Advanced Data Structures (ADS-MIRI): 2-Empirical Study of Binomial Heaps

The goal of this assignment is to implement binomial heaps and conduct an experimental study of their performance in combination with Dijkstra's shortest path algorithm on randomly generated graphs.

You will need first to program the graph generator and Dijkstra's algorithm. C++ is the preferred choice, but other imperative programming languages are also fine. If you choose C++ you can use the priority queues of the STL library as an implementation of binary heaps for testing your implementation.

To generate the graphs we propose you to implement the Erdös-Rényi model with parameter $p = c \ln n/n$, where n is the number of vertices of the graph (previously fixed) $c > 0$ is a constant and p determines, for each pair of vertices independently, the probability that there is an edge between them or not. We will assume for simplicity that the vertex set of the graph is $V = \{0, \ldots, n-1\}$. A possible implementation below.

```
//Random graph generator (Erdos-Renyi)
//This function adds random edges to a graph by
//deciding, with probability p, if there is an edge
//between every pair of vertices.

static void randG(Graph& G, int c) {
 int  n = G.size();
 double p = c*ln(n)*n;
 for (int i = 0; i < n; ++i) {
  for (int j = 0; j < i; ++ j) {
   double r = a random number in [0,1];
   if (r < p) G.insert(Edge(i,j));
  }
}
```

The expected number of edges in a graph of size n will be $pn(n-1)/2$. A density close to $1/n$ will produce a sparse disconnected graph (but there will be a large connected component of size $\Theta(n^{2/3})$ with high probability). For $p = \ln n/n$ the graph will be connected, with high probability, although still sparse with $\Theta(n \log n)$ edges. As $p \to 1$ the graph becomes denser, and complete in the limit case $p = 1$.

The implementation of Dijkstra's algorithm will work on weighted graphs with positive real weights (that have to be added to the graph generated above). For a given graph G and a source vertex s, the output of the procedure will be two arrays D and $path$, to return the weight of all shortest paths from s to all

other vertices, and for every vertex u one shortest path from s to u, whenever at least one path exists. In particular, (1) for all vertices u, $D[u]$ is the minimum weight in a shortest path from s to u if at least one path from s to u exists, and $D[u] = +\infty$, otherwise; (2) $path[s] = s$, $path[u]$ is the second-to-last vertex in a shortest path from s to u and $path[u] = \perp$ if no path from s to u exists (we can take $\perp = -1$ with our convention that $V = \{0, \ldots, n-1\}$.

You can use the following C++ code (actually, C++11):

```
...
typedef int vertex;
struct edge {
  vertex target;
  double weight;
};
const vertex NULL_VERTEX = -1;
typedef vector<list<edge>> weighted_graph;

void Dijkstra(const weighted_graph& G, vertex s,
              vector<double>& D, vector<vertex>& path) {
  for (vertex u = 0; u < G.size(); ++u) {
    D[u] = INFINITY; path[u] = NULL_VERTEX;
  }

  Heap cand.insert(edge(s,0));
  D[s] = 0; path[s] = s;
  while (not cand.empty()) {
    vertex u = cand.min_priority_elem();
    D[u] = cand.prio(u);
    cand.extract_min();
    for (edge e : G[u]) {
      vertex v = e.target;
      double weight_u_v = e.weight;
      if (D[u] + weight_u_v < D[v]) {
        // update D and path here
        ...
        cand.decrease_prio(v, D[v]);
      }
    }
  }
}
```

To that end you should implement a `BinomialHeap` class like this:

```
...
#include <limits>
const double INFINITY = numeric_limits<double>::max();

template <typename Elem>
class BinomialHeap {
public:
  // create an empty Binomial Heap
  BinomialHeap();

  // adds elem x in the BH with priority p;
  void insert(const Elem& x, double p);

  // returns an elem of minimum priority, error if the
  // heap is empty
  Elem min_priority_elem() const;

  // returns the priority of elem x, error if elem x is
  // not in the heap
  double prio(const Elem& x) const;

  // removes an elem of minimum priority (the one returned
```

```
  // by min_priority_elem()), error if the heap is empty
  void extract_min();

  // decrease the priority of elem x to the new value p,
  // error if x is not in the heap or prio(x) < p
  void decrease_prio(const Elem& x, double p);

  // P.meld(Q) melds binomial heap Q into P; the BH Q
  // becomes empty
  void meld(BinomialHeap& Q);

  // returns true iff the heap is empty, returns the number of
  // elems in the heap
  bool empty() const;
  int size() const;
private:
  ...
};
```

As you'll be using the class with `Elem = int`, and your elements will all belong to a range $0..n-1$, you can implement a non-generic version (simply change all `Elem`'s to `int`'s above), and simplify the implementation as you can assume that your elements are integers between 0 and $n-1$ (n is the number of vertices of the graph, and these will be your elements).

For the implementation of binomial heaps you can follow Conrado Martínez's slides for this course; check also the other references given in the slides. For the implementation and further details about Dijkstra's algorithm you can check the *Data Structures and Algorithms* slides. A search in the Internet for Dijkstra's algorithm or binomial queue will produce thousands of tutorials, notes, video lectures, ...

For the experimental study, add code in your binomial heaps to record the number of basic operations made. By basic operations we mean linking one binomial tree to another, removing the root of a binomial tree, swapping father and child in a binomial tree to (locally) reestablish the heap property, ...

Run Dijkstra's algorithm with weighted graphs of several different (large) sizes and with different *densities*. Produce a random graph of size n with density $p = c \ln n/n$, $0 \leq c \leq 2$, and a real weight uniformly drawn at random from the range $(0, MAX]$, for some suitable constant MAX, e.g., $MAX = 1000$. For every size and density generate M instances and run the algorithm of each to get averages, variance, maximum, etc.

Once the full suite of experiments has been executed and data has been gathered, you have to prepare a report.

1. Describe briefly your implementation of binomial heaps and the program to execute the experiments. Give full listings of the code as an appendix of your report.

2. Describe briefly the experimental setup, how many different combinations of the parameters n and p have you studied, how many runs M have you performed for a particular n and p.

3. Provide tables and plots sumarizing the results of the experiments. In particular, you should give plots showing how the average and maximum

number of basic operations evolve with n, and how they vary with p (or m, the number of edges in the graph). Avoid 3-D plots. It can be useful to "normalise" the plots by dividing by the costs by the size of the graph.

4. Compare the experimental results with the theoretical predictions, in particular, the cost of n binomial heap operations of which m of them are `decrease`'s is $O(m \log n)$.

   Plots combining the theoretical values and the experimental results are useful, but it is also important to quantify the difference between the theoretically predicted values and the empirical values.

5. Write down your conclusions.

An extra bonus (not mandatory) is to conduct a similar empirical study of the actual execution times following similar steps as described above. Do not forget to remove the code to count operations from the heap implementation, since it might introduce small disturbances in the measurements.

We encourage you to use LaTeX to prepare your report. For the plots you can use any of the multiple packages that LaTeX has (in particular, the bundle TikZ+PGF) or use independent software such as gnuplot and then include the images/PDF plots thus generated into your document.

Submit your work using the FIB-Racó. It must consist of a zip or tar file containing all your source code, auxiliary files and your report in PDF format. Include a README file that briefly describes the contents of the zip/tar file and gives instructions on how to produce an executable program and reproduce the experiments. The PDF file with your report must be called

`YourLastName_YourFirstName-2-binomial_heaps.pdf`,

and the zip/tar file

`YourLastName_YourFirstName-2-binomial_heaps.zip`

(or `.tar`).