```java
package problems.qbf.solvers;

import java.io.IOException;
import java.util.ArrayList;

import metaheuristics.grasp.AbstractGRASP;
import problems.qbf.QBF_Inverse;
import solutions.Solution;



/**
* Metaheuristic GRASP (Greedy Randomized Adaptive Search Procedure) for
* obtaining an optimal solution to a QBF (Quadractive Binary Function --
* {@link #QuadracticBinaryFunction}). Since by default this GRASP considers
* minimization problems, an inverse QBF function is adopted.
*
* @author ccavellucci, fusberti
*/
public class GRASP_QBF extends AbstractGRASP<Integer> {

   /**
    * Constructor for the GRASP_QBF class. An inverse QBF objective function is
    * passed as argument for the superclass constructor.
    *
    * @param alpha
    *            The GRASP greediness-randomness parameter (within the range
    *            [0,1])
    * @param iterations
    *            The number of iterations which the GRASP will be executed.
    * @param filename
    *            Name of the file for which the objective function parameters
    *            should be read.
    * @throws IOException
    *             necessary for I/O operations.
    */
   public GRASP_QBF(Double alpha, Integer iterations, String filename) throws
IOException {
       super(new QBF_Inverse(filename), alpha, iterations);
   }

   /*
```

```
     * (non-Javadoc)
     *
     * @see grasp.abstracts.AbstractGRASP#makeCL()
     */
    @Override
    public ArrayList<Integer> makeCL() {

        ArrayList<Integer> _CL = new ArrayList<Integer>();
        for (int i = 0; i < ObjFunction.getDomainSize(); i++) {
            Integer cand = i;
            _CL.add(cand);
        }

        return _CL;

    }

    /*
     * (non-Javadoc)
     *
     * @see grasp.abstracts.AbstractGRASP#makeRCL()
     */
    @Override
    public ArrayList<Integer> makeRCL() {

        ArrayList<Integer> _RCL = new ArrayList<Integer>();

        return _RCL;

    }

    /*
     * (non-Javadoc)
     *
     * @see grasp.abstracts.AbstractGRASP#updateCL()
     */
    @Override
    public void updateCL() {

        // do nothing since all elements off the solution are viable candidates.

    }
```

```java
/**
 * {@inheritDoc}
 *
 * This createEmptySol instantiates an empty solution and it attributes a
 * zero cost, since it is known that a QBF solution with all variables set
 * to zero has also zero cost.
 */
@Override
public Solution<Integer> createEmptySol() {
    Solution<Integer> sol = new Solution<Integer>();
    sol.cost = 0.0;
    return sol;
}

/**
 * {@inheritDoc}
 *
 * The local search operator developed for the QBF objective function is
 * composed by the neighborhood moves Insertion, Removal and 2-Exchange.
 */
@Override
public Solution<Integer> localSearch() {

    Double minDeltaCost;
    Integer bestCandIn = null, bestCandOut = null;

    do {
        minDeltaCost = Double.POSITIVE_INFINITY;
        updateCL();

        // Evaluate insertions
        for (Integer candIn : CL) {
            double deltaCost = ObjFunction.evaluateInsertionCost(candIn, sol);
            if (deltaCost < minDeltaCost) {
                minDeltaCost = deltaCost;
                bestCandIn = candIn;
                bestCandOut = null;
            }
        }
        // Evaluate removals
        for (Integer candOut : sol) {
```

```
                double deltaCost = ObjFunction.evaluateRemovalCost(candOut, sol);
                if (deltaCost < minDeltaCost) {
                    minDeltaCost = deltaCost;
                    bestCandIn = null;
                    bestCandOut = candOut;
                }
            }
            // Evaluate exchanges
            for (Integer candIn : CL) {
                for (Integer candOut : sol) {
                    double deltaCost = ObjFunction.evaluateExchangeCost(candIn,
candOut, sol);
                    if (deltaCost < minDeltaCost) {
                        minDeltaCost = deltaCost;
                        bestCandIn = candIn;
                        bestCandOut = candOut;
                    }
                }
            }
            // Implement the best move, if it reduces the solution cost.
            if (minDeltaCost < -Double.MIN_VALUE) {
                if (bestCandOut != null) {
                    sol.remove(bestCandOut);
                    CL.add(bestCandOut);
                }
                if (bestCandIn != null) {
                    sol.add(bestCandIn);
                    CL.remove(bestCandIn);
                }
                ObjFunction.evaluate(sol);
            }
        } while (minDeltaCost < -Double.MIN_VALUE);

        return null;
    }

    /**
     * A main method used for testing the GRASP metaheuristic.
     *
     */
    public static void main(String[] args) throws IOException {
```

```
        long startTime = System.currentTimeMillis();
        GRASP_QBF grasp = new GRASP_QBF(0.05, 1000, "instances/qbf/qbf040");
        Solution<Integer> bestSol = grasp.solve();
        System.out.println("maxVal = " + bestSol);
        long endTime   = System.currentTimeMillis();
        long totalTime = endTime - startTime;
        System.out.println("Time = "+(double)totalTime/(double)1000+" seg");

    }

}
```