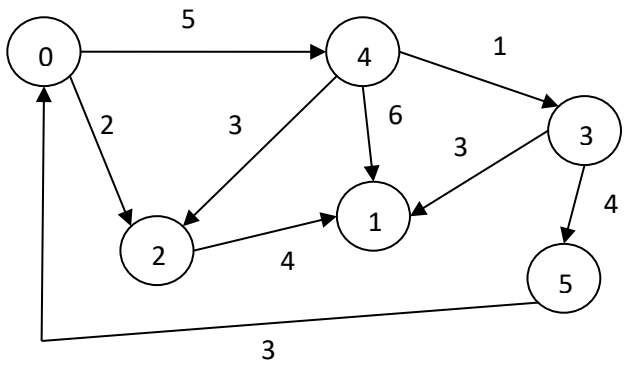
European Funds Knowledge Education Development
Wrocław University of Science and Technology
European Union European Social Fund

ZPR PWr – Zintegrowany Program Rozwoju Politechniki Wrocławskiej

Laboratory – List 10

Introduction

Weighted graph is a structure which can be stored in memory on many ways. One of the simple representation is as a adjacency matrix. Every row present the edged from one vertex to all another vertices. If there is an edge (i, j, w_{ij}) between vertices i and j of the weight w_{ij}, then in i-th row and j-th column it will be stored value w_{ij}. If there are no edge between i and i, so in the i row and j column have to be written a value $+\infty$ (the representation of plus infinity depend on the computer language, sometime it is simple very big value or -1 if there are not negative egdes in the graph). It is also assumed that on diagonal there is value zero.



The above example have to be stored in a matrix 6x6 with values:

	0	1	2	3	4	5
0	0	+∞	2	+∞	5	+∞
1	+∞	0	+∞	+∞	+∞	+∞
2	+∞	4	0	+∞	+∞	+∞
3	+∞	3	+∞	0	+∞	4
4	+∞	6	+∞	1	0	+∞
5	3	+∞	+∞	+∞	+∞	0

Most of the algorithms analyze vertices adjacent to chosen vertex in order in which they are stored in a row. So if in the input problem nodes are labelled not by numbers, it is needed to map original label on number from 0 do *n*-1 (where in is a number of vertices) and also map numbers from the matrix into original labels.

The basic algorithms for a graph is searching nodes. In breadth-first search nodes are visited in order depending on the distance (as number of edges) from starting node. The algorithm uses a queue to remember which nodes have to be visited in a future and colorizing nodes to mark which nodes was previously visited. In depth-first search, from just visited node, the algorithm

tries to visit immediately any adjacent vertex, which was not visited till now. If it is not possible, the algorithm back tract to a node from which it came to the current node and repeats the procedure. There are two ways for implementation of this algorithm: using recursive procedure or iterative with a stack of nodes.

In presented above example of a graph if we select as a starting node 0 the sequence of visited nodes will be as follow:

- For breadth-first search:0,2,4,1,3,5
- For depth-first search: 0,2,1,4,3,5

Remark

In the system constructed till now during laboratories a document will be a node in a graph, and links can be viewed as edges from the node. To do the current task list efficiently you have to use data structures from previous list. A pure array can be used to map a node number into a document, but a hashtable (in Java the HashMap class) is needed to map document names into node number.

List of tasks.

1. Use an implementation of class `Document` from any previous list. Inside the class store the collection of `Link` objects in any chosen way. You can also use any library collection (e.a. `SortedSet`).
2. In the `Main` class use any representation of `Document` objects collection.
3. Implement class `Graph` which implements a graph as two-dimensional array of integers. In the constructor there have to be as a parameter the collection chosen in task 2. In the `Graph` you can store any needed information to map document name into an index and a reverse mapping. An object of the class `Document` is a representation of a vertex, and `Link` are edges from this vertex, but in the Graph class all this information have to be mapped onto integer number.
4. In the `Graph` class implement a `bfs(name)` and `dfs(name)` method which will return a `String` with a sequence of visited document names using **breadth first search** and **depth first search** algorithms. As an argument it will be the name of the starting document. In a case you have many way to go, analyze vertices in lexicographical order.
5. In the `Graph` class implement a `connectedComponents()` method, which will return the number of connected components. Use `DisjointSetForest` class implemented previously. You have to add a function to count number of disjoint sets in this class.

For 100 points present solutions for this list till Week 12.
For 80 points present solutions for this list till Week 13.
For 50 points present solutions for this list till Week 14.
After Week 14 the list is closed.

Appendix
The solution will be automated tested with tests from console of presented below format. The test assumes, that there are documents in a collection.

If a line is empty or starts from `'#'` sign, the line have to be ignored.
In any other case, your program should print an exclamation mark and write (copy) introduced a line and then, depending on the command follow the correct procedure / function.

If the current document is equal **`null`**, the program has to write "no current document".

If a line has a format:
`getdoc <str>`
your program has to choose a document with a name `str`, and all next functions will operate on this document.

If a line has a format:
`ld <name>`
your program has to call a constructor of a document with parameters `name` and `scan` for current position. The `name` have to be a correct identificator (in the constructor the `name` have to be stored in lower case letters). If it is not true, write in one line "`incorrect ID`". If the identificator is correct, the loaded document have to be now the current document. Try to add it to the hashtable. If it is impossible write "error" on the output.

If a line has a format:
`add <str>`
your program has to call `add(new Link(str))` for the current document. If the string is correct, on the output write "`true`". The `str` can be in the same format like a link in this task: only identificator or identificator with a weight in parenthesis. If there is such a name of link in a document, overwrite it.

If a line has a format:
`get str`
your program has to get the link with the reference equals `str` for current document's links and write returned `Link` object. If the `str` is incorrect, write in one line "error".

If a line has a format:
`clear`
your program has to call `clear()` for current document, to remove all links.

If a line has a format:
`show`
your program has to write on the screen the result of caling `toString()` for the current document. The first line has to present text "Document: <name>", the rest of lines will depend on choosen collection of links. **This command will be not executed in automatic tests.**

If a line has a format:
`rem <idStr>`

your program has to remove a link with reference `idStr` for current document and write on the screen returned value in one line.

If a line has a format:
`size`
your program has to write on screen the result of `size()` for current document and write on the screen returned value in one line.

If a line has a format:
`ha`
your program has to end the execution, writing as the last line “END OF EXECUTION”. Every test ends with this line.

If a line has a format:
`bfs <str>`
your program has to execute the `bfs(str)` method, which will return the string of visited documents. Names of documents have to be separated by comma and one space. The returned value present on the output stream.

If a line has a format:
`dfs <str>`
your program has to execute the `dfs(str)` method, which will return the string of visited documents. Names of documents have to be separated by comma and one space. The returned value present on the output stream.

If a line has a format:
`cc`
your program has to execute the `connectedComponents(str)` method, which will return integer value. The returned value present on the output stream.

A simple test for this task:
INPUT:

```
#Test for Lab10
ld a
link=c link=b(4)
eod
ld b
link=g link=e
eod
ld c
link=f
eod
ld g
eod
ld e
eod
ld f
eod
ld x
link=y
eod
ld y
eod
getdoc a
```

```
add v(3)
rem v
getdoc a
get b
size
bfs a
dfs a
cc
getdoc b
clear
bfs a
dfs a
cc
ha
```

OUTPUT:

```
START
!ld a
!ld b
!ld c
!ld g
!ld e
!ld f
!ld x
!ld y
!getdoc a
!add v(3)
true
!rem v
v(3)
!getdoc a
!get b
b(4)
!size
2
!bfs a
a, b, c, e, g, f
!dfs a
a, b, e, g, c, f
!cc
2
!getdoc b
!clear
!bfs a
a, b, c, f
!dfs a
a, b, c, f
!cc
4
!ha
END OF EXECUTION
```