29 Jan 1993

Genetic Algorithms for Vertex Splitting in DAGs

Matthias Mayer

Fikret Erçal
Missouri University of Science and Technology, ercal@mst.edu

Genetic Algorithms For Vertex Splitting in DAGs[1]

Matthias Mayer[2] and Fikret Ercal[3]

CSC-93-02

Fri Jan 29 1993

Department of Computer Science
University of Missouri-Rolla
Rolla, MO 65401, U.S.A.
(314) 341-4491

---

[1]This paper has been submitted to the 5th International Conference on Genetic Algorithms

[2]electronic mail address: matze@cs.umr.edu

[3]electronic mail address: ercal@cs.umr.edu

Genetic Algorithms For Vertex Splitting in DAGs

Abstract

Directed Acyclic Graphs are often used to model circuits and networks. The path length in such Directed Acyclic Graphs represents circuit or network delays. In the vertex splitting problem, the objective is to determine a minimum number of vertices from the graph to split such that the resulting graph has no path of length greater than a given δ. The problem has been proven to be NP-hard. A Genetic Algorithm is used to solve the DAG Vertex Splitting Problem. This approach uses a variable string length to represent the vertices that split the graph and a dynamic population size. The focus of this paper is the comparison of two methods to reduce the string length and of two stepping methods to explore the search space. Experimental results have shown that the multiple binary stepping method outperforms the linear stepping method in yielding better solutions.

Keywords:

Directed acyclic graph, vertex splitting, genetic algorithms, variable string length, dynamic population size, NP-hard.

1. Introduction

Genetic algorithms [9] (GAs) are adaptive search techniques that have been shown to be robust optimization algorithms. In contrast to other optimization techniques, genetic algorithms base their progress on the performance of a population of candidate solutions, rather than on one candidate solution. GAs are loosely based upon Darwin's principle of natural selection and natural genetics. They have become increasingly popular in recent years as a method for solving complex search problems [4].

The DAG vertex splitting problem addressed in this paper has many applications in the fields of computer science and electrical engineering. An application would be to find the minimum number of placements of signal boosters in a network. The placement of flip-flops in partial scan designs [11] is another application. Heuristics [11] have been used earlier to solve the DAG vertex splitting problem.

Section 2 introduces the DAG vertex splitting problem. The general outline of the genetic algorithm, along with a discussion about the functions that perform crossover, mutation, and recombination are discussed in Section 3. Section 4 explains the string length reduction techniques as well as the stepping methods to explore the search space. Experimental results are reported in Section 5. Section 6 summarizes the results and indicates future research areas.

2. The DAG Vertex Splitting Problem

The DAG vertex splitting problem (DVSP) can be stated as follows [11]:

Let G = (V, E, w) be a *weighted directed acyclic graph* (WDAG) with vertex set V, edge

set E, and edge function w. w(i, j) is the weight of the edge $\langle i, j \rangle \in E$. w(i, j) is a positive real number for $\langle i, j \rangle \in E$ and is undefined if $\langle i, j \rangle \notin E$. The *delay, d(P),* on the path P, is the sum of the weights of all the edges on that path. The *delay, d(G),* of the graph G is the maximum path delay in the graph. Figure 1 shows a DAG with a maximum path length of 3.

A *source vertex* is a vertex with no edge coming into the vertex and a *sink vertex* is a vertex with no edge leaving the vertex.

Let G/X be the WDAG that results when each vertex v in X is split into two vertices v^i and v^o such that all edges $\langle v, j \rangle \in E$ are replaced by edges of the form $\langle v^o, j \rangle$ and all edges $\langle i, v \rangle \in E$ are replaced by edges of the form $\langle i, v^i \rangle$. Outbound edges of v now leave v^o, while inbound edges of v now enter v^i. Figure 2 shows the result, G/X, when splitting vertex 3 of the DAG of Figure 1 into vertex 3^i and vertex 3^o.

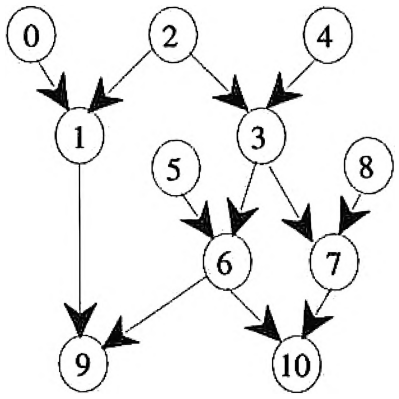

Figure 1: DAG with path length 3

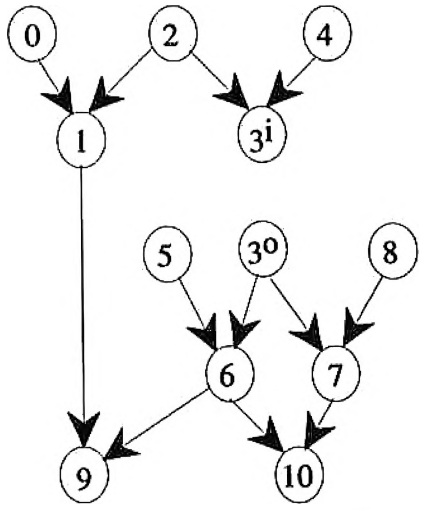

Figure 2: Vertex 3 split into vertex 3^i and vertex 3^o

Note that splitting either source or sink vertices does not reduce the path length.

The *DAG vertex splitting problem* (DVSP) is to find the least cardinality vertex set X such that $d(G/X) \leq \delta$, where δ is a pre-specified maximum delay. For the DAG of Figure 1 and $\delta = 2$, $X = \{3\}$ is a solution to the DVSP. Note that the DVSP has a solution iff $\max[w(i, j)] \leq \delta, \forall \langle i, j \rangle \in E$.

If $w(i, j) = 1 \ \forall \langle i, j \rangle \in E$ then the graph has unit weights[1]. It has been proven in [11] that finding a solution for DVSP is NP-hard for graphs with unit weights. Since the unit weight graphs are only a special case of general graphs the results also apply to the WDAG.

3. The Genetic Algorithm

The objective for the GA is to find a minimal set of vertices that split the graph such that the resulting graph has no path of length $> \delta$.

[1]All the graphs used in this paper have unit edge weights. Since it was not possible for us to determine the weight function used in [11], a direct comparison with the heuristics used in [11] was not possible.

The strings in each individual of the population represent the ***set of splitting vertices*** (or ***split set*** for short) that are used to split the graph. The genetic algorithm starts the search with an ***initial string length*** and continues with multiple rounds of optimization. Each optimization round tries to find a feasible solution with a fixed size split set. If a solution is found in a particular round, the next round attempts to shorten the string length and find a new solution with fewer number of vertices. Variable string length has been used before in GAs [3, 5, 6, 10].

A (suboptimal) solution for the DVSP is found, if the set of splitting vertices of one individual splits the graph in such a way that the resulting graph has a maximum path length $\leq \delta$. The genetic algorithm only works on one certain string length at a time. Different string lengths within a given population are not allowed simultaneously because it makes the GA more complex when applying the select and crossover functions.

Basically, the algorithm works as follows:

1) try to find a suboptimal solution by splitting x vertices
2) if a suboptimal solution is found, *reduce* the number of vertices and try again
3) if no solution has been found within a certain number of generations, *expand* the number of vertices and try again

Theoretically every vertex in the graph, excluding source and sink vertices, can be split. But some of these vertices might not be on a path whose length is greater than δ. Thus it would be worthless to consider them as potential vertices to split. A new set is introduced, called ***potential vertices***, which contains only those vertices that are on paths whose length is greater than delta. A solution to the DVSP exists, if all vertices in the potential vertex set are split. Thus the ***initial string length*** to start the GA would be the cardinality of the set of

potential vertices. For a fast reduction of the initial string length, a method called ***Binary Approximation (BA)*** was developed, which finds a better approximation for the initial string length. The function works as follows:

The solution to the DVSP has to lay somewhere between splitting one vertex and splitting all potential vertices. Thus a binary search is started midway between these two boundaries. A number of individuals is created randomly with a string length halfway between the lower and the upper bound. If a solution is found among these individuals, it is marked as a new upper bound for the BA. If no solution can be found then this is assumed to be a lower bound.

Experiments have shown that the BA does a fairly good job in reducing the initial string length. The number of created individuals is determined by the parameter *number of tries for BA*.

The initial string length is then used to create the initial population. It is important to note that a graph can not be split twice by the same vertex because after the first vertex splits the graph this vertex becomes a sink and a source vertex which can not be split anymore. Thus, a vertex can appear at most once in any particular split set.

The ***select*** function is a standard select which uses the roulette wheel. Instead of a linear search through the population, a binary search which returns the index of the selected individual has been implemented. Since the goal of optimization is to minimize the longest path in the graph by splitting vertices, the ***fitness function*** is defined to be (1/longest_path).

A so called ***uniform crossover*** [13, 14] function is used to generate offspring from the parents. The uniform crossover was shown to outperform the one-point and two-point crossover in most cases [13, 14]. While applying the uniform crossover, generation of multiple copies of the same vertex in a split set must be avoided. Consider the following

situation of two parents, a string length of 5, and a randomly generated crossover mask of 01110:

Parent 1:	15	7	8	19	2
Parent 2:	20	5	15	3	7
Mask:	0	1	1	1	0
Offspring 1:	***15***	5	***15***	3	2
Offspring 2:	20	***7***	8	19	***7***

In the above example, offspring 1 ends up getting vertex 15 twice and Offspring 2 ends up getting vertex 7 twice. This situation must be avoided. Thus all the vertices that appear in both parents are not allowed to undergo crossover. To do this, the duplicated vertices have to be determined in both parents and moved to the end of the set. This can be done because the order of the vertices in the set does not change the outcome of the graph after splitting. After the duplicated vertices have been moved to the end of the set the uniform crossover can be performed without further changes among the remaining vertices.

If these ideas are applied to the previous example, the result would be as follows:

Parent 1:	8	19	2	15	7
Parent 2:	20	5	3	15	7
Mask:	0	1	1	1	0
Offspring 1:	8	5	3	15	7
Offspring 2:	20	19	2	15	7

The last two bits in the uniform mask are not really necessary because they do not have any effect on the offspring.

Every crossover results in two offspring. The GA allows that two parents can produce

more than two offspring which results in a temporary increase in the population. This over population is reduced later by the recombination function. This way of doing the crossover was chosen to ensure that less fit offspring do not overwrite a more fit parent. The parameter *offspring per parents* regulates the number of offspring.

The ***mutation*** function operates only on the newly created offspring. Once a vertex has been chosen for mutation it is replaced by a new vertex picked from the potential vertex set that is not in the set of spitting vertices.

The ***recombination*** function takes the old population and the new offspring and reduces it down to the previous population size. The reduction is based upon the select function which ensures that fit individuals have a higher probability of survival. This means that individuals of the old population can survive into the new population while new offspring may die depending on their fitness values. The recombination function also makes sure that no individual appears twice in the new population.

It is possible that the DVSP has a solution with splitting only one vertex. This means that the string length is only one. If the crossover function is performed on individuals with a string length of one, no new individuals are introduced into the population. Only mutation can introduce new individuals. In order to avoid missing a solution with one vertex, a function called *take care of ones* tries every vertex in the potential vertex set one at a time to find a solution. This function is used before the GA is started and if this function finds a solution by spitting only one vertex, then there is no need to start the GA.

The pseudo code for the GA is shown in Figure 3.

```
BA;                    /* do the Binary Approximation */
take care of ones;     /* tries to find a solution with only one vertex */
create initial population;
while( !stop )
{
    determine new string length using a stepping function; /* see Section 5 */
    reduce the string length using a reduction method; /* see Section 5 */
    evaluate;
    for( i = 0; i < number of generations; i++ )
    {
        crossover;
        mutate;        /* mutate the new offspring */
        evaluate;      /* evaluate the new offspring */
        recombine;     /* old population and new offspring */
        evaluate;
    }
}
```

Figure 3: Pseudo code for the GA

4. String Length Reduction and Stepping Techniques

Why is a string reduction method necessary? The strings represent the vertices that are used to split the graph. The objective is to find a minimal set of vertices that split the graph. Thus, if the GA finds a suboptimal solution with splitting X vertices, it has to try to find a solution with splitting Y vertices, where $Y < X$.

Any string reduction method must address the following two problems: a) How to reduce the size of the split set? b) What should be the next size of the split set? Two strategies have been devised for each of the problems stated in a) and b) that are explained below.

To address a), two different ***deletion methods*** were developed. The first method is called

preserve duplicates. This method makes sure that the duplicate vertices in every individual do not get lost when the number of vertices is reduced. The intuition behind this strategy is that duplicates are important for getting a better fitness value since they have survived in multiple individuals throughout the regeneration process. Recall that the duplicate vertices are located at the end of the split set. Thus, to implement this strategy, the last vertices are moved to the beginning of the split set and the string length is reduced from the end of the split set leaving the duplicate vertices undeleted. The second method deletes a number of vertices randomly out of the split set and is therefore called ***random delete***.

After a suboptimal solution is found, the number of vertices in the split set has to be reduced. Two different, so called ***stepping methods*** were developed, to determine the string size to be tried next by the GA. The first method is called ***linear stepping***, which means that the string length is reduced by a positive integer every time a suboptimal solution is found. This integer is part of the parameter list and is called *strlen_decrement*. If no solution can be found with the reduced number of vertices the string length is incremented by one. The one new vertex that has to be introduced is taken from the set of potential vertices. The string length increment is performed until a new solution is found or until the string length exceeds the string length of the best solution found so far. The second method is called ***multiple binary stepping***. This method starts out with a lower bound of two (note that one vertex case has already been tested) and an upper bound obtained from the BA. The new string length is always determined by the formula (upper bound + lower bound)/2. Every time a solution is found, the population that yielded this solution is saved and the upper bound is set to the current string length. If no solution has been found the lower bound is set to the current string length. It is important to note that the split set is always reduced from the last saved population. This process repeats as long as a new solution with a smaller string length can be found.

5. Experimental Results

The genetic algorithm described in Section 3 and the four methods described above were implemented in C on a NeXTstation. The experiments are divided into four Test Beds (TBs):

- TB 1: preserve duplicates and linear stepping
- TB 2: preserve duplicates and multiple binary stepping
- TB 3: random delete and linear stepping
- TB 4: random delete and multiple binary stepping

The experiments were run on graphs derived from the ISCAS-85 benchmark combinational circuits [2]. The vertices in the DAG model the gates in the circuit and the edges correspond to the connections between the gates. The delay for each edge was set to one. The characteristics of the circuits used in this study are given in Table 1. The graph shown in Figure 1 was obtained from circuit c17.

Circuit	# vertices	# edges	d(G)
c17	11	12	3
c432	196	336	17
c880	443	729	24
c1355	587	1064	24

Table 1: Characteristics of four circuits from ISCAS-85 combinational benchmarks

Three graphs, c432, c880, and c1355 were selected for the tests. The following parameter settings were with different combinations:

crossover rate: 0.5, 0.6, 0.9

mutation rate: 0.001, 0.005

# of generations:	100, 200, 1000
population size:	50, 100, 200, 400, 800
offspring per parents:	2, 4, 8
# of tries for BA:	30, 100
strlen_decrement:	3

Test results obtained after extensive experimentation can be summarized as follows:

- a crossover rate of 0.5 yielded better solutions than 0.6 or 0.9
- a mutation rate of 0.005 yielded better solutions than 0.001
- on the average, 100 generations were sufficient enough to find a satisfactory suboptimal solution.
- the BA method returned initial string lengths which are 60-70% smaller than the cardinality of the potential vertex set
- the more offspring per parents, the better the solutions

The tests also showed that test bed 4 and 2 yielded the best solutions on the average. This is due to the fact that both test beds use the multiple binary stepping which does a more extensive search than linear stepping. Because of this more extensive search it also needs more computation time than linear stepping. Table 2 shows a ranking of all four test beds with respect to solution quality and run time. The solutions and run times were averaged over multiple runs.

TB	Solution Quality	Run Time
1	4	1
2	2	3
3	3	2
4	1	3

Table 2: Ranking of four genetic algorithms using different string reduction methods

The results also indicate that the method, *preserve duplicates* (TB 1 and TB 3) does not help much in finding better solutions. It mostly gets trapped in a local optima.

6. Conclusions

This paper introduced the DAG Vertex Splitting Problem (DVSP) and described a Genetic Algorithm to solve the DVSP. The GA described here is not a standard one. It uses a variable string length and a variable population size. Two different deletion methods were developed to determine the vertices that are to be deleted in conjunction with two different stepping methods to determine the new string length.

Experiments were conducted with the described methods on three graphs obtained from the ISCAS-85 benchmark combinational circuits. The results from the experiments showed that the method, *multiple binary stepping* outperforms *linear stepping* in obtaining better solutions. On the other hand, *linear stepping* has a better run time than that of *multiple binary stepping*. The experiments also showed that the method *preserve duplicates* gets stuck in local optima and therefore does not yield good solutions.

Future research in this area will include an implementation of the GA onto a parallel machine to reduce the run time [12, 15]. In order to avoid premature convergence, different select strategies as suggested in [1] are planned to be implemented and tested. It is also worthwhile to look into a GA where select, crossover and mutate are guided by heuristics [8, 11]. Another method of solving the DVSP would be to start with a string length of one and to increase the string length until a solution can be found.

References

[1] Baker, James E. (1985). *Adaptive Selection Methods for Genetic Algorithms.* Proceedings of an International Conference on Genetic Algorithms and Their Applications, pp. 101-111.

[2] Brglez, F. and Fujiwara, H. (1985). *A Neutral Netlist of Ten Combinational Benchmark Circuits and a Target Translator in Fortran.* Proceedings IEEE Symposium on Circuits and Systems, pp. 663-666.

[3] Cramer, N.L. (1985). *A representation for the adaptive generation of simple sequential programs.* Proceedings of an International Conference on Genetic Algorithms and Their Applications, pp. 183-187.

[4] Goldberg, D.E. (1989). *Genetic Algorithms in Search, Optimization & Machine Learning.* Addison-Wesley

[5] Goldberg, D.E, Korb, B., Deb, K. (1989). *Messy Genetic Algorithms: Motivation, Analysis, and First Results.* Complex Systems 3, pp. 493-530.

[6] Goldberg, D.E., Deb, K., Korb, B. (1990). *Messy Genetic Algorithms Revisited: Studies in Mixed Size and Scale.* Complex Systems 4, pp. 415-444.

[7] Grefenstette, J.J. (1986). *Optimization of Control Parameters for Genetic Algorithms.* IEEE Transactions on Systems, Man, and Cybernetics. vol. SMC-16, no. 1, pp. 122-128.

[8] Grefenstette, J.J. (1987). *Incorporating Problem Specific Knowledge into Genetic Algorithms.* Genetic Algorithms and Simulated Annealing, L. Davis, ed. (Pitman, London, 1987), pp. 42-60.

[9] Holland, J.H. (1975). *Adaption in natural and artificial systems.* Ann Arbor: The University of Michigan Press

[10] Jujiko, C. and Dickinson, J. (1987). *Using the genetic algorithm to generate LISP*

source code to solve the prisoner's dilemma. Proceedings of the Second International Conference on Genetic Algorithms, pp. 236-240.

[11] Paik, D., Reddy, S., and Sahni, S. (1990). *Vertex Splitting in Dags and Applications to Partial Scan Designs and Lossy Circuits.* Technical Report TR90-034, University of Florida

[12] Petty, C.B., Leuze, M.R., and Grefenstette, J.J. (1987). *A Parallel Genetic Algorithm.* Proceedings of the Second International Conference on Genetic Algorithms, pp. 155 - 161.

[13] Spears, W.M. and Anand, V. (1991). *A Study of Crossover Operators in Genetic Programming.* Sixth International Symposium on Methodologies for Intelligent Systems. Charlotte, NC, pp. 409-418.

[14] Syswerda, G. (1989). *Uniform Crossover in Genetic Algorithms.* Proceedings of the Third International Conference on Genetic Algorithms. Morgan Kaufmann Publishers,, pp. 2-9.

[15] Tanese, R. (1987). *Parallel Genetic Algorithm for a Hypercube.* Proceedings of the Second International Conference on Genetic Algorithms, pp. 177-183.