- *All codes must be written in one programming language, amongst C++, Python, Java.*
- *Source files must be sent at guillaume.ducoffe@fmi.unibuc.ro.*
- *Students must solve one subject. Partial solutions of multiple subjects will not be taken into account.*
- *All classroom material is allowed. Students may dispose of their own code, but they are not allowed to use others' codes. If two students share, even partially, the same code (and it is not the one provided in the correction), then both students automatically fail.*
- *Codes are tested using files with the following syntax:*
  - the number n of nodes and the number m of edges are given on the first line
  - every other line represents one edge. The two end-vertices of the edge are given and, if the graph is weighted, the line ends with the integer weight of the edge.
  Students may use in their code any implementation for their Graph class. However, they must provide a main function that reads on the terminal a file name, opens the corresponding graph file, transforms the latter in a Graph object, which is used as input for the students' program.
- *The notation is as follows:*
  - compilation 1p
  - execution 1p (for Python programmers, 2p)
  - correct output for the example graph file 2p
  - correctness of the algorithm 2p
  - time complexity (serial) or correct use of parallelism 2p
  - presence of justifications/explanations as comments in the code 2p

---

1) Consider an undirected unweighted graph. In a LexDFS, vertices are numbered from 1 to n (this number is independent of the vertices' ID, it only depends on the order in which vertices are visited). In particular, the source vertex, which is the first visited, is numbered 1. At any moment during the execution, for every unvisited vertex v, we can define its label L(v) as being the list of all its already visited neighbours, ordered by decreasing number. Then, the next vertex to be visited must have a label which is *lexicographically maximal*.

   Implement LexDFS.
   Complexity: O(m+nlog(n))

   Hint: partition refinement can be used, but the new groups created after each refine operation must be all placed at one end of the partition. Furthermore, in doing so, the relative ordering of groups must be preserved.

2) Consider an undirected weighted graph. Bidirectional search is a variation of A*, where one is given a more powerful heuristic function h: namely, h(x,y) is a heuristic estimate of the distance between x and y. There is one source s and one target t. We simultaneously perform a forward search from the source (the same as in A*) *and* a backward search from the target. At every round, the algorithm selects two open vertices x,y such that g(s,x) + h(x,y) + g(y,t) is minimized. For x, one considers its out-going neighbours, while for y one considers its in-going neighbours. Update of g and of open/closed vertices is done as usual.

   Implement Bidirectional search.

Complexity: at most O(n*log(n)) to find and process the next pair x,y.

3) In Algorithm R for computing connected components, the following steps are repeated until no vertex v changes her parent v.p: parent-root-connect; shortcut.

   Implement Algorithm R. The algorithm must also be modified in order to return a (not necessarily rooted) spanning tree for every connected component.

4) Simulate PBFS with 1-dimensional partitioning (PBFS with distributed memory, seen in class). In order to simulate message passing between processors, use auxiliary vectors indexed by the number of running processors (or by the maximum allowed number of such processors, if it is fixed in advance).