SMAC Manual, version 0.90b

Department of Computer Science
University of British Columbia
Vancouver, BC V6T 1Z4, Canada.

May 25, 2012

Contents

1 Introduction

SMAC (Sequential Model-based Algorithm Configuration) (Update reference) is an SMBO based automatic configuration tool for parameter optimization. It works on parameterized algorithms supporting categorical and continuous spaces with conditionality. SMAC searches through a space of possible parameter configurations by taking those which look promising, running them, updating a regression model with new information, selecting a new set of promising configurations and repeating.

Users of SMAC must provide:

- a parametric algorithm $\mathcal{A}$ (executable to be called from the command line),
- all parameters and their possible values Θ, and
- a set of benchmark instances, Π.
- The objective with which to measure and aggregate algorithm preformance results.

SMAC then executes algorithm $\mathcal{A}$ with different combinations of parameters sampled from Θ with instances sampled from Π, searching for the configuration that yields overall best performance across the benchmark instances under the supplied objective. For more details see (To be replaced). If you use SMAC 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.

1.1 History

SMAC originally grew out of improvements to Bartz-Beielstein et al's Sequential Parameter Optimization Toolbox, called SPO+ in 2009 which was fully automated and more robust than the original. In 2010 development began on a time-bounded variant which reduced computational overheads called TB-SPO. In 2011 the first version of SMAC was written in MATLAB while relying on some Java components, and in early 2012 the remainder was ported to Java.

Some of SMAC's idiosyncracies are the result of previous work on a model-free Automatic Configurator called ParamILS (Reference).

2 Configuration & Usage

This quick start guide is for **version 0.90b** of SMAC. It is important to note that this is a preliminary beta release. It has not been thoroughly tested and validated yet. Download the smac-0.90b.zip file, and unzip it in a new directory. Smac is executed by calling *smac* and by default it will merely output the usage options.

2.1 Requirements

SMAC itself requires nothing more than Java 1.6 to run. It has not been tested on Windows machines, and the supplied scripts are only for running under Unix, some of the examples may require other software such as CPLEX, ruby or perl. All required libraries are provided within the download package Some of the examples supplied are compiled for usage under Linux.

2.2 Configuration Files

Configuration Files are a way of saving different default values for frequent use with SMAC without having to specify them on every execution. Currently you cannot override values set in configuration files via the command line, though this will hopefully change. Additionally there are several configuration files that can be specified for different purposes. The general format for a configuration file is the name of the configuration option (without the two dashes) an equal sign and then the value (for booleans it should be true or false, lowercase).

2.2.1 Tuning Scenario Files

Perhaps the most useful of the configuration files that can be specified are *Tuning scenario files* which contain all the information about running a target algorithm.They are backwards compatible with previous versions of MATLAB SMAC and ParamILS (to facilitate this some of the options names may be slightly different than in section (INSERT SECTION), rest assured these all are aliases of commands specified via the command line.

They can contain the following information:

algo An algorithm executable or a call to a wrapper script around an algorithm that conforms with the input/output format specified in section (INSERT SECTION).

execdir Directory to execute `<algo>` from: (e.g. "cd `<execdir>`; `<algo>`")

deterministic Set to 0 for randomized algorithms, 1 for deterministic .

run_obj Determines how to convert the resulting output line into a scalar quantifying how "good" a single algorithm execution is, (e.g. how long it took to execute, how good of a solution it found, etc...). Implemented examples for this are as follows [1]:

Name	Description
RUNTIME	**The reported runtime of the algorithm.**
RUNLENGTH	**The reported runlength of the algorithm.**
APPROX	$1 - \frac{\textbf{optimal quality stored in instance specific information}}{\textbf{reported quality}}$
SPEEDUP	$\frac{\textbf{runtime stored in instance specific information}}{\textbf{reported runtime}}$
QUALITY	**The reported quality of the algorithm.**

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 for this are as follows[2]:

Name	Description
MEAN	**The mean of the values**
MEDIAN	**The median of the values**
Q90	**The 90th percentile of the values**
ADJ_MEAN	$\frac{\textbf{total run objective}}{\# of\, successful\, runs}$
MEAN1000	**Unsuccessful runs are counted as 1000 $\times$ cutoff_time**
MEAN10	**Unsuccessful runs are counted as 10 $\times$ cutoff_time**
GEOMEAN	**The geometric mean (primarily used in combination with run_obj = SPEEDUP).**

cutoff_time The time after which a single algorithm execution will be terminated unsuccesfully. This is an important parameter: if choosen 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 unsuccesfully. The actual semantic meaning of this value is up to the target algorithm. (NOT SUPPORTED IN THIS VERSION)

tunerTimeout The amount of CPU time allowed for target algorithm execution (specifically once the sum of all runtimes exceeds this value, SMAC will stop configuring).

paramfile Specifies the file with the parameters of the algorithms. The format of this file is covered in...

[1] APPROX, SPEEDUP rely on instance specific information which is not currently implemented

[2] ADJ_MEAN Not implemented currently

outdir Specifies the directory ParamILS should write its results to.

instance_file Specifies the file with a list of training instances. Additionally this parameter can be specified as **instance_seed_file**. The format of these files is covered in...

test_instance_file Specifies the file with a list of test instances. Additionally this parameter can be specified as **test_instance_seed_file**. The format of these files is covered in...

2.2.2 Instance File Format

ParamILS and the MATLAB version of SMAC required specifying the type of file as different mutually exclusive options (e.g. **instance_file** versus **instance_seed_file**). This version detects the format automatically. These files are CSV files with **‘‘** as a cell delimiter, and **,** as the cell seperator. If the file has 1 column, then it is an instance file, and the seeds will be randomly generated. If the file has 2 columns, then the first column must be a number and this represents a seed to be used with this instance. The second column is the filename of the instance. In the first version, each file name can be specified only once. In the second version, the instances are specified on multiple lines with duplicate entries resulting in those seeds being available for that instance. Additionally these files are also parsed with “ “ (space) as a cell seperator (and no cell delimiter), this is for backwards compatibility with ParamILS, note this doesn't work with spaces in filenames.

2.3 Configuration Options Reference

–**algoExec** algorithm call to execute

–**execDir** Execution Directory for Algorithm

–**runObj** Per Target Algorithm Run Objective Type that we are optimizing for

–**overallObj** Aggregate over all Run's Objective Type that we are optimizing for

–**cutoffTime** Cap Time for an Individual Run

–**cutoffLength** Cap Time for an Individual Run

–**tunerTimeout** Total CPU Time to execute for

–**instanceFile** File containing instances in either "<instance filename>", or "<seed>,<instance filename>" format

–**instanceFeatureFile** File that contains the all the instances features

–**testInstanceFile** File containing instances specified one instance per line

–**scenarioFile** Scenario File

–**deterministic** Whether the target algorithm is deterministic

–**skipInstanceFileCheck** Do not check if instances files exist on disk

–**outputDirectory** Output Directory

–**seed** Seed for Random Number Generator [0 means don't use a seed]

–**experimentDir** Root Directory for Experiments Folder

–**paramFile** File containing Parameter Space of Execution

–**numIterations** Total number of iterations to perform

–**runtimeLimit** Total Wall clock time to execute for

–**totalNumRunLimit** Total number of target algorithm runs to execute

–**numTestInstances** Number of instances to test against (Will execute min of this, and number of instances in test Instance File)

–**runHashCodeFile** File containing a list of Run Hashes one per line (Either with just the format on each line, or with the following text per line: "Run Hash Codes: (Hash Code) After (n) runs". The number of runs in this file need not match the number of runs that we execute, this file only ensures that the sequences never diverge. Note the n is completely ignored so the order they are specified in is the order we expect the hash codes in this version

–**modelHashCodeFile** File containing a list of Model Hashes one per line with the following text per line: "Preprocessed Forest Built With Hash Code: (n)" or "Random Forest Built with Hash Code: (n)" where (n) is the hashcode

–**runID** String that identifies this run for logging purposes

–**numPCA** Number of prinicipal components of features

–**expectedImprovementFunction** Expected Improvement Function to Use

–**nuberOfChallengers** Number of Challengers needed for Local Search

–**numberOfRandomConfigsInEI** Number of Random Configurations to evaluate in EI Search

–**stateSerializer** Controls how the state will be saved to disk

–**stateDeserializer** Controls how the state will be saved to disk

–**restoreStateFrom** The Location (State Deserializer Dependent) of States

–**restoreIteration** The Iteration to Restore

–**executionMode** Mode of Automatic Configurator to run

–**splitMin** Minimum number of elements needed to split a node

–**fullTreeBootstrap** Bootstrap all data points into trees

–**storeDataInLeaves** Store full data in leaves of trees

–**logModel** Store data in Log Normal form

–**nTrees** Number of Trees in Random Forest

–**minVariance** Minimum allowed variance

–**ratioFeatures** Number of features to consider when building Regression Forest

–**preprocessMarginal** Build Random Forest with Preprocessed Marginal

–**adaptiveCapping** Enable Adaptive Capping

–**capSlack** Amount to scale computed cap time of challengers by

–**capAddSlack** Amount to increase computed cap time of challengers by [general formula: capTime = capSlack*computedCapTime + capAddSlack]

–**imputationIterations** Amount of times to impute censored data

–**maxConcurrentAlgoExecs** Maximum number of concurrent target algorithm executions

2.4 Usage

To get started with an existing configuration scenerio you simply need to execute smac as follows:

```
./smac --scenarioFile <file>
```

This will execute SMAC with the default options on the scenario specified in the file. Several of the options above can have drastic changes on the performance and usage of smac as follows:

2.4.1 ROAR Mode

```
./smac --scenarioFile <file> --executionMode ROAR
```

This will execute SMAC but will execute it in ROAR mode (See some paper), in short this is a model free based search where a random configuration is raced against the incumbent.

2.4.2 Adaptive Capping

```
./smac --scenarioFile <file> --adaptiveCapping
```

Adaptive Capping (See some paper) will cause SMAC to only schedule algorithm runs for as long as is needed to determine whether or not an algorithm is a promising configuration, as opposed to the default runtime –**cutoffTime**. This can drastically improve tuning performance with limited time budgets. Related configuration options for this are –**capSlack**, –**capAddSlack**, –**imputationIterations**. Note: Adaptive Capping should only be used when the –**runObj** is RUNTIME.

2.4.3 Run Hash Codes

```
./smac --scenarioFile <file> --runHashCodeFile <logfile>
```

A Run Hash Code is a sequence of hashes that represent which runs were scheduled by SMAC. SMAC logs all Run Hash Codes to the log file, and this option allows reading of that log file for subsequent runs to ensure that the exact same set of runs is scheduled. This is primarily of use for developers.

2.4.4 Wall Clock Limit

```
./smac --scenarioFile <file> --runtimeLimit <seconds>
```

SMAC will abort if this amount of wall-clock time expires. This does not over ride the –**tunerTimeout** option and it's basically whichever comes first. This is useful to limit the amount of unaccounted for overhead in an algorithm run.

2.4.5 State Restoration

```
./smac --scenarioFile <file> --restoreStateFrom <dir> --restoreIteration <iteration>
```

SMAC will read the files in the specified directory and will change it's state such that it had previously executed all the runs specified up to the iteration specified and continue executing. Provided other options such as –**seed** and other arguments are identical you should get the exact same result out provided the target algorithm runtime variance on individual instance & seed pairs is small. This can also be used to restore runs from MATLAB (although due to the lossy nature of MATLAB files, and differences in random calls you will not get the same resulting trajectory). By default the possible states that are able to be restored are the start of iteration numbers of the form 2^n $n \in \mathbb{Z}_{\geq 0}$ as well as the previous 2 iterations prior to SMAC completing. If the run crashed additional information is saved, but in general you cannot restore to that iteration (since the crash most likely occured in the middle of an iteration).

Note: When you restore a SMAC state, you are in essence preloading a bunch of runs and then running the scenario. In certain cases if the scenario has changed, this may result in undefined behaivor. Changing something like

–**tunerTimeout** is usually a safe bet, however changing the –**runObj** may not be, as the incumbent may not actually be the best configuration under this new objective.

2.4.6 Checking Available Iterations To Restore

To check the available iterations that can be restored from a saved directory use:

```
./smac-possible-restores <dir>
```

2.4.7 Turning Off State Saving

```
./smac --scenarioFile <file> --stateSerializer NULL
```

SMAC will not save any state information to disk.

2.4.8 Concurrent Algorithm Execution Requests

```
./smac --scenarioFile <file> --maxConcurrentAlgoExecs <num>
```

In certain circumstances it may be much faster to allow more than one algorithm execution at once. Such as when multiple cores are available or when actual algorithm execution is I/O bound (which can happen in certain circumstances). As a result you can have SMAC schedule multiple runs at a time. In general this will result in the same trajectory as when set to 1, with the exception of when –**adaptiveCapping** is set, in this case concurrent runs are scheduled with cutoff times as if each were the first of the runs to be scheduled.

2.4.9 Named Runs

```
./smac --scenarioFile <file> --runID <someID>
```

All output is written to the –**outputDirectory** / –**runID**. This runID defaults to “Run-(Current Time)” a more meaningful name can be set here.

3 Example tuning scenarios

The zip file contains examples for three algorithms: SAPS [**?**], a local search algorithm for SAT; Spear[3], a tree search algorithm for SAT (and satisfiability modulo theories) developed by Domagoj Babic; and the commercial optimization tool ILOG CPLEX [4]. Executables for Saps and Spear are included in the zip file, but CPLEX needs to be purchased to run the CPLEX example.

[3] http://www.cs.ubc.ca/~babic/index_spear.htm, described in some more detail in [**?**]
[4] http://www.ilog.com/products/cplex/

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 five 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 [**?**] for details).

4 Using Instance-Specific information

Instance Specific Information is not supported in this version

5 Running SMAC for your own code

In order to employ SMAC 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.

5.1 Algorithm parameter file

It is 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. `algo-params.txt`. (Examples for such files can be found in the example directories). Comments in the file begin with a #, and run to the end of the line.

The file consists of two types of statements parameter declarations, and conditional declarations:

Parameter Declarations

SMAC supports two types of parameters, continuous and categorical parameters declared as follows:

```
name { value1, ..., value_n } [defaultValue]
Example:
timeout { 1,2,4,8,16,32 } [8]
```

Here a categorical parameter is declared named timeout, it's values are one of the values listed and and the default value is 8.

```
name [minValue, maxValue] [defaultValue](i)(l)
Example 1:
```

```
timeout [1,32] [8]
Example 2:
timeout [1,32] [8]l
Example 3:
timeout [1,32] [8]i
Example 4:
timeout [1,32] [8]il
```

Example 1 We have specified timeout as continuous with a default value of 4. Any value is legally permitted so long as it's in the interval of [1,32]. When drawing a random configuration out of this space they are drawn uniformly.

Example 2 This is the same as Example 1, except that when drawing random configurations we do so on a log scale (e.g. a value between [1,8] is as likely to be selected as between [8,32]).

Example 3 In this example the only legal values are integers in the range [1,32], we continue to select from the integers uniformly.

Example 4 In this example integers in the range [1,32] are the only values permitted, and when randomly selecting them we do so on a log scale.

Conditional Parameters

Conditional Parameters have the following syntax:

```
dependentName | independentName operation { value1, ... , value_n}
Example Param File:
sort-algo { quick, insertion, merge, heap, stooge, bogo } [ bogo ]
quick-revert-to-insertion { 1,2,4,8,16,32 } [16]
quick-revert-to-insertion | sort-algo in { quick }
```

Currently the only supported operation is *in* , and the values are restricted to being specified as categoricals. In the above example the quick-revert-to-insertion is conditional on the sort-algo parameter being set to quick, and will be ignored otherwise.

5.2 Algorithm executable / wrapper

The target algorithm as specified by the –**algoExec** parameter must obey the following general contracts. While writing your own code to achieve this is one option, there are two other common methods outlined in... (Another section)

5.2.1 Invocation

The algorithm must be invocable using the following:

```
<algo_executable> <instance_name> <instance_specific_information>
<cutoff_time> <cutoff_length> <seed> <param> <param> <param>...
```

algo_executable Exactly what is specified in the −**algoExec** argument to SMAC.

instance_name The name of the problem instance we are executing against

instance_specific_information An arbitrary string associated with this instance as specified in the −**instance_file** if no information is present then a 0 is passed here.

cutoff_time The amount of time in seconds that the target algorithm is permitted to run. It is the responsibility of the target algorithm to ensure that this is obeyed. It is not necessary that that the actual algorithm execution time (wall clock time) be below this value. If for instance clean up is needed, or it is only possible for the algorithm to abort at certain stages.

cutoff_length A domain (target algorithm) specific measure of when the algorithm should consider itself done.

seed A positive integer that the algorithm should use to seed itself (for reproducability), -1 is used when the algorithm is −**deterministic.**

param A setting of an active parameter for the selected configuration as specified in the Algorithm Parameter File. SMAC will only pass parameters that are considered active, and SMAC is not guaranteed to pass the parameters in any particular order. The exact format for each parameter is:

```
-name 'value'
```

All the parameters above are mandatory, even if they are inapplicable, in which case a dummy value will be passed which you are free to ignore.

5.2.2 Output

The Target Algorithm is free to output anything, which will be ignored but must at some point output a line (only once) in the following format[5]:

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

[5] ParamILS in not a typo. While other values are possible including SMAC, HAL. ParamILS is probably the most portable. The Exactly Regex that is used in this version is: **(Final)?\s*[Rr]esult\s+(?:(for)|(of))\s+(?:(HAL)|(ParamILS)|(SMAC)|(this wrapper))**

solved Must be one of **SAT** (signifying a successful run with a result of YES), **UNSAT** (signifying a successful run with a result of NO), **TIMEOUT** if the algorithm didn't complete, **CRASHED** if something untoward happened during the algorithm run. SMAC does not differentiate between **SAT** and **UNSAT** responses, and the primary use of these is to serve as a check that the algorithm is executing correctly by allowing the algorithm to output a yes / no output that can be checked. This can be useful for debugging purposes.

runtime The amount of CPU time used during this algorithm run. SMAC does not measure the CPU time directly, and this is the amount that is used with respect to –**tunerTimeout**. You may get unexpected performance degredation when this amount is heavily under reported (this typically happens when targetting very short algorithm runs with large overheads that aren't accounted for).
NOTE: The **runtime** should always be strictly less than the requested **cutoff_time** when reporting **SAT** or **UNSAT**. It should always be the **cutoff_time** when reporting **TIMEOUT**. This is even the case if the algorithm can determine that it will **TIMEOUT** without doing the actual work.

runlength A domain specific measure of how far the algorithm progressed.

quality A domain specific measure of the quality of the solution.

seed The seed value that was used in this target algorithm execution. Note: This seed **MUST** match the seed that the algorithm was called with. This is used as fail-safe check to ensure that the output we are parsing really matches the call we requested.

Like invocation, all fields are mandatory, when not applicable 0's can be substituted.

5.2.3 Wrappers & Native Libraries

Beyond simplying adding the additional functionality to your code by hand, there exist several other approaches from related projects.

Wrappers Executable Scripts that manage the resource limits automatically and format the specified string into something usable by the actual target algorithm. This approach is probably the most common, but among it's draw backs are the fact that they often rely on third party languages, and for smaller execution times have a large amount of overhead that may not be accounted for. Most of the examples included in SMAC use this approach, and the wrappers included can be adapted for your own project

Native Libraries Libraries exist (See: ...) for C and Java currently that facilitate adding the functionality directly to the code, while parsing the arguments into the necessary data structures is still required, they do ensure that the output is written properly in most cases. Draw backs include not necessarily outputting the values in certain crashes.

6 Interpreting SMAC's Output

SMAC outputs a variety of information to log files, trajectory files, and state files. Most of the files are human readable, and this section governs where to find information.

6.1 Logs

SMAC's output, especially in this release is particularly verbose, and perhaps noisey. By default SMAC uses slf4j for logging, and specifically ships with logback. In theory this allows a much richer configuration option by modifying `conf/logback.xml` . To change the output level (which supports TRACE, DEBUG, INFO, WARN, ERROR, edit this file. Future versions will include more simple flexibilty, but the default configuration outputs three files into the **–outputDirectory**.

log.txt A formatted listing of the output of SMAC with information useful for debugging and tracing.

raw.txt The raw messages logged by the program, this is useful for grepping.

runhashes.txt A file that contains the Run Hash Codes generated by this run (not implemented currently, and raw.txt can be used instead).

By default the standard output of the SMAC is in between log.txt and raw.txt. For the most part the same lines are logged however. ***Some*** of the output from the log file was written to be identical to the MATLAB version, but by and large this should not be relied upon as neither the MATLAB version nor this version have any well defined guarantees on what is or isn't outputted, or how. Any mechanisms relying on this are more than likely very brittle, especially as changes to the logback.xml file may result in changes to the formatting of this file.

6.1.1 Interpretting the Log File

SMAC execution basically consists of two stages, setup and then the actual algorithm. In the setup process alot of validation is done to hopefully catch alot of errors. Any warnings that are outputted in the first part of SMAC's run may deserve special attention, and can in some cases be turned off if they are potentially false positives. SMAC also outputs it's configuration for the run at the start of every log file. Eventually SMAC will start running, and this occurs when you see something along the lines of:

```
Automatic Configuration Start Time is May 23, 2012
```

At this point SMAC has started the actual algorithm configuration as outlined in (Insert REF). The two most important pieces of information for the general user, are probably lines like the following:

```
At end of iteration 2, incumbent is 25433[4.0, 2.0, 1.0, 7.0] -alpha '1.189' -ps '0.033
*****Runtime Statistics*****
Iteration: 2
Wallclock Time: 8.317 s
Wallclock Time Remaining:2.147483638683E9 s
Total CPU Time: 7.679999999999996 s
CPU Time Remaining: 22.320000000000004 s
AC CPU Time: 0.75 s
AC User Time: 0.72 s
Max Memory: 5365.375 MB
Total Java Memory: 361.5 MB
Free Java Memory: 318.36956787109375 MB
```

The first line (after the closing]) advises you what the incumbent was at this point of time. The next lines give you the state of SMAC. Most numbers are straight forward except the AC CPU Time: and AC User Time: These are the CPU and User times of SMAC itself. CPU Time remaining is the the –**tunerTimeout** - The Total CPU Time So far.

6.1.2 State Files

The state files stored in the state/ subdirectory of the –**outputDirectory** are a bit more well defined. The txt and csv files are human readable. The **paramstrings-itN.txt** file correspond to a listing of parameter strings. The number in the front corresponds to the numbers used in the **runs_and_results-itN.csv** file

The columns of this file in order represent:

1. The number of the row (run)
2. The number of the configuration in the paramstrings-itN.txt file
3. The number of the instance as specified in –**instanceFile**
4. The response value (–**runObjective**) of the run
5. 1 if the run was censored, 0 if not.
6. The cutoff time of the run
7. The resulting seed of the run.
8. The runtime of the run.
9. The runlength of the run

10. 1 if the run was a successful run (SAT or UNSAT), 0 otherwise.
11. The quality of the run.
12. The iteration the run occured in.
13. The cumulative sum of all the runs up to this point

6.1.3 Trajectory File

SMAC also outputs a trajectory file, which is a CSV file that outpus the following columns

Column Name	Description
Total Time	Sum of all execution times and CPU time of SMA
Incumbents Mean Performance	Performance of the Incumbent under the given –**runObjective** and
Incumbent's Performance σ	Standard deviation of the Incumbent under the given –**runObjective**
Incumbent ID	The ID of the incumbent as listed in the paramstrings file whe
acTime	Total CPU Time of SMAC
Param String Columns	The rest give the parameter strings of the incumbe

7 Usage Differences Between This and Other Versions

This section is primarily for users of ParamILS, but also the MATLAB version of SMAC and outlines some of the differences.

7.1 Configuration Changes

Everything needed to execute SMAC is specified on the Command Line. Certain configuration options allow you to specify files, for instance a scenario file. While most existing scenario files should work without a hitch it's important to note that what is actually happening is that it is just being parsed into Command Line options. You could in theory specify other configuration options in a scenario file, and they would be set accordingly.

7.2 Previous Features Not Supported

Forbidden Parameters

8 Known problems

1. You cannot override options on the command line specified in files
2. Validation of configuration options is poor and limited
3. Instance Specific Information is not supported.

4. Neighbourhood search of discretized continuous parameters may select identical parameter configurations.
5. Cutoff Length is not supported
6. Standard Deviation not reported in Trajectory Files
7. –**tunerTimeout** is computed simply on target algorithm runtimes, it may not agree with ParamILS nor MATLAB SMAC's definition

References

document