Crossover and Mutation in Genetic Algorithms Using Graph-Encoded Chromosomes

Sam Stone[1], Brian Pillmore[2], and Walling Cyre[3]

[1] 905 McBryde Lane #2,
Blacksburg, VA 24060
sastone@vt.edu
[2] 1019 West AJ, Virginia Tech,
Blacksburg, VA 24061
bpillmor@vt.edu
[3] 340 Whittemore Hall, Virginia Tech,
Blacksburg, VA 24061
cyre@vt.edu
http://www.ece.vt.edu/

Abstract. Graph chromosomes provide an elegant and flexible structure whereby genetic algorithms can encode applications not easily represented by the conventional vector, list, or tree chromosomes. While general-purpose mutation operators for graph-encoded genetic algorithms are readily available, a graph-encoded GA also requires a general-purpose crossover operator that enables the GA to efficiently explore the search space. This paper describes the existing graph crossover operators and proposes a new crossover operator, GraphX. By operating on the graph's representation, rather than the graph's structure, the GraphX operator avoids the unnecessary complexities and performance penalties associated with the existing fragmentation/recombination operators. Experiments verify that the GraphX operator outperforms the traditional fragmentation/recombination operators, not only in terms of the fitness of the offspring, but also in terms of the amount of CPU time required to perform the crossover operation.

1 Introduction

In a given application domain, genetic algorithms use the principles of genetics and natural selection to evolve optimal solutions to specified problems, or to adapt existing solutions to changing environments [5]. With search spaces encoded as vector, list, and tree chromosomes, genetic algorithms have demonstrated success in a wide variety of applications [4], including digital circuit design, VLSI cell placement, traveling salesman problems, artificial intelligence, and adaptive filtering.

However, for many applications of interest, including the evolution of digital circuits and biological molecules, graph-encoded chromosomes provide the most elegant, flexible, and intuitive structure whereby a genetic algorithm can encode the application's search space. While graph mutation operators are readily available, the existing graph crossover operators simply do not perform well in more than a handful

of narrowly defined application domains. This paper describes the existing graph crossover operators, which tend to divide each parent into two disjoint fragments and then merge fragments from opposite parents to form offspring. After discussing the unnecessary complexities, restrictions, and performance penalties associated with these fragmentation/recombination operators, this paper presents a new crossover operator, GraphX. Modeled after the traditional 2-point crossover operator used on vector and list chromosomes, the GraphX operator avoids the pitfalls of fragmentation and recombination by operating on the graph's representation rather than on the graph's actual structure. Experiments verify that the GraphX operator outperforms the traditional fragmentation/recombination operators, not only in terms of the fitness of the offspring, but also in terms of the amount of CPU time required to perform the crossover operation.

2 Related Work

A graph can be represented directly as a linked data structure, or indirectly as an adjacency matrix or adjacency list [2]. If the graph's nodes have types associated with them, then the indirect representations must also represent the node types; a vector is sufficient for this task. Thus, an arbitrary graph of any size and any structure can be completely represented by a two-dimensional *adjacency matrix* and a one-dimensional *types array*. While effective crossover operators that manipulate two-dimensional matrices do exist, such as the subsequence exchange crossover of [6] and the window crossover of [8], surprisingly little effort has been expended in adapting these operators to the problem of graph crossover.

2.1 Fragmentation and Recombination Operators

Much of the research related to crossover operators for graph chromosomes has focused on techniques that first divide each parent graph into disjoint fragments (where each fragment is a connected sub-graph), then merge fragments from different parents to produce offspring [1], [3]. While these operators have demonstrated success in certain applications, such as the evolution of self-organizing maps [1] and the evolution of biological molecules [3], an effective general-purpose crossover operator for graph chromosomes has not emerged.

2.2 Globus Crossover

The crossover operator devised by Globus, Atsatt, Lawton, and Wipke is representative of the best work in this area. While many graph crossover operators place substantial restrictions on the structure of the parent graphs [1], the Globus operator can operate on any connected graph, directed or undirected [4]. (In fact, the Globus operator can also operate on graphs that are not connected; however, in that case, the operator really only operates on one connected fragment within the graph; all nodes

and edges outside that fragment are simply discarded.)

As described in [3] and [4], the Globus operator implements graph crossover according to the following algorithm:

To divide a graph into two fragments

1. Choose an initial random edge
2. Repeat until a cut set is found
    a. Find the shortest path between the initial edge's vertices
    b. Remove a random edge in this path from the graph
    c. Save the removed edge in the cut set

To merge two fragments into a graph

Repeat until each broken edge has been processed

1. Select a random broken edge in either fragment's cut set
2. If at least on broken edge exists in the other fragment's cut set
    a. Choose one such edge at random
    b. Merge the broken edges
3. Else flip a coin
    a. If heads, attach broken edge to random node in other fragment
    b. If tails, discard the broken edge

As an example, consider Fig. 1-3, in which the Globus operator crosses an 8-node parent with a 6-node parent, yielding a 10-node child.

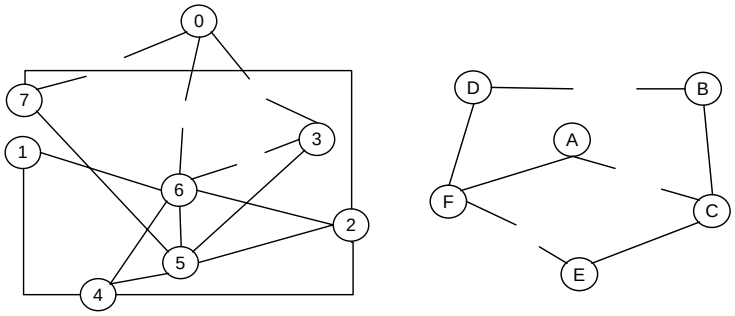

Fig. 1. The operator divides each parent into disjoint fragments. For the 8-node parent, the operator randomly selects edge 0-3 as the initial edge. The shortest path between nodes 0 and 3 is the path {0-3}; therefore, the operator breaks edge 0-3. In the resulting graph, the shortest path between nodes 0 and 3 is the path {0-6, 6-3}; the operator randomly breaks edge 6-3. The shortest paths remaining between the terminal nodes are the paths {0-7, 7-5, 5-3} and {0-6, 6-5, 5-3}. In successive iterations, the operator randomly breaks edges 0-7 and 0-6, yielding a 1-node fragment and a 7-node fragment. Through the same algorithm, the operator breaks edges A-C, F-E, and D-B to divide the 6-node parent into two 3-node fragments

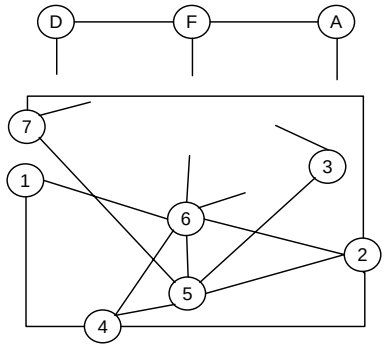

Fig. 2. The operator selects one fragment from each parent graph to be merged into a child graph

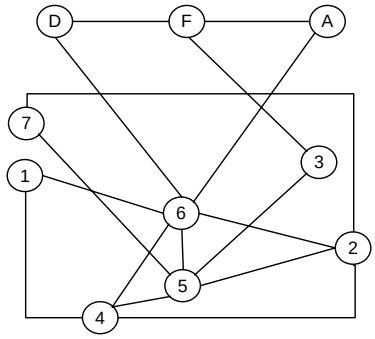

Fig. 3. The operator merges the fragments to form the offspring. The operator randomly selects a broken edge on node 6 and randomly merges it with the broken edge on node D. Successive iterations of the algorithm yield the merged edges 3-F and 6-A, leaving a lone broken edge on node 7. The operator randomly chooses to discard the last broken edge

As the preceding example demonstrates, the Globus algorithm implements a straightforward, intuitive crossover operation on two graphs. However, a closer examination of the algorithm and the example reveals that the Globus operator induces several undesirable side effects into the genetic algorithm. First, as Fig. 1 shows, the Globus operator tends to divide a parent graph into two very unevenly sized fragments. This phenomenon occurs because every path between the initial edge's terminal nodes must include edges that are incident on the terminal nodes. Therefore, over many iterations of the operator's fragmentation algorithm, the edges that are incident to the path's terminal nodes have a much greater probability of breaking than any other edges in the graph. Obviously, as the number of nodes and edges in the parent graph increases, the rate at which the Globus operator unevenly fragments the parent increases. Furthermore, as Fig. 1 shows, the Globus operator tends to break edges *within* the larger fragment of the parent graph – edges that ultimately did not need to

be broken to divide the parent graph into the resulting fragments. Again, as the number of nodes and edges in the parent graph increases, the rate at which the Globus operator unnecessarily breaks edges *within* a fragment of the parent graph increases. Collectively, these two side effects of the Globus operator make it very difficult for the genetic algorithm to identify and preserve high-fitness building blocks during crossover. Upon observing the operator's poor performance in evolving digital circuits, the authors noted in [4], "it may be that the crossover operator does not preserve useful sub-graphs very often…the extremely destructive nature of the crossover operator…can be expected to generate many very unfit children from fit parents."

3 GraphX Crossover

The GraphX operator avoided the unnecessary complexities and performance penalties associated with fragmentation and recombination by operating on the graph's representation, rather than on the graph's structure. That is, GraphX simply performed 2-point crossover on the adjacency matrices and on the matrices of node types (*types matrices*).

3.1 GraphX Crossover on Graphs of the Same Size

In the simplest case, where the parent graphs contained the same number of nodes (N), the GraphX operator simply performed 2-point crossover on the parents' adjacency matrices, selecting the crossover points randomly in the N-by-N matrices. The operator then performed 2-point crossover on the parents' types matrices, with the crossover points again selected randomly. Each offspring then contained the same number of nodes as the parents, but contained edges and node types inherited from both parents. This property of the GraphX operator allowed the GA to efficiently explore the search space for graphs of a particular size.

3.2 GraphX Crossover on Graphs of Different Sizes

To perform crossover on a (smaller) graph of size S and a (larger) graph of size L, the GraphX operator first increased the size of the smaller graph to match the size of the larger graph. In increasing the size of the smaller graph, the operator simply added L-S nodes to the graph; no edges were added to connect these nodes to the graph or to each other. In other words, the additional nodes were *placeholders* – empty rows and columns appended to the adjacency matrix (and empty nodes added to the types matrix) to permit efficient 2-point crossover. Conceptually, this algorithm was consistent with the desired properties of a crossover operator. The two graphs not only exchanged information about high-fitness interconnections (edge placements and weights) between the S nodes that were shared by both graphs, but also exchanged information about the number of nodes that a highly fit graph should contain. The smaller graph's placeholders encouraged the larger graph to disconnect its additional

L-S nodes; the larger graph's interconnections to its additional L-S nodes encouraged the smaller graph to increase its size.

Following the 2-point crossover, the size of the offspring corresponding to the larger parent remained fixed at L nodes. However, the size of the offspring corresponding to the smaller parent varied between S and L nodes, depending on the location of the second crossover point in the adjacency matrix. If the second crossover point was located in a row representing one of the graph's original nodes, then the GraphX operator discarded all the placeholder nodes, trimming the offspring to its original size of S nodes. If the second crossover point was located in a row representing one of the offspring's placeholder nodes, then the GraphX operator preserved that placeholder node and all placeholder nodes preceding it, while discarding all subsequent placeholder nodes. Consider the following example:

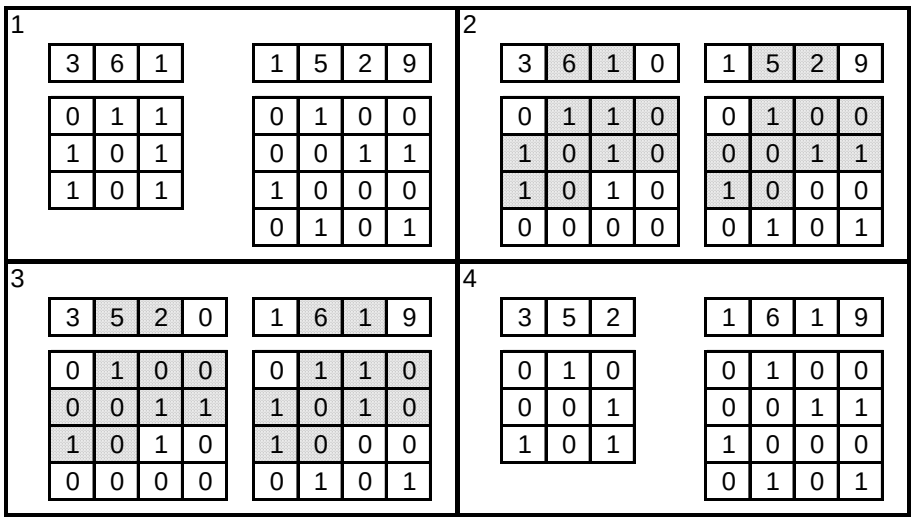

Fig. 4. Performing GraphX Crossover on Graphs of Different Sizes. Each panel depicts the types matrix and the adjacency matrix of each graph. The first panel depicts two parent graphs of sizes 3 and 4 nodes, respectively. In the second panel, the GraphX operator increases the size of the smaller parent to 4 nodes by appending a placeholder node. The second panel also shows the location of the crossover points in both the adjacency matrices and the types matrices; values between the crossover points appear in shaded cells. In the third panel, the operator performs the 2-point crossover by exchanging the shaded values, yielding two offspring that each contain 4 nodes. In the final panel, the operator trims the size of the first child to 3 nodes because the crossover point was in the third row of the adjacency matrix

3.3 GraphX Crossover with Expansion

While the normal mode of GraphX crossover permitted the GA to efficiently explore the search space of all graphs with sizes between the smallest and largest graph in the population, the normal mode did not contain a mechanism whereby the crossover

operation could create a child graph containing more nodes than the larger parent graph. To enable GraphX crossover to generate a child graph containing more nodes than the larger parent (thereby enabling the GA to efficiently explore the search space of *all* graphs), we added an expansion mode to the GraphX operator.

For a given pair of parent graphs, the GraphX operator performed expansion crossover with the user-specified probability *expand_prob* (typically 0.05 or smaller). In expansion mode, the GraphX operator first increased the size of the larger parent by one node, and then increased the size of the smaller parent to match the expanded size of the larger parent. Unlike the placeholder nodes in the non-expansion crossover, these *expansion nodes* were randomly connected to each other and to other nodes in the graph. Finally, as in the normal mode, the GraphX operator performed 2-point crossover on the parents, then trimmed the size of the offspring corresponding to the smaller parent, based on the position of the second crossover point. Consider the following example:

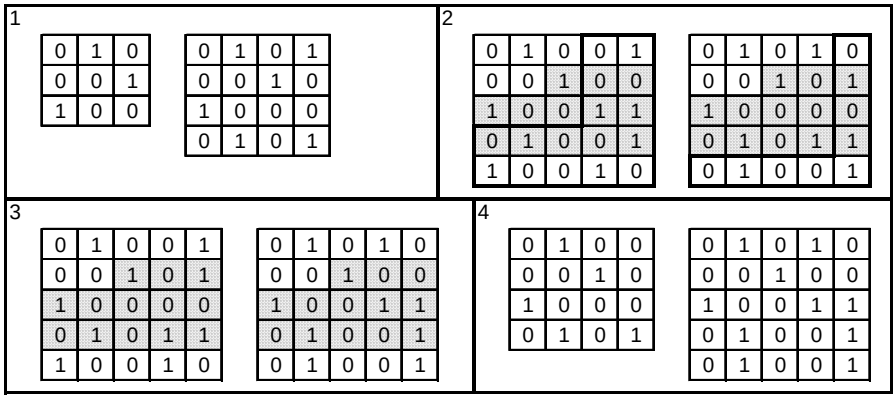

Fig. 5. Performing GraphX Crossover in Expansion Mode. Each panel depicts the adjacency matrix of each graph; node type matrices are omitted for brevity. The first panel depicts two parent graphs of sizes 3 and 4 nodes, respectively. In the second panel, the GraphX operator increases the size of the each parent to 5 nodes by appending 2 expansion nodes to the smaller graph and 1 expansion node to the larger graph. The second panel also shows the locations of the crossover points in the adjacency matrices; edges between the crossover points appear in shaded cells. In the third panel, the operator performs the 2-point crossover by exchanging the shaded edges, yielding two offspring of size 5 nodes. Finally, the operator trims the size of the first offspring to 4 nodes because the crossover point was in the fourth row of the adjacency matrix.

4 Mutation

To augment the GraphX operator, we implemented a set of four general-purpose mutation operators, which mutated a graph by (1) adding an edge, (2) removing an edge, (3) adding a node, and (4) removing a node, respectively [1], [4]. After per-

forming crossover on the entire population, the GA selected an offspring for mutation with user-specified probability *mutation_prob* (typically 0.005). If a graph was selected for mutation, the GA randomly carried out one of the aforementioned mutations.

To add an edge to a graph G of N nodes, the operator randomly selected an edge from the complete graph on N nodes that was not already present in the graph G and added that edge to graph G. The operator also assigned a random edge weight to the new edge. To remove an edge from a graph G, the operator randomly selected an existing edge in G and removed that edge from the graph.

To add a node to a graph G, the operator appended a new row and column to the graph's adjacency matrix, and appended a new entry to the graph's matrix of node types. The operator then assigned a random type to the new node, and randomly added edges between the new node and the graph's existing nodes. To remove a node from graph G, the operator deleted the last row and column from the graph's adjacency matrix, and deleted the last entry from the graph's matrix of node types. Thus, the operator removed the node and all edges incident on it.

5 Data and Results

To determine the effectiveness of the crossover and mutation operators, we designed a genetic algorithm that attempted to evolve a *target graph*. The target graph was completely and uniquely specified by its adjacency matrix and by the integer type assigned to each of its nodes. Each chromosome in the GA was a graph, represented internally by an adjacency matrix of edge weights and a types vector of node types. The GA permitted the user to select Globus crossover or GraphX crossover, and to enable or disable graph mutation. To determine the fitness of a graph, the GA compared that graph to the target graph using the following algorithm, where T_ and C_ represent properties of the target graph and the chromosome graph, respectively:

```
1.  maxFitness = 10*T_numNodes + sum(T_edgeWeights) + sum(T_nodeTypes)
2.  minNodes = min(T_numNodes, C_numNodes)
3.  nodePenalty = 10*abs(T_numNodes – C_numNodes)
4.  For row = 1:minNodes
        a.  typePenalty += abs(T_nodeType – C_nodeType)
        b.  For col = 1:minNodes
                edgePenalty += abs(T_edgeWeight – C_edgeWeight)
5.  If (T_numNodes > C_numNodes)
        a.  For each extra node in target graph
                typePenalty += T_nodeType
        b.  For each extra node pair in target graph
                edgePenalty += T_edgeWeight
6.  Else If (C_numNodes > T_numNodes)
        a.  For each extra node in chromosome graph
                typePenalty += C_nodeType
        b.  For each extra node pair in chromosome graph
                edgePenalty += C_edgeWeight
7.  fitness = maxFitness – nodePenalty – edgePenalty – typePenalty
```

Note that the evaluation algorithm strongly encouraged the GA to evolve graphs

containing the same number of nodes as the target graph, thereby minimizing the search space. Furthermore, the algorithm penalized every erroneous arc weight (and every erroneous node type) in the chromosome, and the penalty was always the difference between the chromosome's edge weight (or node type) and the target graph's edge weight (or node type).

The foregoing experiments were conducted with the following set of GA parameters. A distributed GA with 10 islands was used because the GraphX operator performed better in a distributed GA, while the Globus operator performed as well in the distributed GA as it did in a serial GA. Each island contained a population of 100 graphs, with the each graph in the initial population containing at most 5 nodes.

islands	10	selection	stochastic
epoch	100	crossover	2-point or globus
migrants	20	scaling	linear_fitness
pop_size	100	crossover_prob	.96
replacement	elitism	expand_prob	0.05
elites	1	mutation	none or uniform
graph_nodes	5	mutation_prob	.005

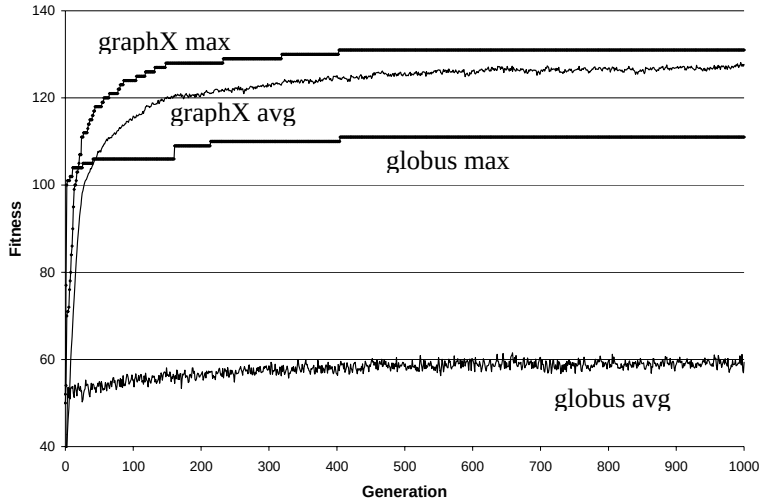

Fig. 6. GraphX Crossover Versus Globus Crossover (Fitness). These fitness curves represent typical runs in which the GA attempted to evolve a 10-node graph with fitness 131

5.1 GraphX Crossover Versus Globus Crossover

In the first experiment, we attempted to evolve a directed graph containing 10 nodes and 25 edges, with a maximum fitness of 131. As a means of minimizing the search space, the target graph contained nodes of only two types (0 and 1), and the target graph's arcs were not weighted. Fig. 6 shows a typical run of the GA with Globus crossover enabled, and a typical run of the GA with GraphX crossover en-

abled. As the figure indicates, GraphX crossover outperformed Globus crossover by a significant margin. In 20 runs of 1000 generations, the GraphX operator evolved a graph with the maximum fitness 11 times, while the Globus operator never evolved a graph with fitness greater than 113.

The poor performance of the Globus operator derived largely from the operator's tendency to disrupt high-fitness building blocks. However, it should be noted that the GA did not attempt to account for isomorphism when evaluating the fitness of a graph. Because Globus crossover truly operated on the graph's structure, rather than on the graph's representation, it is possible that the operator tended to create graphs that were more nearly isomorphic to the target graph than Fig. 6 suggests.

Nevertheless, the superior performance of the GraphX operator should not be discounted. In addition to outperforming the Globus operator for the given application, the GraphX operator also performed crossover on an arbitrary pair of graphs much more quickly and efficiently than did the Globus operator. As Fig. 7 indicates, the time required for the GraphX operator to perform crossover on two graphs increased in proportion to the number of nodes in each graph. However, the amount of time required for the Globus operator to perform crossover on two graphs increased in proportion to the number of nodes *and* the number of edges in each graph. The undesirable increase in the Globus operator's execution time stems primarily from the operator's frequent use of the expensive breadth-first search algorithm.

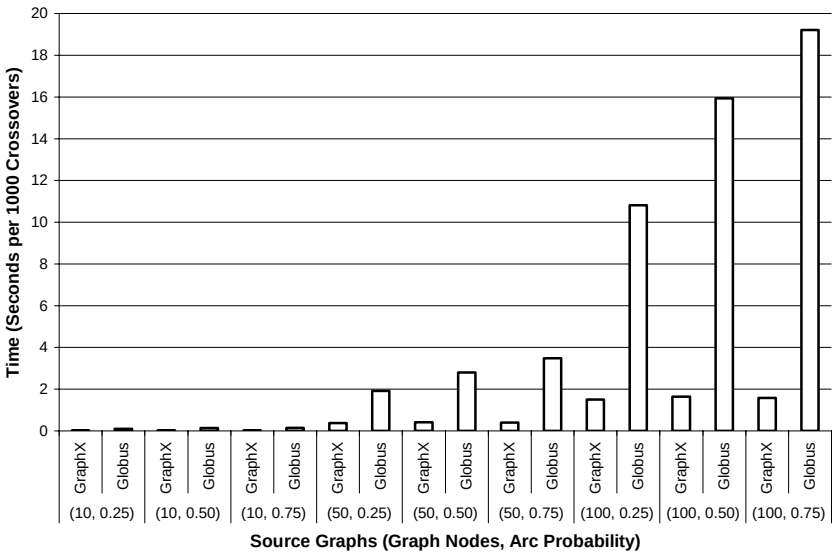

Fig. 7. GraphX Crossover Versus Globus Crossover (Execution Time). Depicts the time required for each operator to perform 1000 crossover operations on two graphs containing the specified number of nodes, with the specified probability of an edge existing between any two nodes in either graph. The authors obtained the data by iteratively crossing over two graphs with the specified properties and discarding the results

5.2 Mutation

Finally, the mutation operators were tested. With GraphX crossover selected, we attempted to evolve a directed graph containing 40 nodes, 159 edges, and maximum fitness 577. As a means of minimizing the search space, the target graph contained nodes of only two types (0 and 1), and the target graph's edges were not weighted. Fig. 8 shows 5 typical runs of the GA with mutation enabled, and 5 typical runs of the GA with mutation disabled. As the figure indicates, the GA with mutation enabled outperformed the GA with mutation disabled. In particular, the effectiveness of the mutation operators increased as the number of generations increased. In the GA with mutation disabled, the maximum fitness leveled off around 460 after nearly 1500 generations. In the GA with mutation enabled, the maximum fitness continued to steadily improve for the duration of each run. Considering that GraphX expansion crossover incorporated the behaviors of all the mutation operators, it seems reasonable to infer that the mutation operators were most beneficial after the graphs in the population stabilized at the target size (and stopped expanding).

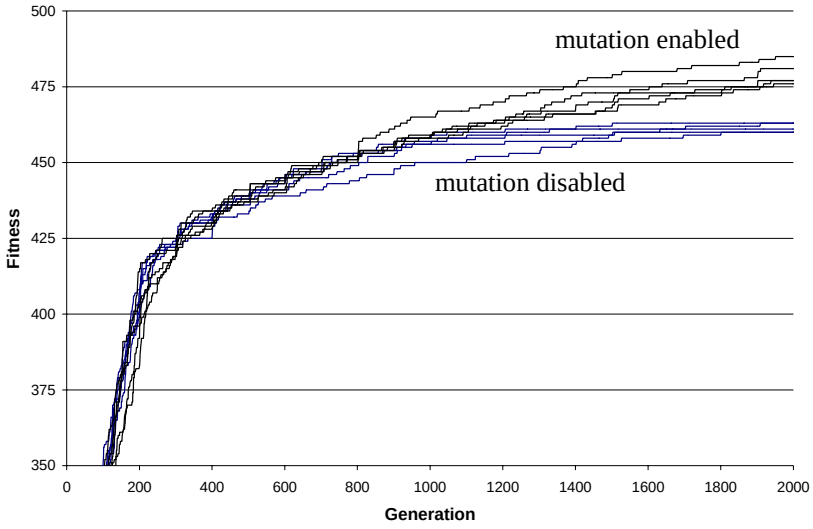

Fig. 8. Benefits of Mutation. Depicts the maximum fitness found by the GA for five runs with mutation enabled and five runs with mutation disabled. The GA with mutation enabled consistently outperformed the GA with mutation disabled

6 Summary

The development of efficient, general-purpose graph crossover and graph mutation operators is absolutely essential if genetic algorithms are to make effective use of graph representations. While existing mutation operators are adequate, the traditional

fragmentation/recombination crossover operators are extremely inefficient. These operators tend to disrupt the high-fitness building blocks that are crucial to the success of any genetic algorithm.

The GraphX crossover operator implements a new and promising approach to graph crossover. Because GraphX crossover does not operate directly on the graph's structure, the operator avoids the unnecessary complexities and performance penalties associated with fragmentation/recombination operators. Modeled after existing vector crossover and matrix crossover techniques, GraphX crossover implements an efficient algorithm that finds and preserves high-fitness building blocks in the search space. The GraphX operator can perform crossover on any graph structure, including directed or undirected graphs, graphs with or without arc weights, and graphs with or without node types. Furthermore, given an arbitrary population of graphs, a genetic algorithm using GraphX crossover can evolve any graph that is not smaller than the smallest graph in the initial population. Early experiments indicate the GraphX operator outperforms existing fragmentation/recombination operators by a significant margin.

In the future, the GraphX operator must be tested against existing crossover operators in a traditional application domain, such as the evolution of digital circuits. It is probable that the GraphX operator would benefit from a *contraction mode* to complement the existing *expansion mode*. It is also possible that the efficiency of the GraphX operator could be improved if the graph were represented by an adjacency list rather than by an adjacency matrix (particularly for large graphs that are not particularly full). However, it is clear that GraphX crossover already offers significant advantages over current graph crossover operators.

References

1. Chang, M., Heh, J. Implements an Evolutionary Self-Organizing Map Based Graph Evolution. http://www.fuzzy.org.tw/5-2t.htm
2. Dossey, J.A., Otto, A.D., Spence, L.E., Eynden, C.V. Discrete Mathematics. 3rd edn. Addison-Wesley, Illinois State University (1998) 101-160
3. Globus, A., Lawton, J., Wipke, T. Automatic Molecular Design Using Evolutionary Techniques. In: Nanotechnology, Vol. 10, No. 3 (1999) 290-299
4. Globus, A., Atsatt, S., Lawton, J., Wipke, T. JavaGenes: Evolving Graphs with Crossover. NAS Technical Report NAS-00-018, October 2000
5. Goldberg, D. Genetic Algorithms in Search, Optimization, and Machine Learning. Addison-Wesley, University of Alabama (1989)
6. Kobayashi, S., Ono, I., Yamamura, M. An Efficient Genetic Algorithm for Job Shop Scheduling Problems. In: Proceedings of ICGA (1995) 506-511
7. Poli, R. Evolution of Graph-Like Programs with Parallel Distributed Genetic Programming. In: Goodman, E. (ed.): Proceedings of Seventh International Conference on Genetic Algorithms. Morgan Kaufman, Michigan State University, East Lansing (1997) 346-353.
8. Valenzuela, J., Smith, A. A Seeded Memetic Algorithm for Large Unit Commitment Problems. In: Journal of Heuristics, Vol. 8, No. 2 (2002) 173-195