Genetic algorithms for the traveling salesman problem using edge assembly crossovers

Dwain Alan Seppala

University of Nevada, Las Vegas

Follow this and additional works at: https://digitalscholarship.unlv.edu/rtds

Repository Citation

Seppala, Dwain Alan, "Genetic algorithms for the traveling salesman problem using edge assembly crossovers" (2003). UNLV Retrospective Theses & Dissertations. 1542.
https://digitalscholarship.unlv.edu/rtds/1542

This Thesis is brought to you for free and open access by Digital Scholarship@UNLV. It has been accepted for inclusion in UNLV Retrospective Theses & Dissertations by an authorized administrator of Digital Scholarship@UNLV. For more information, please contact digitalscholarship@unlv.edu.
GENETIC ALGORITHMS FOR THE TRAVELING SALESMAN PROBLEM

USING EDGE ASSEMBLY CROSSEOVERS

by

Dwain Alan Seppala
Bachelor of Science in Mathematics
University of Illinois at Chicago
1987

Bachelor of Fine Art
Drake University, Des Moines, Iowa
1987

A thesis submitted in partial fulfillment
of the requirements for the

Master of Science Degree in Computer Science
Department of Computer Science
Howard R. Hughes College of Engineering

Graduate College
University of Nevada, Las Vegas
August 2003
Thesis Approval
The Graduate College
University of Nevada, Las Vegas

July 25th, 2003

The Thesis prepared by
Dwain Alan Seppala

Entitled
Genetic Algorithms for the Traveling Salesman Problem
Using Edge Assembly Crossovers

is approved in partial fulfillment of the requirements for the degree of
Master of Science in Computer Science

Examination Committee Chair

Dean of the Graduate College

Examination Committee Member

Examination Committee Member

Graduate College Faculty Representative

1017-03
ABSTRACT

Genetic Algorithms for the Traveling Salesman Problem
Using Edge Assembly Crossovers

by

Dwain Alan Seppala

Dr. Bein, Examination Committee Chair
Professor of Computer Science
University of Nevada, Las Vegas

The central issue in creating new genetic algorithms is the algorithm's crossover method. My focus is on a particular crossover known as the Edge Assembly Crossover, or EAX, by Nagata and Kobayashi. The basics of what make up a genetic algorithm is reviewed. The traveling salesman problem is defined. The EAX as an algorithm within an algorithm is explained. The crossover's implementation is original and is listed. The use of the graphic user interface, TSP View, used to run algorithms is explained as well as the extensions to the interface that were implemented for this study. The results of running a genetic algorithm using the EAX against traveling salesman problems, with a focus on ATT532, is discussed and compared to runs using other optimization algorithms. The question of why EAX works is addressed with conjectures for a possible future research path.
# TABLE OF CONTENTS

**ABSTRACT** ........................................................................................................................................ iii

**LIST OF FIGURES** .......................................................................................................................... v

**ACKNOWLEDGMENTS** ................................................................................................................... vi

**CHAPTER 1 GENETIC ALGORITHMS AND THE TRAVELING SALESMAN PROBLEM** ....................... 1

**CHAPTER 2 EDGE ASSEMBLY CROSSOVER** .................................................................................. 7
  EAX Stage One................................................................................................................................ 7
  EAX Stage Two................................................................................................................................ 7
  EAX Stage Three.............................................................................................................................. 8
  EAX Stage Four................................................................................................................................ 8

**CHAPTER 3 TSP VIEW AND EXTENSIONS** .................................................................................. 15

**CHAPTER 4 CLASS MODULES** ...................................................................................................... 22
  EAXgraph Module............................................................................................................................. 22
  EAXcycle and Cycle Modules.......................................................................................................... 24
  EAXgraph Specification................................................................................................................... 25
  EAXgraph Implementation.............................................................................................................. 27
  Cycle Specification.......................................................................................................................... 43
  Cycle Implementation...................................................................................................................... 44
  EAXcycle Specification.................................................................................................................. 53
  EAXcycle Implementation.............................................................................................................. 55

**CHAPTER 5 DRIVER** .................................................................................................................... 65
  ArrayGenome Specification............................................................................................................ 66
  ArrayGenome Implementation....................................................................................................... 68
  EAX Algorithm Implementation (Cross6)...................................................................................... 69

**CHAPTER 6 METHODOLOGY AND DATA** .................................................................................. 87

**CHAPTER 7 CONCLUSIONS AND RECOMMENDATIONS** .......................................................... 95

**REFERENCE LIST** ....................................................................................................................... 100

**VITA** .............................................................................................................................................. 102
<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>EAX Stage 1, AB cycles</td>
<td>9</td>
</tr>
<tr>
<td>2</td>
<td>EAX Stage 2, E set</td>
<td>10</td>
</tr>
<tr>
<td>3</td>
<td>EAX Stage 3, I Child</td>
<td>11</td>
</tr>
<tr>
<td>4</td>
<td>EAX Stage 4, The Disjoint Set</td>
<td>12</td>
</tr>
<tr>
<td>5</td>
<td>The TSP View Main Window with ATT532 at Generation 1</td>
<td>15</td>
</tr>
<tr>
<td>6</td>
<td>TSP View Controls</td>
<td>16</td>
</tr>
<tr>
<td>7</td>
<td>Choose a GA Window</td>
<td>18</td>
</tr>
<tr>
<td>8</td>
<td>Mutation &amp; Crossover Probabilities Window</td>
<td>19</td>
</tr>
<tr>
<td>9</td>
<td>Abstraction of a Tour within EAXgraph</td>
<td>23</td>
</tr>
<tr>
<td>10</td>
<td>Numerical and Graph Abstraction of the Composite of Two Tours</td>
<td>23</td>
</tr>
<tr>
<td>11</td>
<td>Results of No Sorting of the Disjoint Set, and Not Using the Shortcut Heuristic</td>
<td>88</td>
</tr>
<tr>
<td>12</td>
<td>Results of No Sorting of the Disjoint Set, MaxM=100</td>
<td>88</td>
</tr>
<tr>
<td>13</td>
<td>Results of Sorting the Disjoint Set, Not Using Heuristic Shortcuts in the Exhaustive, Greedy Combining of the Disjoint Set (no Max_M)</td>
<td>89</td>
</tr>
<tr>
<td>14</td>
<td>Results of Sorting the Disjoint Set, MaxM=100</td>
<td>89</td>
</tr>
<tr>
<td>15</td>
<td>Results of Sorting and Using Heuristic Shortcuts in the Exhaustive, Greedy Combining of the Disjoint Set, ch150</td>
<td>90</td>
</tr>
<tr>
<td>16</td>
<td>Results of Sorting and Using Heuristic Shortcuts in the Exhaustive, Greedy Combining of the Disjoint Set, kroA200</td>
<td>90</td>
</tr>
<tr>
<td>17</td>
<td>Results of Sorting and Using Heuristic Shortcuts in the Exhaustive, Greedy Combining of the Disjoint Set, lin318</td>
<td>90</td>
</tr>
<tr>
<td>18</td>
<td>Benchmark ATT532 Test Results with Descending Sort and Shortcut Heuristic where Max_M = 100</td>
<td>91</td>
</tr>
<tr>
<td>19</td>
<td>Benchmark ATT532 Test Results with Descending Sort and No Shortcut Heuristic</td>
<td>91</td>
</tr>
<tr>
<td>20</td>
<td>lin318 Converged</td>
<td>92</td>
</tr>
<tr>
<td>21</td>
<td>kroA200 Converged</td>
<td>92</td>
</tr>
<tr>
<td>22</td>
<td>AT&amp;T532 After 2128 Generations and Max_M=100. Converged at Generation 918,10568 Seconds (2 hrs 56 min. 2GHz, 1 G RAM Pentium)</td>
<td>93</td>
</tr>
<tr>
<td>23</td>
<td>AT&amp;T532 After 2128 Generations and Max_M Not Used. Converged at Generation 958, 15702 Seconds (4 hrs 21 min. 2GHz, 1 G RAM Pentium)</td>
<td>93</td>
</tr>
<tr>
<td>24</td>
<td>Ordered Crossover versus EAX</td>
<td>94</td>
</tr>
<tr>
<td>25</td>
<td>Simulated Annealing</td>
<td>94</td>
</tr>
<tr>
<td>26</td>
<td>The Distinct Graphs of n = 6</td>
<td>97</td>
</tr>
<tr>
<td>27</td>
<td>Demographics for n-gons</td>
<td>98</td>
</tr>
</tbody>
</table>
ACKNOWLEDGMENTS

The arguments creationists put forward for their cause are most often arguments against evolution. Evolution and its proponents are the metaphorical chimpanzee at a typewriter. It is asserted that random chance, even when combined with the forces of nature, nature represented by as intelligent a being as a chimpanzee, could not possibly ever produce a play. The code displayed herein argues otherwise.

Through hearsay I paraphrase, without naming, a member of the department of computer science at UNLV: "A working adult is not a real student of science." Had I not experienced this fact myself, I'd still be in disagreement. I am tempted to say that it is a divine miracle that I am now taking the last steps in a five year effort to quell the curiosities of an artist gone wayward. Instead, I thank my fellow Americans that most don't have the backbone and ignorance I had when entering into such a rigorous curriculum, as otherwise, I'd have been crowded out of this opportunity. I thank the members of the staff of the department of computer science for their positive and respectful attitudes, their knowledge, and their humanistic outlook toward the beautiful students who, with ignorance and hope, place themselves in your hands.

I thank especially Dr. Bein for his gambling that I could use a typewriter for the study of something akin to Shakespeare.
CHAPTER 1

GENETIC ALGORITHMS AND THE
TRAVELING SALESMAN
PROBLEM

Unless you experienced mathematical research in the 1960s when concepts of adaptive methods, including genetic algorithms, were being formulated, you may believe that, just as man has often looked to nature for inspiration in other fields, genetic algorithms are no exception and research in the field begins with dusting off four decade old tombs found on library shelves next to oversized biology books. The analogies of mechanisms of reproduction to combinatorial mathematics has not lost its compelling attraction, as real world applications have been developed to the benefit of especially the engineering disciplines. Yet, I feel that the genetic analogy is weak, and that the field of genetic algorithms will require the strength of refined mathematics to make its methods powerful assets in the area of computation. Though cones are seen in nature, the geometric proofs involving conic sections were not inspired by nature. Genetic algorithms, while inspired by reproduction analogies, will eventually rely on far more synthetic abstractions. This is evidenced in the success of a genetic algorithm crossover called the Edge Assembly Crossover, or EAX, by Nagata and Kobayashi. The EAX is the subject of this paper.

A genetic algorithm is a search algorithm. The objective is to find, among an overwhelming number of individuals, the optimum. The genetic algorithm analogy includes a universal set of individuals that are in the form of a sequence of n symbols.
The sequence, or genome, can be made up of numbers, bits, or letters, or some even more developed type. From the universal set of all possible representations of the sequence, the genetic algorithm chooses, in a blind way, a certain relatively small population of members to begin with. The population may be even in number so that if the population is paired up there is no remaining individual. Or the population may not be fully paired up, but a certain subset of the population may be chosen for pairing, and this choosing can have rigorous requirements. The individuals chosen for pairing, each having a string of n symbols, will endeavor to beget a child by offering copies of parts of their sequences to create a new sequence of symbols that contains all the attributes of the rules that define membership in the universal set. A child will be an element in the universal set. The child, upon its conception, may also undergo mutation, and as such may not be at all like its parents, but must remain an element of the universal set. The population of elements will not grow, though children are reproduced, so a choosing of prized individuals takes place, and the process can now repeat itself as it moves toward the target objective, which is to create from the starting population, an optimal individual.

The basic operators defined in *Genetic Algorithms in Search, Optimization, and Learning* (Goldberg 1989) include reproduction, crossover and mutation.

Reproduction is the choosing of individuals. A parent is chosen by a criteria known as a fitness function or objective function. Such a function also has an inherent implication of the demise of least prized individuals, since only a relatively small population of parents and children exist for computational manipulation out of the myriad of possible beings that make up the universal set. An example of such a set is the set of permutations of a sequence of counting numbers from 1 to 532.

Crossover is the passing to a child some part or parts of the sequence that comprise the parent. Both parents will share in this, but not necessarily equally.
Mutation is a change introduced without the attributes of a parent influencing the change. Mutation may be random, or otherwise a calculated change that is a risk and therefore, as in nature, has a low probability of adding to the prize-worthiness of a resultant member.

Of the three operators within the above definition of a genetic algorithm, crossover offers the conatus of nature. As such, algorithms representing crossovers ought to be as developed as the most sophisticated sorting routines. Yet claims of crossover resultant robustness and efficiency in genetic algorithms abound in mid 1980s reports when the power of modern desktop computers were combined with random methods. Simple bit or digit swapping crossover methods looked very promising on certain problems where the memory enhancements and speed of personal computers dazzled many slide rule despising scholars of the early digital revolution. To see a motion picture like graphic user interface combined with genetic algorithm computation is very appealing. Many problems whose answers at one time required considerable effort are evolved in seconds or minutes before one's very eyes. The promise of finding answers quickly to many practical applications came to fruition. None the less, the ante was upped....

What must be considered is that while applications have been aided by genetic algorithms, many hard problems and open questions are scalable to the methods waged against them. The Traveling Salesman Problem, or TSP, is one such example.

From the book, *The Traveling Salesman Problem* (Lawler et al. editors 1985) comes this description: "If a salesman, starting from his home city, is to visit exactly once each city on a given list and then return home, it is plausible for him to select the order in which he visits the cities so that the total of the distances traveled in his tour is as small as possible. Let us assume he knows, for each pair of cities, the distance from one to the other. Then he has all the data necessary to find the minimum...." But in fact this
problem is in a category of computational difficulty that makes it one of the ongoing
challenges in combinatorial mathematics.

If you imagine yourself as the salesman whose home city makes up one of the n cities
you will visit, there are n-1 possibilities to choose from when embarking from your home
port. The conclusion is that there are (n-1)! routes to take to make a tour. The problem
becomes slightly worse if one makes a sequence out of the one through n counting
numbers that can represent the cities, each city having one of the counting numbers
applied to it as its identifier. If the sequence of counting numbers is generated randomly
and backward routes are not considered the same as forward routes, since a programmed
algorithm will typically differentiate between inverted sequences, then there are n!
sequences to be considered. The set in consideration has a total of n! members. These
members are tours around the cities. A member is represented by an individual sequence
of the counting numbers. So what is the challenge? Just add up each of the tours' total
distances and choose the tour with the smallest sum. First, however, calculate the number
of tours you will be examining. n! is a bit big if your tour is in the United States and you
plan to visit 532 cities on behalf of your employer, AT&T. So next devise a method to
decide just how many steps you may have to take if you are clever about computing tour
distances and finding the optimal one, or at least successively shorter tours. This
challenge comes under the category of computational difficulty called NP Complete.
This term covers not just the Traveling Salesman Problem, but designates a class of
problems that may or may not be tractable. The term tractable refers to problems that can
be solved or proven to be unsolvable within reasonable time constraints. Memory or
space constraints may also to be considered as part of a problem's tractability.

To refer to the difficulty or hardness of a problem conjures up rather subjective
notions seeing as we are not really all created nor educated equally in abilities involving
logic. Thus there is no clear division between a hard and a not so hard problem. Still, we may agree that a tractable problem is one that can be solved in polynomial time. If a problem cannot be considered to have a polynomial time complexity, then it is considered to possibly have an exponential time complexity. Considering exponential time problems, it is not difficult to up the ante, that is, offer an algorithm devised to solve an intractable problem a larger data set. This may be enough to render the algorithm a failure.

Using the notation of *Introduction to Languages and the Theory of Computation* (Martin 1997), the notation expressing the concept of polynomial time complexity is \( P = \bigcup \text{Time}(Cn^k) \), and for intractable problems \( NP = \bigcup \text{NTime}(Cn^k) \). \( N \) refers to the non-deterministic machine or algorithm that would be required to solve an NP problem, if such a machine could exist.

The Traveling Salesman Problem embodies the concept of a possibly intractable challenge. But even more, it exemplifies the entire class of hard problems that, if a polynomial time solution were devised for one, then the entire class is solvable in polynomial time. The term used to refer to this class of problem is NP Complete. That is, if someone were able to demonstrate that an NP Complete problem could be solved by a polynomial time algorithm, then the P or NP question would be resolved, and every NP problem would have similar solution methods applied to them. It would then be justifiable for mathematicians to spend their time solving such known to be polynomial time problems.

That the TSP is widely assumed to be NP Complete does not preclude efforts to write algorithms that endeavor to solve large TSP problems. A number of genetic algorithms with their associated crossovers have been tested against the TSP. Most of these efforts look promising on small data sets, but are thwarted as, again, the ante is upped and
larger data sets are placed against them. Among the genetic algorithms and their important crossovers, The Edge Assembly Crossover stands out.
CHAPTER 2

EDGE ASSEMBLY CROSSOVER

This paper studies a specific, original implementation of the Edge Assembly Crossover (Nagata and Kobayashi 1997) applied to the symmetric traveling salesman problem. There are two different forms of the EAX described by its authors, and while both use random methods, the first is known as the random implementation, or EAX(rand). The second version, which I will refer to as, in honor of the authors, EAX(N&K), utilizes an advantageous heuristic in place of one of the random methods used in the first version. The essentials of the EAX follow.

The Edge Assembly Crossover can be broken down into four stages:

1.) Given a map of n cities, out of a random population of p tours, of which there are a total of n!, two tours, A and B, are joined into one graph from which a set of cycles is created, referred to as AB cycles. The effort always begin with an edge from parent cycle A followed by an edge from parent cycle B and alternate until closing a loop, ending with either an A edge or a B edge. AB cycles must have an even number of edges. If a cycle is closed wherein it has an odd number of edges, a Hamilton cycle like rule is affected and the search continues until a loop closes with an even number in the cycle. A cycle may have more than one loop within it, but any vertex within an AB cycle will have either one incoming and one outgoing edge or two incoming and two outgoing edges. All edges in the parent tours are used up creating a set of AB cycles.

2.) From the Ab cycles set is chosen a subset, referred to as an E set. The E set may
be chosen randomly, the method used within this study, or an advantageous heuristic may be implemented as used by Nagata and Kobayashi. Another team of researchers used a combination of the two methods (Watson et al. 1998).

3.) After the E set is taken from the AB cycles, a copy of one of the two parents is used to build what is referred to as the intermediate child or I child. Every edge in the E set is considered. If a copy of parent A is being used in conjunction with the E set in order to create the I child, then an edge in the E set which originated from parent A is removed from the copy, and an edge in the E set which originated from parent B is added to the copy of A. The copy of parent A is transformed in this manner into an I child. The I child is, in fact, a set of disjoint cycles which have not been identified as such. This description is more easily comprehended using set notation, given cycles X, Y, and Z comprise the E set:

\[ D = A - [(A \cap X) + (A \cap Y) + (A \cap Z)] + [(B \cap X) + (B \cap Y) + (B \cap Z)] \]

4.) The last stage involves the identifying and stitching together of the disjoint set of cycles within the I child in order to form a more perfect union, a child of the two original tours A and B. The newly created child is itself a tour of the n cities.

The above description of the major stages in the Edge Assembly Crossover will hopefully bring to mind a more rigorous and challenging implementation than so many crossover algorithms that have proved to be inadequate in their approach to applying a genetic algorithm attack of the TSP. While a genetic algorithm is composed of the three basic operators, reproduction, mutation, and crossover, the crossover operator's importance was not always so obvious. Simpler crossovers were in the recent past favored in part due to the raw computing power that became available on laptops starting in the late 1980s.

The following pages graphically illustrate the four stages of the EAX.
Parent Tour A: 9 2 3 10 5 1 6 15 4 7 8 14 11 13 12

Parent Tour B: 13 8 9 10 15 6 2 1 14 3 11 12 5 4 7

Parents Composited

AB Cycles:
AB Cycle 0  (6, 15)
AB Cycle 1  (2, 6, 1, 14, 11, 3)
AB Cycle 2  (4, 7)
AB Cycle 3  (9, 2, 1, 5, 4, 15, 10, 3, 14, 8, 13, 12, 5, 10)
AB Cycle 4  (9, 12, 11, 13, 7, 8)

Figure 1: EAX Stage 1, AB cycles

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
AB Cycles to E set through random choosing

- AB Cycle 0 (6, 15)
- AB Cycle 1 (2, 6, 1, 14, 11, 3)
- AB Cycle 2 (4, 7)
- AB Cycle 3 (9, 2, 1, 5, 4, 15, 10, 3, 14, 8, 13, 12, 5, 10)
- Cycle 4 (9, 12, 11, 13, 7, 6)

E Cycle 0

E Cycle 1

The E set = (AB Cycle 1, AB Cycle 3)

- E Cycle 0 (2, 6, 1, 14, 11, 3)
- E Cycle 1 (9, 2, 1, 5, 4, 15, 10, 3, 14, 8, 13, 12, 5, 10)

Figure 2: EAX Stage 2, E set
EAX stage three results in an I Child. An I Child will not, typically, have edges that alternate between A and B. Each city is visited and the number of edges is equal to the number of cities. The I Child contains a set of disjoint cycles. In this example with 15 nodes, there is only one cycle in the set, so the algorithm reaches its objective, a tour.
without having to carry out stage four. During stage three of my implementation of EAX, the I Child is contained within an EAXGraph object, but there is no way of distinguishing edges as members of different cycles. That work is left to stage four.

\[ D = A - [(A \cap X) + (A \cap Y) + (A \cap Z)] + [(B \cap X) + (B \cap Y) + (B \cap Z)] \]

Figure 4: EAX Stage 4, The Disjoint Set

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
The EAX(N&K) includes a heuristic that balances diversity against gain, or exploration against exploitation. The following comes from *Edge Assembly Crossover: A High-power Genetic Algorithm for the Traveling Salesman Problem* (Nagata and Kobayashi 1997).

Notations:

- tour-A: one parent tour with shorter tour length.
- tour-B: the other parent with the longer tour length.
- EA: a set of edges constructing tour-A.
- EB: a set of edges constructing tour-B.
- AB_cyclei: i-th effective AB-cycle (i = 1,2,...,k).
- ni: the number of edges on cyclei.
- e: an edge.
- w(e): a weight of e.
- f(e): the ratio that e exists among the population. At the beginning of each generation, for all edges (nC2 for the STSP or nP2 for ATSP), we calculate the ratio that each edge exists among the population. For example, f(e) = 0.4 means that 40% of the individuals in the population have the edge e.

The following are calculated:

1. gaini = \[ \sum_{e \in AB\text{-cycle}_i \cap E_A} w(e) - \sum_{e \in AB\text{-cycle}_i \cap E_B} w(e) \]

2. \[ f_{A_i} = \sum_{e \in AB\text{-cycle}_i \cap E_A} f(e) \quad f_{B_i} = \sum_{e \in AB\text{-cycle}_i \cap E_B} f(e) \]

3. \[ \bar{f}_A = \frac{\sum_{i} \sum_{e \in AB\text{-cycle}_i \cap E_A} f(e)}{\sum_{i} n_i / 2} \quad \bar{f}_B = \frac{\sum_{i} \sum_{e \in AB\text{-cycle}_i \cap E_B} f(e)}{\sum_{i} n_i / 2} \]
\[
\text{(4)} \quad \text{div}_i = (f_{A_i} - f_{A_i} \cdot n_i / 2) - (f_{B_i} - f_{B_i} \cdot n_i / 2)
\]

\[
\text{(5)} \quad \text{sum\_gain\_plus} = \sum_{i \text{ s.t. } \text{div}_i \geq 0} \text{gain}_i
\]

\[
\text{(6)} \quad \text{sum\_div\_plus} = \sum_{i \text{ s.t. } \text{div}_i \geq 0} \text{div}_i
\]

\[
\text{(7)} \quad \text{GAIN}_i = \frac{\text{gain}_i}{\text{sum\_gain\_plus}}
\]

\[
\text{(8)} \quad \text{DIV}_i = \frac{\text{sum\_div\_plus}}{\text{div}_i}
\]

\[
\text{(9)} \quad F_i = \text{GAIN}_i + \alpha \cdot \text{DIV}_i
\]

The value of \(\text{gain}_i\) by Eq.(1) represents the influence of the tour length contributed by tour-A to an intermediate individual. This comes from the earlier description of constructing the I Child from the E set:

\[
D = A - [(A \cap X) + (A \cap Y) + (A \cap Z)] + [(B \cap X) + (B \cap Y) + (B \cap Z)]
\]

The value of \(\text{div}_i\) by Eq.(4) is one of the measures for representing the contribution of AB_cycle_i to the diversity among the population.

If an AB_cycle with a positive gain is chosen, a \(\text{gain}_i\) in the intermediate child is attained. If an AB_cycle with a positive \(\text{div}_i\) is chosen, the diversity among the population is maintained.

The overall gain or diversity measurements, when stitching the disjoint set cycles together into a valid child tour, is given by \(\text{GAIN}_i\) and \(\text{DIV}_i\). \(F_i\) is utilized by setting up the algorithm to choose only those AB_cycles with positive \(F_i\) values when creating the E set. The trade off between gain and diversity is then adjusted by only the coefficient \(\alpha\).

Nagata and Kobayashi used \(\alpha = 1\) in their experiments.
CHAPTER 3

TSP VIEW AND EXTENSIONS

The implementation of the Edge Assembly Crossover operates within TSP View (Hendricks 1999), a graphic user interface with output showing graphs of the best of each generation as tours evolve. TSP View in turn utilizes MIT's GAlib (Wall 1996), a library of genetic algorithm components. TSP View has been extended to accommodate the choice of the Edge Assembly Crossover as the genetic algorithm crossover to be run.

Figure 5: The TSP View Main Window with ATT532 at Generation 1.
Starting at the upper left of the TSP View controls, a text input gutter or the associated browse button labeled *File* to the right of the gutter is used to designate which file is to be used as the data input file. Below the file input gutter, to the left, are two rectangular outlines surrounding parameter-setting, text input gutters for setting up genetic algorithm runs. Of these, from top to bottom, are *Population Size*, *Generation Size*, *Mutation*.

Figure 6: TSP View Controls.
Probability, and Crossover Probability. There are two square buttons within the genetic algorithms probabilities rectangle labeled $M/C$ and $GA$, which stand for Mutation and Crossover and Genetic Algorithm. I will come back to these buttons.

There are three rectangles on the right below the File button. The top rectangle includes three controls. The top two check boxes control the output within the large graphics window on the right hand side of the GUI. You can set the output to show or hide edges of a graph as well as the numbers that designate the vertices of a graph. The third check box labeled Show Path refers to the window in the lower lefthand corner. Listings of numerical sequences corresponding to graph configurations are seen here during runs.

The second rectangular outline below the input File button includes three check boxes labeled Optimize For Score Only, Optimize Initial Solution, and Optimize Each Step. These parameters are settings for running genetic algorithms other than those involving the Edge Assembly Crossover. The square check box for Optimizing Initial Solution is also used for Simulated Annealing. Un-check all of these when running under genetic algorithm mode with the Edge Assembly Crossover.

There are four rectangular buttons in a row at the bottom of the controls. They are, from left to right, Start, Pause, Reset, and Exit. Rather self explanatory, but it may help to mention that the Exit button works to exit the GUI, while the X button in the upper right corner of the GUI does not.

The check box Apply ATT Metric pertains to running benchmark TSP AT&T532. It allows the measure of distances between cities to be calculated such that round off errors are averaged resulting in the possibility of actually attaining optimum for a run of ATT532. Without this parameter set to on by having the box checked, optimum will not be attained.
Starting with the GA button, clicking upon it opens a window labeled Choose a GA. There are three radio buttons on the left, Simple GA, Steady State GA and Incremental GA. Details of these function are to be found in GAlib: A C++ Library of Genetic Components (Wall 1996). Highlighting Simple GA sets genetic algorithms to run with non-overlapping populations, that is, the entire population is replaced with generated offspring. Highlighting Steady State GA allows the user to set the percentage of the population that is replaced with each generation, should fitter children be produced. The text input gutter to the right of the radio buttons allows you to enter this parameter. The default is 1 percent. The third radio button is Incremental GA, which is a parameter that requires programming knowledge for setting up replacement schemes. One checks this button to turn on a replacement scheme designated in class GAIncrementalGA. This is a good place to write your own replacement scheme.

Finally, on to the button that chooses the crossover scheme when running in genetic algorithm mode. The M/C button, when clicked, opens the Mutations and Crossover Probabilities window. Refer to the illustration on the following page. There are three text input gutters within the M &C window labeled Probability of Rotation, Probability of Inversion and Probability of Swap. The rotation and inversion mutations do not
change the contents of the tour; they change only the way the tour is stored as a sequence. Rotation is a shift of the sequence with wrap around. Inversion is a backwards listing of the sequence. Both of these settings, surprisingly, aid the Edge Assembly Crossover, as tests prove. The default setting is 0.5. The Swap mutation also has a beneficial effect. It swaps two numbers within a sequence if the genetic algorithm is set up for it. Many runs will be required to find the optimum settings for mutation functions.

There are six crossover radio buttons in the M & C window, the bottom two have been added within this project. They correspond to the Simple Correction Crossover and the Edge Assembly Crossover.
The Simple Correction Crossover is a primitive digit swapping type of crossover, with adjustments for maintaining a tour from the bastardized sequence. This is a good place for someone learning to extend TSP View to practice, and create their own mock crossover. The four other crossovers fall into this same category of simplicity, and as such fail when run against even moderately sized TSPs.

The *Edge Assembly Crossover* radio button, when highlighted, not only chooses the EAX algorithm to be run within the genetic algorithm mode of TSP View, it also allows access to further parameters. Not all these parameters are mandatory.

Highlighting the *Edge Assembly Crossover* radio button gives access to two statistical output parameters represented by radio buttons. These are *Save EAX Stats* and *Save Secondary Stats*.

Highlighting *Save EAX Stats* will allow output to the file designated in the text input gutter below the two Stats radio buttons. The *Browse* button allows browsing to the directory of choice. *Save EAX Stats*, in its present form, will output the generation, cost, change in cost, and elapsed time in seconds. This output happens every 10 generations. The generation, cost and time at convergence is also output.

Highlighting the *Save Secondary Stats* radio button turns on more output. In its present form *Save Secondary Stats* outputs debugging information to the same file designated in the text input gutter. The future user of this version of TSP View can easily alter these stats output functions, and should do so with the *Secondary Stats* button.

When the *Save EAX Stats* radio button is highlighted, a text input window with the caption *Known Optimum* becomes available. You must enter the known optimum here, or a low enough value so as not to end the run prematurely. This version of TSP View stops once optimum is attained.

Once the *Edge Assembly Crossover* radio button is highlighted, the *Control Disjoint*
Set Unification thru Maxₘ radio button becomes available. Highlighting the Control
Disjoint Set Unification thru Maxₘ radio button accesses another text input gutter,
which is a parameter controlling the way the disjoint set is stitched together in function
Cycle::Combine_Greedily. This is a deep in the bowels of the algorithm sort of setting.
There is a great efficiency increase gained through the proper setting of this parameter.
However, this setting straddles fence regarding the online approach to developing an
algorithm that can handle any TSP thrown at it. This parameter sets the maximum
distance vertices can be apart to be considered endpoints of viable edges during the
stitching of a cycle to the cycle that is growing into the child. The cycles are elements of
the Disjoint Set. The Cycle::Combine_Greedily(...) function looks for the best edges to
alter in an order n squared algorithm. This function includes code that shortens this
exhaustive approach by looking at every first edge in the growing Child unless a likely
candidate is encountered within the Maxₘ distance. Then ten edges are examined
surrounding the edge with the positive hit. If an entire cycle is circumscribed without a
hit, Maxₘ is increased. Run time is greatly enhanced using this method, but setting this
parameter too small or too large can give worse run times than without it.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
CHAPTER 4

CLASS MODULES

This implementation of the edge assembly crossover was initially developed outside the context of GAlib with no prior knowledge of the code used to previously implement the algorithm. Thus, the classes created to accomplish the task have general enough aspects to offer a programmer at least a skeleton from which to create drivers or useful classes for a host of graph manipulation problems.

The EAXgraph class encapsulates the concept of a single TSP tour as an adjacency list represented graph as well as the juxtaposition of two tours. An instance of EAXgraph is a two dimensional array. Using a city analogy for the vertexes of a graph, the first index of the array represents one of the cities, which I will refer to as the source or subject city. A source city is the starting point of an edge. The second index represents one of two possible parent sources, A or B. Thus the 2-D array is as long as the number of cities and two deep. The array elements of an EAXgraph instance is a struct containing five data members, including the source city's coordinates, doubles x and y, two vertexes, integers vt1 and vt2 which function as the endpoints of the two edges leading from the source city, and information concerning these two vertexes as to whether they exist and have been seen within the algorithm. These include boolean values vt1_seen and vt1_exists, as well as vt2_seen and vt2_exists. While there are n*(n-1)/2 possible edges in a completely connected graph, EAXgraph encapsulates only n edges for a tour, or 2n edges for a juxtaposition of two tours. Graph and numerical output of edges of a tour stored within EAXgraph might look like the following:
Figure 9: Abstraction of a Tour within EAXgraph.

Figure 10: Numerical and Graph Abstraction of the Composite of Two Tours.
Two more data type modules used within my implementation of EAX include EAXcycle and Cycle. EAXcycle is a simplification of EAXgraph, and though it has advantages, need not be used to implement the EAX. The Cycle class encapsulates the concept of a tour as a sequence of integers, namely the counting numbers. My implementation of EAX has the sequence begin with 1. The GALib implementation of genetic algorithm utility classes starts with 0. Those desiring to compete for fastest GAlib/TSP algorithm in the world using EAXgraph and Cycle modules will want to begin the coding improvements here.

The modules I made have member functions that go way beyond a general graph object abstraction. The most noteworthy is class Cycle's function Combine_Greedily(Cycle*, int). This is an order n squared Achilles heel within EAX. It is part of stage four and is the code that stitches a disjoint set of cycles back into a tour. More on this in the following chapter. Much of graph theory may look like soda straws and styrofoam balls, but beneath lies a labyrinth Borges would appreciate. Functions dealing with graphs can be expected to be difficult to work out. I bring this up to underscore that I mixed rather involved, EAX specific functions within the members of the graph type modules. They would, outside this context, not be there, but consider part of my task of implementing the EAX algorithm was to eventually embed it into an existing graphic user interface, TSP View. The algorithm's driver is a function within a GAlib file, and adding yet more functions to that file, and I added a good number, seems less appropriate than overstepping the base functions expected within a programmer created graph data type. Also worthy of mention for speed daemons of TSP coding competitions is the consideration of when to make edge weight calculations. I placed a member function within the Cycle class for this task, but edge weight calculations, simply point to point distances, are done outside my EAX algorithm and passed in.
The following pages list the specification and implementation files of EAXgraph, Cycle, and EAXcycle data types. The sane man will skip on to chapter six.

```c
#ifdef EAX_GRAPH_H
#define EAX_GRAPH_H

/* EAX "Edge Assembly Crossover". This file SPECIFIES the Abstract Data Type 'EAXgraph'.

An EAXgraph is represented through a 2-dimensional array, g. The first index of the array represents a city on a map. In this algorithm two graphs are juxtaposed into a single graph called an Rgraph. It is necessary to keep track of which originating graph (parent) an edge has come from, A or B, and thus the second index of the array, g, represents graph A or graph B.

For a given vertex in a tour, there are two possible edges that can be taken to an adjacent vertex, if the cycle is considered to be an undirected graph. The array contains the two possible destination vertexes, vt1 and vt2, which are members of a struct. The struct contains other bookkeeping members, as shown below.

*/

#include "EAX_cycle.h"
#include "GlobalMax.h"

extern const int MAX_SIZE; //see GlobalMax.h to change this constant

struct graph_struct //data type used within class EAXgraph
{
    int vt1; //Directed outgoing edge endpoint.
    bool vt1_seen; //true implies edge vs-->)vt1 has been traversed
    bool vt1_exists;
    int vt2; //Contra-Directed edge's endpoint, i.e., the reflection of the incoming edge.
    bool vt2_seen; bool vt2_exists;
    double x; //Coordinates of this vertex
    double y;
};

// Edges are looked at from the context of the source vertex, vs,
// and thus vt1 and vt2 are established from the context of vs.
// enum AorB_enum {A,B}; // data type used within class EAXgraph,
// declared in AorB_enum.h
// Used as the 2nd index of a graph's
// 2-D array, designating from which
// parent the graph component originated.
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
class EAXgraph
{
public:
    int length; // The Maximum number of nodes and edges in a graph. // n = e in a tour.
    // This value must be assigned in the constructor via the class declaration
    // parameter example: EAXgraph Rgraph(15);
    int remaining; // number of edges remaining in a graph.
    graph struct g[MAX_SIZE][2];
    // NO LONGER USING DYNAMIC ALLOCATION, see version 6 for dynamic code
    // graph_struct (*g)[2];

public:
    EAXgraph(); // constructor
    EAXgraph(int); // constructor
    EAXgraph(EAXgraph&); // copy constructor
    operator=(EAXgraph&); // Assignment operator
    // ~EAXgraph(); // destructor
    /* Parent Graph */

    // Modifiers
    void Load_Tour_from_Array(const int[], const int, AorB_enum);
    /* Rgraph */
    // Modifiers */
    void Build_Rgraph_from_two_Graphs(EAXgraph&, EAXgraph&);
    void Remove_Cycle(EAXcycle&, int, AorB_enum);
    void Remove_Edge(int, int, AorB_enum);
    void Add_Edge(int, int, AorB_enum);
    void Add_Edge_Without_Refections(int, int, AorB_enum);
    void Mark_Edge_As_Seen(int, int, AorB_enum);
    void Mark_Edge_As_NOT_Seen(int, int, AorB_enum);
    void Clean_Reflections();
    void Assign_Coordinates(int, AorB_enum, double, double);
    AorB_enum Swap_AorB(AorB_enum);

    // Accessors
    int Get_Length();
    int Get_Remaining();
    int Get_vt_Randomly(int[], int[], int&);
    int Get_vt_Randomly(int, AorB_enum);
    int Get_vt1(int, AorB_enum);
    int Get_vt2(int, AorB_enum);
    double Get_x(int, AorB_enum);
    double Get_y(int, AorB_enum);
    bool Get_vt1_seen(int, AorB_enum);
    bool Get_vt1_exists(int, AorB_enum);
    bool Get_vt2_seen(int, AorB_enum);
    bool Get_vt2_exists(int, AorB_enum);
    bool Graph_Is_Empty();
    bool Edge_Can_Be_Seen(int, int, AorB_enum);
bool Edge_Exists(int, int, AorB_enum);
void Write(ofstream&);

#endif

//file EAXgraph.cpp

#include "stdafx.h"
#include "EAX_graph.h"
#include <iostream.h>
#include <stdlib.h>
#include <time.h>
#include "TSP View2.h" // for Write() fn

EAXgraph::EAXgraph() //default constructor used only for debugging purposes.
{
    length = 0;
    remaining = 0;
    //NO LONGER USIG DYNAMIC ARRAY
    // g = new graph_struct[length+1][2];// g[0][x] is not used
    g[0][A].vt1 = -1;  // -1 designates un-assigned  g[0][A].vt1_seen = false;
    g[0][A].vt1_exists = false;
    g[0][A].vt2 = -1;  // g[0][A].vt2_seen = false;
    g[0][A].vt2_exists = false;
    g[0][A].x = -1.0;
    g[0][A].y = -1.0;
    //---
    g[0][B].vt1 = -1;
    g[0][B].vt1_seen = false;
    g[0][B].vt1_exists = false;
    g[0][B].vt2 = -1;
    g[0][B].vt2_seen = false;
    g[0][B].vt2_exists = false;
    g[0][B].x = -1.0;
    g[0][B].y = -1.0;
}

EAXgraph::EAXgraph(int len)
//constructor
//POST: An AB graph with a potential of 'len' vertices is allocated // memory and is initialized to default values. No edges yet exist.
{
    int i;
}
length = len;
remaining = 0;
// NO LONGER USIG DYNAMIC ARRAY
// g = new graph_struct[length+1][2]; // g[0][x] is not used
for(i=0; i<= length; i++) {
    g[i][A].vt1 = -1; // -1 designates un-assigned  g[i][A].vt1_seen = false;
    g[i][A].vt1_exists = false;
    g[i][A].vt2 = -1;
    g[i][A].vt2_seen = false;
    g[i][A].vt2_exists = false;
    g[i][A].x = -1.0;
    g[i][A].y = -1.0;
    //---
    g[i][B].vt1 = -1;  g[i][B].vt1_seen = false;
    g[i][B].vt1_exists = false;
    g[i][B].vt2 = -1;
    g[i][B].vt2_seen = false;
    g[i][B].vt2_exists = false;
    g[i][B].x = -1.0;
    g[i][B].y = -1.0;
}

//********1**********2**********3**********4**********5**********6**********8
//
EAXgraph::EAXgraph(EAXgraph& otherEAXgraph)
// copy constructor:
int i;
length = otherEAXgraph.length;
remaining = otherEAXgraph.remaining;
for(i=0; i <= length; i++) {
    g[i][A] = otherEAXgraph.g[i][A];
    //---
    g[i][B] = otherEAXgraph.g[i][B];
}

//********1**********2**********3**********4**********5**********6**********8
//
EAXgraph::operator=(EAXgraph& otherEAXgraph)
// Assignment operator
{
    int i;
    length = otherEAXgraph.length;
remaining = otherEAXgraph.remaining;

for(i=0; i <= length; i++)
{
    g[i][A] = otherEAXgraph.g[i][A];

    //---
    g[i][B] = otherEAXgraph.g[i][B];
}

EAXgraph::~EAXgraph()
//destructor
{
    // no longer using dynamic memory
    //delete [] g;
}

void EAXgraph::Load_Tour_from_Array(const int array[],
    const int SIZE,
    AorB_enum AorB)
{
    int i;
    remaining = SIZE;
    // load directed graph
    for(i=0; i<SIZE; i++)
    {
        g[array[i]][AorB].vt1 = array[(i+1) % SIZE];
        //NOTE: g[0] is never assigned
        g[array[i]][AorB].vt1_exists = true;
    }
    // load reflections
    for(i=1; i<=SIZE; i++)
    {
        g[ g[i][AorB].vt1 ][AorB].vt2 = i;
        g[ g[i][AorB].vt1 ][AorB].vt2_exists = true;
    }
}
```c
void EAXgraph::Build_Rgraph_from_two_Graphs(EAXgraph& A_tour,
                                          EAXgraph& B_tour)

// ASSERTION: 1 <= i <= length
// POST: Rgraph exists with 2*length edges. { int i;
    for(i=1; i <= A_tour.length; i++)//Note that g[0][AorB] not used
    { g[i][A].vt1 = A_tour.g[i][A].vt1;
        g[i][A].vt1_seen = false;
        g[i][A].vt1_exists = A_tour.g[i][A].vt1_exists;
        g[i][A].vt2 = A_tour.g[i][A].vt2;
        g[i][A].vt2_seen = false;
        g[i][A].vt2_exists = A_tour.g[i][A].vt2_exists;
        //--
        g[i][B].vt1 = B_tour.g[i][B].vt1;
        g[i][B].vt1_seen = false;
        g[i][B].vt1_exists = B_tour.g[i][B].vt1_exists;
        g[i][B].vt2 = B_tour.g[i][B].vt2;
        g[i][B].vt2_seen = false;
        g[i][B].vt2_exists = B_tour.g[i][B].vt2_exists;
    }
    remaining = length*2;// --- Two tours placed into one graph
}
```

```c
void EAXgraph::Remove_Cycle(EAXcycle& cycle,
                       int sv,
                       AorB enum svAorB)

// PRE: An Rgraph calls this function. A cycle exists within the Rgraph, which has been
// saved within an EAXcycle. The cycle must not have a stem. Use
// EAXcycle::Remove_Stem(...) function.
// NOTE: The indexes of the cycle graph coincide with the Rgraph.
// /
// sv = starting vertex
// svAorB = starting vertex is from graph A or B
{ int i, source, target;
    AorB_enum AorB;
    int edges_in_cycle;
    source = sv;
    target = cycle.Get_vt(sv, svAorB);
    AorB = svAorB;
    AorB = svAorB;
    edges_in_cycle = cycle.Get_Remaining();
```
for(i=1; i<=edges_in_cycle; i++)
{
    Remove_Edge(source, target, AorB);
    source = target;
    AorB = Swap_AorB(AorB);
    target = cycle.Get_vt(source, AorB);
}

//*******1**********2**********3**********4**********5**********6**********8

// void EAXgraph::Remove_Edge(int vs, int vt, AorB_enum AorB)
/*PRE: edge exists,...keep in mind that an edge is represented
by a source vertex and a destination vertex, or a target vertex. In this class's
implementation the source vertex is the first index of a 2-D array. The target vertex
is the endpoint of two possible edges, represented by struct graph struct's members vt1
or vt2. The point is: this function passes the endpoints of an edge, but cannot
designate which of the two variables has the correct endpoint.
This is because a cycle, though a directed graph, is built
randomly from an Rgraph. Thus the if - else if blocks below.

The meaning of "reflection". Each edge in a tour is represented
twice. Once as an outgoing edge, and once as an incoming edge.
From a given vertex, the outgoing edge is considered first, or
has a greater status, if you will, because tours, being cycles,
have direction. A source vertex and its outbound vertex, vt1,
together represent a directed edge, vs --> vt1. The "reflection"
of that same edge is the oppositely directed edge,
vt1 --> vs.
*/

//POST: remove edge vs-->vt and vt-->vs to completely remove edge.
{
    //cerr<<"nRemove_Edge function\n";
    if(vt == g[vs][AorB].vt1) //looking for correct edge
    {
        remaining--;
        g[vs][AorB].vt1_exists = false;
        g[vs][AorB].vt1 = -1;
        if(vs == g[vt][AorB].vt1) //looking for correct reflection:
        {
            g[vt][AorB].vt1_exists = false;
            g[vt][AorB].vt1 = -1;
        }
        else if(vs == g[vt][AorB].vt2) {
            g[vt][AorB].vt2_exists = false;
            g[vt][AorB].vt2 = -1;
        }
    }
else {
    cerr<<"Error in file EAXgraph.cpp,";
    cerr<<"fn EAXgraph::Remove_Edge(...),1st else\n";
}
}
else if(vt == g[vs][AorB].vt2) //looking for correct edge
{
    remained--;;
    g[vs][AorB].vt2_exists = false;
    g[vs][AorB].vt2 = -1;
    if(vs == g[vt][AorB].vt1) //looking for correct reflection:
    {
        g[vt][AorB].vt1_exists = false;
        g[vt][AorB].vt1 = -1;
    }
    else if(vs == g[vt][AorB].vt2)
    {
        g[vt][AorB].vt2_exists = false;
        g[vt][AorB].vt2 = -1;
    }
    else
    {
        cerr<<"Error in file EAXgraph.cpp,";
        cerr<<"fn EAXgraph::Remove_Edge(...),2nd else\n";
    }
}
else {
    cerr<<"Error in file EAXgraph.cpp,";
    cerr<<"fn EAXgraph::Remove_Edge(...),3rd else\n";
}
}

//**************************************************************************
// void EAXgraph::Add_Edge(int vs, int vt, AorB_enum AorB)
{
    remaining++;
    if(!g[vs][AorB].vt1_exists)
    {
        g[vs][AorB].vt1 = vt;
        g[vs][AorB].vt1_exists = true;
        // reflection:
        if(!g[vt][AorB].vt2_exists)
        {
            g[vt][AorB].vt2 = vs;
            g[vt][AorB].vt2_exists = true;
        }
        else if(!g[vt][AorB].vt1_exists)
        {
            g[vt][AorB].vt1 = vs;
        }
    }
}

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
g[vt][AorB].vt1_exists = true;
} else cerr<<"\n Error 1 assigning edges within";
cerr<<" EAXgraph::Add_Edge(...)\n";
} else if(!g[vs][AorB].vt2_exists)
{
  g[vs][AorB].vt2 = vt;
  g[vs][AorB].vt2_exists = true;
  // reflection:
  if(!g[vt][AorB].vt1_exists)
  {
    g[vt][AorB].vt1 = vs;
    g[vt][AorB].vt1_exists = true;
  }
  else if(!g[vt][AorB].vt2_exists)
  {
    g[vt][AorB].vt2 = vs;
    g[vt][AorB].vt2_exists = true;
  } else {
    cerr<<"\n Error 2 assigning edges within";
    cerr<<" EAXgraph::Add_Edge(...)\n";
  }
} else {
  cerr<<"\n Error 3 assigning edges within";
  cerr<<" EAXgraph::Add_Edge(...)\n";
}

//************1************2************3************4************5************6************8
//
void EAXgraph::Add_Edge_Without_Reflections(int vs,
    int vt,
    AorB_enum AorB)
{
    remaining++;
    if(!g[vs][AorB].vt1_exists)
    {
      g[vs][AorB].vt1 = vt;
      g[vs][AorB].vt1_exists = true;
    } else if(!g[vs][AorB].vt2_exists)
    {
      g[vs][AorB].vt2 = vt;
      g[vs][AorB].vt2_exists = true;
    } else {
      cerr<<"\n Error 3 assigning edges within";

cerr<<" EAXgraph::Add_Edge(...)\n";
}

void EAXgraph::Make_Stan_Not_Seen(EAXcycle& cycle, int ss,
        AorB_enum ssAorB, int cs )
{
    // cs == cycle source;
    // ss == stem source
    // ssAorB == stem source's parent
    int vs;
    int vt;
    AorB_enum AorB;
    vs = ss;
    AorB = ssAorB;
    vt = cycle.Get_vt(ss,AorB);
    while(vs != cs)
    {
        Mark_Edge_As_NOT_Seen(vs,vt,AorB);
        vs = vt;
        AorB = cycle.Swap_AorB(AorB);
        vt = cycle.Get_vt(vs,AorB);
    }
}

void EAXgraph::Mark_Edge_As_Seen(int vs, int vt, AorB_enum AorB)
{
    if(g[vs][AorB].vt1 == vt)
    {
        g[vs][AorB].vt1_seen = true;
        if(g[vt][AorB].vt1 == vs) //mark the reflection edge
        {
            g[vt][AorB].vt1_seen = true;
        }
    }
    else if(g[vt][AorB].vt2 == vs)
    {
        g[vt][AorB].vt2_seen = true;
    }
    else
    {
        cerr<<"Error1 in EAXgraph::Mark_Edge_As_Seen():";
        cerr<<" AorB,vs, vt = ";
    }
if(AorB == 0)
{
    cerr<<"A,";
}
else
{
    cerr<<"B,";
}
cerr<<vs<<"",<<vt<<endl;
}
else if(g[vs][AorB].vt2 == vt)
{
    g[vs][AorB].vt2_seen = true;
    if(g[vt][AorB].vt1 == vs)
    {
        g[vt][AorB].vt1_seen = true;
    }
    else if(g[vt][AorB].vt2 == vs)
    {
        g[vt][AorB].vt2_seen = true;
    }
    else
    {
        cerr<<"Error2 in EAXgraph::Mark_Edge_As_Seen():";
        cerr<<" AorB,vs,vt = ";
        if(AorB == 0)
        {
            cerr<<"A,"
        }
        else
        {
            cerr<<"B,";
        }
        cerr<<vs<<"",<<vt<<endl;
    }
}
else
{
    cerr<<"Error3 in EAXgraph::Mark_Edge_As_Seen():";
    cerr<<" AorB,vs,vt = ";
    if(AorB == 0)
    {
        cerr<<"A,"
    }
    else
    {
        cerr<<"B,";
    }
}
void EAXgraph::Mark_Edge_As_NOT_Seen(int vs, int vt, AorB_enum AorB)
{
    if(g[vs][AorB].vt1 == vt)
    {
        g[vs][AorB].vt1_seen = false;
        if(g[vt][AorB].vt1 == vs) //mark the reflection edge
        {
            g[vt][AorB].vt1_seen = false;
        }
        else if(g[vt][AorB].vt2 == vs)
        {
            g[vt][AorB].vt2_seen = false;
        }
        else
        {
            cerr << "Error1 in EAXgraph::Mark_Edge_As_NOT_Seen(): ";
            cerr << " AorB, vs, vt = ";
            if(AorB == 0)
            {
                cerr << "A, ";
            }
            else
            {
                cerr << "B, ";
            }
            cerr << vs << " , " << vt << endl;
        }
    }
    else if(g[vs][AorB].vt2 == vt)
    {
        g[vs][AorB].vt2_seen = false;
        if(g[vt][AorB].vt1 == vs)
        {
            g[vt][AorB].vt1_seen = false;
        }
        else if(g[vt][AorB].vt2 == vs)
        {
            g[vt][AorB].vt2_seen = false;
        }
        else
        {
            cerr << "Error2 in EAXgraph::Mark_Edge_As_NOT_Seen(): ";
            cerr << " AorB, vs, vt = ";
        }
    }
}

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
if(AorB == 0)
{
  cerr<<"A,\n";
}
else
{
  cerr<<"B,\n";
}
cerr<<vs<<"",<<vt<<endl;
}
}
else
{
  cerr<<"Error3 in EAXgraph::Mark_Edge_As_NOT_Seen():\n";
  cerr<<" AorB,vs,vt = \n";
  if(AorB == 0)
  {
    cerr<<"A,\n";
  }
  else
  {
    cerr<<"B,\n";
  }
  cerr<<vs<<"",<<vt<<endl;
}

//**********1**********2**********3**********4**********5**********6**********68
//
void EAXgraph::Clean_Reflections()
{
  int i;
  for(i=1; i<=length; i++)
  {
    g[i][A].vt2 = -1;
    g[i][A].vt2_exists = false; g[i][A].vt2_seen = false;
    g[i][B].vt2_exists = false; g[i][B].vt2_seen = false;
  }
}

//**********1**********2**********3**********4**********5**********6**********68
//
void EAXgraph::Assign_Coordinates(int vs, AorB_enum AorB, double x, double y)
{
  g[vs][A].x = x;
  g[vs][A].y = y;
}
```cpp
// AorB enum EAXgraph::Swap_AorB(AorB_enum AorB)
{
  if(AorB == A)
  {
    return B;
  }
  else
  {
    return A;
  }
}

// int EAXgraph::Get_Length()
// POST: private data member 'length' returned
{
  return length;
}

// int EAXgraph::Get_Remaining()
// POST: private data member 'remaining' returned
{
  return remaining;
}

// int EAXgraph::Get_vs_Randomly(int A_n[],int A_p[],int& upper_limitA)
// A_n = Array of available numbers //A_p = Array of pointers to Ae numbers
// This function is called in reference to source parent A alone.
{
  int vs, random_choice;
  srand( (unsigned)time(NULL) ); // Beats me,...srand works here,
  // and is necessary for non-
  // repeating runs.
  random_choice = rand()%upper_limitA + 1;
  vs = A_n[random_choice];
  return vs;
}

// int EAXgraph::Get_vt_Randomly(int vs, AorB_enum AorB)
// Coin toss, but other side is offered if first choice unavailable
{
  int number, randreturn;
  //srand( (unsigned)time(NULL) ); // Turn this on and algorithm becomes NOT random.
  randreturn = rand();
}
```
/* Bloody 'ell. Take a look at these values in the debugger, or output them to screen.
The return value shown for function rand() is different from the value assigned to
randreturn. !!!!*/
number = randreturn%2;
if(number == 0)
{
  if( g[vs][AorB].vt1 != -1 && !Edge_Can_Be_Seen(vs, g[vs][AorB].vt1, AorB) )
  {
    return g[vs][AorB].vt1;
  }
  else if( g[vs][AorB].vt2 != -1 && !Edge_Can_Be_Seen(vs, g[vs][AorB].vt2, AorB) )
  {
    return g[vs][AorB].vt2;
  }
  else
  {
    return -111; // This is an error designation.
  }
}
else
{
  if( g[vs][AorB].vt2 != -1 && !Edge_Can_Be_Seen(vs, g[vs][AorB].vt2, AorB) )
  {
    return g[vs][AorB].vt2;
  }
  else if( g[vs][AorB].vt1 != -1 && !Edge_Can_Be_Seen(vs, g[vs][AorB].vt1, AorB) )
  {
    return g[vs][AorB].vt1;
  }
  else
  {
    return -111; // This is an error designation for debugging.
  }
}

//*********1*********2**********3**********4**********5**********6**********68
//
int EAXgraph::Get_vt1(int vs, AorB_enum AorB)
{
  return g[vs][AorB].vt1;
}

//*********1*********2**********3**********4**********5**********6**********68
//
int EAXgraph::Get_vt2(int vs, AorB_enum AorB)
{
  return g[vs][AorB].vt2;
}
double EAXgraph::Get_x(int vs, AorB_enum AorB)
{
    return g[vs][AorB].x;
}

double EAXgraph::Get_y(int vs, AorB_enum AorB)
{
    return g[vs][AorB].y;
}

bool EAXgraph::Get_vtl_seen(int vs, AorB_enum AorB)
{
    return g[vs][AorB].vtl_seen;
}

bool EAXgraph::Get_vtl_exists(int vs, AorB_enum AorB)
{
    return g[vs][AorB].vtl_exists;
}

bool EAXgraph::Get_vtl2_seen(int vs, AorB_enum AorB)
{
    return g[vs][AorB].vt2_seen;
}

bool EAXgraph::Get_vt2_exists(int vs, AorB_enum AorB)
{
    return g[vs][AorB].vt2_exists;
}
```cpp
// bool EAXgraph::Get_vtl_vt2_both_seen(int vs, AorB_enum AorB)
//
// return g[vs][AorB].vtl_vt2 both seen;
// */

// bool EAXgraph::Graph_Is_Empty()
//
// if(remaining == 0){ return true; } else return false;
//

// bool EAXgraph::Edge_Can_Be_Seen(int vs, int vt, AorB_enum AorB)
// If an edge exists, it may not have been traversed and in such a case is not registered as seen.
//
// if(g[vs][AorB].vtl == vt)
// { if(g[vs][AorB].vtl_seen == true){return true;} else{return false;}}
// else if(g[vs][AorB].vt2 == vt)
// { if(g[vs][AorB].vt2_seen == true){return true;} else{return false;}}
// else return false;
//

// bool EAXgraph::Edge_Exists(int vs, int vt, AorB_enum AorB)
// Though the default values of vtl and vt2 are -1, do not assume that a non-existing edge must have target vertices values of -1. This is not true.
// While only (length) edges are represented in a graph loaded from an array of ints, (2*length) edges are represented in an Rgraph. Up to (2*length) edges are represented 'possible cycle' or cycle.
//
// if(g[vs][AorB].vtl == vt)
// { if(g[vs][AorB].vt1_exists == true)
// { return true;}}
```
else
{
    return false;
}
}
else if(g[vs][AorB].vt2 == vt)
{
    if(g[vs][AorB].vt2_exists == true)
    {
        return true;
    } else
    {
        return false;
    }
}
else return false;

void EAXgraph::Write(ofstream& outfile)
{
    // The formatting in this function is effective only for graphs
    // less than 100 vertices in length. Useful for debugging.
    int i;
    outfile<<"n***Enter fn:Write()********** begin R_graph data";
    outfile<<"nRgraph.remaining= " <<remaining<<endl;
    for(i=1; i<=length; i++)
    {
        if(i<10)outfile<< " 
        outfile<<"A vt1 = ",
        if(g[i][A].vtl>0 && g[i][A].vtl<10)
        {
            outfile<< " 
        if(g[i][A].vtl == -1)
        {
            outfile<< " 
        else outfile<<g[i][A].vtl;
        outfile<<" vt2 = 
        if(g[i][A].vt2>0 && g[i][A].vt2<10)
        {
            outfile<< " 
        if(g[i][A].vt2 == -1)
        {
            outfile<< " 
        else outfile<<g[i][A].vt2;

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
outfile<<" ",
    //-------
if(i<10) outfile<<" ",
outfile<<"B vt1 = ";
if(g[i][B].vt1>0 && g[i][B].vt1<10)
{
    outfile<<" ",
}
else outfile<<g[i][B].vt1;
outfile<<" vt2 = ";
if(g[i][B].vt2>0 && g[i][B].vt2<10)
{
    outfile<<" ",
}
else outfile<<g[i][B].vt2;
outfile<<endl;
}
if(remaining == 0)
{
    outfile<<" 	R_graph is empty\n";
}
outfile<<"***Exit fn:Write()*********** end R_graph data\n";
}
*************** END EAXgraph.cppb ***************

*************** Begin Cycle Specification ***************
#ifndef CYCLE_H
#define CYCLE_H
/* SPECIFICATION file of module "cycle" that contains a series of integers representing a cycle. A class having nothing but public data members is used for two purposes: 1.) the ease of dynamic instantiation, since this will ease memory requirements, and 2.) the primitive array-like ease of assigning and recalling integers The class contains two private data members, both are public: 1.) an array of structs 2.) the length of the array The struct contains two members: 1.) the integer 2.) the designation of which of two parent objects the integer originated from.
#include "EAX_graph.h"
#include "Cycle_AorB_enum.h"

struct vertex_struct
{
    int v; //represents a vertex (in TSP a city's number)
    Cycle_AorB_enum AorB; //parent object designation
    double x; //x coordinate of vertex
    double y; //y coordinate of vertex
};

class Cycle
{
public:
    vertex_struct *vertex; // pointer to an array of type vertex_struct,
    // the size of which is established by the
    // constructor.
    int length;

public:
    Cycle(int); //constructor
    Cycle(Cycle&); //copy constructor
    operator=(Cycle& otherCycle);
    ~Cycle(); //destructor
    Cycle Combine_Greedily(Cycle*, int);
    double Cost_Edge(double, double, double, double);
    void Clean(); //re-initializes the cycle's data
    void Translate_from_EAXgraph_to_Cycle(EAXgraph&, int);
    void Write(int, ofstream&);
};

#endif

// ************ Cycle member functions ************
#include "stdafx.h"
#include "TSP_View2.h"
#include "MutationDlg.h"
#include "cycle.h"
#include "EAX_graph.h"
#include <iostream.h>
#include <math.h>
#include <iomanip.h>
#include <fstream.h>
Cycle::Cycle(int len)
//constructor
// POST: An AB graph with a potential of 'len' vertices is allocated memory and is initialized to default values. No edges yet exist.
{
    int i;
    length = len;
    vertex = new vertex_struct[length+1];//v_array[0] is not used
    for(i=0;i<=length;i++)
    {
        vertex[i].v = -1; // -1 designates un-assigned vertex[i].AorB = cZ; // cZ designates un-assigned vertex[i].x = -99999.9;
        vertex[i].y = -99999.9;
    }
}

Cycle::Cycle(Cycle& otherCycle)
//copy constructor
{
    int i;
    length = otherCycle.length;
    vertex = new vertex_struct[length+1];
    for(i=0; i <= length; i++)
    {
        vertex[i] = otherCycle.vertex[i];
    }
}

Cycle::operator=(Cycle& otherCycle)
//Assignment operator
//PRE: Two cycles having same length exist
{
    int i;
    length = otherCycle.length;
    for(i=0; i <= otherCycle.length; i++)
    {
        vertex[i] = otherCycle.vertex[i];
    }
}

Cycle::~Cycle()
//destructor
{
delete[] vertex;

//********|********|********|********|********|********|********|
//
bool CMutationDlg::m_do_shortgreedy;
double CMutationDlg::m_max_measure;
bool CMutationDlg::m_do_statsoutput_output_set_2;
CArray<int,int> CTSPViewDlg::distance;
int CTSPViewDlg::m_cities;

Cycle Cycle::Combine_Greedily(Cycle* cycle_j, int I_length )
// A greedy algorithm that combines two cycles. One of the two
// cycles is the Child. The Child calls this function and passes within
// parameter number one the other child. Consider the calling cycle
// the i cycle. "this" pointer corresponds to i.
// Consider the other cycle the j cycle, parameter cycle_j.

// beating a possibly dead horse, more on terminology:
// "this" refers to the Cycle that calls this function
// "other" refers to the Cycle passed into the function via the first parameter.
{
  int i, next_i; // indexes of "this" Cycle
  int im,next_im; // m for "master", the full cycle of "this" cycle
  int j, next_j; // indexes of "other" Cycle
  
  const int lenj = cycle_j->length;// i.e. other_cycle->length
  const int lenim = this->length;
  const int leni = lenim;

  int mc = CTSPViewDlg::m_cities;//length of full tour
  
  int endi;//the index of the endpoint of a segment within a cycle

  // These values are graph dependent, graph specific and will
  // exhibit varied efficiency as m_max_measure is changed.
  // This change is made in the GUI in the control labeled:
  // "Control Disjoint Set Unification thru Max_M"
  
  double max_measure = CMutationDlg::m_max_measure;
  const double increment = .05* max_measure;
  double Nth_time = 0.0;
  bool found_edges_flag = false;

  EAXgraph temp_EAXgraph(I_length);
  Cycle temp_cycle(lenim+lenj);
  
  //source and target vertex of edge to be removed from "this" cycle
  int is, it;
//source and target vertex of edge to be removed from "other" cycle
int js, jt;

//source and target vertex of 1st edge to be added to
//the combined cycles
int vs_1, vt_1;

//source and target vertex of 2nd edge to be added to
//the combined cycles
int vs_2, vt_2;

double cost_of_2_edges_to_remove;
double combined_cost1;
double combined_cost2;
double lowest_cost = 9999999.9; //arbitrarily large lowest cost

if(leni < 11 || !CMutationDlg::m_do_shortgreedy)//true -> not doing shortcut
{
    for(j=1; j<=lenj; j++)
    {
        next_j = j+1;
        if( next_j == lenj+1){next_j=1;
        }
        for(im=1; im<=lenim; im++)
        {
            next_im = im+1;
            if( next_im == lenim+1){next_im=1;
            }

            //See Brad Hendrick's code:CTSPViewDlg::read_tsp_file() and set_dist_array()
            //to understand cost calculations below.

            cost_of_2_edges_to_remove =
            CTSPViewDlg::distance[vertex[im].v -1 + mc*(vertex[next_im].v - 1)]
            + CTSPViewDlg::distance[cycleJ->vertex|j].v -1 + mc*(cyclej->vertex[nextj].v - 1)];
            combined_cost1 =
            CTSPViewDlg::distance[vertex[im].v -1 + mc*(cycle_j->vertex[next_j].v - 1)]
            + CTSPViewDlg::distance[vertex[next_im].v -1 + mc*(cycleJ->vertex[nextj].v - 1)]
            - cost_of_2_edges_to_remove;
            combined_cost2 =
            CTSPViewDlg::distance[vertex[im].v -1 + mc*(cycle_j->vertex[j].v - 1)]
            + CTSPViewDlg::distance[vertex[next_im].v -1 + mc*(cycle_j->vertex[next_j].v - 1)]
            - cost_of_2_edges_to_remove;

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
if(combined_cost1 < combined_cost2) // lines cross
{
    if(combined_cost1 < lowest_cost)
    {
        lowest_cost = combined_cost1;
        
        is = vertex[im].v;
        it = vertex[next_im].v;
        js = cycle_j->vertex[j].v;
        jt = cycle_j->vertex[next_j].v;
        
        vs_1 = vertex[im].v; // redundant but comprehensible
        vt_1 = cycle_j->vertex[next_j].v;
        vs_2 = vertex[next_im].v;
        vt_2 = cycle_j->vertex[j].v;
    }
}
else //(combined_cost2 <= combined_cost1) lines parallel
{
    if(combined_cost2 < lowest_cost)
    {
        lowest_cost = combined_cost2;
        
        is = vertex[im].v;
        it = vertex[next_im].v;
        js = cycle_j->vertex[j].v;
        jt = cycle_j->vertex[next_j].v;
        
        vs_1 = vertex[im].v;
        vt_1 = cycle_j->vertex[j].v;
        vs_2 = vertex[next_im].v;
        vt_2 = cycle_j->vertex[next_j].v;
    }
}
} //end for(jm...)
} //end for(i...

else // do the shortcut version of greedy
{
    while(found_edges_flag == false)
    {
        Nth_time = Nth_time + 1.0;
        max_measure = max_measure + increment*Nth_time;
        
        for(j=1; j<=lenj; j++) // ****** j ******
        {
            next_j = j+1;
            if( next_j == lenj+1){next_j=1;}
        } //end for(j)
for(im=1; im<=lenim; im=im+5) // ****** im ****** 
{
  next_im = im+1;
  if( next_im == lenim+1){next_im = 1;}

  if( CTSPViewDlg::distance[vertex[im].v -1 + mc*(cycle_j->vertex[j].v - 1)]< max_measure
      || CTSPViewDlg::distance[vertex[next_im].v -1 + mc*(cycle_j->vertex[next_j].v - 1)]< max_measure
      || CTSPViewDlg::distance[vertex[im].v -1 + mc*(cycle_j->vertex[next_j].v - 1)]< max_measure
      || CTSPViewDlg::distance[vertex[next_im].v -1 + mc*(cycle_j->vertex[j].v - 1)]< max_measure)
    { // <-----------------------------------------------<< max_measure if start
      i = (im-4)%lenim;
      if(i<=0){i=i+lenim;}
      endi = (im+5)%lenim;
      if(endi==0){endi=lenim;}

      while(i!=endi+1 ) // ****** i ******
      {
        if(i==lenim+1){i=1;}
        next_i = i+1;
        if( next_i == lenim+1){next_i=1;}
        found_edges_flag = true;

        cost_of_2_edges_to_remove =
          CTSPViewDlg::distance[vertex[i].v -1 + mc*(vertex[next_i].v - 1)]
          +
          CTSPViewDlg::distance[cycle_j->vertex[j].v -1 + mc*(cycle_j->vertex[next_j].v - 1)];

        combined_cost1 =
          CTSPViewDlg::distance[vertex[i].v -1 + mc*(cycle_j->vertex[next_j].v - 1)]
          +
          CTSPViewDlg::distance[vertex[next_i].v -1 + mc*(cycle_j->vertex[j].v - 1)]
          - cost_of_2_edges_to_remove;

        combined_cost2 =
          CTSPViewDlg::distance[vertex[i].v -1 + mc*(cycle_j->vertex[j].v - 1)]
          +
          CTSPViewDlg::distance[vertex[next_i].v -1 + mc*(cycle_j->vertex[next_j].v - 1)]
          - cost_of_2_edges_to_remove;

    } // <-----------------------------------------------<< max_measure if start
}
if(combined_cost1 < combined_cost2)
{
    if(combined_cost1 < lowest_cost)
    {
        lowest_cost = combined_cost1;
        is = vertex[i].v;
        it = vertex[next_i].v;
        js = cycle_j->vertex[j].v;
        jt = cycle_j->vertex[next_j].v;
        vs_1 = vertex[i].v;
        vs_2 = vertex[next_i].v;
        vt_1 = cycle_j->vertex[next_j].v;
        vt_2 = cycle_j->vertex[j].v;
    }
}
else // (combined_cost2 <= combined_cost1)
{
    if(combined_cost2 < lowest_cost)
    {
        lowest_cost = combined_cost2;
        is = vertex[i].v;
        it = vertex[next_i].v;
        js = cycle_j->vertex[j].v;
        jt = cycle_j->vertex[next_j].v;
        vs_1 = vertex[i].v;
        vs_2 = vertex[next_i].v;
        vt_1 = cycle_j->vertex[next_j].v;
        vt_2 = cycle_j->vertex[j].v;
    }
}
}
}
}
i++;
}} // <-----------------------------<< max_measure if end
} //end for(im...
} //end for(i...}
} // while(found_edges_flag...}
} //end else.
//Combine the two cycles within an EAXgraph
// first cycle
for(i=1; i<=leni; i++)
{
    next_i = i+1;
    if( next_i == leni+1){next_i=1;}
    if(vertex[i].v != is)
    {
        temp_EAXgraph.Add_Edge(vertex[i].v, vertex[next_i].v, A);
    }
}
// first connecting edge
temp_EAXgraph.Add_Edge(vs_1, vt_1, A);
// second cycle
j=1;
// consider vt_1 the starting vertex
while(cycle_j->vertex[j].v != vt_1) {j++;}
next_j = j+1;
if(next_j == lenj+1){next_j=1;}

// do increasing indexing thru cycle's array starting at j
if( cycle_j->vertex[next_j].v != vt_1 )
{
    // loop priming
    temp_EAXgraph.Add_Edge(cycle_j->vertex[j].v, cycle_j->vertex[next_j].v, A);
    j++;
    if(j == lenj+1){j=1;}
    next_j = j+1;
    if(next_j == lenj+1){next_j=1;}
    while(cycle_j->vertex[next_j].v != vt_1)
    {
        temp_EAXgraph.Add_Edge(cycle_j->vertex[j].v, cycle_j->vertex[next_j].v, A);
        j++;
        if(j == lenj+1){j=1;}
        next_j = j+1;
        if(next_j == lenj+1){next_j=1;}
    }
    else // do decreasing indexing thru cycle's array
    {
        // loop priming
        next_j = j-1;
        if(next_j == 0){next_j=lenj;}
        temp_EAXgraph.Add_Edge(cycle_j->vertex[j].v, cycle_j->vertex[next_j].v, A);
        j--;
        if(j == 0){j=lenj;}
        next_j = j-1;
        if(next_j == 0){next_j=lenj;}
        while(cycle_j->vertex[next_j].v != vt_1)
        {
            temp_EAXgraph.Add_Edge(cycle_j->vertex[j].v, cycle_j->vertex[next_j].v, A);
            j--;
            if(j == 0){j=lenj;}
            next_j = j-1;
            if(next_j == 0){next_j=lenj;}
        }
    }
}
//second connecting edge
temp_EAXgraph.Add_Edge(vt_2, vs_2, A); //direction is important

/*debug*/ //temp_EAXgraph.Write();

//translate back into a simple cycle
temp_cycle.Translate_from_EAXgraph_to_Cycle(temp_EAXgraph, is);
return temp_cycle;

} //End fn Combine_Greedily(...)

******1**********2**********3**********4**********5**********6**********7

//void Cycle::Clean()
{
int i;
for(i=0; i <= length; i++)
{
vertex[i].v = -1;
vertex[i].AorB = cZ;
vertex[i].x = -99999.9;
vertex[i].y = -99999.9;
}
}

******1**********2**********3**********4**********5**********6**********7

//void Cycle::Translate_from_EAXgraph_to_Cycle(EAXgraph& EAXg, int start)
{
//start is a non-empty vertex within the EAXgraph

int s;
int i,len;
len = EAXg.Get_Remaining();
vertex[1].v = start;
vertex[1].x = EAXg.Get_x(start,A);
vertex[1].y = EAXg.Get_y(start,A);
vertex[1].AorB = cA;

s = EAXg.Get_vtl(start,A);
for(i=2; i<=len; i++)
{
vertex[i].v = s;
vertex[i].x = EAXg.Get_x(s,A);
vertex[i].y = EAXg.Get_y(s,A);
vertex[i].AorB = cA;
s = EAXg.Get_vtl(s,A);
}
}
void Cycle::Write(int c, ofstream& outfile)
// c is the cycle number, and is not needed to run this output function.
// It is a parameter for debugging purposes.
int i;
    outfile<"n cycle #"<<c<<" cycle length = "<<length;
    outfile<". First edge is from parent ";

switch (vertex[l].AorB)
{
    case 0: outfile<" Z \n\n";
           break;
    case 1: outfile<" A \n";
           break;
    case 2: outfile<" B \n";
           break;
}
for(i=l;i<=length;i++)
{
    outfile<vertex[i].v;
    if(vertex[i].x != -99999.9)
    {
        outfile<" ("<vertex[i].x,"<vertex[i].y")"<endl;
    }
    else if(i!=length)outfile<<","
    outfile<endl<endl;
}

#ifndef EAX_CYCLE_H
#define EAX_CYCLE_H

/* EAX "Edge Assembly Crossover".
   This file SPECIFIES the Abstract Data Type 'EAXcycle'.
   An instance of EAXcycle represents a Hamilton cycle, that is, a
   cycle in which each edge can be traversed only once, but a given
   vertex can be visited more than once. Given that in the EAX
   algorithm a cycle is established from subsets of two tours over
   the same set of vertices, the resulting Hamilton cycle will have
   vertices with no more than two incoming edges and no more than
   two outgoing edges.
*/
#endif EAX_CYCLE_H
#define EAX_CYCLE_H

/* EAX "Edge Assembly Crossover".
   This file SPECIFIES the Abstract Data Type 'EAXcycle'.
   An instance of EAXcycle represents a Hamilton cycle, that is, a
   cycle in which each edge can be traversed only once, but a given
   vertex can be visited more than once. Given that in the EAX
   algorithm a cycle is established from subsets of two tours over
   the same set of vertices, the resulting Hamilton cycle will have
   vertices with no more than two incoming edges and no more than
   two outgoing edges.
*/

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
```cpp
#include "AorB_enum.h"
#include "GlobalMax.h"

struct cycle_struct // data type used within class EAXcycle
{
    bool visited; // The node has been visited
    int vt; // Outgoing edge endpoint, the "target vertex".
    // There are two possible outgoing, directed edges. One A, one B.
    bool vt_seen; // true implies edge vs->vt1 has been traversed,
    // where vs stands for the source vertex.
    bool vt_exists; // edge exists
};

class EAXcycle
{
    private:
    int length; // Maximum number of edges a cycle may hold. By
    // definition of the EAX algorithm, the number // of edges in a cycle must be even.
    int remaining; // current number of edges in a possible cycle
    cycle_struct g[MAX_SIZE][2];
    // NO LONGER USING DYNAMIC ALLOCATION. see version 6.
    // cycle_struct (*g)[2];
    public:
EAXcycle(); //constructor
EAXcycle(int); //constructor
EAXcycle(EAXcycle&); //copy constructor
operator=(EAXcycle&); //assignment operator
// ~EAXcycle(); //destructor
    //Modifiers
void Add_Edge(int,int,AorB_enum);
void Remove_Edge(int,int,AorB_enum);
void Remove_Stem(int ss,AorB_enum ssAorB,
int cs,AorB_enum csAorB);
void Mark_As_Visited(int vs);
void Mark_As_NOT_Visited(int vs);
void Remove_All(int,AorB_enum);
void Clean();
AorB_enum Swap_AorB(AorB_enum);
    //Accessors
int Count_Cycle(int,AorB_enum);
int Get_Length();
int Get_Remaining();
int Get_vt(int,AorB_enum);
bool Can_See_vt(int,AorB_enum);
bool Does_vt_exist(int,AorB_enum);
bool Vertex_Was_Visited(int); 
bool Graph_Is_Empty();
```
bool Edge_Can_Be_Seen(int,int,AorB_enum);
bool Edge_Exists(int,int,AorB_enum);
void Write(int,AorB_enum,int,AorB_enum,ofstream&);

#endif

// Begin EAX_cycle.cpp

#include "stdafx.h" #include "EAX_cycle.h"
#include <iostream.h> #include "AorB_enum.h"
#include "MutationDlg.h"
// This file IMPLEMENTS members of class EAXcycle

EAXcycle::EAXcycle() //default constructor used only for debugging purposes.
{
    length = 0;
    remaining = 0;
    // NO LONGER USING DYNAMIC ALLOCATION. see version 6.
    // g = new cycle_struct[length+1][2]; g[0][x] is not used
    g[0][A].visited = false;
    g[0][A].vt = -1; // -1 designates un-assigned
    g[0][A].vt_seen = false;
    g[0][A].vt_exists = false;
    g[0][B].visited = false;
    g[0][B].vt = -1;
    g[0][B].vt_seen = false;
    g[0][B].vt_exists = false;
}

EAXcycle::EAXcycle(int len)
//constructor
//POST: An AB graph with a potential of 'len' vertices is allocated // memory and is initialized to default values. No edges yet exist.
{
    int i;
    length = len;
    remaining = 0;
    // NO LONGER USING DYNAMIC ALLOCATION. see version 6 for dynamic codes
    // g = new cycle_struct[length+1][2]; g[0][x] is not used
    for(i=0;i<=length;i++)
    {
        g[i][A].visited = false;
        g[i][A].vt = -1; // -1 designates un-assigned
        g[i][A].vt_seen = false;
        g[i][A].vt_exists = false;
    }
g[i][A].vt_exists = false;
g[i][B].visited = false;
g[i][B].vt = -1;
g[i][B].vt_seen = false;
g[i][B].vt_exists = false;
}
}  

//**********1**********2**********3**********4**********5**********6**********8
//
EAXcycle::EAXcycle(EAXcycle& otherEAXcycle) //copy constructor
{
int i;
length = otherEAXcycle.Get_Length();
remaining = otherEAXcycle.Get_Remaining();
for(i=0; i <= length; i++)
{
    g[i][A].visited = otherEAXcycle.Vertex_Was_Visited(i);
g[i][A].vt = otherEAXcycle.Get_vt(i,A);
g[i][A].vt_seen = otherEAXcycle.Can_See_vt(i,A);
g[i][A].vt_exists = otherEAXcycle.Does_vt_exist(i,A);
g[i][B].visited = otherEAXcycle.Vertex_Was_Visited(i);
g[i][B].vt = otherEAXcycle.Get_vt(i,B);
g[i][B].vt_seen = otherEAXcycle.Can_See_vt(i,B);
g[i][B].vt_exists = otherEAXcycle.Does_vt_exist(i,B);
}
}  

//**********1**********2**********3**********4**********5**********6**********8
//
EAXcycle::operator=(EAXcycle& otherEAXcycle) //Assignment operator
//PRE: Two cycles having same length exist
{
int i;
//length = otherEAXcycle.Get_Length();<--- must not change
remaining = otherEAXcycle.Get_Remaining();
for(i=0; i <= length; i++)
{
    g[i][A].visited = otherEAXcycle.Vertex_Was_Visited(i);
g[i][A].vt = otherEAXcycle.Get_vt(i,A);
g[i][A].vt_seen = otherEAXcycle.Can_See_vt(i,A);
g[i][A].vt_exists = otherEAXcycle.Does_vt_exist(i,A);
g[i][B].visited = otherEAXcycle.Vertex_Was_Visited(i);
g[i][B].vt = otherEAXcycle.Get_vt(i,B);
g[i][B].vt_seen = otherEAXcycle.Can_See_vt(i,B);
g[i][B].vt_exists = otherEAXcycle.Does_vt_exist(i,B);
}
}

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
EAXcycle::~EAXcycle()
// destructor
{
    // not using dynamic memory on this version
    // delete [] g;
};

void EAXcycle::Add_Edge(int vs, int vt, AorB enum AorB)
// PRE: edge vs-->vt does not exist  // POST: Edge vs-->vt exists.
// The graph built by a series of calls to this function.
{
    if( !Does_vt_exist(vs, AorB) )
    {
        g[vs][AorB].vt = vt;
        g[vs][AorB].vt_exists = true;
        g[vs][AorB].vt_seen = true;  }
    else
    {
        CString filename = CMutationDlg::stats_output_filename;
        ofstream outfile;
        outfile.open(filename, ios::app);
        outfile<<"\nError in fn:EAXcycle::Add_Edge(...)";
        outfile<<" Attempt to overwrite an existing edge. edge = ",
        if(AorB == 0)
        {
            outfile<<"A, ";
        }
        else
        {
            outfile<<"B, ";
        }
        outfile<<vs<<"",<<vt<<endl;
        outfile.close();
    }
    remaining++; // records the number of edges in the graph
    Mark_As_Visited(vs); // designates that the source vertex has been visited
}

void EAXcycle::Remove_Edge(int vs, int vt, AorB enum AorB)
// PRE: edge exists,...keep in mind that an edge is represented by
// a source vertex and a destination vertex (a target vertex).
// In this class's implementation the source is the first
// index of a 2-D array.
{
    //cerr<<"\nEnter Remove_Edge function. ");
if (vt == g[vs][AorB].vt) //looking for correct edge
{
    remaining--;
    g[vs][AorB].vt = -1;
    g[vs][AorB].vt_seen = false;
    g[vs][AorB].vt_exists = false;
}
else
{
    cerr<<"Error in file EAXcycle.cpp, fn EAXcycle::Remove_Edge(...),";
    cerr<<" Designated edge to be removed not found.\n";
}
//cerr<<"Exit Remove_Edge fimction.\n";

//********1********2********3********4********5********6********

// void EAXcycle::Remove_Stem(int ss, AorB_enum ssAorB,
//    int cs, AorB_enum csAorB)
// ss = stem source, the very first edge added to a possible cycle. // cs = cycle source
// ssAorB = parent designation for stem source
// csAorB = parent designation for cycle source
{
    int r;
    int vs;
    int source;
    AorB_enum AorB;
    bool flag;
    int temp_source;
    vs = ss;
    r = Get_Remaining();
    // Mark ALL edges in the 'cycle' as seen
    source = cs;
    AorB = csAorB;
    flag = true;
    while(flag)
    {
        g[source][AorB].vt_seen = true;
        source = Get_vt(source, AorB);//increment
        AorB = Swap_AorB(AorB);
        if(source == cs)
        {
            flag = false;
        }
    }
    // remove the stem
    source = ss;
    AorB = ssAorB;
    flag = true;
while( Get_vt(source, AorB) != cs )
{
    temp_source = g[source][AorB].vt;
    Remove_Edge( source, g[source][AorB].vt, AorB );
    source = temp_source;
    AorB = Swap_AorB(AorB);
}
// last one:
Remove_Edge( source, g[source][AorB].vt, AorB );
}

void EAXcycle::Mark_As_Visited(int vs)
// When a node is marked as visited, BOTH A and B are marked, since
// a source node is considered one and the same for both parents.
{
    g[vs][A].visited = true;
    g[vs][B].visited = true;
}

void EAXcycle::Mark_As_NOT_Visited(int vs)
// When a node is marked as visited, BOTH A and B are marked, since
// a source node is considered one and the same for both parents.
{
    g[vs][A].visited = false;
    g[vs][B].visited = false;
}

void EAXcycle::Remove_All(int vs, AorB enum A_B)
// removes edges up to first empty edge...short version of Clean()
{
    AorB enum AorB;
    int vs_temp;
    int count;
    count = 1;
    AorB = A_B;
    while( g[vs][AorB].vt != -1 && count<=remaining )
    {
       vs_temp = g[vs][AorB].vt;
       g[vs][AorB].visited = false;
       g[vs][AorB].vt = -1;
       g[vs][AorB].vt_seen = false;
       g[vs][AorB].vt_exists = false;
       AorB = Swap_AorB(AorB);
    }
g[vs][AorB].visited = false;
g[vs][AorB].vt = -1;
g[vs][AorB].vt_seen = false;
g[vs][AorB].vt_exists = false;
vs = vs_temp;
count++;
}
remaining = 0;

// EAXcycle::Clean()
// PRE: A graph has been instantiated via a constructor and may have
// had vertices and edges added to it.
// POST: The represented graph has no existing edges nor any
// vertices, but the private data member 'length' remains the same.
{
int i;
remaining = 0;
for(i=0;i<=length;i++) {
g[i][A].visited = false;
g[i][A].vt = -1; // -1 designates un-assigned  g[i][A].vt_seen = false;
g[i][A].vt_exists = false;
g[i][B].visited = false;
g[i][B].vt = -1;
g[i][B].vt_seen = false;
g[i][B].vt_exists = false;
}

// AorB_enum EAXcycle::Swap_AorB(AorB_enum AorB)
{
    if(AorB == A)
    {
        return B;
    }
    else
    {
        return A;
    }
}

// EAXcycle::Count_Cycle(int cs, AorB_enum csAorB)
// This function called by PossibleCycle.
// Return value is the number of edges in the cycle.
// Returns 0 if this function fails.
// cs = supposed start of cycle
// csAorB = source designation for starting edge
{
  int vs, count;
  AorB_enum AorB;
  vs = cs;
  AorB = csAorB;
  count = 1;
  vs = Get_vt(vs, AorB);
  while( vs != cs )
  {
    count++;
    AorB = Swap_AorB(AorB);
    vs = Get_vt(vs, AorB);
  }
  return count;
}

int EAXcycle::Get_Length()
//POST: private data member 'length' returned
{
  return length;
}

int EAXcycle::Get_Remaining()
//POST: private data member 'remaining' returned
{
  return remaining;
}

int EAXcycle::Get_vt(int vs, AorB_enum AorB)
{
  return g[vs][AorB].vt;
}

bool EAXcycle::Can_See_vt(int vs, AorB_enum AorB)
{
  return g[vs][AorB].vt_seen;
}

bool EAXcycle::Does_vt_exist(int vs, AorB_enum AorB)
{
  return g[vs][AorB].vt_exists;
}
bool EAXcycle::Vertex_Was_Visited(int vs)
{
    return g[vs][A].visited;
    // note that when a vertex is marked as visited both g[vs][A].visited
    // and g[vs][B].visited are true.
}

/*******************************************************************************/
bool EAXcycle::Graph_Is_Empty()
{
    if(remaining == 0) return true;
    else return false;
}

/*******************************************************************************/
bool EAXcycle::Edge_Can_Be_Seen(int vs, int vt, AorB enum AorB)
// If an edge exists, it may not have been traversed and in such
// a case is not registered as seen.
{
    if( Get_vt(vs, AorB) == vt && g[vs][AorB].vt_seen)
    {
        return true;
    }
    else return false;
}

/*******************************************************************************/
bool EAXcycle::Edge_Exists(int vs, int vt, AorB enum AorB)
{
    if( Get_vt(vs, AorB) == vt && Does_vt_exist(vs, AorB) == true)
    {
        return true;
    }
    else
    {
        return false;
    }
}

/*******************************************************************************/
void EAXcycle::Write(int ss, AorB enum ssAorB,
    int cs, AorB enum csAorB,
    ofstream& outfile)
// ss = stem source
// cs = cycle source
{
int vs;
AorB enum AorB;
outfile<<"PossibleCycle.remaining = "<<remaining<<endl;
if(g[ss][ssAorB].vt == -1)
{
    outfile<<"g[ss][ssAorB].vt = "<<g[ss][ssAorB].vt;
    outfile<<" Stem truncated."<<endl;
}
else if(ss == cs)
{
    outfile<<" There is no stem. ss = "<<ss<< " ";
    outfile<<"g[ss][ssAorB].vt = "<<g[ss][ssAorB].vt;
    outfile<<" cs = "<<cs<<endl;
    outfile<<"cycle: ";
    outfile<<cs<<" ";
    vs = cs;
    AorB = csAorB;
    while(g[vs][AorB].vt != cs)
    {
        outfile<<g[vs][AorB].vt<<", ";
        vs = g[vs][AorB].vt;
        AorB = Swap_AorB(AorB);
    }
    outfile<<cs<<endl;
}
else// <------------------ — wherein a stem exists
{
    outfile<<"PossibleCycle Stem: ";
    outfile<<ss<<" ";
    vs = ss;
    AorB = ssAorB;
    if(g[vs][AorB].vt == cs)
    {//true -> there exists just one edge in the stem
        outfile<<cs<<endl;
    }
    else
    {
        while(g[vs][AorB].vt != cs)
        {
            outfile<<g[vs][AorB].vt<<", ";
            vs = g[vs][AorB].vt;
            AorB = Swap_AorB(AorB);
        }
        outfile<<cs<<endl;
    }
}
    outfile<<"PossibleCycle Cycle: ";
    outfile<<cs<<" ";
    vs = cs;
    AorB = csAorB;
while(g[vs][AorB].vt != cs)
{
    outfile<<g[vs][AorB].vt<<"", "
    vs = g[vs][AorB].vt;
    AorB = Swap_AorB(AorB);
}
    outfile<<cs<<endl;
}
CHAPTER 5

DRIVER

The edge assembly crossover algorithm, EAX, is contained in ArrayGenome.cpp, a file within the TSP View graphic user interface. There are five other crossover functions in ArrayGenome.cpp. All six crossovers are instantiations of the CArrayGenome class. EAX is found in function int CArrayGenome::Cross6(...).

I originally implemented EAX using dynamic instantiations wherever it would be possible to have such call outs in an algorithm whose input data set is of unknown size and could be very large. This, however, proves to be idealistic. The program runs faster the fewer dynamic memory requests. Instead of dynamic heap allocations, static stack allocations prove to be fast up to the point of overwhelming the stack limits.

EAX is called as many times as the specified GAlib genetic algorithm concept of reproduction requires. This is set in TSP View at run time, but a reading of GAlib: A C++ Library of Genetic Algorithm Components (Wall 1996) will reveal a number of ways to change the behavior of the genetic algorithm not included in TSP View. I use two parents to create two children within one call of the algorithm. First Parent A is combined with a randomly developed E set in order to produce the first child, then parent B is combined with another randomly developed, independent E set in order to produce the second child. Functions representing EAX stages one, two, three, and four are called in succession with only dynamic bookkeeping between the calls. The array to which the resulting child tours are returned to the GAlib calling function is assigned after the four stages and the number of children resulting, int nc, is incremented, as seen on page 70.
// ArrayGenome.h: interface for the CArrayGenome class.

#include "ga.h"
#include <afxtempl.h>

#if !defined(AFX_ARRAYGENOME_H_BEB2E062_1441_11D3_8317_D1CDBDD76C7F_INCLUDED)
#define AFX_ARRAYGENOME_H_BEB2E062_1441_11D3_8317_D1CDBDD76C7F_INCLUDED
#if _MSC_VER > 1000
#pragma once
#endif // _MSC_VER > 1000

class CArrayGenome : public CArray<int, int>, public GAGenome {
public:
    static int Cross (const GAGenome& gl, const GAGenome& g2, GAGenome* cl, GAGenome* c2);
    static int Cross1(const GAGenome &g1, const GAGenome &g2, GAGenome *c1, GAGenome *c2);
    static int Cross2(const GAGenome &g1, const GAGenome &g2, GAGenome *c1, GAGenome *c2);
    static int Cross3(const GAGenome &g1, const GAGenome &g2, GAGenome *c1, GAGenome *c2);
    static int Cross4(const GAGenome &g1, const GAGenome &g2, GAGenome *c1, GAGenome *c2);
    static int Cross5(const GAGenome &g1, const GAGenome &g2, GAGenome *c1, GAGenome *c2);//SCC
    static int Cross6(const GAGenome &g1, const GAGenome &g2, GAGenome *c1, GAGenome *c2);//EAX
    static float Compare(const GAGenome&, const GAGenome&);
    static int Mutate(GAGenome&, float);
    static void Init(GAGenome&);
    static float p_rotate;
    static float p_swap;
    static float p_invert;

    GADefineldentity("CArrayGenome", 201);

};
public:
CString static cross_type;
BOOL opt_score;
//-----------------------------------------------
CArrayGenome(float rot, float invert, float swap):GAGenome(Init, Mutate, Compare)
{
    crossover(Cross);
    //GAGenome.h has implementation:
    // SexualCrossover crossover(SexualCrossover f){ return sexcross = f; }
    // AsexualCrossover crossover(AsexualCrossover f){ return asexcross = f; }
    // These functions are members of GAGenome class.
    p_rotate = rot;
    p_invert = invert;
    p_swap = swap;
    opt_score = FALSE;
}
//-----------------------------------------------
CArrayGenome(const CArrayGenome& orig)
{
    copy(orig);
}
//-----------------------------------------------
virtual ~CArrayGenome() {}
//-----------------------------------------------
CArrayGenome& operator=(const GAGenome& orig)
{
    if(&orig != this) copy(orig);
    return *this;
}
//-----------------------------------------------
virtual GAGenome* clone(CloneMethod) const
{
    return new CArrayGenome(*this);
}
//-----------------------------------------------
virtual void copy(const GAGenome& orig)
{
    GAGenome::copy(orig);
    CArrayGenome& newbie = (CArrayGenome&) orig;
    SetSize(newbie.GetSize());
    opt_score = newbie.opt_score;
    Copy(newbie);
}
};
#endif

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
The following is a listing of the EAX driver.

// ArrayGenome.cpp: implementation of the CArrayGenome class.

#include "stdafx.h"  #include "TSP View2.h"
#include "TSP ViewDlg.h"
#include "ArrayGenome.h"
#include <iostream.h>
#include <iomanip.h>
#include <fstream.h>
#include <iostream.h>
#include <fstream.h>
#include "MutationDlg.h"
#include "EAX_graph.h"
#include <stdlib.h>
#include <time.h>
#include "AorB_enum.h"
#include "cycle.h"
#include <math.h>
#include "GlobalMax.h"

extern const int MAX_SIZE;  // see GlobalMax.h to change this constant

// primary EAX functions
void EAX_Stage1_Create_ABcycles(const int*, const intQ, Cycle* [], int&, const int, 
                                EAXgraph&, EAXgraph&);

void EAX_Stage2_Create_Eset(Cycle* [], int, Cycle* [], int&);

void EAX_Stage3_Create_Intermediate_Child(Cycle* [], int, 
                                     EAXgraph&, EAXgraph&);

void EAX_Stage4_Create_Child(EAXgraph&, Cycle&, int);

// support functions
void CopyToSimpleContainer(Cycle*, int, EAXcycle&, int, int, AorB Enum);

void AssignCoordinatesToCycles(Cycle* [], int cycles);

void Available_Number_ArrayReducer(const int, const int, 
                                   int*, int*, int&, 
                                   EAXgraph&, AorB Enum);

void Available_Number_ArrayReducer(const int, int*, int*, int&);

void Stem_Rebuilder(const int, const int, int*, int*, int*, int*, 
                    int&, EAXgraph&, AorB Enum, EAXcycle&);
void Available_Number_Array_Rebuilder(const int,const int, int*, int*, int&,
   EAXgraph&, AorB_enum);

void Available_Number_Array_Cycle.Remover(int*, int*, int*, int*, int*, int&,
   AorB_enum, EAXcycle&);

void QuickSortDisjointSet(Cycle*, int, int, Cycle*, Cycle*);

#ifdef_DEBUG
#define THIS_FILE
static char THIS_FILE[] = __FILE__;
#define new DEBUG_NEW
#endif

#include "crossovers_1_thru_5.h"

int CArrayGenome::Cross6(const GAGenome &g1, const GAGenome &g2,
            GAGenome *cl, GAGenome *c2)
   //EAX, Edge Assembly Crossover after Nagata & Kobyashi
   {
      int i;
      int A_array[MAX_SIZE];
      int B_array[MAX_SIZE];

      int nc = 0; // number of children
      int cycles = 0; // The number of cycle created by using Rgraph
      int Ecycles;
      Cycle *pABcycles1[MAX_SIZE]; // dynamics minimized this version
      Cycle *pABcycles2[MAX_SIZE];
      Cycle *Eset1[MAX_SIZE];
      Cycle *Eset2[MAX_SIZE];

      CArrayGenome & mom = (CArrayGenome &)*cl;
      CArrayGenome & dad = (CArrayGenome &)*c2;
      CArrayGenome & sis = (CArrayGenome &)*c1;
      CArrayGenome & bro = (CArrayGenome &)*c2;
      // the GAGenome parameter is a base class without
      // utility functions, so the explicit coercion is necessary.
      int array_size = mom.GetSize();

      Cycle Child_A(array_size);
      Cycle Child_B(array_size);
EAXgraph A_tour(array_size);
EAXgraph B_tour(array_size);

// A_array = new int[array_size]; dynamics: just say no
// B_array = new int[array_size];

EAXgraph Ichild_A(array_size);// Ichild == "Intermediate" child
EAXgraph Ichild_B(array_size);

// Transfer data. Another place code could be made more efficient.
// Add one because GA Genomes of size x are labeled 0 thru x-1, while
// Cross6 EAX uses 1 thru x.
for (i=0;i<array_size;i++)
{
    A_array[i] = mom[i]+1;
    B_array[i] = dad[i]+1;
}

//********** first child **********
cycles = 0;
EAX_Stage1_Create_ABcycles(A_array,B_array,pABcycles1,cycles,
array_size,A_tour,B_tour);
Ecycles = 0;
EAX_Stage2_Create_Eset(pABcycles1, cycles, Eset1, Ecycles);
for(i=0; i<cycles; i++)
{
    delete pABcycles1[i];
}
EAX_Stage3_Create_Intermediate_Child(Eset1, Ecycles, Ichild_A, A_tour);
for(i=0; i<Ecycles; i++)
{
    delete Eset1[i];
}
EAX_Stage4_Create_Child(Ichild_A, Child_A, array_size);
for(i=0;i<array_size;i++)
{
    sis[i] = Child_A.vertex[i+1].v - 1;
    // Subtract one because GA Genomes of size x are labeled
    // 0 thru x-1. EAX algorithm uses 1 thru x.
}
nc++;

//********** second child **********
cycles = 0;
EAX_Stage1_Create_ABcycles(A_array,B_array,pABcycles2,cycles,
array_size,A_tour,B_tour);
Ecycles = 0;
EAX_Stage2_Create_Eset(pABcycles2, cycles, Eset2, Ecycles);
for(i=0; i<cycles; i++)
{
    delete pABcycles2[i];
}

EAX_Stage3_Create_Intermediate_Child(Eset2, Ecycles, Ichild_B, B_tour);

for(i=0; i<Ecycles; i++)
{
    delete Eset2[i];
}
EAX_Stage4_Create_Child(Ichild_B, Child_B, array_size);
for(i=0; i<array_size; i++)
{
    bro[i] = Child_B.vertex[i+1].v - 1;
}
nc++;
return nc;

}  

//*****wj*********2*********3*********4*********5*********6*********68

//
void EAX_Stage1_Create_ABcycles(const int A_array[], const int B_array[],
    Cycle* pABcycles[], int& cycles, int array_size,
    EAXgraph& A_tour,
    EAXgraph& B_tour )

//PRE: two arrays, A_array and B_array, which contain a series of
//numbers representing tours of the same set of nodes.
//POST: A set of AB cycles is constructed
{
    int vs, vt; //source vertex, target vertex, cycle
    int i;
    struct Edge_type
    {
        int vs;
        int vt;
        AorB_enum AorB;
    };
    Edge_type stem_source; //represents the first edge of the possible cycle,
    //and therefore the first edge of a possible stem.

    Edge_type cycle_source;
    //the determined first edge of the cycle
    AorB_enum AorB;
    int vs_temp, vt_temp; AorB_enum AorB_temp;
    bool a_cycle_exists;
    EAXgraph R_graph(array_size);
    EAXcycle PossibleCycle(array_size);
    int A_n[MAX_SIZE];
int A_p[MAX_SIZE];
int B_n[MAX_SIZE];
int B_p[MAX_SIZE];

int upper_limitA = array_size;;
int upper_limitB = arraysize;
/*
int* A_n;
int* A_p;
int* B_n;
int* B_p;
A_n  = new int[upper_limitA+1];
A_p = new int[upper_limitA+1];
B_n = new int[upper_limitB+1];
B_p = new int[upper_limitB+1];
*/
    for( i=1; i<=upper_limitA; i++)
    {
        A_n[i] = i;
        A_p[i] = i;
        B_n[i] = i;
        B_p[i] = i;
    }

//parent A
A_tour.Load_Tour_from_Array(A_array, array_size, A);
//parent B
B_tour.Load_Tour_from_Array(B_array, array_size, B);
//composite of parents
R_graph.Build_Rgraph_from_two_Graphs(A_tour, B_tour);
CString filename = CMutationDlg::stats_output_filename;
ofstream outfile;
outfile.open(filename, ios::app);
R_graph.Write(outfile);
while(!R_graph.Is_Empty())// <--------<< begin outer loop
{
    // Choose first edge at random...
    AorB = A;  //always starts with A
    vs = R_graph.Get_vs_Randomly(A_n, A_p, upper_limitA);
    PossibleCycle.Mark_As_Visited(vs);
    vt = R_graph.Get_vt_Randomly(vs, AorB);
    // Once first edge is found add it to the possible cycle:
    PossibleCycle.Add_Edge(vs, vt, AorB);
    // The R_graph keeps track of the edges it offers to
    // cycle building efforts
    R_graph.Mark_Edge_As_Seen(vs,vt,AorB);
    Available_Number_ArrayReducer(vs,vt,A_n,A_p,
        upper_limitA,R_graph,A);
    // Initial edge of the possible cycle is re-saved,
    // as it may not be part of the future cycle,
// but instead the leading edge of a useless stem.
stem_source.vs = vs;
stem_source.vt = vt;
stem_source.AorB = AorB;
// first edge is used as a priming edge for the following
// loop which adds edges to the possible cycle until an AB cycle is achieved. The
// following additions of edges choose between two possible paths at each new node.
a_cycle_exists = false;
while(!a_cycle_exists) // <----------------<< begin inner loop
{
    AorB = PossibleCycle.Swap_AorB(AorB);
    vs = vt;
    PossibleCycle.Mark_As_Visited(vs);
    vt = R_graph.Get_vt_Randomly(vs, AorB);
    PossibleCycle.Add_Edge(vs, vt, AorB);
    R_graph.Mark_Edge_As_Seen(vs, vt, AorB);
    if(AorB == A)
    {
        Available_Number_ArrayReducer(vs, vt, A_n, A_p, upper_limitA, R_graph, A);
    }
    else // AorB == B
    {
        Available_Number_ArrayReducer(vs, vt, B_n, B_p, upper_limitB, R_graph, B);
    }
} //Since edges > 2, begin check for an A-B cycle.
AorB temp = PossibleCycle.Swap_AorB(AorB);
vs_temp = vt;
vt_temp = PossibleCycle.Get_vt(vs_temp, AorB_temp);
// i.e. Is the endpt of last edge a previously visited
// node, and are the first and last edges
// of the cycle of different parentage?
if(PossibleCycle.Vertex_Was_Visited(vs_temp)
    & &
    PossibleCycle.Edge_Exists(vs_temp, vt_temp, AorB_temp))
{
    cycle_source.vs = vs_temp;
cycle_source.vt = vt_temp;
cycle_source.AorB = AorB_temp;
    if(cycle_source.vs != stem_source.vs)
    {
        Stem_Rebuilder(stem_source.vs, cycle_source.vs,
        A_n, A_p, upper_limitA,
        B_n, B_p, upper_limitB,
        R_graph, stem_source.AorB, PossibleCycle);
        PossibleCycle.Remove_Stencil(stem_source.vs,
        stem_source.AorB,
        cycle_source.vs,
        cycle_source.AorB);
    }
}
pABcycles[cycles] = new Cycle(PossibleCycle.Get_Remaining());
if(!pABcycles[cycles])
{
    CString filename = CMutationDlg::stats_output_filename;
    ofstream outfile;
    outfile.open(filename, ios::app);
    outfile<<"* Memory Error 1 * in fn:EAX_Stage4";
    outfile<<" " Create_Child(...) ";
    outfile<<" file ArrayGenome.cpp\n";
    outfile.close();
}

CopyToSimpleContainer(pABcycles[cycles],cycles,PossibleCycle,
    cycle_source.vs,cycle_source.vt,cycle_source.AorB);

R_graph.Remove_Cycle(PossibleCycle,cycle_source.vs,
    cycle_source.AorB);
    a_cycle_exists = true;
}///<--- end---< if( PossibleCycle.Was_Visited(vt,AorB) )
}///<-------- end---< inner loop
cycles++;
PossibleCycle.Clean();
}///<-------- end---< outer loop
/
    delete [] B_p;
    delete [] B_n;
    delete [] A_p;
    delete [] A_n;
*/
}///<----------end---< function EAX_Stage1(...)

/************1**********2**********3**********4**********5**********6**********68

//

void EAX_Stage2_Create_Eset(Cycle* pABcycles[], int cycles,
    Cycle* Eset[], int& Ecycles)
{
    int i, j, counter, max_cycle_length, two_length_counter;

    max_cycle_length = 0;
    for(i=0; i<cycles; i++)
    {
        if(pABcycles[i]->length > max_cycle_length)
        {
            max_cycle_length = pABcycles[i]->length;
        }
    }

    Cycle temp_cycle(max_cycle_length);
    counter = 0;
    two_length_counter = 0;
while (counter == 0) // insures Eset is not empty
{
    for (i = 0; i < cycles; i++)
    {
        //srand((unsigned)(time(NULL)));
        // Turn this on and algorithm becomes NOT random.
        if ((pABcycles[i]->length > 2) && (rand() % 2 == 1))
        {
            temp_cycle.length = pABcycles[i]->length;
            for (j = 1; j <= pABcycles[i]->length; j++)
            {
                temp_cycle.vertex[j].v = pABcycles[i]->vertex[j].v;
                temp_cycle.vertex[j].AorB = pABcycles[i]->vertex[j].AorB;
                temp_cycle.vertex[j].x = pABcycles[i]->vertex[j].x;
                temp_cycle.vertex[j].y = pABcycles[i]->vertex[j].y;
            }
            Eset[counter] = new Cycle(temp_cycle.length);
            // Cycle type arrays use indexes 0 thru n-1
            *Eset[counter] = temp_cycle;
            // necessary de-reference for deep assignment
            temp_cycle.Clean();
            counter++;
        }
        else if (pABcycles[i]->length == 2)
        {
            two_length_counter++;
        }
    }
    if (two_length_counter == cycles) break; // true -> The two parents are the same tour.
}
Ecycles = counter;

//**********1**********2**********3**********4**********5**********6**********7
//
void EAX_Stage3_Create_Intermediate_Child(Cycle* Eset[],
    int Ecycles,
    EAXgraph& Ichild,
    EAXgraph& X_tour)
{
    int ij, len;
    AorB enum parent_tour_AorB;

    if (X_tour.Get_vt1_exists(1, A))
    {
        parent_tour_AorB = A;
    }
}
else
{
    parent_tour_AorB = B;
}
Ichild = X_tour;
for(i=0; i<=Ecycles; i++)
{
    len = Eset[i]->length;
    for(j=1; j<=len; j++)
    {
        if(parent_tour_AorB = A) // <-------- deal with parent A
        {
            if( Eset[i]->vertex[j].AorB == cA )
            {
                if( Ichild.Edge_Exists(Eset[i]->vertex[j].v, Eset[i]->vertex[j%len+1].v, A))
                {
                    // removals
                    Ichild.Remove_Edge(Eset[i]->vertex[j].v, Eset[i]->vertex[j%len+1].v, A);
                }
                else
                {
                    // additions
                    Ichild.Add_Edge(Eset[i]->vertex[j].v, Eset[i]->vertex[j%len+1].v, B);
                }
            }
            else // <-----------------------------deal with parent B
            {
                if( Eset[i]->vertex[j].AorB == cB )
                {
                    if( Ichild.Edge_Exists(Eset[i]->vertex[j].v, Eset[i]->vertex[j%len+1].v, B))
                    {
                        // removals
                        Ichild.Remove_Edge(Eset[i]->vertex[j].v, Eset[i]->vertex[j%len+1].v, B);
                    }
                    else
                    {
                        // additions
                        Ichild.Add_Edge(Eset[i]->vertex[j].v, Eset[i]->vertex[j%len+1].v, A);
                    }
                }
            }
        }
    }
}

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
/* debug*/ //outfile"Exit Stage_3\n",
} // End Stage_3
//**************2**************3**************4**************5**************6**************68
// void EAX_Stage4_Create_Child(EAXgraph& Ichild,
        Cycle& Child,
        int array_size)
{
    int i;
    int vs = 1;
    int start = 1;
    int previous = 0;
    int cycle_count = 0;
    int minimum_starting_place = 1;
    int len = Ichild.Get_Length();
    int cycles_in_set = 0;
    bool cycle_incomplete = true;
    bool all_vertexes_seen = false;
    int A_n[MAX_SIZE];
    int A_p[MAX_SIZE];
    int upper_limitA = len;
    Cycle* pivot; // Quicksort utility variables
    pivot = new Cycle(array_size);
    Cycle* tempCycle;
    tempCycle = new Cycle(array_size);
    /*
    int* A_n; // grab bag reducer of numbers left in n time
    int* A_p;
    A_n = new int[upper_limitA+1]; //index 0 not used
    if(!A_n)
    {
        CString filename = CMutationDlg::stats_output_filename;
        ofstream outfile;
        outfile.open(filename, ios::app);
        outfile<<"* Memory Error 1 * in fn:EAX_Stage4",
        outfile<<"_Create_Child(...), file ArrayGenome.cpp\n"
        outfile.close();
    }
    A_p = new int[upper_limitA+1];
    if(!A_p)
    {
        CString filename = CMutationDlg::stats_output_filename;
        ofstream outfile;
        outfile.open(filename, ios::app);
        outfile<<"* Memory Error 2 * in fn:EAX_Stage4",
        outfile<<"_Create_Child(...), file ArrayGenome.cpp\n"
        outfile.close();
    }
    */
for (i=1; i<=upper_limitA; i++)
{
    A_n[i] = i;
    A_p[i] = i;
}

// The intermediate child consists of disjoint cycles.
Cycle* Disjoint_Set[MAX_SIZE];
Cycle temp_cycle(len);    //Step 1.) Contain cycles as a set.
while(!all_vertices_seen)
{
    while(cycle_incomplete)
    {
        Available_Number_Array_REducer(vs, A_n, A_p, upper_limitA);
        if (Ichild.Edge_Exists(vs, Ichild.Get_vtl(vs,A), A) &&
            !Ichild.Edge_Can_Be_Seen(vs, Ichild.Get_vtl(vs,A), A) &&
            Ichild.Get_vtl(vs,A) != previous)
        {
            cycle_count++;
            temp_cycle.vertex[cycle_count].v = vs;
            temp_cycle.vertex[cycle_count].AorB = cA;
            Ichild.Mark_Edge_As_Seen(vs, Ichild.Get_vtl(vs,A), A);
            previous = vs;
            vs = Ichild.Get_vtl(vs,A);
        }
        else if (Ichild.Edge_Exists(vs, Ichild.Get_vtl2(vs,A), A) &&
            !Ichild.Edge_Can_Be_Seen(vs, Ichild.Get_vtl2(vs,A), A) &&
            Ichild.Get_vtl2(vs,A) != previous)
        {
            cycle_count++;
            temp_cycle.vertex[cycle_count].v = vs;
            temp_cycle.vertex[cycle_count].AorB = cA;
            Ichild.Mark_Edge_As_Seen(vs, Ichild.Get_vtl2(vs,A), A);
            previous = vs;
            vs = Ichild.Get_vtl2(vs,A);
        }
        else if (Ichild.Edge_Exists(vs, Ichild.Get_vtl(vs,B), B) &&
            !Ichild.Edge_Can_Be_Seen(vs, Ichild.Get_vtl(vs,B), B) &&
            Ichild.Get_vtl(vs,B) != previous)
        {
            cycle_count++;
            temp_cycle.vertex[cycle_count].v = vs;
            temp_cycle.vertex[cycle_count].AorB = cB;
            Ichild.Mark_Edge_As_Seen(vs, Ichild.Get_vtl(vs,B), B);
            previous = vs;
            vs = Ichild.Get_vtl(vs,B);
        }
    }
    while(!cycle_incomplete)
    {
        if (Ichild.Edge_Exists(vs, Ichild.Get_vtl2(vs,A), A) &&
            !Ichild.Edge_Can_Be_Seen(vs, Ichild.Get_vtl2(vs,A), A) &&
            Ichild.Get_vtl2(vs,A) != previous)
        {
            cycle_count++;
            temp_cycle.vertex[cycle_count].v = vs;
            temp_cycle.vertex[cycle_count].AorB = cA;
            Ichild.Mark_Edge_As_Seen(vs, Ichild.Get_vtl2(vs,A), A);
            previous = vs;
            vs = Ichild.Get_vtl2(vs,A);
        }
        else if (Ichild.Edge_Exists(vs, Ichild.Get_vtl2(vs,B), B) &&
            !Ichild.Edge_Can_Be_Seen(vs, Ichild.Get_vtl2(vs,B), B) &&
            Ichild.Get_vtl2(vs,B) != previous)
        {
            cycle_count++;
            temp_cycle.vertex[cycle_count].v = vs;
            temp_cycle.vertex[cycle_count].AorB = cB;
            Ichild.Mark_Edge_As_Seen(vs, Ichild.Get_vtl2(vs,B), B);
            previous = vs;
            vs = Ichild.Get_vtl2(vs,B);
        }
    }
}
else if (Ichild.Edge_Exists(vs, Ichild.Get_vt2(vs, B))
&& !Ichild.Edge_Can_Be_Seen(vs, Ichild.Get_vt2(vs, B))
&& Ichild.Get_vt2(vs, B) != previous)
{
    cycle_count++;
    temp_cycle.vertex[cycle_count].v = vs;
    temp_cycle.vertex[cycle_count].AorB = cB;
    Ichild.Mark_Edge_As_Seen(vs, Ichild.Get_vt2(vs, B), B);
    previous = vs;
    vs = Ichild.Get_vt2(vs, B);
}
else if (Ichild.Get_vtl(vs, A) == previous
|| Ichild.Get_vt2(vs, A) == previous
|| Ichild.Get_vtl(vs, B) == previous
|| Ichild.Get_vt2(vs, B) == previous)
{
    cycle_count++;
    temp_cycle.vertex[cycle_count].v = vs;
    if (temp_cycle.vertex[cycle_count-1].AorB == cA)
    {
        temp_cycle.vertex[cycle_count].AorB = cB;
        Ichild.Mark_Edge_As_Seen(vs, previous, B);
    }
    else
    {
        temp_cycle.vertex[cycle_count].AorB = cA;
        Ichild.Mark_Edge_As_Seen(vs, previous, A);
    }
    vs = previous;
}
else
{
    CString filename = CMutationDlg::stats_output_filename;
    ofstream outfile;
    outfile.open(filename, ios::app);
    outfile<<"\nError in EAX_Stage4_Create_Chi ld(...)\n";
    outfile<<"ladder logic failed.\n";
    outfile.close();
}

if(vs == start) //you have a cycle
{
    cycle_incomplete = false;
    Disjoint_Set[cycles_in_set] = new Cycle(array_size);
    //default size as required by sort routine.
Disjoint_Set[cycles_in_set]->length = cycle_count;
if(!Disjoint_Set[cycles_in_set])
{
    CString filename = CMutationDlg::stats_output_filename;
    ofstream outfile;
    outfile.open(filename, ios::app);
    outfile<<"* Memory Error_4* in fn:EAX_Stage4";
    outfile<<"_Create_Child(...), file ArrayGenome.cpp
";
    outfile.close();
}
temp_cycle.length = cycle_count;
//necessary because temp_cycle's length is maximum possible
*Disjoint_Set[cycles_in_set] = temp_cycle;
// necessary de-reference for deep assignment
cycles_in_set++;
cycle_count = 0;
}
//END while(cycleIncomplete)

cycleIncomplete = true;
vs = A[n][1];
start = vs;
if(upper_limitA == 0)
{
    all_vertexes_seen = true;
}
//END while(!all_vertexes_seen)
AssignCoordinatesToCycles(Disjoint_Set, cycles_in_set);
/*
  Sorting proves inefficient
  if(cycles_in_set > 1)
  {
    loBound = 0;
    hiBound = cycles_in_set-1;
    QuickSortDisjointSet(Disjoint_Set, loBound, hiBound, array_size, pivot, tempCycle);
  }
  */

// Step 2.) Combine Disjoint Cycles in Greedy Method
Child = *Disjoint_Set[0];
for(i=1; i<cycles_in_set; i++)
{
    Child = Child.Combine_Greedily(Disjoint_Set[i], Ichild.Get_Length());

    //What's happening? A cycle, Disjoint_Set[i], is being added
    //to the Child thus growing the child. Assignment necessary.
}
// delete [] A_p;
// delete [] A_n;
for(i=0;i<cycles_in_set;i++)
{
    delete Disjoint_Set[i];
}
}//end EAX_Stage4_Create_Child(...)

/************1************2************3************4************5************6************68

//
void CopyToSimpleContainer(Cycle* a certain_cycle, int cycles,
    EAXcycle& PossibleCycle, int cs, int vt,
    AorB_enum AorB)
// POST: A complex and not fully utilized structure is copied over
// to a simpler and efficient structure(memory fully utilized).
{
    int i;
    int len;
    int vs;
    len = PossibleCycle.Get Remaining();
a certain_cycle->length = len;
a certain_cycle->vertex[1].v = cs;
    // note that v is the source vertex of an edge
    a certain_cycle->vertex[1].AorB = (Cycle_AorB_enum)(AorB+1);
    vs = cs;
    for(i=2; i<=len; i++)
    {
        a certain_cycle->vertex[i].v = PossibleCycle.Get vt(vs, AorB);
        AorB = PossibleCycle.Swap_AorB(AorB);
        a certain_cycle->vertex[i].AorB = (Cycle_AorB_enum)(AorB+1);
        vs = a certain_cycle->vertex[i].v;
    }
}

/************1************2************3************4************5************6************68

// coordinates Struct* CTSPViewDlg::coordinates;//<---static member

void AssignCoordinatesToCycles(Cycle* some_cycle[], int cycles)
{
    int i, j, k;
    for(i=0;i<cycles;i++)
    {
        for(j=1; j<=some_cycle[i]->length;j++)
        {
            k = some_cycle[i]->vertex[j].v;
            some_cycle[i]->vertex[j].x = CTSPViewDlg::coordinates[k].x;
            some_cycle[i]->vertex[j].y = CTSPViewDlg::coordinates[k].y;
        }
    }

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
void Stem_Rebuilder(const int ss, const int cs,
int* A_n, int* A_p, int& upper_limitA,
int* B_n, int* B_p, int& upper_limitB,
EAXgraph& R_graph, AorB_enum AorB, EAXcycle& cycle)
{
    int vs = ss;;
    int vt;
    vt = cycle.Get_vt(ss,AorB);
    while(vs != cs)
    {
        if(AorB == A)
        {
            Available_Number_Array_Rebuilder(vs,vt,A_n,A_p, upper_limitA,R_graph,A);
        }
        else //AorB — B
        {
            Available_Number_Array_Rebuilder(vs,vt,B_n,B_p, upper_limitB,R_graph,B);
        }
        R_graph.Mark_Edge_As_NOT_Seen(vs,vt,AorB);
        vs = vt;
        AorB = cycle.Swap_AorB(AorB);
        vt = cycle.Get_vt(vs,AorB);
    }
}

void Available_Number_Array_Rebuilder(const int vs, const int vt,
int* X_n, int* X_p,
int& upper_limit,
EAXgraph& R_graph,
AorB_enum AorB)
{
    if(R_graph.Get_vtl_seen(vs,AorB) && R_graph.Get_vt2_seen(vs,AorB))
    {
        //the edge
        upper_limit++;
        X_p[vs] = upper_limit;  X_n[upper_limit] = vs;  }
    if( R_graph.Get_vtl1_seen(vt,AorB)  &&  R_graph.Get_vt2_seen(vt,AorB) )
    {
        //the edge's reflection...you get the reflection
        //because the parentage is the same.
upper_limit++;  
X_p[vt] = upper_limit;  
X_n[upper_limit] = vt; }

void Available_Number_ArrayReducer(const int vs, const int vt,  
int* X_n, int* X_p,  
int& upper_limit,  
EAQgraph& R_graph,  
AorB enum AorB)

// vs = the number being eliminated from the available series  
// X_n = Array of available numbers // X_p = Array of pointers to the numbers
{
  if(AorB == A)
  {
    if(R_graph.Get_vtl_seen(vs,A) && R_graph.Get_vt2_seen(vs,A))
    {
      X_n[X_p[vs]] = X_n[upper_limit];  
      X_p[X_n[upper_limit]] = X_p[vs];  
      upper_limit--;  
    }
    if(R_graph.Get_vt1_seen(vt,A) && R_graph.Get_vt2_seen(vt,A))
    {
      X_n[X_p[vt]] = X_n[upper_limit];  
      X_p[X_n[upper_limit]] = X_p[vt];  
      upper_limit--;  
    }
  }
  else // AorB == B
  {
    if(R_graph.Get_vtl_seen(vs,B) && R_graph.Get_vt2_seen(vs,B))
    {
      X_n[X_p[vs]] = X_n[upper_limit];  
      X_p[X_n[upper_limit]] = X_p[vs];  
      upper_limit--;  
    }
    if(R_graph.Get_vt1_seen(vt,B) && R_graph.Get_vt2_seen(vt,B))
    {
      X_n[X_p[vt]] = X_n[upper_limit];  
      X_p[X_n[upper_limit]] = X_p[vt];  
      upper_limit--;  
    }
  }
}
void Available_Number_Array_Reducer(const int vs, int* X_n, int* X_p,
   int& upper_limit)

// vs = the number being eliminated from the available series
// X_n = Array of available numbers // X_p = Array of pointers to the numbers
{
   X_n[vs] = X_n[upper_limit];
   X_p[vs] = X_p[upper_limit];
   upper_limit--;
}

// This function removes integers representing nodes from the
// two array structures that keeps track of what nodes are available
// to be chosen. This function is called when it is time to remove
// a cycle from an Rgraph. See function // EAX_Stage1_Create_ABcycles(...).
{
   int cs = cycle_source; // cycle source vertex
   int vs = cs;
   while( PossibleCycle.Get_vt(vs, AorB) != cs )
   {
      if(AorB == A)
      {
         Available_Number_Array_Reducer(vs, A_n, A_p, upper_limitA);
      }
      else // AorB == B
      {
         Available_Number_Array_Reducer(vs, B_n, B_p, upper_limitB);
      }
      vs = PossibleCycle.Get_vt(vs, AorB);
      AorB = PossibleCycle.Swap_AorB(AorB);
   }

   // one more for the road, since loop leaves one edge unaccounted for
   if(AorB == A)
   {
      Available_Number_Array_Reducer(vs, A_n, A_p, upper_limitA);
   }
   else // AorB == B
   {
      Available_Number_Array_Reducer(vs, B_n, B_p, upper_limitB);
   }
}
void QuickSortDisjointSet(Cycle* Disjoint_Set[], int loBound, int hiBound, int array_size, Cycle* pivot, Cycle* tempCycle)

// POST: The Disjoint_Set is sorted by number of edges in each cycle, shortest first.
// This function is Quick Sort keying on the lengths of cycles.
// Note that Cycle data type has no private members, so write access
// is available without use of member functions. pivot and tempCycle
// are instantiated in the calling function so as to avoid having to
// use deep copying here in this function.

// This function proved detrimental in actual runs.

{  
  int loSwap;
  int hiSwap;

  if (hiBound-loBound == 1) // Two items to sort
  {
    if (Disjoint_Set[loBound]->length > Disjoint_Set[hiBound]->length)
    {
      tempCycle = Disjoint_Set[loBound];
      Disjoint_Set[loBound] = Disjoint_Set[hiBound];
      Disjoint_Set[hiBound] = tempCycle;
    }
  }
  else // 3 or more items to sort
  {
    pivot = Disjoint_Set[(loBound+hiBound)/2];
    Disjoint_Set[(loBound+hiBound)/2] = Disjoint_Set[loBound];
    Disjoint_Set[loBound] = pivot;
    loSwap = loBound + 1;
    hiSwap = hiBound;
    do
    {
      while (loSwap <= hiSwap &&
             Disjoint_Set[loSwap]->length <= pivot->length)
      {
        loSwap++;
      }
      while (Disjoint_Set[hiSwap]->length > pivot->length)
      {
        hiSwap--;
      }
    }
if (loSwap < hiSwap)
{
    tempCycle = Disjoint_Set[loSwap];
    Disjoint_Set[loSwap] = Disjoint_Set[hiSwap];
    Disjoint_Set[hiSwap] = tempCycle;
}
} while (loSwap < hiSwap); /* end of do-while*/

Disjoint_Set[loBound] = Disjoint_Set[hiSwap];
Disjoint_Set[hiSwap] = pivot;
    // 2 or more items in 1st subarray
if (loBound < hiSwap-1)
{
    QuickSortDisjointSet(Disjoint_Set,loBound,hiSwap-1, array_size, pivot, tempCycle);
}
    // 2 or more items in 2nd subarray
if (hiSwap+1 < hiBound)
{
    QuickSortDisjointSet(Disjoint_Set, hiSwap+1, hiBound, array_size, pivot,tempCycle);
}

/******1********2***********3************4**********5********6*******68
/**************************** END OF arraygenome.cpp  6**************68

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
CHAPTER 6

METHODOLOGY AND DATA

My research focused upon the bare essentials of the algorithm. The EAX authors (Nagata and Kobayashi 1997) and another team (Watson et. al 1998) that implemented the EAX used heuristics to help produce impressively fast code. I tested the algorithm without the heuristics in its supposedly less efficient but fundamental form. I then added heuristics.

The four EAX configurations run include:

1. EAX with no sorting of the disjoint set and not using heuristic shortcut;
2. EAX with no sorting the disjoint set and using heuristic shortcut;
3. EAX with sorting the disjoint set and not using heuristic shortcut;
4.) EAX with sorting the disjoint set and using heuristic shortcut;

Various TSPs were run and the populations of the genetic algorithm varied.

Data assembled includes:

1. generations to convergence;
2. time elapsed to convergence;
3. generation at optimum;
4. convergence to runs ratio;

Time is recorded using MFC function difftime(...). All measurements are given in seconds. For each measurement given, twenty runs of the algorithm upon the listed TSP were executed with the exception of ATT532 which was run only five times on account of its size. Convergence is within ten percent of optimum (Reinelt 1995).
<table>
<thead>
<tr>
<th>TSP Optimum Generations Population</th>
<th>Average Generation at Convergence</th>
<th>Average Time at Convergence</th>
<th>Generation at Optimum and Success Ratio</th>
<th>Convergence to Runs Ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>ch150 6528 300 300</td>
<td>61.7</td>
<td>16.5</td>
<td>112</td>
<td>20/20</td>
</tr>
<tr>
<td>kroA200 29368 400 400</td>
<td>92.3</td>
<td>57.6</td>
<td>207</td>
<td>20/20</td>
</tr>
<tr>
<td>lin318 42029 636 636</td>
<td>202.0</td>
<td>400.8</td>
<td>not achieved</td>
<td>20/20</td>
</tr>
</tbody>
</table>

Figure 11: Results of No Sorting of the Disjoint Set, and Not Using the Shortcut Heuristic.

<table>
<thead>
<tr>
<th>TSP Optimum Generations Population</th>
<th>Average Generation at Convergence</th>
<th>Average Time at Convergence</th>
<th>Generation at Optimum and Success Ratio</th>
<th>Convergence to Runs Ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>ch150 6528 300 300</td>
<td>61.2</td>
<td>15.6</td>
<td>115</td>
<td>20/20</td>
</tr>
<tr>
<td>kroA200 29368 400 400</td>
<td>118.0</td>
<td>66.4</td>
<td>235</td>
<td>20/20</td>
</tr>
<tr>
<td>lin318 42029 636 636</td>
<td>252.4</td>
<td>410.8</td>
<td>not achieved</td>
<td>20/20</td>
</tr>
</tbody>
</table>

Figure 12: Results of No Sorting of the Disjoint Set, MaxM=100.
<table>
<thead>
<tr>
<th>TSP</th>
<th>Optimum Generations</th>
<th>Population</th>
<th>Average Generation at Convergence</th>
<th>Average Time at Convergence</th>
<th>Generation at Optimum and Success Ratio</th>
<th>Convergence to Runs Ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>ch150</td>
<td>6528</td>
<td>300</td>
<td>60.1</td>
<td>25.3</td>
<td>117</td>
<td>20/20</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>kroA200</td>
<td>29368</td>
<td>400</td>
<td>89.9</td>
<td>63.5</td>
<td>219</td>
<td>20/20</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>lin318</td>
<td>42029</td>
<td>636</td>
<td>200.8</td>
<td>402.0</td>
<td>not achieved</td>
<td>20/20</td>
</tr>
</tbody>
</table>

Figure 13: Results of Sorting the Disjoint Set, Not Using Heuristic Shortcuts in the Exhaustive, Greedy Combining of the Disjoint Set (no Max\_M).

<table>
<thead>
<tr>
<th>TSP</th>
<th>Optimum Generations</th>
<th>Population</th>
<th>Average Generation at Convergence</th>
<th>Average Time at Convergence</th>
<th>Generation at Optimum and Success Ratio</th>
<th>Convergence to Runs Ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>ch150</td>
<td>6528</td>
<td>300</td>
<td>60.6</td>
<td>13.8</td>
<td>not achieved</td>
<td>20/20</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>kroA200</td>
<td>29368</td>
<td>400</td>
<td>115.9</td>
<td>67.7</td>
<td>248</td>
<td>20/20</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>lin318</td>
<td>42029</td>
<td>636</td>
<td>252.0</td>
<td>416.5</td>
<td>not achieved</td>
<td>20/20</td>
</tr>
</tbody>
</table>

Figure 14: Results of Sorting the Disjoint Set, Max\_M=100.
<table>
<thead>
<tr>
<th>TSP Optimum Generations Population</th>
<th>Average Generation at Convergence</th>
<th>Average Time at Convergence</th>
<th>Generation at Optimum</th>
<th>Convergence to Runs Ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>ch150 6528 150 150</td>
<td>63</td>
<td>16.0</td>
<td>not achieved</td>
<td>20/20</td>
</tr>
<tr>
<td>ch150 6528 300 300</td>
<td>60.1</td>
<td>25.3</td>
<td>117</td>
<td>20/20</td>
</tr>
</tbody>
</table>

Figure 15: Results of Sorting and Using Heuristic Shortcuts in the Exhaustive, Greedy Combining of the Disjoint Set, ch150.

<table>
<thead>
<tr>
<th>TSP Optimum Generations Population</th>
<th>Average Generation at Convergence</th>
<th>Average Time at Convergence</th>
<th>Average Generation at Optimum</th>
<th>Convergence to Runs Ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>kroA200 29368 200 200</td>
<td>109.8</td>
<td>37.7</td>
<td>not achieved</td>
<td>0</td>
</tr>
<tr>
<td>kroA200 29368 400 400</td>
<td>115.9</td>
<td>67.7</td>
<td>248</td>
<td>4/20</td>
</tr>
</tbody>
</table>

Figure 16: Results of Sorting and Using Heuristic Shortcuts in the Exhaustive, Greedy Combining of the Disjoint Set, kroA200.

<table>
<thead>
<tr>
<th>TSP Optimum Generations Population</th>
<th>Average Generation at Convergence</th>
<th>Average Time at Convergence</th>
<th>Generation at Optimum Success Ratio</th>
<th>Convergence to Runs Ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>lin318 42029 318 318</td>
<td>286.8</td>
<td>232.1</td>
<td>not achieved</td>
<td>20/20</td>
</tr>
<tr>
<td>lin318 42029 636 636</td>
<td>252.0</td>
<td>416.5</td>
<td>not achieved</td>
<td>20/20</td>
</tr>
<tr>
<td>lin318 42029 1272 1272</td>
<td>251.8</td>
<td>815.8</td>
<td>not achieved</td>
<td>20/20</td>
</tr>
</tbody>
</table>

Figure 17: Results of Sorting and Using Heuristic Shortcuts in the Exhaustive, Greedy Combining of the Disjoint Set, lin318.
<table>
<thead>
<tr>
<th>TSP Optimum Generations Population</th>
<th>Average Generation at Convergence (final cost if not converged)</th>
<th>Average Time at Convergence (final time if not converged)</th>
<th>Generation at Optimum</th>
<th>Convergence to Runs Ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>ATT532 27686 532</td>
<td>not achieved (35255)</td>
<td>not achieved (1483)</td>
<td>not achieved</td>
<td>not achieved</td>
</tr>
<tr>
<td>ATT532 27686 1064</td>
<td>1048</td>
<td>8465</td>
<td>not achieved</td>
<td>3/5</td>
</tr>
<tr>
<td>ATT532 27686 2128</td>
<td>904</td>
<td>10482</td>
<td>not achieved</td>
<td>9/10</td>
</tr>
</tbody>
</table>

Figure 18: Benchmark ATT532 Test Results with Descending Sort and Shortcut Heuristic where Max_M = 100.

<table>
<thead>
<tr>
<th>TSP Optimum Generations Population</th>
<th>Average Generation at Convergence</th>
<th>Average Time at Convergence</th>
<th>Generation at Optimum</th>
<th>Convergence to Runs Ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>ATT532 27686 2128</td>
<td>969</td>
<td>15980</td>
<td>not achieved</td>
<td>0</td>
</tr>
</tbody>
</table>

Figure 19: Benchmark ATT532 Test Results with Descending Sort and No Shortcut Heuristic.
Figure 20: lin318 Converged.

Figure 21: kroA200 Converged.
Figure 22: AT&T532 After 2128 Generations and Max_M = 100. Converged at Generation 918, 10568 seconds (2hrs. 56min. 2GHz, 1G RAM Pentium)

Figure 23: AT&T532 After 2128 Generations and Max_M Not Used. Converged at Generation 958, 15702 seconds (4hrs. 21min. 2GHz, 1G RAM Pentium).
<table>
<thead>
<tr>
<th>Ordered Crossover</th>
<th>Average Generation at Convergence</th>
<th>Average Time at Convergence</th>
<th>Generation at Optimum</th>
<th>Convergence to Runs Ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>pop:2128 gens:5000</td>
<td>not achieved</td>
<td>not achieved</td>
<td>not achieved</td>
<td>not achieved</td>
</tr>
<tr>
<td>EAX</td>
<td>904</td>
<td>10482</td>
<td>not achieved</td>
<td>9/10</td>
</tr>
<tr>
<td>pop:2128 gens:2128</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Figure 24: Ordered Crossover versus EAX.**

Ordered Crossover settles into a local optimum early and never achieves convergence for ATT532.

Simulated Annealing is far faster in achieving a cost of twice optimum than EAX, but then settles into a local minimum that is very slowly reduced. Through ten tests the crossover did not achieve convergence.
CHAPTER 7

CONCLUSIONS AND RECOMMENDATIONS

Compared to the efforts of the two other teams of EAX implementors mentioned earlier, my own efforts resulted in an algorithm that converges consistently but struggles to attain optimum. Empirical proof has been given that the Edge Assembly Crossover achieves a high degree of success if measured by convergence alone.

The effects of the heuristic to stitch the disjoint set into a tour, which is meant to shortcut the n squared function works well through convergence and reduces run time by one third. However, beyond convergence, the algorithm crawls toward optimum, which, not surprisingly, is not often attained. At the point of convergence it would be wise to shut the short cut heuristic off, since it proves to increase run times once diversity wains, but this would constitute a prior knowledge situation and is not in the spirit of fairly approaching an NP Complete problem. The desire was to examine the EAX in its most basic form. Since the gain versus diversity heuristic of Nagata and Kobayashi was not implemented, I used higher population settings to maintain diversity. This works, but the success that the Japanese duo attained in reaching optimum runs is not seen from my implementation.

Speed is desirable. I suggest a student capitalizing on my efforts implement the Nagata and Kobayashi gain versus diversity heuristic. This heuristic may not only speed things up, but should also give the reported higher rate of achieving optimum.

Certainly, a look at streamlining my code and logic for the sake of efficiency is desired. The EAXcycle class could be done away with altogether. The sequential form
of a tour inherent to GAlib should not be translated more than at the beginning and the end of the driver, and best speeds for my algorithm will be achieved by defining the EAXgraph type as a genome type, given this extension of GAlib is possible.

The question of why the Edge Assembly Crossover works as well as it does is not satisfied through this study. After this work with the Edge Assembly Crossover I began imagining that tours of any TSP problem can be mapped onto circles such that the edge weights remain the same, I began to wonder if the edge weights were all averaged and the designs of two degree geometric graphs were drawn, if the drawings would exhibit enumerable patterns, that is groups. I have begun to look at theories of combinatorial geometry from the past century.

I offer what follows as conjecture for a path of study that I have begun without the benefit of knowing what has come before me. Combinatorial Enumeration of Groups, Graphs, and Chemical Compounds (tr. of Polya 1987) was recommended to me by a combinatorial mathematician in regards to the thoughts that follow, and thus far I have seen only tidbits of Polya's work in other books. However, I warn the reader that the constructions to follow may not hold up under scrutiny and likely will lead to a dead end. None the less, the EAX is offered in the context of a heuristic, which in Webster's means something like, "prove it to yourself." So I have begun to give my imagination free reign in this regard.

First, allow that the definitions I construct are my own and may be at odds with terminology traditionally used in graph theory. I begin by imagining that any TSP tour can be unfolded and mapped onto a circle of some correct radius and that the optimum tour will coincide with the smallest circle of such a set. Not every tour will require a different circle, as there are graph groups to be identified within any TSP, the simplest of which include the different sequences generated by merely shifting the starting vertex.
I start with a graph of six nodes equally spaced on a circle and I plot a rather subjective set of drawings. But an objective, enumerative examination gives me surprising results, even while expecting this diminishment of the TSP to be a ruinous oversimplification. I can form a table of the predicted distinct graphs of any number of vertices. I describe a distinct graph as one in which no rotation or mirror image of the graph is allowed to distinguish separate configurations.

Figure 26: The Distinct Graphs of $n = 6$
The subscripts in the 12 drawings denote the number of rotations within each distinct graph that give separate tours numerically, without the use of shifting. This number can be described by a recurrence relation, the function $f(n)$, below. These are the tour groups which represent the same paths with merely 6 different starting points. The total number of tour groups for a 6 node graph is 60. There are 6 different starting points in each of the 60 groups, giving 360 sequences. And the sequences can be expressed backwards, so double the number to 720, giving the $n!$ number of sequences total.

![Figure 27: Demographics for n-gons.](image)

<table>
<thead>
<tr>
<th>nodes $n$</th>
<th>distinct graphs</th>
<th>$g(n)$</th>
<th>tour groups $f(n)$</th>
<th>$n!$</th>
<th>edges in a complete graph $\binom{n}{2}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>1</td>
<td>3</td>
<td>1</td>
<td>6</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>6</td>
<td>3</td>
<td>24</td>
<td>6</td>
</tr>
<tr>
<td>5</td>
<td>4</td>
<td>15</td>
<td>12</td>
<td>120</td>
<td>10</td>
</tr>
<tr>
<td>6</td>
<td>12</td>
<td>30</td>
<td>60</td>
<td>720</td>
<td>15</td>
</tr>
<tr>
<td>7</td>
<td>28</td>
<td>90</td>
<td>360</td>
<td>5040</td>
<td>21</td>
</tr>
<tr>
<td>8</td>
<td>56</td>
<td>360</td>
<td>2520</td>
<td>40320</td>
<td>28</td>
</tr>
<tr>
<td>9</td>
<td>104</td>
<td>2700</td>
<td>20160</td>
<td>362880</td>
<td>36</td>
</tr>
</tbody>
</table>

$g(n) = 2 \cdot g(n-2) \cdot g(n-4)$ \quad f(n) = f(n-1) \cdot (n-1) = (n-1)!! / 2 \quad \text{distinct graphs} = \frac{f(n)}{g(n)}$

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
Informal proof of the tabulation of Figure 25: The number of distinct graphs is known through 6 nodes. Tour groups are known through 6 nodes. By induction $f(n)$ is produced for all $n$. $g(n)$ is then calculated thru 6 nodes and a resulting recurrence relation is established. $g(n)$ is proven by induction for all $n$. Then by division the number of distinct graphs is known for all $n$. I cannot express $g(n)$ as a function independent of previous function values, though others may have already conquered this problem. The expansion $g(n)$ involves binomial coefficients as powers of functions of $g$, very messy, leaving me looking for help, yet this is only an inconvenience and not a dead end, since a program can be written to calculate the $g(n)$ for any $n$ based on the recursion $g(n) = 2g(n-2)g(n-4)$. Thus the number of distinct graphs is known for any $n$.

The questions are: "How fast does the number of distinct graphs grow as $n$ increases as compared to the $n!$ number of tours?" "Can any tour of $n$ nodes be mapped onto a graph of an $n$-gon upon the radius of a circle with meaningful correlation?" Each distinct graph represents a distinct cost, and a class of costs. If the edges of an inscribed $n$-gon were not of equal length, the problem becomes one of dealing with $f(n)$ similarity classes, which might deliver this process to where it began, at a factorial input set. So, "Can the TSP be related to the non-exponential enumeration of distinct graph patterns?" If so, this could be an aid in analyzing the probabilities involved in finding cost similarity classes in TSP. These intuitive ideas express my thoughts that the EAX is tapping into the patterns of graphs through cycle constructions. The algorithm is not manipulating symbols in a sequence as in simpler genetic algorithm crossovers. "Is it possible to show the EAX crossover algorithm is able to jump from one graph group to another of decreasing cost?"
REFERENCE LIST

Books


Articles


Papers


VITA
Graduate College
University of Nevada, Las Vegas
Dwain Seppala

Local Address:
4101 Roxanne Drive
Las Vegas, Nevada 89108

Degrees:
Bachelor of Science, Mathematics, 1987
University of Illinois at Chicago, Illinois

Bachelor of Arts, Fine Art, 1987
Drake University, Des Moines, Iowa

Thesis Title:
Genetic Algorithms for the Traveling Salesman Problem Using Edge Assembly Crossovers

Thesis Examination Committee:
Chairperson, Dr. Wolfgang Bein, Ph. D.
Committee Member, Dr. Lawrence Larmore, Ph. D.
Committee Member, Dr. John Minor, Ph. D.
Graduate Faculty Representative, Dr. Zhiyong Wang, Ph. D.