eXplore-Climb-Jump: A Hill Climbing based Cross-Domain Hyper-Heuristic

Kamran Shafi and Hussein A. Abbass

School of Engineering and Information Technology,
University of New South Wales,
Australian Defence Force Academy,
Canberra, Australia.
{k.shafi,h.abbass}@adfa.edu.au

Abstract. Hyper-heuristics refer to a class of optimisation techniques that aim at finding a right set of low-level search heuristics from a given pool and their right application sequence to produce effective and efficient solutions to optimisation problems. A simple but effective hyper-heuristic is presented which we developed to participate in the first Cross-domain Heuristic Search Competition (CHeSC-2011). The hyper-heuristic works by iteratively applying disruptive low-level heuristics to explore the search space and then a combination of low-level local-search and mutational heuristics that exploit the neighbourhood structure by improving upon the explored solutions. The results show that the hyper-heuristic perform competitively when compared with the eight default hyper-heuristics provided by the competition organisers.

Keywords: Hyper-heuristic, Optimisation, Hill-Climbing, Cross-Domain Heuristic Search

1 Introduction

Hyperheuristics, in search and optimisation, [1][4] are relatively recent techniques that focus on automating the development of optimisation heuristics. Their main motivation is to transpire domain-independent generalised optimisation methodologies that can provide effective and efficient solutions to diverse problems within and across domains. From design perspective, these heuristics are generally classified into two categories; those which are based on managing the application of existing domain-specific techniques or their parts and those that aim at entirely replacing the ad-hoc process of heuristic generation for a specific domain. From operational perspective, they can be categorised into online and offline learning methodologies where the latter in comparison to the former rely on a training phase to evolve their strategies. The hyperheuristics have successfully been applied across various problem domains including scheduling, routing and bin-packing problems.

The Automated Scheduling, Optimisation and Planning (ASAP) group at the University of Nottingham with the help of several other researchers introduced

the first Cross-domain Heuristic Search Challenge (CHeSC 2011) [2], opened in August 2010. The purpose of this challenge is to enhance the research in hyper-heuristics field by means of an open competition. *The challenge is to design a high level search strategy that controls a set of problem specific low level heuristics* for four known and two unknown problem domains. The known domains include Boolean Satisfiability (MAX-SAT), Bin-Packing, Personnel Scheduling and Permutation Flow-shop. Two other domains are hidden from the competitors and will be used for evaluating the submitted entries. The competition organising team provided a Java-based framework - HyFlex, which allows access to low-level heuristics for each domain. The low-level heuristics are categorised into mutational, ruin-recreate, local search and crossover based heuristics. To help participant preparing and evaluating new hyperheuristics for the competition, the organising team also provided ten problem instances from each of the known problem domains, eight *default hyperheuristics* along with their performance results for each of the forty problems and a scoring script based on the scoring strategy adopted by the competition. The competition evaluation consist of scoring each submitted entry of hyperheuristic against each other on three randomly chosen known instances from each domain, two unknown instances from each domain and five instances from each of the hidden domains. Furthermore, the scores for each entry will be averaged over multiple runs.

This extended abstract presents the details of our methodology that we developed for participating in CHeSC 2011. Our hyperheuristic methodology can be considered to fall in the category of meta-heuristics that work on the search space of heuristics (so-called, heuristic to choose heuristics). It works by iteratively applying disruptive low-level heuristics (ruin-recreate and crossover heuristics) to explore the search space and then a combination of low-level local-search and mutational heuristics that exploit the neighbourhood structure by improving upon the explored solutions. The complete algorithm is described in the following section. The initial evaluation of our methodology gave us a mediocre score of 182 (4th rank) when competing with the eight default hyperheuristic. We participated in the last leaderboard [3] update of the competition (our only submission to the leaderboard, since we entered the competition quite late) with this output and were ranked 13th among all the participants. A further tuning of the algorithm at the time of submission of this report improved our score to around 250 (1st rank) against the default heuristics. Considering the simplicity of the algorithm we believe this to be a competitive score among the submitted entries.

2 Algorithmic Description

Algorithm 1 below shows the pseudo-code for our hyperheuristic using the HyFlex API notation. In the initialisation phase, n more one solutions, where n corresponds to the sum of ruin-recreate and crossover heuristics for each problem, are generated using HyFlex's default initialisation function. The main loop of the algorithm works in two phases: in the first phase each of the explore heuristics

Algorithm 1 *XCJ HyperHeuristic*

```
1: XS ← indices of ruin-recreate and crossover heuristics
2: ES ← indices of local-search and mutational heuristics
3: n ← XS.size()
4: initialiseSolution(0)
5: curr_obj_value ← getFunctionValue(0)
6: best_obj_value ← curr_obj_value
7: for i = 1 → n + 1 do
8:    initialiseSolution(i)
9: end for
10: while !hasTimeExpired() do
11:    foreach h in XS do
12:       if h is_crossover_heuristic then
13:          applyHeuristic(h, 0, h + 1, h + 1)
14:       else
15:          applyHeuristic(h, 0, h + 1)
16:       end if
17:    end for
18:    for i = 0 → n do
19:       foreach h in ES do
20:          prev_obj_value ← getFunctionValue(i)
21:          improvement_steps ← c1
22:          while !hasTimeExpired() ⋀ improvement_steps > 0 do
23:             curr_obj_value ← applyHeuristic(h, i + 1, h + 1)
24:             delta ← 0
25:             if curr_obj_value < best_obj_value then
26:                best_obj_value ← curr_obj_value
27:                improvement_steps ← c1
28:                copySolution(h + 1, 0)
29:             else if curr_obj_value ≥ prev_obj_value then
30:                improvement_steps ← improvement_steps − 1
31:                delta ← 1.0 − prev_obj_value/curr_obj_value
32:             else
33:                improvement_steps ← c1
34:                prev_obj_value ← curr_obj_value
35:             end if
36:             if delta < c2 then
37:                copySolution(h + 1, i + 1)
38:             end if
39:          end while
40:       end for
41:    end for
42: end while
```

is applied to the solution saved at memory location 0 and the new solutions are recorded at their respective heuristic indices. Then all exploit heuristics are applied to each of the solutions generated by explore heuristics in sequence. Each heuristic gets c_1 chances to improve a solution, which is decremented every time a non-improved solution is found and reset to c_1 if an improved solution is found. All improved solutions are accepted without condition. However, only the non-improved solutions which are within a user-defined range c_2 of the previous improved solution are accepted. The best solution found in each iteration of the main loop is recorded at location 0. The main loops iterates until the time for search expires.

2.1 Variants

We experimented with various variants of the above algorithm which included accepting the solutions generated during exploitation phase without any constraints, using a Z-compression based distance function during exploration phase to ensure diverse solutions are generated during exploration phase and different conditions on setting the previous objective values. We also carried out some sensitivity analysis of the two user-defined parameters in our algorithm. Notwithstanding the crude analysis techniques, the best results were obtained for $c_1 = 10$ and $c_2 = 0.2$. As mentioned above, with these settings we scored 253 points when competing against the default hyper-heuristics. Our algorithm outperformed all default hyperheuristics on SAT and bin-packing problems and overall. The worst performance was recorded for personnel scheduling followed by permutation flowshop. We strongly believe that further adaptation of the two parameters (c_1 and c_2) and an in-depth analysis could significantly improve the performance of the algorithm.

References

1. E. Burke, G. Kendall, J. Newall, E. Hart, P. Ross, and S. Schulenburg. Hyper-heuristics: An emerging direction in modern search technology. *Handbook of meta-heuristics*, pages 457–474, 2003.
2. E.K. Burke, M. Gendreau, M. Hyde, G. Kendall, B. McCollum, G. Ochoa, A. Parkes, and S. Petrovic. The cross-domain heuristic search challenge–an international research competition. *Proc. Fifth International Conference on Learning and Intelligent Optimization (LION5), Lecture Notes in Computer Science, Vol 6683*, 2011.
3. Matthew Hyde and Gabriela Ochoa. Chesc leaderboard. `http://www.asap.cs.nott.ac.uk/chesc2011/leaderboard.html`, 2011. Accessed on 9th June 2011.
4. P. Ross. Hyper-heuristics. *Search Methodologies*, pages 529–556, 2005.