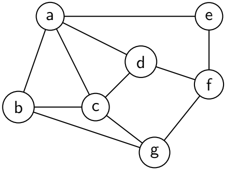
Mathematical structure used to represent relations between elements



What is a Graph ?

Graph

Mathematical structure used to represent relations between elements



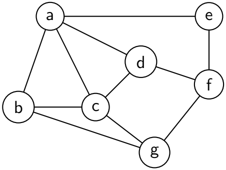
Definitions

- Nodes also known as vertices
- Edges

What is a Graph ?

Graph

Mathematical structure used to represent relations between elements



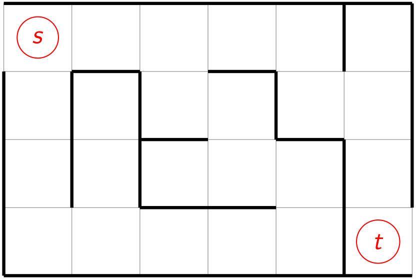
Definitions

- Nodes also known as vertices
- Edges

Notation

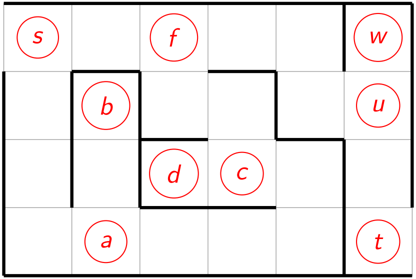
Graph $G = (V, E)$
where V is the set of vertices and E the set of edges.

Maze modeling



A maze seen as a graph

Maze modeling

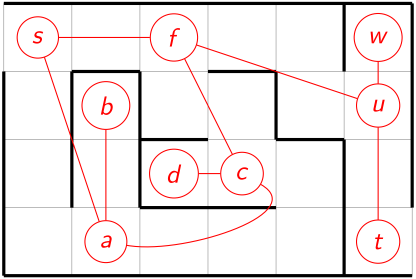


A maze seen as a graph

- The intersections and the dead-ends are represented by vertices ;

  Each vertex can be associated with a label.

Maze modeling

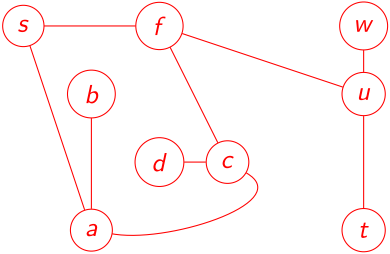


A maze seen as a graph

- The intersections and the dead-ends are represented by vertices ;

  Each vertex can be associated with a label.
- Each corridor is represented by an edge.

Maze modeling



A maze seen as a graph

- The intersections and the dead-ends are represented by vertices ;

  Each vertex can be associated with a label.
- Each corridor is represented by an edge.

Remarks & definitions

Remarks

- This type of graph is a non-directed graph.

  We will see directed graphs in the next lecture
- Each edge is characterised by its two ending vertices : $E \subseteq V \times V$

Remarks & definitions

Remarks

- This type of graph is a non-directed graph.
  We will see directed graphs in the next lecture
- Each edge is characterised by its two ending vertices : $E \subseteq V \times V$

Definitions

- A chain from x to y is a finite series of consecutive edges connecting x to y.
- A graph is connected when there exists a chain between any pair of vertices.

Model of the maze problem

With the same data,

(in our case, a graph $G = (V, E)$ and two vertices s and t)

we can build different types of problems !

- Decision problem
- Construction problem
- Optimization problem

Model of the maze problem – Decision

Existence of a chain

- Inputs : Given a graph $G = (V, E)$, a starting vertex $s \in V$ and a target vertex $t \in V$
- Question : Is there a chain from s to t ?

Decision problem

✓ The answer to the above question is either yes or no.

Model of the maze problem – Construction

Construction of a chain

- Inputs : Given a graph $G = (V, E)$, a starting vertex $s \in V$ and a target vertex $t \in V$
- Question : Build a chain from s to t

Construction problem

✓ The answer is a solution to the problem.
  - ➜ Compute a data structure that satisfies the constraints of the problem.

- Such a structure might not exist !
  - ➜ The answer to the corresponding decision problem is no

Model of the maze problem – Optimisation

Shortest chain

- Inputs : Given a graph $G = (V, E)$, a starting vertex $s \in V$ and a target vertex $t \in V$
- Question : What is the shortest chain from s to t ?

Optimisation problem

✓ The answer to the question is a solution.

✓ There exists a function (in this case, the length of the chain) that needs to be maximised or minimised.

Model of the maze problem – Optimisation

Shortest chain

- Inputs : Given a graph $G = (V, E)$, a starting vertex $s \in V$ and a target vertex $t \in V$
- Question : What is the shortest chain from s to t ?

Optimisation problem

- ✓ The answer to the question is a solution.
- ✓ There exists a function (in this case, the length of the chain) that needs to be maximised or minimised.

In this lecture, we will focus on decision problems.
(existence of a chain)

Instance of a problem

Definition : instance

- An instance is a set of inputs that satisfy the constraints of the problem.
- The size of an instance corresponds to the size of the data :
  - ➜ It depends on the computer representation ;
  - ➜ Atomic element : *number of elements in a list, number of vertices and edges in a graph. . .*

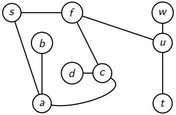
Instance of a problem

Definition : instance

- An instance is a set of inputs that satisfy the constraints of the problem.
- The size of an instance corresponds to the size of the data :
  - ➜ It depends on the computer representation ;
  - ➜ Atomic element : *number of elements in a list, number of vertices and edges in a graph. . .*

Example

- A graph $G = (V, E)$ and two vertices s and $t \in V$.



- Size of the instance $= |V| + |E|$

Solving algorithm

Definition : algorithm

An algorithm is a finite and non ambiguous series of operations or instructions that can be used to solve a problem.

Example : existence of a chain

1. Starting from s, select a vertex connected to the currently selected vertex
2. Continue as long as the currently selected vertex is not t
3. Answer "yes" when t is reached

Solving algorithm

Definition : algorithm

An algorithm is a finite and non ambiguous series of operations or instructions that can be used to solve a problem.

Example : existence of a chain

1. Starting from s, select a vertex connected to the currently selected vertex
2. Continue as long as the currently selected vertex is not t
3. Answer "yes" when t is reached

Remarks

- This algorithm never answers "no"
- This algorithm might not terminate.

  → One cannot always predict whether an algorithm terminates or not

Solving algorithm

Algorithm

In Computer Science, an algorithm is a finite series of instructions using :

- variables, data structures,
- control instructions (loops, conditional instructions, function calls, *etc*).

that can be executed step by step by a deterministic computer.

Solving algorithm

Algorithm

In Computer Science, an algorithm is a finite series of instructions using :

- variables, data structures,
- control instructions (loops, conditional instructions, function calls, *etc*).

that can be executed step by step by a deterministic computer.

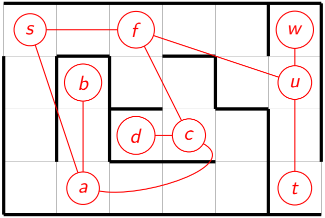
Can we write this algorithm in Python ?

`def exists_chain(V,E,s,t): ...` (to be continued)

We assume that there exists a function `neighbours(x,E)` that returns the list of neighbouring vertices

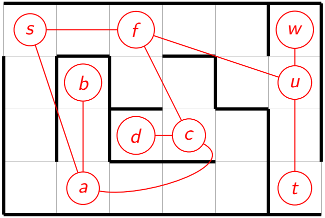
Back to mazes

Find the exit = existence of a chain !



Back to mazes

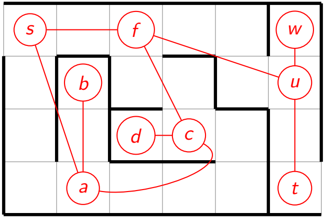
Find the exit = existence of a chain !



→ Graph traversal algorithms.

Back to mazes

Find the exit = existence of a chain !



→ Graph traversal algorithms.

Beware of cycles !

➜ We must store the visited vertices to avoid a loop.
(unlike the previous algorithm. . .)

Plan

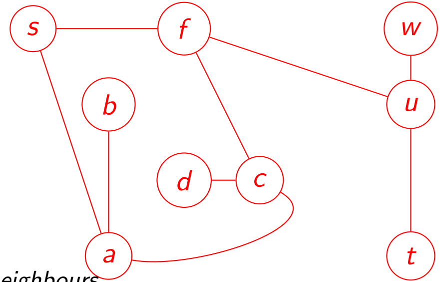
1. Graph-based problems
2. Depth-First Search
   - Principle
   - Recursive implementation
   - Iterative implementation
3. Breadth-First Search
4. Complexity
5. Connectivity
6. Conclusion

Depth-First Search

General idea of the algorithm…

Always select the first connected vertex (not already visited) with respect to the current one *and retrace its steps*

⏩ Skip



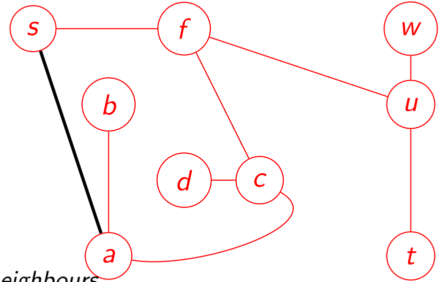
Lexical order on neighbours

Depth-First Search

General idea of the algorithm...

Always select the first connected vertex (not already visited) with respect to the current one *and retrace its steps*

⏩ Skip



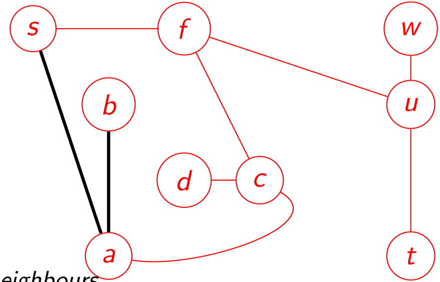
Lexical order on neighbours

Depth-First Search

General idea of the algorithm. . .

Always select the first connected vertex (not already visited) with respect to the current one and retrace its steps

⏩ Skip



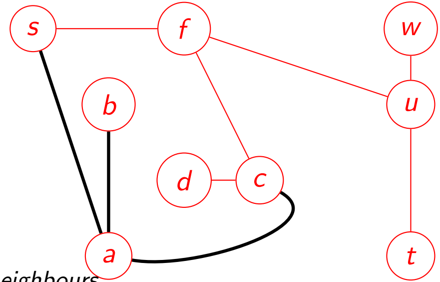
Lexical order on neighbours

Depth-First Search

General idea of the algorithm. . .

Always select the first connected vertex (not already visited) with respect to the current one and retrace its steps

⏩ Skip



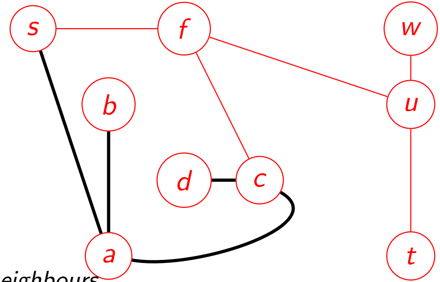
Lexical order on neighbours

Depth-First Search

General idea of the algorithm...

Always select the first connected vertex (not already visited) with respect to the current one *and retrace its steps*

⏩ Skip



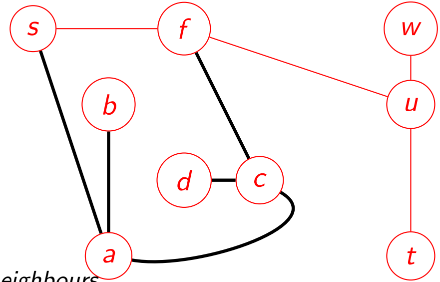
Lexical order on neighbours

Depth-First Search

General idea of the algorithm. . .

Always select the first connected vertex (not already visited) with respect to the current one *and retrace its steps*

⏩ Skip



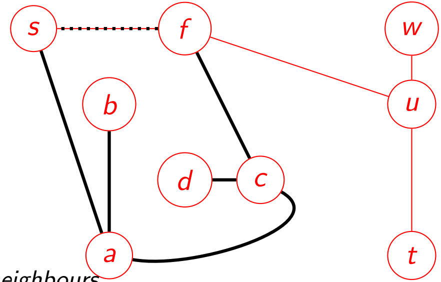
Lexical order on neighbours

Depth-First Search

General idea of the algorithm...

Always select the first connected vertex (not already visited) with respect to the current one *and retrace its steps*

⏩ Skip



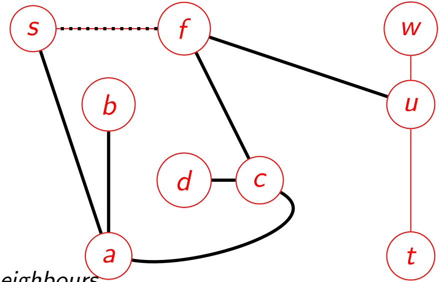
Lexical order on neighbours

Depth-First Search

General idea of the algorithm. . .

Always select the first connected vertex (not already visited) with respect to the current one *and retrace its steps*

⏩ Skip

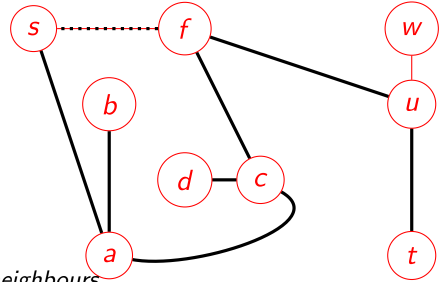


Lexical order on neighbours

Depth-First Search

General idea of the algorithm. . .

Always select the first connected vertex (not already visited) with respect to the current one *and retrace its steps*



Lexical order on neighbours

Stop when t is reached

How to implement this idea

i.e. turn it into an algorithm. . .

What do we need ?

1. Knowing the neigbours of a vertex (to select the first one)
2. Knowing if a vertex was already visited (to avoid looping)
3. Select systematically the first non-visited neighbour
4. If this is not t, repeat from the current node

How to implement this idea

i.e. turn it into an algorithm…

What do we need ?

1. Knowing the neigbours of a vertex (to select the first one)
   ➜ `neighbours(x,E)` function that returns the list of neighbours
2. Knowing if a vertex was already visited (to avoid looping)
3. Select systematically the first non-visited neighbour
4. If this is not t, repeat from the current node

How to implement this idea

i.e. turn it into an algorithm...

What do we need ?

1. Knowing the neigbours of a vertex (to select the first one)
   ➜ `neighbours(x,E)` function that returns the list of neighbours
2. Knowing if a vertex was already visited (to avoid looping)
   ➜ An array visited that associates nodes with a boolean value.
3. Select systematically the first non-visited neighbour
4. If this is not t, repeat from the current node

How to implement this idea

i.e. turn it into an algorithm. . .

What do we need ?

1. Knowing the neigbours of a vertex (to select the first one)
   - ➜ `neighbours(x,E)` function that returns the list of neighbours
2. Knowing if a vertex was already visited (to avoid looping)
   - ➜ An array visited that associates nodes with a boolean value.
   - ➜ Python dictionary
3. Select systematically the first non-visited neighbour
4. If this is not t, repeat from the current node

How to implement this idea

i.e. turn it into an algorithm...

What do we need ?

1. Knowing the neigbours of a vertex (to select the first one)
   - ➜ `neighbours(x,E)` function that returns the list of neighbours
2. Knowing if a vertex was already visited (to avoid looping)
   - ➜ An array visited that associates nodes with a boolean value.
   - ➜ Python dictionary
3. Select systematically the first non-visited neighbour
4. If this is not t, repeat from the current node
   - ➜ Recursive function

Reminder : Python dictionaries

Set of (key,value) data structure

```
dico = { key:value, ... }
```

- The data structure associates a value to a name (the key), using : `dico[key]=value`
- Keys are often character strings
- Values are accessed using : `dico[key]`
- We can iterate over the keys :

```
for k in dico: ...
```

- We can test if a key exists :

```
if k in dico: ...
```

Depth-First Search : recursive implementation

1. Initialization of the dictionary : `visited = { }`

Depth-First Search : recursive implementation

1. Initialization of the dictionary : `visited = { }`
2. Recursive function `DFS_rec(V,E,n,t)`

```
def DFS_rec(V,E,n,t):    # n : the current node
  visited[n] = True
  if n==t: return True
  for v in neighbours(n,E):
    if not v in visited:
      if DFS_rec(V,E,v,t):
        return True
  return False
```

Depth-First Search : recursive implementation

1. Initialization of the dictionary : `visited = { }`
2. Recursive function `DFS_rec(V,E,n,t)`

```
def DFS_rec(V,E,n,t):     # n : the current node
  visited[n] = True
  if n==t: return True
  for v in neighbours(n,E):
    if not v in visited:
      if DFS_rec(V,E,v,t):
        return True
  return False
```

3. Calling the function : `DFS_rec(V,E,s,t)`

Depth-First Search : recursive implementation

1. Initialization of the dictionary : `visited = { }`
2. Recursive function `DFS_rec(V,E,n,t)`

```
def DFS_rec(V,E,n,t):     # n : the current node
  visited[n] = True
  if n==t: return True
  for v in neighbours(n,E):
    if not v in visited:
      if DFS_rec(V,E,v,t):
        return True
  return False
```

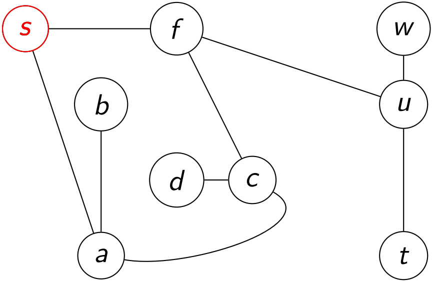
3. Calling the function : `DFS_rec(V,E,s,t)`

Remarks

- The list of nodes V is not used in `DFS_rec`
- This algorithm does not return the path, only `True` or `False` depending on whether it reaches t or not.

Implementation : demo

⏩ Skip Demo



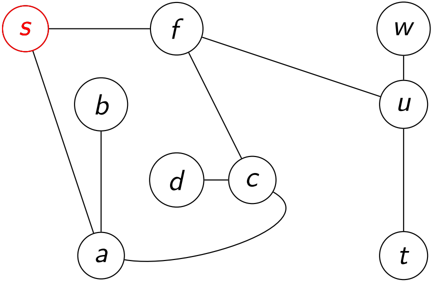
visited

a :
b :
c :
d :
f :
s :
t :
u :
w :

```
DFS_rec(V,E,'s','t')
```

Implementation : demo

⏩ Skip Demo



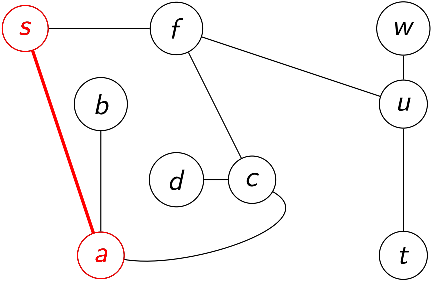
`DFS_rec(V,E,'s','t')`

visited

a :
b :
c :
d :
f :
s : *True*
t :
u :
w :

Implementation : demo

⏩ Skip Demo



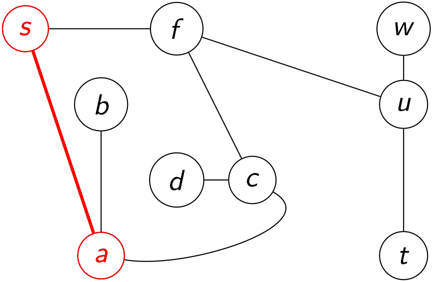
`DFS_rec(V,E,'s','t')`

visited

a :	
b :	
c :	
d :	
f :	
s :	*True*
t :	
u :	
w :	

Implementation : demo

⏩ Skip Demo



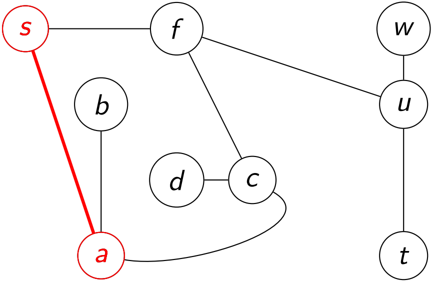
visited

a :
b :
c :
d :
f :
s : $True$
t :
u :
w :

```
DFS_rec(V,E,'s','t')
↪ DFS_rec(V,E,'a','t')
```

Implementation : demo

⏩ Skip Demo



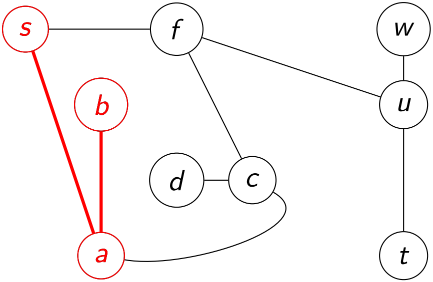
visited

a :	*True*
b :	
c :	
d :	
f :	
s :	*True*
t :	
u :	
w :	

```
DFS_rec(V,E,'s','t')
↪ DFS_rec(V,E,'a','t')
```

Implementation : demo

⏩ Skip Demo



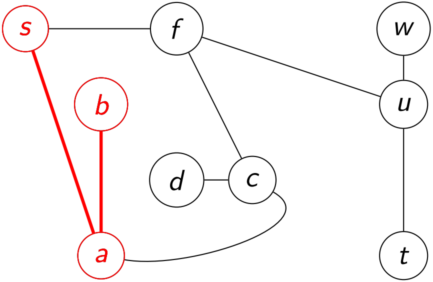
```
DFS_rec(V,E,'s','t')
↪ DFS_rec(V,E,'a','t')
```

visited

a :	True
b :	
c :	
d :	
f :	
s :	True
t :	
u :	
w :	

Implementation : demo

⏩ Skip Demo



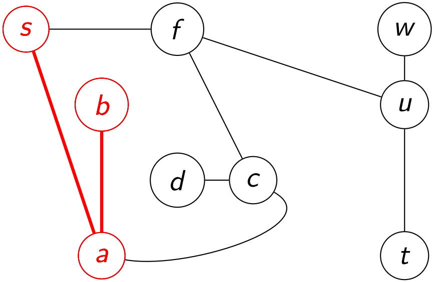
visited

a : $True$
b :
c :
d :
f :
s : $True$
t :
u :
w :

```
DFS_rec(V,E,'s','t')
↪ DFS_rec(V,E,'a','t')
    ↪ DFS_rec(V,E,'b','t')
```

Implementation : demo

⏩ Skip Demo



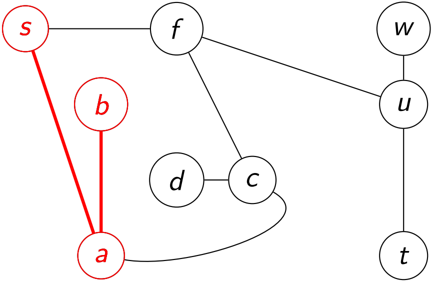
visited

a :	*True*
b :	*True*
c :	
d :	
f :	
s :	*True*
t :	
u :	
w :	

```
DFS_rec(V,E,'s','t')
↪ DFS_rec(V,E,'a','t')
    ↪ DFS_rec(V,E,'b','t')
```

Implementation : demo

⏩ Skip Demo



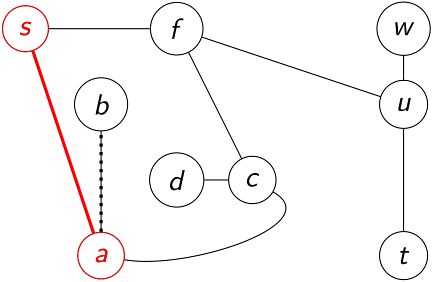
visited

a :	*True*
b :	*True*
c :	
d :	
f :	
s :	*True*
t :	
u :	
w :	

```
DFS_rec(V,E,'s','t')
↪ DFS_rec(V,E,'a','t')
   ↪ DFS_rec(V,E,'b','t') : False
```

Implementation : demo

⏩ Skip Demo



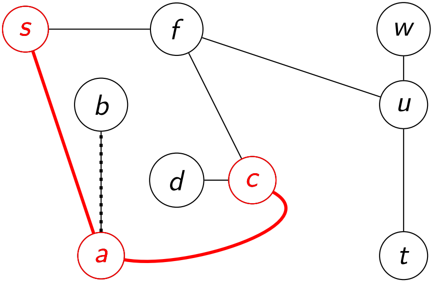
visited

a : *True*
b : *True*
c :
d :
f :
s : *True*
t :
u :
w :

```
DFS_rec(V,E,'s','t')
↪ DFS_rec(V,E,'a','t')
```

Implementation : demo

⏩ Skip Demo



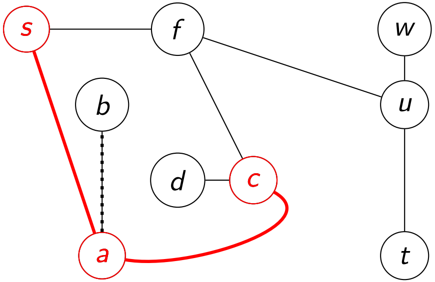
visited

a :	$True$
b :	$True$
c :	
d :	
f :	
s :	$True$
t :	
u :	
w :	

```
DFS_rec(V,E,'s','t')
↪ DFS_rec(V,E,'a','t')
```

Implementation : demo

⏩ Skip Demo



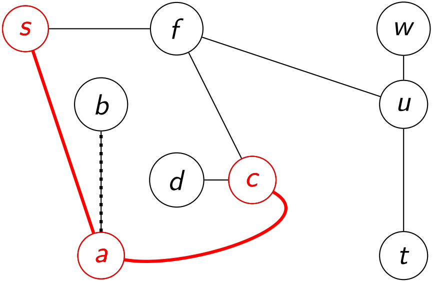
visited

a :	*True*
b :	*True*
c :	
d :	
f :	
s :	*True*
t :	
u :	
w :	

```
DFS_rec(V,E,'s','t')
↪ DFS_rec(V,E,'a','t')
    ↪ DFS_rec(V,E,'c','t')
```

Implementation : demo

⏩ Skip Demo



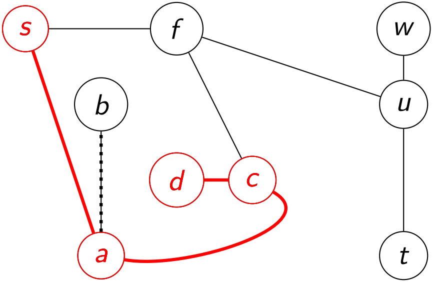
visited

a :	*True*
b :	*True*
c :	*True*
d :	
f :	
s :	*True*
t :	
u :	
w :	

```
DFS_rec(V,E,'s','t')
↪ DFS_rec(V,E,'a','t')
   ↪ DFS_rec(V,E,'c','t')
```

Implementation : demo

⏩ Skip Demo



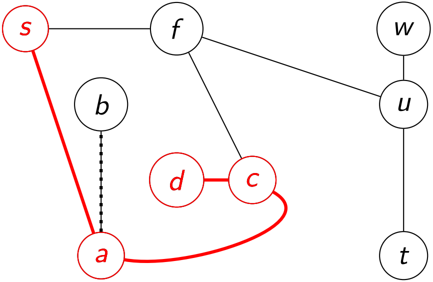
visited

a :	*True*
b :	*True*
c :	*True*
d :	
f :	
s :	*True*
t :	
u :	
w :	

```
DFS_rec(V,E,'s','t')
↪ DFS_rec(V,E,'a','t')
    ↪ DFS_rec(V,E,'c','t')
```

Implementation : demo

⏩ Skip Demo



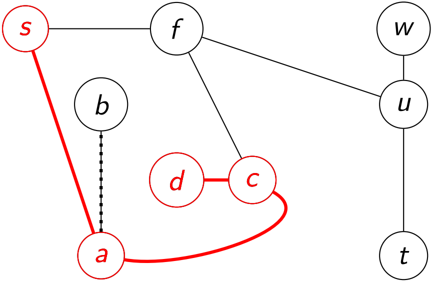
visited

a : *True*
b : *True*
c : *True*
d :
f :
s : *True*
t :
u :
w :

```
DFS_rec(V,E,'s','t')
↪ DFS_rec(V,E,'a','t')
    ↪ DFS_rec(V,E,'c','t')
        ↪ DFS_rec(V,E,'d','t')
```

Implementation : demo

⏩ Skip Demo



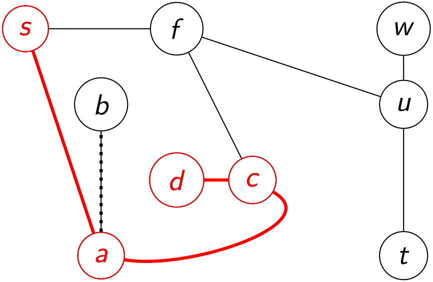
visited

a : *True*
b : *True*
c : *True*
d : *True*
f :
s : *True*
t :
u :
w :

```
DFS_rec(V,E,'s','t')
↪ DFS_rec(V,E,'a','t')
    ↪ DFS_rec(V,E,'c','t')
        ↪ DFS_rec(V,E,'d','t')
```

Implementation : demo

⏩ Skip Demo



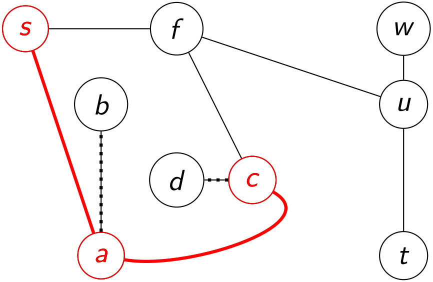
visited

a :	*True*
b :	*True*
c :	*True*
d :	*True*
f :	
s :	*True*
t :	
u :	
w :	

```
DFS_rec(V,E,'s','t')
↪ DFS_rec(V,E,'a','t')
   ↪ DFS_rec(V,E,'c','t')
      ↪ DFS_rec(V,E,'d','t') : False
```

Implementation : demo

⏩ Skip Demo



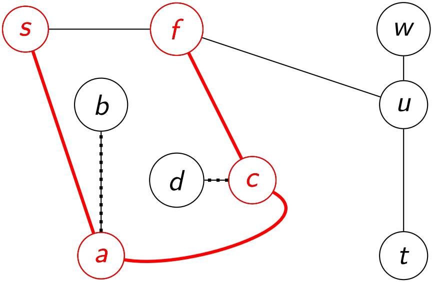
visited

a :	*True*
b :	*True*
c :	*True*
d :	*True*
f :	
s :	*True*
t :	
u :	
w :	

```
DFS_rec(V,E,'s','t')
↪ DFS_rec(V,E,'a','t')
    ↪ DFS_rec(V,E,'c','t')
```

Implementation : demo

⏩ Skip Demo



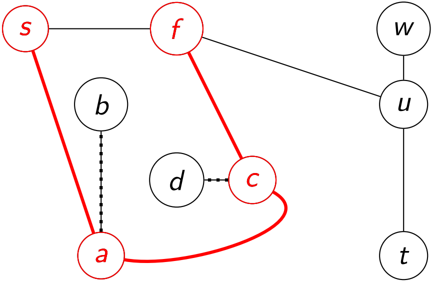
visited

a : *True*
b : *True*
c : *True*
d : *True*
f :
s : *True*
t :
u :
w :

```
DFS_rec(V,E,'s','t')
↪ DFS_rec(V,E,'a','t')
   ↪ DFS_rec(V,E,'c','t')
```

Implementation : demo

⏩ Skip Demo



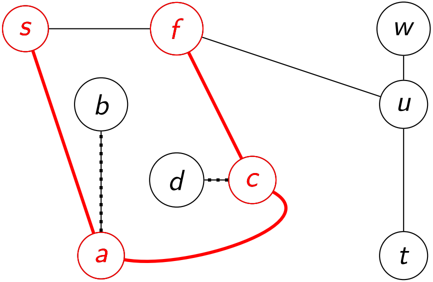
visited

a :	*True*
b :	*True*
c :	*True*
d :	*True*
f :	
s :	*True*
t :	
u :	
w :	

```
DFS_rec(V,E,'s','t')
↪ DFS_rec(V,E,'a','t')
   ↪ DFS_rec(V,E,'c','t')
      ↪ DFS_rec(V,E,'f','t')
```

Implementation : demo

⏩ Skip Demo



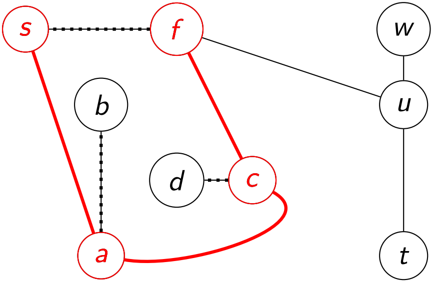
visited

a :	$True$
b :	$True$
c :	$True$
d :	$True$
f :	$True$
s :	$True$
t :	
u :	
w :	

```
DFS_rec(V,E,'s','t')
↪ DFS_rec(V,E,'a','t')
    ↪ DFS_rec(V,E,'c','t')
        ↪ DFS_rec(V,E,'f','t')
```

Implementation : demo

⏩ Skip Demo



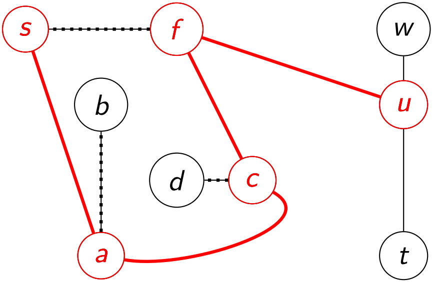
visited

a : True
b : True
c : True
d : True
f : True
s : True
t :
u :
w :

```
DFS_rec(V,E,'s','t')
↪ DFS_rec(V,E,'a','t')
   ↪ DFS_rec(V,E,'c','t')
      ↪ DFS_rec(V,E,'f','t')
```

Implementation : demo

⏩ Skip Demo



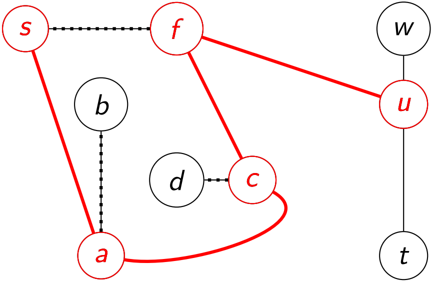
visited

a :	*True*
b :	*True*
c :	*True*
d :	*True*
f :	*True*
s :	*True*
t :	
u :	
w :	

```
DFS_rec(V,E,'s','t')
↪ DFS_rec(V,E,'a','t')
    ↪ DFS_rec(V,E,'c','t')
        ↪ DFS_rec(V,E,'f','t')
```

Implementation : demo

⏩ Skip Demo



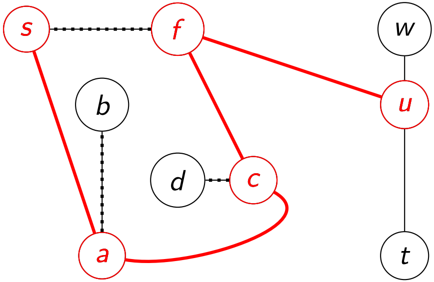
visited

a :	*True*
b :	*True*
c :	*True*
d :	*True*
f :	*True*
s :	*True*
t :	
u :	
w :	

```
DFS_rec(V,E,'s','t')
↪ DFS_rec(V,E,'a','t')
    ↪ DFS_rec(V,E,'c','t')
        ↪ DFS_rec(V,E,'f','t')
            ↪ DFS_rec(V,E,'u','t')
```

Implementation : demo

⏩ Skip Demo



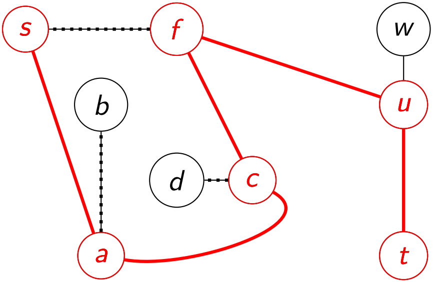
visited

a :	*True*
b :	*True*
c :	*True*
d :	*True*
f :	*True*
s :	*True*
t :	
u :	*True*
w :	

```
DFS_rec(V,E,'s','t')
↪ DFS_rec(V,E,'a','t')
    ↪ DFS_rec(V,E,'c','t')
        ↪ DFS_rec(V,E,'f','t')
            ↪ DFS_rec(V,E,'u','t')
```

Implementation : demo

⏩ Skip Demo



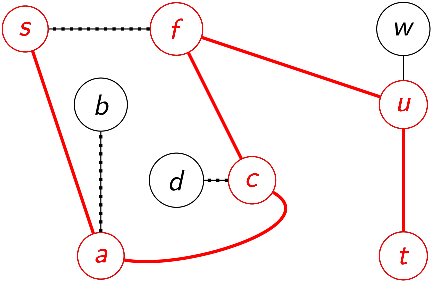
visited

a :	*True*
b :	*True*
c :	*True*
d :	*True*
f :	*True*
s :	*True*
t :	
u :	*True*
w :	

```
DFS_rec(V,E,'s','t')
↪ DFS_rec(V,E,'a','t')
   ↪ DFS_rec(V,E,'c','t')
      ↪ DFS_rec(V,E,'f','t')
         ↪ DFS_rec(V,E,'u','t')
```

Implementation : demo

⏩ Skip Demo



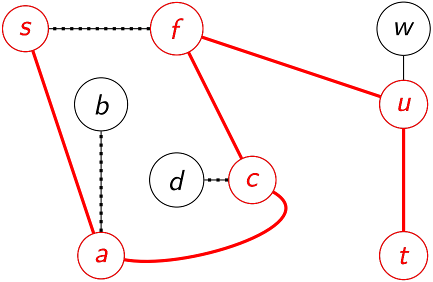
visited

a :	*True*
b :	*True*
c :	*True*
d :	*True*
f :	*True*
s :	*True*
t :	
u :	*True*
w :	

```
DFS_rec(V,E,'s','t')
↪ DFS_rec(V,E,'a','t')
   ↪ DFS_rec(V,E,'c','t')
      ↪ DFS_rec(V,E,'f','t')
         ↪ DFS_rec(V,E,'u','t')
            ↪ DFS_rec(V,E,'t','t')
```

Implementation : demo

⏩ Skip Demo



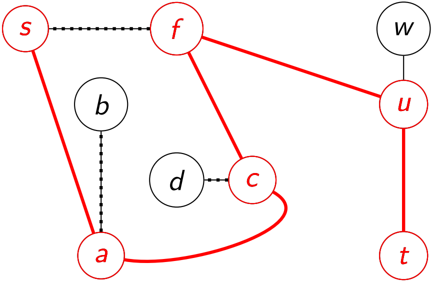
visited

a : *True*
b : *True*
c : *True*
d : *True*
f : *True*
s : *True*
t : *True*
u : *True*
w :

```
DFS_rec(V,E,'s','t')
↪ DFS_rec(V,E,'a','t')
    ↪ DFS_rec(V,E,'c','t')
        ↪ DFS_rec(V,E,'f','t')
            ↪ DFS_rec(V,E,'u','t')
                ↪ DFS_rec(V,E,'t','t')
```

Implementation : demo

⏩ Skip Demo



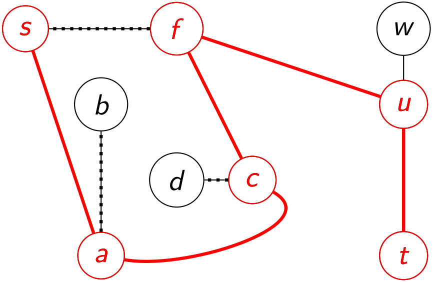
visited

a : *True*
b : *True*
c : *True*
d : *True*
f : *True*
s : *True*
t : *True*
u : *True*
w :

```
DFS_rec(V,E,'s','t')
↪ DFS_rec(V,E,'a','t')
    ↪ DFS_rec(V,E,'c','t')
        ↪ DFS_rec(V,E,'f','t')
            ↪ DFS_rec(V,E,'u','t')
                ↪ DFS_rec(V,E,'t','t') : True
```

Implementation : demo

⏩ Skip Demo



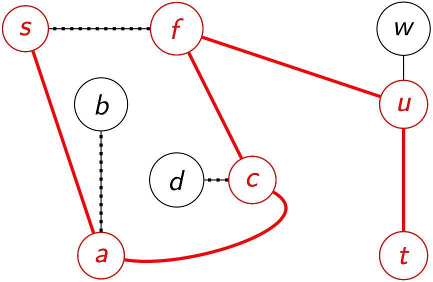
visited

a : True
b : True
c : True
d : True
f : True
s : True
t : True
u : True
w :

```
DFS_rec(V,E,'s','t')
↪ DFS_rec(V,E,'a','t')
    ↪ DFS_rec(V,E,'c','t')
        ↪ DFS_rec(V,E,'f','t')
            ↪ DFS_rec(V,E,'u','t') : True
```

Implementation : demo

⏩ Skip Demo



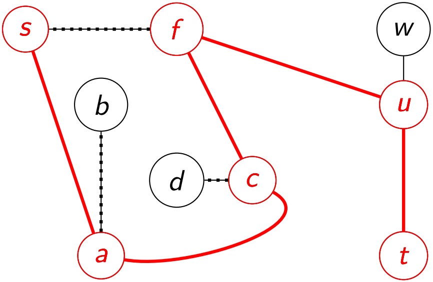
visited

a :	$True$
b :	$True$
c :	$True$
d :	$True$
f :	$True$
s :	$True$
t :	$True$
u :	$True$
w :	

```
DFS_rec(V,E,'s','t')
↪ DFS_rec(V,E,'a','t')
   ↪ DFS_rec(V,E,'c','t')
      ↪ DFS_rec(V,E,'f','t') : True
```

Implementation : demo

⏩ Skip Demo



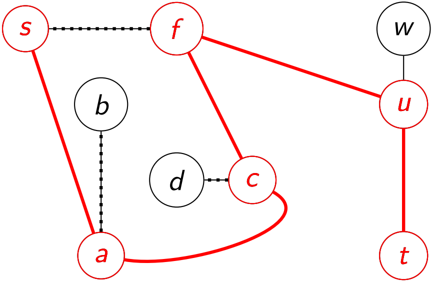
visited

a :	*True*
b :	*True*
c :	*True*
d :	*True*
f :	*True*
s :	*True*
t :	*True*
u :	*True*
w :	

```
DFS_rec(V,E,'s','t')
↪ DFS_rec(V,E,'a','t')
    ↪ DFS_rec(V,E,'c','t') : True
```

Implementation : demo

⏩ Skip Demo

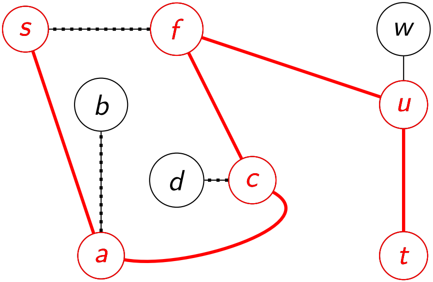


visited

a : *True*
b : *True*
c : *True*
d : *True*
f : *True*
s : *True*
t : *True*
u : *True*
w :

`DFS_rec(V,E,'s','t')`
↪ `DFS_rec(V,E,'a','t')` : True

Implementation : demo

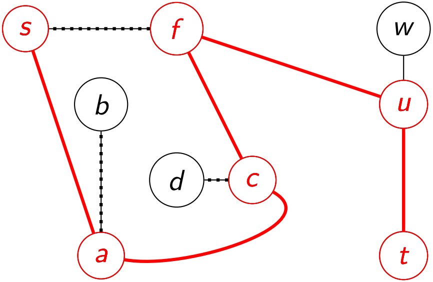


`DFS_rec(V,E,'s','t')` : True

visited

a : *True*
b : *True*
c : *True*
d : *True*
f : *True*
s : *True*
t : *True*
u : *True*
w :

Implementation : demo



visited

a :	*True*
b :	*True*
c :	*True*
d :	*True*
f :	*True*
s :	*True*
t :	*True*
u :	*True*
w :	

`DFS_rec(V,E,'s','t')` : True

Remark

All vertices marked to `True` in `visited` are accessible from s.

We will come back to that property later on. . .

Remove the recursion ?

Remove the recursion ?

Introduction of a stack

- ✓ Turns a recursive algorithm into an iterative one using a stack.
- ➜ The next vertices to be explored are stored in the stack.

Remove the recursion ?

Introduction of a stack

✓ Turns a recursive algorithm into an iterative one using a stack.

➜ The next vertices to be explored are stored in the stack.

Reminder : stack

- Data structure.
- Sequence of elements in which one adds and retrieves elements always on the same end.
- Objects pop out of the stack in reverse order (Last In First Out).

Stack in Python

In Python

Methods `append` and `pop` on a `list`

In the *Algorithmics and Complexity* course

We use two ad-hoc functions that “hide” the implementation :

```
def add_end(x,l):
    l.append(x)

def pop_end(l):
    return l.pop()
```

In computer science

Push and pop methods on stack

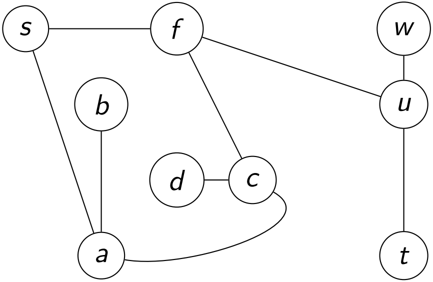
Depth-First Search : iterative implementation

```
def DFS_iter(V,E,s,t):
    lnext = [s]                 # the stack
    reached = { s: True }       # avoid adding multiple times

    while len(lnext)>0:
      n = pop_end(lnext)

      if n==t:
        return True

      for v in neighbours(n,E):
        if not v in reached:
          reached[v] = True
          add_end(v,lnext)      # recursion -> add to stack

    return False
```

Remark : Does not return the founded path → see first tutorial

Iterative implementation : demo

⏩ Skip Demo



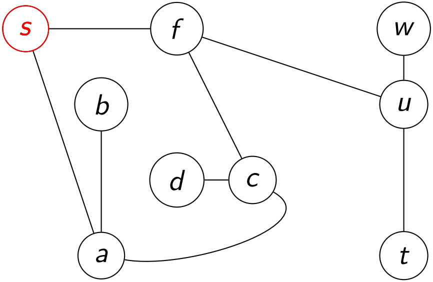
reached

a :
b :
c :
d :
f :
s : *True*
t :
u :
w :

```
lnext = [ 's' ]
n =
```

Iterative implementation : demo

⏩ Skip Demo



reached

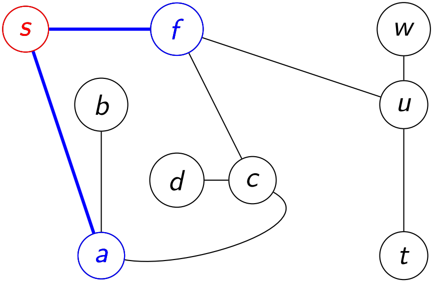
a :
b :
c :
d :
f :
s : *True*
t :
u :
w :

`lnext = []`

`n = 's'`

Iterative implementation : demo

⏩ Skip Demo



reached

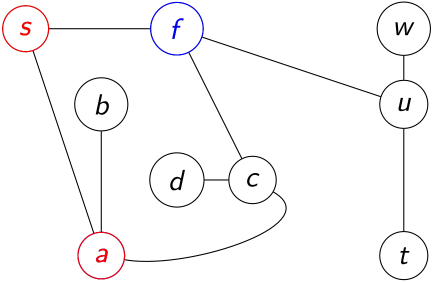
a : *True*
b :
c :
d :
f : *True*
s : *True*
t :
u :
w :

`lnext =` ['f', 'a']

`n =` 's'

Iterative implementation : demo

⏩ Skip Demo



reached

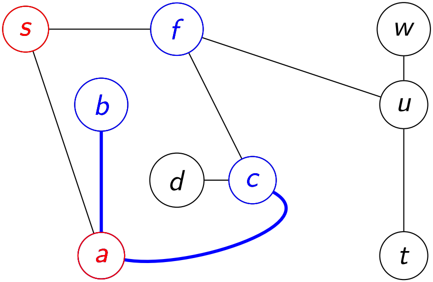
a : *True*
b :
c :
d :
f : *True*
s : *True*
t :
u :
w :

`lnext =` ['f']

`n =` 'a'

Iterative implementation : demo

⏩ Skip Demo



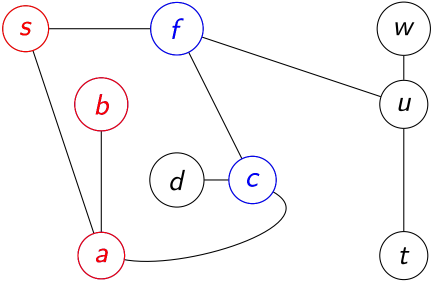
reached

a :	*True*
b :	*True*
c :	*True*
d :	
f :	*True*
s :	*True*
t :	
u :	
w :	

```
lnext = [ 'f', 'c', 'b' ]
n = 'a'
```

Iterative implementation : demo

⏩ Skip Demo



reached

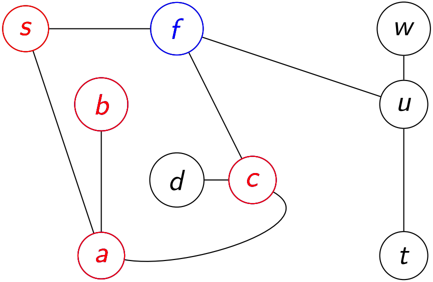
a : *True*
b : *True*
c : *True*
d :
f : *True*
s : *True*
t :
u :
w :

`lnext =` ['f', 'c']

`n =` 'b'

Iterative implementation : demo

⏩ Skip Demo



reached

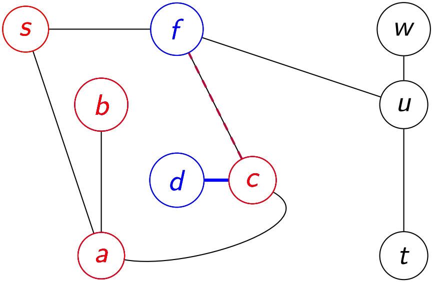
a : *True*
b : *True*
c : *True*
d :
f : *True*
s : *True*
t :
u :
w :

`lnext =` ['f']

`n =` 'c'

Iterative implementation : demo

⏩ Skip Demo



reached

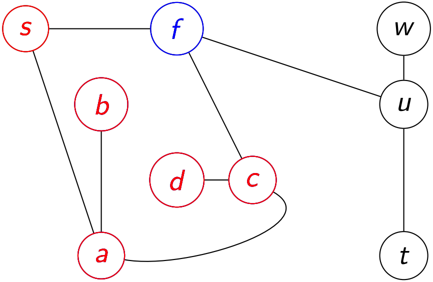
a :	*True*
b :	*True*
c :	*True*
d :	*True*
f :	*True*
s :	*True*
t :	
u :	
w :	

`lnext =` ['f', 'd']

`n =` 'c'

Iterative implementation : demo

⏩ Skip Demo



reached

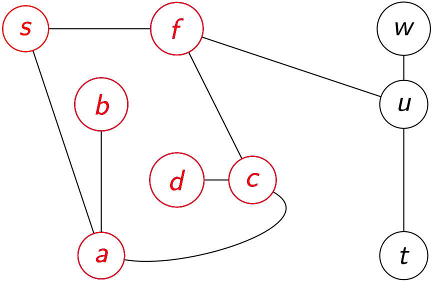
a : *True*
b : *True*
c : *True*
d : *True*
f : *True*
s : *True*
t :
u :
w :

`lnext =` ['f']

`n =` 'd'

Iterative implementation : demo

⏩ Skip Demo



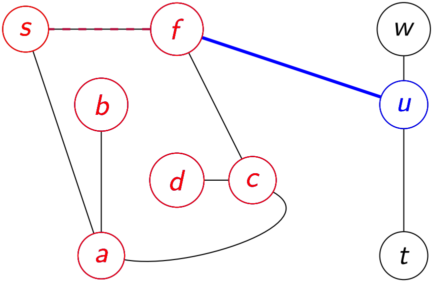
reached

a :	*True*
b :	*True*
c :	*True*
d :	*True*
f :	*True*
s :	*True*
t :	
u :	
w :	

```
lnext = []
n = 'f'
```

Iterative implementation : demo

⏩ Skip Demo



reached

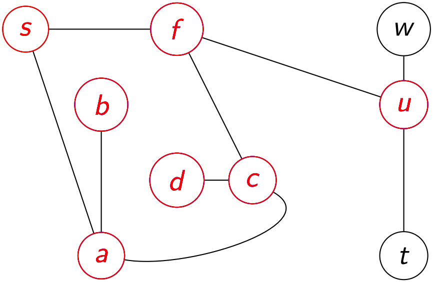
a : *True*
b : *True*
c : *True*
d : *True*
f : *True*
s : *True*
t :
u : *True*
w :

`lnext = [ 'u' ]`

`n = 'f'`

Iterative implementation : demo

⏩ Skip Demo



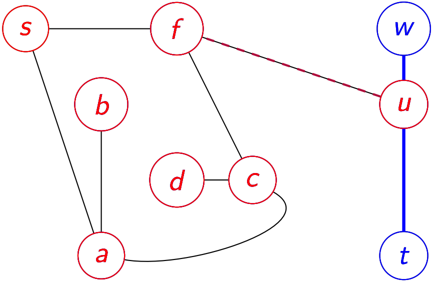
reached

a : *True*
b : *True*
c : *True*
d : *True*
f : *True*
s : *True*
t :
u : *True*
w :

```
lnext = []
n = 'u'
```

Iterative implementation : demo

⏩ Skip Demo



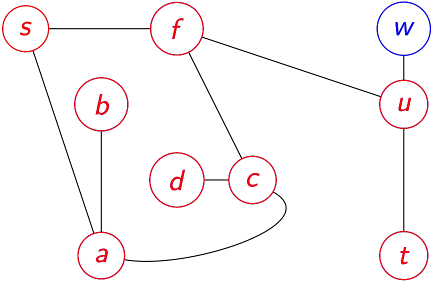
reached

a : *True*
b : *True*
c : *True*
d : *True*
f : *True*
s : *True*
t : *True*
u : *True*
w : *True*

`lnext =` ['w', 't']

`n =` 'u'

Iterative implementation : demo



reached

a : *True*
b : *True*
c : *True*
d : *True*
f : *True*
s : *True*
t : *True*
u : *True*
w : *True*

`lnext =` ['w']

`n =` 't'

Iterative implementation : remarks

Order on the neighbours

To obtain the same traversal as in the recursive version, we added the neighbours in the inversed lexical order !

Iterative implementation : remarks

Order on the neighbours

To obtain the same traversal as in the recursive version, we added the neighbours in the inversed lexical order !

➜ **Exercise :** Run the algorithm with the lexical order on neighbours

Iterative implementation : remarks

Order on the neighbours

To obtain the same traversal as in the recursive version, we added the neighbours in the inversed lexical order !

- ➜ **Exercise :** Run the algorithm with the lexical order on neighbours

Visited/reachable nodes (the remark of the recursive version is always valid)

The nodes with `True` in `reached` are reachable from s.

Plan

Breadth-First Search

Principle

- Visit nodes by order of proximity with the starting vertex.
- Implementation using a queue instead of a stack.
- Difficult to implement in a recursive manner.

Definition of a queue

- Data structure.
- Sequence of elements in which one adds elements in one end and retrieves them from the other.
- Objects pop out of the queue in the same order as they entered (First In First Out).

Queues in Python

In Python

Methods `append` and `pop(0)` on a `list`

➜ with a parameter to remove in the begining instead of the end !

In Algorithmics and Complexity course

Two fonctions “hiding” the implementation :

```
def add_end(x,l):
    l.append(x)

def pop_begin(l):
    return l.pop(0)
```

In computer science

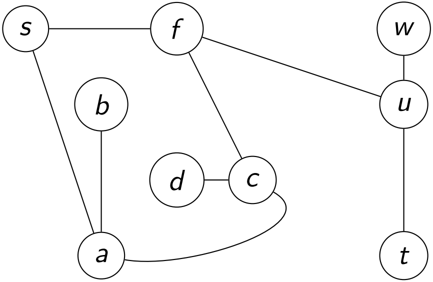
Methods `enqueue` and `dequeue` on a `queue`

Breadth-First Search

```
def BFS(V,E,s,t):
    lnext = [s]                 # the queue
    reached = { s : True }
    while len(lnext)>0:
      n = pop_begin(lnext)
      if n==t:
        return True
      for v in neighbours(n,E):
        if not v in reached:
          reached[v] = True
          add_end(v,lnext)
    return False
```

Iterative implementation : demo

⏩ Skip Demo



reached

a :

b :

c :

d :

f :

s : $True$

t :

u :

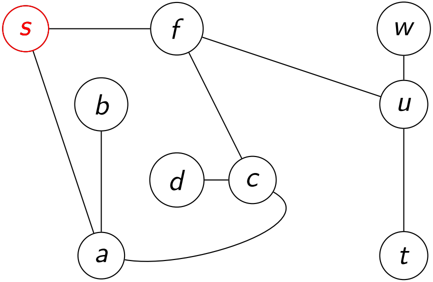
w :

`lnext = ` ['s']

`n =`

Iterative implementation : demo

⏩ Skip Demo



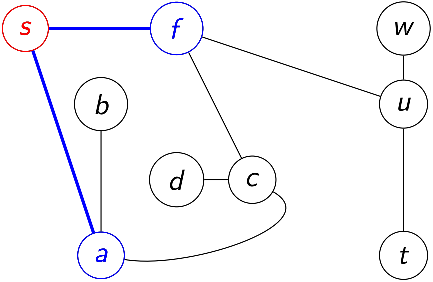
reached

a :	
b :	
c :	
d :	
f :	
s :	$True$
t :	
u :	
w :	

```
lnext =  [ ]
n =  's'
```

Iterative implementation : demo

⏩ Skip Demo



reached

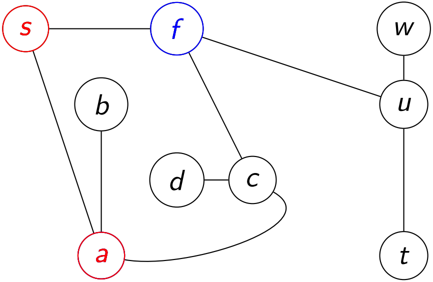
a : *True*
b :
c :
d :
f : *True*
s : *True*
t :
u :
w :

`lnext` = ['a', 'f']

`n` = 's'

Iterative implementation : demo

⏩ Skip Demo



reached

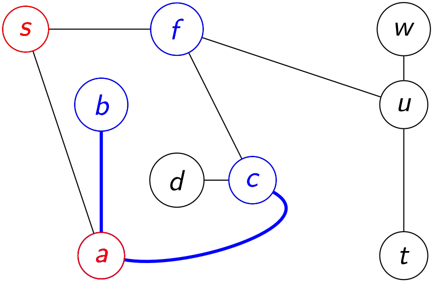
a : *True*
b :
c :
d :
f : *True*
s : *True*
t :
u :
w :

`lnext =` ['f']

`n =` 'a'

Iterative implementation : demo

⏩ Skip Demo



reached

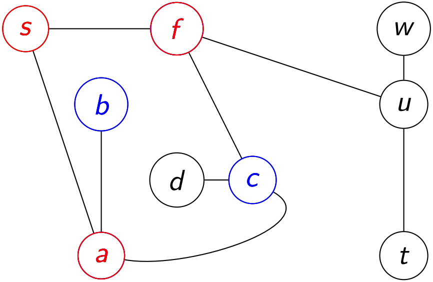
a :	*True*
b :	*True*
c :	*True*
d :	
f :	*True*
s :	*True*
t :	
u :	
w :	

`lnext =` ['f', 'b', 'c']

`n =` 'a'

Iterative implementation : demo

⏩ Skip Demo



reached

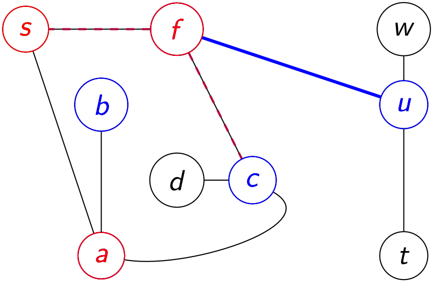
a : *True*
b : *True*
c : *True*
d :
f : *True*
s : *True*
t :
u :
w :

`lnext =` ['b', 'c']

`n =` 'f'

Iterative implementation : demo

⏩ Skip Demo



reached

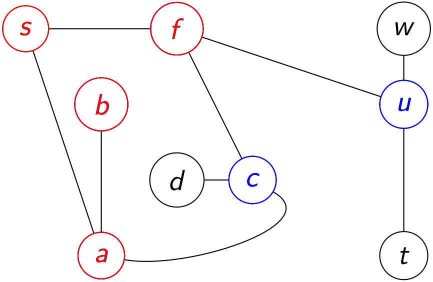
a : *True*
b : *True*
c : *True*
d :
f : *True*
s : *True*
t :
u : *True*
w :

`lnext =` ['b', 'c', 'u']

`n =` 'f'

Iterative implementation : demo

⏩ Skip Demo



reached

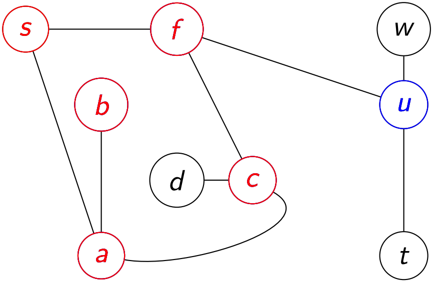
a : *True*
b : *True*
c : *True*
d :
f : *True*
s : *True*
t :
u : *True*
w :

`lnext` = ['c', 'u']

`n` = 'b'

Iterative implementation : demo

⏩ Skip Demo



reached

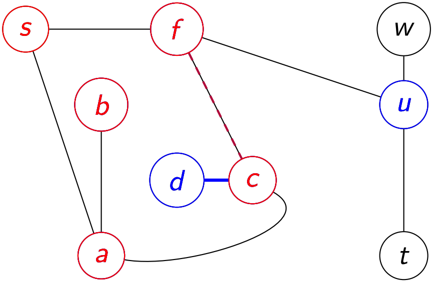
a : *True*
b : *True*
c : *True*
d :
f : *True*
s : *True*
t :
u : *True*
w :

`lnext =` ['u']

`n =` 'c'

Iterative implementation : demo

⏩ Skip Demo



reached

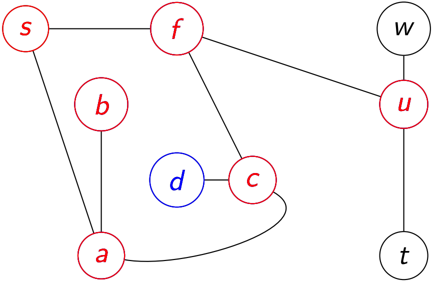
a : *True*
b : *True*
c : *True*
d : *True*
f : *True*
s : *True*
t :
u : *True*
w :

`lnext =` ['u', 'd']

`n =` 'c'

Iterative implementation : demo

⏩ Skip Demo



reached

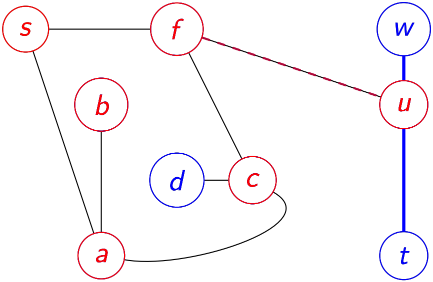
a :	*True*
b :	*True*
c :	*True*
d :	*True*
f :	*True*
s :	*True*
t :	
u :	*True*
w :	

`lnext =` ['d']

`n =` 'u'

Iterative implementation : demo

⏩ Skip Demo



reached

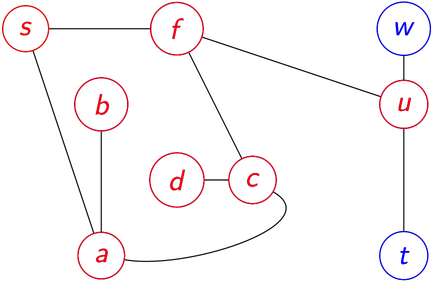
a : *True*
b : *True*
c : *True*
d : *True*
f : *True*
s : *True*
t : *True*
u : *True*
w : *True*

`lnext` = ['d', 't', 'w']

`n` = 'u'

Iterative implementation : demo

▸▸ Skip Demo



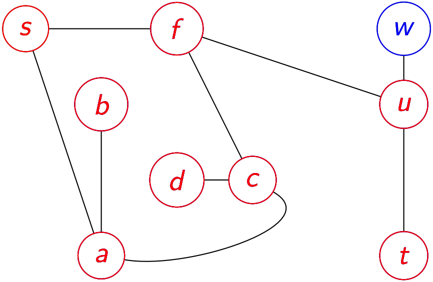
reached

a :	*True*
b :	*True*
c :	*True*
d :	*True*
f :	*True*
s :	*True*
t :	*True*
u :	*True*
w :	*True*

`lnext` = ['t', 'w']

`n` = 'd'

Iterative implementation : demo



reached

a : *True*
b : *True*
c : *True*
d : *True*
f : *True*
s : *True*
t : *True*
u : *True*
w : *True*

```
lnext = [ 'w' ]
n = 't'
```

Plan

1. Graph-based problems
2. Depth-First Search
3. Breadth-First Search
4. Complexity
    - Principle
    - Complexity of iterative search
    - Complexity with data structures
5. Connectivity
6. Conclusion

How to evaluate the algorithm ?

Complexity analysis

Complexity analysis of a algorithm consists in studying the amount of resources (time and space) required to run the algorithm.

Warning

Not to be confused with *complexity theory*, which studies the inherent difficulty of problems (and see later in this algorithmics course).

How to evaluate the algorithm ?

Complexity analysis

Complexity analysis of a algorithm consists in studying the amount of resources (time and space) required to run the algorithm.

Warning

Not to be confused with *complexity theory*, which studies the inherent difficulty of problems (and see later in this algorithmics course).

Utilization

Compare algorithms independently from the implementation, the processor, the memory, the programming language. . .

Calculation of complexity

- The complexity of an algorithm depends on the size of an instance.

  Number of elements in a list, number of vertices and edges in a graph. . .

Calculation of complexity

- The complexity of an algorithm depends on the size of an instance.

  Number of elements in a list, number of vertices and edges in a graph. . .

- Count the number of elementary operations, *i.e.* whose cost does **not** depend on the size of the instance.

```
def contains(T,x):
    i = 0
    while i<len(T) and T[i]!=x:
        i = i + 1
    return i<len(T)
```

$\Rightarrow$ at minimum 0 additions and 2 comparisons (if T is empty)
at maximum n additions and $2n + 2$ cmp with `n=len(T)`

Calculation of complexity

- The complexity of an algorithm depends on the size of an instance.

  Number of elements in a list, number of vertices and edges in a graph...

- Count the number of elementary operations, *i.e.* whose cost **does not** depend on the size of the instance.

  ```
  def contains(T,x):
    i = 0
    while i<len(T) and T[i]!=x:
      i = i + 1
    return i<len(T)
  ```

  ⇒ at minimum 0 additions and 2 comparisons (if T is empty)
  at maximum n additions and $2n+2$ cmp with `n=len(T)`

- It is an asymptotic measure, most often a domination, in this case $\mathcal{O}(n)$.

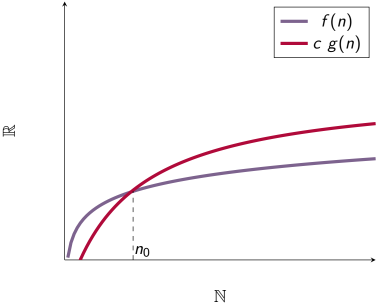
Asymptotical dominance

Definition : asymptotically dominated

A function $f : \mathbb{N} \rightarrow \mathbb{R}$ is asymptotically dominated by another function $g : \mathbb{N} \rightarrow \mathbb{R}$ if and only if :

- $\exists c \in \mathbb{R}^+, \exists n_0 \in \mathbb{N}, \forall n > n_0. \qquad f(n) \leq c\ g(n)$

We write : $f \in \mathcal{O}(g)$



Complexity of DFS_iter or BFS_iter in the worst case

```
def DFS_iter(V,E,s,t):
    lnext = [s]                                 𝒪(1)
    reached = { s: True }                       𝒪(1)
    while len(lnext)>0:                         ×?
      n = pop_...(lnext)                        𝒪(1)
      if n==t:                                  𝒪(1)
        return True
      for v in neighbours(n,E):                 ×?
        if not v in reached:                    𝒪(1)
          reached[v] = True                     𝒪(1)
          add_end(v,lnext)                      𝒪(1)
    return False
```

- We know how to construct add_end, pop_begin and pop_end in $\mathcal{O}(1)$
- The existence checking and writing in a dictionary is in $\mathcal{O}(1)$

Complexity of DFS_iter or BFS_iter in the worst case

```
def DFS_iter(V,E,s,t):
    lnext = [s]                              𝒪(1)
    reached = { s: True }                    𝒪(1)
    while len(lnext)>0:                      ×a
      n = pop_...(lnext)                     𝒪(1)×a
      if n==t:                               𝒪(1)×a
        return True
      for v in neighbours(n,E):              ×b
        if not v in reached:                 𝒪(1)×b
          reached[v] = True                  𝒪(1)×b
          add_end(v,lnext)                   𝒪(1)×b
    return False
```

- We know how to construct add_end, pop_begin and pop_end in $\mathcal{O}(1)$
- The existence checking and writing in a dictionary is in $\mathcal{O}(1)$
- How many loops ?

Complexity of DFS_iter or BFS_iter in the worst case

```
def DFS_iter(V,E,s,t):
    lnext = [s]                              𝒪(1)
    reached = { s: True }                    𝒪(1)
    while len(lnext)>0:                      ×|V|
      n = pop_...(lnext)                     𝒪(1)×|V|
      if n==t:                               𝒪(1)×|V|
        return True
      for v in neighbours(n,E):              ×b
        if not v in reached:                 𝒪(1)×b
          reached[v] = True                  𝒪(1)×b
          add_end(v,lnext)                   𝒪(1)×b
    return False
```

- We know how to construct add_end, pop_begin and pop_end in $\mathcal{O}(1)$
- The existence checking and writing in a dictionary is in $\mathcal{O}(1)$
- How many loops ?
    - $a \rightarrow$ at worst $|V|$ times if all vertices are added in `lnext`

Complexity of DFS_iter or BFS_iter in the worst case

```
def DFS_iter(V,E,s,t):
    lnext = [s]                                   O(1)
    reached = { s: True }                         O(1)
    while len(lnext)>0:                           ×|V|
      n = pop_...(lnext)                          O(1)×|V|
      if n==t:                                    O(1)×|V|
        return True
      for v in neighbours(n,E):                   ×|E|
        if not v in reached:                      O(1)×|E|
          reached[v] = True                       O(1)×|E|
          add_end(v,lnext)                        O(1)×|E|
    return False
```

- We know how to construct add_end, pop_begin and pop_end in $\mathcal{O}(1)$
- The existence checking and writing in a dictionary is in $\mathcal{O}(1)$
- How many loops ?
  a → at worst $|V|$ times if all vertices are added in `lnext`
  b → as many additions as edge number ! as we only add the neighbors.

Complexity of DFS_iter or BFS_iter in the worst case

```
def DFS_iter(V,E,s,t):
    lnext = [s]                              O(1)
    reached = { s: True }                    O(1)
    while len(lnext)>0:                      ×|V|
      n = pop_...(lnext)                     O(1)×|V|
      if n==t:                               O(1)×|V|
        return True
      for v in neighbours(n,E):              ×|E|
        if not v in reached:                 O(1)×|E|
          reached[v] = True                  O(1)×|E|
          add_end(v,lnext)                   O(1)×|E|
    return False
```

- We know how to construct add_end, pop_begin and pop_end in $\mathcal{O}(1)$
- The existence checking and writing in a dictionary is in $\mathcal{O}(1)$
- How many loops ?
  - a → at worst $|V|$ times if all vertices are added in `lnext`
  - b → as many additions as edge number ! as we only add the neighbors.

➜ Complexity of the algorithm in $\mathcal{O}(|V|+|E|) \approx \mathcal{O}(|E|)$ (if G sufficiently dense)

Data Structures

Attention

It depends on the implementation !

Example : *Check if element is in list*

```
if not v in reached
```

- With a list : $\mathcal{O}(|list|)$
- With a dictionary : $\mathcal{O}(1)$

  ➜ *See lab session next week. . .*

Data Structures

Attention

It depends on the implementation !

Example : *Check if element is in list*

```
if not v in reached
```

- With a list : $\mathcal{O}(|list|)$
- With a dictionary : $\mathcal{O}(1)$

➜ *See lab session next week. . .*

Everything is important !

✓ List for successors

✓ Dictionary for visited vertices (`reached`)

➜ And for the graph (edges E) ?

What implementation(s) for graphs ?

Data structures

A grah is an abstract data structure.

➜ How to implement it ?

- What representation for vertices and edges ?
- What data structure groups vertices and edges ?

➜ What are the consequences on the time complexity of algorithms ?

What implementation(s) for graphs ?

Data structures

A grah is an abstract data structure.

- ➜ How to implement it ?
  - What representation for vertices and edges ?
  - What data structure groups vertices and edges ?
- ➜ What are the consequences on the time complexity of algorithms ?

Many possible implementations

1. Adjacency list
2. Adjacency matrix
3. Incidence matrix (we won't see them)

Representation 1 : adjacency list

Idea

For each vertex, store in memory the direct list of its neighbours
(the neighbouring function is often denoted by Γ) :

$$\Gamma : V \to \mathcal{P}(V), \qquad x \mapsto \Gamma(x) = \{y \in V \mid (x, y) \in E\}$$

by using a key-value table.

Representation 1 : adjacency list

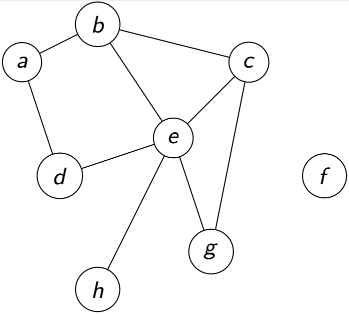
Idea

For each vertex, store in memory the direct list of its neighbours (the neighbouring function is often denoted by Γ) :

$$\Gamma : V \to \mathcal{P}(V), \qquad x \mapsto \Gamma(x) = \{y \in V \mid (x, y) \in E\}$$

by using a key-value table.

x	$\Gamma(x)$
a	$\{b, d\}$
b	$\{a, c, e\}$
c	$\{b, e, g\}$
d	$\{a, e\}$
e	$\{b, c, d, g, h\}$
f	$\{\}$
g	$\{c, e\}$
h	$\{e\}$



Representation 1 : adjacency list

- Memory space in $\mathcal{O}(|E| + |V|)$
- Browsing the set of neighbours of a vertex u in $\mathcal{O}(deg(u))$[1]
  Useful for BFS/DFS, Dijkstra (Lecture 2), Prim (Lecture 3)...
- Facilitate edge storage in the structure (existence by the value 1)
  `{a:{b:1,c:1},...}`
- Check existence of an edge (u, v) in $\mathcal{O}(1)$[2]
- Add an edge in $\mathcal{O}(1)$[2]
- Delete an edge in $\mathcal{O}(1)$[2]

---

[1] Degree of a node = number of adjacent edges
worst case : $deg(u) = |V| - 1$ when u is connected to all.

[2] See lab session for the dictionary

Representation 2 : adjacency matrix

Idea

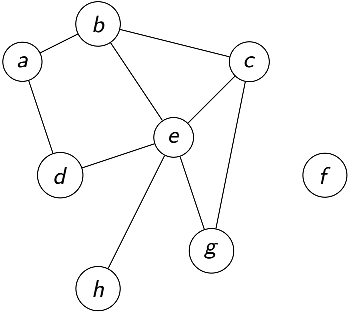
- 2D array indexed by the set $|V|$ of vertices ;
- $tab[i, j] = 1$ if the vertices are linked by an edge else 0.

Representation 2 : adjacency matrix

Idea

- 2D array indexed by the set $|V|$ of vertices ;
- $tab[i, j] = 1$ if the vertices are linked by an edge else 0.

	a	b	c	d	e	f	g	h
a	0	1	0	1	0	0	0	0
b	1	0	1	0	1	0	0	0
c	0	1	0	0	1	0	1	0
d	1	0	0	0	1	0	0	0
e	0	1	1	1	0	0	1	1
f	0	0	0	0	0	0	0	0
g	0	0	1	0	1	0	0	0
h	0	0	0	0	1	0	0	0



Representation 2 : adjacency matrix

- ✗ Memory space in $\mathcal{O}(|V| \times |V|)$
- ✗ Browsing the set of neighbours of a vertex u in $\mathcal{O}(|V|)$

  You need to walk the whole line of the matrix. . .
- ✓ Check existence of an edge (u, v) in $\mathcal{O}(1)$

  And this writes `tab[i][j]` in Python !
- ✓ Add an edge in $\mathcal{O}(1)$
- ✓ Delete an edge in $\mathcal{O}(1)$

A concrete example

BFS algorithm with two different implementations

➜ only the neighbours function changes !

```
def BFS(g,s,t,neighbours):
  lnext = [s]                  # la file
  reached = { s : True }
  while len(lnext)>0:
    n = pop_begin(lnext)
    if n==t:
      return True
    for v in neighbours(n,E):
      if not v in reached:
        reached[v] = True
        add_end(v,lnext)
  return False
```

```
def neighbours_mat(i,g):
  l=[]
  for j in range(len(g[i])):
    if g[i][j]:
      l.append(j)
  return l

def neighbours_list(i,g):
  return g[i]
```

vertices and indexes are identical. . .

➜ Compare the time of `BFS(mat,0,neighbours_mat)` and `BFS(list,0,neighbours_list)` on a reasonably large graph. . .

Complexity of graph search algorithms

Complexity

How to iterate over neighbours ?

➜ The running time complexity of a graph search algorithm depends on the graph implementation !

Complexity of graph search algorithms

Complexity

How to iterate over neighbours ?

➜ The running time complexity of a graph search algorithm depends on the graph implementation !

Adjacency matrix

We iterate over neighbours in $\mathcal{O}(|V|)$

➜ Time complexity of the algorithm is $\mathcal{O}(|V|^2)$

Complexity of graph search algorithms

Complexity

How to iterate over neighbours ?

➜ The running time complexity of a graph search algorithm depends on the graph implementation !

Adjacency matrix

We iterate over neighbours in $\mathcal{O}(|V|)$

➜ Time complexity of the algorithm is $\mathcal{O}(|V|^2)$

Adjacency list

We iterate over neighbours of u in $\mathcal{O}(deg(u))$

➜ Time complexity of the algorithm is $\mathcal{O}(|E|)$ as $2|E| = \sum_{u \in V} deg(u)$

To be remembered about DFS and BFS

- Two similar algorithms
- Find out whether there exists a chain between two vertices (decision)

  and to build one in this case (construction) → TD1 !
- Can detect cycles in the graph*(when a neighbour is already visited).*
- DFS can be implemented by a recursive or an iterative function with a stack.
- BFS uses an Iterative implementation with a queue.
- Theoretical time-complexity in $\mathcal{O}(|E|)$

  ($\mathcal{O}(|V|^2)$ *with matrix and* $\mathcal{O}(|E|)$ *with list).*

To be remembered about DFS and BFS

- Two similar algorithms
- Find out whether there exists a chain between two vertices (decision)

  and to build one in this case (construction) → TD1 !
- Can detect cycles in the graph *(when a neighbour is already visited).*
- DFS can be implemented by a recursive or an iterative function with a stack.
- BFS uses an Iterative implementation with a queue.
- Theoretical time-complexity in $\mathcal{O}(|E|)$

  ($\mathcal{O}(|V|^2)$ *with matrix and* $\mathcal{O}(|E|)$ *with list).*

Connected subgraph

Both algorithms (DFS and BFS) can be used to compute a connected subgraph. . .

Plan

1. Graph-based problems
2. Depth-First Search
3. Breadth-First Search
4. Complexity
5. Connectivity
    - Connected subgraph
    - Algorithm
    - Application
6. Conclusion

Identify connected subgraphs

Remark

If t cannot be reached from s, the algorithm returns False.

Identify connected subgraphs

Remark

If t cannot be reached from s, the algorithm returns False.

- ➜ The dictionary reached gives the set of nodes that can be reached from s. This is called a connected subgraph.

Definition : connected subgraph

In a graph $G = (V, E)$, any maximal subset $V' \subseteq V$ of vertices is called a connected subgraph of G if there exists a chain between any pair of vertices in V'.

Identify connected subgraphs

Remark

If t cannot be reached from s, the algorithm returns False.

- ➜ The dictionary reached gives the set of nodes that can be reached from s. This is called a connected subgraph.

Definition : connected subgraph

In a graph $G = (V, E)$, any maximal subset $V' \subseteq V$ of vertices is called a connected subgraph of G if there exists a chain between any pair of vertices in V'.

Idea ?

- ➜ Could we modifier BFS/DFS to compute a connected subgraph ?

Connected subgraph – Construction

Problem definition

- Input : Given a graph $G = (V, E)$ and a starting vertex $s \in V$
- Question : Build connected subgraph of G that contains s.

Connected subgraph – Construction

Problem definition

- Input : Given a graph $G = (V, E)$ and a starting vertex $s \in V$
- Question : Build connected subgraph of G that contains s.

Solution

We simply need to modify the depth-first search or breadth-first search algorithm so that it does not stop when reaching a given vertex but continues until the stack/queue is empty.

Connected subgraph with BFS

```
def BFS_connex(V,E,s):
    lnext = [s]
    reached = {s}
    while len(lnext)>0:
        n = pop_begin(lnext)
        # we delete the test n==t
        for m in neighbours(n,E):
            if m not in reached:
                reached.add(m)
                lnext.append(m)
    # we return the list of reachable nodes
    return reached
```

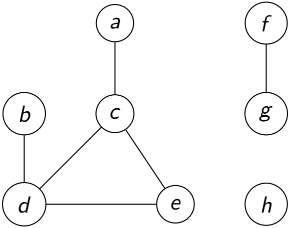
Connected subgraph with BFS

```
def BFS_connex(V,E,s):
    lnext = [s]
    reached = {s}
    while len(lnext)>0:
        n = pop_begin(lnext)
        # we delete the test n==t
        for m in neighbours(n,E):
            if m not in reached:
                reached.add(m)
                lnext.append(m)
    # we return the list of reachable nodes
    return reached
```

Exercise

Modify the DFS and DFS_iter algorithms presented earlier to compute a connected subgraph.

Identify all connected subgraphs of a graph



Problem definition

- Input : Given a graph $G = (V, E)$
- Question : Build a data structure that associates each vertex in V with an integer such that two different vertices s and t are associated to the same value if and only if they are in the same connected subgraph.

Algorithm identifying subgraphs

```
def ident_CC(V,E):
    res={v:-1 for v in V}
    i=0
    for v in res:
        if res[v]==-1:
            cc=BFS_connex(V,E,v)
            for c in cc:
                res[c]=i
            i = i+1
    return res
```

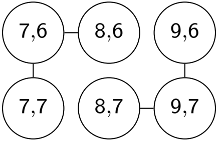
Algorithm identifying subgraphs

```
def ident_CC(V,E):
    res={v:-1 for v in V}
    i=0
    for v in res:
        if res[v]==-1:
            cc=BFS_connex(V,E,v)
            for c in cc:
                res[c]=i
            i = i+1
    return res
```

The time complexity of this algorithm is also $\mathcal{O}(|V| + |E|)$

➜ Each subgraph is visited only once

Counting chromosomes



Algorithm to identify chromosomes

1. Load the grey-scaled image.
2. Apply a threshold to obtain a black-and-white image.
3. Turn the image into a graph where each white pixel is a vertex. Edges connect vertices that correspond to two adjacent white pixels.
4. Use the subgraph identification algorithm

Plan

1. Graph-based problems
2. Depth-First Search
3. Breadth-First Search
4. Complexity
5. Connectivity
6. Conclusion

To be remembered

- Definition of a graph, notation
- Decision, construction and optimization problems
- Instance of a problem
- Definition of an algorithm
- Breadth-First Search and Depth-First Search
  - General algorithm
  - Implementation with a queue or a stack
  - Properties
- Complexity
  - General complexity
  - Complexity with adjacency list
  - Complexity with adjacency matrix
- Application to a concrete problem