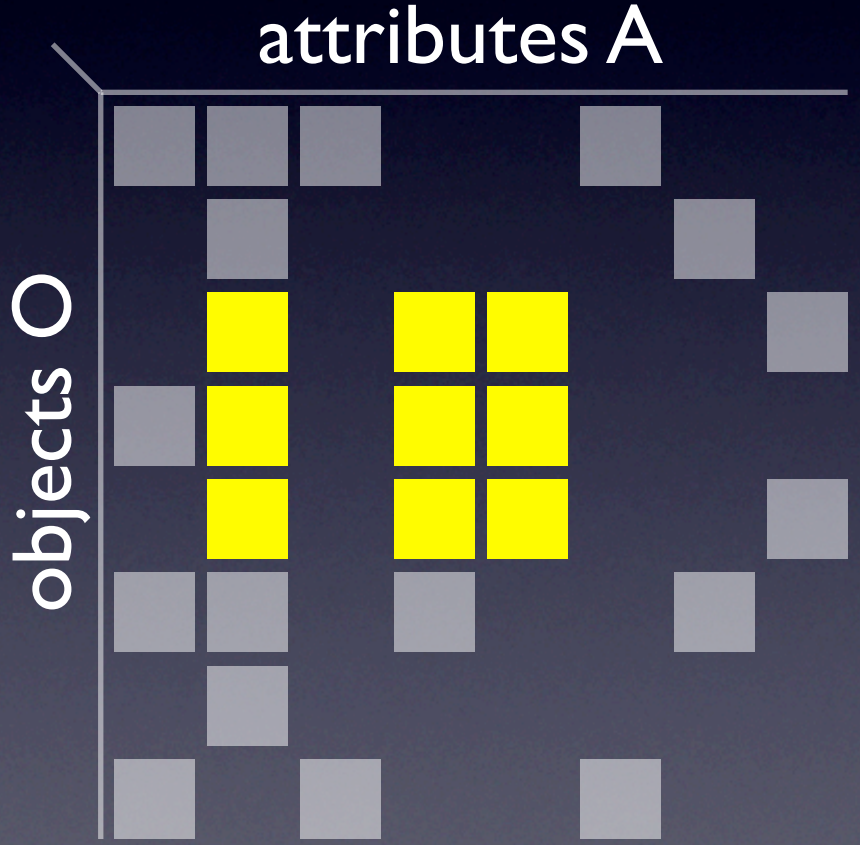
Formal Concept Analysis in Java

Daniel Götzmann

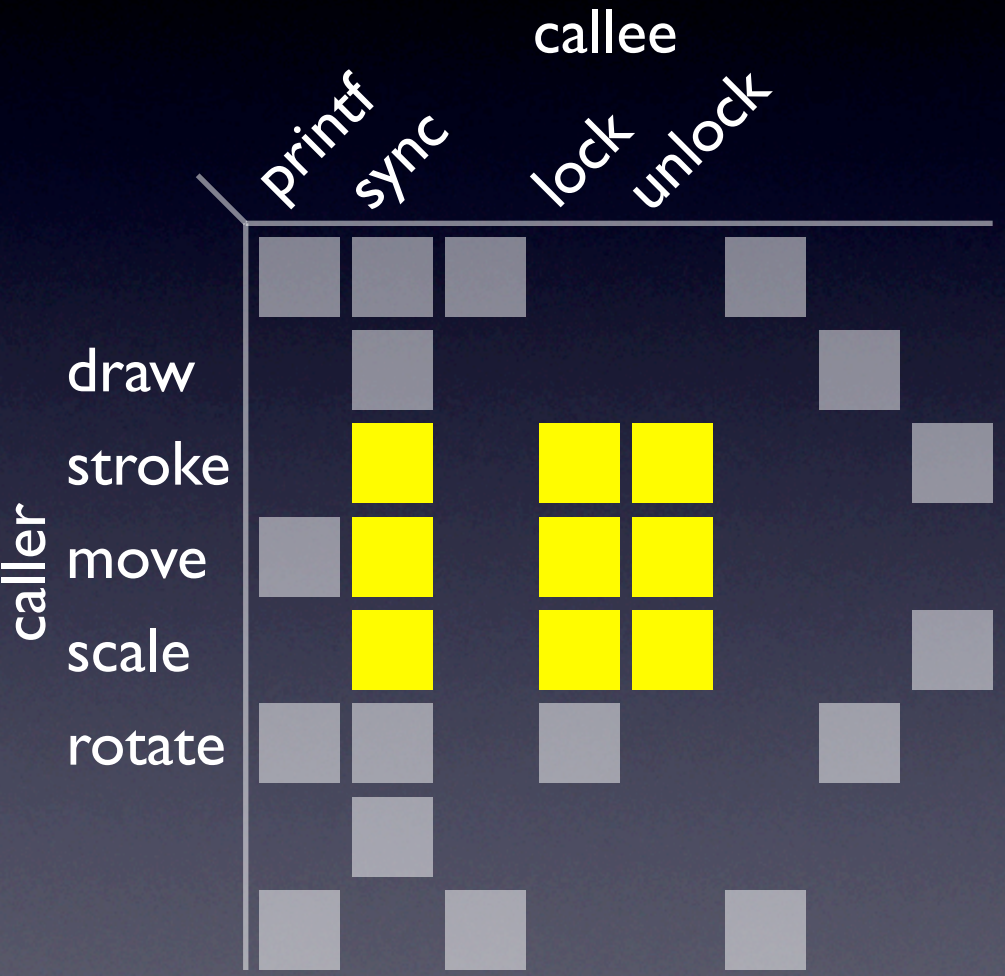
Concept Analysis



context $R \subseteq O \times A$

want to compute concepts

Concept Analysis

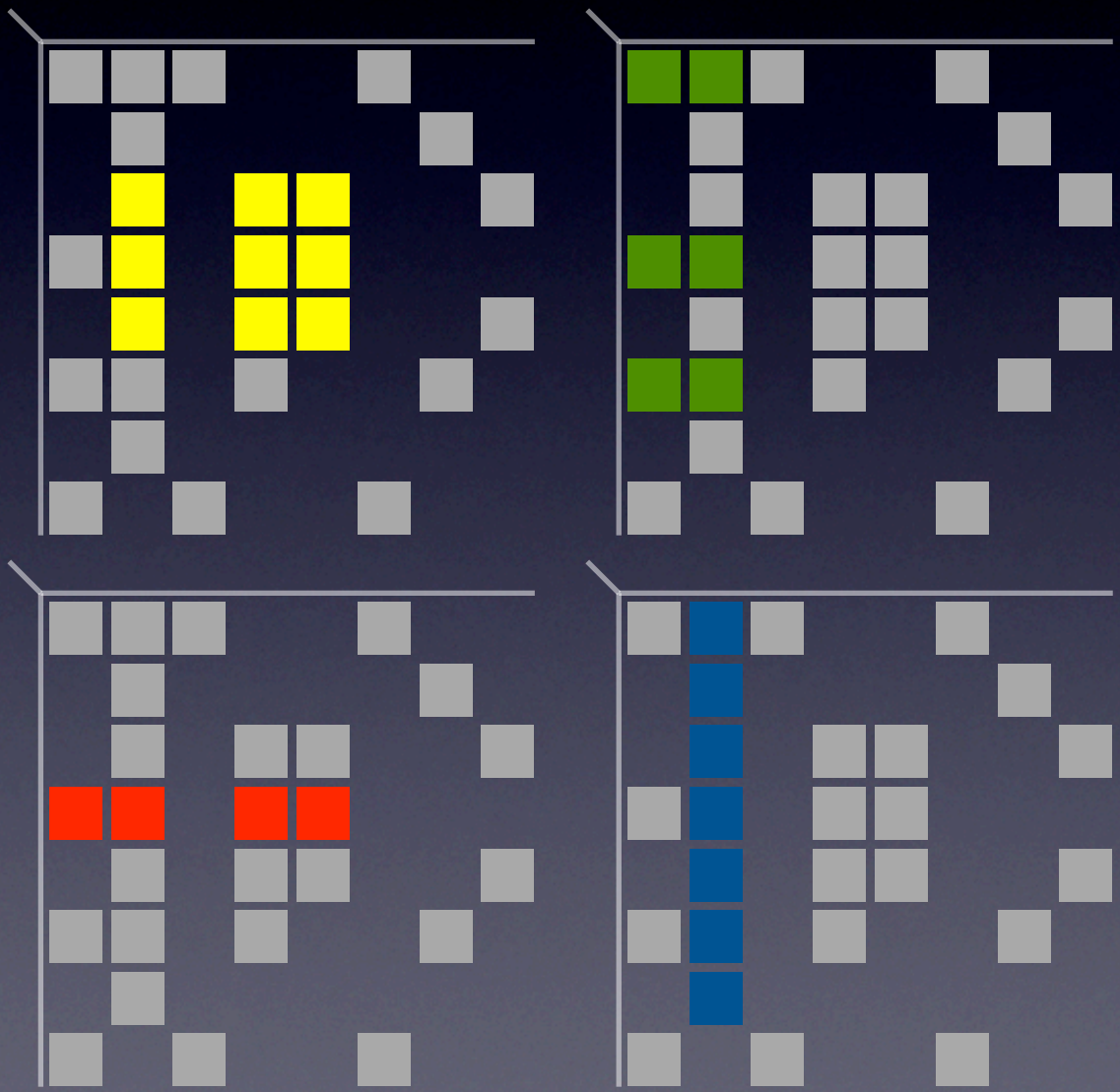


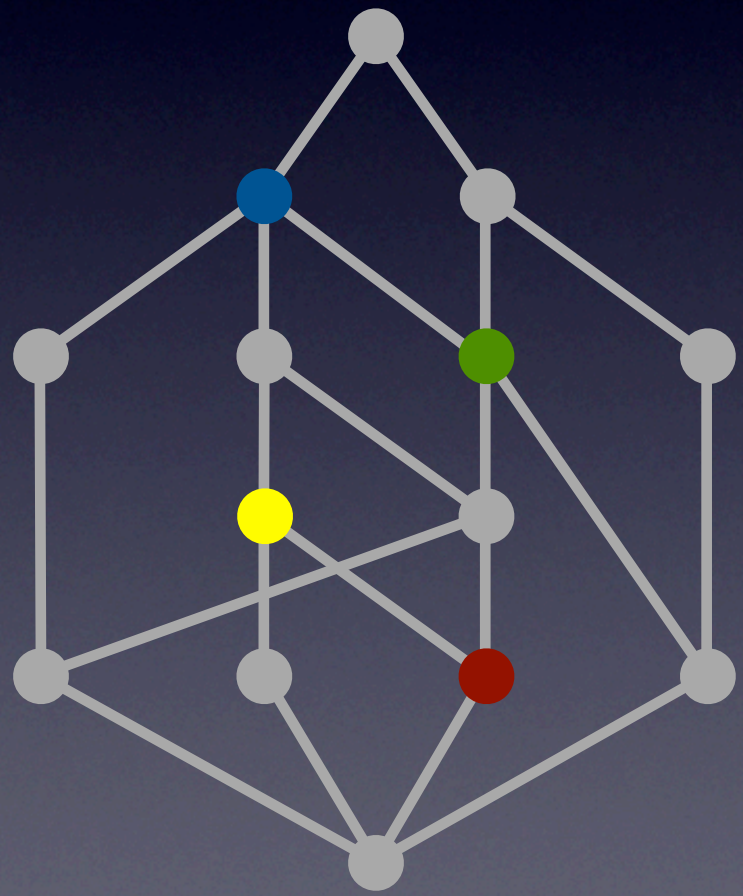
context R ⊆ O x A

want to compute concepts

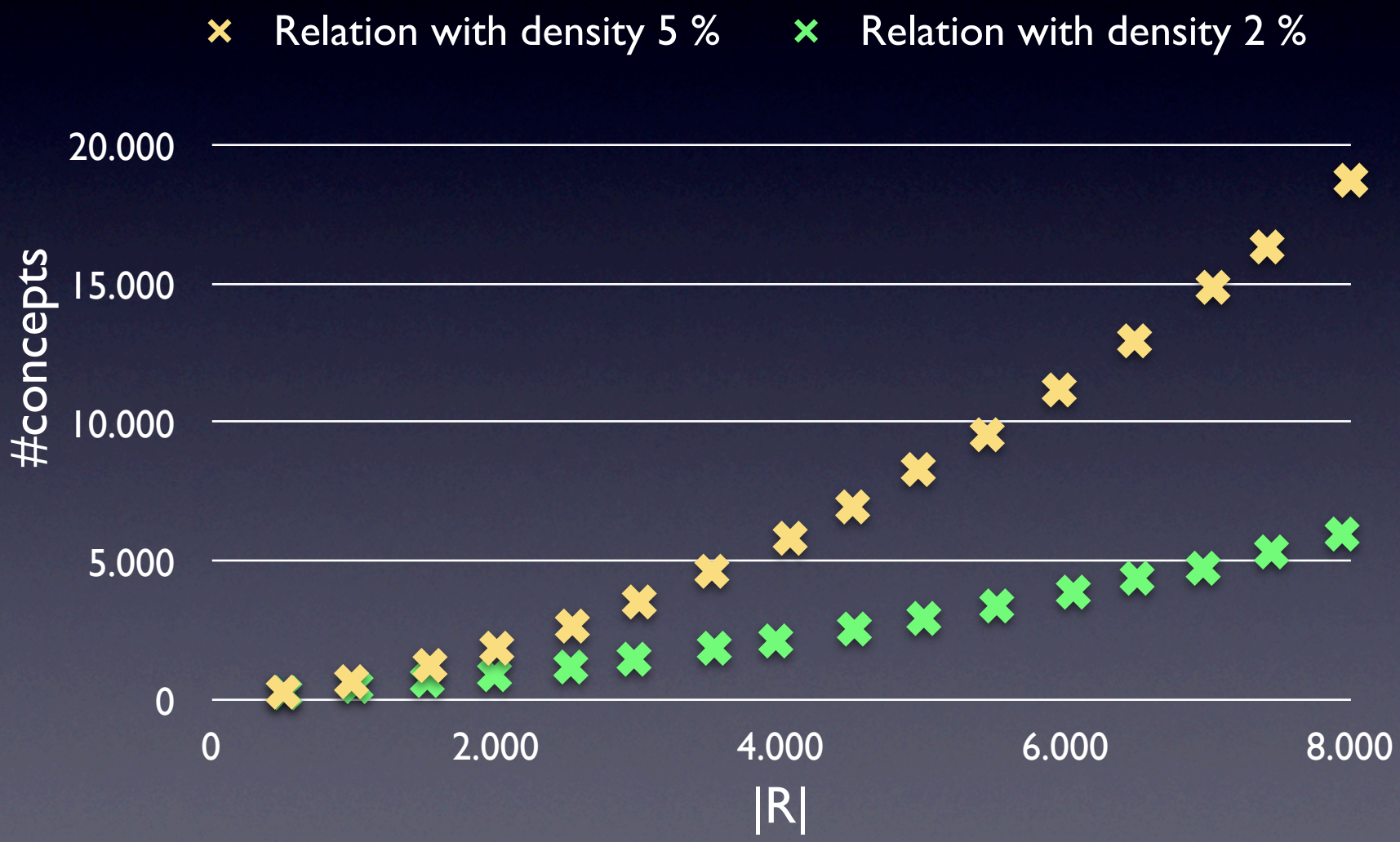
({stroke, move, scale},
{sync, lock, unlock})

Concept Lattice

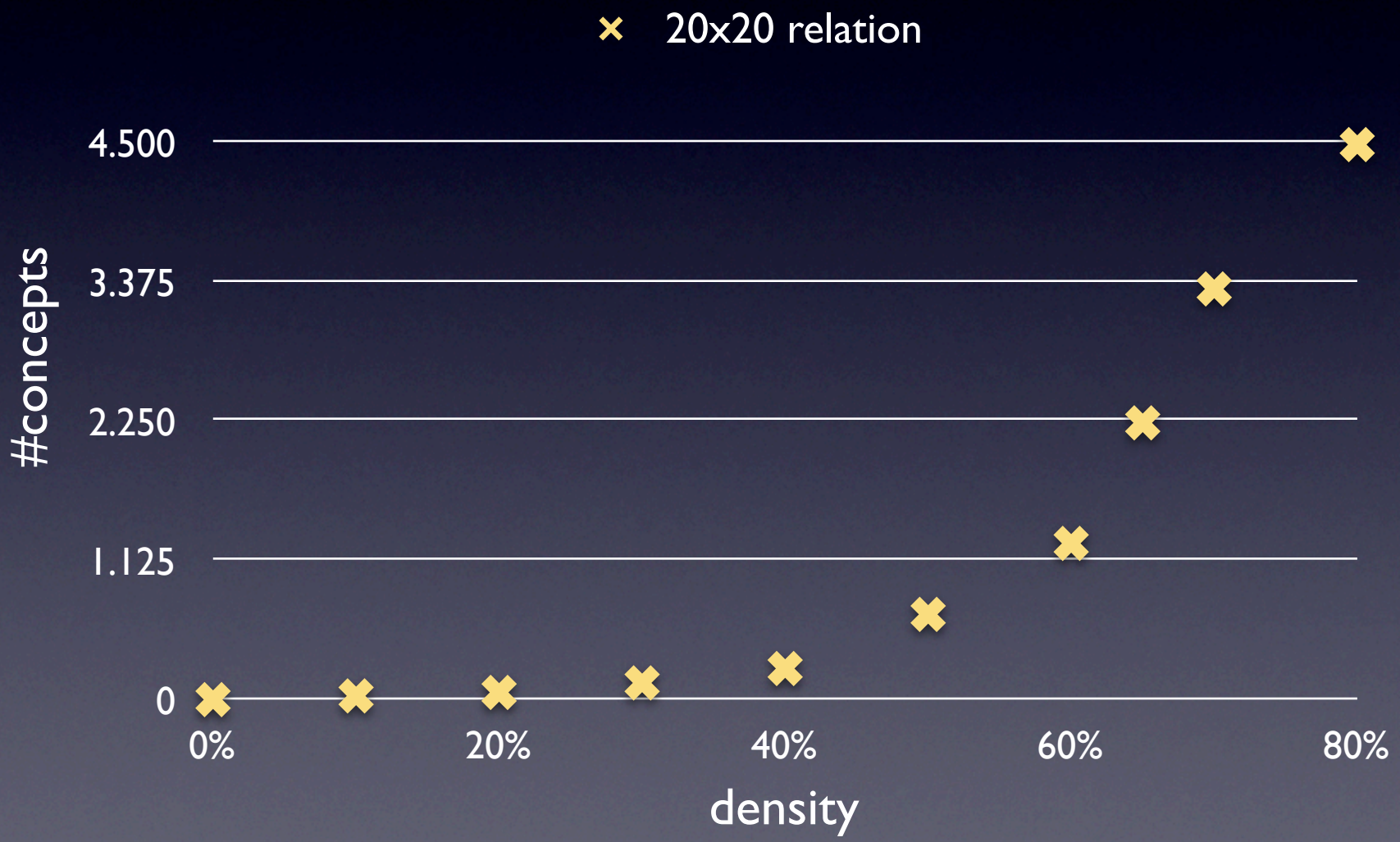




Lattice Size

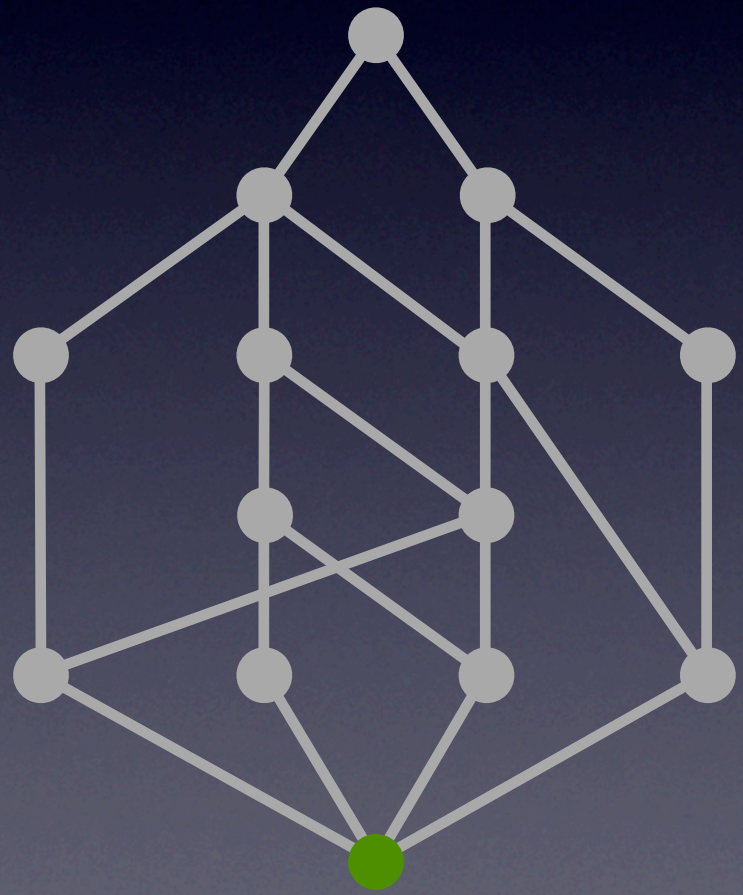


Worst Case Scenario



Algorithm

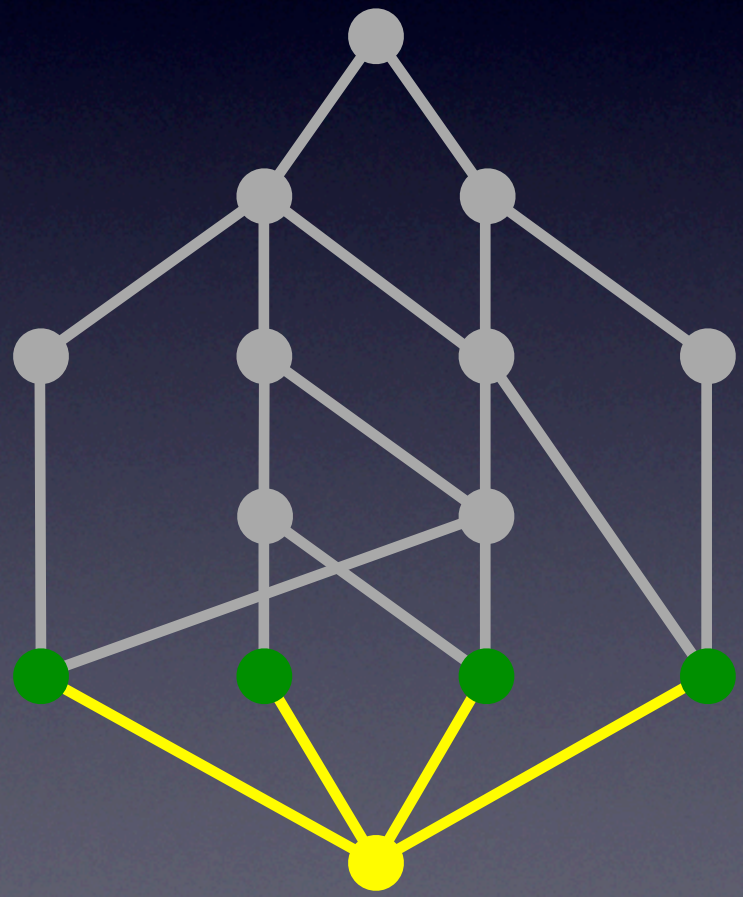
start at the bottom



Algorithm

start at the bottom

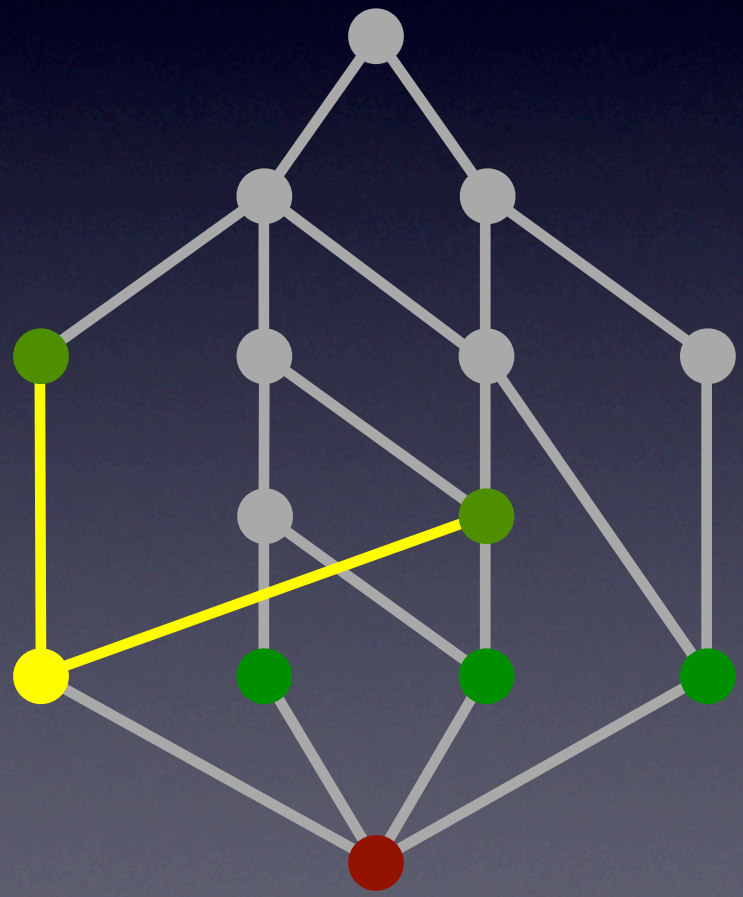
compute the upper neighbors recursively



Algorithm

start at the bottom

compute the upper neighbors recursively

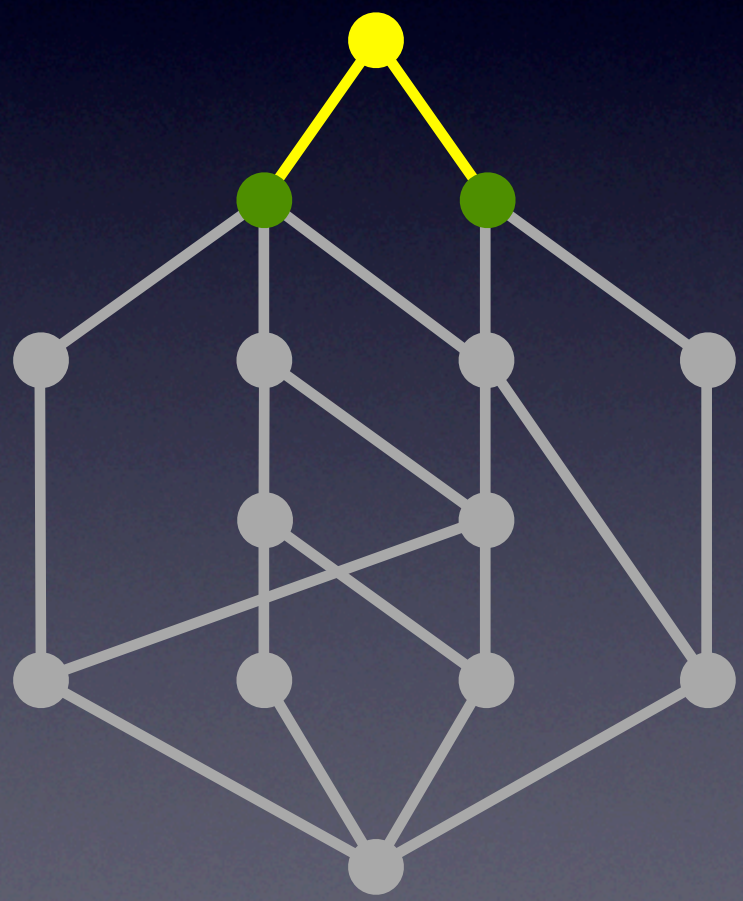


Algorithm

start at the bottom

compute the upper neighbors recursively

alternatively: start at the top



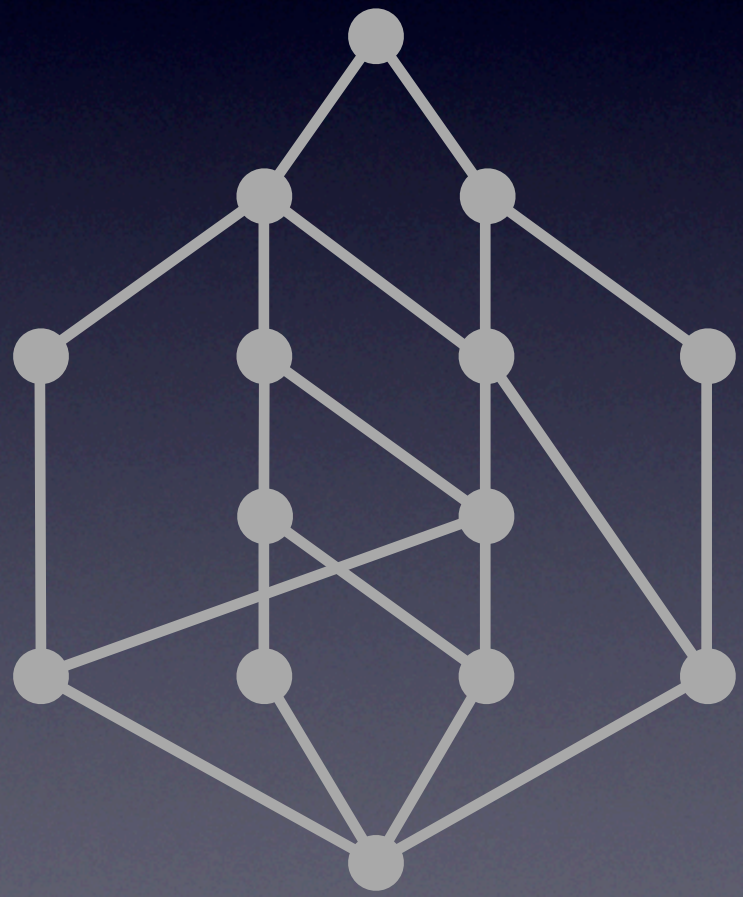
Algorithm

start at the bottom

compute the upper neighbors recursively

alternatively: start at the top

same idea for the edges

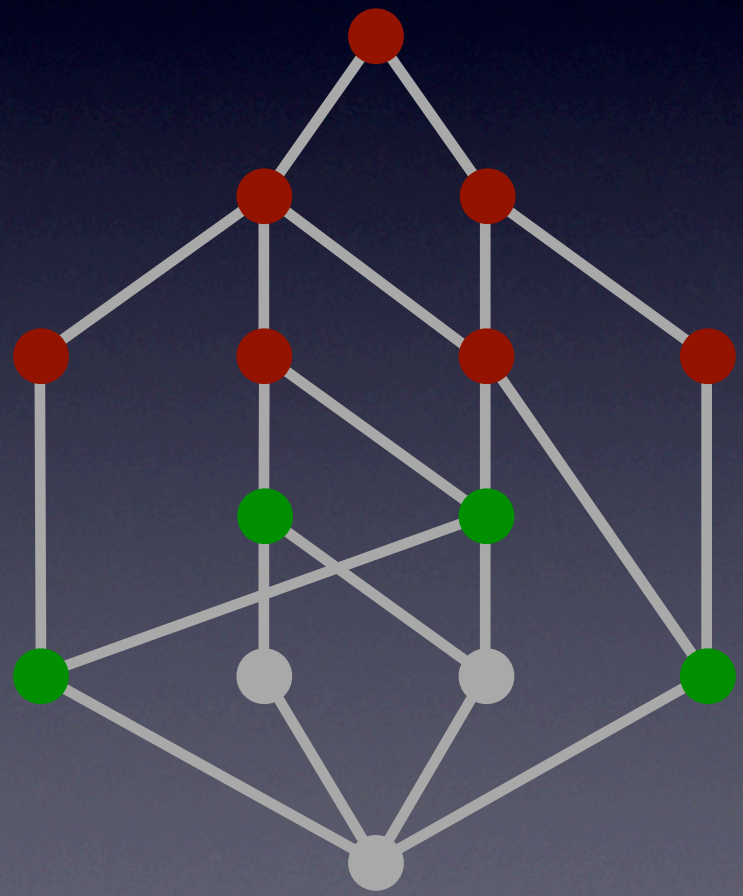


Implementation

iterators

don‘t pre-compute the entire lattice

less data in memory



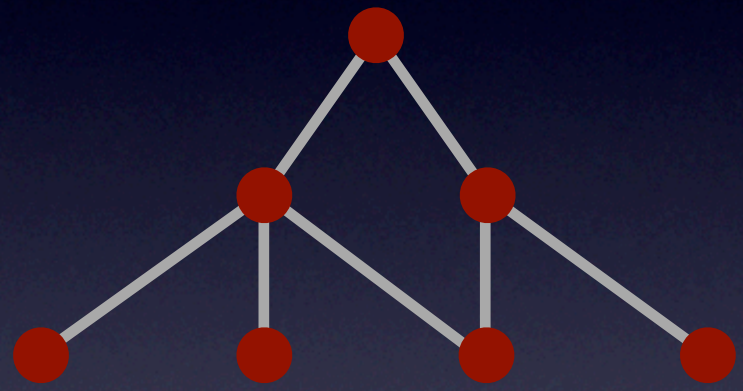
Implementation

iterators

don‘t pre-compute the entire lattice

less data in memory

only compute upper/lower parts of the lattice



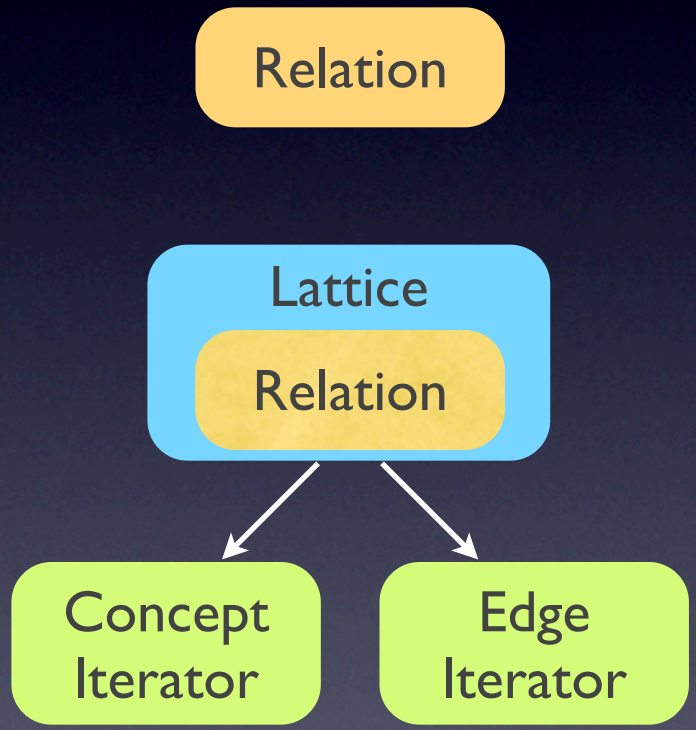
How to use it

generate a relation

generate a lattice for that relation

get an iterator from that lattice

use iterator as usual



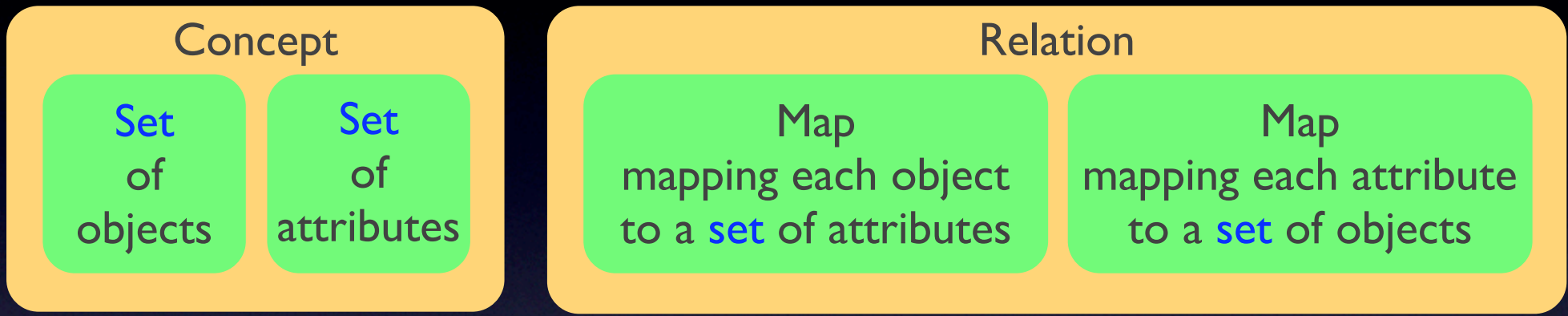
How to use it

```
Lattice lattice = new HybridLattice (rel);
Iterator<Concept> it = lattice.conceptIterator
                        (Traversal.TOP_ATTRSIZE);

while (it.hasNext()) {
    Concept c = it.next();
    System.out.println(c.toString());
}
```

How to use it

```
Lattice lattice = new HybridLattice (rel);
Iterator<Concept> it = lattice.conceptIterator
                              (Traversal.TOP_ATTRSIZE);

while (it.hasNext()) {
    Concept c = it.next();
    if (c.getAttributes().size > 10)
        break;
    System.out.println(c.toString());
}
```

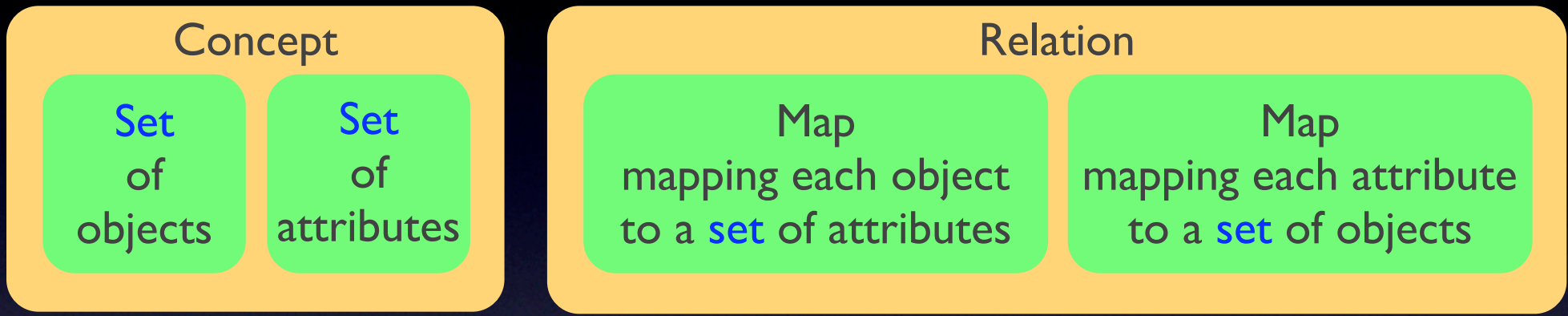
Data structures



Only allow java.lang.Comparable objects because

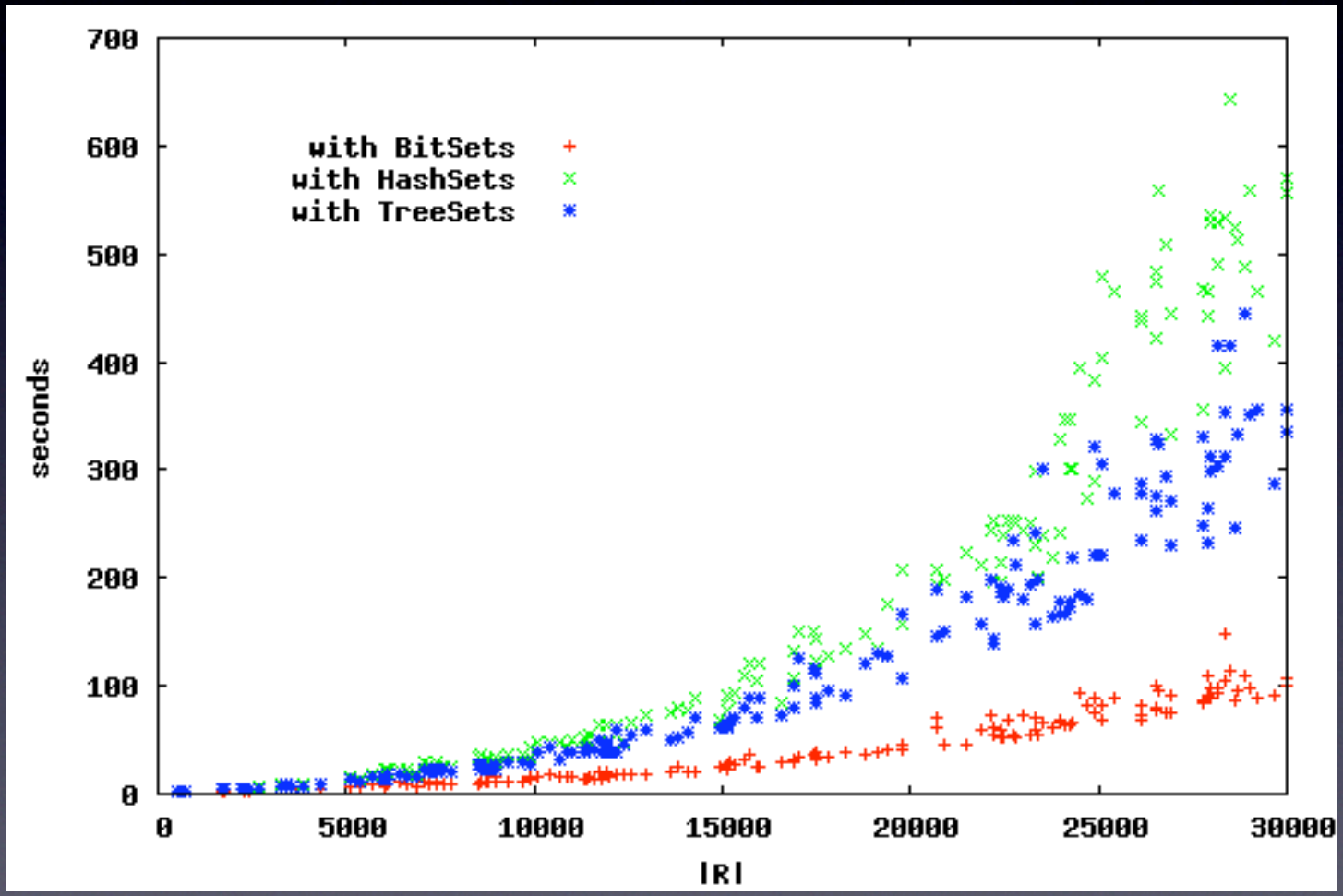
- they are usually immutable
- they can be sorted easily
- the user still has many possibilities

Data structures

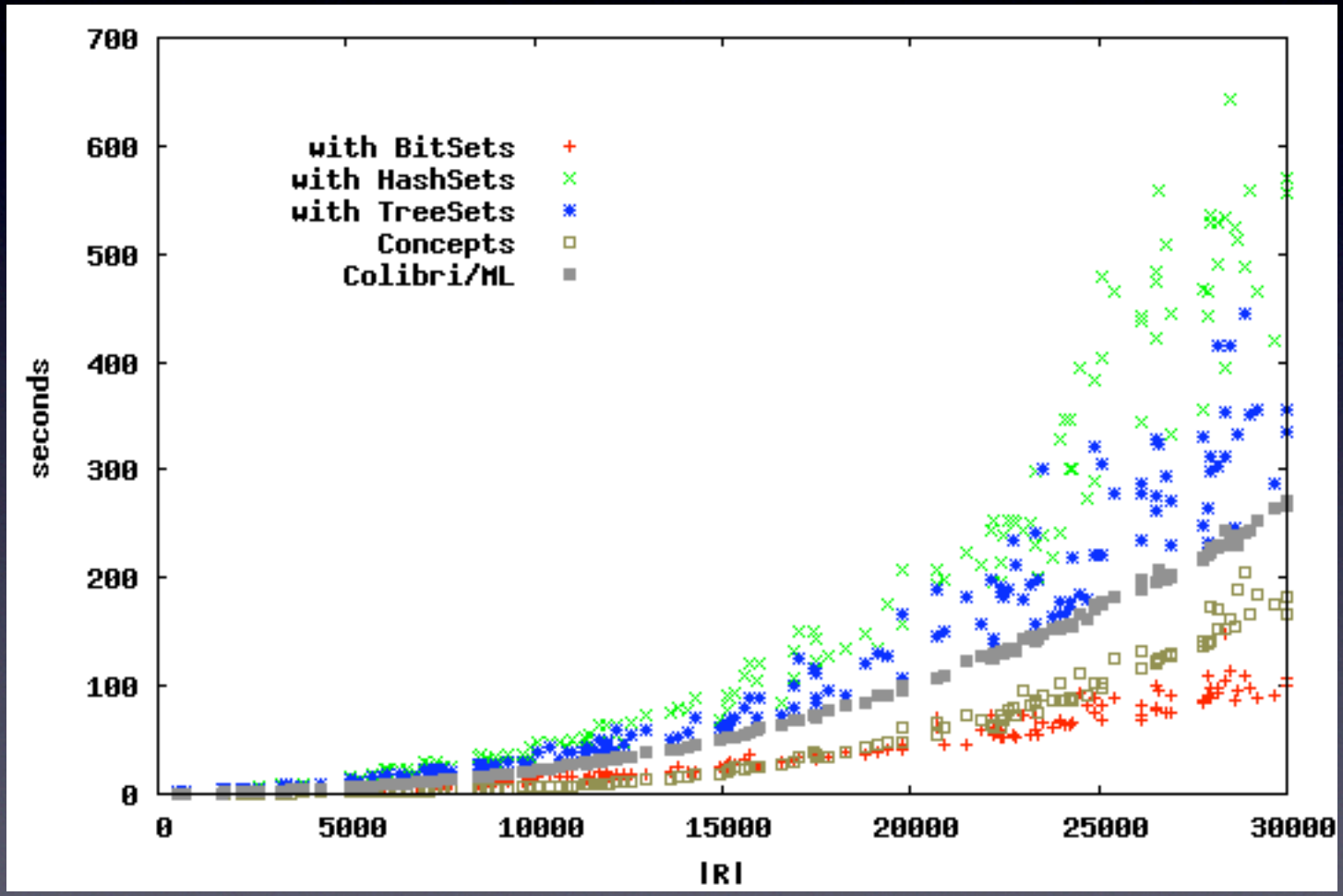


TreeSet	is a sorted set
HashSet	not sorted, but faster operations
BitSet	sorted, fast intersection operation, but can not be used directly

Performance



Performance



Correctness

JUnit tests

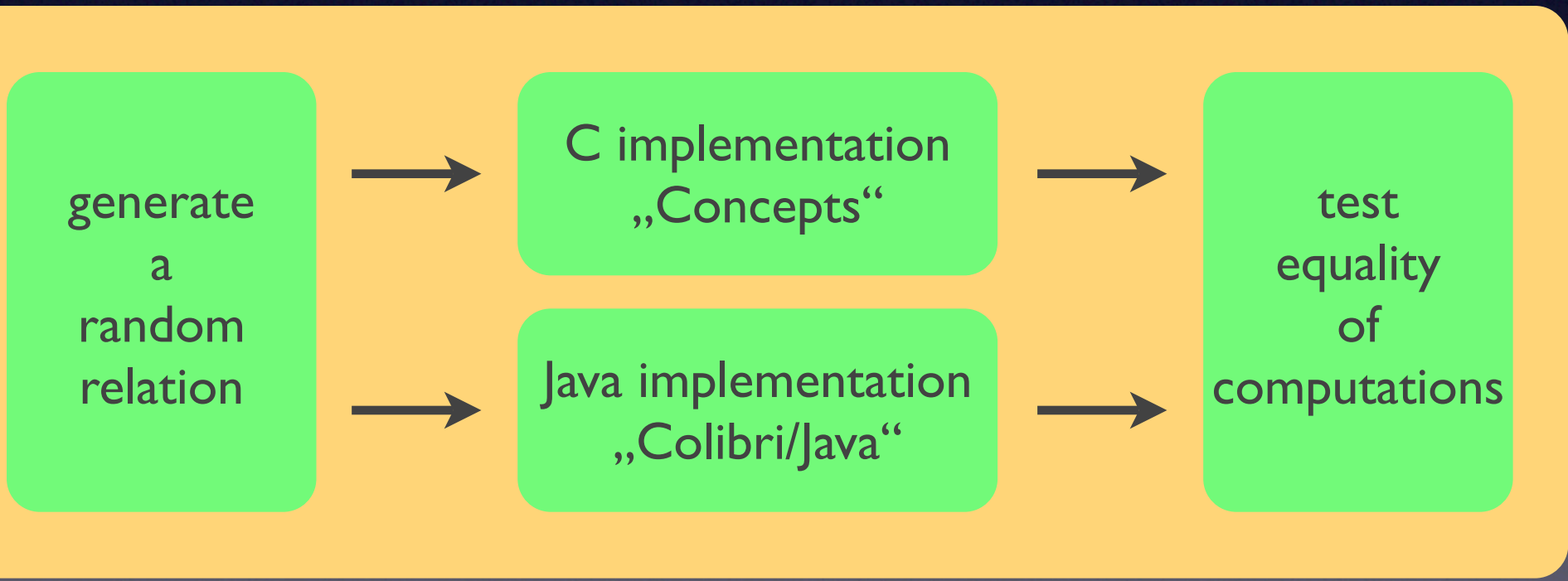
Problem:

Whether or not all concepts and edges are computed correctly can not be tested with unit tests.

Correctness

JUnit tests

+



Conclusion

Iterators are convenient to use.

BitSet-based data structure has best performance.

Good performance for large contexts with low density.

100% faster
than Colibri/ML

	|R| = 16000 density 1%	|R| = 30000 density 1%	|R| = 16000 density 4%
Bitset-based „Colibri/Java“	25 s	101 s	64 s
C implementation „Concepts“	24 s	166 s	586 s
ML implementation „Colibri/ML“	60 s	265 s	120 s

Next Step

clean up code

add some JavaDoc

release under GPL

publish at Google Code

become famous

Conclusion

Iterators are convenient to use.

BitSet-based data structure has best performance.

Good performance for large contexts with low density.

100% faster
than Colibri/ML

	|R| = 16000 density 1%	|R| = 30000 density 1%	|R| = 16000 density 4%
Bitset-based „Colibri/Java“	25 s	101 s	64 s
C implementation „Concepts“	24 s	166 s	586 s
ML implementation „Colibri/ML“	60 s	265 s	120 s