0.1 Option Files

Option Files are a way of saving a different set of values frequently used with SMAC without having to specify them on every execution. The general format for an option 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). Currently options that take multiple arguments are not supported. Additionally you can not use aliases that are single dashed (*e.g.* to override the Experiment Directory, you must use **–experimentDir** and not **-e**)

When using Option Files it is important that no two files (including the Scenario File), specify the same option, the resulting configuration is undefined, and in general this will not throw an error.

0.1.1 Scenario File

The Scenario Option File, or Scenario File, is the recommended way of configuring SMAC [1]. The Scenario Files used in SMAC are backwards compatible with ParamILS and the name of option names here reflect that[2]. NOTE: **cutoff_length** is not currently supported.

algo An algorithm executable or wrapper script around an algorithm that conforms with the input/output format specified in section 0.5. The string here should be invokable via the system shell.

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

deterministic A boolean that governs whether or not the algorithm should be treated as deterministic. For backwards compatibility with ParamILS, this option also supports using 0 for false, and 1 for true. SMAC will never invoke the target algorithm more than once for any given instance, seed and configuration. If this is set to `true`, SMAC will never invoke the target algorithm more than once for any given instance and configuration.

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...). Currently implemented objectives are the following:

Name	Description
RUNTIME	**The reported runtime of the algorithm.**
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:

Name	Description
MEAN	**The mean of the values**
MEAN1000	**Unsuccessful runs are counted as 1000 $\times$ cutoff_time**
MEAN10	**Unsuccessful runs are counted as 10 $\times$ cutoff_time**

[1] Nothing in general prevents you from specifying non-scenario options in these files, but in general you should restrict your files to these.

[2] Every option name listed here is in fact an alias for an existing option listed in the section **??** and it is entirely possible to use SMAC without using Scenario Files.

cutoff_time The CPU time after which a single algorithm execution will be terminated as unsuccess (and treated as a **TIMEOUT**). 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.

tunerTimeout The limit of the CPU time allowed for configuration (*i.e.*The sum of all algorithm runtimes, and by default the sum of the CPU time of SMAC itself).

paramfile Specifies the file with the parameters of the algorithm. The format of this file is covered in Section 0.4.

outdir Specifies the directory SMAC should write its results to.

instance_file Specifies the file containing the list of problem instances (and possibly seeds) for SMAC to use during the *Automatic Configuration Phase*. The ParamILS parameter **instance_seed_file** aliases this one and the format is auto-detected. The format of these files is covered in section 0.2.

test_instance_file Specifies the file containing the list of problem instances (and possibly seeds) for SMAC to use during *Validation Phase*. The ParamILS parameter **test_instance_seed_file** aliases this one and the format is auto-detected. The format of these files is covered in section 0.2.

feature_file Specifies the a file with the features for the instances in the **instance_file** and possibly the **test_instance_file** [3]. The format of this file is covered in section 0.3.

0.2 Instance File Format

The files used by the **instance_file** & **test_instance_file** options come in four potential formats, all of which are CSV based[4]. Before specifying the formats it is important to note the three kinds of information that are specified with instances [5].

Instance Name The name of the instance that was selected. This should be meaningful to the target algorithm we are configuring [6].

Instance Specific Information A free form text string (with no spaces or line breaks) that will be passed to the Target Algorithm whenever executed.

Seed A specific seed to use when executing the target algorithm.

The possible formats are as follows, and depend on what information you'd like to specify.

1. Each line specifies only a unique **Instance Name**. No **Instance Specific Information** will be used, and **Seed**'s will be automatically generated.

[3]The Validator will load features into memory for test instances if they exist.

[4]Specifically each cell should be double-quoted (*i.e.*"), and use a comma as a cell delimiter. SMAC also supports the old method of reading files that use space as a cell delimiter and do not enclose values. However these files cannot handle **Instance Name**'s that contain spaces.

[5]Features which are required for SMAC but not ParamILS are specified in a seperate file see section 0.3.

[6]Generally **Instance Names** reference specific files on disk.

2. Each line specifies a **Seed** followed by the **Instance Name**. Every line must be unique, but for each **Instance Name** additional seeds will be used in order, when that instance is selected.

3. Each line specifies a **Instance Name** followed by the **Instance Specific Information**. Every **Instance Name** must be unique, **Seed**'s will be automatically generated.

4. Each line specifies a **Seed** followed by the **Instance Name** followed by the **Instance Specific Information**. Every line must be unique, and furthermore, for all **Instance Name**'s the **Instance Specific Information** must be the same for all **Seed** values (*i.e.*You cannot specify different instance specific information that is a function of the seed used).

0.3 Feature File Format

The **feature_file** specifies features that are to be used for instances. Feature Files are specified in CSV format, the first column of every row should list an **Instance Name** as it appears in the **instance_file**. The subsequent columns should list double values specifying a computed continuous feature. By convention the value -512, and -1024 are used to signify that a feature value is missing or not applicable. All instances must have the same number of features.

At the top of the file there MUST appear a header row, the cell that appears above the instance names is unimportant, but for each feature a unique and *non-numeric* feature name must be specified.

0.4 Algorithm Parameter File

The parameter configuration space of your algorithm need to be defined in a file that is specified by the **paramfile** option. Comments in the file begin with a #, and run to the end of the line.

The file consists of three types of statements:

Parameter Declaration Clauses specifies the name of parameters, and their domains.

Conditional Parameter Clauses specify when a parameter is active.

Forbidden Parameter Clauses specify when a combination of parameter settings is illegal and shouldn't be ignored.

0.4.1 Parameter Declaration Clauses

SMAC supports two types of parameters, categorical and numeric. The former is specified as follows:

```
name { value1, ..., value_n } [defaultValue]
```

Example:

```
timeout { 1,5,10,50,100,500,1000,5000,10000 } [500]
```

Here a categorical parameter is declared named `timeout`, its values must be one of the values listed, and it has a default of 500.

Numeric Parameters (both continuous and integral) are specified as follows:

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

Example 1:

```
timeout [1, 10000] [500]
```

We have specified timeout as numeric with a default value of 500. Any value is legally permitted so long as it's in the Real interval of [1, 10000]. When drawing a random configuration out of this space they are drawn uniformly.

Example 2:

```
timeout [1, 10000] [500]l
```

This example is identical to the previous, except that when drawing random configurations we do so uniformly on a $\log_{10}$ scale (*e.g.* a value between [1, 100] is as likely to be selected as between [100, 10000]).

Example 3:

```
timeout [1, 10000] [500]i
```

In this example the only legal values are integers in the range [1, 10000], we select from these integers uniformly.

Example 4:

```
timeout [1, 10000] [500]il
```

In this example integers in the range [1, 10000] are the only values permitted, and when randomly selecting them we do so on a $\log_{10}$ scale.

Restrictions

Integer Numeric integral parameters must have all values specified as integers, even though strictly speaking the notation should permit fractional values. Additionally the default value must be a integer.

Log Log parameters must have strictly positive lower and upper bounds.

0.4.2 Conditional Parameter Clause

Conditional parameter clauses specify when a parameter is active. A parameter is active when for each clause that lists it as a dependent, the independent variable is active and has a value that satisfies the operation [7]. Conditional Parameter Clauses have the following syntax:

```
dependentName | independentName operation { value1, ... , value_n}
```

Example:

```
sort-algo { quick, insertion, merge, heap, stooge, bogo } [ bogo ]
quick-revert-to-insertion { 1,2,4,8,16,32,64 } [16]
quick-revert-to-insertion | sort-algo in { quick }
```

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.

0.4.3 Forbidden Parameter Clauses

Forbidden Parameters are combinations of parameter settings which should not be treated as valid by SMAC. During the search phase, parameters matching a forbidden parameter configuration, will not be explored [8].

The Syntax is as follows:

```
{ name1=val1 , name2=val2, ...}
```

[7] The only supported operation presently is `in`.

[8] Specifying a large number of forbidden parameters may degrade SMAC's performance substantially.

Example

```
quick-sort { on, off } [on]
bubble-sort { on, off } [off]
{ quick-sort=on, bubble-sort=on }
{ quick-sort=off, bubble-sort=off }
```

The above example implements an exclusive-or [9]. The first forbidden parameter clause prevents both sort techniques from being on at the same time. The second ensures that atleast one of them is on. NOTE: The default parameter setting cannot itself be a forbidden parameter setting.

0.5 Algorithm executable / wrapper

The target algorithm as specified by the **algo** parameter must obey the following general contracts. While modifying your own code to directly achieve this is one option there are other methods outlined in section 0.5.3.

0.5.1 Invocation

The algorithm must be invokable via the system command-line using the following command with arguments:

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

algo_executable Exactly what is specified in the **algo** argument in the scenario file.

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 always passed here.

cutoff_time The amount of time in seconds that the target algorithm is permitted to run. It is the responsibility of the callee to ensure that this is obeyed. It is not necessary that that the actual algorithm execution time (wall clock time) be below this value (*e.g.*If the algorithm needs to cleanup, or it's only possible to terminate the algorithm at certain stages).

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

seed A positive integer that the algorithm should use to seed itself (for reproducibility). "-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 active. Additionally SMAC is not guaranteed to pass the parameters in any particular order. The exact format for each parameter is:

`-name 'value'`

All of the arguments above will always be passed, even if they are inapplicable, in which case a dummy value will be passed.

[9] Admittedly it could be better modelled with a simple categorical parameter.

0.5.2 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[10]

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

solved Must be one of **SAT** (signifying a successful run that was satisfiable), **UNSAT** (signifying a successful run that was unsatisfiable), **TIMEOUT** if the algorithm didn't finish within the allotted time, **CRASHED** if something untoward happened during the algorithm run, or **ABORT** if something prevents the target algorithm for successfully executing and it is believed that further attempts would be futile.

SMAC does not differentiate between **SAT** and **UNSAT** responses, and the primary use of these is historical and serves as a check that the algorithm is executing correctly by outputting whether the instance in question is satisfiable or not.

SMAC also supports reporting **SATISFIABLE** for **SAT** and **UNSATISFIABLE** for **UNSAT**. NOTE: These are only aliases and SMAC will not preserve which alias was used in the log or state files.

ABORT can be useful in cases where the target algorithm cannot find required files, or a permission problem prevents access to them. This will cause SMAC to stop running immediately. Use this option with care, it should only be reported when the algorithm knows for CERTAIN that subsequent results may fail. For things like sporadic network failures, and other cosmic-ray induced failures, one should consider using **CRASHED** in combination with the **--retryTargetAlgorithmRunCount** and **–abortOnCrash** options, to mitigate these.

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 [11].

NOTE: The **runtime** should always be strictly less than the requested **cutoff_time** when reporting **SAT** or **UNSAT**.

If an algorithm reports **TIMEOUT** or **CRASHED** the algorithm can report the actual CPU time used, and SMAC will treat it correctly as a timeout for optimization purposes, but count the actual time for $--$**tunerTimeout** purposes.

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.

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

[11] This typically happens when targeting very short algorithm runs with large overheads that aren't accounted for.

additional rundata A string that will be associated with the run as far as SMAC is concerned. This string will be saved in run and results file (Section **??**).

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

0.5.3 Wrappers & Native Libraries

In order to optimize an algorithm, SMAC needs a method of invoking it. While modifying the code to manage the timing and input mechanisms manually is possible, this can sometimes be invasive and difficult to manage. There exist three other methods that one could consider using.

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 its drawbacks are the fact that they often rely on third party scripting languages, and for smaller execution times have a large amount of overhead that may not be accounted for as far as the **tunerTimeout** limit is concerned. Most of the examples included in SMAC use this approach, and the wrappers included can be adapted for your own projects.

NOTE: When writing wrappers it's important not to poll the output stream of the target algorithm, especially if there is lots of output. Doing so often results in lock-contention and significantly modifies the runtime performance of the algorithm enough that the resulting configuration is not well tuned to the real algorithm performance.

Native Libraries Augmentation Libraries exist (See: **TBD**) for C and Java currently that facilitate adding the required functionality directly to the code. While parsing the arguments into the necessary data structures is still required, they do manage the timing and output requirements in most cases. Unlike the previous approach however, certain crashes may not allow the the values to be outputted (*e.g.* a segmentation fault occurs).

Target Algorithm Evaluators This is probably the most powerful, but also the most complicated approach. SMAC is architected in a way that makes it fairly simple to replace the mechanism for execution with something completely custom. This can be done without even recompiling SMAC by creating a new implementation of the `TargetAlgorithmEvalutor` interface, which is responsible for converting run requests (`RunConfig` objects) into run results (`AlgorithmRun` objects). Both the input and output objects are simple *Value Objects* so the coupling between SMAC and the rest of your code is almost zero with this approach. For more information see **??**