Hybrid Adaptive Evolutionary Algorithm Hyper Heuristic

Jonatan Gómez

Universidad Nacional de Colombia

Abstract. This paper presents a hyper heuristic that is able to adapt two low level parameters (depth of search rate and mutation intensity) and the probability of applying a low level heuristic to an evolving solution in order to improve the solution quality. Basically, two random subsets of heuristics are maintained: one sub set of the full set of heuristics and other sub-set of the local heuristics. For the full set of heuristics, the random selection is performed in such a way that at least one local heuristic is included in the sub-set. In each iteration, one heuristic is selected from the sub-set of heuristic according to its associated probability (just the heuristics in the subset are considered and their initial probability is the same for all of them). The same process is performed for selecting a local heuristic. Then, the heuristic and the local heuristic that were selected are applied to the candidate solution. The generated solution is compared against the candidate solution. If the solution is improved then the candidate solution will be replaced by the new one and both the heuristic and the local heuristics are rewarded (an increase in the probability of being selected is perform in a random fashion). If not improvement is made, both the heuristic and the local heuristic are punished (a decrease in the probability of being selected is perform in a random fashion), low level heuristic parameters are adjusted and a soft replacement policy is applied. The low level parameter depth of search rate (starting at 0.1) is increased in a random value in the range $[0.0, 0.1]$ up to a maximum value of 0.5. When the depth of search rate reaches 0.5, the parameter is reset to a value between $[0.0, 0.1]$. The intensity of mutation is set as the inverse value of the depth search rate. The soft replacement policy includes a control variable that determines if the range of depth of search rates has being tested or not. If so a new subset of heuristics and a new subset of local heuristics are selected and a new candidate solution is created by a randomly selected crossover heuristic (between the best reached solution and the generated solution) and a local heuristic. Otherwise the candidate solution is maintained only if the new solution has lower performance.

1 Algorithm description

The Hybrid Adaptive Evolutionary Hyper Heuristic (**HaEaHH**) proposed in this paper is based on the Hybrid Adaptive Evolutionary Algorithm (**HaEa**) proposed by Gomez in [1]. Since HaEa is an individual based approach, we

use only three individuals in the population for evolving a solution: the current candidate solution (parent), the generated solution (child), and the best solution reached during the evolution (best), see Algorithm 1.

Basically, the algorithm is divided in four main steps: setting initial variable values[1] (line 1), setting the low level heuristic parameters (lines 3 and 4), offspring generation (line 5), and replacement strategy (line 6).Notice that the dept of search parameter is always set in oppisited manner to the intensity of mutation parameter. When depth of search is high, intensity of mutation is low.

1.1 Initializing Variable Values

The initialization process is shown in Algorithm 2. We set the memory size to 3 (line 1), since we use only three individuals in the population for evolving a solution: the current candidate solution (parent), the generated solution (child), and the best solution reached during the evolution (best). We initialize two of them and select the best one as parent and initial best solution (lines 2-6). We obtain the full set of heuristics and the set of local heuristics (lines 7, 8) and set the "in use" heuristics to 4 and the "in use" local heuristic to be maximum 4 (lines 9, 10). We decided to keep just 4 heuristics (an maximum 4 local heuristics) since the probability of being selected will be no useful if a large number of heuristics are in use (the probability will tend to zero as the number of heuristics increase). The mechanism for selecting and changing the "in use" heuristics and local heuristics, (line 11) is shown in Algorithm 3. This is applied in the replacement strategy when required, see sub section 1.3 for more details. Finally, the Search Depth rate parameter is set to 0.1 meaning that more exploration (mutation) is performed at first than local search.

In the reset of the heuristics process (Algorithm 3), the "in use" heuristics are selected from the full set of heuristics after performing a random permutation of the set of heuristics that includes one local heuristic in the first N heuristics, and considering the first N heuristics in the set (line 1). Similar process is performed for the local heuristics and selecting the first M local heuristics as the "in use" ones (line 2). Two heuristics rates vectors are associated to each of the "in use" (local) heuristic sets, every "in use" heuristic having the same probability of being selected (lines 3,4).

1.2 Offspring Generation

The offspring generation process is shown in Algorithm 4. In each iteration, one heuristic is selected from the sub-set of "in use" heuristic according to its associated probability (line 1)[2]. The same process is performed for selecting a local heuristic (line 2). Then, the heuristic and the local heuristic that were selected are applied (in that order) to the candidate solution (lines 3-8). Notice that when the heuristic is a croosver heuristic, it is perform between the candidate solution and the best solution (line 4).

[1] See Table 1 for a reference of the set of variables used by HAEAHH.

[2] Just the heuristics in the subset are considered

Algorithm 1 Hybrid Adaptive Evolutionary Algorithm (HAEA)

HAEAHH(Pr)
1. init(Pr)
2. **while**(!hasTimeExpired()) **do**
3. setDepthOfSearch(P, α)
4. setIntensityOfMutation(P, $\beta - \alpha$)
5. offspring(P)
6. replacement(P, h, l)
7. **end**

Algorithm 2 Initializing Variable Values

init(P)
1. setMemorySize(P, 3)
2. initialiseSolution(P, 0)
3. initialiseSolution(P, 1)
4. p =indexOfBest(0, 1)
5. c =indexOfWorst(0, 1)
6. copySolution(p, b)
7. heu =getHeuristics(P)
8. loc = getLocalHeuristics(P)
9. $N = 4$
10. $M = min\{N, size(loc)\}$
11. reset_operators();
12. $\alpha = 0.1$

Algorithm 3 Resetting the "in-use" heuristics

reset_heuristics()
1. permutation'(heu)
2. permutation(loc)
3. $r_h = \left[\frac{1}{N}\right]^N$
4. $r_l = \left[\frac{1}{M}\right]^M$

Algorithm 4 Offspring Generation

offspring(Pr)
1. h =roulette(r_h)
2. l =roulette(r_l)
3. **if** $heu[h]$ is xover
4. applyHeuristic(P, $heu[h]$, p, b,c)
5. **else**
6. applyHeuristic(P, $heu[h]$, p, c)
7. **end**
8. f_c =applyHeuristic(P, $loc[l]$, c, c)

1.3 Replacement Strategy

The generated solution is compared against the candidate solution (line 1 Algoritmh 5). If the solution is improved then the candidate solution will be replaced by the new one and both the heuristic and the local heuristics are rewarded (lines 2-6). If not improvement is made, both the heuristic and the local heuristic are punished (lines 8,9), low level heuristic parameters are adjusted (line 10) and a soft replacement policy is applied (lines 11-29).

The reward/punish scheme is performed by increasing or decreasing, in a random fashion, the (local) heuristic probability of being selected, see Algorithm 6. The low level parameter depth of search rate (starting at 0.1) is increased in a random value in the range $[0.0, 0.1]$ up to a maximum value of 0.5. When the depth of search rate reaches 0.5, the parameter is reset to a value between $[0.0, 0.1]$, see Algorithm 6.

The soft replacement policy starts by trying to improve the best solution (lines 11-13) by applying a local heuristic, and uses a control variable to determine if the range of depth of search rates has being tested or not (lines 14-29) . If so a new subset of heuristics and a new subset of local heuristics are selected (line 27) and a new candidate solution is created by a randomly selected crossover heuristic (between the best reached solution and the generated solution) and a local heuristic (lines 17-22). Otherwise the candidate solution is maintained only if the new solution has lower performance (line 29).

Variable	**Description**
c, p, b	Child, parent and best (up to now) solution indices
f_c, f_p, f_b	Fitness of the child, parent and best solutions
N	Number of in use heuristics
M	Number of in use local heuristics
r_h	"in use" heuristic selection rates
r_l	"in use" local heuristic selection rates
P	Problem instance
h	Heuristic index
l	Local heuristic index
α	Depth of search rate
β	Depth of search rate limit
$\triangle$	Depth of search rate control variable
loc	Array of local heuristics
heu	Array of heuristics

Table 1. HAEA Variables

References

1. J. Gomez. Self adaptation of operator rates in evolutionary algorithms. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO 2004)*, June 2004.

Algorithm 5 Replacement Strategy

```
replacement( P, h, l )
1.  if f_c < f_p
2.    reward(r_h, h)
3.    reward(r_l, l)
4.    if f_c < f_b     copySolution(P, c, b)
5.    Δ = 0
6.    copySolution(P, c, p)
7.  else
8.    punish(r_h, h)
9.    punish(r_l, l)
10.   update_alpha()
11.   setDepthOfSearch(P, 0.2)
12.   setIntensityOfMutation(P, 0.2)
13.   f_b = applyHeuristic(P, loc[last], b, b);
14.   if △ ≥ β
15.     △ = 0
16.     if f_p ≠ f_b
17.       if heu[last] is xover
18.         f_p =applyHeuristic(P, hue[last], c, b, p)
19.       else
20.         f_p =applyHeuristic(P, hue[last], b, p)
21.       end
22.       f_p =applyHeuristic(P, loc[last], p, p)
23.       if f_p < f_b     copySolution(P, p, b)
24.     else
25.       copySolution(P, c)
26.     end
27.     reset_heuristics()
28.   else
29.     if f_p = f_c     copySolution(P, c, p)
30.   end
```

Algorithm 6 Reward/punish and Update of Depth of search rate strategies

```
reward( r, i )
1.  r[i] = (1.0 + δ) * r[i]
2.  normalize(r)

punish( r, i )
1.  r[i] = (1.0 − δ) * r[i]
2.  normalize(r)

update_alpha()
1.  x = 0.1 * δ
2.  α = α + δ
3.  △ = △ + δ
4.  if α > β     α = α − β
```