Memory-based Memetic Algorithms

Marco Wiering
Intelligent Systems Group
Institute of Information and Computing Sciences
Utrecht University
marco@cs.uu.nl

Abstract

Memetic algorithms combine genetic algorithms with local search; each time a new individual is generated by the genetic algorithm, a simple local search procedure brings it (closer) to a local maximum. Although memetic algorithms have been shown to work well for particular combinatorial optimization problems, they may sometimes suffer from early convergence to a local maximum. This paper describes (steady-state) memory-based memetic algorithms, which search more efficiently by increasing the diversity of the population. Each time a new individual is created, it is brought to its local maximum using local search, and then the algorithm checks whether the individual has already been found before. If that is the case, the lowest possible fitness value is assigned, so that the individual will be replaced during the next iteration. The experiments compare memory-based memetic algorithms to memetic algorithms, genetic algorithms and multiple-restart local search on deceptive problems. The results indicate that the memory-based memetic algorithm finds the global optimum much more often than the normal memetic algorithm, and performs about the same as genetic algorithms on the chosen test problems which are very difficult for conventional local search algorithms.

1 Introduction

Memetic algorithms (Radcliffe and Surry, 1994; Merz and Freisleben, 1997) combine genetic algorithms (GAs) (Holland, 1975; Goldberg, 1989) with local search (LS). Often local search is applied to every individual until it cannot be improved anymore by the local search operators, although some researchers have shown benefits on particular problems by only partially updating each individual or only updating some individuals (Hart, 1994). Local search can be very efficient for solving particular (simple) problems, but usually suffers from quickly becoming trapped in local maxima. Multiple restart local search algorithms such as GRASP (Feo and Resende, 1995) deal with this problem by iteratively constructing random starting points after which local search brings the individuals to their local maxima. However, for large state spaces with many local maxima, the use of random starting points may be far from efficient. Therefore memetic algorithms evolve a population of individuals in which genetic operators are used to create new individuals which are all brought (closer) to their local maximum using local search. This may provide us with much better candidate starting points for local search to improve.

Comparisons between memetic algorithms and other algorithms. Memetic algorithms have already been compared to GAs on a number of combinatorial optimization problems such as the traveling salesman problem (TSP) (Radcliffe and Surry, 1994) and experimental results indicated that the memetic algorithms found much better solutions than standard genetic algorithms. Memetic algorithms have also been compared to the Ant Colony System (Dorigo et al., 1996), (Dorigo and Gambardella, 1997) and to Tabu Search (Glover and Laguna, 1997) and results indicated that memetic algorithms outperformed both of them on the Quadratic Assignment Problem (Merz and Freisleben, 1999). Thus, memetic algorithms provide us with a powerful method for solving complex problems, and more research on these promising algorithms should be done to find their advantages and disadvantages compared to other algorithms. For this reason, we perform experiments with deceptive problems

containing many local maxima which makes them difficult for local search and memetic algorithms.

Memory-based memetic algorithms. One of the problems of local search, as stated above, is that they usually end up in local maxima. By keeping a population of individuals, memetic algorithms tend to cope with that problem, but it may happen that the population converges quite fast to a single individual which takes over the whole population. Although in this case, the algorithm may of course be restarted or additional random solutions may be inserted in the population, we opt for another approach. In our approach, we preserve diversity by keeping all found (local maxima) solutions in memory. When we generate a new individual and bring it to its maximum using local search, we check whether we already have seen this solution, and it this is the case, we assign a fitness of 0 (the worst possible fitness value) to the individual. In this way, we do not keep the same solution in the population a long time.[1] By using hash-tables, checking whether we already have found an individual can be efficiently implemented, since operations like storing an individual with hash-tables consumes as much time as a mutation or crossover operator. Note that although we could use the complete updated hash-table for representing the population and for generating offspring, we keep track of a separate limited population which is evolved.

Steady state algorithms. We will use steady state algorithms, in which at each time step we produce one offspring, bring it to its local maximum, check whether we already found this solution, and if not we store it in memory, and we replace the worst individual in the population by the newly found solution. In this way, the population always contains the best individuals found so far, and the population is maximally diverse (i.e., it is not likely to contain any duplicates). Using the crossover operator we hope to be able to fruitfully combine individuals in the population to generate new ones.

[1]It can happen that the same solution is stored twice, but one of these will be replaced during the next time-step.

Coping with large memory requirements. We only use the memory implemented in the hash-table to assign a fitness of 0 to already found solutions, we never use the memory to select parents for recombination. Note also that by only storing local maxima, we strongly reduce the size of the memory. Although for very large problems involving many local maxima, the storage space will become very large, it will grow less fast than the computational time needed to generate them. Furthermore, since we use steady-state algorithms, we do not have to store individuals below the lowest fitness of an individual in the population, thereby making the hash-tables much smaller and more memory efficient.

Outline of this paper. We will discuss memetic algorithms in section 2. In section 3, we describe memory-based memetic algorithms. Experimental results comparing memetic algorithms, memory-based memetic algorithms, genetic algorithms, and local search on deceptive problems are presented and discussed in section 4. Finally, section 5 concludes this paper.

2 Memetic Algorithms

Memetic algorithms combine genetic algorithms with local search. Memetic algorithms are inspired by memes (Dawkins, 1976), pieces of mental ideas, like stories, ideas, and gossip, which reproduce (propagate) themselves through a population of meme carriers. Corresponding to the selfish gene idea (Dawkins, 1976) in this mechanism each meme uses the host (the individual) to propagate itself further through the population, and in this way competes with different memes for the limited resources (there is always limited memory and time for knowing and telling all ideas and stories).

Memetic evolution. The difference between genes and memes is that the first are inspired by biological evolution and the second by cultural evolution. Cultural evolution is different because Lamarckian learning is possible in this model. That means that each transmitted meme can be changed according to receiving more information from the environment. This makes it possible to locally optimize each different meme before it is transmitted to other individuals. Although optimization of trans-

mitted memes before they are propagated further seems an efficient way for knowledge propagation or population-based optimization, the question is how we can optimize a meme or individual. For this we can combine genetic algorithms with different optimization methods. The optimization technique we use in this paper is a simple local hillclimber, but others have also proposed different techniques such as Tabu Search. Because we use a local hillclimber, each individual is not truly optimized, but only brought to its local maximum. If we would be able to fully optimize the individual, we would not need a genetic algorithm at all.

First-visit local hillclimbing. We use local hillclimbing on each newly created individual. This local hillclimber is a first-visit method; the first change which improves the individual is used. We use binary strings for the individuals in our experiments and use the simplest neighbourhood function as possible which contains only individuals generated by a mutation of a single bit. Of course, this is an important choice, but we did not want to optimize the neighbourhood function, since this would require a-priori knowledge or a lot of testing from which only the memetic algorithms could profit and not the genetic algorithms to which they are compared. Furthermore, the best used neighbourhood size seems to be very dependent on the specific chosen test-problems. The local hillclimber starts with a random bit of an individual and examines whether changing this bit improves the fitness of the individual. Then it goes to the next bit, etc. After it tried out all bits of the individual, it checks whether it has made at least one improvement, and if so it continues to try to change bits and otherwise it stops. Note that the local hillclimber may need many evaluations to change a single individual. Therefore, the use of local hillclimbing may not always be very effective. However, for a problem such as one-max which favours individuals having more 1's in the bitstring, the local search method will lead to an optimal result in a number of evaluations given by the length of the individual, which is very fast indeed.

The algorithm. The algorithm is shown below. First an initial population of individuals is created. Then local hillclimbing is applied to all individuals. After this, two parents are selected for recombination and mutation is applied to the new individual. Then local hillclimbing is applied to the new individual and the worst individual is replaced by the new individual.

Memetic Algorithm

```
1) Make a population of random
   individuals.
2) ∀ individuals i do:
   3) Ind(i) = Local-Hillclimbing(Ind(i))
4) Parent1 = Select-Parent(Population)
5) Parent2 = Select-Parent(Population)
6) Offspr = Crossover(Parent1,Parent2)
7) Offspr = Mutate(Offspr)
8) Offspr = Local-Hillclimbing(Offspr)
9) Replace worst individual by Offspr
10) If termination criterion not met
    goto 4.
```

3 Memory-based Memetic Algorithms

Although memetic algorithms can be quite efficient on their own, the whole population may converge quite quickly to a (bad) local maximum. The reason is that the same building blocks can be quickly found in all individuals, and mutation to escape local maxima may not be very useful in combination with memetic algorithms, since the mutated parts have to be brought to a local maximum each time again. In our experiments on deceptive problems we found that using mutation operators with memetic algorithms does not work well, since it just costs of lot of evaluations and harmful mutations are most likely. Therefore we do not use mutation at all with our memetic algorithms. To compensate for this, we use big populations which are likely to contain the required genetic material. But without mutation, the problem of early convergence is even bigger, and therefore we have to cope with lack of diversity using another method.

Maximally diverse populations. In our approach we store all found solutions using a hash-table. After we created a new offspring and we brought it to its local maximum using local hillclimbing, we check whether the solution has already been found before. If that is the case, we assign a fitness value of zero to this individual (but still replace the worst individual in the population by this individual). In

this case, the new individual will have the lowest fitness in the population and is replaced by the next offspring immediately, and thus only different individuals are incorporated in the population. On one hand we keep all local maxima in the hash-table, and therefore there is no loss of information. On the other hand, only the best individuals are in the population and used for recombination, thereby making evolution more efficient. Thus, the memory combined with the steady state memetic algorithm allows us not only to keep maximally diverse populations which do not contain copies of individuals, it also allows us to store all best (local maxima) individuals found so far during an experiment in the population.

Memory-based Memetic Algorithm

```
1) Make population of random
   individuals.
2) ∀ individuals i do:
  3) Ind(i) = Local-Hillclimbing(Ind(i))
  4) If Ind(i) is in Memory
       assign fitness 0 to it,
     Else Store Ind(i) in Memory.
5) Parent1 = Select-Parent(Population)
6) Parent2 = Select-Parent(Population)
7) Offspr = Crossover(Parent1,Parent2)
8) Offspr = Mutate(Offspr)
9) Offspr = Local-Hillclimbing(Offspr)
10) If Offspr is in Memory
    assign fitness 0 to it,
   Else Store Offspr in Memory.
11) Replace worst individual in
    Population by Offspr
12) If termination criterion not
    met goto 5.
```

The algorithm. The algorithm is shown above. First an initial population of individuals is created. Then local hillclimbing is applied to all individuals. All different individuals are stored in memory, and solutions which have already been found before will receive a fitness of 0 (the lowest possible fitness value). After this, two parents are selected for recombination and mutation is applied to the new individual with a specific probability (which we set to 0 in our experiments). Then local hillclimbing is applied to the new individual. Again the solution is stored in memory or if it already was in memory it receives a fitness of 0. Finally, the worst individual is replaced by the new individual.

Time and space requirements. Since we use a hash-table, the time requirements for this method are not different from conventional genetic algorithms. The cost for checking whether an individual has already been found before and the cost of inserting a new individual in the hash-table is equal to the number of bits in the individual. Thus, these operators are just as fast as the use of mutation or crossover operators, and the use of memory does not slow things down. In the most general case, the storage space requirements grows with the number of different found individuals during a run, but since we only store local maxima, the storage space grows less fast than the time needed for running an experiment. Still, for very large experiments, the needed memory may not fit anymore in computer memory, so we would have to manage the hash-table in some other way. Since we use steady state algorithms and each time remove the worst individual in the population, we do not need to store any (new) individuals in the hash-table below the lowest fitness value of an individual in a population, since they will be immediately replaced anyway. This is therefore a very efficient implementation of the steady-state memory-based memetic algorithm. Note that if we would not use a hash-table, but immediately check whether a new individual's solution was already in the population, time requirements would become much larger.

4 Experiments

We compare genetic algorithms, memetic algorithms, and memory-based memetic algorithms on four different experiments with deceptive problems which contain many local maxima, and therefore makes search for optimal solutions difficult, especially for local search based algorithms. All evolutionary algorithms are steady-state algorithms to make comparisons clearer. We also compare these three algorithms to multiple restart local hillclimbing. In our deceptive problems, the building blocks are separable, so we do not need to use any genetic linkage learning, and we can (for example) use 1-point crossover, that we used in the simulations and works well for the generated problems.

We run experiments on deceptive trap functions of different building block length and different individual (bitstring) length. A trap function (Goldberg et al., 1992) of size n, called trap-n, divides the total individual of length l in $\frac{l}{n}$ different building blocks of size n. For each building block, the highest fitness of 1 is given if all bits are 0, but if one bit is 1 and the others are 0, the fitness is the lowest and equals 0. After this, with s bits set to 1, the fitness increases with increasing $s > 1$, until it reaches a local deceptive maximum at $s = n$. Figure 1 shows how the fitness of a single building block for the two different trap functions of our experiments is computed. The total fitness of an individual equals the sum of the fitness values over all building blocks.

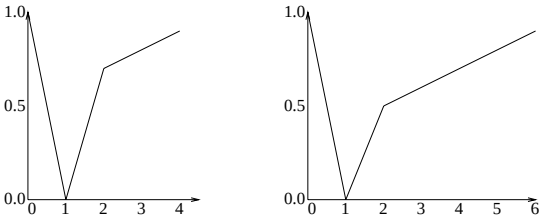

Figure 1: The fitness values of a building block in a trap-4 and trap-6 problem is determined by the number of ones in the building block. The fitness is maximal with only 0's, but a deceptive maximum arises with only 1's.

4.1 Experiments: trap-4 functions

Experimental set-up. We first performed experiments on trap-4 functions with total individual lengths of 72 and 108 bits. After a search through parameter space, we decided to use the following experimental parameters for the methods. The genetic algorithm (GA) used a population size of 8000 for the 72-bit problem and 12000 for the 108-bit problem. We noted that the GA needed a very large population to work well. The crossover probability is 1.0 and the mutation probability is 0.02 for the 72-bit problem and 0.01 for the 108-bit problem. The memetic algorithm (MA) used a population size of 900 for the 72-bit problem and a population size of 1200 for the 108-bit problem, a crossover probability of 1.0 and no mutation. The memory-based memetic algorithm (MBMA) used the same parameters as the normal memetic algorithm. For selecting the two parents for recombination in the three evolutionary algorithms we used tournament selection with tournament size 4. The multiple random restart local search (LS) algorithm does not use any parameters. We let all methods execute for 200,000 evaluations for the 72-bit problem and for 330,000 evaluations for the 108-bit problem.

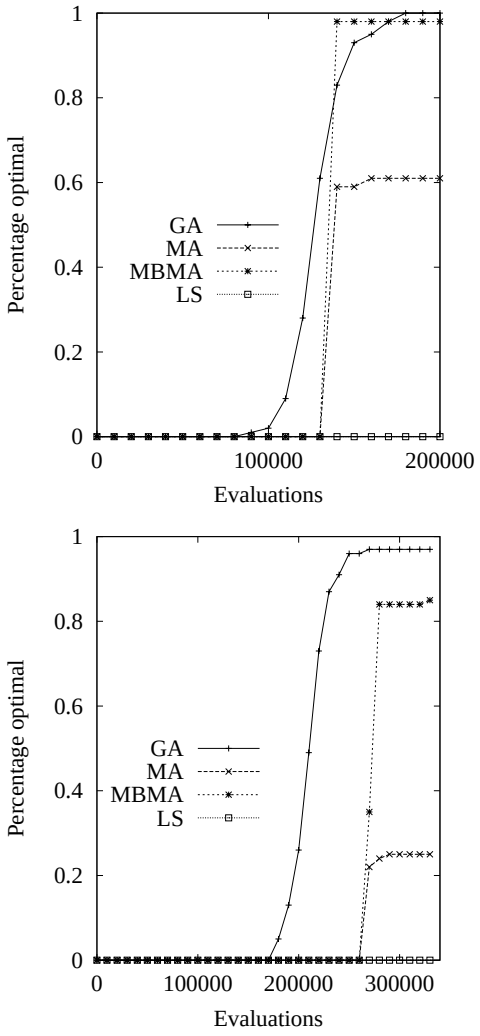

Figure 2: (A) The experimental results on the Trap-4 function of 72 bits. (B) The experimental results on the Trap-4 function of 108 bits. The figures show the percentage of simulations in which the optimum (all 0's) has been found. The total number of simulations with each method is 100.

Experimental results. The results are

shown in figures 2(A) and 2(B). The figures show the percentage of simulations (out of 100) that the optimal solution has been found for individual length 72 and 108.

We can see that the genetic algorithm outperforms the other methods on the trap-4 functions. The memory-based memetic algorithm significantly outperforms the normal memetic algorithm, and multiple restart local hillclimbing is not able to find any optimal solution.

4.2 Experiments: trap-6 functions

The parameters are the same as the ones used for trap-4 functions. The results are shown in figures 3(A) and 3(B)..

We can see that for the trap-6 function, the memory-based memetic algorithm performs significantly better than all other algorithms. The genetic algorithm comes as second best, and the local hillclimbing method again does not find the optimal result at all.

4.3 Discussion

The overall experimental results show that the genetic algorithm and the memory-based memetic algorithm perform best; they find the optimal result in more than 90% of the simulations on the smaller problems and more than 60% on the larger problems (given the maximum number of evaluations). We have to say that the genetic algorithm profited from the large population, with smaller populations the results were much worse. Thus, it seems that a steady-state genetic algorithm can work quite well on deceptive problems if the population size is large. The memetic algorithm performs worse than the genetic algorithm, often the population converges too fast and there is not any progress anymore. We did not use mutation on our experiments with memetic algorithms, since the probability of mutating a whole building block consisting of only 1's to a building block of only 0 or 1 1's is very small, and harmful mutations are much more likely in combination with local search (i.e. mutating a building block of only 0's to a building block containing at least 2 1's is much more probable). Instead of using mutation, we used a fairly large population, and using this, the memory-based memetic algorithm performs very well. The optimal building blocks should all be in the initial population, and therefore if there are many local maxima we need to use a fairly large population size. If we could afford to use larger populations the results of the memory-based memetic algorithms would of course improve, but this is costly due to the local search procedure which is executed on the initial population. The figures of our experimental results clearly show the long initial phase used for local hillclimbing on the large initial population. As soon as recombination is used, the performance quickly increases. Therefore there might be advantages by using

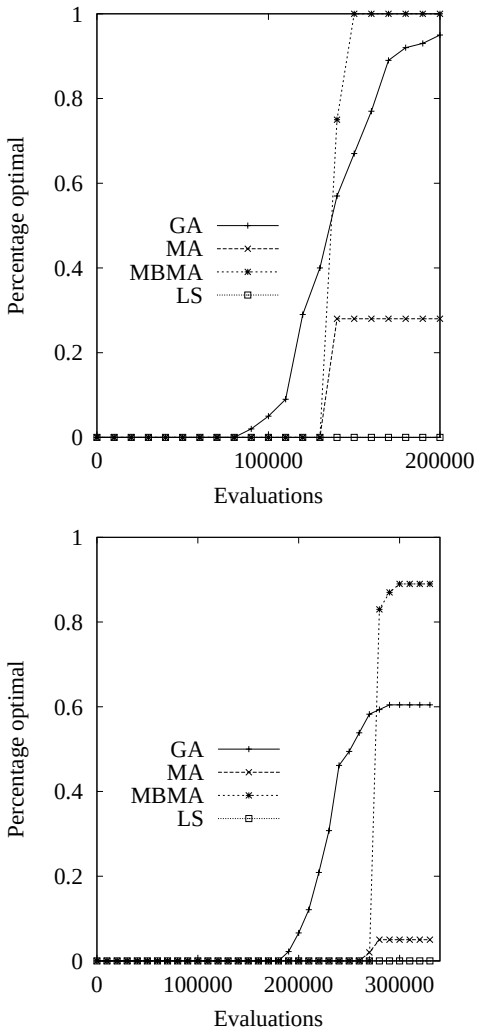

Figure 3: (A) The experimental results on the Trap-6 function of 72 bits. (B) The experimental results on the Trap-6 function of 108 bits. The figures show the percentage of simulations the optimum has been found. The total number of simulations with each method is 100.

adaptive population sizes for (memory-based) memetic algorithms.

Although we cannot afford to use such large populations as used by the genetic algorithm, the memory-based memetic algorithms perform also very well and clearly outperform the normal memetic algorithms. Since we keep maximum diversity the memory-based memetic algorithm can find the optimal result in most of the runs in all experiments.

If we compare the genetic algorithm and the memory-based memetic algorithm, we can see that the genetic algorithm performs better on the trap-4 than on the trap-6 function, whereas the memory-based memetic algorithm performs better on the trap-6 function. Thus, it seems that for difficult deceptive problems with large building blocks, the memory-based memetic algorithm may be an effective candidate, as long as the initial population can be chosen big enough to contain all the necessary building blocks. If all local building blocks can be found, recombining the solution-parts is done very effectively.

5 Conclusion

We introduced memory-based memetic algorithms, which use memory in memetic algorithms to keep maximal diversity in the population. The method stores all found individuals in a hash-table and each time a new individual is computed it is checked whether the individual has already been found before. If that is the case, the individual receives a fitness of 0 (the lowest possible). In this way, memory-based memetic algorithms can keep maximal diversity, which is not the case for normal memetic algorithms. We combine the use of memory with steady-state memetic algorithms which always replace the worst individual, which provides us with a very efficient implementation for using memory.

We compared the new approach to memetic algorithms, genetic algorithms, and local hill-climbing with multiple restarts on four different deceptive problems. The results showed that the memory-based memetic algorithm and the genetic algorithm performed much better than the other algorithms. The genetic algorithms performed best when the locally deceptive building blocks of the problem were smaller, whereas the memory-based memetic algorithm performed best for larger building blocks.

In future work we want to study adaptive population sizes for the (memory-based) memetic algorithms, since they seem to suffer much from the long initial phase of local hill-climbing on the initial population. We also want to get more insight in the trade-off in the time spent by the local search algorithm compared to the global search GA algorithm. Furthermore, we want to combine linkage learning algorithms such as BOA (Pelikan et al., 1999) with memory-based memetic algorithms. Finally, we want to use memory-based memetic algorithms on combinatorial optimization problems. Since memetic algorithms have been shown to outperform genetic algorithms for solving combinatorial optimization problems in a number of experiments, we expect that memory-based memetic algorithms may even perform better.

References

R. Dawkins. 1976. *The Selfish Gene*. Oxford University Press.

M. Dorigo and L. M. Gambardella. 1997. Ant colony system: A cooperative learning approach to the traveling salesman problem. *Evolutionary Computation*, 1(1):53–66.

M. Dorigo, V. Maniezzo, and A. Colorni. 1996. The ant system: Optimization by a colony of cooperating agents. *IEEE Transactions on Systems, Man, and Cybernetics-Part B*, 26(1):29–41.

T. Feo and M. Resende. 1995. Greedy randomized adaptive search procedures. *Journal of Global Optimization*, 6:109–133.

F. Glover and M. Laguna. 1997. *Tabu Search*. Kluwer Academic Publishers.

D. E. Goldberg, K. Deb, and J. Horn. 1992. Massive multimodality, deception, and Genetic Algorithms. In *Proceedings of Parallel Problem Solving from Nature*, pages 37–46.

D. E. Goldberg. 1989. *Genetic Algorithms in Search, Optimization, and Machine Learning*. Reading, MA: Addison-Wesley.

W. E. Hart. 1994. *Adaptive Global Optimization with Local Search*. Ph.D. thesis, University of California, San Diego.

J. H. Holland. 1975. *Adaptation in Natural and Artificial Systems*. University of Michigan Press, Ann Arbor.

Peter Merz and Bernd Freisleben. 1997. A genetic local search approach to the quadratic assignment problem. In Thomas Bäck, editor, *Proceedings of the Seventh International Conference on Genetic Algorithms (ICGA97)*, San Francisco, CA. Morgan Kaufmann.

Peter Merz and Bernd Freisleben. 1999. A comparison of memetic algorithms, tabu search, and ant colonies for the quadratic assignment problem. In Peter J. Angeline et al., editor, *Proceedings of the Congress on Evolutionary Computation*, volume 3, pages 2063–2070, 6-9.

M. Pelikan, D.E. Goldberg, and E. Cantu-Paz. 1999. BOA: The Bayesian optimization algorithm. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO-99)*, volume 1, pages 525–532.

Nicholas J. Radcliffe and Patrick D. Surry. 1994. Formal memetic algorithms. In *Evolutionary Computing, AISB Workshop*, pages 1–16.