A Stochastic Hyper-Heuristic for Optimising through Comparisons

Kieran Greer, Senior Member, IEEE.
Distributed Computing Systems, Belfast, UK.
Email: kgreer@distributedcomputingsystems.co.uk

Abstract: **This paper introduces a new hyper-heuristic framework for automatically searching and changing potential solutions to a particular problem. The solutions and the problem datasets are placed into a grid and then a game is played to try and optimise the total cost over the whole grid, using a randomising process. The randomisation could be compared to a simulated annealing approach, where the aim is to improve the solution space as a whole, possibly at the expense of certain better solutions. It is hoped that this will give the solution search an appropriate level of robustness to allow it to avoid local optima.**

Keywords: **hyper-heuristic, stochastic, corroborative evidence, genetic algorithms, simulated annealing.**

I. Introduction

This paper introduces a new algorithm for automatically searching and changing potential solutions to a particular problem using a hyper-heuristic. The solutions and the problem datasets are placed into a grid and then a game is played to try and optimise the total cost over the whole grid, using a randomising process. It would also work well for problems that might require some sort of symbolic evaluation instead of a numerical one. In that case, an exact evaluation of what value is 'better' might not be possible and so some sort of matching evaluation would be required instead. This paper only proposes the new algorithm and tries to justify it through a logical reasoning argument. Only a small amount of testing has been performed so far to prove its effectiveness, but the argument should make it clear what the algorithm is about and why it should work. This algorithm automates a lot of the selection process for selecting solutions to mutate, or keep, for evolving further. It also tries to produce a solution that is robust across the whole solution set and therefore should be able to generalise better than a more direct approach that might lead quickly to a local optimum.

The rest of the paper is organised as follows: section II introduces hyper-heuristics and section III describes some related technologies. Section IV describes the new heuristic, while section V gives an explanation of why it should work. Finally, section VI gives some conclusions on the work.

II. Hyper-Heuristics

Wikipedia describes a hyper-heuristic as:

> 'A hyper-heuristic is a heuristic search method that seeks to automate, often by the incorporation of machine learning techniques, the process of selecting, combining, generating or adapting several simpler heuristics (or components of such heuristics) to efficiently solve computational search problems. One of the motivations for studying hyper-heuristics is to build systems which can handle classes of problems rather than solving just one problem.'
>
> ...
>
> 'In a typical hyper-heuristic framework there is a high-level methodology and a set of low-level heuristics (either constructive or perturbative heuristics). Given a problem instance, the high-level method selects which low-level heuristic should be applied at any given time, depending upon the current problem state, or search stage.'

This problem solving method should probably be called a hyper-heuristic rather than a metaheuristic, because it controls the search performed by other heuristics rather than evaluate any problem directly itself. Two introductory chapters on hyper-heuristics can be found in [1] and [2]. The most likely application for this heuristic would be to select the best or most likely solutions to a problem through corroborative evidence. As described in [1], a typical hyper-heuristic algorithm might be:

1. Start with a set H of heuristic ingredients, each of which are applicable to a problem state and transform it to a new problem state. Examples of such ingredients in bin-packing are a single top-level iteration of 'Exact Fit' or a single top-level iteration of 'largest first, first fit';
2. Let the initial problem state be S_0.
3. If the problem state is S_i then find the ingredient that is in some sense most suitable for transforming that state. Apply it, to get a new state of the problem S_{i+1}.
4. If the problem is solved, stop. Otherwise go to 3.

They note that:

> 'the important point is that the hyper-heuristic has no knowledge as to the function of each heuristic.'
> ...
> 'Using its internal state, the hyper-heuristic has to decide which low level heuristic(s) it should call next.'

The new hyper-heuristic algorithm tries to automate the selection process and also robustly improve the solution pool that potential solutions can be selected from. This process uses a random or stochastic element which means that the best solutions will not always automatically be chosen. Instead, it tries to optimise over the whole solution and problem set, where if a number of solutions would benefit from the best solution being removed, then this is done. The algorithm relies not on selecting the best value, but on matching values over different solutions. This would be particularly useful if there is not a numerical evaluation, but a symbolic one being performed, for example. Then the notion of a 'better' evaluation is more difficult and possibly matching is the best method to try. This heuristic could also be useful for problems that try to reason over incomplete or uncertain information. The problem set can present the information in parts and it is up to the problem solver to recognise the whole from the bits that it receives. This could include fuzzy or noisy data that would confuse the evaluation, or partial information might lead to different plausible scenarios. The system then needs to judge which is the correct or most likely case. You could imagine a real world scenario where there are many different factors bringing in information, but a complete picture is not possible. Therefore, the system is required to reason with uncertainty, or only partial information. In that case, higher and incorrect evaluations could also result from information that is missing. Therefore, it might be preferable to select a solution based on the number of potential solutions that produce the same result. In percentage terms, if 10 solutions say that the value is 1 and only 5 say that it is 2, then it is more likely to be 1. This matching process is therefore key to the success of the hyper-heuristic.

III. Related Work

There is quite a lot of related work in the area of hyper-heuristics and problem solving and so only a few points will be made. Most important for this heuristic are probably genetic algorithms and simulated annealing, but problem solving is relevant to almost all areas of AI.

A. *Simulated Annealing*

The selection process contains a randomising element that could be compared to a simulated annealing approach. Simulated annealing provides a probability function, where a worse move may be made at certain iterations with a pre-determined probability. This also means that if the solution is trapped in a local optimum there is an opportunity of moving out of it again. Simulated annealing has a 'cooling factor' which is used to determine how often the probability function will allow a worse move. Initially the cooling factor is set quite high and worse moves are permitted more often. As the number of iterations completed increases, the cooling factor is decreased, making it less likely that a move to a worse solution will be allowed. This has the effect of allowing the search strategy to roam widely at the start and then settle on a particular area of the search space as the number of iterations increases. The new process could work in a similar manner, where a hill climbing approach of heading directly for the best solution will probably not be done. Any genetic mutations, or weight updates for machine learning algorithms, will be selected from a wider gene pool initially because of the randomness, making the solution more robust. Over time however, the process may eventually settle on a pool of more similar solutions that will naturally change their values much less.

B. *Genetic Programming*

Genetic algorithms work by creating a pool of potential solutions that try to solve a problem. The better solutions then mutate with each other to produce offspring that should in theory lead to an even better solution. They therefore search the solution space by mutating the better current solutions using certain selection criteria to do this. A recent paper [4] actually suggests a process that is in some ways similar to this one. They note that it might be better to select solutions based on their similarity to each other. They appear to state that it is more suitable for similar solutions to mutate and produce new solutions than for very different ones to do so. This is possibly a more consistent way of creating mutations. They note that a current trend is to have parallel or multiple groups of solution sets that start from different places to try and solve the problem. This would increase the robustness of the whole process. Potential solutions then need to be able to cross-over to a different solution pool to mutate, if they are found to produce fitness functions that are similar to the other solution set. They note that previous algorithms have suffered from problems such as the need to specify certain criteria for selecting which solutions to move to a different pool and their algorithm is able to automate this. Genetic algorithms would therefore really benefit from a good hyper-heuristic framework that could control how they crossed-over and mutated with each other. In [2] they point out the link between genetic programming and hyper-heuristics. They note that a number of authors have pointed out the suitability of genetic programming over other machine learning methods to automatically produce heuristics, but also note the disadvantages just written about. In particular, the often unintuitive values for parameters, which are typically found through a trial and error process.

IV. The New Hyper-Heuristic

This section describes the algorithm in more detail and then traces through one test illustrative example to show how it would work in practice. The new heuristic works as follows: A number of solutions are initially generated and there are also a set of problems that need to be solved. The problems are evaluated and the results are placed into a grid-like structure, where for example, the solutions make up the rows and the problems make up the columns. The values for the related solutions and problems should be added in a random manner so that no particular solution or problem is preferred. It is assumed that it is not known what a better value would be.

A game is then played that tries to optimise a total value over the whole grid. This is done my matching values across rows or columns. A match across rows would mean that two different solutions are evaluating a problem part in the same way. A match across columns would mean that two problem parts are related or apply to the same larger problem. For this first algorithm version, to match any two entities, the algorithm must remove any rows or columns that are in-between the two to be matched. This is simply the chosen process for discarding certain solutions and also the criteria for allowing a match. If the process is initially random, then this is fair to all potential solutions. The removed entries are not then part of any final solution. The algorithm must then try to calculate what entries to remove and what ones to keep, to produce the largest total over the whole grid, or the best match over the whole solution space. The solutions that are kept can then be used in the next iteration to generate new solutions through mutations, etc., that should slowly improve the solution pool overall. Any problems that are kept can be included in any new problem set and could be recognised as being important by the fact that any future solutions should be able to evaluate them properly as well. In pseudo-code format, this algorithm could look as follows:

1. Evaluate the problem set using the current solution set.
2. Place the solutions and problems into a grid in a random order.
3. Run the algorithm to determine the best matches.
4. If the new solution set is worse, then the resulting values do not need to be kept, but repeat again with a different set of new solutions/problems.
5. If the change between the current evaluation/solution set and the new evaluation/solution set closes to be very small or no change, then possibly an optimal solution set has been achieved. GoTo 10.
6. Keep the best solutions and mutate them to get the next solution set. Fill up the remaining solution numbers with new random ones.
7. Keep the better problems as part of the next dataset and fill up the remaining numbers possibly with new ones.
8. If the original positions for the kept solutions or problems is maintained, then their related optimal evaluation should always be possible and so a new solution set will have to improve on it.
9. GoTo 1.
10. Select the best solution from the current set.

A. Test Example

The following figures show how the algorithm might work in practice. Figure 1shows an example grid constructed from 4 initial possible solutions and 4 initial problems. The first matching operation removes solution 2 from the pool, so that solutions 1 and 3 can match over problems 2 and 4. So while solutions 2 and 4 have the best single matching value of 10, this is not included, because the two matches of 8 and 5 for solutions 1 and 3 produces a larger total overall. Solution 3 can then be removed to allow for matches between solutions 1 and 4. This leads to a new solution pool with the previous solutions 1, 3 and 4 included in it. The problem set to be solved could possibly include the problems 1, 2 and 4.

Initial Random State

	P1	**P2**	**P3**	**P4**
S1	3	5	7	8
S2	1	3	7	10
S3	6	5	9	8
S4	3	5	4	10

Solution Set: < empty >; *Problem Set:* < empty >

First Matching Operation

	P1	**P2**	**P3**	**P4**
S1	3	5	7	8
S3	6	5	9	8
S4	3	5	4	10

Solution Set: S1, S3; *Problem Set:* P2, P4.

Second Matching Operation

	P1	**P2**	**P3**	**P4**
S1	3	5	7	8
S4	3	5	4	10

Solution Set: S1, S3, S4; *Problem Set:* P1, P2, P4.

Figure 1. Example of the matching process to generate an overall optimal solution total of 21 points.

The next iteration would then mutate these kept solutions and also possibly also add random new ones to fill up the pool to the required number. New problems can also be added, but if the ones that were matched are still important, then they have been recognised and can be part of any subsequent optimisation process as well. The next iteration then randomly adds the solution and problem sets to the same grid-like structure and tries to optimise again.

This first iteration has returned an optimising total value of 21. It will eventually become the case that the optimising values or kept solution sets do not change by very much. This is then similar to the simulated annealing cooling factor, because any subsequent changes will become less and less. When this happens, the final solution set can be used as the optimal set with which to select a final answer from.

V. Why The Heuristic Should Work

The new heuristic uses the following principles to justify its process:

- The process is constructive – actually guiding the search for a solution and marking good solutions or important problem sets along the way.
- To generate an optimal solution, you need to remove worse solutions or problems as well as keep the better ones, to eventually terminate the process. There therefore needs to be a consistent way of selecting what solutions to keep/remove, to guide the process.
- The problem solving process should be as robust as possible; therefore there should be as little bias in it as possible. The algorithm will select new solutions from the widest possible range of values, but that range must also be accurate.
- The process works based on percentages – if more solutions suggest that one value is correct, then that should be the selected value.
- The process works on the assumption that better values can be as incorrect as worse values, which is valid when an exact evaluation is not possible.
- The process works best with matching problems but numerical data would also be suitable. The numerical data could also conceivably represent more than one problem, when the different problems (parts) would then be matched separately.
- One argument is that you should always keep the best solution(s) and the algorithm will try to do that unless the solution pool as a whole would benefit more from any of them being removed. Matching the better solutions, whenever possible, will also give a larger overall total.

Often, a genetic programming system will not produce good solutions on a first run as poor parameters are chosen. This is especially the case with the novice practitioner. It is therefore essential that different parameter settings are thoroughly investigated. The mutation process will then refine the parameter values and the results of the grid optimisation would be able to suggest what initial solutions should provide crossovers with each other to make the mutations. As it is only a framework, it can also match across different types of heuristic (genetic and nearest neighbour, or neural, etc).

VI. Conclusions

This new algorithm takes the view that the solutions do not know for certain what good or poor evaluations are. A higher value could be as poor an evaluation as a lower one. This might be because of uncertainty or incomplete information. Or if there is no specific numerical evaluation, then possibly matching is the only feasible option. There is also a possibility for combining problem parts (features) over a single solution that are shown to have similarities. Because of this, the solutions need to make use of corroborating evidence to come to some sort of conclusion. Also, as the heuristic does not know what solutions are best, it needs to be fair over all of them and for the same reason, would like some level of robustness that is obtained through the randomising process. The solution pool as a whole will improve, possibly at the expense of one or two better solutions.

This is only an initial attempt to use this sort of algorithm to solve a problem. Optimality might work best if the same position is retained for the solutions and problems that are kept, or possibly all matched solutions should be kept and not just the ones from the most recent iteration. So this sort of thing would need to be considered as well, but as this is a new framework there are lots of different possibilities as to how the evaluation process might be carried out.

If the positions in the grid for the solutions and problems that are kept is maintained, then it is clear that the next iteration will only change the kept solution set if it produces an optimal value that is better than the current one. The grid can always be solved in the same way again if there is no better solution. If the mutated solutions are then placed in the slots around the parent solutions that they were created from, then there is also a sense of the natural grouping that was written about in [4]. It will also then be more likely that one kept solution will be replaced by one derived from it, although crossover to a different group (different area in the grid) would also be possible. Future work will test this algorithm and try to determine what levels of accuracy it can achieve.

VII. References

[1] Burke, E., Kendall, G., Newall, J., Hart, E., Ross, P., and Schulenburg, S., Hyper-Heuristics: an Emerging Direction in Modern Search Technology. In Handbook of metaheuristics, chapter 16, pp. 457–474. Kluwer Academic Publishers, 2003.

[2] Burke, E.K., Hyde, M.R., Kendall, G., Ochoa, G., Ozcan, E., and Woodward, J.R., Exploring Hyper-heuristic Methodologies with Genetic Programming. In C. Mumford and L. Jain, editors, Collaborative Computational Intelligence. Springer, 2009.

[3] Cowling, P., Kendall, G., and Soubeiga, E., A parameter-free hyperheuristic for scheduling a sales summit. In: Proceedings of 4th Metahuristics International Conference (MIC 2001), Porto Portugal, 16–20 July, 2001, pp. 127–131.

[4] Sathya, S. and Kuppuswami, S., Analysing the migration effects in nomadic genetic algorithm, Int. J. Adaptive and Innovative Systems, Vol. 1, No. 2, 2010.