- *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 LexUP, vertices are numbered from n to 1 (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 n. 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 LexUP.
   Complexity: O(m+n)

2) Consider an undirected weighted graph. Beam search is a variation of A*, where one is given an additional parameter β. The only (but important) difference with A* is that there can be at most β vertices in the open set, at any moment during the execution of the algorithm. Whenever there are more than β vertices in OPEN, only the β vertices x with the least estimate g(s,x) + h(x) must be kept, while other vertices ate discarded.

   Implement Beam search.
   Complexity: at most O(log(n)) to find and process the next open vertex x.

3) Consider an undirected unweighted graph. Reif defined the following randomized variation of parent-connect: Initially, every vertex flips a coin independently at random. As usual, every vertex v also stores its current parent in an auxiliary variable v.o. Then all edges vw are scanned in parallel. If v.o flipped heads and w.o flipped tails, then w.o becomes the new parent of v.o. Similarly, if w.o flipped heads and v.o flipped tails, then v.o becomes the new

parent of w.o. Note that there may be several concurrent attempts to change the parent of a vertex: if so, then conflicts between processors are resolved arbitrarily.

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

Implement Algorithm Reif. No synchronization mechanism should be required.

4) Implement Hash Distributed A*, the parallel implementation of A* discussed in class.