

Quick start guide for ParamILS, version 2.2

Frank Hutter
Department of Computer Science
University of British Columbia
Vancouver, BC V6T 1Z4, Canada. `hutter at cs dot ubc dot ca`

October 23, 2007

1 Introduction

ParamILS [1, 3] is a tool for parameter optimization. It works for any parameterized algorithm whose parameters can be discretized. ParamILS searches through the space of possible parameter configurations, evaluating configurations by running the algorithm to be optimized on a set of benchmark instances.

Users provide

- a parametric algorithm \mathcal{A} (executable to be called from the command line),
- all parameters and their possible values (parameters need to be configurable from the command line), and
- a set of benchmark problems, \mathcal{S} .

Users can also choose from a multitude of optimization objectives, reaching from minimizing average runtime to maximizing median approximation qualities.

ParamILS then executes algorithm \mathcal{A} with different combinations of parameters on instances sampled from \mathcal{S} , searching for the configuration that yields overall best performance across the benchmark problems. For details, see [3]. If you use ParamILS in your research, please cite that article. It would also be nice if you sent us an email – we are always interested in additional application domains.

2 Download and Configuration

This quick start guide is for **version 2.2** of ParamILS. There is also an older quick start guide for version 1.0. Compared to that version, the user interface has been much improved.

Download the zip file `paramils2.2.zip` from <http://www.cs.ubc.ca/labs/beta/Projects/ParamILS> and unzip it in a new directory. The version in this zip file is a Linux executable,

and currently such an executable is only available for Linux. If you wish to use ParamILS for other platforms, you can do so, but you will have to download the source code version (paramils2.2-source.zip), and probably fiddle with path issues etc. (see the quick start guide for version 1.0)

ParamILS is written in Ruby. Ruby is a scripting language similar to Perl, but object-oriented and *a lot* easier to read – Ruby can be freely downloaded for all platforms from <http://www.ruby-lang.org/en/>. The executable above has been generated with RubyScript2Exe¹.

3 Example tuning scenarios

The zip file contains examples for three algorithms: Saps [4], a local search algorithm for SAT; Spear², a tree search algorithm for SAT (and satisfiability modulo theories) developed by Domagoj Babic; and the commercial optimization tool ILOG CPLEX³. Executables for Saps and Spear are included in the zip file, but CPLEX needs to be purchased to run the CPLEX example.

We include two tuning scenarios for Saps, one for Spear and one for CPLEX. In the Spear example, the instance collection is the same as in one of the Saps examples: the algorithm is optimized on 5 graph colouring problem instances and tested on five other ones. (Note that our examples are toy examples with very small training and test data sets; we recommend substantially larger data sets for real applications!) The second tuning scenario for Saps optimizes Saps performance for a single instance, and tests on the same one, but of course with different seeds – this is useful to study peak performance of an algorithm. For CPLEX, we include a tuning scenario for mixed integer programs from combinatorial auctions (see [5] for details).

Type `bin/paramils` to see the syntax for starting these four examples. (You may run an example if you wish - these toy scenarios should take less than a minute.) For example, the call for the first scenario is `paramils -numRun 0 -scenariofile example_saps/scenario-Saps-SWGCP-sat-small-train-small-test.txt -validN 100`.

4 Tuning scenario files

Note that most information in the above example call is not given on the command line, but hidden in a text file description of the tuning scenario, in this case `example_saps/scenario-Saps-SWGCP-sat-small-train-small-test.txt`. *Tuning scenario files* such as this define a tuning scenario completely, and also contain some information about where ParamILS should write its results etc. They can contain the following information:

¹<http://www.erikveen.dds.nl/rubyscript2exe/index.html>

²http://www.cs.ubc.ca/~babic/index_spear.htm, described in some more detail in [2]

³<http://www.ilog.com/products/cplex/>

algo An algorithm executable or a call to a wrapper script around an algorithm that conforms with the input/output format of ParamILS.

execdir Directory to execute `<algo>` from: “`cd <execdir>; <algo>`”

deterministic Set to 0 for randomized algorithms, 1 for deterministic

run_obj A scalar quantifying how “good” a single algorithm execution is, such as its required runtime. Implemented examples for this include **runtime**, **runlength**, **approx** (approximation quality, i.e., $1 - (\text{optimal quality} - \text{found quality})$), **speedup** (speedup over a reference runtime for this instance – note that for this option the reference needs to be defined in the **instance_seed_file** as covered in Section 6). Additional objectives for single algorithm executions can be defined by modifying function **single_run_objective** in file **algo_specifics.rb**.

overall_obj While **run_obj** defines the objective function for a single algorithm run, **overall_obj** defines how those single objectives are combined to reach a single scalar value to compare two parameter configurations. Implemented examples include **mean**, **median**, **q90** (the 90% quantile), **adj_mean** (a version of the mean accounting for unsuccessful runs: total runtime divided by number of successful runs), **mean1000** (another version of the mean accounting for unsuccessful runs: (total runtime of successful runs + $1000 \times$ runtime of unsuccessful runs) divided by number of runs – this effectively maximizes the number of successful runs, breaking ties by the runtime of successful runs; it is the criterion I use in most of my experiments), and **geomean** (geometric mean, primarily used in combination with **run_obj=speedup**). The empirical statistic of the cost distribution (across multiple instances and seeds) to be minimized, such as the mean (of the single run_objectives).

cutoff_time The time after which a single algorithm execution will be terminated unsuccessfully. This is an important parameter: if chosen too high, lots of time will be wasted with unsuccessful runs. If chosen too low the optimization is biased to perform well on easy instances only.

cutoff_length The runlength after which a single algorithm execution will be terminated unsuccessfully. This length can, e.g. be defined in flips for an SLS algorithm or decisions for a tree search.

tunerTimeout The timeout of the tuner. Validation of the final best found parameter configuration starts after the timeout.

paramfile Specifies the file with the parameters of the algorithms.

outdir Specifies the directory ParamILS should write its results to.

instance_file Specifies the file with a list of training instances.

test_instance_file Specifies the file with a list of test instances.

instance_seed_file Specifies the file with a list of training instance/seed pairs – this and `instance_file` are mutually exclusive.

test_instance_seed_file Specifies the file with a list of training instance/seed pairs – this and `test_instance_file` are mutually exclusive.

ParamILS writes output about its progress to stdout and also writes to files in the specified directory `<outdir>`. The file with `-log` in its name logs ParamILS behaviour, the file with `-traj` only keeps track of its current best solution trajectory, and the `-result` file summarizes the final result. Note that for FocusedILS, the quality of the incumbent solution does not improve monotonically because the number of runs it is based on varies. Solution quality starts at the worst possible value, 1000000000000000000. It then typically quickly improves to a very low value because in the beginning each evaluation is only based on a single algorithm execution (leading to initial over-tuning); as more runs become available the performance estimates get more realistic. Detailed information about the incumbent configurations found is written to the log file.

In order to get reasonable performance estimates, ParamILS performs multiple runs for each configuration. These runs can differ in the input instance and the algorithm seed. For deterministic algorithms, the runs only differ in their input instances (the seed for deterministic algorithms should always be fixed to -1). Different input instances and seeds will lead to different results, or the same final result being found faster or slower. Thus, if multiple CPUs are available for the optimization, we recommend to start several copies of ParamILS, differing only in the parameter `-numRun`. This is especially important for FocusedILS, whose performance depends quite strongly on the order of training instances (and the seeds) used.

5 Configurable parameters

There are a number of configurable parameters the user can set:

maxEvals The number of algorithm executions after which the optimization is terminated.

maxIts The number of ILS iterations after which the optimization is terminated.

approach Use `basic` for BasicILS, `focused` for FocusedILS, and `random` for random search.

N For BasicILS, N is the number of runs to perform to evaluate each parameter configuration. For FocusedILS, it is the *maximal* number of runs to perform to evaluate a parameter configuration.

There is also a number of internal parameters that control the heuristics in ParamILS.

6 Using Instance-Specific information

Whether you choose to provide an `instance_file` (i.e. a list of problem instance filenames), or an `instance_seed_file` (i.e. a list of pairs of problem instance filenames and seeds), you specify one instance per line. You may choose to include additional information after the instance filename, such as the optimal solution quality for the instance, or the instance hardness for one or more other algorithms. Thus, the syntax for each line of the `instance_file` is `<instance_filename> <rest>`, where `<rest>` is an arbitrary (possibly empty) string; when using an the `instance_seed_file`, the syntax is `<seed> <instance_filename> <rest>`. The syntax for these files allows you to do this easily: the `<rest>` string is always parsed and passed on to the objective function computation. The rest may, for example, specify a reference runtime (or runlength, or whatever) for the instance. This is very useful if the objective is to beat a competing algorithm, or a previous version of the same algorithm (In my opinion, this objective is used too much in computer science research, but since the demand is there I provide the option). The single run objective `speedup` is currently the only objective function using this reference value, but you are welcome to implement additional objective functions (using `paramils2.2-source.zip`).

7 Running ParamILS for your own code

In order to employ ParamILS to optimize your own code, you need to provide instance lists in the same format as in the above example, provide a file listing your algorithm's parameters in a predefined format, and match the required input/output format. These two latter points are covered in this section.

7.1 Algorithm parameter file

I recommend you create a separate subdirectory for each algorithm you want to optimize. The parameters of your algorithm need to be defined in a file e.g. called `params.txt`.

This file consists of three parts: basic parameters, specification of conditional parameters, and forbidden parameter combinations, where each of the latter two can be empty. (Examples for such files can be found in the example directories.) In the first part, each line lists one parameter, in curly parentheses the possible values considered, and in square parentheses the default value. In the second part, conditional parameters that are only active when some higher-level parameters take on certain values are specified as follows: `conditional_param — higher_level_param in values for higher_level_param` that allow `conditional_param` to be active, separated by commas. In the third part, forbidden combinations of parameters may be listed. These forbidden combinations are listed one per line, in curly parentheses in the form: `{param1=value1,param2=value2,...}`.

7.2 Algorithm executable / wrapper

The algorithm executable must comply with the following input/output criteria.

It is called as:

```
<algo_executable> <instance_name> <instance-specific information> <cutoff_time>  
<cutoff_length> <seed> <params>
```

and outputs (possibly amongst others) a line

```
Result for ParamILS: <solved>, <runtime>, <runlength>, <best_sol>,  
<seed>
```

containing information about the algorithm execution.

The part `<instance-specific information>` is a string in double quotes containing exactly the `<rest>` string for the current instance that was discussed in Section 6. Thus, for example, when no additional instance information is provided for an instance in the `instance_file/instance_seed_file`, this part is the empty string: `""`.

As for the output, `<solved>` can be either of the strings “SAT”, “UNSAT”, or “TIMEOUT”; `<best_sol>` is the best solution found (for SAT, the lowest number of unsatisfied clauses), and the other fields should be self-explanatory. It is important to output a value for each of these fields, even if they don’t make sense for your algorithm (just output e.g. -1 in that case; for the seed, please output the value that is passed as an input – everything else will lead to an error). If you don’t want to change your algorithm output, you can write a simple wrapper around it; I did this for the SAPS example above. In fact, a wrapper could reuse most parts of that SAPS wrapper. If you want to write a simple ruby wrapper around your algorithm executable, have a look at `saps_wrapper.rb` in directory `example_saps`. Note that this wrapper by no means needs to be written in Ruby – this is just the scripting language I am most comfortable with. In fact, you do not require a Ruby installation at all to run ParamILS when you use the Linux executable discussed above.

References

- [1] F. Hutter. Stochastic local search for solving the most probable explanation problem in Bayesian networks. Master’s thesis, Department of Computer Science, Darmstadt University of Technology, Darmstadt, Germany, September 2004.
- [2] F. Hutter, D. Babić, H. H. Hoos, and A. J. Hu. Boosting verification by automatic tuning of decision procedures. In *Formal Methods in Computer Aided Design (FMCAD’07)*, 2007. To appear.
- [3] F. Hutter, H. H. Hoos, and T. Stützle. Automatic algorithm configuration based on local search. In *Proc. of the Twenty-Second Conference on Artificial Intelligence (AAAI ’07)*, pages 1152–1157, 2007.
- [4] F. Hutter, D. A. D. Tompkins, and H. H. Hoos. Scaling and probabilistic smoothing: Efficient dynamic local search for SAT. In *Proc. of CP-02*, pages 233–248, 2002.
- [5] K. Leyton-Brown, M. Pearson, and Y. Shoham. Towards a universal test suite for combinatorial auction algorithms. In *ACM Conference on Electronic Commerce (EC-00)*, 2000.