

In the lab-classes this week we experiment with *disjoint-sets data structures*.

Update the environment: In your existing directory CS-242-Algorithms:

```
git checkout master
git pull git://github.com/OKullmann/CS-242-Algorithms.git master
```

Or if this directory (repository) doesn't exist (anymore), resp. you want to create a new repository, create a new clone by

```
git clone git://github.com/OKullmann/CS-242-Algorithms.git
```

Using the second form, in the following weeks by `git pull` you will pull from OKullmann's Github repository. Now change to the subdirectory for week 5:

```
cd CS-242-Algorithms/200910/Week05/
```

Basic setup: Today (as well as in the following two weeks) no files are to be changed, but we

1. read the code
2. run the code on small examples, to understand the algorithms
3. perform some experiments.

Compilation is done by

```
Week05> make
```

The executables produced are

```
LinkedLists
ConnectedComponents
ConnectedComponentsE
```

First with `LinkedLists` hands-on experience with small examples is to be gained. Then via `ConnectedComponents` the connected components of random graphs are computed, using the six different variations on data structures from the module. Your task is to investigate the structure of the random graphs and the run-time of the various algorithms.

Comments on the code The files are as follows:

1. `LinkedLists.hpp` contains two classes, `LinkedLists` for the basic linked-lists data structure, and `LinkedListsH` for the improved data structure using the size-heuristics for the union.
2. `LinkedLists.cpp` allows you to create and manipulate these data structures, and to inspect them.
3. `Framework.hpp` contains some generic framework used in `LinkedLists.cpp`.
4. `RootedTrees.hpp` contains the classes `RootedTrees`, `RootedTreesHS`, `RootedTreesHP`, `RootedTreesHSP`, implemented the disjoint-forests data structure with the heuristics using the size of the trees resp. path-compression.

5. `RandomGraphs.hpp` contains the class `RandomGraph` for creating random graphs.
6. `ConnectedComponents.hpp` contains the class `Number_connected_components` for computing the number of connected components of a graph (a slight extension of the algorithm presented in the first lecture of this week).
7. Finally `ConnectedComponents.cpp` allows you to create random graphs and to run the connected-components algorithm on it for the six implementations of the disjoint-sets data structure.
8. Additionally, `RandomGraphs.hpp` contains the class `RandomGraphE` with a different model of random graphs, which allows faster generation (so we can run larger experiments). This is used in `ConnectedComponentsE.cpp`, which now only runs the four fast implementations.

More precisely, in most cases we actually don't have classes, but *class templates*, which take for example the element-type of the sets (what is actually stored in the sets) as a *type parameter*. Do not try to understand the details, however the general flow of the code should be easy to grasp.

Understanding the implementation via linked-lists `LinkedLists` allows you to enter the number N of different elements as a parameter; it generates then the singleton sets $\{1\}, \dots, \{N\}$, shows the two data structures (without and with the size-heuristics), and then waits for pairs of element-indices to be entered, where it performs union of the corresponding set-elements belong to different sets. For example:

```
Week05> ./LinkedLists 5
```

```
Simple:
```

```
0: 0x804c078-> 1 0x804c078 0 0x804c078
1: 0x804c090-> 2 0x804c090 0 0x804c090
2: 0x804c0a8-> 3 0x804c0a8 0 0x804c0a8
3: 0x804c0c0-> 4 0x804c0c0 0 0x804c0c0
4: 0x804c0d8-> 5 0x804c0d8 0 0x804c0d8
```

```
With size heuristics:
```

```
0: 0x804c108-> 1 0x804c108 0 0x804c108 1
1: 0x804c120-> 2 0x804c120 0 0x804c120 1
2: 0x804c138-> 3 0x804c138 0 0x804c138 1
3: 0x804c150-> 4 0x804c150 0 0x804c150 1
4: 0x804c168-> 5 0x804c168 0 0x804c168 1
```

```
0 1
```

```
Simple:
```

```
Cell for 0:
```

```
0x804c078-> 1 0x804c078 0 0x804c078
```

```
Cell for 1:
```

```
0x804c090-> 2 0x804c090 0 0x804c090
```

```
Both sets are different, thus union is performed.
```

```
Now the whole data structure is as follows:
```

```
0: 0x804c078-> 1 0x804c090 0 0x804c078
1: 0x804c090-> 2 0x804c090 0x804c078 0x804c078
2: 0x804c0a8-> 3 0x804c0a8 0 0x804c0a8
3: 0x804c0c0-> 4 0x804c0c0 0 0x804c0c0
4: 0x804c0d8-> 5 0x804c0d8 0 0x804c0d8
```

With size heuristics:

Cell for 0:

```
0x804c108-> 1 0x804c108 0 0x804c108 1
```

Cell for 1:

```
0x804c120-> 2 0x804c120 0 0x804c120 1
```

Both sets are different, thus union is performed.

Now the whole data structure is as follows:

```
0: 0x804c108-> 1 0x804c120 0 0x804c108 1
```

```
1: 0x804c120-> 2 0x804c120 0x804c108 0x804c108 2
```

```
2: 0x804c138-> 3 0x804c138 0 0x804c138 1
```

```
3: 0x804c150-> 4 0x804c150 0 0x804c150 1
```

```
4: 0x804c168-> 5 0x804c168 0 0x804c168 1
```

Each (non-text) output line shows a *cell*, with its address, the element it contains, the `rep`-pointer, the `next`-pointer and the `last`-pointer. When all sets are output, then each line is prefixed with the *index* of that node — via this index (instead of the unhandy pointer-values) you can then ask for a union.

When entering two indices (above “0 1”), first `FIND-SET` is performed, and the (representative) cells found are shown; in case the representative cells are different (that is, the represented sets are different) `UNION` is performed, and the new state of the (whole) data structure is shown.

Tasks (**answers to be shown to the postgrads**):

1. Create your own example runs, draw the data structures on paper, and precisely match it with the outputs.
2. Reproduce the worst-case example from the lecture.
3. How to change the inputs, so that the simple data structure performs as good as the improved one?

Random graphs Via

```
> ./ConnectedComponents N p
```

a random graph with N vertices is created, where each possible edge is present with probability p . So for $p = 0$ there is no edge, while for $p = 1$ all edges are present.

Your **first task** here is to predict the number of edges, e.g. for

```
> ./ConnectedComponents 10 0.5
```

```
Creating the graph with 10 vertices: 20 edges created in 0.00s.
```

— can you predict the “20”? (it could be also somewhat different — it’s (pseudo-)random). For this you need to find out how many edges could there be in total (every vertex could be connected with every other vertex). Use `R` as a calculator; you might also use the function `choose(n, k)` there, which computes a binomial coefficient.

Your **second task** here is then to find out, for various (not too small) N , to get everything connected. The number of connected components found is also output, together with the running times. So the question is how to get just one connected component, with not too many edges (of course, setting $p = 1$ it is guaranteed that everything is connected — however far fewer edges suffice). Experiment a bit. *Hint*: A phase transition happens when you have about half as many edges as vertices — there the shape of the random graph drastically changes, from many tiny connected components to a few big components.

Ranking the six implementations Finally, you need to look at the running times of the six implementations, experimenting with various N and p . A somewhat longer computation would be

```
> ./ConnectedComponents 60000 0.0005
Creating the graph with 60000 vertices: 901505 edges created in 38.93s.
  LinkedLists:      1; 155.560s
  LinkedListsH:    1;  0.040s
  RootedTrees:    1; 518.640s
  RootedTreesHS:  1;  0.030s
  RootedTreesHP:  1;  0.050s
  RootedTreesHSP: 1;  0.030s
```

Especially look at the two simplest case, `LinkedLists` and `RootedTrees`: Above one sees that `LinkedLists` is faster, however for different p this is reversed — **when and why?!**

Creating larger graphs Since for the creation of random graphs with a given edge-probability p one needs to run through all potential edges, creation of large graphs takes very long. Now via

```
> ./ConnectedComponentsE N K
```

a random graph with N vertices and K edges is created, where all edges are equally likely. This is much faster, and larger experiments can be run:

```
> ./ConnectedComponentsE 10000000 10000000
Creating the graph with 10000000 vertices and 10000000 edges: 0.55s.
  LinkedListsH: 1619854; 6.970s
  RootedTreesHS: 1619854; 4.480s
  RootedTreesHP: 1619854; 5.570s
  RootedTreesHSP: 1619854; 4.450s
```

```
> ./ConnectedComponentsE 15000000 15000000
Creating the graph with 15000000 vertices and 15000000 edges: 0.84s.
  LinkedListsH: 2428010; 13.420s
  RootedTreesHS: 2428010; 7.390s
  RootedTreesHP: 2428010; 9.650s
  RootedTreesHSP: 2428010; 7.060s
```

So you can experimentally evaluate (for this model!) which of the four fast implementations is fastest (under what circumstances).