EvoGeneSys, a New Evolutionary Approach to Graph Generation

L. Cordella[a], C. De Stefano[b], F. Fontanella[b,*], A. Marcelli[c]

[a]*Dipartimento di Informatica e Sistemistica*
Università di Napoli Federico II
Via Claudio 21, 80125 Naples – Italy
[b]*Dipartimento di Ingegneria Elettrica e dell'Informazione*
Università di Cassino e del Lazio meridionale
Via G. Di Biasio 43, 03043 Cassino (FR) – Italy
[c] *Dipartimento di Ingegneria Elettronica ed Ingegneria Informatica*
Università di Salerno
Via Ponte don Melillo
84084 Fisciano (SA) – Italy

Abstract

Graphs are widely used to represent complex and structured information of interest in various fields of science and engineering. When using graph representations, problems of special interest often imply searching. For example, searching for the prototypes representing a dataset of graphs or for the graph that optimizes a set of parameters. In any case, it is necessary that the problem solution be expressed in terms of graphs. Therefore, defining effective methods for automatically generating single graphs, or sets of graphs, representing problem solutions, is a key issue. A new evolutionary computation–based approach specifically devised for generating graphs is presented. The method is based on a special data structure, called *multilist*, which allows the encoding of any type of graph, directed or undirected, with or without attributes. Graph encoding by multilists makes it possible to define effective crossover and mutation operators, overcoming the problems normally encountered when implementing genetic operators on graphs. Further advantages of the proposed approach are that it does not require any problem specific knowledge and it is able to search for graphs whose number of nodes is not known a priori. Three sets of experiments were performed to test the proposed approach and the solutions found were compared with those obtained by other approaches proposed in the literature.

Keywords: evolutionary algorithm, evolutionary graph generation, genetic operators, graph encoding

1. Introduction

It is well known that graphs can be used effectively to represent complex and structured information. However, processing large graphs is generally a high computational cost problem. In the last few years, mainly because of computer technology developments, there has been an increasing interest in studying and using graphs in many applications. Graphs may effectively represent physical networks, such as transportation systems, power systems, and mobile communication infrastructures [1, 2, 3], but have been also used to model less tangible interactions, as might occur in ecosystems, databases or in the control flow of computer programs [1]. Graphs have proved to be suited for modeling complex patterns in Pattern Recognition and Machine Vision, in terms of parts and their relations. Attributes of graph nodes and arcs are often added to incorporate further information, leading to a graph representation generally known as Attributed Relational Graph (ARG) [4, 5]. Examples of successful applications include shape analysis and 3-D object recognition [6, 7], character recognition [8], classification of ideograms and symbols in document analysis and technical drawing interpretation [9].

Independently of the specific application considered, using graph representations often requires that the problem solution be expressed in terms of graphs: for instance, in the case of pattern recognition applications, the solution to be searched for may be a set of graphs representing the prototypes associated with each class to be recognized, whereas in communication infrastructure applications the solution may be a single graph representing the best network configuration. Therefore, defining effective methods for automatically generating single graphs, or sets of graphs, representing problem solutions, is a key issue.

The above problem can be reformulated as a search problem in which graphs represent solutions in a search space. As a consequence, effective techniques to explore such a space and to generate graphs representing tentative solutions are required. To this purpose, two main different approaches have been proposed in the literature, according to the nature of the problem: in the case of applications in

*Corresponding author
Email addresses: cordel@unina.it (L. Cordella), destefano@unicas.it (C. De Stefano), fontanella@unicas.it (F. Fontanella), marcelli@unisa.it (A. Marcelli)

which examples of the solutions to be modeled are available, the graphs may be generated by exploiting the information held by a training set of samples. In all the other cases, solutions are searched for by defining a function f able to measure the goodness of the solutions in the given search space: graphs representing solutions are found by looking for maxima of f. Combinatorial, heuristic and inductive learning approaches have been used, among the others, to generate graphs [10, 11].

In this framework, Evolutionary Computation (EC) techniques have also been used because they make it possible to explore complex high-dimensional search spaces effectively [12]. Their use requires the definition of techniques for graph encoding, in order to represent solutions in a way suitable for genetic operators.

Graph encoding schemes can be grouped in two categories: direct and indirect [12]. The former scheme uses data structures in which each graph element, node or arc, is directly associated with a particular element of the structure. An example of this scheme is the adjacency matrix, in which a graph g having n nodes is represented by an $n \times n$ matrix M, whose element $M(i, j)$ represents the arc connecting the i–th and j–th node of g. The latter scheme, instead, includes methods in which a graph g is represented by the set of rules to be used for building it. Genetic Programming (GP), for instance, can be adopted to generate a tree structure encoding such rules.

Most of the methods proposed in the literature use direct encoding schemes since they provide a simple and natural way of representing graph structures. One of the most successful applications using a direct encoding scheme is presented in [13], in which an ANN is optimized and used to play checkers. EC-based methods that use a direct encoding have been also proposed in the fields of molecular design [14], electrical circuit design [15] and network design [16, 17]. In this latter field, a direct encoding scheme has been specifically devised for representing tree structured solutions, without using any crossover operator.

It is worth noting that these methods adopt graph encoding techniques specifically devised for the task taken into account, whose properties cannot be easily generalized. The success of this kind of encoding, although it could result inefficient in some cases, is due to its simplicity and convergence properties as shown in [18, 19], while its main drawback is due to the complexity of implementing genetic operators in the general case. In fact, implementing a crossover operator without imposing any particular constraint to the graph structure requires the choice of more than one cut point for each graph, in order to obtain two separate sub-graphs, and of a way for reassembling redundant or missing arcs.

More recently, in [20] real valued vectors have been used to directly encode tree–structured graphs. The aim is to use standard genetic operators for exploring tree search spaces. In particular, a simple crossover operator has been defined, which does not require searching for multiple cut points. The main drawback of this method, however, is that it cannot be easily generalized for dealing with any kind of graphs. In [21] a geometric crossover is presented for solving multiway graph partitioning problems. This method was devised for solving a specific class of problems, namely the optimal way of partitioning a given graph in a fixed number of sub–graphs, and cannot be used for evolving graphs representing solutions in a given search space.

Indirect encoding was developed for reducing both the size of graph representations, and the complexity of the genetic operators. In [22], for instance, sets of developmental rules are used for encoding graphs to solve various size encoder/decoder problems. Although some good results was obtained, the method has shown some limitations because it often needs to predefine the number of rewriting steps and it is unable to find detailed connectivity topologies among graph nodes [12].

In the last decade, Genetic Programming techniques have been also used for evolving graphs [23]. In this context, a graph is indirectly encoded into an individual by means of a program tree. The graph is then built by executing the program encoded by the tree, which contains various functions for creating components or modifying topology. This approach was used in [24] for evolving analog electronic circuits: the reported results are satisfactory even if the system requires a huge amount of computational resources. In [25], GP was used for evolving bond graphs, a modeling tool employed to design multi-domain dynamic systems such as analog filter circuits and typewriter drives. Furthermore, in [26] GP was used to generate graphs representing wireless networks. It is worth noting that in all the mentioned GP–based systems the size of the trees encoding the obtained solutions are not quoted, making it difficult to evaluate the effectiveness of GP–trees in case of medium and large size graphs.

Even if indirect encoding gave satisfactory results in a number of applications, its main drawback is that genetic operators are not applied to graphs, but to the rules generating graphs. This choice makes easier the implementation of genetic operators, but it also produces two undesired effects: on the one hand, genetic operators generally exhibit a small locality, since it is very difficult to ensure that small changes in the structure representing a graph result in small changes in the corresponding graph, making it difficult to maintain the discovered solutions during the evolution process; on the other hand, even small changes in a graph, needed to obtain detailed topologies, may require complex changes of the rules, which cannot be easily produced by the genetic operators, thus reducing the search space exploration effectiveness.

Within the EC framework, in the Genetic Programming (GP) field, graphs have also been used as an alternative way of representing programs. In fact, while the standard GP approach uses trees for representing programs, some researchers have proposed to represent programs by means of graphs [27, 28, 29, 30, 31, 32, 33, 34]. It is worth noticing that the just mentioned approaches have been specifically devised for evolving programs, and seem hardly exportable to other fields. Actually, as far as we know, they have not been applied for different purposes. On the contrary, the aim of this paper is to present a general purpose approach for graph generation. Its possible application fields are not limited to evolving programs, but range from optimization problems to Pattern Recognition and Bayesian networks.

Moving from the above considerations, in [35] we presented a preliminary study of an EC–based system for evolving general purpose graphs, whose number of nodes is not a priori known. The work presented here is an extension of that study in which a new version of EC–based system, called *EvoGeneSys* (Evolutionary Graph Generation System), is proposed. In this study, the properties of the genetic operators have been better investigated, the evolutionary algorithm has been reformulated and further experiments have been performed.

Our approach uses a direct graph encoding scheme to explore the search space effectively, and adopts a specifically devised data structure, called *multilist*, which allows graphs to suitably represented so as to overcome the above discussed problems in implementing genetic operators. In a multilist, attributes can be associated with both nodes and arcs, thus allowing either simple graphs or ARG's to be represented.

This data structure has proved particularly convenient for defining a general purpose crossover operator, which can generate graphs of variable size. In the graph domain, crossover involves splitting parent graphs and merging the resulting parts in order to obtain offspring. As previously mentioned, a graph cannot be split into two subgraphs by choosing a single cut point since, generally, more than one arc has to be broken, while the subgraphs to be merged may have a different number of arcs to be reattached: this is the reason why most of the crossover operators presented in the literature [14, 15] imply complex searching procedures on the graphs. The crossover operator presented here, on the contrary, requires the choice of a single cut point on each multilist and allows automatic determination of the arcs to be reattached during the merging process. Furthermore, a mutation operator has been also defined, which modifies the multilist in such a way as to generate a new graph whose node number is unchanged, whereas both node and arc attributes may be different. In Section 3.4, it will be shown that the proposed mutation operator may affect also the number of arcs. As regards the computational complexity, the application of our genetic operators implies a computational cost which is quadratic with respect to the number of nodes in the individual.

The approach devised was tested on three kinds of problems. In the first one, the ability of the system to generate graphs with variable number of nodes and arcs was tested. The second problem taken into account involves the configuration of a wireless network, where the solution to be found is represented by a graph in which nodes correspond to access points, and arcs to wired connections between access points. The results were compared with those obtained by using the GP-based approach for graph generation presented in [26]. The third test involved the OneMax-tree problem [36], in which a target spanning tree must be found. The results obtained were compared with those obtained by a different tree representation scheme [36].

The paper is organized as follows: Section 2 introduces the multilist data structure and illustrates how graphs are encoded by multilists; section 3 describes the proposed evolutionary algorithm for evolving graphs and provides a detailed description of the genetic operators; in Section 4 the experimental results are reported, while some discussions and concluding remarks are left to Section 6.

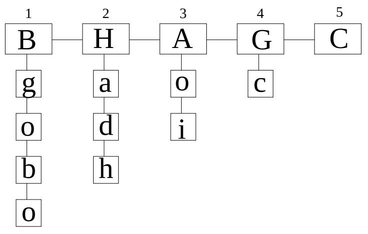
2. Graph Encoding by Multilists

Let G be an Attributed Relational Graph[1] of N nodes. Nodes and arcs in G have attributes respectively belonging to the sets A_n and A_a. The data structure devised for representing ARG's is called *multilist* (ML in the following) since it is based on the list concept and consists of $(N + 1)$ lists. The first one, called *main list*, represents graph nodes with their attributes, thus its number of elements is equal to the number N of nodes in G. In the following, N will be referred to as *size* of both the graph and the multilist representing it. The remaining lists are called *sublists*. Each sublist is associated with one node of G and includes the attributes of a subset of the arcs connected to that node (see Fig. 1).

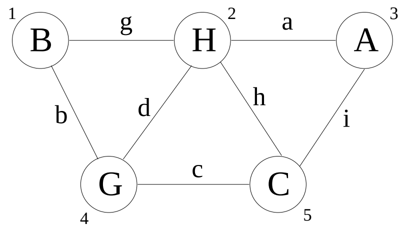
In order to preserve information about the nodes interconnected by each arc, arc attributes are suitably sorted in each sublist. Namely, the i-th sublist contains information on the arcs connecting the i-th node of the graph to the nodes following it in the main list, in the order they appear in such list. If two nodes are not connected, this information is anyway suitably stored in the proper place of a sublist. In practice, a *NULL* relation has been defined so that even the absent arcs are encoded in the ML representation of a graph. For this reason, the length of the sublist associated with a node regularly decreases as the position of that node in the main list increases: the first sublist is made up of (N-1) elements, the second sublist has (N-2) elements and so on. In fact, the information on the link between each node and those preceding it in the main list is already expressed in the preceding sublists. As a consequence, the sublist of the last node of the graph is void. Thus, the ML has a triangular shape: the base of the triangle is the main list and has length equal to the size N, while the height is represented by the first sublist and has length $(N - 1)$.

Note that here, in order not to make the paper dull reading, only simple (i.e., without self-loops) undirected graphs are considered, thus the relation linking the i-th

[1]A formal definition of ARG can be found in the Appendix.



(a) A generic ML



(b) The encoded graph

Figure 1: In (a) a generic ML. The horizontal list is the main one, while the vertical lists are the sublists. The elements of the set A_n are denoted by capital letters, while those of A_a are denoted by small letters. The NULL element is denoted by the letter 'o'. In (b) the graph encoded by the ML. For each node its position in the main list is also indicated.

node to the j-th node coincides with the relation linking the j-th node to the i-th node. This choice does not limit the generality of the treatment, since considering directed graphs only requires adding a second attribute for each sublist element in order to specify arc direction.

The proposed encoding scheme uses a positional notation: considered a sublist element of a given node n, the other node possibly connected to it is determined by both the position of such an element within the sublist, and the position of the node n within the main list. In practice, the i-th element of the j-th sublist contains information on the arc connecting the i-th node to the $(j+i)$-th node. In the following it will be shown that this encoding technique exhibits some properties that will be useful to simplify graph split and merge.

It is worth noting that the proposed encoding scheme is not free from the permutation problem [37]. Namely, there is a many-to-one mapping from the MLs (genotype) to the actual graph (phenotype). In fact, two different node orderings of the same graph (see fig. 2(a) and 2(b)) give place to two different MLs (see fig. 2(c) and 2(d)).

The main consequence of the permutation problem is that it may reduce the effectiveness of the crossover operator. In fact, if the crossover is applied to different MLs that represent the same graph, the offspring are likely to contain two copies of some nodes and none of the others. For this reason, in some cases (e.g., [19], in the framework of an ANN architecture optimization task) the crossover was not used, but only a mutation operator was adopted. On the other hand, it has been shown that the crossover operator may result very effective in increasing evolution efficiency [38]. Moreover, Hancock [37], again with reference to ANN architecture optimization, showed the advantage of using the crossover operator, in spite of the permutation problem. The defined crossover operator proved effective for our purposes, making it also possible to generate individuals of different size.

3. The Evolutionary Algorithm

The ML data structure has been used for developing an evolutionary algorithm able to evolve graphs. According to our approach, each individual in the evolving population is an ML representing a graph. MLs may be different sizes, thus allowing exploration of a search space of graphs with a variable number of nodes. As anticipated in the previous sections, the ML data structure proved particularly convenient for implementing both *mutation* and *crossover* genetic operators. While the definition of a mutation operator is relatively simple, independently of the adopted data structure, the definition of a general purpose crossover operator is much more difficult. In fact, the crossover operator is easy to implement for data structures such as strings or trees, because in these cases it is simple to select a single cut point for splitting an individual into two parts. On the contrary, in case of graphs, splitting an individual into two subgraphs is much more complex since, generally, more than one arc has to be broken, while the subgraphs to be merged may have a different number of arcs to be reattached. The crossover operator presented here, exploiting the fact that the ML arranges the nodes of a graph in a linear structure (the main list), makes it possible to define effective criteria for automatically determining both arcs to be broken during splitting and arcs to be reattached during merging. In order to describe these criteria, let us introduce two basic operations required by the crossover operator, namely *t-cut* and *merge*.

3.1. t-cut operation

Given a multilist L of size N, representing a graph G having N nodes, the *t-cut* operation $(1 \leq t < N)$ divides L in a *left* multilist and a *right* multilist, respectively containing the first t nodes and the remaining $(N-t)$ ones. The effect is that of dividing the original graph G into two *sub–graphs*, let us say G_1 and G_2, respectively encoded by the left multilist and the right multilist. The sublist of each node in the left multilist contains a subset of elements representing the connections between that node and each of the nodes included in the right multilist. Note that such "redundant" connections represent the arcs that have been broken in G, except for those encoding the NULL relation. The number of redundant connections is the same for every node and will be called *redundancy degree* (Rd) of the multilist, which, on its turn, will be denominated

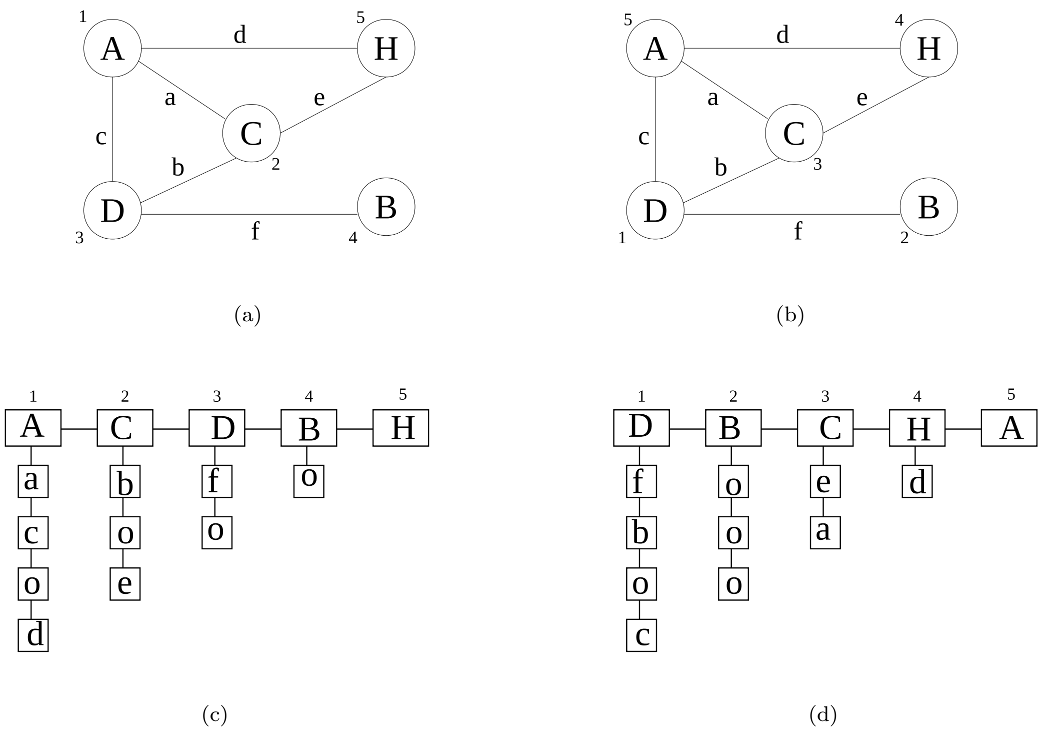


Figure 2: Effects of node order permutation: two different node ordering of the same graph (top) give place to different MLs (bottom).

redundant. The graph G_1 containing broken arcs will also be denominated *redundant*.
On the contrary, the right multilist cannot contain redundant connections and represents the subgraph made of the last $(N - t)$ nodes of G. Such multilist will be denominated *complete*. Note that the corresponding graph G_2 is canonical.

Considering, for instance, the multilist shown in fig. 3(a) representing the graph shown in fig. 3(b), the *t-cut operation* with $t = 3$ yields the multilists illustrated in fig. 3(c) and 3(d). The corresponding graphs are shown in fig. 3(e) and 3(f).

3.2. Merge Operation

The *merge* operation generates a new multilist by joining a redundant multilist and a complete multilist. Let us denote with L_1 and L_2 the multilists to be merged. Let us assume that L_1 is redundant, is made of N_1 nodes and represents a graph G_1 with broken arcs. Let us also assume that L_2 is complete, is made of N_2 nodes and represents a canonical graph G_2.

The merge operation is obtained by carrying out the following steps:

1. generate a new multilist L by appending the main list of L_2 to the main list of L_1;
2. compare the redundancy degree Rd_1 of L_1 with the size N_2:
3. three different cases may occur, according to the result of this comparison:

- if $Rd_1 = N_2$ (see fig. 4) the resulting multilist L is complete, the corresponding graph G is canonical and no further actions are needed.
- if $Rd_1 < N_2$ (see fig. 5) the resulting multilist L is a data structure in which some connections between nodes are missing. In fact, the first N_1 elements in the main list of L represent a graph G_1 where each node may have at most Rd_1 broken arcs, thus allowing merging with at most Rd_1 nodes. This implies that each sublist of the first N_1 nodes holds information about the connections with Rd_1 nodes of L_2, while no information is stored about the connections with the remaining $\alpha = N_2 - Rd_1$ nodes of L_2. These α nodes occupy the last positions in the main list of L. Therefore, in order to make L a complete multilist, it is necessary to add α new elements to each sublist of the first N_1 nodes. Note that to minimize the arbitrariness of the merging process, only NULL connections are added: in this way, no new arcs are added and the obtained graph G contains only the arcs that were present in both G_1 and G_2.
- if $Rd_1 > N_2$ (see fig. 6) the resulting multilist L is a data structure in which some connections between nodes are redundant; as previously discussed, in fact, the first N_1 elements in the main list of L hold information about the connections with Rd_1 nodes to be merged.

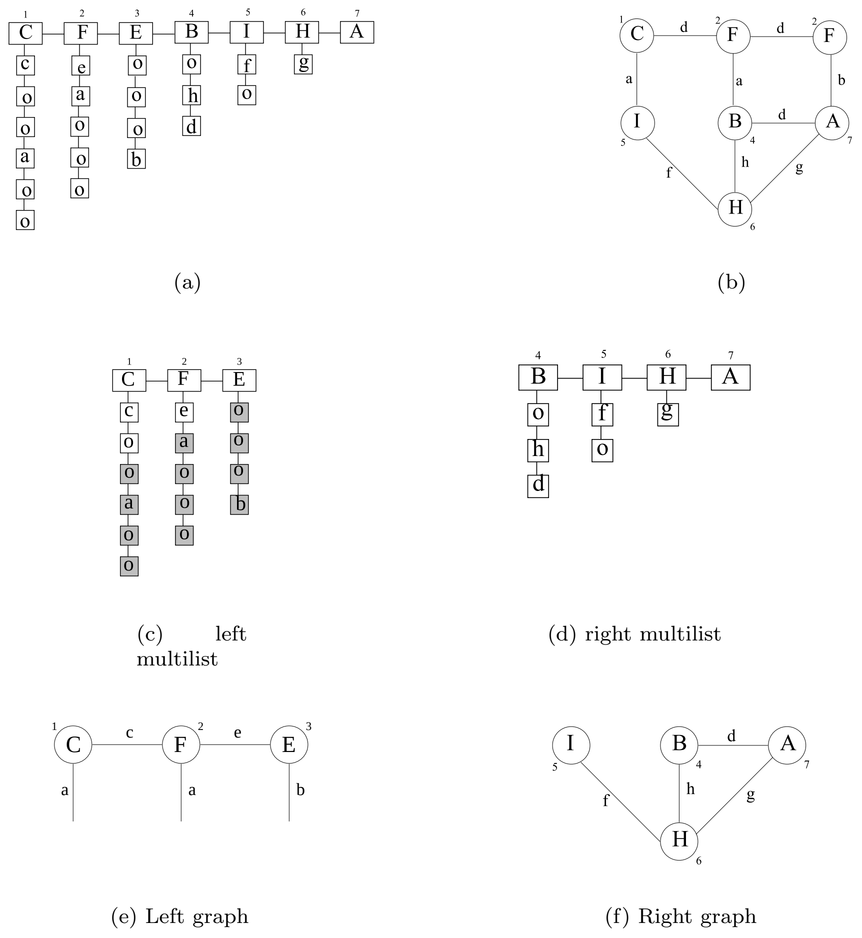


Figure 3: A generic ML (a) and the encoded graph (b). The two multilists (c) and (d) generated by the 3-cut operation applied to the multilist (a); redundant connections are shaded. The subgraphs encoded by the above multilists (e) and (f).

Since L_2 has less than Rd_1 nodes, each sublist of the first N_1 nodes contains $\beta = Rd_1 - N_2$ redundant elements. Therefore, in order to make L a complete multilist, β elements in each sublist of the first N_1 nodes must be deleted. It is worth noting that a complete multilist L might also be obtained by appending β new randomly generated nodes in its main list: following this approach, however, would lead to increasing the variability of the obtained multilist, since new nodes would be added belonging neither to G_1 nor to G_2.

3.3. Crossover Operator

The crossover operator allows us to generate two new individuals (*offspring*) by swapping parts of two individuals (*parents*) previously selected in the current population. In our case, individuals are multilists representing graphs, thus the effect in the graph space (i.e. the phenotype space) is that of selecting two graphs, dividing each of them in two subgraphs and swapping the extracted subgraphs to form two new graphs.

The crossover operator was defined by using the previously described t-cut and merge operations. Let us assume that two multilists L_1 and L_2, having size N_1 and N_2 respectively, have been selected. Without loosing generality, let us also assume that $N_1 \leq N_2$. The crossover operator produces two new multilists M_1 and M_2, whose size vary in the interval $[N_1, N_2]$, by carrying out the following steps:

1. Choose randomly a number t_1 in the interval $[1, N_1 - 1]$. Apply the t_1-cut operation to L_1;
2. Choose randomly a number t_2 in the interval $[t_1, t_1 + (N_2 - N_1)]$. Apply the t_2-cut operation to L_2.
3. Apply the merge operation to the left multilist produced by the t_1-cut operation applied to L_1 (point 1) and to the right multilist produced by the t_2-cut operation applied to L_2 (point 2);
4. Apply the merge operation to the left multilist produced by the t_2-cut operation applied to L_2 (point 2) and to the right multilist produced by the t_1-cut operation applied to L_1 (point 1);

The constraint on the value of t_2 is introduced to avoid

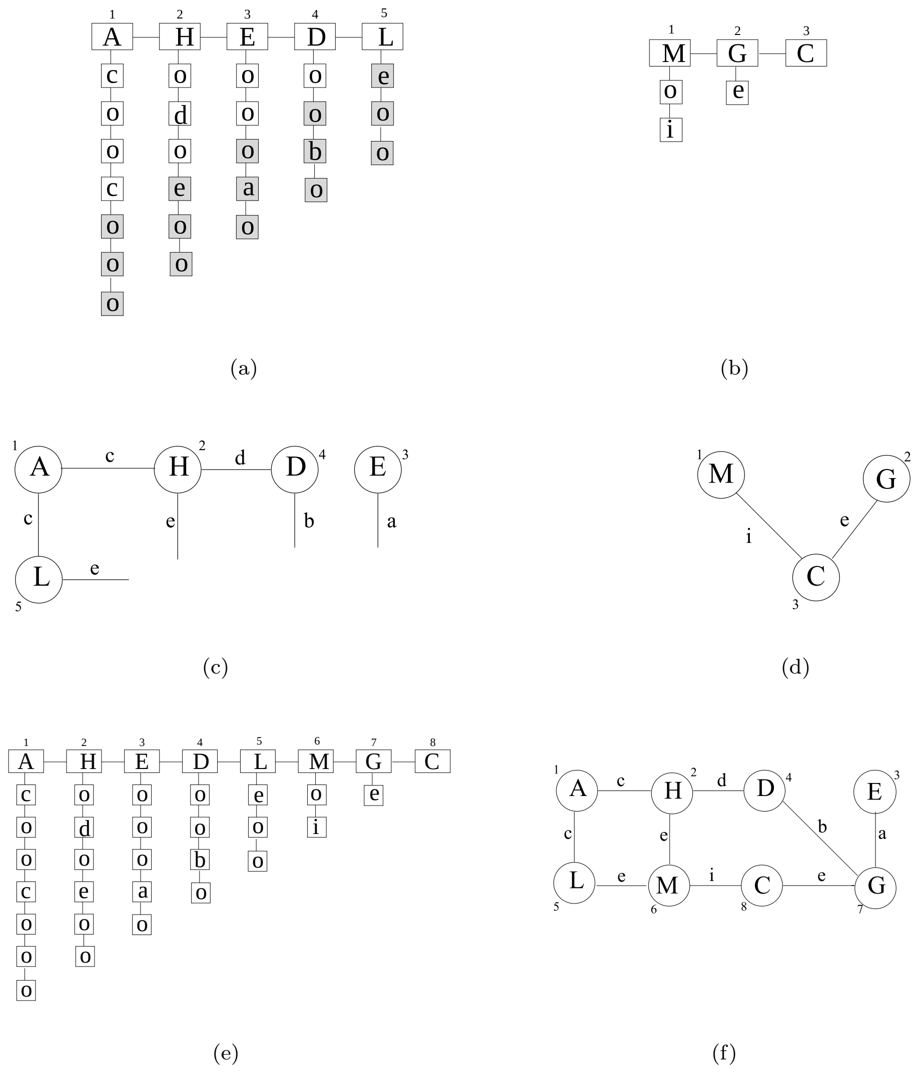


Figure 4: An example of merge operation with $Rd_1 = N_2$. A redundant ML with redundancy degree equal to 3 (a), and a complete ML of size 3 (b). The corresponding graphs are shown in (c) and (d). The resulting ML after merging (e) and the corresponding graph (f).

the generation of offspring exhibiting very different size between them. This constraint force crossover to generate new individuals whose size is included in the interval $[N_1, N_2]$. Nonetheless, in order to improve the exploration ability of the evolutionary algorithm, the above constraint has been relaxed by extending the range in which the size of the offspring may vary. In particular, a tolerance θ has been introduced, such that the value of t_2 may be chosen in the interval $[t_1 - \theta, t_1 + (N_2 - N_1) + \theta]$, so allowing the generation of individuals whose minimum and maximum size respectively is $N_1 - \theta$ and $N_2 + \theta$. In the experiments reported in Section 4, the value of the tolerance θ was arbitrarily fixed to 1.

Notice that the defined crossover operator, differently from other approaches, does not require any search operation on the graphs to be crossed and exhibits a computational complexity which is quadratic with respect to the number of nodes in the individual. In [15], for instance, the crossover is implemented by randomly choosing a fixed number of subgraphs for each individual to be crossed. Therefore, as the number of graph nodes grows, the number of available subgraphs raises exponentially and the operator becomes very soon unfeasible.
In [14] the splitting of a graph to be crossed is performed by randomly choosing a starting arc between two nodes and then finding and breaking all the paths in the graph connecting such nodes (a path is broken by randomly deleting one of its arcs). Also in this case, splitting a graph requires searching for all the possible paths between two nodes: this operation may be computationally very expensive in case of large graphs.

3.4. Mutation Operator

Our mutation operator actually performs a “micro” mutation, because it does not modify the structure of the

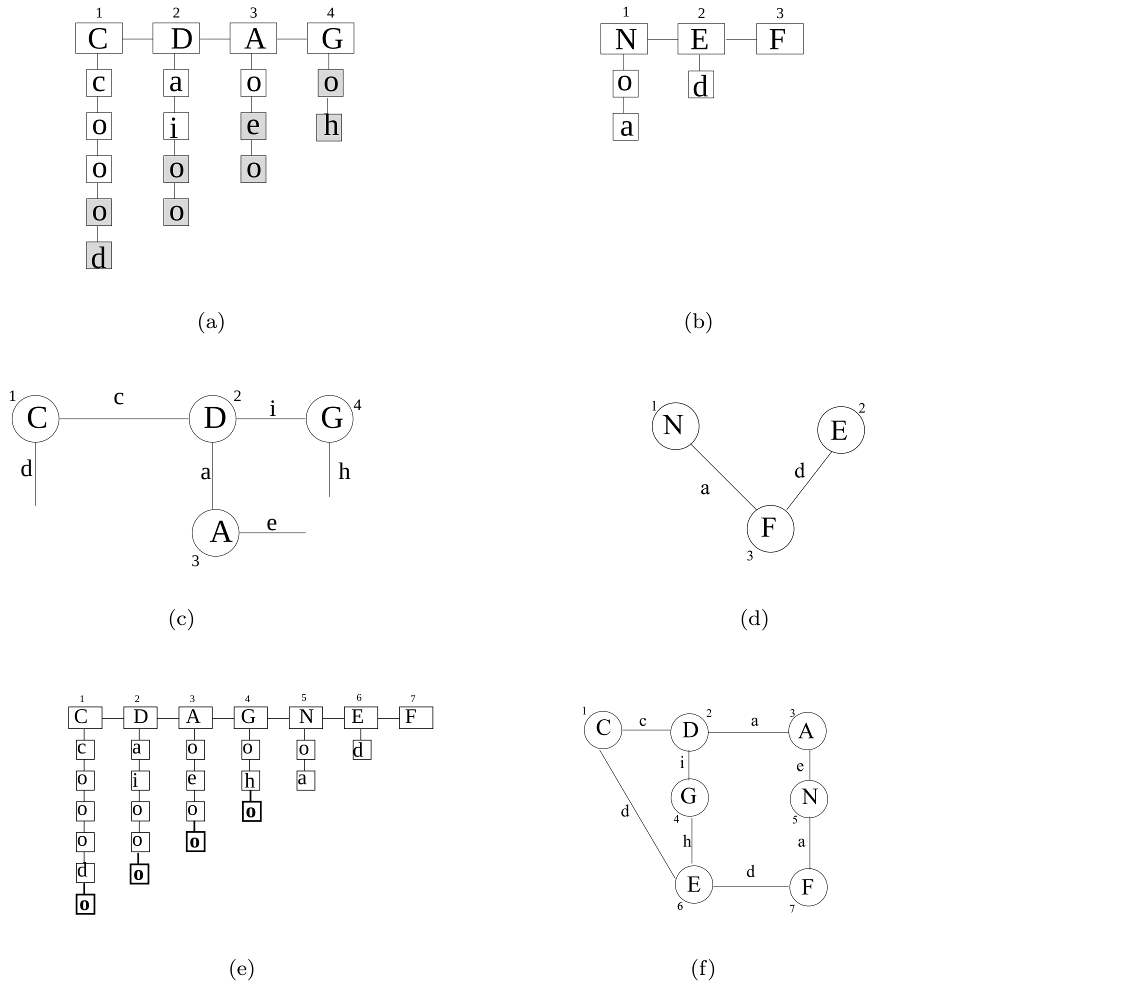


Figure 5: An example of merge operation with $Rd_1 < N2$. A redundant ML with redundancy degree equal to 2 (a), and a complete ML of size 3 (b). The corresponding graphs are shown in (c) and (d). The resulting ML after merging (e) and the corresponding graph (f). Note that this ML has been made complete by adding a null element to each sublist of the first 4 nodes. These added elements are bold.

multilist it is applied to, but just the attributs of main list and sublist elements.

Given a multilist L of size N, the mutation is applied according to a suitable probability p_m through the following steps:

```
for each node $n_i$ of the main list do
  if flip($p_m$) then
    Choose randomly a value belonging to the set $A_n$;
    Substitute the value of $n_i$ with the chosen value;
  for each of the elements $e_{ij}$ of the $i-th$ sublist do
    if flip($p_m$) then
      Choose randomly a value belonging to
      the set $A_a \bigcup\{NULL\}$;
      Substitute the value of $e_{ij}$ with the chosen value;
  end
end
```

The function flip(p) returns the value *true* with a probability p and the value *false* with a probability $(1-p)$. Let $L^{'}$ and $G^{'}$ respectively be the mutated multilist and the corresponding graph. Note that $G^{'}$ and G contain the same number of nodes, while the number of their arcs may be different. In fact, since one of the arc attribute is the presence or absence of a connection, changing arc attributes may also result in adding or removing a connection between nodes. Namely, substituting a value belonging to the set A_a with the $NULL$ value has the effect of deleting the corresponding arc, while the opposite has the effect of adding a new arc. In case of graphs without attributes, this is obviously the only effect of the mutation operator. Finally, the computational complexity of the mutation operator is quadratic with respect to the number of nodes in the individual considered. An example of the application of the mutation operator is shown in fig. 7.

3.5. Outline of the algorithm

Let us recall that individuals of the evolving population are MLs, each encoding a graph representing a possible solution of the problem to be solved. The algorithm starts by

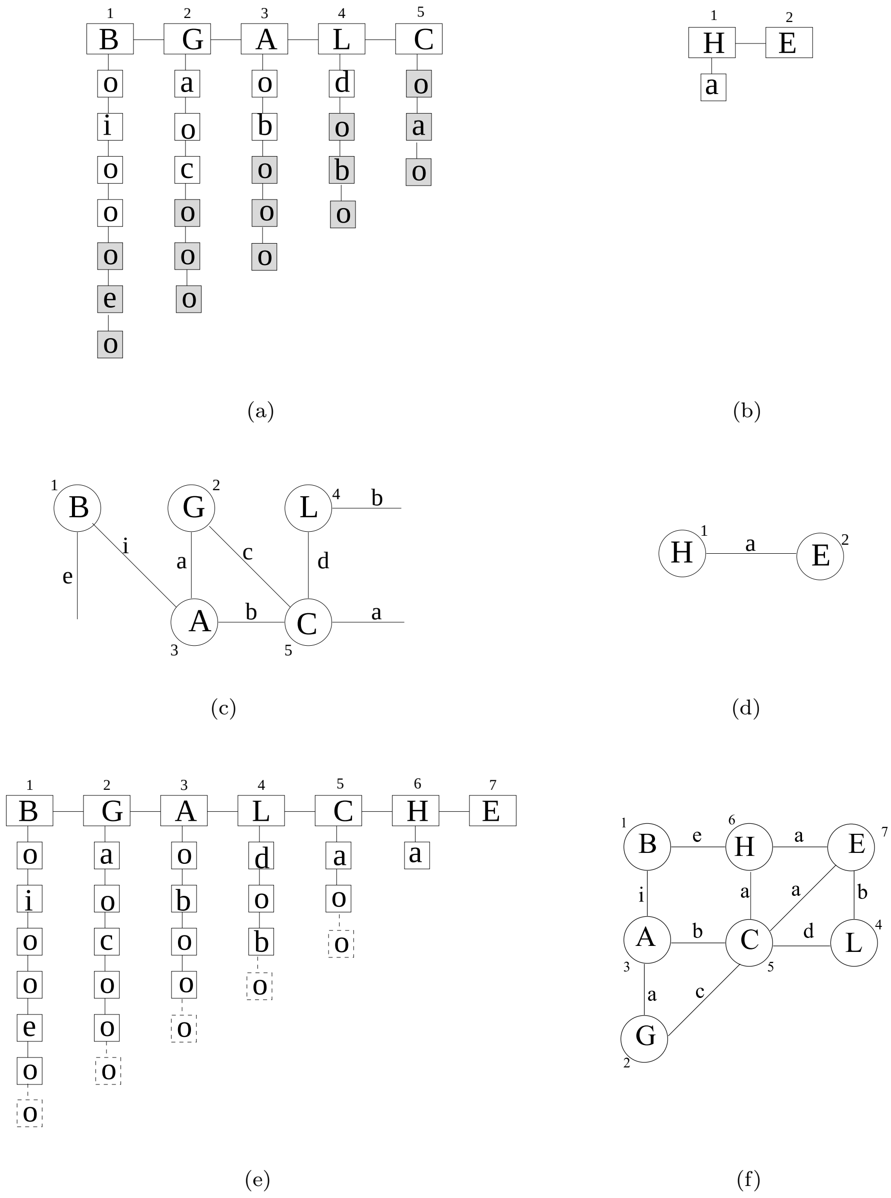


Figure 6: An example of merge operation with $Rd_1 > N2$. A redundant ML with redundancy degree equal to 3 (a), and a complete ML of size 2 (b). The corresponding graphs are shown in (c) and (d). The resulting ML after merging (e) and the corresponding graph (f). Note that this ML has been made complete by deleting the last element from each sublist of the first 4 nodes. For the sake of clarity, these elements are still shown using dashed lines.

randomly generating an initial population of $\mathcal{P}$ individuals, whose number of nodes ranges from 2 to N_{max}. Afterwards, the fitness of these individuals is evaluated. At each generation, a new population is generated by first selecting the best e individuals in the current population, in order to implement an elitist strategy. Then,$(\mathcal{P} - e)/2$ pairs of individuals are selected with the tournament method. For each pair, two new individuals are generated by applying the above defined genetic operators, and copied in the new population. The algorithm is outlined in the sequel:

```
begin
    randomly initialize a population of $\mathcal{P}$ individuals;
    evaluate the fitness of each individual;
    evaluate the fitness of each individual;
    while (termination criteria are not fulfilled) do
        copy the best $e$ individuals in the new population;
        for $i = 0$ to $(\mathcal{P} - e)/2$ do
            apply the selection mechanism to the current
            population: 2 individuals are selected;
            replicate the selected individuals;
            apply the crossover operator to the selected
            individuals (with probability $p_c$);
            apply mutation to the offspring
            (with probability $p_m$);
```

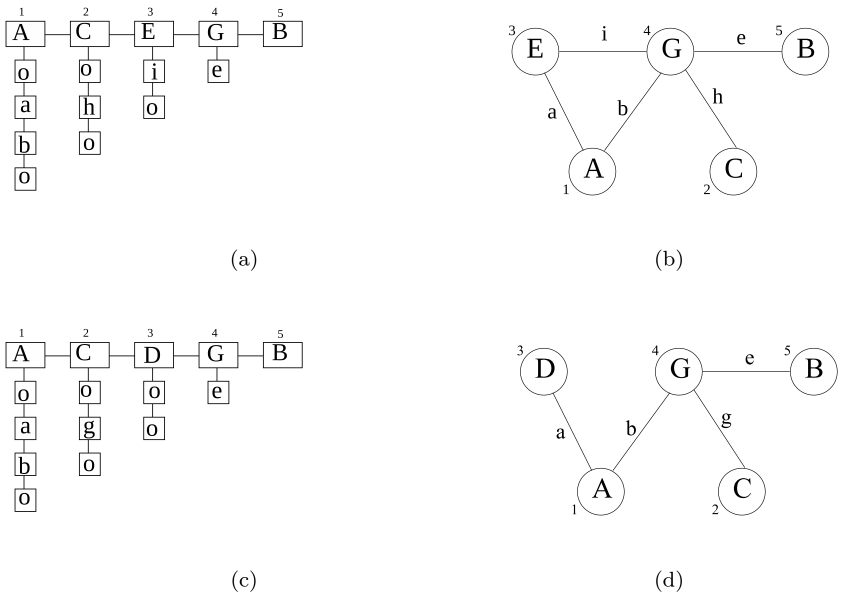


Figure 7: An example of application of the mutation operator: the initial ML (a) and he corresponding graph (b). The resulting ML (c) and the corresponding graph (d). The mutation changed the attributes of node 3 and of the arc between nodes 2 and 3. Moreover, the arc connecting nodes 3 and 4 has been deleted.

```
            evaluate the fitness of the offspring;
            copy the offspring in the new population;
        end
        replace the current population with the new one;
        update variables for termination criteria;
    end
end
```

4. Experimental Results

In order to ascertain the effectiveness of the proposed approach, three sets of experiments were performed. The first set regards a synthetic graph search problem, whereas the second one concerns a real world planning and optimization problem involving graphs. Finally, the third set deals with the standard one-max tree problems. In the following, the achieved results on the three problems taken into account will be detailed.

4.1. Synthetic Graphs

The purpose of this set of experiments was that of evaluating the effectiveness of the proposed method for exploring a complex graph search space. Given a target graph and a fitness function measuring the distance between a generic graph and the target one, our aim is that of evaluating the ability of our algorithm to evolve a population of graphs in order to generate the target graph. In each experiment, a synthetic graph generated according to the random graph model [2] was chosen as target. This a model requires fixing both number of nodes and occurrence probability of arcs (p_a), representing the probability that, given two nodes, an arc is inserted between them. Node and arc attributes are randomly selected within the sets $\mathcal{A}_n$ and $\mathcal{A}_a$ whose cardinalities were arbitrarily fixed to 26. Experimenting on synthetic graphs allowed us to evaluate the performance of the algorithm as a function of the graph complexity, expressed in terms of both number of nodes and p_a. Target graphs were obtained by using the random graph generator $\gamma(N, p)$, defined in [39], where the parameter N represents the number of nodes, while the parameter p is p_a. The sets $\mathcal{A}_n$ and $\mathcal{A}_a$ are made of the following elements:

$$\mathcal{A}_n = \{A, B, C \ldots, X, Y, Z\} \qquad \mathcal{A}_a = \{a, b, c \ldots, x, y, z\}$$

A set of thirty target graphs whose number of nodes varies from 10 to 60, with steps of 10, and whose p_a ranges from 0.1 to 0.9, with steps of 0.2, has been generated. In Fig. 8 the target graph generated by the model $\gamma(20, 0.3)$ is shown.

Let's recall that, according to our encoding scheme, target graphs, as well as population individuals, are represented by multilists.

4.1.1. The Fitness Function

The defined fitness function consists of two terms. The former term measures the node number (i.e., the size) difference between a sample multilist and the target one. The latter term measures the attribute similarity between corresponding nodes and corresponding arcs of the two multilists by using a distance function $\mathcal{D}$ defined in the same way

[2]Random graphs are formally described in Appendix 6.

for both sets $\mathcal{A}_n$ and $\mathcal{A}_a$. If x and y are respectively the i-th and the j-th element of an ordered set of attributes, then $\mathcal{D}(x, y)$ is equal to $|i - j|$.

If T is the target multilist, and I a generic individual in the evolving population, the term of the fitness function measuring the size difference is:

$$f_s(I) = \left|N^T - N^I\right|$$

where N^T and N^I respectively denote the size of T and I. The term measuring the attribute difference is:

$$f_a(I) = \sum_{i=1}^{N} \left|\nu_i^T - \nu_i^I\right| + \sum_{j=1}^{N}\sum_{k=1}^{N} \left|\alpha_{jk}^T - \alpha_{jk}^I\right|$$

where $N = min(N^T, N^I)$ and ν_i^T and ν_i^I respectively indicate the attributes of the i-th node of T and I. The attribute of the arc of T connecting node n_i to node n_j is denoted by α_{jk}^T, while the corresponding arc attribute of I is denoted by α_{jk}^I. Summarizing, the fitness function f of a generic individual I is defined as a weighted sum of two terms:

$$f(I) = w_s f_s(I) + w_a f_a(I)$$

where w_s and w_a represent the weights. Note that $f(I) = 0$ if and only if I and T are identical.

For all the experiments reported in the following, only graphs whose size is less or equal to N_{max} are allowed to evolve. Moreover, w_s was fixed to 8.0 and w_a was fixed to 0.01. These values were selected, after some preliminary trials, since they allowed the algorithm to achieve better performances. The high difference between the two values is justified by the fact that, by definition, the term $f_a(I)$ may assume values much higher than those achievable by $f_s(I)$.

4.1.2. Evolutionary Parameter Tuning

Preliminary experiments were carried out in order to find effective values for crossover probability p_c and mutation probability p_m. To this aim, three graph sizes, namely 10, 30 and 45, are considered and, for each size, two random graphs with p_a equal to 0.5 and 0.9 respectively, were generated, totaling 6 target graphs. For each of them, the performance of the graph search algorithm was evaluated in terms of number of generations needed to find the target graph.

A set of experiments were performed varying the mutation probability value in the range $[10^{-4}, 10^{-1}]$, keeping the value of the crossover probability constant (heuristically fixed to 0.8). This choice is motivated by the consideration that mutation probability is usually assumed to be very small, so that, on the average, at most one gene in each individual is modified. Crossover probability is usually higher, assuming values that typically range from 0.6 to 1.0. During these experiments, the size of the interval (called *mutation size*, m_s) in which node and arc attributes may vary, has been also tuned: assigned a value to m_s, a node or arc attribute may be substituted by one of the m_s symbols preceding or following it in the corresponding alphabet. Values of m_s belonging to the interval $[1 - 8]$ were tested.

The results of this set of experiments are shown in figure 9. For the sake of clarity, the results obtained with mutation probability values for which the algorithm exhibited very slow convergence or did not converge at all, are not reported. Finally, each plot reports the performance averaged over 30 runs. The analysis of the results shows that the optimal value for the mutation probability is determined by the number N of nodes in the target graph. In particular, denoting with m the total number of elements included in the main list and in all the sub-lists (according to the multilist definition, $m = (N*(N+1))/2$), it can be simply verified that the best performance is always obtained for a value of the mutation probability equal to $1/m$. This behavior suggests that, similarly to classical genetic algorithms [40], the optimal value for the mutation probability is the one allowing modification, on average, of only one element of the multilist: either node or arc. In fact, considering that the fitness function tends to favor individuals with a number of nodes equal to that of the target graph, after the early stages of the evolution, most of the individuals in the population will have a size very similar to that of the target graph, and thus for such individuals the total number of elements included in the main list and in all the sub-lists will be very similar to m.

The analysis of the results also shows that in case of target graphs with 10 nodes, the search algorithm is almost independent of the mutation size, exhibiting very similar performance for values greater than 1. For larger graph sizes, the performance reaches its maximum for values of the mutation size in the range $[3 - 5]$, and then decreases for higher values. Since the value 5 allowed the best results in all the experiments to be obtained, this value has been assigned to the mutation size.

As regards the crossover probability, two graph sizes, namely 10 and 45 were considered. For each size, two random graphs with p_a equal to 0.5 and 0.9 respectively, were

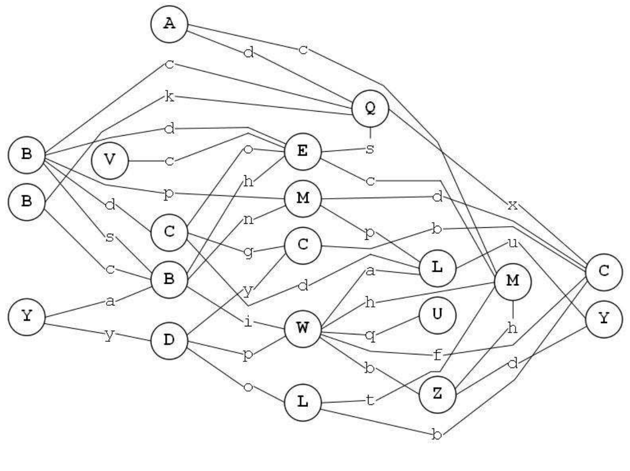
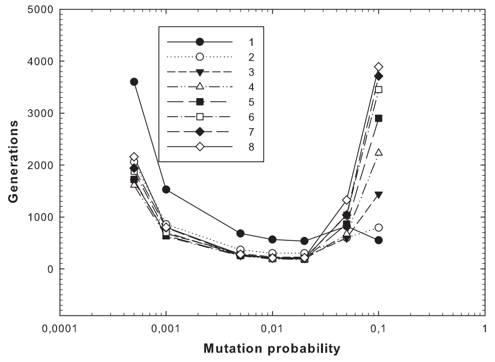
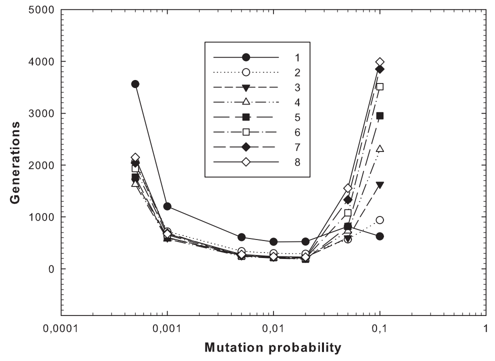


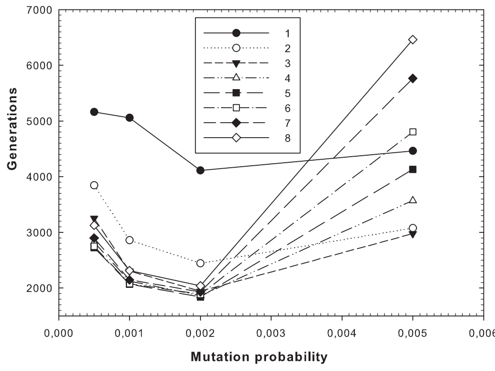
Figure 8: The random graph generated by the model $\gamma(20, 0.3)$. In this case, the model generated 37 arcs interconnecting the 20 nodes).



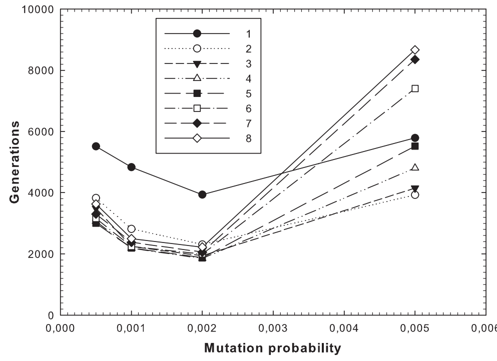
(a)



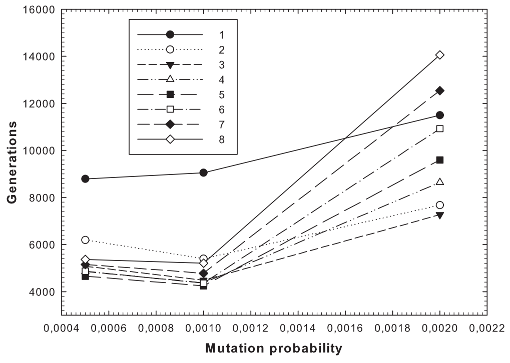
(b)



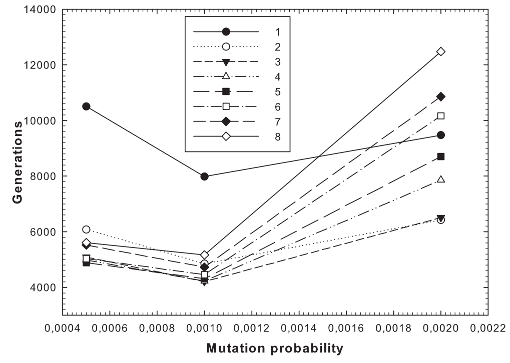
(c)



(d)



(e)

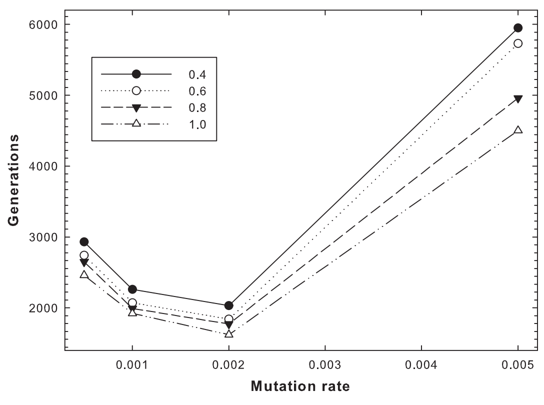


(f)

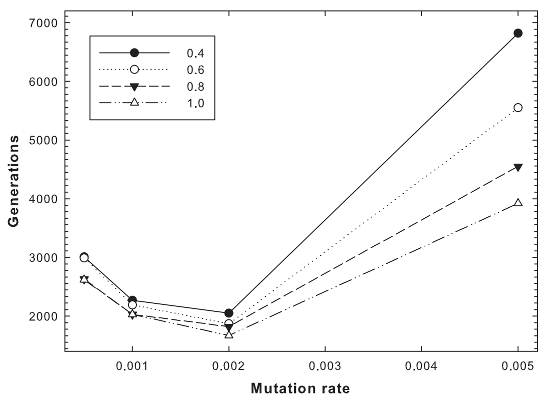
Figure 9: Number of generations needed to find a target graph as a function of the mutation probability, for p_a values 0.5 (left) and 0.9 (right). (a) and (b), 10 node graphs; (c) and (d), 30 node graphs; (e) and (f), 45 node graphs. In (a) and (b), a logarithmic scale has been adopted for better displaying the results.

generated, totaling 4 target graphs. For each of them, the experiments were performed using mutation probability and mutation size values determined according to the results of the experiments previously discussed. In all the experiments the best performance are obtained with crossover probability equal to 1.0. Table 1 shows the values of the evolutionary parameters used in the experiments.

Finally, in order to investigate the relationship between crossover and mutation probabilities, a factorial experiment was performed in which four different values for each probability were considered. More specifically, the the values 0.0005, 0.001, 0.002 and 0.005 were used used for the mutation probability, while the values 0.4, 0.6, 0.8 and 1.0 were used for the crossover probability. As for the re-



(a)



(b)

Figure 10: Number of generations needed to find a 30 node target graph as a function of the mutation probability for different crossover probability values. (a) $p_a = 0.5$ and (b) $p_a = 0.9$.

maining parameters, the values reported in Table 1 were used. The results obtained for a 30 node graph are shown in Fig. 10. From the figure, it can be observed that the curves for different crossover probability values have a very similar shape, this fact seems to suggest that these two parameters are weakly related, or even not related at all. The only conclusion that can be drawn from these plots is that higher crossover probability values give better results (faster convergence). Note that the curves for other target graph sizes have a very similar shape and thus they have been omitted.

4.1.3. Crossover Operator and Heritability

In this subsection several experiments were performed in order to test the effectiveness of the crossover operator and, in particular, its heritability. Heritability refers to the capacity of the crossover operator to produce offspring containing meaningful features of the parents. In Fig. 11 the fitness value of the best individual is plotted over the number of generations for 30 and 45 target graphs. In each plots 4 curves are shown, corresponding to the following values of the crossover probability p_c: $0.0, 0.25, 0.5, 1.0$. Note that the values are averaged over 30 runs. The number of generations necessary to find the target graph is also reported: this allows the differences in the convergence behavior of the algorithm for different p_c values to be highlighted. In particular, the plots show that the best performance are obtained with $p_c = 1$ and, more generally, the higher p_c, the better the convergence of the algorithm. These results seem to demonstrate that the crossover operator has a good heritability. In fact, it is able to preserve the good sub–graphs structures found during the evolution, and to combine them in a very effective way.

Table 1: Values of the evolutionary parameters used in the experiments on synthetic graphs. The value m is given by the number of nodes plus the number of sublist elements in the multilist to be mutated.

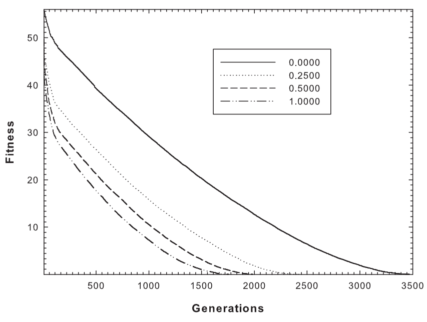
Parameter	**symbol**	**value**
Population size	P	100
Tournament size	$\mathcal{T}$	6
Elitism size	e_s	1
Number of Generations	n_g	20000
Crossover probability	p_c	1.0
Mutation probability	p_m	$1/m$
Mutation size	m_s	5
Maximum number of nodes	N_{max}	100
Minimum number of nodes	N_{min}	2

4.1.4. Testing

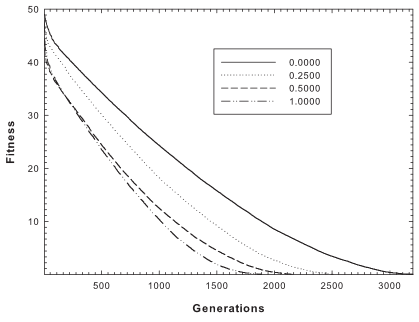
Testing was carried out by generating several target random graphs with different size and p_a. More specifically, the following sizes were considered: {10,20,30,40,50, 60}. For each size, the following values of p_a were used: {0.1,0.3,0.5,0.7,0.9}, totaling thirty random graphs. For each graph, thirty runs were performed. The system was able to find the target graph in every run, demonstrating its ability to explore search spaces of different size. In Fig. 12 the average number of generations needed to find the target graph is shown as a function of target graph size, for different p_a values. The figure shows that, for each size, target graphs having more arcs need a relatively larger number of generations to be found, in spite of the fact that, for a given size, graphs with different p_a are represented by multilists with the same number of sublist elements.

4.2. Real World Problem

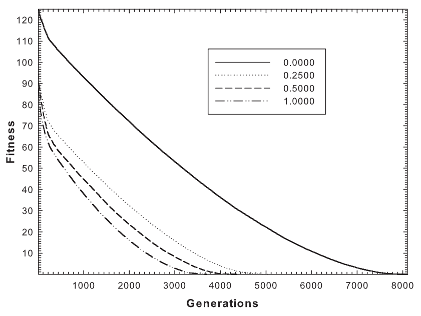
The real world problem taken into account deals with the design and optimization of a wireless network. In particular, the wireless access point configuration problem has been considered. This is a hard non-linear optimization



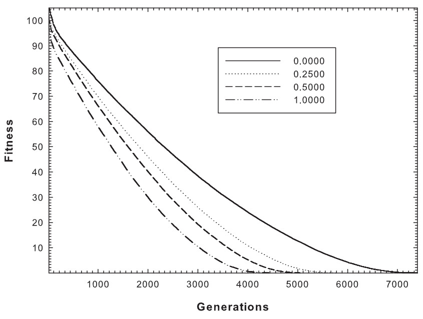
(a)



(b)



(c)



(d)

Figure 11: Fitness of the best individual as function of the number of generations for p_a values 0.5 (left) and 0.9 (rigth). (a) and (b) 30 node graphs; (b) and (c) 45 node graphs.

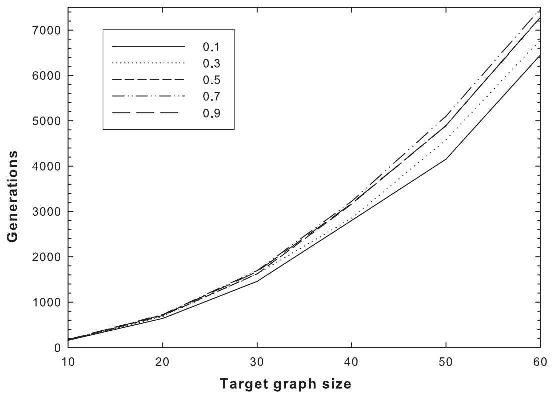


Figure 12: Number of generations needed to find the target graph as a function of target graph size.

problem whose scenario is the following: a community is planning to provide wireless Internet service to its members (clients) who are scattered around a given area. A certain number of access points need to be placed to cover all clients, because each access point has a limited service radius. All access points are wired and one of them is connected to an Internet gateway. The design problem consists in determining the optimal configuration of the APs in the area to be covered. In order to reduce the cost, a configuration with minimum number of APs and minimum length of the wires connecting them is considered optimal. According to the constraints imposed, the wireless access point configuration problem was formulated in different ways. E.g., in [41] it is assumed that APs are selected within a pre-specified set of possible points, while in [26] APs can be located at any place. For the sake of generality, this second working hypothesis is used and thus our results will be compared with those presented in [26]. In order to avoid time consuming evaluations, the fitness function does not include parameters like transmission power, channel allocation, bandwidth, antenna direction, etc. be evolved. Nonetheless, this fitness still has the essentials of the wireless configuration problem, allowing fast fitness evaluations. The problem can be formally defined as follows:

ASSUME that all APs are equal and have service radius r_s;

GIVEN a gateway G located at (x^G, y^G) and a set of N

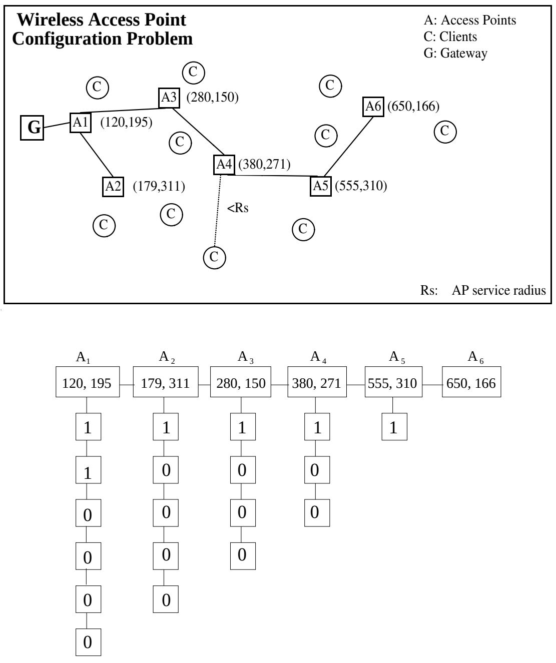


Figure 13: An instance of the wireless access point configuration problem (top). Circled C's represent clients, while squares represent APs. The multilist encoding a possible solution (bottom).

clients located at (x_i^c, y_i^c) $(i = 1 \dots N)$, in an area of size $W \times H$;
FIND a configuration of wired access points located at $(x_i^{AP},\ y_i^{AP})$ $(i = 1 \dots\ N_{AP})$, connected to the gateway port G in such a way that each client is covered by at least one AP and the total cost of both APs and wires is minimal.
Thus, let C_{AP} be the cost of each AP and C_w be the cost of a unit length wire, the aim is to minimize the function:

$$f = C_{AP} * N_{AP} + C_w * \sum |L_i|$$

where L_i is the wire length between two connected APs.

Problem solutions can be represented by a graph whose nodes are the APs and whose arcs are the wired connections between APs. The set of node attributes is made up of the AP coordinates in the area to be covered, represented by couples of integers (see Fig. 13). In the ML representation encoding the graphs, the value 1 is used to indicate the presence of an arc, while the value 0 indicates the absence of an arc, i.e. the NULL relation. In the experiments reported below, AP coordinates assume values (expressed in unit length) in the range $[0 - 1000]$. The positions of the clients to be served are also represented by integer coordinates within the same range.

In the following the fitness function used for evaluating the solutions will be defined and the experimental results obtained for several problem instances will be discussed.

4.2.1. The Fitness Function

To compare the performance of our method properly with that of the method proposed in [26], the same fitness function was adopted. In this way, it is possible to ascribe any difference in the performance of the methods only to the way the solutions are generated, and not to the way they are evaluated. According to [26], the definition of the fitness function has to take into account three aspects of the problem: the percentage of covered clients, the number of APs employed and the total length of the wires connecting them. For this reason, the fitness function was defined as the weighted sum of three terms. The first term F_c measures how well the clients are covered by the AP configuration: the more clients are covered, the better. The second term F_w measures how good the connection topology is: the shorter the wires used, the better. Finally, the term F_{AP} estimates the goodness of a configuration as regards the number of wireless APs employed: the fewer AP used the better. In order to reflect their different importance for evaluating the goodness of a configuration, the above three fitness terms are normalized and suitably weighted. The fitness terms are:

$$F_c = \frac{4.0 * C_c}{C_c + C_T}; F_w = \frac{10000}{10000 + L_w}; F_A = \frac{C_T}{C_T + 1.5 * N_{AP}}$$

where C_c is the number of covered clients, C_T is the total number of clients, N_{AP} is the number of APs and L_w is the total length of wire segments connecting the APs. The fitness function F_{tot} is the weighted sum of the above three terms:

$$F_{\mathrm{tot}} = 0.7 * F_c + 0.1 * F_w + 0.2 * F_{\mathrm{AP}}$$

As previously mentioned, numeric constants and weights in the above equations are those proposed in [26].

4.2.2. Testing

The results reported below refer to a set of experiments performed on 10 different problem instances, obtained by varying the number of clients from 10 up to 100, with increments equal to 10. The clients in the area to be covered are randomly placed. In order to have statistically valid results, 30 runs with different initial populations were performed for each instance. At the end of each run, the best solution found was stored.

As concerns the parameter values, they were set to the same values used for the experiments on synthetic graphs (see Table 1), except than for the mutation size m_s. This value was chosen according to the results on synthetic graphs, where the optimal value found for the mutation size m_s was 5, i.e. $\approx 1/5$ of the size of the alphabet used (26), in this case, the value $m_s = 200$, i.e. $1000/5$, was chosen.

In Fig. 14 the average number of APs and their standard deviation (over the 30 performed runs), as a function of the number of clients, is shown. It can be observed that the number of APs slightly increases with the number of clients. This fact demonstrates that our system is able to optimize the APs number since, as the number of clients increases, it adds only the number of APs just needed to

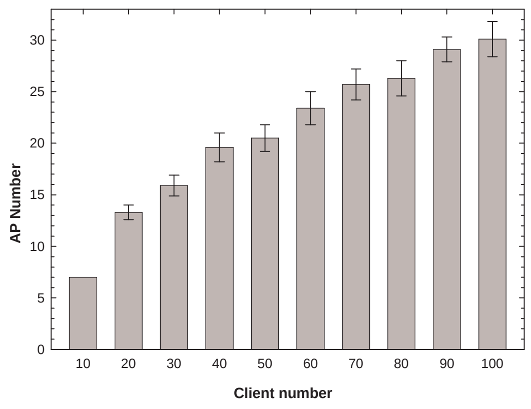


Figure 14: The average number of APs and its standard deviation as a function of the number of clients are respectively represented by bars and segments on top of bars.

cover the added clients. Moreover, the system has always been able to find solutions that cover all the clients. Finally, the standard deviations assume small values, thus indicating that the algorithm converges to solutions with almost the same number of APs, independently of the initial conditions.

In order to assess the effectiveness of the proposed approach in finding near optimal connection topology, the Minimum Spanning Tree (MST) [1] of the complete graph obtained by considering the AP configuration provided by our method, has been separately computed. The MST, in fact, represents the connection topology with the minimum wiring cost. Problems with different number of clients were considered and, for each problem, the connection length found by our algorithm was compared with the optimal length computed by the MST. In order to get a statistically valid comparison, 30 runs of the EA were performed for each considered number of clients.

In the following we will denote by $\overline{L}$ the mean of the wire lengths found by our system, and by $\overline{L}_{\mathrm{MST}}$ the mean of the wire lengths corresponding to the MST, over the 30 performed runs. The plot of $\overline{L}$ and $\overline{L}_{\mathrm{MST}}$ as a function of the number of clients, is shown in Fig. 15. This plot makes clear that our system is also able to optimize connection topologies. In fact, the connection lengths of our solutions (white bars in the plot) are very close to the optimal ones of the MST (grey bars). Moreover, the standard deviations assume quite small values, demonstrating the robustness of the search process. In Fig. 16 the best solutions found for the 20, 50 and 100 clients instances are shown. In these solutions the number of APs used is respectively equal to 12, 18 and 25. Note that, for these solutions, the connection topology found by our system coincides with that found by using the MST. The figure clearly shows that the APs are distributed so as to cover all clients, and that the overlaps among APs are very limited. It is worth noting that pairs of connected APs are not too far from each other, so reducing the length of the connections.

The results obtained by our system in solving the wireless network access point configuration problem were compared with those presented in [26] where results for 25 and 40 clients instances are reported. These results show that the approach fails to optimize time AP placement and connection topology at the same, demonstrating that the method was not able to evolve graphs. Although a good AP arrangement was obtained at the end of the evolution, the system was never able to find a good connection topology. Consequently, the authors chose to optimize only the AP placement and then to use an MST algorithm for optimizing the connections: even if the results are quite effective, they were actually obtained by using a GP–based approach for evolving only the positions of the APs.

On the contrary, our system allows us to optimize number and position of the APs in the area, minimizing time the connection length at the same. The experimental results confirmed the effectiveness of our approach: good solutions, comparable with those presented in [26], were obtained for all the considered instances (see Fig. 15). These results are very meaningful since they demonstrate that our algorithm is able to evolve graphs and to find effective solutions in complex search space.

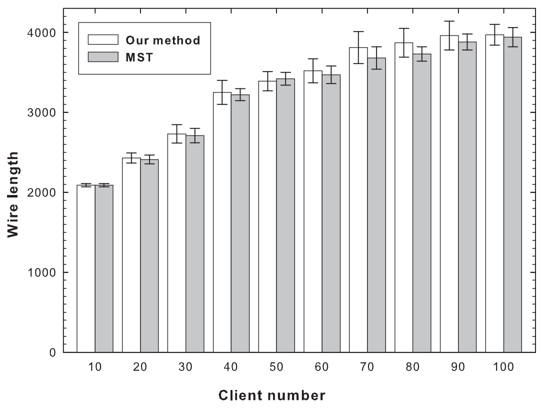


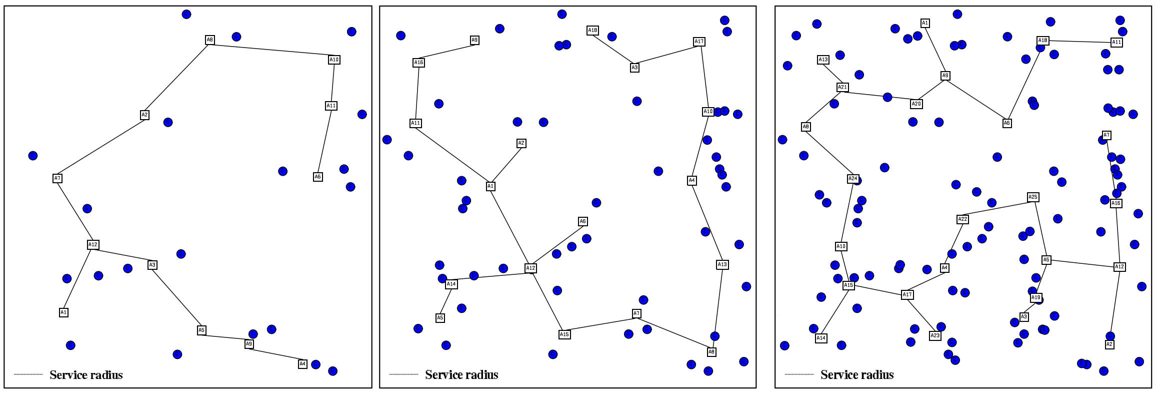
Figure 15: The average value of the wire length and its standard deviation as a function of the number of clients, for the solutions found by our method and by using the MST.

4.3. The One-Max Tree Problem

In the One-Max-Tree problem [36], a target spanning tree G_T must be found, and the fitness of any other tree is the number of edges that it shares with the target. More specifically, the fitness of a generic individual I, encoding the graph G_I is defined as:

$$f(I) = \sum_{i=1}^{N} \sum_{j=i+1}^{N} |l_{ij}^{G_T} - l_{ij}^{G_I}|$$

where the value of l_{ij} is 1 if the arc connecting the nodes i and j exists, 0 otherwise. The one-max-problem has been



(a) 20 clients (b) 50 clients (c) 100 clients

Figure 16: The best solutions obtained for 20, 50 and 100 clients. Clients are represented by circles, while APs are represented by white squares.

used several times as benchmark problem [36, 42, 43]. For the sake of comparison, the results obtained by our approach on the One-Max tree problem were compared with those obtained by a different tree representation scheme for evolutionary algorithms, called Network Random Keys (NRK in the following), presented in [36]. In this approach, a spanning tree T of N nodes is represented by means of a real valued vector $\vec{v} = (v_1, v_2, \ldots, v_l)$, with $l = N*(N-1)/2$ and $v_i \in [0.0, 1.0]$. Each element v_i corresponds to a possible arc of T, and its value represents the probability of that arc belonging to T. In [36], optimal solutions are then searched by means of a simple GA [44].

The comparison was performed on eight randomly generated instances, with different number of nodes (10, 20, 30, 60) and topology (tree or star). For each instance, the average number of generations needed to find the optimal solution was computed over 30 runs with different initial populations. As concerns the parameters, the values shown in Table 1 (first six rows) were used for both our approach and the comparing one.

In Fig. 17 the average number of generations and its standard deviation, over the 30 runs, needed to find the target spanning tree as a function of target spanning tree size is shown for the compared methods. The figure clearly shows that the proposed approach needs many fewer generations to find the optimal solution, and that the performance differences strongly increase with the solution size. In order to investigate whether the performance difference was due to the parameter values chosen, or to the topology of the spanning tree, both methods were tested by using different selection strategies (tournament and roulette wheel), different topologies (star or tree) and different values (0.5 and 1.0) for the crossover probabilities. The results obtained are reported in Table 2. The table shows the mean μ and the standard deviation σ, over the 30 runs, of the number of generations needed for finding the optimal solution. The t-test was performed on the achieved

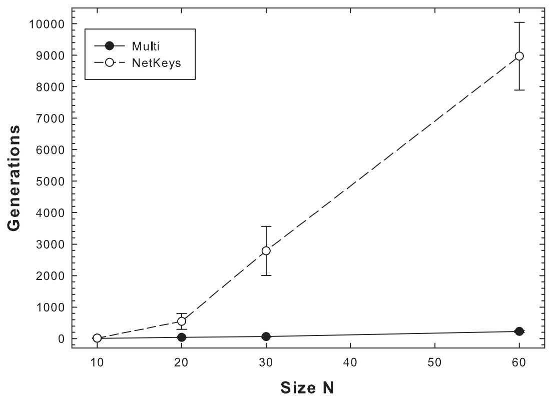


Figure 17: Average number of generations (and standard deviations) over the 30 runs, needed to find the target spanning tree as function of target spanning tree size.

results, and the probability p_t that they are drawn from the same population is also shown. The reported results confirm the trend shown in Fig. 17: the proposed approach obtains better results than those of the approach considered for comparison, and the differences are highly statistically significant ($p_t < 10^{-6}$). Only for the 10 node graphs the differences are not statistically significant (except for the star topology with tournament selection and $p_c = 0.5$, where $p_t = 0.002$). As for the crossover probability p_c, from the table it can be observed that the runs with $p_c = 1$ achieve better results than those obtained by using the value $p_c = 0.5$. Moreover, it can be also noted that the tournament selection always yields better results than the roulette wheel.

μ μ

The first one regarded the search for synthetically generated target graphs. It was considered in order to test the ability of the system to find solutions as the complex-

Table 2: Comparisom results.

Size N	Type		Generations							
			Tournament				Roulette wheel			
			$p_c = 0.5$		$p_c = 1.0$		$p_c = 0.5$		$p_c = 1.0$	
			multi	NRK	multi	NRK	multi	NR	multi	NRK
10	tree	μ	11.87	11.96	10.83	10.5	18.67	19.77	17.23	19.8
		σ	1.8	5.7	1.7	8.6	6.7	8.2	6.5	8.1
		p_t	0.83		0.4		0.57		0.19	
	star	μ	13.8	26.27	12.27	18.97	30.6	36.2	27.3	33
		σ	1.8	20	1.7	17.2	8	17	8	19.5
		p_t	0.002		0.042		0.11		0.42	
20	tree	μ	42	730	35	545	226	685	187	643
		σ	5	320	4.4	250	42	340	42	365
		p_t	$< 10^{-6}$		$< 10^{-6}$		$< 10^{-6}$		$< 10^{-6}$	
	star	μ	46	750	46	676	230	794	192	800
		σ	6.4	370	7.6	385	35	350	37	200
		p_t	$< 10^{-6}$		$< 10^{-6}$		$< 10^{-6}$		$< 10^{-6}$	
30	tree	μ	75	2768	61.43	2787	693	3142	643	3713
		σ	10	1312	10.3	780	128	1172	138	1590
		p_t	$< 10^{-6}$		$< 10^{-6}$		$< 10^{-6}$		$< 10^{-6}$	
	star	μ	87	2602	74	2783	667	3204	728	3124
		σ	13	900	10	850	125	1114	175	1300
		p_t	$< 10^{-6}$		$< 10^{-6}$		$< 10^{-6}$		$< 10^{-6}$	
60	tree	μ	269	9835	226	9456	7095	26,894	5020	25,945
		σ	43	1900	36	2050	1050	4100	730	4450
		p_t	$< 10^{-6}$		$< 10^{-6}$		$< 10^{-6}$		$< 10^{-6}$	
	star	μ	248	10320	221	9945	6880	27,050	4980	27,450
		σ	35	1750	37	1800	920	3750	680	4280
		p_t	$< 10^{-6}$		$< 10^{-6}$		$< 10^{-6}$		$< 10^{-6}$	

ity of the target graphs increases. A set of experiments was performed using several target graphs with different number of nodes and occurrence probability of arcs. The second problem regarded the configuration of a real world wireless network. This is a hard problem requiring the optimization of number, position and connection topology of the APs in a given area.

5. Results and Discussion

We devised an evolutionary algorithm for generating graphs, which does not require to fix the number of nodes of the graphs to be found a priori. The graph representation is based on a specifically devised data structure, called multilist, which uses a direct encoding scheme. This representation allows us to explore the graph search space effectively, particularly because it makes the implementation of the crossover operator very simple and effective. The defined crossover operator has three main properties: (i) it is able to generate variable size graphs; (ii) it splits a graph into two subgraphs without requiring any search procedure: (iii) it merges two subgraphs identifying the nodes to be connected in a simple and effective way

Three sets of experiments were performed to test the proposed approach. The first one regards the search for synthetically generated target graphs. It was considered in order to test the ability of the system to find solutions as the complexity of the target graphs increases. These experiments were performed using several target graphs with a different number of nodes and occurrence probability of arcs. In all the experiments, the proposed system was able to find the given target graphs. Moreover, the heritability of the crossover operator was also investigated. The results obtained show that this operator is able to preserve good solutions found during the evolution process and it effectively combines the meaningful features of the parents.
The second set regarded the configuration of a real world wireless network. This is a hard problem requiring the optimization of number, position and connection topology of the APs in a given area. The results obtained were compared with those achieved by a GP–based approach for graph generation, available in the literature. Our system was able to find good solutions even when the approach considered for comparison failed.
Finally, the third set of experiments deal with the standard one max tree problem in which a target spanning tree must be found. The results achieved were compared with those obtained by using a different tree representation scheme published in the literature, specifically devised for evolving spanning trees. The results showed that our

system is much faster than the approach considered for the comparison in finding the optimal solution, and that the performance differences strongly increase with the solution size.

6. Conclusions

We have proposed a new general purpose Evolutionary Computation based approach for evolving graphs (either simple graphs or ARG's).

The proposed approach solves two key problems encountered when an evolutionary algorithm is used for generating graphs: it provides a suitable data structure for the direct representation of graphs and defines effective operators for manipulating graphs with a variable number of nodes. Eventually, it has to be remarked that the proposed approach does not rely on any problem specific knowledge and therefore it is suitable for dealing with any problem whose solutions can be represented with graphs.

The main limitation of the proposed approach concerns the permutation problem, i.e. a many-to-one mapping from the genotype to the actual graph (phenotype). Even if this multiple mapping can in principle reduce the effectiveness in exploring the search space, all the experiments showed that our method allows very interesting results to be obtained.

Future work will include exploiting the general purpose nature of the proposed approach. In particular, pattern recognition applications in which the patterns to be recognized are represented by means of graphs, and systems evolving programs represented by graphs, will be considered.

Appendix

Graphs

A *graph* G with N nodes, is defined as $G = (V, E)$, where $V = \{n_1, n_2, \ldots, n_N\}$ is a set of *nodes* and $E = \{\langle n_i, n_j\rangle \,|n_i, n_j \in A\}$ is a binary relation defined on V, i.e. a set of ordered pairs of distinct elements in V. The couples in E are called *arcs*. A graph G is denominated *undirected* if the relation E is symmetric, *directed* otherwise. An *Attributed Relational Graph (ARG)* is a graph enjoying the following further properties:

1. there is a set, finite or infinite, A_n, called *node attribute set*, and a function ϕ_n:
$$\phi_n : V \to A_n$$
which binds an attribute (an element of the set A_n) to each node of the graph;
2. there is a set, finite or infinite, A_a, called *arc attribute set* and a function ϕ_a:
$$\phi_a : E \to A_a$$
which binds an attribute (an element of the set A_a) to each arc of the graph.

Random Graphs

Generating graphs according to a *random graph model* requires starting with a set of N nodes and randomly adding arcs between node pairs, according to a given probability distribution. Different random graph models produce graphs with different average number of arcs per node. The most commonly used model is the Erdös–Reényi one [39], called $\gamma(N, p)$, in which a graph G with N nodes is built by fixing a uniform probability p and by adding an arc between a pair of nodes, according to p, independently of the other arcs in G.

We adapted the $\gamma(N, p)$ model in order to generate undirected random ARG's. Given the number N of nodes, the node attribute alphabet A_n and the arc attribute alphabet A_a, the following algorithm was implemented:

```
for $i = 1$ to $N$ do
    $attribute(n_i)=rand(A_n)$;
end
for $i = 1$ to $N$do
    for $j = i$ to $N$do
        if $flip(p)$ then
            $attribute(\langle n_i, n_j\rangle)=rand(A_a)$;
        else
            $attribute(\langle n_i, n_j\rangle)=NULL$;
    end
end
```

The function *rand(A)* randomly picks an element from the alphabet A by using a uniform probability distribution. The function flip(p) returns the value 1 with a probability p and the value 0 with a probability $(1 - p)$.

References

[1] J. Gross, J. Yellen, Graph Theory and Its Application, McGrawHill, 2001.

[2] E. Cascetta, Transportation systems engineering: theory and methods, Kluwer Academic, 2001.

[3] M. Crow, Computational Methods for Electric Power Systems, CRC Press, 2003.

[4] W. Tsai, K. Fu, Error-Correcting Isomorphisms of Attributed Relational Graphs for Pattern Analysis, IEEE Trans. on SMC 9 (12) (1979) 757–768.

[5] M. A. Eshera, K. S. Fu, A graph distance measure between attributed relational graphs for image analysis, in: Proceedings of 7th Int. Conf. on Pattern Recognition, IEEE Press, 1984, pp. 75–77.

[6] M. Pelillo, K. Siddiqi, S. W. Zucker, Matching hierarchical structures using association graphs, Lecture Notes in Computer Science 1407 (1998) 3–13.

[7] C. Arcelli, L. Cordella, G. S. di Baja (Eds.), Visual Form 2001, LNCS 2059, Springer-Verlag, 2001.

[8] A. Filatov, A. Gitis, I. Kil, Graph-based handwritten digit string recognition, in: Proceedings of the Third International Conference on Document Analysis and Recognition (Volume 2), IEEE Computer Society, 1995, p. 845.

[9] L. P. Cordella, M. Vento, Symbol Recognition in Documents: A Collection of Techniques, International Journal on Document Analysis and Recognition (IJDAR) 3 (2) (2000) 73–78.

[10] L. P. Cordella, P. Foggia, C. Sansone, M. Vento, Learning Structural Shape Descriptions from Examples, Pattern Recognition Letters 23 (2002) 1427–1437.

[11] T. G. Dietterich, R. S. Michalski, A Comparative Review of Selected Methods for Learning from Examples, in: R. S. Michalski, J. G. Carbonell, T. M. Mitchell (Eds.), Machine Learning: An Artificial Intelligence Approach, Springer, Berlin, Heidelberg, 1984, pp. 41–81.

[12] Yao, Evolving Artificial Neural Networks, PIEEE: Proceedings of the IEEE 87 (9) (1999) 1423–1447.

[13] K. Chellapilla, D. B. Fogel, Evolving an expert checkers playing program without using human expertise, IEEE Trans. Evolutionary Computation 5 (4) (2001) 422–428.

[14] A. Globus, J. Lawtonb, T. Wipkeb, Automatic molecular design using evolutionary techniques, in: A. Globus, D. Srivastava (Eds.), The Sixth Foresight Conference on Molecular Nanotechnology, Westin Hotel in Santa Clara, CA, USA, 1998.

[15] D. Chen, T. Aoki, N. Homma, T. Terasaki, T. Higuchi, Graph-based evolutionary design of arithmetic circuits, IEEE Trans. Evolutionary Computation 6 (1) (2002) 86–100.

[16] G. L. Libralao, T. W. de Lima, K. Honda, A. C. B. Delbem, Node-depth encoding for directed graphs, in: The 2005 Congress on Evolutionary Computation (CEC05), IEEE, 2005, pp. 2196–2201.

[17] G. L. Libralao, F. C. Pereira, T. W. de Lima, A. C. B. Delbem, Node-Depth Encoding for Evolutionary Algorithms Applied to Multi-vehicle Routing Problem, in: M. Ali, F. Esposito (Eds.), IEA/AIE, Vol. 3533 of Lecture Notes in Computer Science, Springer, 2005, pp. 557–559.

[18] V. Maniezzo, Genetic Evolution of the Topology and Weight Distribution of Neural Networks, IEEE Transactions on Neural Networks 5 (1) (1994) 39–53.

[19] X. Yao, Y. Liu, A New Evolutionary System for Evolving Artificial Neural Networks, IEEE Transactions on Neural Networks 8 (3) (1997) 694–713.

[20] E. G. Carrano, R. H. C. Takahashi, C. M. Fonseca, O. M. Neto, Nonlinear Network Optimization - An Embedding Vector Space Approach, IEEE Trans. Evolutionary Computation 14 (2) (2010) 206–226.

[21] A. Moraglio, Y.-H. Kim, Y. Yoon, B. R. Moon, Geometric Crossovers for Multiway Graph Partitioning, Evolutionary Computation 15 (4) (2007) 445–474.

[22] H. Kitano, Designing Neural Networks Using Genetic Algorithms with Graph Generation, Complex Systems 4 (4) (1990) 461–476.

[23] J. R. Koza, F. H. B. III, D. Andre, M. A. Keane, Genetic programming III: darwinian invention and problem solving, Morgan Kaufmann, 1999.

[24] J. R. Koza, F. H. Bennett III, D. Andre, M. A. Keane, F. Dunlap, Automated Synthesis of Analog Electrical Circuits by Means of Genetic Programming, IEEE Transactions on Evolutionary Computation 1 (2) (1997) 109–128.

[25] K. Seo, Z. Fan, J. Hu, E. D. Goodman, R. C. Rosenberg, Toward an Automated Design Method for Multi-Domain Dynamic Systems Using Bond Graphs and Genetic Programming, Mechatronics 13 (2003) 851–885.

[26] J. Hu, E. Goodman, Wireless Access Point Configuration by Genetic Programming, in: Proceedings of the 2004 IEEE Congress on Evolutionary Computation, IEEE Press, Portland, Oregon, 2004, pp. 1178–1184.
URL `http://www.egr.msu.edu/ hujianju/evograph`

[27] S. Shirakawa, T. Nagao, Graph Structured Program Evolution: Evolution of Loop Structures, in: R. Riolo, U.-M. O'Reilly, T. McConaghy (Eds.), Genetic Programming Theory and Practice VII, Genetic and Evolutionary Computation, Springer US, 2010, pp. 177–194.

[28] S. Shirakawa, S. Ogino, T. Nagao, Graph structured program evolution, in: Proceedings of the 9th annual conference on Genetic and evolutionary computation, GECCO '07, ACM, New York, NY, USA, 2007, pp. 1686–1693.

[29] S. Harding, J. Miller, W. Banzhaf, Self Modifying Cartesian Genetic Programming: Fibonacci, Squares, Regression and Summing, in: L. Vanneschi, S. Gustafson, A. Moraglio, I. De Falco, M. Ebner (Eds.), Genetic Programming, Vol. 5481 of Lecture Notes in Computer Science, Springer Berlin / Heidelberg, 2009, pp. 133–144.

[30] J. Walker, J. Miller, The Automatic Acquisition, Evolution and Reuse of Modules in Cartesian Genetic Programming, Evolutionary Computation, IEEE Transactions on 12 (4) (2008) 397–417.

[31] A. Teller, M. Veloso, Pado: A New Learning Architecture For Object Recognition, in: Symbolic Visual Learning, Oxford University Press, 1995, pp. 81–116.

[32] R. Poli, Parallel Distributed Genetic Programming, Tech. rep., School of computer science University of Birmingham (1999).

[33] W. Kantschik, W. Banzhaf, Linear-Graph GP - A New GP Structure, in: Proceedings of the 5th European Conference on Genetic Programming, EuroGP '02, Springer-Verlag, London, UK, 2002, pp. 83–92.

[34] T. Eguchi, K. Hirasawa, J. Hu, N. Ota, A study of evolutionary multiagent models based on symbiosis, Systems, Man, and Cybernetics, Part B: Cybernetics, IEEE Transactions on 36 (1) (2006) 179–193.

[35] L. P. Cordella, C. De Stefano, F. Fontanella, A. Marcelli, EvoGeneS, a New Evolutionary Approach to Graph Generation, in: G. R. Raidl, J. Gottlieb (Eds.), Evolutionary Computation in Combinatorial Optimization – EvoCOP 2005, Vol. 3448 of LNCS, Springer Verlag, 2005, pp. 46–57.

[36] F. Rothlauf, D. E. Goldberg, A. Heinzl, Network Random Keys-A Tree Representation Scheme for Genetic and Evolutionary Algorithms, Evolutionary Computation 10 (1) (2002) 75–97.

[37] P. J. B. Hancock, Genetic Algorithms and Permutation Problems: a Comparison of Recombination Operators for Neural Net Structure Specification, in: D. Whitley (Ed.), Proceedings of COGANN workshop, IJCNN, Baltimore, IEEE, 1992.

[38] F. Menczer, D. Parisi, Evidence of Hyperplanes in the Genetic Learning of Neural Networks, Biological Cybernetics 66 (1992) 283–289.

[39] P. Erdös, A. Reényi, On the evolution of random graphs, Publ. Math. Inst. Hung. Acad. Sci 5 (1960) 17–61.

[40] G. Ochoa, Error thresholds in genetic algorithms, Evolutionary Computation 14 (2) (2006) 157–182.

[41] E. Koichi, W. Yoishinori, Automatic Cell Design for Wide Area Wireless LAN Systems, Special Issue on Devices and Systems for Mobile Communications 44(4).

[42] G. R. Raidl, B. A. Julstrom, Edge sets: an effective evolutionary coding of spanning trees, IEEE Trans. Evolutionary Computation 7 (3) (2003) 225–239.

[43] F. Rothlauf, On the Bias and Performance of the Edge-Set Encoding, IEEE Trans. Evolutionary Computation 13 (3) (2009) 486–499.

[44] D. E. Goldberg, Genetic Algorithms in Search, Optimization and Machine Learning, Addison-Wesley Longman Publishing Co., Inc., 1989.