Graph Theory

Polytech Tours
2018-2019

TP n°1

I. Graph representation – first algorithms

  I. Graph representation
  II. Graph reading & coding
  III. SearchChain(dep, arr)
  IV. SearchChain_ts(u_0)

TP n°1

TP n°1

I. Graph representation

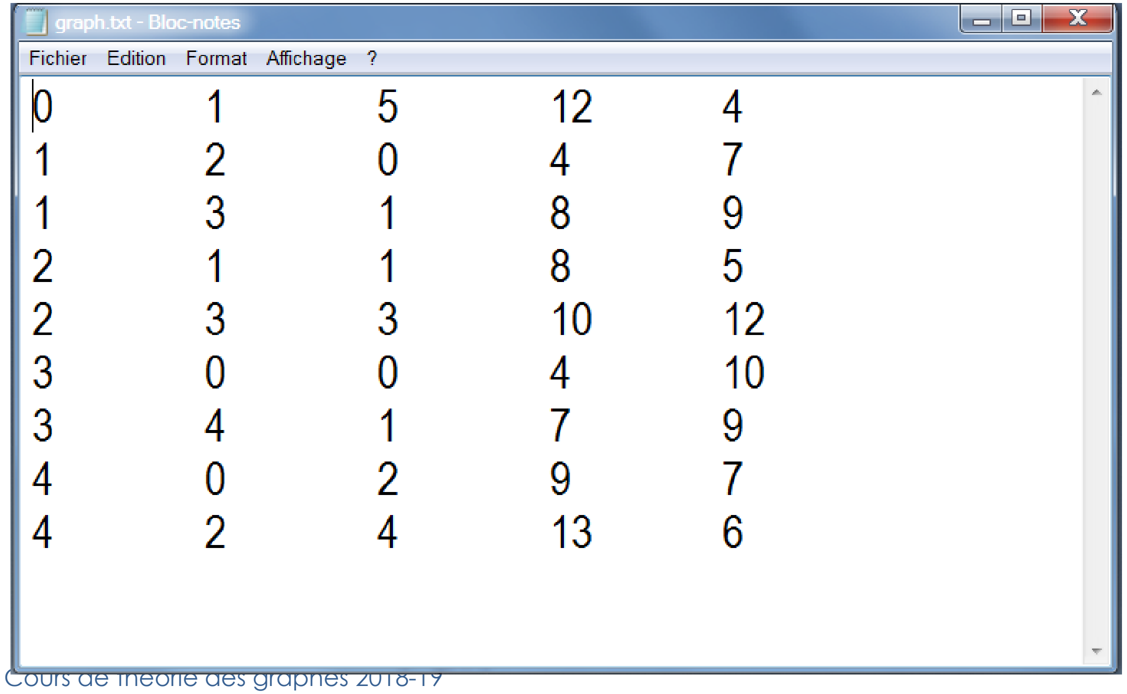
TP n°1

I. Graph representation

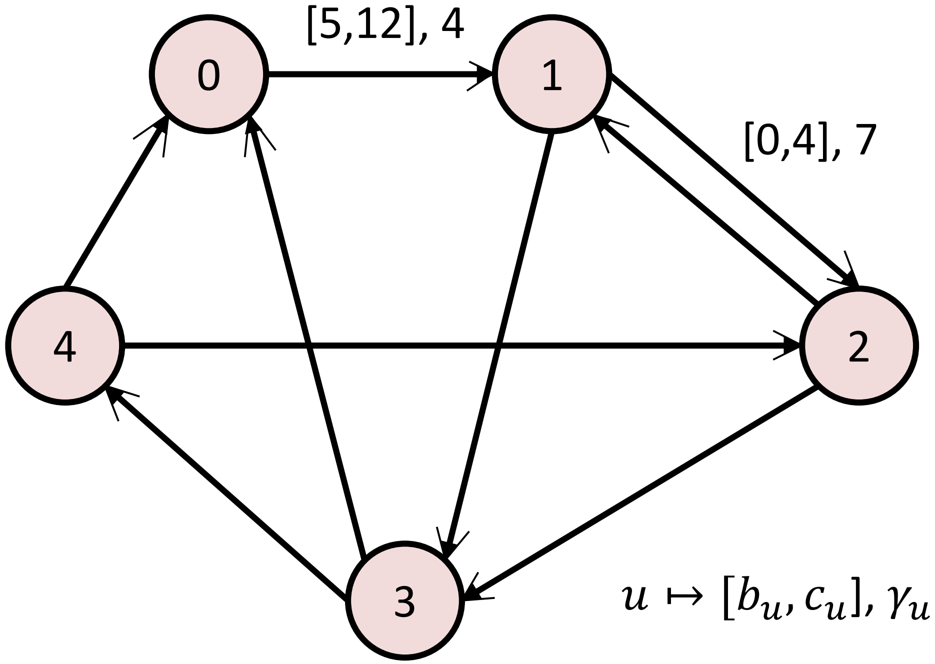
- A graph is described in a text file in a very simple way :

For all the arcs:

```
orig_v - tab - dest_v - tab - min_cap - tab - max_cap - tab - cost
```



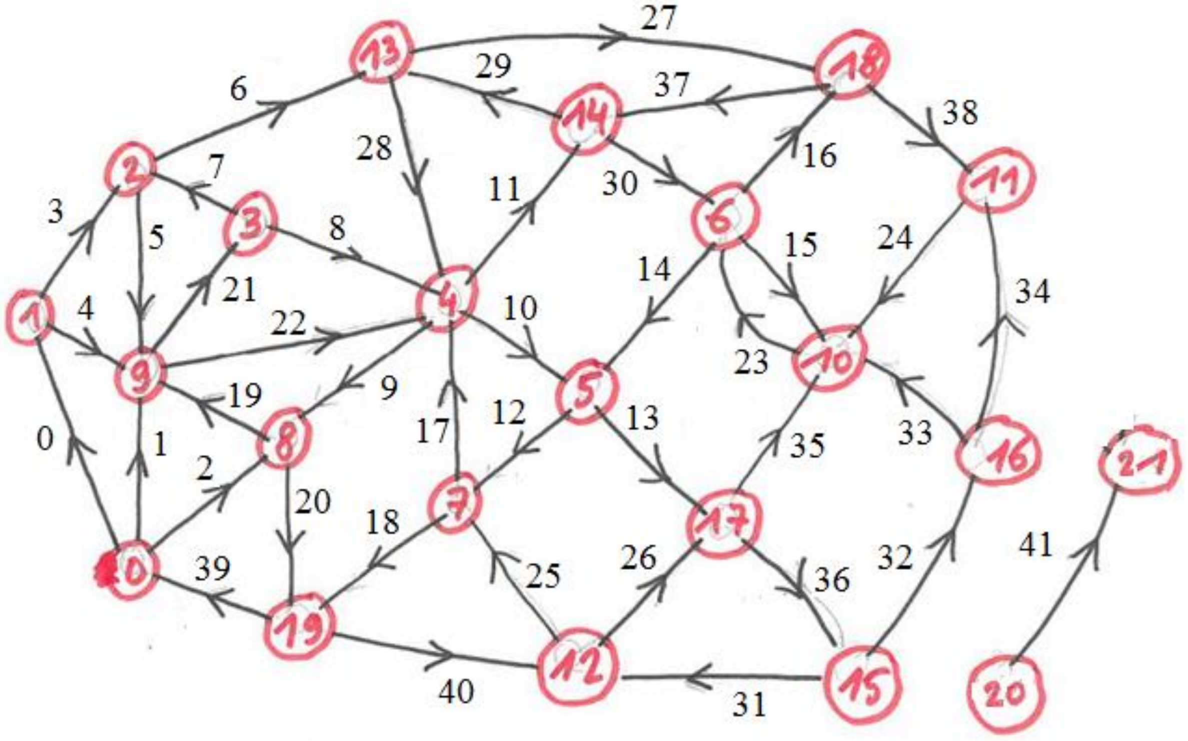
```
0	1	5	12	4
1	2	0	4	7
1	3	1	8	9
2	1	1	8	5
2	3	3	10	12
3	0	0	4	10
3	4	1	7	9
4	0	2	9	7
4	2	4	13	6
```



$u \mapsto [b_u, c_u], \gamma_u$

TP n°1

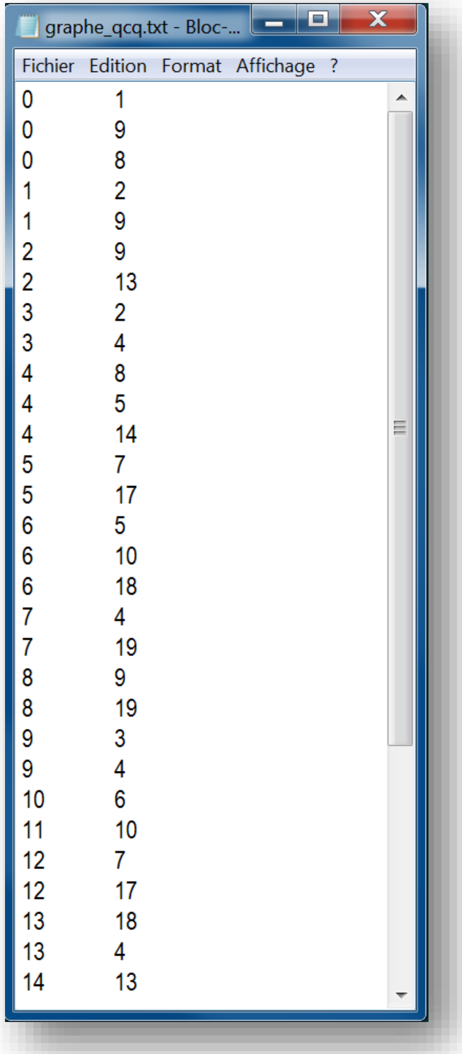
I. Graph representation



$u \mapsto \mathit{its\ number}$

Celene

Graph_TP1.txt



```
0     1
0     9
0     8
1     2
1     9
2     9
2     13
3     2
3     4
4     8
4     5
4     14
5     7
5     17
6     5
6     10
6     18
7     4
7     19
8     9
8     19
9     3
9     4
10    6
11    10
12    7
12    17
13    18
13    4
14    13
```

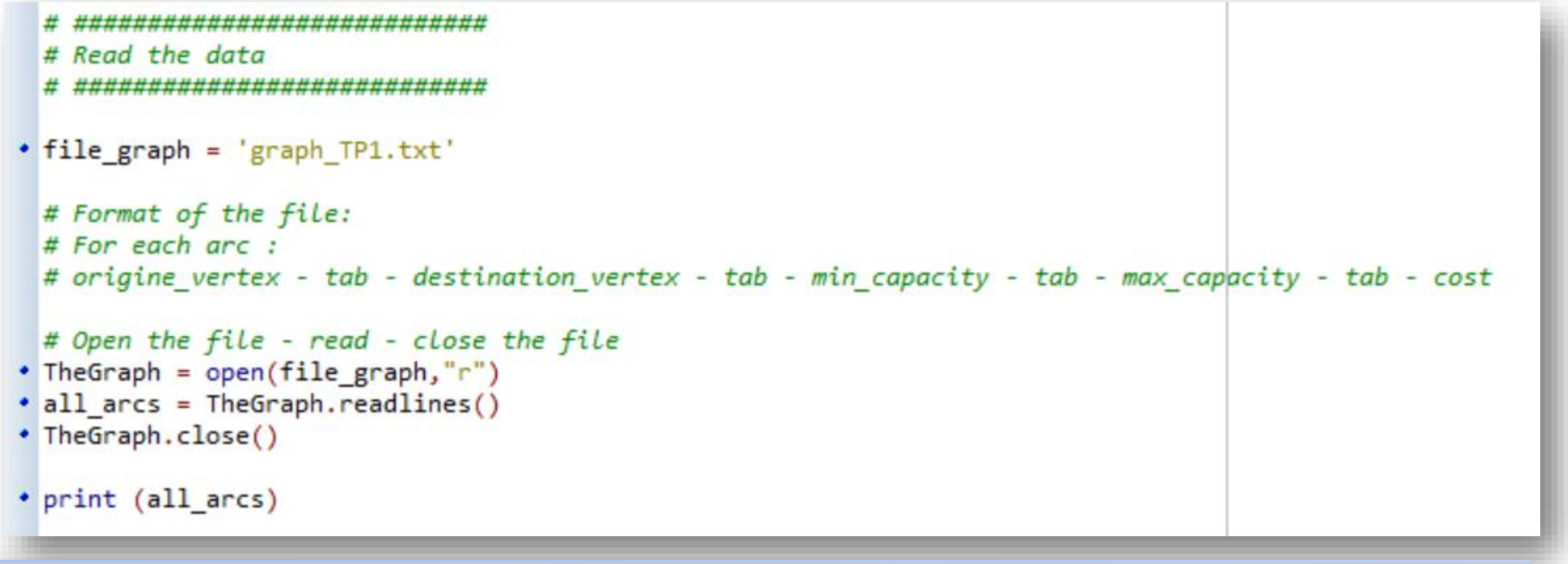
TP n°1

II. Graph reading & coding

TP n°1

II. Graph reading & coding

- 1st step
  - Open the file
  - Read the file
  - Close the file



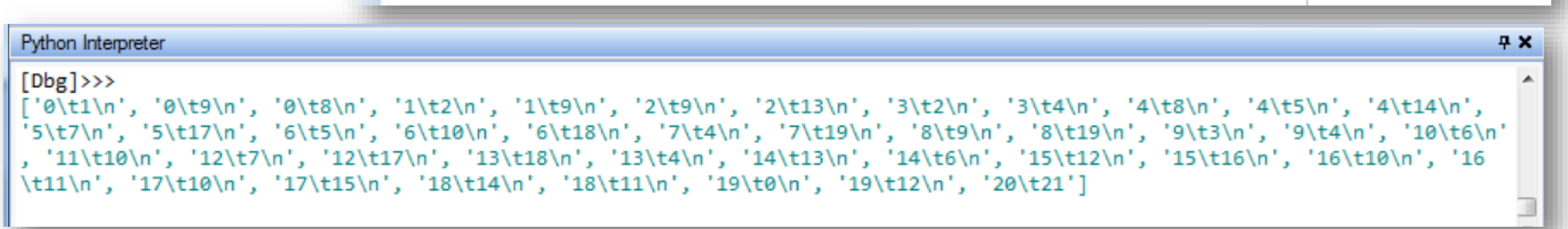
```
# ###############################
# Read the data
# ###############################

file_graph = 'graph_TP1.txt'

# Format of the file:
# For each arc :
# origine_vertex - tab - destination_vertex - tab - min_capacity - tab - max_capacity - tab - cost

# Open the file - read - close the file
TheGraph = open(file_graph,"r")
all_arcs = TheGraph.readlines()
TheGraph.close()

print (all_arcs)
```



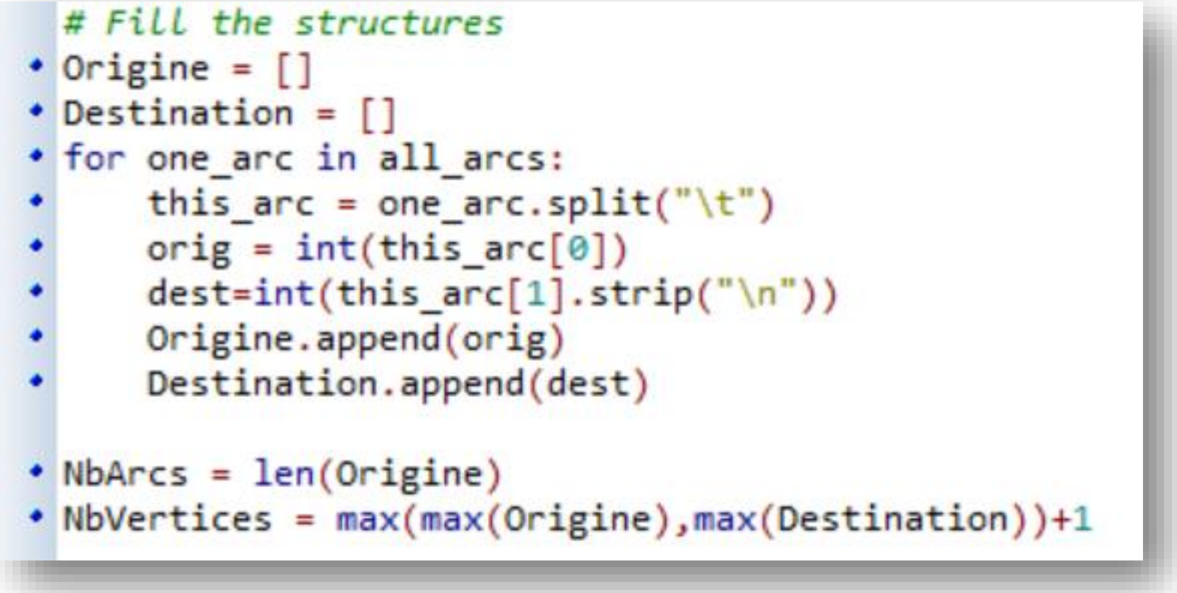
```
[Dbg]>>>
['0\t1\n', '0\t9\n', '0\t8\n', '1\t2\n', '1\t9\n', '2\t9\n', '2\t13\n', '3\t2\n', '3\t4\n', '4\t8\n', '4\t5\n', '4\t14\n',
'5\t7\n', '5\t17\n', '6\t5\n', '6\t10\n', '6\t18\n', '7\t4\n', '7\t19\n', '8\t9\n', '8\t19\n', '9\t3\n', '9\t4\n', '10\t6\n'
, '11\t10\n', '12\t7\n', '12\t17\n', '13\t18\n', '13\t4\n', '14\t13\n', '14\t6\n', '15\t12\n', '15\t16\n', '16\t10\n', '16
\t11\n', '17\t10\n', '17\t15\n', '18\t14\n', '18\t11\n', '19\t0\n', '19\t12\n', '20\t21']
```

TP n°1

II. Graph reading & coding

Coding 1: Origine, Destination

- 2nd step
  - Code the graph
  - We have to read '0\t1\n' and to extract '0' and '1' ⇒ we use **split**
  - We have to delete '\n' ⇒ we use **strip**

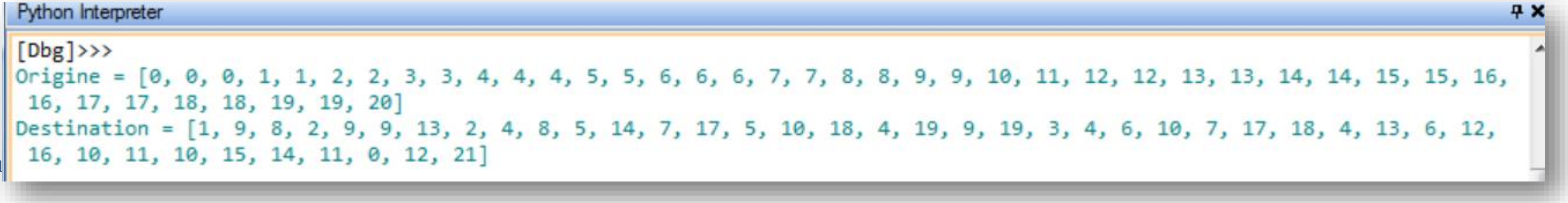


```
# Fill the structures
Origine = []
Destination = []
for one_arc in all_arcs:
    this_arc = one_arc.split("\t")
    orig = int(this_arc[0])
    dest=int(this_arc[1].strip("\n"))
    Origine.append(orig)
    Destination.append(dest)

NbArcs = len(Origine)
NbVertices = max(max(Origine),max(Destination))+1
```

- We define two lists of size M:
  - `Origine`[u] = vertex origine of arc u
  - `Destination`[u] = vertex destination of arc u

At the end, we have:



```
[Dbg]>>>
Origine = [0, 0, 0, 1, 1, 2, 2, 3, 3, 4, 4, 4, 5, 5, 6, 6, 6, 7, 7, 8, 8, 9, 9, 10, 11, 12, 12, 13, 13, 14, 14, 15, 15, 16,
 16, 17, 17, 18, 18, 19, 19, 20]
Destination = [1, 9, 8, 2, 9, 9, 13, 2, 4, 8, 5, 14, 7, 17, 5, 10, 18, 4, 19, 9, 19, 3, 4, 6, 10, 7, 17, 18, 4, 13, 6, 12,
 16, 10, 11, 10, 15, 14, 11, 0, 12, 21]
```

TP n°1

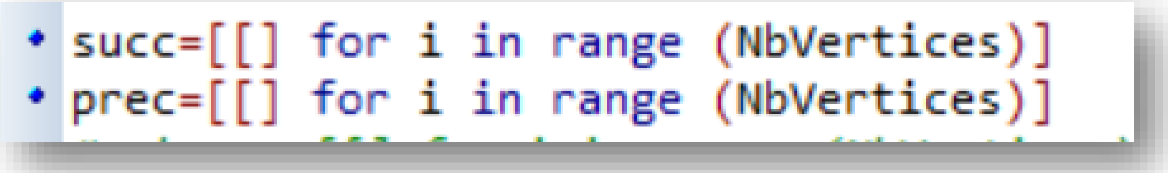
II. Graph reading & coding

Coding 2: prec,succ

Write the code to build `prec` and `succ`

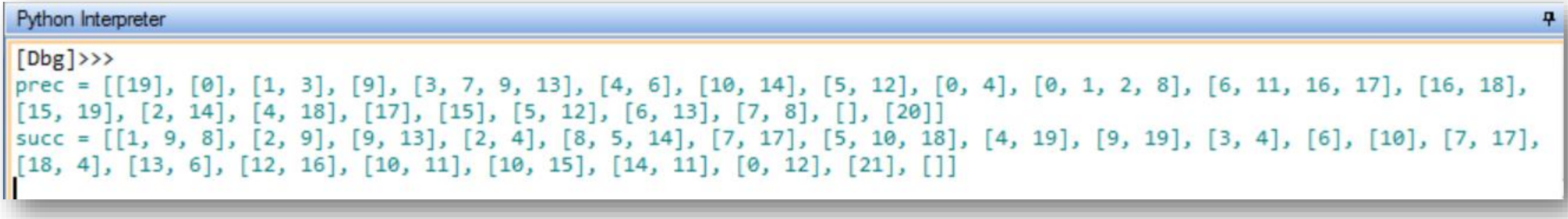
- 3rd step
  - Extra coding

- Lists `Origine` and `Destination` are not easy to use. We introduce two lists of lists.



```
succ=[[] for i in range (NbVertices)]
prec=[[] for i in range (NbVertices)]
```

- Lists `Prec` and `Succ` of size N:
  - `Prec`$[u]$ is the list of predecessors of u
  - `Succ`$[u]$ is the list of successors of u



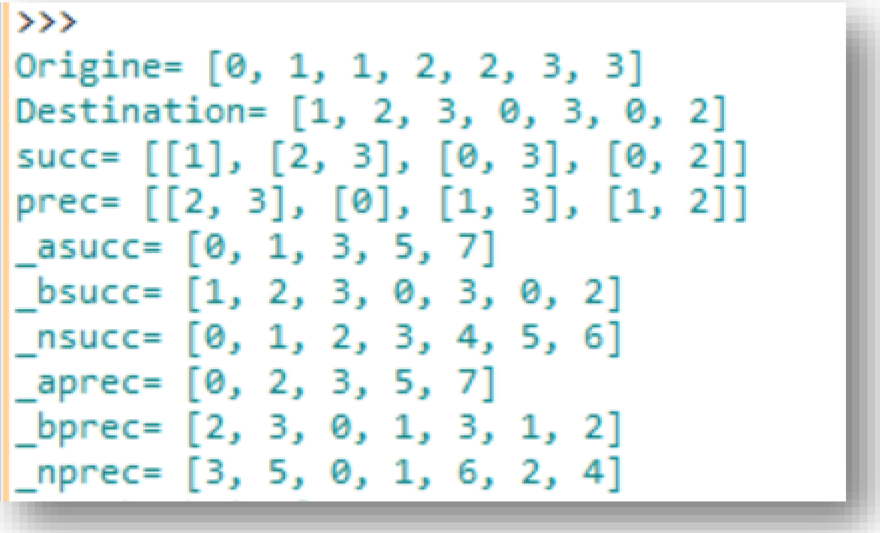
```
[Dbg]>>>
prec = [[19], [0], [1, 3], [9], [3, 7, 9, 13], [4, 6], [10, 14], [5, 12], [0, 4], [0, 1, 2, 8], [6, 11, 16, 17], [16, 18],
[15, 19], [2, 14], [4, 18], [17], [15], [5, 12], [6, 13], [7, 8], [], [20]]
succ = [[1, 9, 8], [2, 9], [9, 13], [2, 4], [8, 5, 14], [7, 17], [5, 10, 18], [4, 19], [9, 19], [3, 4], [6], [10], [7, 17],
[18, 4], [13, 6], [12, 16], [10, 11], [10, 15], [14, 11], [0, 12], [21], []]
```

TP n°1

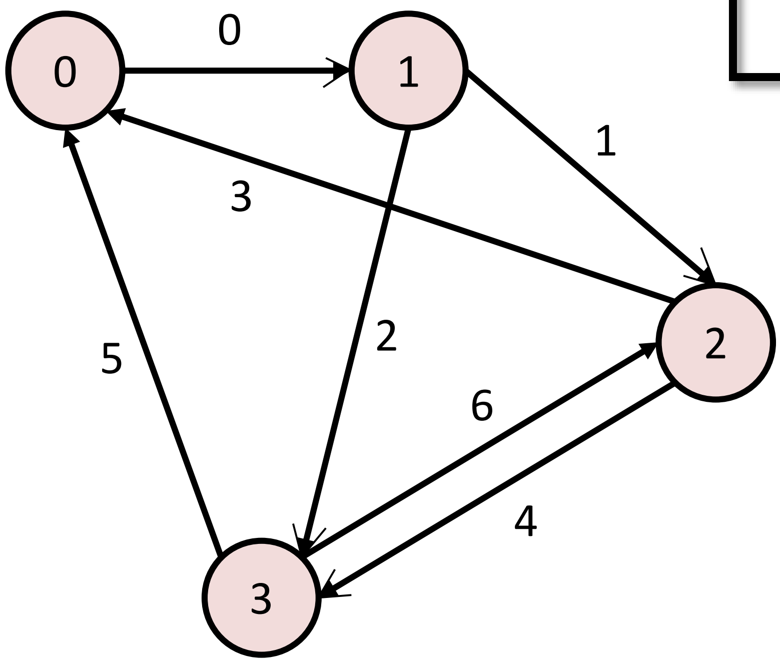
II. Graph reading & coding

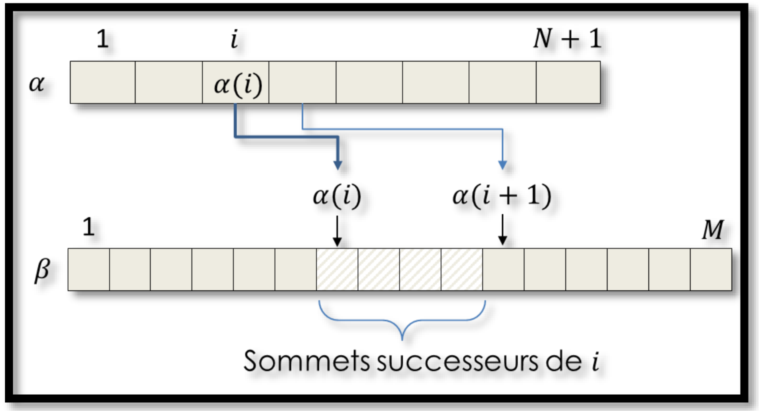
Coding 3: a_prec,b_prec,a_succ,b_succ

- 3rd step
  - Extra coding



```
>>>
Origine= [0, 1, 1, 2, 2, 3, 3]
Destination= [1, 2, 3, 0, 3, 0, 2]
succ= [[1], [2, 3], [0, 3], [0, 2]]
prec= [[2, 3], [0], [1, 3], [1, 2]]
_asucc= [0, 1, 3, 5, 7]
_bsucc= [1, 2, 3, 0, 3, 0, 2]
_nsucc= [0, 1, 2, 3, 4, 5, 6]
_aprec= [0, 2, 3, 5, 7]
_bprec= [2, 3, 0, 1, 3, 1, 2]
_nprec= [3, 5, 0, 1, 6, 2, 4]
```





TP n°1

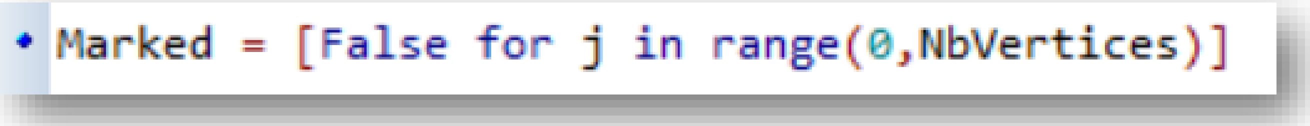
III. SearchChain(dep, arr)

TP n°1

III. SearchChain(dep, arr)

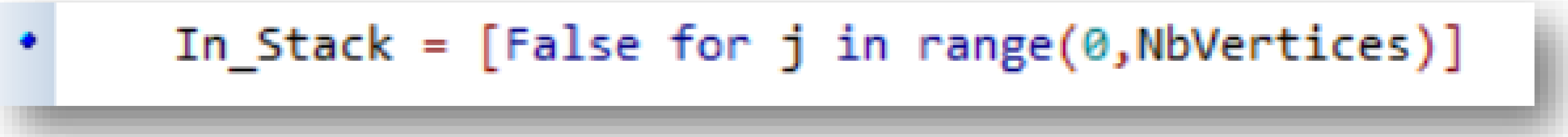
- `SearchChain` (dep, arr)
  - is a routine searching for a chain from vertex dep to vertex arr.
  - returns a boolean equal to `True` if there is a chain, and `False` otherwise.

  - `Marked` is a global list of boolean of size N. `Marked[`j`]=True` means that vertex j is 'marked'.



```
Marked = [False for j in range(0,NbVertices)]
```

  - `In_Stack` is a local list of boolean of size N. `In_Stack[`j`]=True` means that vertex j is already in the stack.



```
    In_Stack = [False for j in range(0,NbVertices)]
```

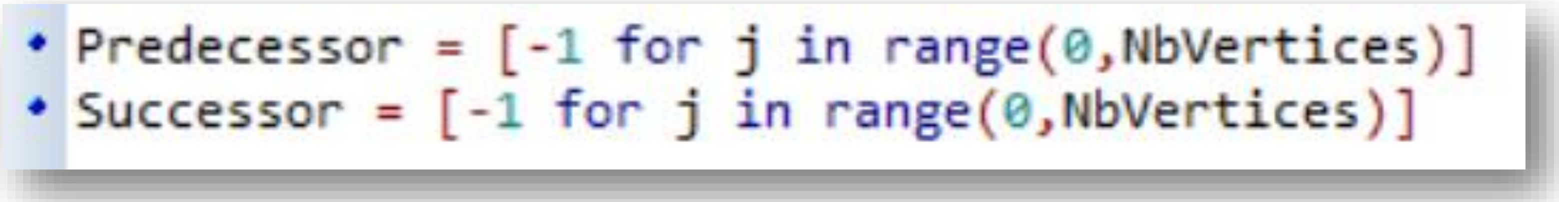
Write the algorithm of `SearchChain`
Then, code it.

TP n°1

III. SearchChain(dep, arr)

- `SearchChain`(dep, arr)
  - Let us introduce now two lists: `Predecessor` and `Successor` of size N.
  - `Predecessor`$[j]=i$, means that in the chain, the predecessor of vertex j, is vertex i, in other words, that arc (i,j) belongs to the chain.
  - `Successor`$[j]=i$, means that in the chain, the successor of vertex j, is vertex i, in other words, that arc (j,i) belongs to the chain.

  - Notice that `Predecessor`$[j]=i \nRightarrow$ `Successor`$[j]=i$ and reciprocally



```
Predecessor = [-1 for j in range(0,NbVertices)]
Successor = [-1 for j in range(0,NbVertices)]
```

Insert these
two lists
in the code
of `SearchChain`

TP n°1

III. SearchChain(dep, arr)

run `SearchChain()`
for several dep and arr
and give the list
of arcs of the chain

TP n°1

IV. SearchChain_ts(u_0)

TP n°1

IV. SearchChain_ts(u_0)

- Copy&paste `SearchChain()` for `SearchChain_ts()`
- This routine has only one parameter called u_0, the index of one arc.
- `SearchChain_ts(`u_0`)` returns `True` if there is a chain from `Destination[`u_0`]` to `Origine[`u_0`]`, `False` otherwise.

Modify
`SearchChain_ts()`
accordingly and test it.