AG44
UTBM – A2018
abderrahim.chariete@utbm.fr, gaber@utbm.fr

Project: Graph Theory and Applications

Introduction

The project consists in making the program that applies different algorithms on the graphs.

We will present in the first part the desired source files structure. In the second part we will define the data structures to use for the representation of the graphs in the code and we will give the steps to construct the graph from a file or from another graph. Then we will detail the algorithms and their complexity. The program is organized into several parts:

Part 1: Graphs representation

1. Graph class
2. Edge class
3. Vertex class
4. Heap Class

Part 2: Graph construction and useful functions

1. Graph representation format: object format, adjacency matrix and adjacency list
2. Constructing the graph from a file
3. Converting the graph formats

Part 3: Algorithms on Graphs

1. Browses the graph (searching the graph)
2. Topological sorting
3. Highly related components
4. Minimum spanning tree
5. Shortest paths

Organization:

Students work in **Binomial**, exceptionally in **trinomial** (I would be very demanding on result in this case).

What to return:

- The complete source code, widely commented, with the Makefile compiler if not using an IDE
- A report presenting the work carried out:
  1. Program structure: organized into classes and functions, role of these class and/or functions,
  2. Qualitative balance of work, difficulties encountered, etc.

The rendering of the project is to be deposited on Moodle. In case of difficulty of deposit, the rendering is to be sent by mail in the form of archive named "AG44_ NAME1_NAME 2_NAME 3", to be sent to abderrahim.chariete@utbm.fr

Watch out for memory leaks! Tests will be done! We recommend that you use the free tools available to check your program.

Evaluating the Project

It is based on the following elements:

- The source program:
  - ✓ Respect for the statement, originality of the proposed work
  - ✓ Technical quality of the program: configurability, classes, functions, display, efficiency, error management...
- Program quality: Indentation, comments and naming
- Documentation provided:
  - ✓ Organization of the program and how to use it
  - ✓ Balance sheet
- Job defense
  - ✓ Demonstration of the program
  - ✓ Individual interrogation of the work carried out

Calendar and rating

Projects are to be returned for the **January 08, 2019 at 11:59 PM** as the last delay (any delay will be penalized). The defense of the work, lasting **7 minutes**, will take place **at the last TD session on 10 January**.

Part 1: Graphs representing

1. Program Structure:

Separate the program into several classes for a better understanding

- ***Graph.h*** and ***Graph.cpp***: contain the graph structure and functions on graphs: construction, conversion, paths...
- ***Edge.h*** and ***Edge.cpp***: contain the attributes and functions on the edges, ...
- ***Vertex.h*** and ***Vertex.cpp***: contain attributes and functions on the vertices, ...
- ***Heap.h*** and ***Heap.cpp***: contain the structure of the heap and the functions that manipulate it. The heap is used for one of the algorithms of search for the shortest path, the *Dijkstra*.

2. File formats for the graph construction

These files contain a description of a graph in the form of an adjacency matrix or adjacency list.

We want graphs **weighted positively** to be able to apply all our algorithms.

a. File format description:

- First line: the number of vertices
- Second line: a letter indicating the type of the graph: *o* for directed graph, *n* for undirected graph
- Third line: a letter indicating the type of representation: *m* adjacency matrix, *l* adjacency list.
- Following lines: the rows of the matrix or the continuation of the lists.

b. File errors:

You must check all the following possible errors:

- If the number of vertices is negative.
- If the type of the graph is unknown.
- If the representation type of the graph is unknown.
- If we do not have numbers on the following lines.
- If the cost is negative.
- If a line is not entirely correct: if there are not enough elements to build the graph or if at the end of the entry of the necessary elements there are still elements on the line.
- If at the end of the complete entry of the graph there are still items in the file.
- For adjacencies lists if a vertex does not exist.

In case of error, you should display an error-message and interrupt the program. If there is no error, you do not need to display anything.

c. Output formats: all the integers shown on the same line are separated by a single **white character** (**comma** "**,**", **semicolon** "**;**", etc.). All lines end with "**\n**".

- **GRAPH format:** this file is created by the function of ***Warshall-Floyd***. The result of this algorithm is a graph represented by an adjacency matrix. This file is the same as the one described earlier.
- **TREE format:** this file is created by the functions on the minimum overlay trees.
  - ✓ First line: the number of edges
  - ✓ Second line: the weight of the tree
  - ✓ Following lines: on each line, one of the arcs of the overlapping tree (indicated by a couple of vertices in ascending order), the different pairs being sorted in ascending order.
- **LIST format:** on each line, a list of vertices. If there are multiple responses, the rows are sorted and ranked in ascending order.
- **SORT format:**
  - ✓ First line: the states achievable at the beginning (vertices without constraints, in ascending order)
  - ✓ Following lines: on each line, the list of vertices (in ascending order) achievable if those of the previous line (and those above) are made (for a given graph, this file is unique).

Part 2: Construction of graphs and useful functions

2.1 Data structures

2.1.1 Graph

A graph must be represented firstly by the object format, then additionally it can be represented by a list or matrix of adjacency.

The structure of the graph contains all the information provided by the input graph file.

As we know the number of vertices of the graph, we allocate exactly the necessary space for the graph.

2.1.2 Adjacency list

The adjacency list is represented by a table of *Nb_vertices* lists. Each cell in a list represents an edge of the graph. Either the list *i* (*i* is the index of the table where this list is located). Each of its cells indicates the vertex and the cost of the edge (*i*, *vertex*).

2.1.3 adjacency matrix

The adjacency matrix is represented by a two-dimensional array:

- A square matrix of size *Nb_vertices* * *Nb_vertices* for the graph directed.
- A matrix triangularine lower of *Nb_vertices* lines for the undirected graph.

2.2 Constructing the graph from a file

We have two types of graph (***directed*** and ***undirected***) and two possible representations (list and matrix of adjacencies). So, we must distinguish four possible constructors for the graph class.

Moreover, to facilitate the application of certain algorithms (such as *Dijkstra*), we verify that the graph is positively weighted.

2.2.1 *File2graph*

Implement function ***file2graph*** checks the first three lines of the file (if error format or unknown type, number of vertices negative or zero), fills the fields of the graph class, and calls one of the graph's construction functions according to the type (directed or not) and the format (matrix or list of adjacencies).

Arguments:

1. Input file.
2. Address of the graph to be built.

Return value:

- -1 if there is an error in the first three lines of the file, we do not have to release the graph since its construction did not begin.
- 0 if there is an error in the graph (memory problem or error in the file). The calling function must free the graph.
- 1 if the graph is built.

Complexity:

The complexity of this function depends on that of the called construction function. It is either in $O(n)$ with n the number of edges is $O(n^2)$ with n the number of vertices.

2.2.2 List of adjacencies:

Implement functions ***graph_o_list*** and ***graph_n_list*** that produce a list of adjacencies.

Argument:

1. Input file.
2. Address of the graph to be built.

return value:

- 0 if there is an error in the graph (problem of memory or error in the file).
- 1 if the graph is built.

These two functions are almost identical to a few conditions.

Algorithm:

```
Lst = new int * [Nb_vertices]                    We allocate a table Lst of lists of size Nb_vertices.
for (int i = 0; I < Nb_vertices; i++) {
        …                                        We read a line, containing a number of pairs (vertex and cost).
        k= 0;                                    If this line is empty we go to the next list ➔ i has no neighbors
        while (k < Nb_pairs) {
                j =...;                          We recover the vertex j.
```

```
                c =...                          We recover the cost of the edge (i, j).
                if (c > 0) {                    We check that the cost is not negative.
                        Lst [i].add(j, c) ;
                }                               We look that there is nothing on the end of line of the file.
        }
        …                                       We also check that there is nothing at the end of the file.
}
```

Complexity:

Tests, assignments, allowances are constant operations. We go through all the lines of the file. At best, we have a complex $O(n)$ with n the number of vertices in the graph. At worst we also have a complex $O(n)$ but with n the number of edges in the graph.

2.2.3 adjacency Matrix

Implement functions *graph_o_matrix* and *graph_n_matrix* that build a matrix of adjacency.

Arguments:

1. Input file.
2. Address of the graph to be built.

Return value:

- 0 if there is an error in the graph (problem of memory or error in the file).
- 1 if the graph is built.

These two functions are almost identical. The only difference is that for the undirected graph we have a variable that decreases the number of columns at each enter to the loop to finally get a triangular matrix.

Algorithm:

```
Adj = new int * [Nb_vertices]                   We allocate a table Adj of tables of size Nb_vertices.
for (int i = 0; I < Nb_vertices; i++ ){
        Adj[i] = new int  [Nb_vertices]         We allocate the lines of the matrix.
        …                                       We read a line from the file.
        for (int j = 0; J < Nb_vertices ; j++ ){
                c = read() ;                    We recover the cost of the edge (i, j).
                Adj[i][j] = 0 ;
                if (c > 0){                     We check that the cost is not negative.
                        Adj[i][j] = c ;
                }                               We look that there is nothing on the end of line of the file.
        }
        …                                       We also check that there is nothing at the end of the file.
}
```

Complexity:

Tests, assignments, allowances are constant operations. We browse the file to fill the entire matrix, so the complexity is in $O(n^2)$ with n the number of vertices in the graph.

2.3 Converting the graph from one format to another

Some algorithms on graphs work faster or are easier to write with a particular type of memory representation. Therefore, it is useful for us to be able to convert the formats of the graphs. These conversions are not difficult to make since at this stage the costs and the vertices are correct. The only problem that can be posed is that of the memory place.

The conversion functions take the same arguments and return the same values.

Arguments:

1. Original graph (source).
2. Address of the converted graph (destination).

Return value:

- 0 if there is a memory problem.
- 1 if the graph has been converted.

At the end of these functions the source graph still exists. It must be released into the function that calls one of the conversion functions since we no longer need it.

2.3.1 from a matrix to a list

The function ***matrix2list*** assigns the fields of the destination graph to those of the source graph by obviously changing the format field. It allocates the list array and calls one of the two conversion functions that follow depending on the type of the graph.

The functions ***o_matrix2list*** and ***n_matrix2list*** are very similar. For the undirected graph, the indices of the triangular matrix are staggered, you must be careful.

Algorithm:

For each box in the matrix, if the cost is not zero we add the vertex and the cost in the corresponding list.

Complexity:

Tests, assignments, allowances are constant operations. We go through the whole matrix, so the complexity is in $O(n^2)$ with n the number of vertices in the graph.

2.3.2 from a list to a matrix

The function ***list2matrix*** populates the fields of the destination graph with those of the source graph by modifying the format field. It allocates the array of tables int and calls one of the two conversion functions that follow depending on the type of the graph.

The functions ***o_list2matrix*** and ***n_list2matrix*** are quite simple. For the undirected graph, we have a variable that decreases to get the triangular matrix.

Algorithm:

For each row in the matrix, we allocate the line. We go through the corresponding list to the line and to element from the list, we recover the cost and we assign it to the corresponding vertex.

Complexity:

We go through all the lists. Each cell represents an edge, so the complexity is in $O(n)$ with n the number of vertices or edges in the graph.

2.4 Various functions on graphs

2.4.1 Release of the graph

If the graph is represented by a list of adjacencies, we release each list by the recursive function ***free_list***, then we release the table of lists. The complexity is so in $O(n)$ with n the number of vertices or edges in the graph.

If the graph is a matrix of adjacency, we release each of its lines and then the array. The complexity is $O(n)$ with n the number of vertices.

2.4.2 Relatedness of the graph

To apply the algorithms on undirected graphs the graphs must be related, whether in one piece.

To check this, we will try to see if we can reach all the vertices by a single path.

As we want to look from the beginning if the graph is related we have written two path functions, one for a list representation of adjacency, one for a matrix representation.

The functions ***pathes_list_prefixe*** and ***pathes_matrix_prefixe*** traverse the graph and count the number of vertices visited.

The function ***graph_connexe*** calls one of the two previous functions according to the format of the graph and compares the number of vertices visited with the number of total vertices in the graph. It returns 1 if the graph is related, 0 otherwise. The complexity of this function depends on the complexity of the function called, i.e. $O(n)$ for a list with n the number of edges and $O(n^2)$ for a matrix with n the number of vertices.

Part 3: Algorithms on Graphs

The algorithms on graphs to implement are:

1. **Graph Searching algorithms:**
   - ***Depth-first-search.h*** and ***Depth-first-search.cpp***.
   - ***Breadth-first search.h*** and ***Breadth-first search.cpp***.
2. **Topological sorting:**
   - ***Topological_Sort.h*** and ***Topological_Sort.cpp***.
3. **Highly related components:**
   - ***Strongly_Related.h*** and ***Strongly_Related.cpp***.
4. **Minimum recovery shaft:**
   - ***Kruskal.h*** and ***Kruskal.cpp***.
   - ***Prim.h*** and ***Prim.cpp***.
5. **Shortest path:**
   - ***Dijkstra.h*** and ***Dijkstra.cpp***.
   - ***Warshall-Floyd.h*** and ***Warshal-Floyd.cpp***.

3.1 undirected Graph

When executing algorithms on undirected graphs, we assume that the graph passed into argument is well undirected and related.

3.1.1 Kruskal algorithm

The ***Kruskal*** function constructs the minimum covering tree of a graph represented by a list of adjacencies. For this we try to build a tree with *Nb_vertex*-1 edges without cycles. As the graph is related, we can build this **Tree**.

Arguments:

1. The graph to study.
2. The output file.

Return value:

- ✓ 0 if the tree cannot be built (memory problem).
- ✓ 1 if the tree is written to the file.

Data Structures:

The tree is represented by a triangular matrix. This representation makes it easier to display the edges in ascending order.
All the edges of the graph are stored in a table. The maximum number of edges in the graph is:

$Max\_Edges = \sum_{i=1}^{Nb\_vertices} i = Nb\_vertices * (Nb\_vertices + 1)/2$

This table contains the vertex numbers of both ends of the edge (the smallest front number) and its cost.

Algorithm:

At the beginning we consider that each vertex constitutes a sub-tree and the total weight of the overlapping tree is initialized to 0.

While we do not have the *Nb_vertices* -1 tree edges:

- We look for the minimum weight edge and remove it from the edge set.
- If the two ends of the edge are not in the same subtree, we can add it to the tree without creating a cycle and merging the two sub-trees. The weight of the covering tree is increased by the cost of the edge.

We write in the file the number of edges, the weight of the tree and the edges (couple of vertices) by traversing the triangular matrix.

The *Kruskal* function uses the function *Merge_trees* to connect the two sub-trees. This function updates the table number that indicates the number of the sub-tree to which a vertex belongs, so its complexity is $O(n)$ with *n* the size of the table is *Nb_vertices*.

Complexity:

We are looking for *Nb_vertices* -1 edges among *Nb_Edges*. So, at worst we have a complexity of $O(Nb\_vertices * Max\_Edges)$ or in $O(n^3)$ with *n* the number of vertices.
If the graph does not have many edges we have a complexity in $O(n^2)$ with *n* the number of vertices because we call *Nb_vertices* time the function *Merge_trees* who is in $O(n)$.

3.1.2 Prim algorithm

Prim algorithm also allows to construct a minimal covering tree, but a graph represented by a triangular matrix (attention should be paid to the indices).

Arguments:

1. The graph to study.
2. A vertex origin of the graph (root).
3. The output file.

Return value:

- ✓ 0 if the tree cannot be built (memory problem).
- ✓ 1 if the tree is written to the file.

Unlike the *Kruskal* function which built several sub-trees, the *Prim* function builds only one tree. It adds the vertices one by one. The graph being related, all the vertices of the graph will be added in the tree.

We distinguish two sets of vertices in this algorithm:

- ✓ Vertices of the covering tree.
- ✓ Untreated vertices (outside the tree).

Data Structures:

- ✓ The tree is represented by a triangular matrix as well.
- ✓ The table *nearest* indicates the closest vertex to another.
- ✓ The table *distance* gives the smallest cost between a vertex of the tree and an unprocessed vertex.

Algorithm:

We consider the past vertex in parameter like the first vertex of the tree. So, we initialize the table *nearest* by this vertex. The *distance* table is so initialized by the root line of the triangular matrix.

While there are still vertices outside the tree:

- We are looking for the untreated vertex closest to the tree. Is *s* this vertex.
- We mark the vertex as a treaty and we add the edge (*s*, *nearest* [*s*]) in the covering tree.
- For all vertices *u* adjacent to the last treated vertex (updated tables *distance* and *nearest*):
    - If the cost between the adjacent vertex *u* and the treaty vertex *s* is less than *distance*[*u*] then we have a new smaller cost between a vertex of the tree and an untreated vertex. We assign to *distance*[*u*] this new smaller cost and the *nearest* [*u*] is the vertex *s*.

We write in the file the number of edges, the weight of the tree and the edges (couple of vertices) by traversing the triangular matrix.

Complexity:

We are looking for the vertex s by route of the distance table and we make this route *Nb_vertices* times to process all the vertices of the graph. So, the complexity of the *Prim* function is $O(n^2)$ with *n* the number of vertices in the graph.

3.2 Directed graph

When executing algorithms on directed graphs, we assume that the graph passed in argument is well directed.

3.2.1 Dijkstra algorithm

Dijkstra algorithm is to find all the shortest paths that originate from a given vertex. This algorithm only applies to graphs valued positively. We apply it to a graph represented by a list of adjacencies.

Arguments:

1. The graph to study.
2. The output file.

Data Structures:

- ✓ The table *distance* that contains the value of the smallest distance between the origin and a vertex.
- ✓ The table *Predecessor* that gives the predecessor of a vertex in the shortest paths.
- ✓ Then we will call *S* all the vertices *i* for which the optimum length of the shortest path of origin *s* to *i* is known and *E* its complementary.

Algorithm:

We initialize the table *distance* to ¥ (we could have initialized *distance*[*i*] at the cost of the edge (*s*, *i*) but it's the same thing when doing it later in the loop). We put *distance*[*s*] to 0.

While there is at least one vertex in *E*:

- We are looking for the vertex *x* from *E* such as *distance*[*x*] is minimum.
- If *distance*[*x*] is worth ¥, the remaining vertices of the entire *E* are inaccessible from the Beginning. We are getting out of the loop.

- We add *x* to the set *S*
- For any vertex *u* adjacent to *x*
    - If the distance between the origin and *u* is more short passing through *x*, we assign it to *distance*[*u*]. We also update *Predecessor*[*u*].

For all *vertices*[*i*] of the graph

- If there is no path from the origin to the vertex *i* we write a blank line in the file.
- Otherwise we write the shortest distance between the origin *s* and the vertex *i* followed by the intermediate vertices on the shortest path. This suite is obtained at using the table *Predecessor*.

Complexity:

We are looking for the shortest path from a given vertex to the other vertices, so we calculate *Nb_vertices* Paths.

For each path we look for the smallest distance among the untreated vertices. The complexity of the function *Dijkstra* is in $O(n^2)$ with *n* the number of vertices in the graph.

3.2.2 Dijkstra algorithm with a heap

Several algorithms have been developed to improve the complexity of the algorithm of *Dijkstra* including introducing a heap structure to choose the vertex *i* entering to the set *S* at every step.

Structure of the Heap:

We represent the heap by an array. Each element of the array contains a vertex and the distance from the path from the origin to this vertex.

We start the array at 1 instead of 0 to facilitate the calculations. Therefore, *HEAP.table*[1] is the root.

We consider the heap as a binary tree, so the vertex *HEAP.table*[*i*] of the heap:

- ✓ Has as left child the: *HEAP.table*[2*i*].
- ✓ Has as right child the: *HEAP.table*[2*i*+ 1].
- ✓ Has for father the: *HEAP.table*[*i*/2].
- ✓ Is a leaf (has no child's) iff 2*i* > *HEAP.NB_Elements*.

We have the functions:

- ✓ *Create_Heap* that allocates the array to a given size as a parameter.
- ✓ *Destroy_Heap* that releases the array.
- ✓ *Is_Empty* that indicates whether the heap is empty or not.
- ✓ *Exchange* that exchanges the values of the two elements of the tree.
- ✓ *Add_Element* that adds the item to the last leaf of the tree (at the end of the table).
- ✓ *Take_Minimum* that takes and removes the root of the tree and reorganizes the tree.
- ✓ *Exchange_Element* that changes the value of an item *e* and reorganizes the *HEAP*.

We have three possible reorganizations:

- ✓ Starting from a leaf (end of the table): we perform the exchanges while the element is lower than its father (or until the root).
- ✓ Starting from the root (beginning of the table):
  While *HEAP.table*[*i*] is not a leaf,
    - we look for the smallest of the two childs of *HEAP.table*[*i*].
    - we exchange if the child is smaller than the father.
- ✓ Starting from a given vertex (anywhere in the table): in *Echanger_element* the new value is smaller than the old one, so we reorganize the heap starting from the vertex modified and we go back until it is in its place (at worst in place of the root).

The Dijkstra algorithm:

Arguments:

1. The graph to study.
2. The output file.

Return value:

- ✓ 0 if the heap cannot be created.
- ✓ 1 if the file has been Filled.

Algorithm:

We initialize the tables and create the HEAP. We add the origin *s* with a distance equal to 0.

While the heap is not empty:

- We take the vertex *x* of the heap as *distance*[*x*] is minimum.
- For all vertices *u* adjacent to *x* we update *distance*[*u*] and *Predecessor*[*u*]. If the vertex does not exist in the heap, we add it otherwise we change only its distance value.

We destroy the HEAP.

- We write the results in the file.

Complexity:

With the heap, the complexity of the algorithm is improved. The function *dijkstra_heap* is in $O(n \log(n))$.

3.2.3 Algorithm of *Warshall- Floyd*

The algorithm of ***Warshall-Floyd*** gives the shortest paths between any What a couple of vertices. We apply it on a graph represented by a matrix of Adjacency.

Arguments:

1. The graph to study.
2. The output File.

Data Structures:

- ✓ A cost matrix containing the cost of the shortest paths
- ✓ A matrix *Father* containing the father vertices in the shortest paths

Algorithm:

We copy the adjacency matrix into the cost matrix. If there are no edges between two vertices we put the cost to infinity.

- For *k* varying from 0 to *Nb_vertices* 1
  - For *i* varying from 0 to *Nb_vertices* 1
    - For *j* varying from 0 to *Nb_vertices* 1
      - If the distance of the path to go from *i* to *j* passing through *k* is smaller than the previous path, we update *distance*[*i*][*j*] and *Predecessor*[*i*][*j*].

We copy the distance matrix to the file.

Complexity:

We are looking for a path for each couple of vertices passing through each vertex. So, we have a complex $O(n^3)$ with *n* the number of vertices in the graph.

3.2.4 Highly related components

A graph is strongly related if for all vertices *u* and *v* there is a path from *u* to *v* and from *v* to *u*. A highly related subgraph with a maximum number of vertices is called highly related component. We will determine these components from a graph represented by a matrix.

The choice of the graph's representation is important because we must traverse the transposed graph (the edges (*i*, *j*) become edges (*j*, *i*)). We will not have to transform the graph because its transposed is easy to obtain by the matrix.

Arguments:

1. The graph to Study.
2. The output files.

Algorithm:

- We perform a ***depth-first-search*** of the graph by noting the vertices in a table at the end of their exploration. If there are still unvisited vertices we restart the depth-first-search on the smallest vertex.
- We perform a second ***depth-first-search*** but on the graph (DFS-Tree) returned first time. We make a prefix search with as starting vertex the last one in the table obtained by the first search. If there are still unvisited vertices we will start the search again with the last unvisited vertex of the table. From this second course we indicate to which related component each vertex belongs by a number.
- We write each related component in the file in ascending order of the vertices.

Complexity:

We do several ***depth-first-search***. The ***depth-first-search*** is carried out in $O(n)$ with *n* the number of vertices. At the end of the search we get a table indicating for each vertex which related component it is part of. To view the components sorted in order we need to go through this array as many times that there are highly related components. So, in the worst-case scenario we have a complex $O(n^2)$ with *n* the number of vertices in the graph.

3.2.5 Tri Topological

Topological sorting indicates the order in which states (*vertices*) of a graph can be realized knowing that some are feasible at the same time (we sort them in this case).

Arguments:

1. The graph to study.
2. The output file.

Return value:

- ✓ 0 if the graph has a cycle, it is not possible to sort.
- ✓ 1 if the file is edited.

As we wish the sequences of vertices in ascending order we will not use a set *S* for untreated vertices (as in *Dijkstra*) that are likely to mix them but rather a marker. So, we choose to browse the table *Predecessor* (which indicates the number of predecessors for a vertex) several times this which implies a complexity that can go up to $O(n^2)$.

Algorithm:

We indicate in a table *Predecessor* the number of predecessors for each vertex.

While we have dealt with all the vertices:

- We are looking in the table *Predecessors* the vertices that do not have predecessors and we display them one by one.
- If we have not found any vertex, then there is a cycle in the graph, we go out of function.
- Otherwise, we mark these vertices and remove the edges associated to them.

Complexity:

We are browsing the table *Predecessor* of size *Nb_vertices* as many times as there are steps in topological sorting. In the worst case, if we have only one vertex per step, we'll have *Nb_vertices* steps and therefore we will have a complexity in $O(n^2)$.

4.2 Implementation of the program

4.2.1 Creating the graph

The graph is created from the input file. If this file does not exist, we use the standard input.

We call the function ***Fichier2graphe*** to create the graph. If there is an error in the construction of the graph we release it and leave the program.

4.2.2 Execution of algorithms

We have two cases to distinguish: directed graphs and undirected graphs.

For each case, the order of execution of the algorithms depends on the format of the graph.

For example, if we have a graph represented by a matrix we first perform the algorithms that apply on the matrices of adjacency then we convert the graph into a list of adjacencies to be able to apply the other algorithms.

So, here is a part of the main function concerning the execution of the algorithms:

1. If the created graph is not directed, we test its connectedness.
   - If the graph is a matrix of adjacency:
     - We call the functions on the undirected graphs represented by a matrix.
     - We convert the graph to a list of adjacencies.
     - We call functions on undirected graphs represented by a list.
   - Otherwise, we perform the same operations in the other case.
2. If the graph is directed:
   - The ***Dijkstra*** algorithm is a part of algorithms on the directed graphs, so we still must test the vertex passed in argument. It is necessary to check that the vertex number is less than the number of vertices in the graph (in case there is no vertex, we have initialized the vertex to -1 so makes the test true).

   We perform the same operations as for the undirected graph by calling in this case the functions on the directed graphs.

To not overload the function ***main***, you're going write four functions that group algorithms according to their type and format:

- ✓ ***Algo_n_matrix***

- ✓ ***Algo_n_list***
- ✓ ***Algo_o_matrix***
- ✓ ***Algo_o_list***

These functions have nearly the same prototype and the same algorithm:

Arguments:

1. The graph to study.
2. A vertex (only for the function ***Algo_o_list*** that calls the function ***Dijkstra*** and so need a vertex).
3. The name of the output file.

Algorithm:

The output file is initialized to the standard output.

For each algorithm:

- If there is a filename, we add the extension and we open the corresponding file.
    - If it there's an error in the opening of the file, we move to the next algorithm.
- We call the function corresponding to the algorithm.

The only mistake we can encounter in calling these functions is the inability to create the output files. This can be a rights issue so if we have an error for one of the output files, the program does not stop and goes to the next algorithm. So, we execute as much algorithms as possible.

Complexity:

The complexity of these functions depends on of the functions called.