- *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 LexDOWN, 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 increasing number. Then, the next vertex to be visited must have a label which is *lexicographically maximal*.

   Implement LexDOWN.
   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. IDA* is a variation of A*, where one maintains an additional parameter F during the algorithm. Roughly, F acts like a threshold, representing an estimate of the distance from the source s to the target (initially, you may take F = h(s)). Then, the only (but important) difference with A* is that vertices x such that f(x) = g(s,x) + h(x) is larger than F are not put in OPEN. If at some point, OPEN is emptied but the target has not been reached, then F is increased, and the neighbourhoods of closed vertices are scanned in order to find new vertices to be (re)opened.

   Implement IDA*.
   Complexity: at most O(log(n)) to find and process the next open vertex x (assuming OPEN is

nonempty; otherwise, the complexity can be higher).

3) In Algorithm H for computing connected components, the following steps are repeated until no vertex v changes her parent v.p:
   - first, every vertex v stores in two auxiliary variables v.o and v.n its current father (in particular, v.o = v.n). This is done in parallel.
   - then, all edges vw are considered in parallel:
   If v.o > w.o then we set v.o.n = min{v.o.n,w.o};
   Else we set w.o.n = min{w.o.n,v.o}
   - finally, we consider every vertex v in parallel and we set:
   v.p = min{v.p,v.n,v.o.n,v.n.n}.

   Conflicts are resolved using the min rule.

   Implement Algorithm H.

4) Recall the array implementation of a queue, using two variables in order to retain the front and back elements. Modify the bitmap PBFS algorithm in order to use this array implementation for the queue. In doing so, there should be no synchronization mechanism for dequeue operations.

5) Implement UFSCC, the parallel version of Purdom-Monroe algorithm seen in class in order to compute strong connected components.