ParamILS: Iterated Local Search in Parameter Configuration Space

---

Jiří Kubalík
Department of Cybernetics, CTU Prague

Substantial part of this material is based on the paper
Frank Hutter, Holger H. Hoos, Kevin Leyton-Brown, and Thomas Stützle: ParamILS: An Automatic Algorithm Configuration Framework,
Journal of Artificial Intelligence Research (JAIR),
volume 36, pp. 267-306, October 2009.
See `http://www.cs.ubc.ca/ hutter/papers/Hutter09PhD.pdf`

`http://cw.felk.cvut.cz/doku.php/courses/a0m33eoa/start`

Algorithm Configuration: Motivation

Often, finding performance-optimizing **parameter configurations of heuristic algorithms requires considerable effort**. In many cases, this task is performed **manually in an ad-hoc way**.

Automating this task is of high practical relevance in several contexts:

- **Development of complex algorithms** - setting the parameters of a heuristic algorithm is a highly labour-intensive task, and indeed can consume a large fraction of overall development time. The use of automated algorithm configuration methods can lead to significant time savings and potentially achieve better results than manual, ad-hoc methods.
- **Empirical studies, evaluations, and comparisons of algorithms** - a central question in comparing heuristic algorithms is whether one algorithm outperforms another because it is fundamentally superior, or because its developers more successfully optimized its parameters. Automatic algorithm configuration methods can mitigate this problem of unfair comparisons and thus facilitate more meaningful comparative studies.
- **Practical use of algorithms** - the ability of complex heuristic algorithms to solve large and hard problem instances often depends critically on the use of suitable parameter settings. End users often have little or no knowledge about the impact of an algorithm's parameter settings on its performance, and thus simply use default settings. Automatic algorithm configuration methods can be used to improve performance in a principled and convenient way.

Algorithm Configuration (Parameter Tuning) Problem

Let $\mathcal{A}$ denote an algorithm, whose parameters are to be optimized for a probability distribution of problem instances, $\mathcal{D}$.

$\mathcal{D}$ may be given

- implicitly, as through a random instance generator or a distribution over such generators, or
- as the uniform distribution over a finite sample of problem instances.

With each of the algorithm parameters, $p_1 \ldots p_k$, a domain of possible values is associated and the parameter configuration space, Θ, is the cross-product of these domains (or a subset thereof).

- We assume that all parameter domains are finite sets.

  This assumption can be met by discretizing all numerical parameters to a finite number of values.
- While parameters may be ordered, we do not exploit such ordering relations $\Longrightarrow$ all parameters are finite and categorical.

The elements of Θ are called parameter configurations, θ_i, and $\mathcal{A}(\theta)$ denotes algorithm $\mathcal{A}$ with parameter configuration $\theta \in \Theta$.

Algorithm Configuration (Parameter Tuning) Problem

The objective of the parameter configuration (parameter tuning) problem is to find the parameter configuration $\theta \in \Theta$ resulting in the best performance of $\mathcal{A}$ on distribution $\mathcal{D}$.

There are many ways of measuring the cost, $o(\mathcal{A}, \theta, I, s)$, of running algorithm $\mathcal{A}$ with parameter configuration θ on an instance I, using seed s in case of randomized algorithm. For example, we might be interested in

- the computational resources consumed by the given algorithm (such as runtime, memory or communication bandwidth), or
- the approximation error, or
- the improvement achieved over an instance-specific reference cost,
- the quality of the solution found.

Algorithm Configuration (Parameter Tuning) Problem

The behaviour of the algorithms can vary significantly between multiple runs on different instances or when randomized algorithms are run repeatedly with fixed parameters on a single problem instance.

Therefore, the cost of a candidate solution θ is defined as

$$c(\theta) = m(O_\theta)$$

the statistical population parameter m of the cost distribution $O_\theta(\mathcal{A}, \theta, \mathcal{D})$, over instances drawn from distribution of instances, $\mathcal{D}$, and multiple independent runs.

An optimal solution, θ^*, minimizes $c(\theta)$:

$$\theta^* \in \arg\min_{\theta \in \Theta} c(\theta).$$

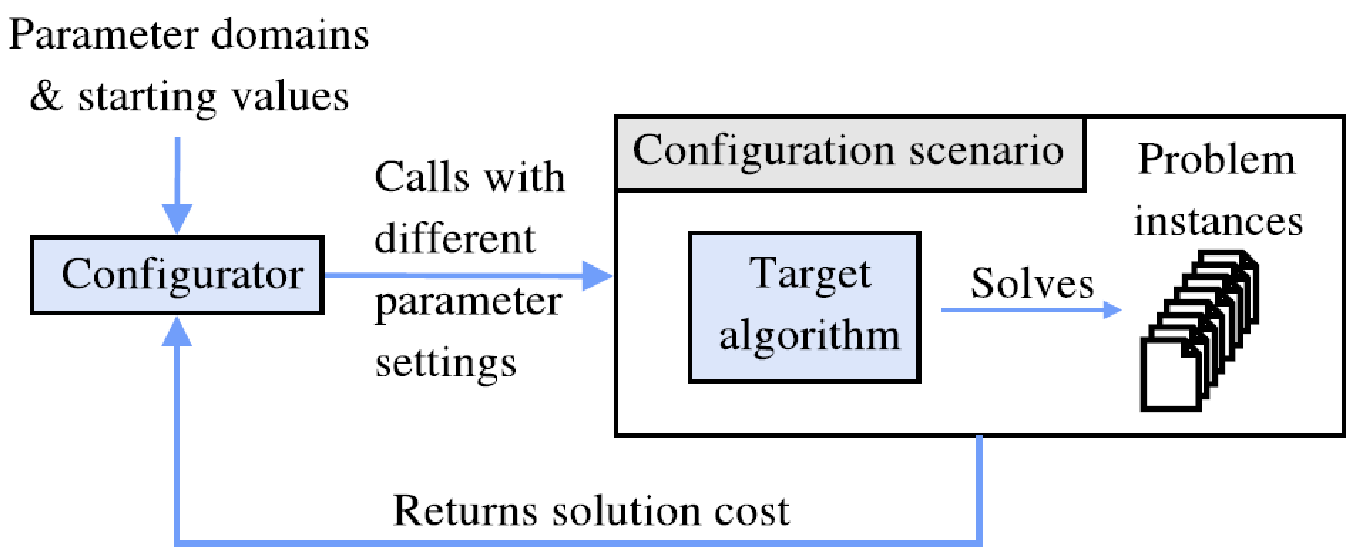
For example, we might aim to minimize mean runtime or median solution cost.

The $O_\theta(\mathcal{A}, \theta, \mathcal{D})$ is typically unknown, so we can only acquire approximations of their statistics, $c(\theta)$, based on a limited number of samples (i.e. the cost of single executions of $\mathcal{A}(\theta)$) – let's denote an approximation of $c(\theta)$ based on N samples by $\widehat{c}_N(\theta)$.

- For deterministic algorithms, the algorithm $\mathcal{A}$ is run on $N \leq M$ instances (M is the size of the finite training set of instances).
- For randomized, algorithms, we can run multiple runs with different seeds if $M < N$.

Parameter Tuning Scenario

Configuration scenario [Hutter, 2009]



Searching the Parameter Configuration Space

Manually-executed local search in parameter configuration space

1. begin with some initial parameter configuration;
2. experiment with modifications to single parameter values at a time, accepting new configurations whenever they result in improved performance (**iterative first improvement procedure**);
3. repeat step 2 until no single-parameter change yields an improvement.

This procedure stops as soon as it reaches a local optimum (a parameter configuration that cannot be improved by modifying a single parameter value).

Issues when trying to automate the process

- Which parameter configurations should be evaluated?
- Which problem instances should be used and how many runs should be performed on each instance?
- Which cutoff time κ_i should be used for each run?

ParamILS: Iterated Local Search in Parameter Configuration Space

Employs **Iterated Local Search** that builds a chain of local optima by iterating through a main loop consisting of:

1. a solution perturbation to escape from local optima,
2. a subsidiary local search procedure and
3. an acceptance criterion to decide whether to keep or reject a newly obtained candidate solution.

ParamILS($\theta_0, r, p_{restart}, s$)

1. uses a combination of default and random settings for initialization,
   θ_0 is the initial parameter configuration, and
   r is the number of randomly chosen configurations for initialization,
2. uses a one-exchange neighborhood (one parameter is modified in each search step),
3. employs iterative first improvement as a subsidiary local search procedure,
4. uses a fixed number, s, of random moves for perturbation,
5. always accepts better or equally-good parameter configurations,
6. re-initializes the search at random with probability $p_{restart}$.

$ParamILS(N)$: Algorithm

Input : Initial configuration $\boldsymbol{\theta}_0 \in \boldsymbol{\Theta}$, algorithm parameters r, $p_{restart}$, and s.
Output : Best parameter configuration $\boldsymbol{\theta}$ found.

for $i = 1, \ldots, r$ **do**
    $\boldsymbol{\theta} \leftarrow$ random $\boldsymbol{\theta} \in \boldsymbol{\Theta}$;
    if better($\boldsymbol{\theta}, \boldsymbol{\theta}_0$) **then** $\boldsymbol{\theta}_0 \leftarrow \boldsymbol{\theta}$;

$\boldsymbol{\theta}_{ils} \leftarrow$ *IterativeFirstImprovement* ($\boldsymbol{\theta}_0$);
while not *TerminationCriterion()* **do**
    $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta}_{ils}$;

    // ===== Perturbation
    for $i = 1, \ldots, s$ **do** $\boldsymbol{\theta} \leftarrow$ random $\boldsymbol{\theta}' \in Nbh(\boldsymbol{\theta})$;

    // ===== Basic local search
    $\boldsymbol{\theta} \leftarrow$ *IterativeFirstImprovement* ($\boldsymbol{\theta}$);

    // ===== AcceptanceCriterion
    if better($\boldsymbol{\theta}, \boldsymbol{\theta}_{ils}$) **then** $\boldsymbol{\theta}_{ils} \leftarrow \boldsymbol{\theta}$;
    with probability $p_{restart}$ **do** $\boldsymbol{\theta}_{ils} \leftarrow$ random $\boldsymbol{\theta} \in \boldsymbol{\Theta}$;

return overall best $\boldsymbol{\theta}_{inc}$ found;

Procedure *IterativeFirstImprovement* ($\boldsymbol{\theta}$)
repeat
    $\boldsymbol{\theta}' \leftarrow \boldsymbol{\theta}$;
    foreach $\boldsymbol{\theta}'' \in Nbh(\boldsymbol{\theta}')$ *in randomized order* **do**
        if better($\boldsymbol{\theta}'', \boldsymbol{\theta}'$) **then** $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta}''$; **break;**

until $\boldsymbol{\theta}' = \boldsymbol{\theta}$;
return $\boldsymbol{\theta}$;

$BasicILS(N)$: Procedure $better_N(\theta_1, \theta_2)$

Basic variant, $BasicILS(N)$, uses procedure $better(\theta_1, \theta_2)$ that compares two cost approximations $\widehat{c}_N(\theta_1)$ and $\widehat{c}_N(\theta_2)$ based on exactly N samples from the respective cost distributions $O_\theta(\mathcal{A}, \theta_1, \mathcal{D})$ and $O_\theta(\mathcal{A}, \theta_2, \mathcal{D})$ – the same N instances are used for all configurations θ_i.

Procedure $better_N(\theta_1, \theta_2)$ simply compares estimates $\widehat{c}_N(\theta_1)$ and $\widehat{c}_N(\theta_2)$ based on the same N instances using the same random seeds.

- It updates the best-so-far solution, θ_{inc}.

Input : Parameter configuration $\boldsymbol{\theta}_1$, parameter configuration $\boldsymbol{\theta}_2$
Output : True if $\boldsymbol{\theta}_1$ does better than or equal to $\boldsymbol{\theta}_2$ on the first N instances; false otherwise
Side Effect : Adds runs to the global caches of performed algorithm runs $\mathbf{R}_{\boldsymbol{\theta}_1}$ and $\mathbf{R}_{\boldsymbol{\theta}_2}$; potentially updates the incumbent $\boldsymbol{\theta}_{inc}$

```
1 ĉ_N(θ2) ← objective(θ2, N)
2 ĉ_N(θ1) ← objective(θ1, N)
3 return ĉ_N(θ1) ≤ ĉ_N(θ2)
```

Over-Confidence and Over-Tuning

There are two main problems associated with the use of a fixed number of N training samples.

1. **Over-confidence** – the cost estimated for the best training configuration, θ^*_{train}, underestimates the cost of the same configuration on the test set (i.e. previously unseen samples).

   $\widehat{c}_N(\theta^*_{train}) = \min\{\widehat{c}_N(\theta_1), \ldots, \widehat{c}_N(\theta_n)\}$ is a biased estimator of $c(\theta^*)$,

   where $\theta^* \in \arg\min_{\theta_i}\{c(\theta_1), \ldots, c(\theta_n)\}$.

   It was shown that $E[\min\{\widehat{c}_N(\theta_1), \ldots, \widehat{c}_N(\theta_n)\}] \leq E[c(\theta^*)]$ with equality only holding in pathological cases.

   This effect increases with the number of configurations evaluated and the variance of the cost distributions, and it **decreases as N is increased**.

2. **Over-tuning** – the situation where additional tuning actually impairs test performance.

   This effect can be suppressed by using large number of runs.

Over-Confidence and Over-Tuning

There are two main problems associated with the use of a fixed number of N training samples.

1. **Over-confidence** – the cost estimated for the best training configuration, θ^*_{train}, underestimates the cost of the same configuration on the test set (i.e. previously unseen samples).

   $\widehat{c}_N(\theta^*_{train}) = \min\{\widehat{c}_N(\theta_1), \ldots, \widehat{c}_N(\theta_n)\}$ is a biased estimator of $c(\theta^*)$,

   where $\theta^* \in \arg\min_{\theta_i}\{c(\theta_1), \ldots, c(\theta_n)\}$.

   It was shown that $E[\min\{\widehat{c}_N(\theta_1), \ldots, \widehat{c}_N(\theta_n)\}] \leq E[c(\theta^*)]$ with equality only holding in pathological cases.

   This effect increases with the number of configurations evaluated and the variance of the cost distributions, and it **decreases as N is increased**.

2. **Over-tuning** – the situation where additional tuning actually impairs test performance.

   This effect can be suppressed by using large number of runs.

The question is how to choose the optimal number of training instances, N?

- Using too small N leads to good training performance, but poor generalization to previously unseen test benchmarks.
- On the other hand, we cannot evaluate every parameter configuration on an enormous training set - if we did, search progress would be unreasonably slow.

FocusedILS

$FocusedILS$ avoids the problems with over-confidence and over-tuning by adaptively varying the number of training samples considered from one parameter configuration to another in order to focus samples on promising configurations.

- $N(\theta)$ denotes the number of runs available to estimate the cost statistic $c(\theta)$.

The question is how to compare two parameter configurations θ_1 and θ_2 for which $N(\theta_1) \leq N(\theta_2)$?

- What if we computed the empirical statistics based on the available number of runs for each configuration?

FocusedILS

$FocusedILS$ avoids the problems with over-confidence and over-tuning by adaptively varying the number of training samples considered from one parameter configuration to another in order to focus samples on promising configurations.

- $N(\theta)$ denotes the number of runs available to estimate the cost statistic $c(\theta)$.

The question is how to compare two parameter configurations θ_1 and θ_2 for which $N(\theta_1) \leq N(\theta_2)$?

- What if we computed the empirical statistics based on the available number of runs for each configuration?

  Can lead to systematic bias if, for example, the first instances are easier than the average ones.

Domination: Configuration θ_1 dominates θ_2 when at least as many runs have been conducted on θ_1 as on θ_2, and the performance of $\mathcal{A}(\theta)$ on the first $N(\theta_1)$ runs is at least as good as that of $\mathcal{A}(\theta_2)$ on all of its runs.

$$\theta_1 \text{ dominates } \theta_2 \text{ if and only if } N(\theta_1) \geq N(\theta_2) \text{ and } \widehat{c}_{N(\theta_2)}(\theta_1) \leq \widehat{c}_{N(\theta_2)}(\theta_2).$$

ParamILS version, called FocusedILS, encodes a comparison strategy based on the domination in procedure $better_{Foc}(\theta_1, \theta_2)$.

FocusedILS: Procedure $better_{Foc}(\Theta_1, \Theta_2)$

Procedure $better_{Foc}(\Theta_1, \Theta_2)$

1. first it acquires one additional run for the configuration i having smaller $N(\theta_i)$, or one run for both configurations if $N(\theta_1) = N(\theta_2)$;
2. then, it continues performing runs in this way until one configuration dominates the other.

   It returns true if θ_1 dominates θ_2, and false otherwise.

It keeps track of the total number, B, of configurations evaluated since the last improving step.

- Whenever $better_{Foc}(\Theta_1, \Theta_2)$ returns true, B extra (bonus) runs are performed for θ_1 and B is reset to 0.
- This way it is ensured that many runs are performed with good configurations $\Longrightarrow$ the error made in every comparison of two configurations θ_1 and θ_2 decreases on expectation.

  A theoretical analysis of the resulting algorithm proves its convergence to the globally optimal parameter configuration.

FocusedILS: Procedure $better_{Foc}(\Theta_1, \Theta_2)$

Input : Parameter configuration $\boldsymbol{\theta}_1$, parameter configuration $\boldsymbol{\theta}_2$
Output : True if $\boldsymbol{\theta}_1$ dominates $\boldsymbol{\theta}_2$, false otherwise
Side Effect: Adds runs to the global caches of performed algorithm runs $\mathbf{R}_{\boldsymbol{\theta}_1}$ and $\mathbf{R}_{\boldsymbol{\theta}_2}$; updates the global counter B of bonus runs, and potentially the incumbent $\boldsymbol{\theta}_{inc}$

$B \leftarrow B + 1$
if $N(\boldsymbol{\theta}_1) \leq N(\boldsymbol{\theta}_2)$ **then**
    $\boldsymbol{\theta}_{min} \leftarrow \boldsymbol{\theta}_1; \boldsymbol{\theta}_{max} \leftarrow \boldsymbol{\theta}_2$
    if $N(\boldsymbol{\theta}_1) = N(\boldsymbol{\theta}_2)$ **then** $B \leftarrow B + 1$
else $\boldsymbol{\theta}_{min} \leftarrow \boldsymbol{\theta}_2; \boldsymbol{\theta}_{max} \leftarrow \boldsymbol{\theta}_1$
repeat
    $i \leftarrow N(\boldsymbol{\theta}_{min}) + 1$
    $\hat{c}_i(\boldsymbol{\theta}_{max}) \leftarrow objective(\boldsymbol{\theta}_{max}, i)$ *// If $N(\boldsymbol{\theta}_{min}) = N(\boldsymbol{\theta}_{max})$, adds a new run to $\mathbf{R}_{\boldsymbol{\theta}_{max}}$.*
    $\hat{c}_i(\boldsymbol{\theta}_{min}) \leftarrow objective(\boldsymbol{\theta}_{min}, i)$ *// Adds a new run to $\mathbf{R}_{\boldsymbol{\theta}_{min}}$.*
until *dominates($\boldsymbol{\theta}_1$, $\boldsymbol{\theta}_2$) or dominates($\boldsymbol{\theta}_2$, $\boldsymbol{\theta}_1$)*
if *dominates($\boldsymbol{\theta}_1$, $\boldsymbol{\theta}_2$)* **then**
    // ===== Perform B bonus runs.
    $\hat{c}_{N(\boldsymbol{\theta}_1)+B}(\boldsymbol{\theta}_1) \leftarrow objective(\boldsymbol{\theta}_1, N(\boldsymbol{\theta}_1) + B)$ *// Adds B new runs to $\mathbf{R}_{\boldsymbol{\theta}_1}$.*
    $B \leftarrow 0$
    return true
else return false

Procedure dominates($\boldsymbol{\theta}_1, \boldsymbol{\theta}_2$)
if $N(\boldsymbol{\theta}_1) < N(\boldsymbol{\theta}_2)$ **then return** false
return $objective(\boldsymbol{\theta}_1, N(\boldsymbol{\theta}_2)) \leq objective(\boldsymbol{\theta}_2, N(\boldsymbol{\theta}_2))$

Adaptive Capping of Algorithm Runs

Often, the search for a performance-optimizing parameter setting spends a lot of time with evaluating a parameter configuration that is much worse than other, previously-seen configurations.

Ex.: Let's assume a case where parameter configuration θ_1 takes a total of 10 seconds to solve $N = 100$ instances (i.e. it has a mean time runtime of 0.1 seconds per instance), and another parameter configuration θ_2 takes 100 seconds to solve the first of these instances.

Clearly, when comparing the mean runtimes of θ_1 and θ_2 based on this set of instances, it is not necessary to run θ_2 on remaining 99 instances. Instead, we can terminate the first run of θ_2 after $10 + \epsilon$ seconds, which is a lower bound on θ_2's mean runtime of $0.1 + \epsilon/100$. this lower bound exceeds the mean runtime of θ_1, so we can already be sure that θ_2 cannot do better than θ_1.

Question is how to determine the cutoff time for each run of the target algorithm, $\mathcal{A}$, in an automated way?

Adaptive Capping of Algorithm Runs

Often, the search for a performance-optimizing parameter setting spends a lot of time with evaluating a parameter configuration that is much worse than other, previously-seen configurations.

Ex.: Let's assume a case where parameter configuration θ_1 takes a total of 10 seconds to solve $N = 100$ instances (i.e. it has a mean time runtime of 0.1 seconds per instance), and another parameter configuration θ_2 takes 100 seconds to solve the first of these instances.

Clearly, when comparing the mean runtimes of θ_1 and θ_2 based on this set of instances, it is not necessary to run θ_2 on remaining 99 instances. Instead, we can terminate the first run of θ_2 after $10 + \epsilon$ seconds, which is a lower bound on θ_2's mean runtime of $0.1 + \epsilon/100$. this lower bound exceeds the mean runtime of θ_1, so we can already be sure that θ_2 cannot do better than θ_1.

Question is how to determine the cutoff time for each run of the target algorithm, $\mathcal{A}$, in an automated way?

Adaptive capping is based on the idea of avoiding unnecessary runs of the algorithm $\mathcal{A}$ by developing bounds on the performance measure to be optimized.

- **Trajectory-preserving capping** – provably does not change $BasicILS$'s trajectory, but can lead to large computational savings.
- **Aggressive capping** – potentially yielding even better performance.

BasicILS: Trajectory-Preserving Capping

Trajectory-Preserving Capping (for the case of optimizing the mean of non-negative cost function) – implements bounded evaluation of a parameter configuration θ, procedure $objective(\theta, bound)$, based on N runs so that

- it sequentially performs runs for θ and after each run computes a lower bound on $\widehat{c}_N(\theta)$ based on $i \leq N$ runs performed so far;
- once the lower bound exceeds the $bound$ passed as an argument, the remaining runs for θ_2 are skipped.

Procedure $better_N(\theta_1, \theta_2)$ is modified as follows

$bound \leftarrow \infty$
$\widehat{c}_N(\theta_2) \leftarrow objective(\theta_2, N, bound)$
$bound \leftarrow \widehat{c}_N(\theta_2)$
$\widehat{c}_N(\theta_1) \leftarrow objective(\theta_1, N, bound)$
return $\widehat{c}_N(\theta_1) \leq \widehat{c}_N(\theta_2)$

BasicILS: Aggressive Capping

The *trajectory-preserving capping* computes the upper bound on the cumulative runtime from the best configuration encountered in the current ILS iteration.

- The new parameter configuration θ often is of poor quality, thus the capping criterion does not apply as quickly as it would if the comparison was performed against the overall *incumbent*.

Aggressive Capping – bounds the evaluation of any configuration by the performance of the best-so-far (incumbent) configuration, θ_{inc}, multiplied by a factor **bound multiplier**, bm.

- When two configurations θ_1 and θ_2 are compared and the evaluations **both are terminated preemptively**, the configuration having solved **more instances within the allowed time is** considered the **better** one.

  Ties are broken to favor moving to a new parameter configuration.

- For $bm = \infty$, this reduces to trajectory-preserving but no savings strategy.

  $bm = 1$ means that once we know the evaluated configuration θ is worse than the θ_{inc}, its evaluation is terminated (recommended setting is $bm = 2$).

FocusedILS: Adaptive Capping

FocusedILS varies the number of runs used to evaluate each parameter configuration.

How can the adaptive capping be used here?

FocusedILS: Adaptive Capping

FocusedILS varies the number of runs used to evaluate each parameter configuration.

How can the adaptive capping be used here?

- By using separate bounds for every number of runs, N.

Recommended Material

Frank Hutter, Holger H. Hoos, Kevin Leyton-Brown, and Thomas Stützle: ParamILS: An Automatic Algorithm Configuration Framework. In *Journal of Artificial Intelligence Research* (JAIR), volume 36, pp. 267-306, October 2009.

Other papers and SW available at `http://www.cs.ubc.ca/labs/beta/Projects/ParamILS/`