JAMES: A modern object-oriented Java framework for discrete optimization using local search metaheuristics

Herman De Beukelaer
Ghent University
Krijgslaan 281 S9
9000 Gent, Belgium

Guy F. Davenport
Bayer CropScience NV
Technologiepark 38
9052 Zwijnaarde, Belgium

Geert De Meyer
Bayer CropScience NV
Technologiepark 38
9052 Zwijnaarde, Belgium

Veerle Fack
Ghent University
Krijgslaan 281 S9
9000 Gent, Belgium

Abstract

This paper describes JAMES, a modern object-oriented Java framework for discrete optimization using local search algorithms that exploits the generality of such metaheuristics by clearly separating search implementation and application from problem specification. A wide range of generic local searches are provided, including (stochastic) hill climbing, tabu search, variable neighbourhood search and parallel tempering. These can be applied easily to any user-defined problem by plugging in a custom neighbourhood for the corresponding solution type. The performance of several different search algorithms can be assessed and compared in order to select an appropriate optimization strategy. Also, the influence of parameter values can be studied. Implementations of specific components are included for subset selection, such as a predefined solution type, a generic problem definition and several subset neighbourhoods used to modify the set of selected items. Additional components for other types of problems (e.g. permutation problems) are provided through an extensions module. Releases of JAMES are deployed to the Maven Central Repository so that the framework can easily be included as a dependency in other Java applications. The project is fully open source and hosted on GitHub. More information can be found at http://www.jamesframework.org.

KEYWORDS

Discrete optimization, metaheuristics, local search, Java framework, object-oriented architecture.

1. INTRODUCTION

Many optimization problems are difficult to solve, e.g. due to NP-completeness, in which case exact techniques are often not applicable. A common practical approach to deal with this issue is to use inexact algorithms that find valuable approximations of the best solution within reasonable time. For this purpose, metaheuristics are frequently applied, with the major advantage that they can be adjusted easily to solve various optimization problems arising from different fields, i.e. with the addition of only the necessary problem specific components such as neighbourhood functions in case of a local search or crossover, mutation and selection operators in case of a genetic algorithm. In this context, software frameworks are valuable tools to reduce the effort needed to apply well-established metaheuristics to newly defined problems. Such frameworks are also helpful for the implementation of new ideas and comparison with existing algorithms, and to create hybrid combinations of different search techniques.

Several metaheuristic optimization frameworks have been developed over the last few decades [1], each targeting a certain class of algorithms and/or specific type of applications, implemented in a variety of object-oriented programming languages such as C++, C# and Java. For example, ParadisEO [2] is an extensive C++ framework that supports both single- and multiobjective optimization using local search and population-based metaheuristics, with extensions for parallel and distributed computation. Other options for C++ users include EasyLocal [3] and MALLBA [4]. The widely used Java framework jMetal [5] mainly targets multiobjective optimization using population-based algorithms. Similarly, other proposed Java frameworks are also focused on population-based methods and especially evolutionary algorithms (e.g. ECJ [6], EvA2 [7], JCLEC [8], Opt4j [9]). However, these are computationally expensive techniques that might not be needed when dealing with problems of moderate complexity for which simpler, local search based techniques may perform well enough. To our knowledge, the only available Java framework that includes a variety of local search metaheuristics is FOM [10] which unfortunately suffers from issues such as a lack of sufficient documentation and limited code transparency, and has not recently been updated.

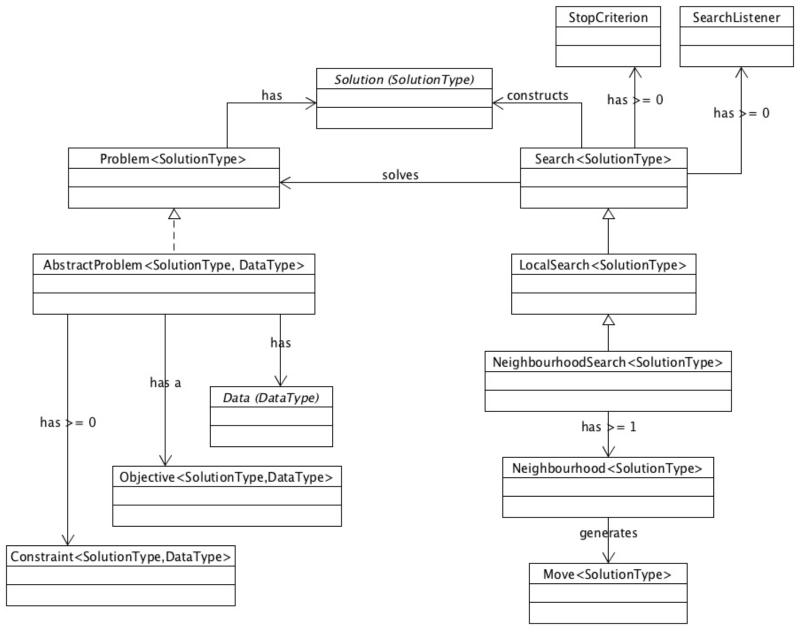
The above considerations led to the development of JAMES, a modern Java 8 framework for discrete optimization using local search metaheuristics. As Java is one of the most used programming languages, this framework is a valuable addition to the currently available tools. It is desirable that such framework is transparent, flexible, well-documented, easy to use, and preferably open-source (distributed under a permissive license) and hosted on a generally accessible code sharing platform such as GitHub[1]. Java has the additional advantage of being highly portable across different systems (Windows, Unix). JAMES includes a wide range of basic and advanced local searches, including simple (stochastic) hill climbing as well as tabu search, variable neighbourhood search and parallel tempering (also called Replica Exchange Monte Carlo search, REMC). Releases are deployed to the Maven Central Repository[2] so that JAMES can easily be included as a dependency in other Java applications. The project is licensed under the Apache License v2.0[3]. More information and extensive documentation can be found at http://www.jamesframework.org.

The remainder of this paper is organized as follows. In the next section, the high-level architecture of JAMES v0.2 is described. Sections 3 and 4 demonstrate how to define a problem and how to obtain good solutions using one of the available optimization algorithms, respectively. Finally, section 5 presents conclusions and future work.

2. ARCHITECTURE OF JAMES

The JAMES framework strongly separates problem specification from search application so that existing algorithms can easily be applied to obtain solutions for newly implemented problems (Figure 1). Each problem has a specific solution type and a search creates solutions of this type to solve the problem. The search talks to the problem to obtain random solutions (e.g. used as the default initial solution of a local search) and to evaluate and validate constructed solutions. Some utilities are provided to split the problem specification into an objective, data and possibly a number of constraints. The objective and constraints are then responsible for evaluating and validating solutions, respectively, using the data.

Figure 1 High-level architecture of the JAMES framework



The optimization algorithms are organized hierarchically. The top-level search definition handles general behaviour such as tracking the best solution found so far and termination (stop criteria). It also informs any listeners when certain events have occurred, e.g. when a new best solution has been found. A local search adds the concept of a current (and initial) solution which is modified in an attempt to improve it, i.e. which moves towards an optimum along a certain trajectory. The latter is usually performed by repeatedly sampling moves from one or more neighbourhoods that slightly change, and hopefully improve, the current solution. Such

[1] GitHub: https://github.com.
[2] Maven Central Repository: http://search.maven.org.
[3] Apache License v2.0: http://www.apache.org/licenses/LICENSE-2.0.

algorithms belong to the class of neighbourhood searches. The applied neighbourhoods should be compatible with the solution type of the problem being solved and are used to adjust the search strategy to a specific application.

The core module of JAMES contains all high-level components shown in Figure 1, as well as algorithm implementations, general stop criteria, etc. It also includes implementations of specific components for subset selection, such as a predefined solution type, a generic problem definition and several subset neighbourhoods. Similar components can easily be added for other types of problems, and distributed through an extensions module when desired (currently, the extensions include additional components for permutation problems). The next two sections demonstrate how to implement and solve a basic subset selection problem. More examples are available at the website, which also address other types of problems (e.g. the well-known travelling salesman problem, TSP).

3. PROBLEM SPECIFICATION

This section describes the implementation of the following example problem: given a set of items and a complete distance matrix, select a fixed size subset with maximum average distance between all pairs of selected items. This problem is referred to as the core selection problem and originates from the field of plant genomics and crop science, where diverse subsets of large collections of crop varieties often need to be selected [11]. This is a selection problem for which the predefined components in JAMES can be used.

A generic *SubsetProblem* implementation is provided which extends *AbstractProblem* and thus separates the data from the objective and constraints (see Figure 1). The high-level *SubsetProblem* definition requires that each item from the data set is identified with a unique integer ID so that the problem can be generically solved by selecting a subset of these IDs. This translates to the requirement that the data class needs to implement the *IntegerIdentifiedData* interface, which defines a single method *getIDs()* that returns the set of all item IDs. Figure 2 shows the implementation of a custom *CoreSubsetData* class that wraps a distance matrix, where the IDs correspond to the indices in this matrix.

Figure 2 Providing data for the core selection problem

```
public class CoreSubsetData implements IntegerIdentifiedData {

  private double[][] dist;
  private Set<Integer> ids;

  public CoreSubsetData(double[][] dist){
    this.dist = dist;
    // infer IDs (indices in distance matrix)
    ids = new HashSet<>();
    for(int id=0; id<dist.length; id++){
      ids.add(id);
    }
  }

  public double getDistance(int id1, int id2){
    return dist[id1][id2];
  }

  public Set<Integer> getIDs() {
    return ids;
  }

}
```

The objective is defined by implementing the *Objective* interface and specifying the solution and data types. For any selection problem, the predefined solution type *SubsetSolution* can be used, which models the set of selected items. For this specific example, the data type is set to *CoreSubsetData*. The objective is responsible for evaluating a given solution, using the data, and informs the search whether these evaluations are to be maximized or minimized. Figure 3 shows an implementation of the core selection objective that evaluates a subset by computing the average distance between all pairs of selected items, which is to be maximized. The result of *evaluate(solution, data)* is a *SimpleEvaluation* that simply wraps a double value (line 14). More complicated evaluation types can be used as well, e.g. when providing an efficient delta evaluation; for examples, the reader is directed to the website.

Figure 3 Defining the objective of the core selection problem

```
public class CoreSubsetObjective implements Objective<SubsetSolution, CoreSubsetData>{

  public Evaluation evaluate(SubsetSolution solution, CoreSubsetData data) {
    int numDist = 0;
    double sumDist = 0.0;
    Integer[] selected = new Integer[solution.getNumSelectedIDs()];
    solution.getSelectedIDs().toArray(selected);
    for(int i=0; i<selected.length; i++){
      for(int j=i+1; j<selected.length; j++){
        sumDist += data.getDistance(selected[i], selected[j]);
        numDist++;
      }
    }
    return new SimpleEvaluation(sumDist/numDist);
  }

  public boolean isMinimizing() {
    return false;
  }

}
```

Now that the data and objective have been defined, they can be combined in a *SubsetProblem* (Figure 4). The desired subset size is specified as well (line 4). There are no constraints for the core selection problem, i.e. all subsets are valid solutions. Also, the high-level *SubsetProblem* definition is already capable of generating random subsets, so this does not need to be addressed here.

Figure 4 Finalizing the core selection problem specification

```
1: double[][] dist = ... // set distance matrix (e.g. read from a file)
2: CoreSubsetData data = new CoreSubsetData(dist); // initialize data
3: CoreSubsetObjective obj = new CoreSubsetObjective(); // create objective
4: int subsetSize = ... // specify desired subset size
5: SubsetProblem<CoreSubsetData> problem = new SubsetProblem<>(obj, data, subsetSize);
```

There are two options when specifying other types of problems besides subset selection. One can directly implement the *Problem* interface to create a single self-contained problem definition that includes all necessary data and performs both evaluation and validation of constructed solutions. Alternatively, the data, objective and constraints can be separated by extending *AbstractProblem*, similarly to what has been demonstrated above for selection problems. Examples of both approaches are provided at the website.

4. SEARCH APPLICATION

The core selection problem, as defined in the previous section, can now easily be solved using any of the available metaheuristics. This section demonstrates how to apply a basic stochastic hill climber (random descent) which starts from a random solution and iteratively applies randomly chosen moves, from a given neighbourhood, to the current solution. A move is only accepted if it improves the current solution, else a different move is tried. Several predefined subset neighbourhoods are available that can be used for any selection problem. Here, a *SingleSwapNeighbourhood* is applied, which randomly removes one item from the selection and replaces it with a randomly chosen, currently unselected item. Figure 5 shows how to create and run a random descent algorithm to optimize the selected core subset.

A variety of stop criteria can be used to decide when the search should terminate, such as a runtime or step count limit, or a maximum amount of time or number of steps without finding any improvements. In this example, a runtime limit of 30 seconds is set. Calling *search.start()* (line 5) executes the optimization algorithm and blocks until it has terminated, after which the best found solution and corresponding score are printed. Finally, the search is disposed so that all resources are properly released (line 8).

Figure 5 Selecting a good core subset using a stochastic hill climber (random descent)

```
Neighbourhood<SubsetSolution> neigh = new SingleSwapNeighbourhood();
RandomDescent<SubsetSolution> search = new RandomDescent<>(problem, neigh);
search.addStopCriterion(new MaxRuntime(30, TimeUnit.SECONDS));

search.start();
System.out.println("Best solution: " + search.getBestSolution().getSelectedIDs());
System.out.println("Best score: " + search.getBestSolutionEvaluation());
search.dispose();
```

For many problems, a simple hill climber is not powerful enough to find good, stable solutions because it can not escape from local optima. However, for the considered core selection problem it performs very well and there is no need to turn to more advanced methods. Examples of more difficult problems which are solved using other techniques such as parallel tempering or variable neighbourhood search, with both predefined or custom neighbourhoods, are provided at the website.

5. CONCLUSIONS

This paper proposed the JAMES framework (v0.2) for discrete optimization using local searches in Java. It is a valuable addition to the currently available Java metaheuristic optimization tools which mainly focus on evolutionary algorithms. By clearly separating problem specification from search application, the provided algorithms can easily be used to solve newly defined problems, as was demonstrated for a simple fixed size subset selection problem. Future work includes the addition of analysis tools that can be used to compare the performance of different algorithms and to assess the influence of parameter values. Also, new search methods will be added when needed.

ACKNOWLEDGEMENT

Herman De Beukelaer is supported by a Ph.D. grant from the Research Foundation of Flanders (FWO).

REFERENCES

[1] Parejo J.A., Ruiz-Cortés A., Lozano S. and Fernandez P., 2012. Metaheuristic optimization frameworks: a survey and benchmarking. *Soft Computing*, 16(3), 527-561.

[2] Cahon S., Melab N. and Talbi E.G., 2004. ParadisEO: A framework for the reusable design of parallel and distributed metaheuristics. *Journal of Heuristics*, 10(3), 357-380.

[3] Di Gaspero L. and Schaerf A., 2003. EASYLOCAL++: an object-oriented framework for the flexible design of local-search algorithms. *Software: Practice and Experience*, 33(8), 733-765.

[4] Alba E., Luque G., Garcia-Nieto J. and Ordonez G., 2007. MALLBA: a software library to design efficient optimisation algorithms. *International Journal of Innovative Computing and Applications*, 1(1), 74-85.

[5] Durillo J.J. and Nebro A.J., 2011. jMetal: A Java framework for multi-objective optimization. *Advances in Engineering Software*, 42(10), 760-771.

[6] White D.R., 2012. Software review: the ECJ toolkit. *Genetic Programming and Evolvable Machines*, 13(1), 65-67.

[7] Kronfeld M., Planatscher H. and Zell A., 2010, The EvA2 optimization framework. *Learning and Intelligent Optimization*. Venice, Italy, 247-250.

[8] Ventura S., Romero C., Zafra A., Delgado J.A. and Hervás C., 2008. JCLEC: a Java framework for evolutionary computation. *Soft Computing*, 12(4), 381-392.

[9] Lukasiewycz M., Glaß M., Reimann F. and Teich J., 2011, Opt4J: a modular framework for meta-heuristic optimization. *Proceedings of the 13th annual conference on Genetic and evolutionary computation*. Dublin, Ireland, 1723-1730.

[10] Parejo J.A., Racero J., Guerrero F., Kwok T. and Smith K.A., 2003, Fom: A framework for metaheuristic optimization. *Computational Science—ICCS 2003*. Melbourne, Australia, 886-895.

[11] Brown A.H.D., 1989. Core collections: a practical approach to genetic resources management. *Genome*, 31(2), 818-824.