During the three practical assignments you will create a fully operational probabilistic model checker for discrete time Markov chains (DTMC) and (a variant of) the probabilistic computation tree logic (PCTL) as well as continuous time Markov chains (CTMC) and the continuous stochastic logic (CSL) in Java 1.6.

For each of the assignments we will provide the basic framework with the intended class structure. Consequently, your task is to implement the respective methods. You can infer the intended behavior of each method from its Java Doc comments.

You will find the framework for this assignment (Practical01.tgz) in the course management system. For your convenience we also provide a basic testing environment in terms of JUnit 3 test cases inside the frameworks. Please note that these tests do by far not cover all methods and do not prove correctness of your code and consequently, we may still manually investigate your submissions. Since the test cases depend on JUnit 3 you should add the respective library to your build path. Please also make sure that the folder test containing several test models can be found by your program.

Exercise 1.1 (Sparse Matrix Class)

Since the states of our models are usually not fully connected with each other, i.e. we assume a constant upper bound on the number of successor states, we will use a sparse matrix representation of the transition matrix of a DTMC (and later the rate matrix of a CTMC).

\[ P = \begin{pmatrix}
1 & 0 & 0 & 0 \\
0 & 0 & 0 & 1 \\
0 & 0.5 & 0.5 & 0 \\
1 & 0 & 0 & 0
\end{pmatrix} \]

\[ \text{Sp} = (\text{nonZeros} = \begin{bmatrix}
1 & 2 & 3 \\
3 & 1 & 0 \\
p_{13} = 1 & p_{21} = 0.5 & p_{30} = 1
\end{bmatrix}, \text{diagonal} = [1, 0, 0.5, 0]) \]

Consequently, for the off-diagonal part we only store the non-zero entries via their row and column indices as well as their value. The diagonal entries are stored as a vector. Please note that state indices within our model checker start with zero. The entries are instances of the class qmc.SparseMatrixEntry and the sparse matrix class qmc.SparseMatrix stores its non-diagonal entries invariantly within the vector nonZeros and the diagonal entries within the array diagonal. Your task is to implement the following methods:
• public SparseMatrix copy(): Create a new deep copy of the matrix, i.e. return a new SparseMatrix class that represents the same matrix and each sub-component (non-zero entries, diagonal) is also a new object. Consequently, changing the original matrix should not affect any of its copies.

• public void order(eOrder order): Order the entries of the vector nonZeros according to row or column depending on order. You should use Collections.sort and implement SparseMatrixEntry.RowOrder as well as SparseMatrixEntry.ColumnOrder.

• public void transpose(): Transpose the matrix, i.e. exchange row and column indices.

• public void multiplyVectorMatrix(double in[], double out[]): Assume the sparse matrix encodes matrix \( P \), your task is to calculate \( \text{out} = \text{in} \times P \).
  
  Hint: Do not forget to incorporate the diagonal entries.

• public void multiplyMatrixVector(double in[], double out[]): Assume the sparse matrix encodes matrix \( P \), your task is to calculate \( \text{out} = P \times \text{in} \).
  
  Hint: Do not forget to incorporate the diagonal entries.

• public void subMatrix(SparseMatrix submatrix, Set<Integer> states, HashMap<Integer,Integer> matrixToSub, HashMap<Integer,Integer> subToMatrix): Extract a sub-matrix containing only states specified in the input set states, i.e. it should represent exactly those transitions from states \( \in \text{states} \) to states \( \in \text{states} \). In order to relate the sub-matrix with the original one, the two HashMaps matrixToSub and subToMatrix shall map the state indices \( \in \text{states} \) of the original matrix to the state indices of the sub-matrix and vice versa. For example, consider the matrix \( P \) and its sparse matrix representation \( S_P \) from before. Then, \( S_P.\text{submatrix}(S'_P, \{1,2\}, \text{sub} \rightarrow \text{sub'}, \text{sub'} \rightarrow \text{sub}) \) would yield (up to the ordering of states inside \( S'_P \)):

\[
S'_P = \begin{bmatrix}
1 & 0 \\
0 & 0.5
\end{bmatrix}, \text{diagonal} = \begin{bmatrix}
0 & 0.5
\end{bmatrix}
\]

\[
S_P \rightarrow S'_P = \begin{bmatrix}
0 & 1 \\
1 & 2
\end{bmatrix}, S'_P \rightarrow S_P = \begin{bmatrix}
S'_P & S_P \\
1 & 0 \\
2 & 1
\end{bmatrix}
\]

• public void loadTRA(String filename): Implement a loader for sparse matrices from MRMC\(^1\) TRA files. The structure of a TRA file is:

```
STATES n
TRANSITIONS m
  row_1 column_1 value_1
  row_2 column_2 value_2
  ...
  row_m column_m value_m
```

\(^1\)http://www.mrmc-tool.org/
where row and column indices are integers \( \in \{1, \ldots, n\} \) and the values are double precision floating point numbers. Please note that when loading matrices from TRA files we will internally identify state 1 from the file with state index 0 inside the sparse matrix class. You may throw an exception if the file can not be parsed.

- **public void print()**: Print the sparse matrix non-zero entries as well as the diagonal to standard output. The purpose of this method is to ease debugging. Therefore, the output format is up to you.

**Exercise 1.2 (Discrete Time Markov Chain Class - Part 1)**

The `qmc.DTMC` class is derived from the `qmc.SparseMatrix` class and shall contain additional methods for DTMC transient and steady state analysis as well as in future assignment sheets methods for bounded and unbounded reachability. Your task is to implement the following methods:

- **public DTMC copy()**: Again, implement a method returning a deep copy of the object.

- **public double maxRelDiff(double[] cur, double[] next)**: Calculate the maximum relative difference between the vectors `cur` and `next` by the means of the following formula:

  \[
  \text{maxRelDiff}(\text{cur}, \text{next}) = \max_i d(\text{cur}[i], \text{next}[i])
  \]

  where

  \[
  d(x, x') = \begin{cases} 
  \frac{|x - x'|}{x'} & x' > 0 \\
  0 & |x - x'| = 0 \land x' = 0 \\
  1 & \text{else}
  \end{cases}
  \]

- **public void steadyState(eSteadyStateSolver solver, double[] p[], double precision, double omega)**: Assuming the steady-state limit exists, this method shall approximate this distribution. Depending on `solver` you shall use different algorithms for that task. If `solver=POWER`, the power method shall be applied. More precisely, given the start distribution \( p_0 = p \) you shall compute \( p_{i+1} = p_i P \) as long as the maximum relative difference between \( p_i \) and \( p_{i+1} \) is greater than `precision`. The final probability distribution shall be returned in `p` again.

  *Hint*: In order to speed up your implementation you should use two vectors for the current and next distribution whose references you can swap at each iteration. This way you do not have to create a new array at each step. When swapping the referenced arrays, make sure you return the right vector in the end.

  **Bonus**: Also implement the **GAUSS_SEIDEL** and **SOR** (Successive-Over-Relaxation) solvers. These methods may converge faster than the power method. You might refer to the respective Wikipedia pages. Please implement one iteration of the SOR solving step with relaxation parameter `omega` in the `DTMC.sor` method and implement the normalization \( v \cdot \|v\|_1^{-1} \) of a vector \( v \) in `DTMC.normalize`. Hint: When you solve the steady state using Gauss-Seidel and SOR, you may use `SparseMatrix.order` to order your non-zero entries column-wise since in these methods – in contrast to the power method – you do not need the information where the probability mass flows from the current state but where the probability mass comes from. Do not forget to normalize your solution in the end.
public void transientSteps(double[] p, int steps, double precision):
Starting with distribution \( p_0 = p \), iteratively calculate \( p_{i+1} = p_i P \) up to index \( \text{steps} \) and return the resulting distribution in \( p = p_{\text{steps}} \). You should terminate prematurely if you detect a steady state, i.e. \( \text{maxRelDiff}(p_i, p_{i+1}) < \text{precision} \).

*Hint*: Again, you should use the reference swapping trick to speed up your implementation.

The unit tests qmc.test.DTMCSteadyStateTest and qmc.test.DTMCTransientTest will both load TRA files from test/DTMC/, calculate some transient and steady state probability distributions, and compare them to reference values. Please manually make sure that all other methods (in any case subMatrix, and SOR/Gauss-Seidel solving - if you are heading for bonus points) are correct. You are encouraged to send in unit tests for these methods.