Benchmarking CHC on a New Application: The Software Project Scheduling Problem

Javier Matos and Enrique Alba

Universidad de Málaga, Spain
{jmo,eat}@lcc.uma.es

Abstract. In this article we analyze the behavior and scalability of the CHC algorithm over a benchmark of instances of the software project scheduling problem. Our goal is to analyze the performance of the CHC algorithm when solving realistic NP-hard combinatorial problems and test whether its previously reported high performance on similar problems also holds on this one. We perform a preliminary study to obtain a suitable configuration of the parameters in the algorithm. After choosing the configuration, we show the results for the problem instances in the benchmark. To give a reference on how CHC performs and scales, its results are compared against those of a GA. We conclude that CHC outperforms GA in large problem instances. Moreover, CHC produces promising results for the software project scheduling problem domain, and could be used by practitioners.

Keywords: Software Project Scheduling, Metaheuristics, Evolutionary Algorithms, Comparison, Benchmark.

1 Introduction

The CHC algorithm (*Cross generational elitist selection, Heterogeneous recombination, and Cataclysmic mutation*) has been applied with success for solving hard combinatorial optimization problems. For instance, several problems in which CHC has been used include the design of robust network topologies [11], the placement of wind turbines in a wind farm [3], the scheduling of tasks to processors in an heterogeneous environment [10,12], and a multiobjective antenna placement problem [9]. Previous works have shown that CHC is a competitive algorithm for solving optimization problems, frequently obtaining results that outperform those of the algorithms that were compared with it. However, it still remains not well-known in the community, in which many theses and articles do not use this kind of GA of low complexity and high numerical benefits.

In this article we apply for the first time CHC on this software problem. We push CHC to the limit using this new problem with the purpose of studying the behavior and scalability of the algorithm. Application results themselves are competitive and help locating CHC as a state-of-the-art technique for other applications in search based software engineering [7]. For the sake of the study, and to highlight CHC benefits, we compare it with a GA.

C.A. Coello Coello et al. (Eds.): PPSN 2012, Part II, LNCS 7492, pp. 448–457, 2012.
© Springer-Verlag Berlin Heidelberg 2012

The rest of the document is organized as follows. A description of the CHC algorithm is shown in Section 2. Section 3 presents the problem instances used in the benchmark, the initial study of the parameters to tune up CHC, the discussions of the results of CHC for the benchmark, and a detailed comparison with a GA. Conclusions of the study are outlined in Section 4.

2 The CHC Algorithm

The CHC algorithm is a special type of a GA designed to promote the best individuals in the population. One of the main characteristics of CHC is that it does not use mutation, that is a way to introduce new information in the population and avoid premature convergence; instead, it uses two mechanisms to stimulate diversity: an *incest prevention*, which only allows the recombination of individuals that are different enough (in terms of the Hamming distance), and a restart of part of the population when stagnation is detected. Initially, the threshold for allowing recombination is set to 1/4 of the chromosome length. During the recombination process, if the two randomly selected parents meet the condition to be recombined, then, the threshold is reduced by 1. As the algorithm runs, individuals become similar to each other, and eventually, the threshold to allow recombination reaches the value 0. This is how CHC detects that the population is stuck; thus, the algorithm performs a restart in part of the population: only the best p_r individuals are kept, whereas the others are restarted to increase the diversity.

The recombination operator in CHC is the *half uniform crossover* or HUX, that is a variant of the *uniform crossover* (UX), and consists in the random exchange of a half of the bits in which parents differ, as shown in Figure 1.

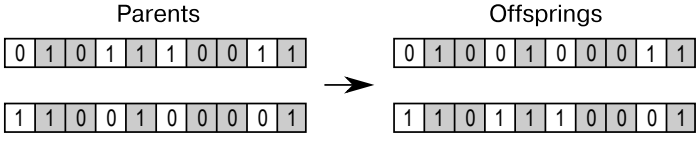

Fig. 1. The HUX recombination operator takes two parents and randomly decides on a swap for those bits at which their strings differ. Bits of the string for which the parents have the same value (highlighted in the figure) are not changed.

In Algorithm 1 we show the pseudocode of CHC as initially proposed by L. Eshelman [5]. The code reveals those features that make CHC different from traditional GAs: the elitist replacement strategy, the use of the HUX recombination operator, the absence of mutation, and the mechanism to restrict the recombination. The premature convergence of the population is reduced by the recombination policy and the diversity of individuals is ensured with the restart of a part of the population.

Algorithm 1. Pseudocode of the CHC algorithm

```
initialize(P(0))
generation ← 0
threshold ← 1/4 · chromosomelength
while not stopcriterion do
  parents ← selection(P(generation))
  if distance(parents) ≥ threshold then
    offspring ← HUX(parents)
    evaluate(offspring)
    newpop ← replacement(offspring, P(generation))
  end if
  if newpop == P(generation) then
    threshold ← threshold − 1
  end if
  generation ← generation + 1
  P(generation) ← newpop
  if threshold == 0 then
    reinitialization(P(generation))
    threshold ← 1/4 · chromosomelength
  end if
end while
return best solution ever found
```

3 Experimental Analysis

This section presents the experimental analysis performed in this work. First, we explain the problem instances of the benchmark. Then, we tune up CHC and apply it on a set of representative instances. Finally, we compare CHC against a GA.

3.1 Problem Instances: A Wide Representative Benchmark

To carry out the analysis of CHC, we have used 250 instances of a hard combinatorial problem in our benchmark. The problem itself is the software project scheduling (SPS), that consists on the assignment of employees to tasks in a software project in order to reduce its duration and cost [1,4]. This problem belongs to the domain of search based software engineering [7]. The software project scheduling is a realistic problem with capital importance in software factories.

An instance of the SPS problem specifies a set of employees, tasks, and skills to indicate which employee can participate in which task. For every employee, it is necessary to set his/her maximum dedication, salary, and skills. For a task, it has to be known an estimation of the effort required, the skills needed to accomplish it, and a list of tasks that are prerequisite of it.

A solution to the SPS problem is an assignment matrix that represents the degree of involvement of employees to tasks (cells in this matrix have values in set [0 1]). Such a solution has to meet all the constraints imposed by the problem.

The objectives pursued in this problem are to reduce the duration and cost of the software project and to fulfil the constraints.

To solve this problem with metaheuristics like CHC and GA we have to encode the assignment matrix as a binary string. This can be done as shown in Figure 2, where cell values are discretized using four bits ($[0\ 1] \rightarrow \{0,1\}^4$). Additionally, we need a fitness function to quantify the relative quality of solutions. The fitness function that we use is presented in Equation (1), and consists on a weighted sum of the project duration (p_{dur}), the project cost (p_{cost}), the number of tasks not covered by any employee (p_{ut}), the number of skills not covered for the tasks (p_{us}), and the amount of overwork done by the employees (p_{ow}).

$$\begin{aligned} fitness\ function = & +0.1 \times p_{dur} \\ & +(5.0 \times 10^{-6}) \times p_{cost} \\ & +p_{ut} + p_{us} + p_{ow} \end{aligned} \tag{1}$$

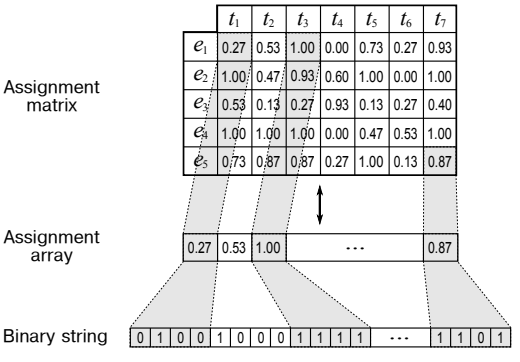

Fig. 2. Representation of an assignment matrix as a binary string

Table 1. Features of the problem instances in the benchmark. The size of the instance is the main indicator of its difficulty.

N	Size	Tasks	Employee
1	50	10	5
2	100	20	5
3	250	25	10
4	450	30	15
5	1250	50	25

In Table 1 we present the features of the 250 problem instances contained in the 5 test sets (50 instances per set). For every test set we show its identification number, the size of the contained instances, and the number of employees and tasks. All the instances share the same number of total skills, that is 10. The tasks of the instances require a random set of 4 to 6 skills. Additionally, employees have a random set of 2 to 4 skills. For a more precise understanding of the instances, we defer the interested reader to the original definition of this problem [4].

This benchmark is a large and wide set of instances since we want to actually deal with the problem class, not just with a few instances. Also, it will allow us to analyze algorithms at very different dimensions and difficulties, what will constitute a real challenge for any algorithm.

The objective of the optimization technique is to compute a solution with the lowest fitness value for every problem instance in the benchmark. The test sets have been arranged in increasing size or difficulty, where the first one has the smallest search space and the sixth the largest one. This arrangement of the test sets allows us to study trends of the algorithm with increasing size of the search

space. For the benchmark, we have used a set of problem instances created by the generator described at `http://mstar.lcc.uma.es`.

3.2 Parameter Settings

Instead of fixing an arbitrary set of parameters, we perform an initial configuration analysis to determine the best parameter for CHC. One random problem instance of every test set is used to tune up the algorithm during the configuration analysis. The parameters studied for CHC are the population size, the recombination probability (p_c), and the percentage of population restarted (p_r). The values studied for the three parameters are:

Population size: 64, 128, 256
Recombination probability (p_c): 0.5, 0.7, 0.9
Percentage of population restarted (p_r): 40%, 60%, 80%

After the analysis of the parameters we computed Table 2 to study their impact for CHC. This table contains the average fitness, its relative standard deviation (σ), and the difference between the highest and the lowest average fitness for the test sets 1, 2, and 4 given the values of the parameters. We performed 30 independent executions for the 27 different configurations for all the 150 instances in the three test sets. If we focus on the population size, we realize that this parameter has the highest impact in the results of the algorithm. As the population is increased, the results of CHC clearly improve (the fitness reaches lower values); but there is a point at which the improvement of the fitness is nonexistent or small enough not to justify a further increase of the population. This behavior depends on the problem instance: for small instances (test set 1) large populations involve no improvement but a time penalty, whereas for large instances (test set 4) large populations produce better results.

On the other hand, if we focus on the recombination probability (p_c) and the percentage of population restarted (p_r), we conclude that the average fitness and its relative standard deviation are almost the same for the test sets. This means that whatever the value we choose for these parameters, the average result of CHC will be almost the same. There is also a second reading for these results, and it is that CHC is a robust algorithm, even if the best values for the recombination probability and the percentage of population restarted are not properly chosen. For instance, once that the population size has been fixed to 256 individuals, the differences between the best and the worst average fitness for test sets 1, 2, and 4 are 0.33%, 0.54%, and 0.44% respectively, thus, the impact of p_c and p_r in the results is upper bounded by these values.

After these initial experiments we conclude that the values for the parameters of CHC that perform the best are 256 for the population size, 0.9 for the recombination probability, and 60% for the percentage of population restarted. As a summary, the parameters used to test and study CHC with the problem instances are listed in Table 3.

Table 2. Average fitness, relative standard deviation (σ), and difference between the best and worst average fitness for a fixed value of a CHC parameter

Parameter	Value	Fitness								
		Test set 1			Test set 2			Test set 4		
		Avg.	σ	Diff.	Avg.	σ	Diff.	Avg.	σ	Diff.
Pop. size	64	4.75	8.75%	0.69%	11.38	13.00%	1.18%	30.62	67.70%	2.11%
	128	4.48	8.00%	0.38%	10.35	9.88%	0.69%	11.33	48.25%	3.37%
	256	4.35	7.52%	0.33%	9.77	9.39%	0.54%	8.09	11.19%	0.44%
p_c	0.5	4.52	8.91%	9.37%	10.49	12.66%	15.81%	16.60	95.36%	136.33%
	0.7	4.52	8.94%	9.09%	10.50	12.80%	15.69%	16.71	95.16%	136.43%
	0.9	4.53	9.0%6	9.49%	10.51	12.89%	16.14%	16.74	95.22%	136.17%
p_r	40%	4.53	9.00%	9.25%	10.49	12.65%	15.60%	16.67	95.05%	136.69%
	60%	4.53	8.96%	9.26%	10.50	12.86%	15.83%	16.69	95.87%	136.54%
	80%	4.53	8.95%	9.54%	10.51	12.85%	16.08%	16.68	94.81%	135.41%

3.3 Discussion on the Results

Here we describe the results obtained after applying the CHC algorithm to solve the problem instances of the benchmark. The two aspects in which we focus are the fitness value and the execution time. On the one side, the fitness value quantifies the performance of the algorithm to allow future comparisons. On the other side, the execution time accounts the time it takes the algorithm to compute a solution. The execution time makes it possible to study how the CHC algorithm could behave on optimization problems as complex as this, and how does it scale when the size of the problem gets increased.

In Table 4 we show the results of CHC. The table presents the number of the test set in the benchmark and for it the fitness and time values. For the fitness we show its average value for all the instances in the test set. Also, we include the maximum (σ_{max}) and the average (σ) of the relative standard deviation. The maximum is the one of the instance in the test set which has the largest deviation; the average is that of the instances in the test set. The values were computed by running 30 independent executions for the 250 instances of the 5 test sets. In total we made 7500 independent runs to get lessons on the algorithm and the problem class instead of just on one instance or small problem study.

Table 3. Parameter settings for CHC

Parameter	Value
Max. number of iterations	500
Population size	256
Offspring size	256
Recombination probability	0.9
Recombination operator	HUX
Restarted population	60%
Selection strategy	Random
Replacement strategy	Ranking

Table 4. Experimental results for CHC

N	Fitness			Time (s)	
	Avg.	σ_{max}	σ	Avg.	σ
1	4.35	4.53%	2.50%	2.98	11.70%
2	9.77	7.37%	4.06%	8.64	12.23%
3	7.91	8.15%	3.62%	21.64	13.26%
4	8.01	11.00%	3.86%	38.06	7.68%
5	25.56	23.31%	12.60%	100.93	5.32%

The analysis of the fitness indicates that CHC is a stable algorithm that produces solutions that are similar in terms of quality. This means that even with a few executions CHC is capable of finding promising solutions for the instances in a robust manner. As we studied in the previous section, the quality of the results have more to do with the population size than with the recombination probability (p_c) or the percentage of population restarted (p_r).

Now, if we turn to the execution time we find that the larger the problem size, the longer it takes to finish the computation. Generally, we observe that the execution time of the algorithm is close to linear, even when the search space grows exponentially (Figure 4). This is a great point for CHC, because it means that we can expect the algorithm to solve even larger problem instances in an acceptable amount of time.

3.4 Comparison against GA

To put the results of CHC in a wider context, we compare it with a GA. GA is a metaheuristic inspired in biological evolution [8]. It codifies problem solutions as individuals subjected to an evolutionary process [2,6]. During each iteration the algorithm selects, recombines, and mutates individuals to evolve the population. As iterations go by, new individuals are computed with better solutions codified.

We use the classic formulation of GA: it combines the *single point crossover* (SPX) recombination operator, and the mutation operator that randomly modifies selected positions in the solution. For the configuration analysis we follow the same procedure as in CHC. The parameters that we consider in the initial analysis are the populations size, the recombination probability, and the mutation probability, whose candidate values are:

Population size: 64, 128, 256
Recombination probability: 0.5, 0.7, 0.9
Mutation probability: 0.01, 0.05, 0.1

Applying the same guidelines as with CHC, we conclude that the best values for the parameters are 256 individuals for the population, 0.7 for the recombination

Table 5. Parameters used for GA

Parameter	Value
Max. number of iterations	500
Population size	256
Offspring size	256
Recombination probability	0.7
Recombination operator	SPX
Mutation probability	0.1
Bit flip probability	0.01
Selection strategy	Random
Replacement strategy	Ranking

Table 6. Results of the experiments for GA

N	Fitness			Time (s)	
	Avg.	σ_{max}	σ	Avg.	σ
1	4.65	16.12%	6.26%	6.70	5.87%
2	11.54	27.58%	13.52%	14.84	4.30%
3	21.14	39.87%	19.91%	25.06	3.33%
4	59.16	28.61%	15.25%	39.16	3.92%
5	249.89	11.23%	6.32%	106.96	4.80%

probability, and 0.1 for the mutation probability. The settings finally used for GA in the experiments are listed in Table 5.

The results obtained using the GA are listed in Table 6. The table presents the number of the test set in the benchmark and for it the fitness and time values. For the fitness we show its average value for all the instances in the test set. Also, we include the maximum (σ_{max}) and the average (σ) of the relative standard deviation. The values were computed by running 30 independent executions for the 250 instances of the 5 test sets. Compared with CHC, GA produces worse solutions and it takes more time to compute them. We also realize that GA has serious problems for test sets 3, 4, and 5: the fitness, which has to be minimized, is on average 2.67, 7.31, and 9.78 times larger than in CHC. This comparison is shown in Figure 3, where the central mark is the median, the edges of the filled box the 25th and 75th percentiles, and the whiskers extend to the most extreme values. The results show that the GA is not an efficient algorithm to solve as large and difficult instances for the software project scheduling problem.

It is interesting to study the results for each test set independently. In Figure 3 we offer a graphical comparison of the fitness value for CHC and GA. We can see that for small size instances (test sets 1 and 2) CHC beats GA by a thin margin; on the other side, when the size of the instances get increased (test sets 3, 4, and 5), then CHC overcomes GA in a notorious way. It is also important to note that CHC requires fewer fitness evaluations to reach certain fitness value. While GA carried out 128256 evaluations for every instance in all the test sets, CHC performed by average 44500, 66995, 104861, 123427, and 128256 evaluations for test sets 1 to 5. This happens because of the incest prevention mechanism in CHC, that avoids the recombination of solutions that are similar to each other. For the comparison of the fitness, the Kruskal-Wallis test has been carried out to check if the differences in the algorithms are statistically significant. All the statistical tests are performed with a confidence level of 99%, and all of them have passed this tests.

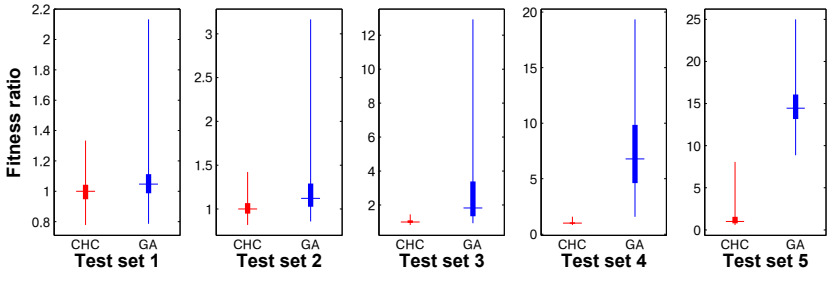

Fig. 3. Fitness comparison of CHC and GA for the test sets in the benchmark

In Figure 4 we show the execution time that it takes for CHC and GA to perform 500 iterations depending on the size of the instances in the test set. The sizes of the instances in the test sets are 50, 100, 250, 450, and 1250 respectively (Table 1). We see that the CHC algorithm always takes less time than GA

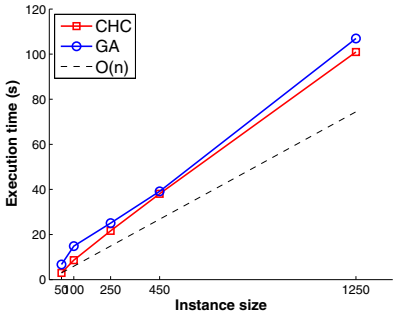

Fig. 4. Average execution time of the algorithms for increasing size problem instances in contrast with a theoretical linear time increment of complexity $O(n)$ (lower is better)

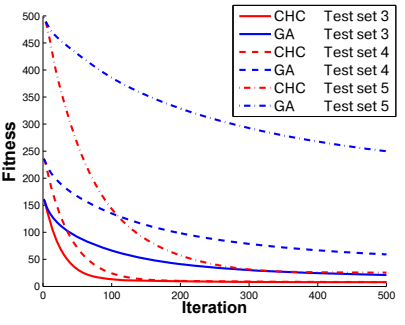

Fig. 5. Average fitness evolution through iterations (lower is better). The CHC algorithm reaches promising solutions in less iterations than GA for every test set.

to finish the computation. This is because CHC performs fewer fitness evaluations than GA, as stated previously. This means that CHC can solve the same instances than GA in less execution time.

To conclude, in Figure 5 we present the average fitness evolution of CHC and GA for the 500 iterations. We notice that the fitness value of CHC always remains below the fitness of the GA, no matter the test set. Thus, it is obvious that CHC converges faster to a promising solution. Globally, the GA needs more iterations to find promising solutions, and specifically, for test set number 5 it seems that 500 iterations are not even enough.

4 Conclusions

In this work we have presented a study on the performance of CHC when solving a benchmark of problem instances concerning software project scheduling. We focused on CHC because it has proven to be an efficient, fast, and powerful algorithm in the past, but still not well-known compared to other evolutionary algorithms. For the experiments, we faced the algorithm to a set of instances of the software project scheduling problem, that is a capital problem in software engineering. Finally, we compared the results achieved by CHC with a GA.

The analysis of the results obtained allows drawing some conclusions on the behavior of CHC. For instance, the population size was the parameter of the algorithm which had the highest impact in the results for the benchmark. Particularly, once that the population size was fixed to 256 individuals, the improvement produced by the variation of the probability of recombination and the percentage of population restarted was at most 0.54% (small, in comparison). This means that CHC is a robust algorithm that produces good results with a wide set of values for the parameters.

Regarding the comparison of CHC with GA, the CHC algorithm beats the GA in every single test set for both: in quality of the solutions and in execution time. The CHC algorithm always produce better solutions than the GA, and the larger the instance the better the result of CHC compared with the GA. Additionally, the execution time of CHC is always shorter than the execution time of the GA. In relation with this, CHC not only needs less time to find a promising solutions but also it takes less iterations to reach it. As a consequence, we can suggest that CHC is a better algorithm than the GA for this problem.

Acknowledgements. Authors acknowledge funds from the Spanish Ministry MICINN and FEDER under contracts TIN2008-06491-C04-01 (M* `http://mstar.lcc.uma.es`) and TIN2011-28194 (roadME) and CICE, Junta de Andalucía, under contract P07-TIC-03044 (DIRICOM `http://diricom.lcc.uma.es`).

References

1. Alba, E., Chicano, F.: Software project management with GAs. Information Sciences 177(11), 2380–2401 (2007) (in press)
2. Back, T., Fogel, D.B., Michalewicz, Z.: Handbook of Evolutionary Computation, 1st edn. IOP Publishing Ltd., Bristol (1997)
3. Bilbao, M., Alba, E.: CHC and SA applied to wind energy optimization using real data. In: IEEE Congress on Evolutionary Computation, pp. 1–8 (2010)
4. Chicano, F., Luna, F., Nebro, A.J., Alba, E.: Using multi-objective metaheuristics to solve the software project scheduling problem. In: Proceedings of the 13th Annual Conference on Genetic and Evolutionary Computation, GECCO 2011, pp. 1915–1922. ACM, New York (2011)
5. Eshelman, L.: The CHC adaptive search algorithm: How to have safe search when engaging in nontraditional genetic recombination. In: Rawlins, G. (ed.) Foudations of Genetic Algorithms, pp. 265–283. Morgan Kaufmann (1990)
6. Goldberg, D.E.: Genetic Algorithms in Search, Optimization and Machine Learning, 1st edn. Addison-Wesley Longman Publishing Co., Inc. (1989)
7. Harman, M., Jones, B.F.: Search-based software engineering. Information & Software Technology 43(14), 833–839 (2001)
8. Holland, J.H.: Adaptation in natural and artificial systems (1975)
9. Nebro, A.J., Alba, E., Molina, G., Chicano, F., Luna, F., Durillo, J.J.: Optimal antenna placement using a new multi-objective CHC algorithm. In: Proceedings of the 9th Annual Conference on Genetic and Evolutionary Computation, GECCO 2007, New York, NY, USA, pp. 876–883 (2007)
10. Nesmachnow, S., Cancela, H., Alba, E.: Heterogeneous computing scheduling with evolutionary algorithms. Soft Computing, 685–701 (2010)
11. Nesmachnow, S., Cancela, H., Alba, E.: Evolutionary algorithms applied to reliable communication network design. Engineering Optimization 39(7), 831–855 (2007)
12. Nesmachnow, S., Cancela, H., Alba, E.: A parallel micro evolutionary algorithm for heterogeneous computing and grid scheduling. Appl. Soft Comput. 12(2), 626–639 (2012)