Lab report: Datastructure enhancements in DNA

Nico Haase
Technische Universität Darmstadt
nico.haase@stud.tu-darmstadt.de

I. Motivation

Former DNA version had data structures hard coded into Java classes. Classes like `DirectedNode` were extended to store internal structures in specific data structures, such that `DirectedNodeAl` stored its edges in an arraylist. For each internal data structure, a dedicated class needs to be written. Classes containing the whole graph were named after their internal data structures, eg. `DirectedGraphAlAl`.

- Adding a data structure is much work, as in addition to the data structure itself many classes for its usage need to be created
- Combining data structures leads to chaos, as for each combination of data structures, a specific graph class needs to be created

II. Ideas for enhancements

- Decouple data structure providers (classes that provide a common interface for our needs and use specific data structures, but hide this from the outside) and their clients (classes for the whole graph, for nodes, and for edges)
- Implement more data structure providers which have not yet been present, but could be interesting for different reasons (faster run time, less memory consumption,...)
- Create a recommender system which analyses a whole workflow consisting of graph generation, batch generation and application, metric application,... over multiple runs of a series, computes the complexity of the accesses that happened, and gives recommendations about better data structures to use for this collection of accesses

III. Data structures

- Data structure classes to hold single properties (nodes and edges) were decoupled from the graph and node classes by using a common interface for the "storage" classes. At each possible place, this interface is used instead of concrete classes
- The common interface `IDataStructure` holds methods for all actions that are necessary for common accesses, like `add`, `contains`, `remove`, `size`,..., and internally uses another data structures (eg. `DArray` stores everything in an array, `DHashSet` in a hashset,...)
- Extended interfaces hold methods that are not available on all data structures, like `getRandom`. A data structure that makes use of this constraint in the base interface is a Bloom filter which probabilistically stores only flags for contained elements, but not a pointer to the elements itself, for less memory consumption
- The class `GraphDataStructure` holds the currently used storage classes for the global edge and node lists, and the node-local edge lists.
- Generating a new instance of a storage class is done through this class, using Reflection such that the used storage class can be exchanged on runtime
- Currently included data structures: pure array, `ArrayList`, `HashMap`, `HashSet`, `LinkedList`

This concept allows us to add more data structure classes and combine and exchange them very easily, without having a cluttered namespace

IV. Profiling

- Question: different data structures have different complexities. Adding something might be cheap on one, but extensive on another that is faster on removing them. But how can we compare them based on the accesses used?
- Idea: count each call to the data structures, compute the complexities at the end and give recommendations
- Achieved through AspectJ which can be used to add functionalities to existing code (see section A). This enables us to keep nearly all profiler functionality seperated from the other code: the aspects to enable counting reside in `ProfilerAspects.aj`, the counterpart to count the accesses reside in the class `Profiler`
- On runtime, the profiler counts calls based on the performed action (add, remove, contains) and callee signature (metric name, update type,...), such that we can determine the complexities for different use cases with one execution
- Combined complexities are computed based on the complexities for each action on each data structure, which are stored within each data structure class

- Accesses are counted based on a list of distinguishable access types (add, remove, contains) and the callee signature (eg. metric name, update type). For the output, these are concatenated such that each line contains the signature and the access type
- The output ends with the aggregated complexity that occured in the current run and a recommendation of three data structure combinations. As the computation of recommendations is time consuming, this is not done for each callee, but only for aggregations.
- Example output (shortened, full output at B):

```
DegreeDistributionU.AddNodeGlobal=0
DegreeDistributionU.AddNodeLocal=0
[...]
DegreeDistributionU.SizeEdgeLocal=96000
DegreeDistributionU.RandomNodeGlobal=0
DegreeDistributionU.RandomEdgeGlobal=0
[...]
#  Aggr: 96000*O(1)
#   Recommendations:
#    DArray;DArray;DArray: 96000*O(1)
```

- Access counters are written to files split up into different use cases, summarized for each callee, and a recommendation for optimized complexities. These files are stored in the following locations:

```
Root directory of data output
├─ aggr
├─ run.0
│  ├─ batch.0
│  │  ├─ metric.firstExample
│  │  │  └─ ___metric.profiler.values ..... Output for a single metric
│  │  ├─ metric.secondExample
│  │  │  └─ ...
│  │  ├─ ___aggregated.profiler.values ..... Aggregation over the whole batch
│  │  ├─ ___batchGeneration.profiler.values ..... Output for the batch generation
│  │  ├─ ___metric.profiler.values ..... Aggregation over all metrics in this batch
│  │  └─ ___updates.profiler.values ..... Output for the batch application
│  ├─ batch.1
│  │  └─ ...
│  ├─ ___aggregated.profiler.values ..... Aggregation over the whole run
│  ├─ ___batchGeneration.profiler.values ..... Aggregation of all batch generations in this run
│  ├─ ___graphGeneration.profiler.values ..... Output for the initial graph generation
│  ├─ ___metric.profiler.values ..... Aggregation over all metrics in this run
│  └─ ___updates.profiler.values ..... Aggregation of all batch applications in this run
├─ run.1
│  └─ ...
├─ ___aggregated.profiler.values ..... Aggregation over the whole series
├─ ___batchGeneration.profiler.values ..... Aggregation of all batch generations in this series
├─ ___metric.profiler.values ..... Aggregation over all metrics in this series
└─ ___updates.profiler.values ..... Aggregation of all batch applications in this series
```

- Important side note about the recommender: different data structures use different ways to perform the same action (eg. `DArray.add` calls `DArray.contains`, while `DHashSet.add` does not). The profiler can only count each access type for the currently used data structures and give recommendations for this combination of calls. Thus, using another combination might not lead to the complexity the recommender computed. For the given example, the recommender will work based on calls for both `add` and `contains`, although `DHashSet` never calls `contains` within `add`. It might be necessary to run the recommender more than once to yield better results.
- It takes a nearly static amount of time to compute the recommendations, independent of the chosen data structures and the graph size. Having it active in a setup with a small graph predominates the optimization of data structures
- The granularity of the recommender can be set in three levels through the DNA configuration:
  - `PROFILER_DISABLE_ALL_RECOMMENDATIONS = true` completely disables the recommender
  - `PROFILER_DISABLE_ALL_RECOMMENDATIONS = false` and `PROFILER_WRITE_ALL_RECOMMENDATIONS = false` (default) outputs recommendations only for aggregations, but not for each individual access type
  - `PROFILER_DISABLE_ALL_RECOMMENDATIONS = false` and `PROFILER_WRITE_ALL_RECOMMENDATIONS`

= `true` outputs recommendations for each individual access type

V. Building DNA

- For building DNA in Eclipse, one should install the "AspectJ Development Tools" over the Eclipse marketplace. The DNA project in Eclipse needs to be converted to an AspectJ project (right click on the project, select Configure → Convert to AspectJ project). Now the profiler can be told to count accesses through calling `Profiler.activate();`
- Remark: while the project is configured with AspectJ, the additional code is always woven in. Not activating the profiler saves some runtime (as computing the recommendations is the most time consuming part of the profiler), but there is a runtime difference between having AspectJ completely disabled and having it enabled, but not activating the profiler
- Building the project without Eclipse requires the AspectJ compiler available at http://eclipse.org/aspectj/ and some modifications for the proper locations of AspectJ and JUnit in `build.xml`. Afterwards, on can compile the code and create JARs using the specified build targets. See C for an example file

VI. Performance evaluation of the profiler

The reworked class structure leads to a more readable and intuitive way to use DNA for the programmer, but at which cost? The data structures are constructed using reflection, which causes some slowdown[1], and having the profiler enabled also causes additional work.

- Runtime comparisons for the profiler part strongly rely on the monitored use case
- Data structure combinations:
  1) Initial data structure combination: `DArrayList` / `DArrayList` / `DArrayList`
  2) Recommended combination: `DArray` / `DHashSet` / `DHashSet`
  3) Worst case from the recommendation list: `DLinkedList` / `DArray` / `DArray`
  4) As a comparison: `DArray` / `DArray` / `DArray`
- Common parts for the analysis: random undirected weighted graph with 1 000 nodes, 10 000 edges
- Common batch generation, if not stated otherwise: random batch generation with 1 000 node additions, 500 node removals, 50 changes of node weights, 2 000 edge additions, 500 edge removals, 50 changes of edge weights
- Runtimes are averaged over five series generations, running 3 runs with 4 batches each
- Sources are available via https://github.com/NicoHaase/
- Test setup: Windows 7 on a Dell laptop, Intel i5 dual core CPU with 2.7 Ghz, Java 1.7.0_45

For much larger graphs, the slowdown of having the profiler enabled falls dramatically to less than 10%

[1] http://stackoverflow.com/questions/435553/java-reflection-performance reports about a slowdown in the magnitude of factor 10

Task	using usual initialization	using reflection
500 000 `ArrayLists`, putting one string into them	$\sim$ 125 msec	$\sim$ 650 msec
500 000 `HashSets`, putting one string into them	$\sim$ 310 msec	$\sim$ 800 msec

Task performed on data structure set 1	AspectJ disabled	AspectJ enabled, aspects commented out	AspectJ with aspects, profiler disabled	Profiler enabled
Metric `UndirectedClusteringCoefficientU`	$\sim$ 5190 msec	$\sim$ 5205 msec (-0.3%)	$\sim$ 5500 msec ($+6\%$)	$\sim$ 6030 msec ($+16\%$)
Metric `DegreeDistributionR`	$\sim$ 4065 msec	$\sim$ 4055 msec (-0.3%)	$\sim$ 4195 msec ($+3\%$)	$\sim$ 4650 msec ($+14\%$)
Metric `UndirectedShortestPathsR`	$\sim$ 8120 msec	$\sim$ 8045 msec (-1%)	$\sim$ 9255 msec ($+14\%$)	$\sim$ 10445 msec ($+29\%$)

Task performed on data structure set 2 (Recommender)	AspectJ disabled	AspectJ enabled, aspects commented out	AspectJ with aspects, profiler disabled	Profiler enabled
Metric `UndirectedClusteringCoefficientU`	$\sim$ 2885 msec	$\sim$ 2900 msec ($+0.5\%$)	$\sim$ 3140 msec ($+9\%$)	$\sim$ 3615 msec ($+25\%$)
Metric `DegreeDistributionR`	$\sim$ 1455 msec	$\sim$ 1475 msec ($+1.4\%$)	$\sim$ 1565 msec ($+8\%$)	$\sim$ 2060 msec ($+42\%$)
Metric `UndirectedShortestPathsR`	$\sim$ 8790 msec	$\sim$ 8750 msec (-0.5%)	$\sim$ 10030 msec ($+14\%$)	$\sim$ 11400 msec ($+30\%$)

Task performed on data structure set 3	AspectJ disabled	AspectJ enabled, aspects commented out	AspectJ with aspects, profiler disabled	Profiler enabled
Metric `UndirectedClusteringCoefficientU`	$\sim$ 7900 msec	$\sim$ 8100 msec ($+2.5\%$)	$\sim$ 8640 msec ($+9\%$)	$\sim$ 9565 msec ($+21\%$)
Metric `DegreeDistributionR`	$\sim$ 6450 msec	$\sim$ 6760 msec ($+5\%$)	$\sim$ 7060 msec ($+9.5\%$)	$\sim$ 7980 msec ($+24\%$)
Metric `UndirectedShortestPathsR`	$\sim$ 10820 msec	$\sim$ 11325 msec ($+4.7\%$)	$\sim$ 12520 msec ($+16\%$)	$\sim$ 14150 msec ($+31\%$)

Task performed on data structure set 4	AspectJ disabled	AspectJ enabled, aspects commented out	AspectJ with aspects, profiler disabled	Profiler enabled
Metric `UndirectedClusteringCoefficientU`	$\sim$ 6490 msec	$\sim$ 6530 msec ($+0.6\%$)	$\sim$ 6800 msec ($+5\%$)	$\sim$ 7300 msec ($+12.5\%$)
Metric `DegreeDistributionR`	$\sim$ 5130 msec	$\sim$ 5180 msec ($+1\%$)	$\sim$ 5380 msec ($+5\%$)	$\sim$ 5900 msec ($+15\%$)
Metric `UndirectedShortestPathsR`	$\sim$ 9530 msec	$\sim$ 9600 msec ($+0.7\%$)	$\sim$ 10950 msec ($+15\%$)	$\sim$ 12080 msec ($+27\%$)

VII. Evaluation / demonstration

The profiler should give one the ability to run DNA with an optimized combination of data structures. In this section, we want to demonstrate the capabilites of the recommender. After having already evaluated some very simple cases in section VI, we additionally chose two more scenarios for this to evaluate the recommender. We initially select data structures for the global node and edge list and the node-local edge list as bad as possible to demonstrate the later enhancements.

In the first scenario, the graph should grow from batch to batch by 1000 nodes and 2000 edges without removing anything. Before each graph update, the degree distribution is refreshed. We store all lists in arrays as it is expensive to add entries to an array: the first step is checking whether the element is already contained in that array by iterating over the whole array (linear complexity - other data structures can do this with static complexity), the second step is adding the element. The calculated complexity for one example run with this data structure combination is $549770 * O(1) + 324688 * O(d) + 295205 * O(E)$, the runtime can be averaged to ~ 9000 msec. Using arrays to store the global node list and HashSets for both edge lists, the complexity can be estimated to $1007319 * O(1)$. Using the recommended combination leads to a complexity of $844477 * O(1)$ and an averaged runtime of ~ 1745 msec (-80%).

In the second scenario, it should shrink from a graph with 2 000 nodes and 3 000 edges by 100 nodes and 150 edges per batch, computing shortest paths alongside. Using the initial data structure combination, the complexity of an example run is $74937234 * O(1) + 36012 * O(d) + 24012 * O(N) + 9006 * O(E)$, the averaged runtime is ~ 10690 msec. The recommended data structures (here: `DArray` / `DHashMap` / `DHashSet`) lead to an estimated complexity of $75075849 * O(1) + 1863 * O(E)$. Using this combination leads to a complexity of $75231610 * O(1) + 1858 * O(E)$ for an example run and an averaged runtime of ~ 14540 msec. We also see this correlation in evaluations with larger and smaller graphs. Other recommended combinations did neither perform noticeable better than the initial combination.

VIII. Conclusion & further work

We have introduced more data structures and a profiling and recommendation extension for DNA. Users of DNA can combine the data structures under many aspects like time or memory consumption. But the runtime comparisons we evaluated show a problem of our current recommendation system: it is only based on complexities. This can give a fast first view about the scalability of the data structure performances, but the comparison between different combinations of data structures leads to misinterpretations. For a brief example: both the LinkedList and the HashSet have a complexity of $O(1)$ for inserting elements. But obviously, inserting an element into a hash based data structure takes longer as we have to compute the hash of the element first to find the corresponding bucket. In a future work, we want to evaluate additional ways to compare the performance of data structures.

Appendix

A. AspectJ

- AspectJ can be used to add functionalities using completely seperate files that are woven into existing code on compile time
- Defined through point cuts that hook into the existing code and aspects that extend the code

```
pointcut methodCall() :
  execution(* Class.Method())

before() : methodCall() {
  System.out.println("Calling Method");
}
```

- On compile time, the aspects are woven into the byte code transparently

B. Example output

```
DegreeDistributionU.AddNodeGlobal=0
DegreeDistributionU.AddNodeLocal=0
DegreeDistributionU.AddEdgeGlobal=0
DegreeDistributionU.AddEdgeLocal=0
DegreeDistributionU.RemoveNodeGlobal=0
DegreeDistributionU.RemoveNodeLocal=0
DegreeDistributionU.RemoveEdgeGlobal=0
DegreeDistributionU.RemoveEdgeLocal=0
DegreeDistributionU.ContainsNodeGlobal=0
DegreeDistributionU.ContainsNodeLocal=0
DegreeDistributionU.ContainsEdgeGlobal=0
```

```
DegreeDistributionU.ContainsEdgeLocal=0
DegreeDistributionU.GetNodeGlobal=0
DegreeDistributionU.GetNodeLocal=0
DegreeDistributionU.GetEdgeGlobal=0
DegreeDistributionU.GetEdgeLocal=0
DegreeDistributionU.SizeNodeGlobal=0
DegreeDistributionU.SizeNodeLocal=0
DegreeDistributionU.SizeEdgeGlobal=0
DegreeDistributionU.SizeEdgeLocal=96000
DegreeDistributionU.RandomNodeGlobal=0
DegreeDistributionU.RandomEdgeGlobal=0
DegreeDistributionU.IteratorNodeGlobal=0
DegreeDistributionU.IteratorNodeLocal=0
DegreeDistributionU.IteratorEdgeGlobal=0
DegreeDistributionU.IteratorEdgeLocal=0
#  Aggr: 96000*O(1)
#   Recommendations:
#    DArray;DArray;DArray: 96000*O(1)
```

C. Build file for ANT

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<project default="jarJava" name="DNA">

   <!-- The following two properties need to be changed -->
   <property name="jUnitJar" location="./junit.jar" />
   <property name="aspectJDir" location="../aspectj-1.7.4" />

   <property name="src" location="src" />
   <property name="bin" location="bin" />

   <property name="mainClass" value="DNA.test" />

   <path id="class.path">
      <fileset dir="lib">
         <include name="**/*.jar" />
      </fileset>
      <pathelement location="${jUnitJar}" />
   </path>

   <taskdef resource="org/aspectj/tools/ant/taskdefs/aspectjTaskdefs.properties">
      <classpath>
         <pathelement location="${aspectJDir}/lib/aspectjtools.jar" />
      </classpath>
   </taskdef>

   <target name="init">
      <mkdir dir="${bin}" />
   </target>

   <target name="clean">
      <input message="Delete everything from ${bin} (neccessary if you switch from
          AspectJ to Java or the other way around)" addproperty="confirmDeleteBin"
          validargs="y,n" defaultvalue="n" />

      <condition property="executeDeleteBin">
         <and>
```

```
            <isset property="confirmDeleteBin" />
            <equals arg1="${confirmDeleteBin}" arg2="y" />
         </and>
      </condition>

      <antcall target="deleteBin" />
   </target>

   <target name="deleteBin" if="executeDeleteBin">
      <echo>Deleting ${bin}</echo>
      <delete dir="${bin}" />
   </target>

   <target name="compileJava" depends="clean, init">
      <javac srcdir="${src}" destdir="${bin}" classpathref="class.path" />
   </target>

   <target name="compileAspectJ" depends="clean, init">
      <iajc source="1.7" target="1.7" sourceroots="${src}" destDir="${bin}">
         <classpath>
            <path refid="class.path" />
            <pathelement location="${aspectJDir}/lib/aspectjrt.jar" />
         </classpath>
      </iajc>
   </target>

   <target name="jarAspectJ" depends="compileAspectJ">
      <jar destfile="DNA-aspectJ.jar" filesetmanifest="mergewithoutmain">
         <manifest>
            <attribute name="Main-Class" value="${mainClass}" />
            <attribute name="Class-Path" value="./" />
         </manifest>
         <fileset dir="${bin}" />
      </jar>
   </target>

   <target name="jarJava" depends="compileJava">
      <jar destfile="DNA.jar" filesetmanifest="mergewithoutmain">
         <manifest>
            <attribute name="Main-Class" value="${mainClass}" />
            <attribute name="Class-Path" value="./" />
         </manifest>
         <fileset dir="${bin}" />
      </jar>
   </target>
</project>
```