Self-Search (Extended Abstract)

Jawad Elomari

Warwick Business School, University of Warwick, Coventry, United Kingdom
J.Elomari@warwick.ac.uk

Abstract. An online operator control algorithm, Self-Search, is described. The main idea is to consider multiple performance measures upon deciding which operator to apply next, these measures are: quality variation, application time, number of time an operator is applied, and number of times an operator produced the same solution.

Keywords: parameter optimization, online parameter control, hyper-heuristics, algorithm configuration.

1 Introduction

Almost all search algorithms have a number of free parameters that enable them to solve a wide variety of optimization problems, these parameters, if incorrectly set, can hinder the algorithm's performance in terms of solution quality and running time. Parameter optimization for search algorithms is concerned with finding parameter values at which the algorithm performs as best as possible, of course the term "*best*" depends on the application, it can be running time, solution quality, or number of problem instances, or domains, an algorithm can solve.

Solution approaches to this problem can be broadly classified into offline tuning and online control, the difference being whether parameter settings are learnt online, in which case they change dynamically, or offline, in which case they remain constant or change according to a pre-defined schedule. Here we present an online control algorithm, dubbed Self-Search, which selects a Lower-Level Heuristic (LLH) based on its performance and the current status of the search. The algorithm is built within an Evolutionary Algorithm (EA) framework and is applied to the Cross-domain Heuristic Search Challenge (CHeSC) of [1].

2 The Algorithm

The main idea is to implement different strategies during the search depending on the performance of the LLH and the current search status. The performance measures for a LLH are: quality variation, implementation time, number of times it was applied, number of times it produced the same solution, and the number of times it did not improve a solution. Search statuses that affect which strategy to use include: expected number of generations within the allowed time limit, number of generations since the best-so-far solution has improved, number of generations since the generation-best solution has improved, and the expected number of generations remaining. The Self-Search algorithm has its own set of hyper parameters which will be highlighted throughout the text with a different `font`, and listed at the end along with their values.

The first important decision to make is whether the algorithm should follow an explorative strategy or an exploitative one, such a decision is based on the expected number of generations the algorithm can achieve within the allowed time. An initial rough estimate is obtained by applying each LLH to an initial solution (randomly generated) and measuring its application time, these times are averaged and, based on the population size chosen and the time limit, an expected number of generations is obtained. If the number of generations is below a pre-defined limit (`minGenerationsReq`), the algorithm follows an exploitative strategy; otherwise, it follows an explorative one. Before going into the details of the explorative and exploitative strategies, the EA framework used should be described.

- Initial random solutions are generated according to the population size (`populationSize`)
- The selected LLH is applied to all individuals in the population (no parent selection). Selection of LLHs is probabilistic.
- The number of offspring created is equal to the population size.
- Elitist survival selection with no duplicate solutions.

2.1 Explorative Strategy

Within this strategy one looks for LLHs which do not produce the same solution and cause high improvements. This is done as follows:

- Find the LLH with the minimum number of times it produced the same solution relative to how many times it was applied (minSameSol%). At the beginning of the search this percentage is likely to be zero for most LLHs, since most will produce better solutions.
- Identify other LLHs with the same minSameSol%.
- Out of all the LLHs with the same minSameSol%, find the one with the highest improvement rate (i.e. improvement made per unit time). Call this LLH_Best.
- Update the selection probabilities using the Adaptive Pursuit[1] (AP) method [2], where LLH_Best gets a high selection probability while the rest get a low selection probability.
- This process is repeated in each generation and the LLH_Best is updated accordingly.

Continuing with this cycle will cause the algorithm to stagnate, the stagnation point used here is: no improvement on the best-so-far solution found for a number of consecutive iterations, this number is taken as a percentage of the expected generations the algorithm can do within the time limit (`noImprovementFactor`).

[1] AP has its own set of parameters: minimum selection probability P_{min} and adaptation rate β. the chosen values for these parameters will be listed at the end as well.

The expected number of generations is updated with each generation to obtain a more accurate estimate. Once the stagnation point is reached, the algorithm checks how many generations it can still do within the available time, if it is higher than `remainingGenerations`, the algorithm explores more; otherwise, it exploits more.

More exploration means increasing the `IntensityOfMutation` parameter by (number of times stagnation has occurred/10) and maintaining the `DepthOfSearch` at its default low value. These two parameters control how the LLHs work; they have different meanings depending on the problem domain the LLHs are used in. For more details on them see [3]. The algorithm applies LLH_Best, with the new value of `IntensityOfMutation`, to each individual in the population. If the same solution is produced, the algorithm applies another LLH with the same minSameSol%, this LLH will have a lower improvement rate than LLH_Best of course, but will still be one which is expected to cause an improvement based on its performance so far. This is why one always maintains a *candidateList* of LLHs with the same minSameSol% and different improvement rates. It is possible that neither the LLH_Best nor any of the LLHs in the *candidateList* are able to produce a different solution when the stagnation point is reached, in that case the solution remains as is and the algorithm moves on to the next solution in the population. On the other hand, if LLH_Best or any of the LLHs in *candidateList* does produce a different solution, the algorithm shifts its focus on that LLH by increasing its selection probability according to the AP method.

More exploitation means decreasing the `IntensityOfMutation` to its low default value and setting `DepthOfSearch` to a high value plus (number of times stagnation has occurred/10). Similar to more exploration above, LLH_Best is applied, if it produces the same solution, the algorithm tries other LLHs which are expected to improve the solution, but are not necessarily expected to produce different solutions. Such LLHs are maintained in another list *candidateList2*. Again, if LLH_Best or any of the LLHs in *candidateList2* does produce a different solution, the algorithm shifts its focus on that LLH by increasing its selection probability according to the AP method.

2.2 Exploitative Strategy

First, the `IntensityOfMutation` is set to a low value and the `DepthOfSearch` is set to a high value, then one only looks for LLHs with high improvement rates. This is done as follows:

- Find the LLH with the highest improvement rate relative to how many times it was applied and set it to be the LLH_Best.
- Identify other LLHs which cause improvements but not as high as LLH_Best. Store these LLHs in a list.
- Update the selection probabilities of LLH_Best and other LLHs according to the AP method.

The optimization cycle runs with these settings until stagnation occurs (same criterion as before), in which case the algorithm increases the `DepthOfSearch` parameter by (number of times stagnation has occurred/10) and tries out different

LLHs that are expected to improve the solution. Note that upon stagnation in the exploitative strategy, the algorithm does not consider how many remaining generations are expected, since this entire strategy is followed only if one expects a low number of generations, where it is better to allocate computational effort on exploitation rather than exploration. The exploitation strategy is the same as that used when the algorithm stagnates in the exploration strategy and the expected number of generations is below `remainingGenerations`.

3 The Algorithm's Hyper Parameters

As mentioned earlier, the proposed algorithm introduces its own set of parameters which need to be set in advance or adapted as well. A listing of them is in Table 1.

Table 1: List of Hyper Parameters

Hyper Parameter	Value
populationSize	5
minGenerationsReq	300
Adaptation rate β	0.5 for exploration 0.8 for exploitation
Minimum selection probability P_{min}	1/(2*number of LLHs)
IntensityOfMutation low	0.2
IntensityOfMutation high	0.8
DepthOfSearch low	0.2
DepthOfSearch high	0.8
remainingGenerations	0.5*expected number of generations
noImprovementFactor	0.05 for exploration 0.1 for exploitation

These values were set according to initial experimentations on the CHeSC data set. Moderate changes in these values, except the population size which needs to remain low, did not affect the algorithm's performance; however, a proper sensitivity analysis is needed to see how robust the algorithm is to these parameters.

4 Conclusion

The main idea of the presented algorithm is to apply different strategies to adapt to the current search status and LLHs performance. The algorithm introduces its own set of hyper parameters that need to be set or adapted. Initial experimentations did show that the algorithm's performance is not affected by moderate changes to the hyper parameters; however, more structured tests are needed to show the algorithm's robustness.

References

1. Ochoa, G. and M. Hyde. *Cross-domain Heuristic Search Challenge*. 2011 15-Mar-2011 [cited 2011 04-April]; Available from: http://www.asap.cs.nott.ac.uk/chesc2011/.
2. Thierens, D., *An adaptive pursuit strategy for allocating operator probabilities*, in *Proceedings of the 7th Annual Conference on Genetic and Evolutionary Computation*. 2005, ACM: Washington DC, USA. p. 1539-1546.
3. Curtois, T., et al. *HyFlex Software Documentation*. 2011 [cited 2011 13-May]; Available from: http://www.asap.cs.nott.ac.uk/chesc2011/javadoc/overview-summary.html.