An implementation of a generic memetic algorithm for the edge biconnectivity augmentation problem*

Fatos Xhafa

Departament de LSI

Universitat Politècnica de Catalunya

Jordi Girona, 1-3, 08034 Barcelona, Spain

fatos@lsi.upc.es

Abstract

In this paper we present an implementation of a generic memetic algorithm for the edge bi-connectivity augmentation problem –the problem of augmenting a given graph by a cheapest possible set of additional edges in order to make the graph edge bi-connected. This problem is known for its applications to communication network design –the extension of an existing communication network to become robust against single link failures– as well as in VLSI floor planning. We provide a C++ implementation of a generic memetic algorithm for the problem, as a good alternative for approximately solving it. We use known benchmarks in the literature for the problem as to experimentally evaluate how good the generic memetic algorithm works for the problem.

1 Introduction

The edge bi-connectivity augmentation problem (E2AUG), introduced by Eswaran and Tarjan [1], is the problem of augmenting a given graph by a cheapest possible set of additional edges in order to make the graph edge bi-connected. We recall that a graph is edge bi-connected, if at least two edges

*This research was partially supported by the ALCOM FT Research Project No. IST-1999-1 4186 and CICYT Project TRACER TIC2002-04498-C05-03.

need to be removed in order to separate the graph into disconnected components. More formally, in E2AUG we are given an undirected, connected graph $G = (V, E)$ and a disjoint set A of edges between nodes in V. Each edge $e \in A$ has associated a postive cost and can be used to augment graph G. The objective is to find a subset $S \subseteq A$ of edges with minimum total costs such that the augmented graph $G_S = (V, E \cup S)$ is edge bi-connected.

E2AUG is of relevance to both theory and industrial applications attracting thus the attention of researcher to study its computational complexity and to find methods for solving it. Eswaran and Tarjan [1] showed, for the general case, the E2AUG to be NP-complete and gave a polynomial time algorithm for the special case of the E2AUG where all edge costs are equal and $G_A = (V, E \cup A)$ is a complete graph. (The reader is referred to Khuller [4] for a general survey on related graph-connectivity problems and algorithms.) Regarding the applications of the E2AUG, we can cite the design of communication networks and VLSI floor planning.

Given the *hardness* of the problem to optimally solve it, researchers have sought to solve it suboptimally in a reasonable amount of time. The first efforts towards this direction were done to obtain efficient approximation algorithms with guaranteed approximation factor (Frederickson and Jájá [2], Gabow et al. [3], Khuller and Thurimella [5] and Zhu et al. [10, 11]). More recently, researchers have focused in implementing meta-heuristics such as genetic algorithms, evolutionary algorithms, hybrid algorithms etc. for the problem. These heursitic methods, despite of lack of guarantee on the quality of solutions, in general tend to provide good solutions and are very efficient thus allowing to cope in practice with real size instances of the problem. We distinguish here the results of Ljubić and Raidl [7, 6, 9] obtained in several implementations of evolutionary and genetic algorithms for the E2AUG.

In this paper we provide an implementation of a generic memetic algorithm (MA) for the E2AUG. To the best of our knowledge, such algorithm has not been previously reported for the problem though the evolotionary algorithms of Ljubić and Raidl [6, 9] are similar in spirit with memetic algorithms. Our implementation is derived from a generic memetic algorithm –a *template* for MAs– thus reducing the effort of the implementation and encouraging the reusability of the template for other combinatorial optimization problems. We use known benchmarks in the literature for the problem [10] as to experimentally evaluate how good the generic MA works for the problem.

The paper is organized as follows. We present in Section 2 a generic memetic algorithm –the basis for our implementation. Implementation issues of MA template are given in Section 3. The instantiation of MA template for E2AUG and some experimental results are given in Section 4. Finally, in Section 5 we give some conclusions about our results together with some remarks on ongoing and future work.

2 A generic memetic algorithm

A generic memetic algorithm –a *template* for MAs– based on local search was proposed by Moscato in [8]. We give its pseudocode in Fig. 1.

```
begin
    initializePopulation $Pop$ using FirstPop();
    foreach $i \in Pop$ do $i$ := Local-Search-Engine($i$);
    foreach $i \in Pop$ do Evaluate($i$);
    repeat /* generations loop */
        for $j := 1$ to $\#recombinations$ do
            selectToMerge a set $S_{par} \subseteq Pop$; $offspring$ := Recombine($S_{par}$);
            $offspring$ := Local-Search-Engine($offspring$); Evaluate($offspring$);
            addInPopulation $offspring$ to $Pop$;
        endfor;
        for $j := 1$ to $\#mutations$ do
            selectToMutate $i \in Pop$;
            $i_m$ := Mutate($i$); $i_m$ := Local-Search-Engine($i_m$); Evaluate($i_m$);
            addInPopulation $i_m$ to $Pop$;
        endfor;
        $Pop$ := SelectPop($Pop$);
        if $Pop$ meetsPopConvCriteria then $Pop$ := RestartPop($Pop$);
    until termination-condition=True;
end
```

Figure 1: The memetic algorithm template

This *template* is made up of several methods and entities. Some methods are problem-independent –their implementation doesn't depend on the problem– and some others are problem-dependent. In the first group we have methods like **addInPopulation** and in the later group, we have methods like `Mutate()` whose implementation depends on the problem. Further, there are different entities, some of them *explicit* to the template like Population, Individual and some others that remain *implicit* to the template such as Problem, Solution. Of a special interest is the method `Local-Search-Engine()` that applies to the individuals of the population to improve them. In a certain sense, this method is external to the template, and clearly it plays an important role in the implementation of the MA. In general, for a given problem, we can dispose several implementations of `Local-Search-Engine()` therefore when implementing MAs, it is quite desirable to find the one that gives better results without changing the rest of the implementation. This is possible in terms of the MA template due to its genericity and flexibility.

Finally, the template uses a set of *setup* parameters, whose numerical values loosely depend on the problem to be solved, such as ***#mutations***, that need to be specified, usually through a fine tuning process.

3 Implementation of the MA template

We can easily observe that the MA template defines the main method of the (generic) memetic algorithm through other methods/entities that are either problem-dependent or problem-independent. The problem-independent methods/entities are implemented without any knowledge of the problem being solved while the problem-dependent ones need to be implemented according to problem definition and data structures chosen for representing it. In this sense, we can see the MA template as an *algorithmic skeleton* some parts of which need to be *"filled in"* for any concrete problem to be solved.

Clearly, the implementation offers a separation of concerns: the problem-independent part is *provided* by the skeleton while the problem dependent-part is *requiered*, i.e. the user has to provide it in instantiating the skeleton for a concrete problem. In order for the two parts to *communicate*, the skeleton fixes the interface of both problem-independent and problem-dependent methods/entities in such a way that the problem-independent part uses the problem-dependent methods/entities through their signature/interface and, vice-versa, the problem-dependent methods can be implemented without knowing the implementation of the MA template itself (see Fig. 2). This separation has several advantages such as flexibility, reusability etc.

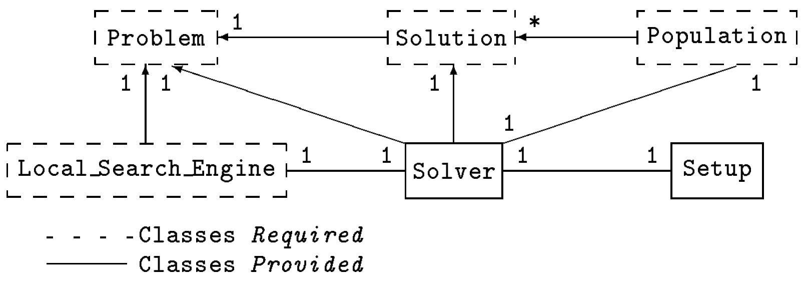

Figure 2: Class diagram of MA

In the diagram, the class `Solver` is in charge of the main method of MA and uses other methods and entities. Thus the implementation of the MA template consists in fully implemented classes `Solver` and `Setup` and fixed interfaces for the rest of the classes whose implementation will be latter completed by the user for a concrete problem. The interfaces for the problem-dependent entities are obtained through a careful abstraction process by taking into account the necessities of the user while instantiating a concrete problem. Observe that in our implementation we have considered an unstructured `Population`, i.e. simply a container of individuals –feasible solutions– without any relation between them.

4 Instantiation of the MA template for E2AUG.

To instantiate the MA template for E2AUG we just need to instantiate the problem-dependent methods/entities according to their interfaces included in the skeleton implementation. For simplicity, we will show the pseudocode of the methods (some of them are based on methods of Raidl and Ljubić [9]).

We use the following definitions and notation through the pseudocode. We call the edges of E, *fixed edges* and those of A *non fixed edges*. For any non fixed edge e, $Q(e)$ denotes the set of fixed edges covered by e, that is, the path in graph[1] G that connects the vertices of e. For any fixed edge e, $Cov(e)$ is the set of non fixed edges that cover e. Finally, for each fixed edge e and a solution S, $n_{\text{cov}}(e)$ dentotes the number on non fixed edges belonging to S that cover e.

Many of the problem-dependent methods use a procedure `LocallyImprove()` that consists in eliminating redundant edges from a feasible solution S while mantaining the feasibility of S (see Fig. 3).

```
LocallyImprove($S$)
begin
  foreach $e_0 \in E$ do $n_{\text{cov}}(e_0) := 0$;
  foreach $e \in S$ do
    foreach $e_0 \in Q(e)$ do $n_{\text{cov}}(e_0) := n_{\text{cov}}(e_0) + 1$;
  endfor;
  $T := S$;
  do
    select $e \in T$ via tournament selection;
      (prefer more expensive edges);
    $T := T \backslash \{e\}$;
    if $\forall e_0 \in Q(e) : n_{\text{cov}}(e_0) \geq 2$ then
      $S := S \backslash \{e\}$;
      foreach $e_0 \in Q(e)$ do $n_{\text{cov}}(e_0) := n_{\text{cov}}(e_0) - 1$;
    endif;
  while $T \neq \emptyset$;
  return S;
end
```

```
FirstPop()
begin
  $Pop := \emptyset$;
  for $i := 1$ to $\#pop\_size$ do
    $S :=$ LocallyImprove($A$);
    $Pop := Pop \cup \{S\}$;
  endfor;
  return $Pop$;
end
```

Figure 3: Procedure LocallyImprove and FirstPop method

We proceed now, in turn, with the problem-dependent methods for E2AUG.

`FirstPop`. The procedure LocallyImprove() can be used in a straightforward way generate an initial population by invoking it with $S = A$. The diversity of the population is assured due to the randomness at processing the edges (see Fig. 3).

`Recombine`. Given S_1 and S_2, two feasible solutions, the recombination is done by simply applying LocallyImprove() to $S_1 \cup S_2$ (see Fig. 4).

[1]Due to results of Frederickson et al. [2] the problem of augmenting a connected graph is reduced to that of augmenting a tree, therefore the input graph is a tree.

```
Recombine($S_1$, $S_2$)
begin
  $S := S_1 \cup S_2$;
  $S :=$ LocallyImprove($S$);
  return $S$;
end
```

```
Mutate($S$)
begin
  do $n_{\mathrm{mut}}$ times
    choose $e \in S$ randomly;
    $S := S \backslash \{e\}$;
    foreach $e_0 \in Q(e)$ do $n_{\mathrm{cov}}(e_0) := n_{\mathrm{cov}}(e_0) - 1$;
    foreach $e_0 \in Q(e)$ s.t. $n_{\mathrm{cov}}(e_0) = 0$ in random order do
      select $e' \in Cov(e_0) \backslash \{e\}$ (prefer cheaper edges);
      $S := S \cup \{e'\}$;
      foreach $e'_0 \in Q(e')$ do $n_{\mathrm{cov}}(e'_0) := n_{\mathrm{cov}}(e'_0) - 1$;
    endfor;
    $S :=$ LocallyImprove($S$);
  enddo;
  return $S$;
end
```

Figure 4: Recombine and Mutate methods

`Mutation`. The purpose of this method is to modify the solutions in order to prevent the population from a premature convergence by introducing edges from A not present in some individual of the population (see Fig. 4).

```
RestartPop($Pop$)
begin
  $Pop := \{\mathrm{Best}(Pop)\}$;
  for $i := 1$ to $\#pop\_size - 1$ do
    $S :=$ LocallyImprove($A$);
    $Pop := Pop \cup \{S\}$;
  endfor;
  return $Pop$;
end
```

```
Local-Search-Engine($S$)
begin
  /* $T$ is the best current neighbour */
  foreach $e \in S$ do
    foreach $e' \in (A - S)$ s.t. $c(e) > c(e')$ do
      if $\forall e_0 \in Q(e)(n_{\mathrm{cov}}(e_0) > 1 \vee e_0 \in Q(e'))$ and
      $(S - \{e\} + \{e'\})$ is better than $T$ then
        $T := S - \{e\} + \{e'\}$;
    endfor
  endfor;
  $S := T$ and update $n_{\mathrm{cov}}$;
  return $S$;
end
```

Figure 5: RestartPop and Local-Search-Engine methods

`RestartPop`. This method generates a new population, as FirstPop() method, but preserving the best solution of previous population (see Fig. 5).

`Local-Search-Engine`. The objective of this method is to improve the solution by changing un edge of the solution by another not in the solution that yields to a better solution (in case of different choices the cheapest edge is chosen) (see Fig. 5).

Experimental evaluation. We have drawn by now the first experimental results in order to evaluate the quality of implementation and the choice of parameters of MA for E2AUG. These results would allow us to experimentally evaluate how good the MA template works for the E2AUG. We have fixed two parameters to be mesuared through the experiments: the executation time and the cost of the obtained solution.

The design of the experiment is done with the aim to compare our results with those of the *adhoc* implementation of Raidl and Ljubić reported in [9]. To this end we have run our program with the same execution times as in [9] and the problem instances to run the implementation are taken as well from [9] (Raidl and Ljubić obained these instances by an instance generator of [10]). We have limited ourselves to instances that are known as most difficult ones, classified into five groups: M1, N1, N2, R1 and R2. Each group itself consists of 30 instances. The characteristics of each group are given in Fig. 6. The experimentation also tries to find how does the population size influences in the cost of the solutions, for the same execution time.

Regarding the AM's parameters (see Fig. 6), their numerical values are fixed after a previous experimentation. Se experimentará con poblaciones de tamaño 10, 30 y 50. El número de recombinaciones será la mitad de la población y el de mutaciones el 20% de la población.

Group	$\|V\|$	$\|A\|$	$c(e)$
M1	70	290	[1,2415]
N1	100	1104	[10,50]
N2	110	1161	[10,50]
R1	200	9715	[1,100]
R2	200	9745	[5,100]

Population size	10,30,50
#recombinations	5,15,25
#mutations	2,6,10
k_{locimp}	5
k_{mut}	4
n_{mut}	5

Figure 6: Instance groups (left) and AM's parameter values (right)

The execution times for the instances of each group, are given in Fig. 7 (left) where we also show the number of generations needed to achieve the given executation time according to the population size. The machine configuration under which we run the implementation is given in Fig. 7 (right).

Group	t(s)	#generations		
		pop. size 10	pop. size 30	pop. size 50
M1	1.2	12	2	-
N1	10.4	170	23	8
N2	11.3	155	13	4
R1	135.3	1200	45	11
R2	135.3	1200	45	11

Processor	AMD K6(tm) 3D proc. (450 MHz)
Main Mem.	256 Mb
OS	Debian (Linux 2.4.19)
Software	LEDA, gcc

Figure 7: Setup parameters (left) and machine configuration (right)

Given that the execution may lead to different outputs for the same input instance and configuration (due to the randomly taken decisions), the execution is repeated 25 times for the same input and configuration therefore we report the average cost of the best solutions encountered.

The experimental results are summarized in Table 1, where for each group and population size we give: the average cost of the best solutions found, the deviation w.r.t. the best known cost and the average execution time. The

Table 1: Experimental Results

Group	$c(S^*)$	Raidl et al.		Pop. size 10		Pop. size 30		Pop. size 50	
		dev.	t(s)	dev.	t(s)	dev.	t(s)	dev.	t(s)
M1	2940.0	0.00%	1.2	18.1%	1.3	18.1%	1.3	-	-
N1	383.0	0.47%	10.4	7.7%	10.7	7.7%	11.2	10.2%	11.1
N2	429.0	0.00%	11.3	3.7%	11.1	5.5%	11.0	8.0%	10.1
R1	121.4	0.00%	135.3	8.2%	146.2	7.0%	131.3	8.2%	144.4
R2	320.5	0.67%	218.5	14.0%	141.3	11.7%	139.2	14.3%	144.4

column $c(S^*)$ shows the best known cost and the third and forth columns refer to results of Raidl and Ljubić reported in [9].

Analysis of the results. The results obtained from our implementation of the *generic MA* do not match the results of the *ad hoc* implementation of the evolutionary algorithm of Raidl at al. In general, it is acceptable that generic implementations do not report better results than those of *ad hoc* implementations since ad hoc implementations manage to embed much more knowledge of the problem into the implementation than generic implementations do. The generic implementations, on the other hand, offer advantages proper to generic programming paradigm such as reusability, flexibility etc. We should remark, however, that we could not settle, by now, an appropriate comparison between the two implementations due to the following observations: (a) we have run our implementation 25 times for each instance while Raidl at al. have run 100 times; (b) we do not dispose of machine configuration used by Raidl at al.; (c) we tried to maintain the same execution times as those of Raidl at al. although we have observed that, in most cases, the best solutions were reported early due to population convergence. The worst results of our implementation are obtained for instance group M1 (due to small execution time the MA is far from convergence) and for instance group R2 (our execution time is much inferior to the other implementation –roughly 80s less.)

Furthermore, the results of MA implementation depends a lot on the proper parameters such as population size. As we can see from the results, for the instance groups M1, N1 and N2 the best results are obtained for population size equal to 10 while for instance groups R1 and R2 the best results correspond to population size equal to 30. This observation indicates that the same configuration is not appropriate for different instance groups, therefore, we need a careful fine tuning specific to each instance group. The experimentation shows also the expected relationship between the instance size and population size: for bigger instance size, bigger population size is more appropriate to the price of less generations.

5 Ongoing and future work

We have presented an implementation of a generic memetic algorithm for E2AUG problem; to the best of our knowledge, such implementation has not been previously reported. By now, we have concluded a first experimental evaluation of our generic implementation, the results are good though they do not match the results of known ad hoc implementations for the problem.

The experimental results and the log execution files we have maintained show us the efficiency of the implementation but also indicate us that we need a better fine tuning of parameters, especially those of MA algorithm. Thus we have observed that population size and number of generations are crucial to the quality of results since inappropriate values causes algorith to make useless work. The appropriate values will be fixed through a complete experimenting.

Another direction we plan to improve is the implementation of Local-Search-Engine procedure. We have observed that, unfortunately, right now this procedure in several cases do no yields improved soltuions or wastes more time than necessary for improvement. Thanks to the genericity of our MA implementation we can deal with this problem by providing a repository of implementations for Local-Search-Engine that would allow us to plug in the most appropriate implementation into the MA implementation.

References

[1] K.P. Eswaran and R. E. Tarjan. Augmentation problems. SIAM J. on Comp., 5(4):653-665, 1976.

[2] G.N. Frederickson and J. Jájá. Approximation algorithms for several graph augmentation problems. SIAM J. on Comp., 10(2):270-283, 1981.

[3] H.N. Gabow, Z. Galil, T. Spencer, and R.E. Tarjan. Efficient algorithms for finding minimum spanning trees in undirected and directed graphs. Combinatorica, 6(2):109-122, 1986.

[4] S. Khuller. Approximation algorithms for finding highly connected subgraphs. In D. Hochbaum, ed., Approximation Algorithms for NP-hard Problems, 236-265. PWS Publishing, Boston, MA, 1996.

[5] S. Khuller and R. Thurimella. Approximation algorithms for graph augmentation. J. of Algorithms, 14(2):214-225, 1993.

[6] I. Ljubić and G.R. Raidl. An evolutionary algorithm with stochastic hill-climbing for the edge-biconnectivity augmentation problem. In E. Boers et al. eds., Applications of Evol. Comp., vol. 2037 of LNCS, 20-29. Springer, 2001.

[7] I. Ljubić, G.R. Raidl, and J. Kratica. A hybrid GA for the edge-biconnectivity augmentation problem. In K. Deb et al. eds., Parallel Problem Solving from Nature VI Conference, vol. 1917 of LNCS, 641-650. Springer, 2000.

[8] P. Moscato. Memetic Algorithms: A short introduction. *New Ideas in Optimization*, D. Corne et al. (Eds.), McGraw-Hill, 219-234, 1999.

[9] G.R. Raidl and I. Ljubic. *Evolutionary local search for the edge-biconnectivity augmentation problem.* Information Processing Letters, Elsevier, 82, pp. 39-45, 2002.

[10] A. Zhu. A uniform framework for approximating weighted connectivity problems. B.Sc. thesis, University of Maryland, MD, May 1999. 17.

[11] A. Zhu, S. Khuller, and B. Raghavachari. A uniform framework for approximating weighted connectivity problems. In Proc. of the 10th ACM-SIAM Symposium on Discrete Algorithms, 937-938, 1999.