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

**Update the environment:** Update the repository via

```
git pull
```
or, if you didn’t create it before, create a (new) clone by

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

Change to the subdirectory for week 6:

```
cd ~/CS-242-Algorithms/201112/Week06/
```

**Basic setup:** We

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

Compilation is done by

```
Week06> 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.

**Understanding the implementation via linked-lists** *LinkedLists* allows you to enter the number $N$ of different elements as a parameter:

1. it generates then the singleton sets $\{1\}, \ldots, \{N\}$,
2. shows the two data structures, without and with the size-heuristics,
3. and then waits for pairs of cell-indices to be entered (one pair after another),
4. where it then performs union of the corresponding sets (when the two sets are different, that is, disjoint).
5. If you ask for the union of elements (more precisely, cells) already belonging to the same set, then just information on the cells of these elements is shown.
6. The elements are $1, \ldots, N$. For each element, we have initially one cell, given by it address, containing this element.
7. To make input easier, instead of the address of a cell the cell-index is used. Element $i$ is initially in a cell which has index $i - 1$.

For example (showing just one cycle):
Week06> ./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
Cell for 1:
0x804c120-> 2 0x804c120 0 0x804c120
Both sets are different, thus union is performed.
Now the whole data structure is as follows:
0: 0x804c108-> 1 0x804c120 0 0x804c108
1: 0x804c120-> 2 0x804c120 0x804c108 0x804c108
2: 0x804c138-> 3 0x804c138 0 0x804c138
3: 0x804c150-> 4 0x804c150 0 0x804c150
4: 0x804c168-> 5 0x804c168 0 0x804c168

Each (non-text) output line shows a cell:

1. first its address,
2. then the element it contains,
3. then the rep-pointer, the next-pointer, and finally the last-pointer.
4. In case of the size heuristics, an additional entry shows the size of the list, in case that node is the head (representative element) of the set (otherwise the information is old, and no longer of use).
For the list of all cells, 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 match it with the outputs.
2. Reproduce the worst-case example from the lecture.
3. How to change the order of the inputs (the union-instructions), so that the simple data structure performs as good as the improved one (for the worst-case examples)?

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”?

1. Note that deviation from the predicated theoretical values are natural — the result is (pseudo-)random, while your task is to predict the average number of edges.
2. First you need to find out how many edges could there be in total (every vertex could be connected with every other vertex).
3. This should correspond to what you get with p = 1.
4. You might also consider the case p = 0. After that you need then to tackle the general case.
5. Work out a formula on paper (using the formulas on graphs from the script).
6. For the numerical computation, use R as a calculator.

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 (taking more than 10 minutes, due to the two slow performers) 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
```

**Task:** 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 of creating inputs!) which of the four fast implementations is fastest (under what circumstances).