HAHA: A new Hyperheuristic Algorithm for the Cross Domain Search Challenge

Andreas Lehrbaum and Nysret Musliu

Vienna University of Technology, Database and Artificial Intelligence Group
lehrbaum@dbai.tuwien.ac.at

Abstract. This paper describes a new Hyperheuristic algorithm that performs well over a variety of different problem classes. A novel method for switching between working on a single solution and a pool of solutions is proposed. This method is combined with an adaptive strategy that guides the selection of the underlying low-level heuristics throughout the search. The algorithm is implemented based on the HyFlex framework and is going to be submitted as a candidate for the Cross-Domain Heuristic Search Challenge 2011.

1 Introduction

Hyperheuristics are a way to incorporate existing problem-class specific domain knowledge in the form of simple low-level heuristics into a higher-level search strategy which schedules and guides the execution of the lower-level heuristics. The idea is, that an ensemble of heuristics orchestrated by a top-level strategy is able to perform better in average on solving a wide range of problems as any of the underlying heuristics alone. A good survey of existing Hyperheuristic techniques is given in [1]. The HyFlex [2] framework offers an intuitive interface to utilise a set of given low-level search and mutation heuristics, easing the task of working purely on a high-level search strategy without any a priori knowledge about the problem instance or the heuristics available.

In this paper we propose an algorithm that consists of two distinct phases and uses a systematic, quality based selection strategy for the local-search heuristics based on the feedback from the search progress. The implementation is tailored towards the interface of the HyFlex framework and the specific requirements of the CHeSC competition rules[1]. Development and testing was performed using the provided test environment, which offered 4 different problem domains (MaxSat, Bin Packing, Personnel Scheduling and Flow Shop) along with according low-level search and mutation heuristics.

2 Algorithm Description

2.1 Overview

Following the classification of Burke et al. [1], the algorithm is an online learning Hyperheuristic, working mainly on heuristic selection. The novelity of the

[1] http://www.asap.cs.nott.ac.uk/chesc2011/rules.html

proposed approach lies in the repeated switching between two search variants, namely a systematic serial search phase working only on a single solution and a systematic parallel search phase working with a set of different solutions at the same time. We further propose a grading mechanism for the ordering of the low-level heuristics and a selection strategy for the available mutation heuristics.

The following main components of the algorithm are executed repeatedly:

- Serial search
- Generation of mutated solutions
- Parallel search
- Selection of a new working solution

After an initialisation phase in which the preliminary scores of the available heuristics are determined, the search continues by systematically executing the local-search heuristics in order of their respective quality scores. The splitting in a serial and a parallel search phase balances the focus of the search between exploration of new uncharted search space and the exploitation of the quality of the currently best working solution. Throughout the search process, the performance and runtime characteristics of the low-level heuristics are measured and the scores responsible for their selection are updated accordingly. In addition to that, the overall search progress is monitored continuously and mechanisms such as the temporary blocking of ineffective heuristics or the restart of the algorithm from the last best solution are put in place. A tabu-list in form of a ringbuffer containing a determined number of already visited solutions is used to avoid cycles in the search process.

2.2 Detailed Description of the Algorithm

Initialisation Phase: HAHA begins with calculating preliminary quality-scores of all the local-search heuristics available for the given problem instance. It does this by initialising the first solution slot and applying the heuristics in turn, recording their gain and their required runtime. At this stage, the parameters $depthOfSearch$ and $intensityOfMutation$ are both set to 1.0. The quality-score for each heuristic is calculated as average gain per runtime. The best solution found so far is returned as the working solution for the subsequent phase.

Serial Search Phase: The available local-search heuristics are applied to the current working solution sequentially in order of decreasing quality. Whenever a solution is found which either is better or equally good but different, it is accepted as current working solution and the search restarts with the best local-search heuristic. Additionally, a ringbuffer of limited size (determined experimentally) keeps track of the solutions visited so far. A solution which is already contained inside the ringbuffer is never accepted. The parameters $depthOfSearch$ and $intensityOfMutation$ are set to random values betwen $0.0 - 1.0$ before the application of each heuristic. This serial search phase ends whenever no improvement could be found with all the available heuristics therefore resulting in a locally optimal solution.

Quality Updates: In predetermined time intervals, the qualities of the local-search heuristics are updated to reflect their performance during the search so far. This can result in the re-ordering of the heuristic's search sequence.

Generation of Mutated Solutions: The available mutation and ruin-recreate heuristics are placed in a roulette-wheel reflecting their relative performance. Initially all mutation heuristics have the same chance of being selected. The performance of a mutation heuristic is judged based on the solution quality of the mutated solution after a local search has been applied to it. Given the current working solution as input, a set of mutated offsprings is generated by applying a set of mutation heuristics chosen by the roulette-wheel process. The $intensityOfMutation$ and $depthOfSearch$ parameters are set to a new random value each time before the mutation heuristics are applied.

Parallel Search Phase: Starting from the set of mutated solutions, the parallel search phase begins to work on the candidate solutions one after another. It begins with applying the best local-search heuristic to the first candidate. If it does not result in an improvement, the result is discarded and in the next round, the next best local-search heuristic will be applied to this solution. If a local improvement is found, it is accepted and the best local-search heuristic will be applied to this solution in the next round. Afterwards the search continues with the next solution slot and the local-search heuristic scheduled for this slot. Whenever a global improvement is found (i.e. the result is better than the currently best found solution so far), it is immediately accepted as the working solution for the next serial search phase and the parallel search is aborted. Otherwise the search goes on until all solutions have reached a local optimum with respect to all available local-search heuristics.

Working Solution Selection: Assuming no global improvement was found during the parallel search phase, a stochastic selection process of the available set of solutions takes place. One of the best solutions from the output of the serial search phase together with the locally optimal solutions from the parallel phase is selected with decreasing probability according to their qualities. If none of the solutions in the set is a new solution according to the ringbuffer, a random mutation will be applied to the best solution until the result is not contained inside the ringbuffer.

Blocking Inefficient Heuristics: If a local-search heuristic is found to be ineffective (determined by a low success-count), it is temporarily marked as blocked for the search process in both the serial and the parallel phase.

Restarting the Search: Whenever the search continues for a certain amount of time, producing only solutions which are worse than a given threshold (relative to the currently best solution), the search continues with the generation of mutated solutions from the currently best solution.

3 Results

The following tables show the results of the HAHA algorithm for the 40 test instances from the CHeSC competition in comparison to the 8 provided default

Hyperheuristics[2]. The value represents the median result per instance of 5 subsequent runs with different random seeds. Runtime was limited to a 600 second CPU-time equivalent measured by the official benchmarking program[3]

Inst.	HH1	HH2	HH3	HH4	HH5	HH6	HH7	HH8	HAHA
0	47	35	23	38	125	14	51	46	**4**
1	35	31	38	51	109	30	37	57	**23**
2	32	24	26	44	115	27	29	54	**16**
3	19	13	31	15	54	17	18	25	**1**
4	11	8	39	32	56	33	10	42	**2**
5	25	17	56	46	110	50	23	54	**2**
6	7	**6**	12	12	16	10	**6**	18	**6**
7	**6**	**6**	11	12	16	11	7	15	**6**
8	9	**8**	13	17	26	14	10	21	**8**
9	213	**211**	216	235	263	219	215	233	**211**

(a) Max-Sat

Inst.	HH1	HH2	HH3	HH4	HH5	HH6	HH7	HH8	HAHA
0	0.0174	0.0174	0.0068	0.0118	0.0507	0.0157	0.0220	0.0717	**0.0066**
1	0.0164	0.0170	0.0072	0.0117	0.0501	0.0118	0.0211	0.0679	**0.0066**
2	0.0234	0.0235	0.0242	0.0235	0.0276	0.0231	0.0242	0.0307	**0.0216**
3	0.0248	0.0246	0.0260	0.0246	0.0315	0.0242	0.0255	0.0329	**0.0240**
4	0.0063	0.0065	**0.0003**	0.0046	0.0151	0.0068	0.0069	0.0220	0.0050
5	0.0043	0.0084	0.0033	0.0036	0.0178	0.0084	0.0090	0.0237	**0.0031**
6	0.1148	0.0988	**0.0108**	0.0221	0.1718	0.0484	0.1388	0.1850	0.0513
7	0.1364	0.1360	**0.0191**	0.0637	0.1823	0.0843	0.1505	0.1841	0.0836
8	0.0550	0.0544	0.0580	0.0919	0.0928	0.0607	0.0555	0.1258	**0.0471**
9	0.0124	0.0113	0.0157	0.0267	0.0352	0.0167	0.0150	0.0429	**0.0058**

(b) BinPacking

Inst.	HH1	HH2	HH3	HH4	HH5	HH6	HH7	HH8	HAHA
0	3324	3309	3354	3339	3340	8342	3324	3334	**3292**
1	2240	2280	2485	2218	2260	11673	2135	3983	**2038**
2	360	340	545	340	380	755	370	**315**	345
3	22	18	28	17	21	74	19	21	**13**
4	25	23	35	25	22	69	22	27	**21**
5	23	26	34	22	22	73	**19**	26	21
6	1129	1113	1241	1125	1130	30094	1117	1643	**1104**
7	2261	2266	2270	2290	2278	46800	**2197**	4035	2246
8	3238	3173	3252	3236	3276	42151	3166	8074	**3172**
9	**9690**	9735	14321	12752	9815	95582	9789	19148	9712

(c) PersonnelScheduling

Inst.	HH1	HH2	HH3	HH4	HH5	HH6	HH7	HH8	HAHA
0	6381	6383	6368	6326	6387	6312	6391	6315	**6311**
1	6329	6336	6339	6263	6315	6271	6334	6265	**6242**
2	6404	6404	6398	6362	6407	6344	6404	6351	**6338**
3	6390	6389	6369	6366	6392	6350	6385	6366	**6325**
4	6481	6468	6439	6407	6468	6398	6480	6419	**6325**
5	10547	10549	10544	10503	10549	10500	10542	10522	**10497**
6	10965	10965	10965	10923	10965	**10922**	10968	10957	10923
7	26440	26440	26487	26382	26476	26424	26450	26406	**26348**
8	26984	26958	26998	26864	26974	26896	26928	26939	**26827**
9	26779	26754	26818	26721	26756	26704	26767	26726	**26685**

(d) FlowShop

4 Conclusion

According to these results, HAHA outperforms other standard Hyperheuristics in the majority of the problem instances provided by the CHeSC competition. We are confident that the combination of mechanisms put in place will work well outside the scope of this test data due to their flexible and straight-forward nature. Howevery, more extensive study on a broader range of problem domains and instances is needed to assess the generality of our approach.

References

1. E. Burke, M. Hyde, G. Kendall, G. Ochoa, E. Ozcan, and R. Qu, "A survey of hyper-heuristics," *Computer Science Technical Report No. NOTTCS-TR-SUB-0906241418-2747, School of Computer Science and Information Technology, University of Nottingham*, 2009.
2. E. Burke, T. Curtois, M. Hyde, G. Kendall, G. Ochoa, S. Petrovic, and J. Antonio, "HyFlex: A Flexible Framework for the Design and Analysis of Hyper-heuristics," in *Multidisciplinary International Scheduling Conference (MISTA 2009), Dublin, Ireland*, 2009, p. 790.

[2] `http://www.asap.cs.nott.ac.uk/chesc2011/defaulthh.html`

[3] `http://www.asap.cs.nott.ac.uk/chesc2011/benchmarking.html`