DISSERTATION

THE MIXING GENETIC ALGORITHM FOR TRAVELING SALESMAN PROBLEM

Submitted by

Swetha Varadarajan

Department of Computer Science

In partial fulfillment of the requirements

For the Degree of Doctor of Philosophy

Colorado State University

Fort Collins, Colorado

Spring 2022

Doctoral Committee:

Advisor: Darrell Whitley

Wim Böhm (deceased)
Ross Beveridge
Louis-Noël Pouchet
Edwin Chong

Copyright by Swetha Varadarajan 2022

All Rights Reserved

ABSTRACT

THE MIXING GENETIC ALGORITHM FOR TRAVELING SALESMAN PROBLEM

The Traveling Salesman Problem (TSP) is one of the most intensively studied NP-Hard problems. The TSP solvers are well-suited for multi-core CPU-based architectures. With the decline in Moore's law, there is an increasing need to port the codes to parallel architectures such as the GPU massively. This thesis focuses on the Genetic Algorithm (GA) based TSP solvers.

The major drawback in porting the state-of-the-art GA based TSP solver (called the Edge Assembly Crossover (EAX)) are (a) the memory per crossover operation is large and limits the scalability of the solver (b) the communication per crossover operation is random and not favorable for the SIMD machines. We designed a new solver, the Mixing Genetic Algorithm (MGA), using the Generalized Partition Crossover (GPX) operator to overcome these aspects. The GPX consumes $4\times$ lesser memory and does not access the memory during crossover operation.

The MGA is used in three different modes. (1) MGA can converge fast on problems smaller than 2,000 cities as a single solver. (2) As a hybrid solver, together with EAX, it speeds up the convergence rate for problems up to 85,900 cities. (3) In an ensemble setting, together with EAX and an iterated local search (called the Lin-Kernighan Helsgaun (LKH) heuristic), it increases the success rate of some of the hard TSP instances.

The MGA is parallelized on shared memory (using OpenMP), distributed memory (using MPI), and GPU (using CUDA). A combination of OpenMP and MPI parallelization is examined on problems ranging between 5,000 to 85,900 cities. We show near-linear speedup (proportional to the number of parallel units) on these instances. Preliminary results on GPU parallelization of the GPX crossover operator partition phase show a 48x to 625x speedup over the naive sequential implementation. This is the first step towards the fine-grain parallelization of GA operators for TSP. The results are tested on problems ranging from 10,000 to 2M cities.

ACKNOWLEDGEMENTS

First and foremost, I thank Dr. Blanche Hughes, Vice President for Student Affairs, Colorado State University (CSU). She is my inspiration to believe in myself. I read an article about her talk in a college newspaper. I was inspired by her strength and courage and emailed her but didn't expect her to respond. However, she replied. Her unwavering enthusiasm, support, and encouragement have been my vital motivating factors to stay in the Ph.D. program.

Next, I thank Prof. Louis Noel Pouchet for his selfless advice on "how to give a talk". I was surprised by the appreciation from several professors for my talks. I thank Prof. Sanjay Rajopadhye, my MS thesis advisor, whose "high expectations" and rigorous training via weekly meetings helped strengthen critical thinking skills. I thank all the "Melange" group members - Yun Zou, Waruna Ranasinghe, Tarequl Islam, Jana Sharma, Steve Kommrusch, Revathy Rajasree, Nirmal Prajapati, Guillaume Iooss, Tomofumi Yuki, Mugdha Puranik, Raj Barath, Prerana Ghalsasi, Corentin Ferry.

I thank Prof. Darrell Whitley, my Ph.D. advisor, for his unwavering dedication to the field. His in-depth knowledge, sincere feedback, and finding value from every result is contagious. I also thank him for asking leading questions about my ideas, leading to interesting results. I learned the "art of publishing papers" from him. I always enjoyed having good food at his house, cooked by his wife, Dr. Beth Thurston. I always find her adding zucchini in the salad to be refreshing.

Darrell provided me with the opportunity to work with Joseph Bates. Dr. Bates is one of the greatest minds I have met so far. I learned a lot in the two-month internship. Darrell also provided me with the opportunity to attend weekly research meetings at the MIT-CSAIL ALFA group led by Dr. Una-May O'Reilly. It was a wonderful experience exchanging ideas and presenting my project with MIT students. I am grateful to Dr. O'Reilly for her positive encouragement every time I meet with her. I am also thankful for the opportunity to work with Dr. Gabriella Ochoa. Finally, I thank Darrell and GECCO (Genetic and Evolutionary Computation Conference) community for the opportunity to co-organize the women@GECCO workshop. This opportunity helped me understand my privileges and be a good "world citizen" by contributing to the community.

I thank Prof. Wim Bohm, who made meticulous efforts in providing valuable feedback on my documents. I am also honored to receive my thesis proposal's Wim Bohm and partners Ph.D. award. It helped in completing my thesis on time. I thank Prof. Bruce Draper and Prof. Ross Beveridge for providing me with the opportunity to do the CS793 research seminar with their group. I thank my committee member Prof. Edwin Chong, for his time and patience in answering my questions. His dedication and authority in the optimization field are commendable.

I thank several people with whom I worked during this journey. They inspired and lifted my spirits a lot. Wendy Stevenson, Brooke Wichmann, Bridgette Johnson, Melissa Edwards, Joann Cornell, Laurel Bond, Colleen Webb, Lumina Albert, Kristina Quinn, Ramadan Abdunabi, Zaria Vick, Amanda Koch, Lindsay Winkenbach, Sagar Rathod, Eyal Marder, Nancy Frimpong, Stephen, John, Audrey, Casey, Karl, Andrea, Juliet, Balbino, Anthony King, Mandana Ashouri, Penny Gonzales, Rae Hirsch, Anirban Mukhopadhyay, Jason Yu, Hieu Bui, Dhruva Patil, Kartikay Sharma, Lisa Pappas, Sorna Momtaz, Rawn Henry, Ali Ebrahimpour, Sachini Weerawadhana.

I thank my friends Karthik Ganesan, Sri Gautham, Sandhya Kannan, Dr. Haritha Katragadda, Naveen Prasad, Abitha Venkat, Vaishnavi Deivanai, Anusha Subramanyam, Dr. Vishnu Shankar Hariharan, Bhargav Narayana, Bhavani Singh, Jaisre Rao, Kavitha Chandrasekaran, Ben Scott, Michelle Dixon, Rizwan Syed, Santosh Vangaveti. My mom Sugantha, dad Varadarajan and grandma Vasantha for their consistent positivity and faith in me. Finally, my brother, Krishna, truly deserves a Ph.D. for his dedication and hard work in the field of computer vision and AI.

DEDICATION

To my parents R. Varadarajan, S. Sugantha, my grandma S. Vasantha and, my brother Sri Krishna

TABLE OF CONTENTS

ABSTRACT ii
ACKNOWLEDGEMENTS iii
DEDICATION v
LIST OF TABLES x
LIST OF FIGURES xii

Chapter 1 Introduction 1
1.1 Single best solver ? 2
1.2 Computer Architecture Paradigms 5
1.3 Motivation and Challenges 6
1.4 Research Questions 8
1.5 Organization and Contributions 8

Chapter 2 Literature Review 11
2.1 Lower Bounds 11
2.1.1 Minimum Spanning 1-tree 12
2.1.2 Held-Karp bound 13
2.2 Tour construction 14
2.3 Stochastic Local search 15
2.3.1 2-opt 15
2.3.2 3-opt 17
2.3.3 K-opt 18
2.4 Exact Solvers 19
2.4.1 Brute-force Methods 19
2.4.2 Dynamic Programming (DP) 19
2.4.3 Linear Programming 20
2.4.4 Branch and bound algorithms 21
2.4.5 Concorde 21
2.5 Inexact Solvers 22
2.5.1 Multi Agent Optimization Systems (MAOS) 22
2.5.2 Lin-Kernighan-Helsgaun (LKH) 23
2.6 Genetic Algorithm Solvers 25
2.6.1 GA solver terminologies 25
2.6.2 Edge-Assembly Crossover (EAX) 27
2.6.3 Generalized Partition Crossover (GPX) 29
2.6.4 Iterative Partial Transcription (IPT) 34
2.7 Discussion 35
2.7.1 Parallel TSP solvers 35
2.7.2 Parallel GA solvers 37
2.7.3 Design of massively parallel GA based TSP solver 39

Chapter 3 Mixing Genetic Algorithm . . . 41
3.1 Generalized Partition Crossover . . . 42
3.1.1 Communication costs . . . 43
3.1.2 Memory costs . . . 43
3.1.3 Premature Convergence . . . 43
3.2 The Mixing Population . . . 43
3.2.1 Genetic invariance . . . 46
3.2.2 The Mixing Theorem . . . 48
3.3 The Mixing Genetic Algorithm . . . 48
3.3.1 Limitation of MGA . . . 49
3.3.2 On Convergence and improving moves . . . 51
3.3.3 Termination condition . . . 52
3.3.4 Parallelization . . . 53
3.3.5 Summary . . . 55
3.4 Experimental setup and results . . . 55
3.4.1 Initial population . . . 57
3.4.2 Population size scalability . . . 57
3.4.3 Problem size scalability . . . 58
3.4.4 Recombinations to global optima . . . 58
3.4.5 Success Rate . . . 58
3.4.6 Execution time . . . 59
3.5 Conclusion . . . 61

Chapter 4 On Runtime behaviour and Convergence . . . 62
4.1 Background . . . 63
4.2 Micro-level analysis . . . 69
4.2.1 Edge Frequency Distribution . . . 69
4.2.2 Multiple Global optima . . . 73
4.2.3 Unknown Global optima . . . 73
4.2.4 Visualization methods . . . 73
4.2.5 Summary . . . 76
4.3 Hybrid algorithms . . . 78
4.3.1 Experimental setup . . . 80
4.3.2 Recombination, Generation, Success Rate . . . 80
4.3.3 Final Populations Edge Frequencies . . . 81
4.3.4 Multiple Global Optima and Frequencies . . . 84
4.3.5 Final population visualization . . . 86
4.3.6 Execution time . . . 88
4.4 Conclusions . . . 89

Chapter 5 Distributed memory parallelization of MGA . . . 91
5.1 Background . . . 93
5.1.1 Island model topologies . . . 95
5.1.2 Island model migration policies . . . 96
5.1.3 Other parameters . . . 96

5.2 Island Model Mixing GA . . . 97
5.2.1 Mixing island sub-populations . . . 97
5.2.2 Replacement policy . . . 99
5.2.3 Ring Topology . . . 100
5.2.4 One batch of island model MGA . . . 101
5.2.5 Experiments on small problem sizes . . . 102
5.3 Hybrid algorithms . . . 107
5.3.1 Experimental setup . . . 108
5.3.2 Choice of batch size (x) . . . 109
5.3.3 Results on hybrid algorithms . . . 110
5.4 A Parallel Ensemble of Solvers . . . 116
5.4.1 Ensemble setup . . . 116
5.4.2 Comparisons with EAX . . . 118
5.4.3 Comparisons with LKH . . . 118
5.4.4 GA Ensemble . . . 121
5.4.5 Full Ensemble . . . 123
5.5 Conclusions . . . 125

Chapter 6 Preliminary findings on operator-level parallelization . . . 127
6.1 Background . . . 129
6.2 Parallelization of the partition phase . . . 131
6.2.1 Union of two-parent tours . . . 131
6.2.2 Splitting nodes of degree four . . . 133
6.2.3 Identifying recombining components . . . 140
6.2.4 Integrating with the recombination phase . . . 144
6.2.5 Integrating with MGA . . . 145
6.3 Experimental setup . . . 146
6.4 Results . . . 147
6.4.1 Execution time speedup . . . 150
6.4.2 On partitions, fusions and performance . . . 151
6.4.3 Memory savings . . . 151
6.5 Conclusions . . . 152

Chapter 7 Future work . . . 153
7.1 Initialization techniques . . . 153
7.2 Convergence techniques . . . 154
7.3 Execution time improvements . . . 154
7.4 Performance and usefulness of the solver . . . 155
7.5 Generalization and beyond TSP . . . 155

Chapter 8 Conclusion . . . 156

Bibliography . . . 159

Appendix A Case Study on Singular Computing Machine 174
A.1 Challenge 1: Memory costs 174
A.2 Challenge 2: Communication costs 175
A.3 Challenge 3: Accuracy of Solution 178

Appendix B Dataset and Codes 179
B.1 TSP File format and calculations 179
B.2 Dataset and references 179
B.3 Software codes 182

LIST OF TABLES

2.1 List of state-of-the-art TSP solver and references to parallel implementations. 35
2.2 List of GA-based TSP solver and their properties 38

3.1 List of problem instances and the corresponding smallest population size that has all the edges found in P* after performing EAX based 2-opt on the initial population. . . . 50
3.2 Percentage of improving moves found by running the MGA on a population of size 1024. 52
3.3 Design parameters of Mixing Genetic Algorithm with openMP parallelization 55
3.4 Results on applying MGA to 25 problem instances to two population sizes (1024, 16,384). A population of 16,384 on MGA is comparable to EAX on a population of 300 with 30 children per recombination. Note that for fl1577, the minimum population required to store all P* edges is 4096. Therefore the results reported under MGA-1024 column is for a population of 4096. 56

4.1 Rank of instances sorted by the percentage of globally optimal edges found in the initial population. Instance u2319 had the fewest edges; instance rl1304 had the most edges from the global optimum. For 20 of the 25 instances, more than 71% of the edges in the population come from the global optimum. This percentage is almost constant with increasing population size 70
4.2 Number of missing global optima edges 71
4.3 Results on applying MGA, MGA+EAX, EAX-default and EAX-optimal to 25 problem instances. *The results shown for fl1577 under last 3 columns are on a population of size 4096. Here S.R means success rate. Each instance is run for 30 times. 79
4.4 Frequency of global edges in the final population, sorted by EAX results. The particular global optima were obtained using the LKH algorithm. For over half the instances, edges in the global optimum make up over 98% of all edges in the EAX population. This is good if EAX converges, but also can lead to premature convergence. MGA+EAX is only slightly more converged than MGA. The pop. size is 300. 84
4.5 Summary of mixed success rates and behaviors of the three algorithms on 8 instances . 86

5.1 Results on applying parallel MGA versions. Results are compared with sequential MGA and EAX. MGA population size = 16384, No. of islands = 64. The EAX population size is 300, No. of children = 30. All algorithms achieve 100% success. (Exception: The Success rate of EAX on fl1577 is 3/30) 106
5.2 Sequential MGA+EAX (hybrid-1) x=2 111
5.3 Sequential MGA+EAX (hybrid-1) x=5 111
5.4 MGA+EAX - Ring topology 112
5.5 MGA+EAX - Hypercube topology 113
5.6 Levels of parallelism 117
5.7 EAX versus MGA+EAX Ensemble 122
5.8 Results on ensemble setup. S.R refers to Success Rate. Time is in seconds. The average success rates at the last row is out of 30 trials. 124

6.1 Example edge table of a 10-city problem 131
6.2 Look up table. "X" denotes the dont't care condition. 139
6.3 Example edge-table for a 10-city problem after splitting degree 4 vertices and deleting the common edges. - Forward direction 140
6.4 Example edge-table for a 10-city problem after splitting degree 4 vertices and deleting the common edges. - Reverse direction 141
6.5 Results on applying the naive CPU-based GPX operation without fusions 147
6.6 Results on applying the GPU-partition phase of GPX operation without fusions 147
6.7 Results on applying the naive CPU-based GPX operation with fusions 148
6.8 Results on applying the GPU-partition phase of GPX operation with fusions 148

LIST OF FIGURES

1.1 Graph showing the scalability of TSP solvers. The years correspond to when an optimal solution was found for the particular problem size. The 1.9M, 10M and 1.3B problems are not solved to optimality. 2

1.2 Diagram showing a typical Genetic Algorithm (GA) and operations. A GA starts with an initial population (P). Every individual has a chromosome representation. For Traveling Salesman Problem (TSP), the chromosome is a real-valued vector representing the tour permutation of the N cities. Every chromosome has an *evaluation function* associated with it. For TSP, the *evaluation function* is the length of the tour, a simple addition of the distances between the cities. Genetic operators such as the *selection*, *mutation* and *crossover* are applied to *minimize* the length of the tour. The population keeps *evolving* generation after generation until the desired tour length is obtained. In this example, the population P' is obtained by applying the genetic *mutation* operation on P. Population P" is obtained by applying the genetic *crossover* operation on P'. The *selection* operator decides which chromosome will take part in the evolutionary process. In this example, all chromosomes are selected. However, the partners for crossover are randomly selected. Consider the two chromosomes C1 and C2. The mutation operation changes the tour permutation. However, the *crossover* operation exchanges parts of the chromosome between the two individuals. Note that the resulting chromosome (also called the *offspring*) must be *valid*. For TSP, a chromosome is valid if it contains all the city numbers between 1 and N exactly once. 3

1.3 The *many-core* CPU architecture has specialized cache and control logic units. The majority of hardware units in *many-thread* GPU architecture is devoted to the data processing cores. 5

2.1 Diagrams illutrating the Minimum spanning 1-tree. The left-most graph G is connected except for the vertex v. The second graph is the spanning tree of G with a cost = 13. The third graph is the minimum spanning tree of G with cost = 7. The final, right-most graph is the Minimum Spanning 1-tree, where the vertex v is now connected to the MST using the two of its short edges, represented in red color. 12

2.2 Figure showing a 2-opt move on the original tour to the left. After removing the edges AB and CD, edges AC and BD are added and BC segment is reversed. The tour to the right has a shorter length. 15

2.3 Figure showing 3-opt moves. The prime symbol means that the segment is reversed. There edges (X12, Y1Y2, Z1Z2) are modified to get a shorter tour. There are 7 possible ways of modifying these 3 edges. Case 1, 2, 3 is equivalent to performing one 2opt operation. Case 4,5,6 is equivalent to performing two subsequent 2opt operations. Casr 7 is equivalent to performing three subsequent 2opt operation. 17

2.4 Figure illustrating a cutting plane. 20

2.5 Figure showing the use of Iterative Local Search in the LKH algorithm. LKH uses a combination of "soft" and "hard" restarts in combination with efficient and selective k-opt local search. In this figure, a sequence such as A0, B0, C0, D0 represents a sequence of local optima reached using soft restarts. Search is started, then stagnates at solution E0. Then the current solution is perturbed using some random partial alternation of the solution; the soft restart is designed to move the search to a different local optima. A sequence such as A1, B1, C1 represents a different sequence after a full "hard" restart from a different random initial solution. Thus, in this figure we see 10 "hard" restarts, each which has 5 "soft" restarts. [1] 23
2.6 These plots show EAX crossover. The graph union of parent 1 and parent 2 yields the AB cycles. The subcycles are formed from parent 1 and E-set. The parent 1 edges in E-set are deleted from the original parent 1. The parent 2 edges in E-set are added to the original parent 1. 27
2.7 This figure illustrates a simple example of GPX. All of the vertices are of degree 3 and common edges are deleted: this guarantees the graph composes into AB-cycles. An AB-cycle is also a recombining component if the tours enter and exit the AB-cycle at the same vertices. For every recombining component, GPX calculates which parent has the shortest partial tour visiting all the vertices in the component. 30
2.8 A more complex example of GPX. The boxes labeled from A to I are candidate components corresponding to AB-cycles. We can automatically identify components A, I and B as recombining components, since the AB-cycles contain only two portals (which function as an entry and an exit). After AB-cycle B is removed, AB-cycle C can now be recognized as a recombining component. AB-cycle D and E are both recognized as a recombining components because the solid tour and the dashed tour enter and exit at the same portals. The AB-cycles labeled F, G and H are not recombining components, but all three AB-cycles can be fused to create a single recombining component. 32
2.9 Figure showing an example of the IPT crossover operation. 34

3.1 Simple GPX with one crossover opportunity and two recombining components. One parent is represented by the solid (red) edges. The other parent is represented by the dashed (blue) edges. If Child 1 represents the best possible offspring, then Child 2 must be the worst possible offspring generated by GPX. 42
3.2 For a population size 16, there are four generations in one epoch of the Mixing Genetic Algorithm. In generation one, parents from P0 to P7 are recombined with parents P8 to P15 to produce the best offspring (P0' to P7') and worst offspring (P8' to P15'). The pattern is color-coded where blue is the first parent and red is the second parent. The crossover arrows are shown for some recombinations. At the end of an epoch, P0 is usually the best solution in the population, and P15 is usually the worst solution. . . 45
3.3 The bars show the fitness value of 16 individuals at the end of first and last generations of an epoch. It can be seen that the first individual has the best and the last individual has the worst fitness value. 47

3.4 Results on running att532 on a population of size 16. The bars show the difference in fitness value of first and last generations of an epoch. Adding all the values produces **zero**. This shows that at the end of an epoch, the tour evaluations are different from that of first generation. However, the total evaluation of the popualtion remains constant, thereby achieving genetic invariance 47
3.5 MGA does not produce an improving move after continuously applying four or five epochs without any randomization of the population. 51
3.6 The Mixing Genetic Algorithm – Parallel Loops 53
3.7 Figure illusrtating an example of a shared memory architecture. 54
3.8 Percentage improvement of Mixing GA over EAX in terms of median recombinations for instances where both EAX and Mixing GA achieve 100% success 59
3.9 Comparison of execution times of EAX and MGA 60

4.1 Figure showing the first 200 generations of running MGA and EAX. EAX overtakes the MGA trace between 25th and 175th generations. MGA converges faster on higher population sizes. 63
4.2 A classic example of the "Big Valley" distribution. Each circle is a randomly generated local optima under 2-opt for TSP instance ATT532. The x-axis is the objective evaluation $f(x)$, and the y-axis is the number of edges in each local optima that are **not** shared with the global optima. [2] 66
4.3 Figure illustrating a two funnel landscape [3]. 67
4.4 3D Visualization of att532 funnels [3]. The red funnel shows the presence of global optima. The second funnel is colored yellow to indicate that the cost of the local optima is higher than those in the red funnel. for att32, the two funnels are over-lapping. . . . 68
4.5 Plots showing the frequency of each global and non-global edges in the population after applying the 2opt local search. For instance, the graph to the right is the frequency of global optima edges. The graph to the left is the frequency of non-global optima edges. 72
4.6 Figures showing heatmaps of the intial population (size=512) of att532. The heatmap has 512 rows and 532 columns. Each row represents a tour. Black color represents the non-global edges. The non-black edges represent the global edges. 74
4.7 Figures showing heatmaps of the intial population (size=512) of att532. The heatmap has 512 rows and 532 columns. Each row represents a tour sorted according to the edge number. The black color represents the non-global edges and have a number 0. The non-black edges represent the global edges and are numbered from 1 to 532. Hence, the gradient in color from red to white. 75
4.8 The heatmaps on EAX population for every 10 generation on att532 with a population of size 512. Two individuals reach global optima at the 60th generation. 76
4.9 The heatmaps on MGA population for every other epoch on att532 with a population of size 512. One epoch consists of 9 generations (G). 77
4.10 Progress of EAX and MGA and the hybrid MGA+EAX on C3k.0 for popsize=300. . . 82
4.11 Progress of EAX and MGA and the hybrid MGA+EAX on C3k.1 for popsize=300. . . 82
4.12 Progress of EAX and MGA and the hybrid MGA+EAX on pcb3038 for popsize=300. 83
4.13 Progress of EAX and MGA and the hybrid MGA+EAX on fl1400 for popsize=300. . . 83

4.14 Overlap percentage of the two global optima edges for a population of 300 on (a) att532 (the top venn diagram) and (b) u2319 (the botton venn diagram) instances . . . 85
4.15 The heatmaps on MGA+EAX population on att532 with a population of size 512. Here, one epoch of MGA (9 generations) is followed by one generation of EAX. 87

5.1 In a distributed memory system, the parallel units do not share memory. An interconnect system is used for communication. Each parallel unit has its own memory and input/output system. 92
5.2 Figure showing examples of two Parallel GA models [4] . (a) Master-slave model: The master node contains the population. It sends an individual to the slave node to perform a genetic operation. The slave node performs the genetic operation and sends the individual back to the master node. The slave nodes do not communicate with each other. (b) Island Model: The population is divided amongst the four islands (CPUs). The islands communicate with each other by migrating the individuals in the population. 93
5.3 Figure showing examples of different island model topologies [4]. Top row (left to right): chain, ring. Middle row (left to right): toroid, cartwheel, mesh. Bottom row (left to right): hypercube, star, fuly-connected 95
5.4 Hypercube topology - Migrate policy 98
5.5 Hypercube Topology - Replace policy 99
5.6 Ring topology - Migrate policy 100
5.7 Ring Topology - Replace policy 100
5.8 Plots showing performance of att532 on various parallelization techniques. The base of comparison is the MGA sequential code. The population size is fixed to 16384. The top graph shows the linear time speedup of 2opt and parallel MGA algorithms over the sequential version. Note that the y-axis is in log scale. The middle graph shows that the convergence rate in terms of number of generations to global optima. Note that both the replace strategy converges at a faster rate with increase in island . The bottom graph shows that success rates of all parallel strategies is 100% for upto 256 islands. . . 104
5.9 Plots showing the time speedup of the island model versions over the sequential code. . 107
5.10 Plots showing a single run of att532 on all the four hybrid (MGA+EAX) island model versions, sequential EAX and sequential MGA. The MGA traces are quicker in the initial stages because of the GPX operator. EAX cuts the traces between 40 and 60 generations. 110
5.11 Figure showing the nested ensemble layers. The inner layer consists of three ensemble (a) all four Hybrid MGA (b) The four hybrid MGA+EAX algorithms. (c) LKH and EAX. The outer layer consists of all six inexact solvers, also called the Full ensemble. . 116
5.12 Speedup of the hybrid versions over EAX with respect to (a) generations (b) time . . . 119
5.13 The speedup of (a) Hybrid MGA+EAX ensemble over sequential LKH and EAX. The hybrid MGA+EAX is parallelized to run within a single-multicore and across multiple CPUs. (b) Full ensemble over the two sequential algorithms and two ensemble setups. . 120

6.1 Partition Phase of GPX operator finds the union of the two parent tours, splits the degree 4 vertices, deletes the common edges to form the two candidate AB cycles. The path of the two tours inside each component can be replaced by a single red and blue edge. 127
6.2 Figure showing the time profiling of the GPX operator. The DR1+HYG and GaiaDR2 instances are 1M and 2M city problems, respectively. For problems smaller than 160,000 recombination phases is dominating. For problems higher than 160,000, the parititon phase is dominating. 128
6.3 Plot showing the speedup of GPX crossover on the GPU with respect to the CPU code. 129
6.4 Figure showing the thread and memory mapping for a 16-city problem. There are 16 threads corresponding to each city. Each thread access four memory consecutive memory locations. Thus, a total of 64 memory locations are accessed. Threads are coalesced since memory access is sequential and aligned. 132
6.5 Figure showing a common vertex (v) of the two parents aligned in (a)forward (b) reverse directions. Vertices A and B are left and right neighbors of the vertex v in the red parent. Vertices C and D are the left and right vertex v in the blue parent. 134
6.6 Figure showing parents aligned in (a)forward (b) reverse directions. The common vertex v is of degree 2. After removing the common edges, the resultant is (c) a single vertex with no edges. 134
6.7 In this figure, the common vertex v has degree 3. The parents are aligned in forward direction. Figures (a) and (c) show the two possible cases of vertices before partition. Figures (b) and (d) show the vertices after deleting the common edges. 135
6.8 In this figure, the common vertex v has degree 3. The parents are aligned in reverse direction. Figures (a) and (c) show the two possible cases of vertices before partition. Figures (b) and (d) show the vertices after deleting the common edges. 136
6.9 Figure (a) shows a degree 4 common vertex v. The parents are aligned in forward direction. In figure (b), the degree 4 vertex is split by introducing a ghost vertex. Note that the numbering of the ghost node is v+N. The common edge is removed and the vertex is split into (c) and (d) partitions. 137
6.10 Figure (a) shows a degree 4 common vertex v. The parents are aligned in reverse direction. In figure (b), the degree 4 vertex is split by introducing a ghost vertex. Note that the numbering of the ghost node is v+N. The common edge is removed and the vertex is split into (c) and (d) partitions. 138
6.11 Figure showing the steps in ECL-CC. The four graphs from left to right: (a) Initial graph (b) Initialization (c) Intermediate Pointer Jumping and (d) Hooking. The subscripts represent the label. 142
6.12 Figure showing the integration of partition phase with the recombination phase. 144
6.13 Figure showing the block diagram for integrating the fine-grain partition phase with MGA. 145
6.14 Average Speedup of the GPU-based partition phase over the naive sequential CPU codes. The number of GPU threads=26,624 (104 blocks, 256 threads per block). 149
6.15 Speedup of the GPU+CPU based GPX operation over the naive sequential CPU codes. The number of GPU threads=26,624 (104 blocks, 256 threads per block). The overhead includes memory allocation, deallocation, transfers between CPU and GPU. 149

A.1 Random Access Reads (RAR) count for different EX versions on dsj1000 city problem collected over 200 individual runs 177

B.1 Example of a TSP file format 180
B.2 Calculation of TSP edge distances for various input formats 180

Chapter 1

Introduction

The Traveling Salesman Problem (TSP) is one of the most intensively studied NP-hard problems. Given a set of N cities, and the distance d_{ij} between them, where for city i and j such that $0 \leq i, j < N$, the goal of TSP is to find the shortest Hamiltonian circuit through these cities. Let π denote a vector of length N representing a permutation of the cities. Mathematically, the goal of TSP is to find a permutation with minimum length, given by the equation

$$\sum_{i=0}^{N-2} d_{\pi(i)\pi(i+1)} + d_{\pi(N-1)\pi(0)} \tag{1.1}$$

The direct application of TSP is in the transportation industry, to find the shortest route. However, TSP has been studied extensively and has contributed to numerous scientific innovations and discoveries. TSP serves as a popular benchmark in the design and development of metaheuristic solvers [5, 6]. It is also used in a wide variety of industries [7]. The TSP solvers have been used in the semiconductor industry to optimize scan chains in integrated circuits. Scan chains are routes included on a chip for testing purposes. It is useful to minimize their length for both timing and power reasons. A modified version of TSP has found its place in DNA sequencing [8], genetic linkage mapping [9], protein function modeling [10] and X-ray crystallography [11].

The TSP solvers can be broadly classified into exact [12] and inexact methods. Exact solvers are based on dynamic programming, linear programming, and branch and bound techniques. They can guarantee the optimality of the solution. However, exact solvers consume considerable time and space on large problem sizes. In the worst case, the exact methods can take exponential time. Concorde [13] is a powerful exact solver and has contributed to several mathematical discoveries. On the other hand, inexact solvers are based on approximate and heuristic methods and can find solutions quickly but cannot prove optimality. The Lin-Kernighan Helsgaun (LKH) heuristic [14], and Edge Assembly Crossover (EAX) [15] are two of the best inexact methods.

1.1 Single best solver ?

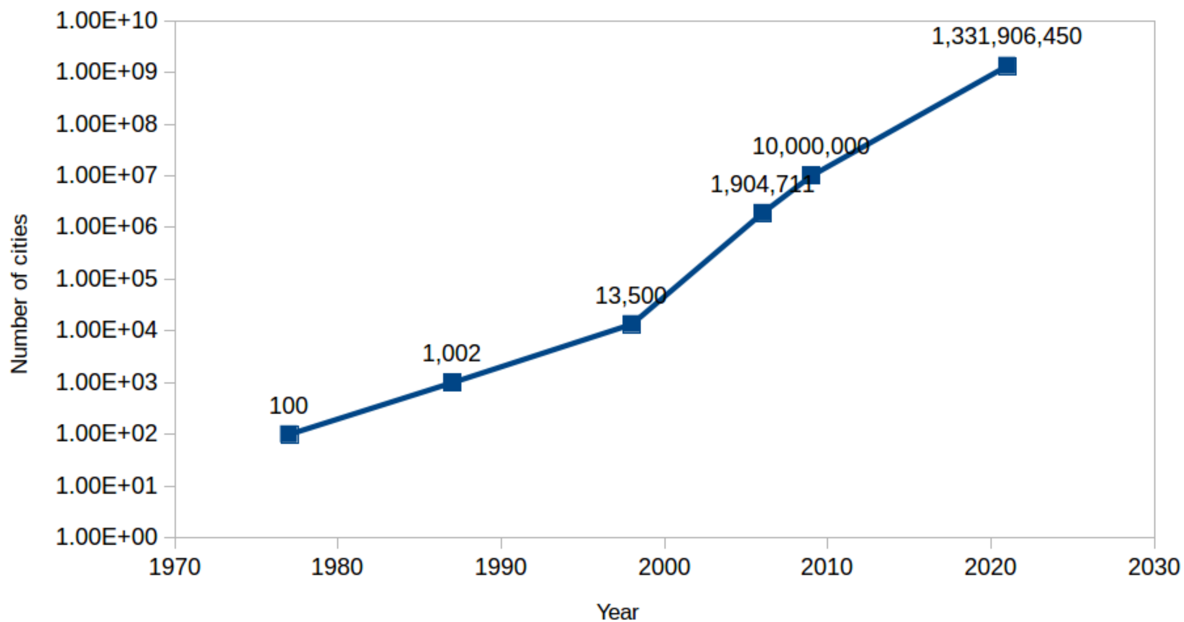


Figure 1.1: Graph showing the scalability of TSP solvers. The years correspond to when an optimal solution was found for the particular problem size. The 1.9M, 10M and 1.3B problems are not solved to optimality.

Figure 1.1 shows the semi-log plot of TSP milestones with the year on the horizontal axis and problem size on the vertical axis. It can be seen that there is an exponential growth in scalability over the past 45 years. As of 2021, the Concorde and LKH are used to find the best-known solution for the 1.3B instance. Please refer to Appendix B for references to the TSP instances in the graph.

The Concorde is computationally intensive. However, when an approximate solution is given as input, it can determine if the solution is optimal or not within a reasonable time. The LKH is based on stochastic local search (SLS), whereas the EAX is based on genetic algorithms. The SLS algorithms maintain and modify the search on a single solution, moving from one solution to another in the search space. On the other hand, genetic algorithms work on a population of solutions. Thus, genetic algorithms are memory intensive. In hindsight, if we consider just the time and compute resources, LKH is undoubtedly the best solver. However, this judgment is biased because the performance of the solvers varies depending on the type of instance.

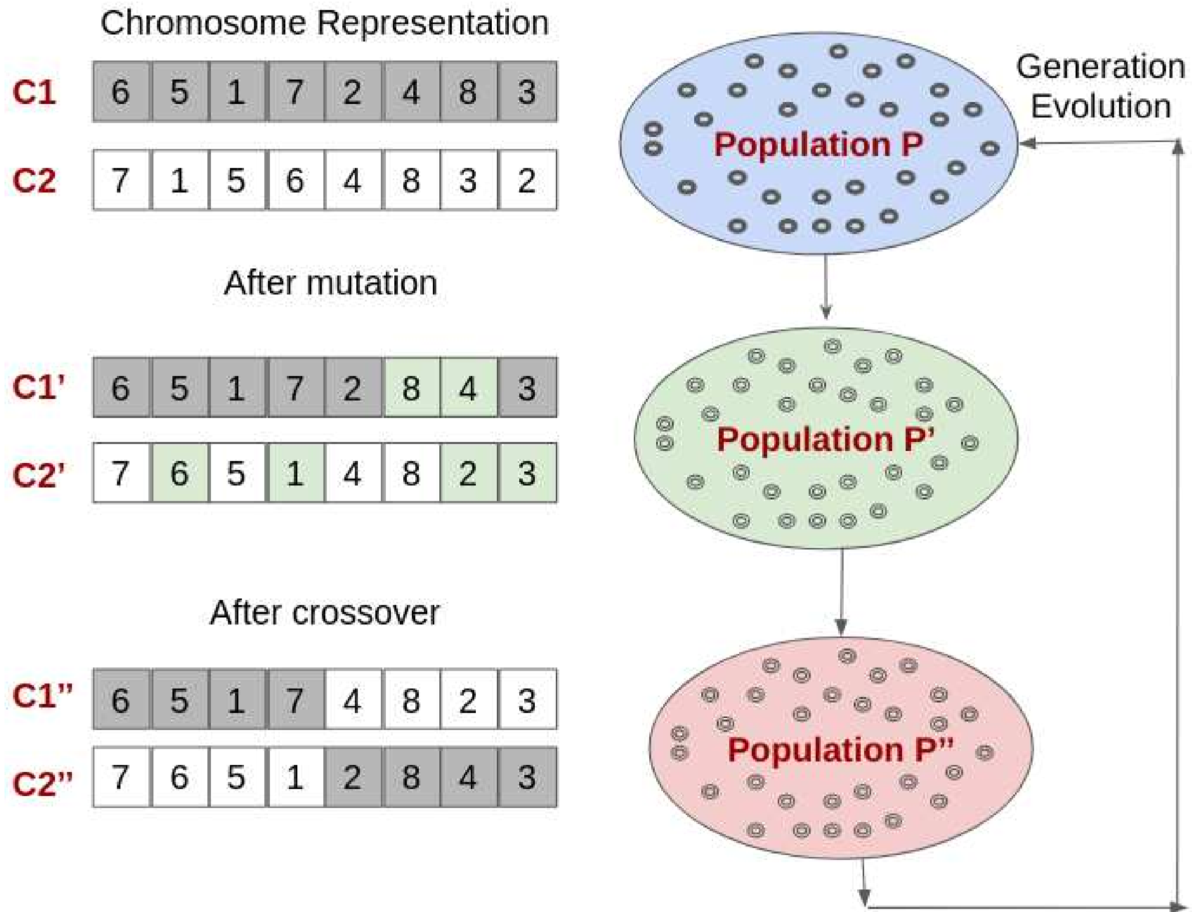


Figure 1.2: Diagram showing a typical Genetic Algorithm (GA) and operations. A GA starts with an initial population (P). Every individual has a chromosome representation. For Traveling Salesman Problem (TSP), the chromosome is a real-valued vector representing the tour permutation of the N cities. Every chromosome has an *evaluation function* associated with it. For TSP, the *evaluation function* is the length of the tour, a simple addition of the distances between the cities. Genetic operators such as the *selection*, *mutation* and *crossover* are applied to *minimize* the length of the tour. The population keeps *evolving* generation after generation until the desired tour length is obtained. In this example, the population P' is obtained by applying the genetic *mutation* operation on P. Population P" is obtained by applying the genetic *crossover* operation on P'. The *selection* operator decides which chromosome will take part in the evolutionary process. In this example, all chromosomes are selected. However, the partners for crossover are randomly selected. Consider the two chromosomes C1 and C2. The mutation operation changes the tour permutation. However, the *crossover* operation exchanges parts of the chromosome between the two individuals. Note that the resulting chromosome (also called the *offspring*) must be *valid*. For TSP, a chromosome is valid if it contains all the city numbers between 1 and N exactly once.

Kerschke et al. [16] study the performance of five algorithms - LKH, a variant of LKH, EAX, EAX with restart mechanisms, and Multi-Agent Optimization Systems [17]. They select a wide

variety of TSP instances from five different dataset[1]. An algorithm selector based on three supervised learning techniques reveals that *EAX with a restart mechanism* is the single best solver. However, their research is limited to 1,875 problems smaller than 2,000 cities.

EAX is the single best solver for smaller instances, so a natural question arises about larger problem sizes. As of 2013, a multi-core CPU implementation of EAX [15] can solve up to 200,000 city problems within 28 days using 30 CPU cores. However, the memory resources for larger problems are high. For instance, to solve the 200,000 city problem on a population of size 300, EAX takes about 6 GB of data. The 6GB data was distributed across 30 cores, resulting in 10 solutions per core. However, consider the 1.3 billion-city problem. The problem would take 40 TB of data. Distributing this data across multiple cores and the corresponding communication time can slow down the convergence rate. In hindsight, the LKH seems to be the best solver for larger-size problems, owing to its resource requirements. LKH works on one single tour, and thus, the memory and communication requirements scale better than population-based solvers. However, these arguments are purely based on CPU-based implementations.

EAX is a highly parallel algorithm. Figure 1.2 shows the functioning of a typical genetic algorithm. It can be seen that the GAs are naturally parallel in nature. The GAs exhibit two levels of parallelism. Let us call it as *population-level* and *operation-level*. The *population-grain* parallelism is when the population evolves in parallel. Here, operations on each solution or group of solutions are executed in parallel. The *operation-level* is when the genetic operations are parallelized. Here, operations on each city or group of cities are executed in parallel.

Massively parallel hardware architectures such as the GPU can be effective for genetic algorithms because of the abundance of parallelization opportunities. EAX algorithm is parallelized only at the population level. However, the EAX crossover operation is highly sequential. On the other hand, the LKH does not exhibit the massively parallel features of EAX. Thus, in this thesis, we focus on one broad problem statement - Can the GA-based TSP solver be efficiently implemented to use the inherent massive parallelism for large size problems?

[1] https://tspalgsel.github.io/

1.2 Computer Architecture Paradigms

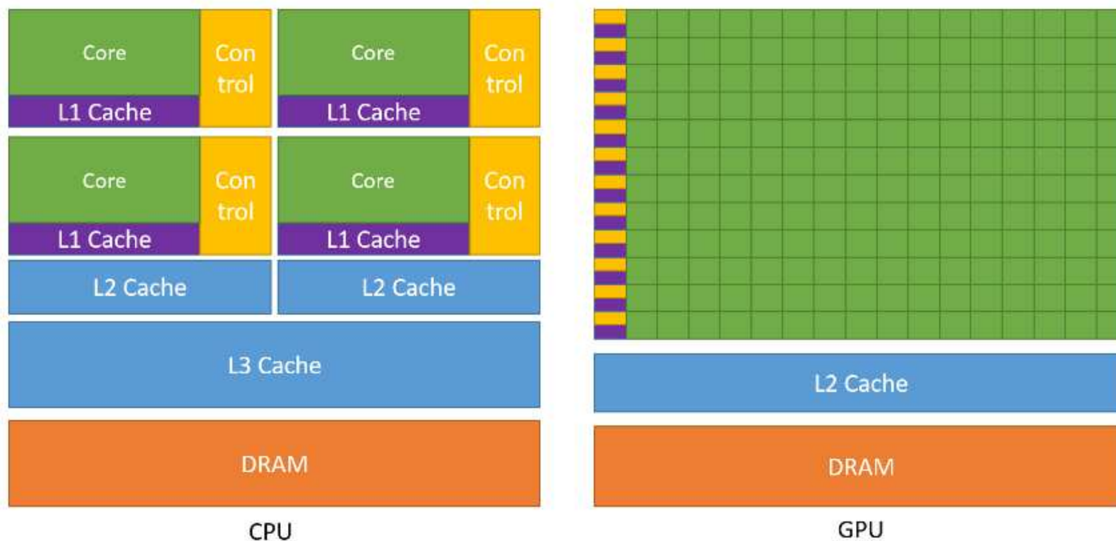


Figure 1.3: The *many-core* CPU architecture has specialized cache and control logic units. The majority of hardware units in *many-thread* GPU architecture is devoted to the data processing cores.

Genetic Algorithm solvers' design parameters and requirements vary with the type of parallel computer architecture we consider. In the past decade, the semiconductor industry has been focusing on two parallel trajectories, namely, the *multi-core* and *many-thread* architectures. Figure 1.3 is taken from CUDA Programming Guide [2]. It shows the architectural difference between a typical *multi-core* CPU and *many-thread* GPU. The *multi-core* architectures are designed to improve the speed of the sequential algorithms by developing highly specialized hardware units such as the cache and control logic. In contrast, the *many-thread* architectures focuses on *execution throughput* of parallel implementations. Thus, the *many-thread* architectures have less sophisticated hardware units and a larger number of data processing units, also called the cores. The focus of GPU architecture is to use as many threads as possible and process data in parallel.

Now, the EAX algorithm can run efficiently on multi-core CPU architectures. However, there is no implementation of the EAX on the GPUs. There are two GPU-based TSP solvers [18, 19] that perform *population-level* parallelization. However, *population-level* parallelization on the

[2] https://docs.nvidia.com/cuda/cuda−c−programming−guide/

GPUs can limit the solver's scalability. The work by Fujimoto and Tsutsui [18] uses the Ordered Crossover operator and could scale only up to to 500 cities. The work by Zhang et al. [19] using the Partially Mapped Crossover (PMX) operator could scale up to 2392 cities. The GPU implementation by Kang et al. [20] can find the best solution for the 1M city problem. However, the effectiveness of the solver is not studied on a variety of problems, and the performance is not evaluated against EAX or other state-of-the-art techniques.

The scalability of TSP solvers on other parallel computer architectures such as the Mesh [21], Hypercube [22], FPGAs [23, 24] is also limited to a few thousand cities. This thesis considers four different computer architectures: the shared-memory CPU, distributed memory CPUs, GPUs, and Mesh architectures. In section 1.3, we briefly provide the learning outcome of the case study on Mesh architecture. The details of the case study are provided in Appendix A. Based on the case study from the Mesh architecture, we pose research questions in section 1.4. The chapter organization and contributions are enumerated in section 1.5.

1.3 Motivation and Challenges

The motivation of the thesis stems from an industrial internship with Singular Computing LLC. The Singular Computing machine (also called S1) [3] is a massively parallel mesh architecture with 34,000 cores arranged in a 4×4 grid of chips. Each chip, in turn, has about 2,112 cores arranged in a 48×44 two-dimensional grid. The peak performance of the machine is 8.5 TFLOPS, and the peak throughput is 8.5Tb/s. The key cool idea of this machine is that it uses approximate computing and thereby achieves less power and area consumption. An approximate computing machine produces an inaccurate answer and can be used for applications where accurate results are not required. In the case of TSP, we do need accurate results for smaller problem sizes. However, an approximate solution is adequate for larger problem sizes, where the optimal solution is unknown. The S1 machine belongs to the Single Instruction Multiple Data (SIMD) parallel architecture. It

[3] http://www.singularcomputing.com/

can produce results as good as a modern GPU, but with $30 - 50\times$ better compute per watt. S1 machine uses a low-level programming language called the Nova.

The proposed task was to implement the EAX algorithm on the S1 machine. The proposed parallelism was across the population. Ideally, with 34,000 cores, one would imagine that the solver could perform genetic operations on 34,000 individuals in parallel. However, this was not the case. Appendix A gives a detailed description of the study. The learning outcome showed that implementing GA based TSP solvers on this particular machine has three significant challenges:

1. The *Memory footprint* required to store an individual and the data structures associated with the genetic operation is too large. EAX crossover operation consumes about $100N$ of space, where N is the problem size. Therefore, several cores had to be combined to perform one crossover operation. These memory restrictions limit the solver's scalability in terms of problem size and population size.

2. *Communication costs* are expensive. The EAX crossover operation randomly accesses the memory to calculate the distances between the cities. Since the SIMD machines do not have specialized cache and control logic, these irregular random memory access patterns are expensive and undesirable.

3. The *Rate of convergence* of EAX solver is the best amongst the GA-based TSP solvers. However, suppose we use a solver that has a smaller memory footprint and restricted communication. In that case, we run into the risk of having a solver with a slower convergence.

Therefore, to design an efficient TSP solver on massively parallel computer machines, there is a need to modify the existing algorithms. Such a modified algorithm must satisfy both the algorithmic and architectural properties. In this thesis, we use eight metrics to evaluate the solver. The memory and communication costs of a solver affect its performance depending on the architecture used. The solver should scale well and converge fast, irrespective of the computer architecture. The execution time is dependent on both the algorithm and the architecture used.

1.4 Research Questions

We ask and answer the following questions.

- Can we design a GA for the Traveling Salesman Problem that satisfies the algorithmic and architectural properties? How does the performance of such a GA compare to EAX in terms of (a) Memory and (b) Communication costs? (c) execution time (d) number of generations to converge (e) scalability in terms of population size (f) scalability in terms of problem size (g) number of recombinations to reach the global optima (h) success rate of the solver ? Chapter 3 answers these questions.

- How does the performance of such a GA compare to EAX on a variety of problems? What features help understand the performance of the solver? Can we find a single best solver on a variety of problems? Chapter 4 answers these questions.

- Does scalability of such a GA benefit from its *population-level* parallel implementations? How useful are the parallel implementations? Can we find a single best solver for problems larger than 2,000 cities? Chapter 5 answers these questions.

- Does scalability of such a GA benefit from its *operation-level* parallel implementations? Is there a need to modify the sequential implementation? Chapter 6 answers these questions.

1.5 Organization and Contributions

The rest of the thesis is organized in to eight chapters as follows.

- Chapter 2 presents the literature review. The state-of-the-art exact, inexact solvers, parallel genetic algorithm solvers are described, and the gaps in the literature are discussed.

- Chapter 3 is the **main contribution** of this thesis. The Mixing Genetic Algorithm (MGA) using the Generalized Partition Crossover (GPX) operator was designed to satisfy the memory and communication costs constraints of SIMD architecture. The GPX consumes 4× less memory when compared to the EAX crossover operator. The GPX crossover incurs *zero*

communication cost to the memory. The MGA using GPX reaches the global optima using fewer recombinations in comparison with EAX. The MGA is shown to have faster convergence with increasing population size. The solver is tested on problems smaller than 5,000 cities. This chapter is published in GECCO 2019 [25].

- Chapter 4 is a study on the convergence of MGA. Two properties - *edge frequency distribution* and *multiple global optima* help understand the working of the solvers. Visualization methods to understand the population-based solvers are introduced. MGA can effectively mix the *edge distances*. It pushes the global optima edges to the top, non-global edges to the bottom of the population. Whereas EAX filters out the non-global edges. A hybrid of EAX and MGA can solve some difficult TSP instances. The experiments are run on problems smaller than 5,000 cities. A part of this chapter is published in GECCO 2020 [26].

- Chapter 5 presents the *population-level* parallelization of MGA. Several island model versions of MGA are studied. For smaller problem sizes, the parallel versions achieve linear speed-up. However, for larger problem sizes, the convergence rate is slow. Therefore, several hybrids of MGA and EAX are studied to overcome the convergence shortcoming. These hybrids improve the success rate of certain difficult TSP instances. Experiments are conducted on problems as large as 85,900. A parallel ensemble of this hybrid solver outperform the performance of EAX. A parallel ensemble of inexact solvers and the hybrid solvers is shown to be more effective. Part of this chapter is published in GECCO 2021 [27].

- Chapter 6 discusses research findings on *operation-level* parallelization of GPX operators. The findings indicate that the data structure and representation of the traditional GA-based TSP solver has to be modified to utilize the GPU features fully. Two major phases of the GPX operator are identified, namely, the partition and the recombination. For problems larger than 140,000 cities,the majority of the time is spent in the partition phase. For problems ranging between 10K and 10M city problems, the GPU implementation of the partition phase achieves a 48x to 625x speedup over the naive sequential CPU implementation.

- Chapters 7 and 8 concludes the thesis with discussions on various future research directions.
- Appendix A provides the case study on Singular Computing Mesh Architecture.
- Appendix B provides the references to all the TSP instances discussed in this thesis.

Chapter 2

Literature Review

The Traveling Salesman Problem (TSP) is one of the most intensively studied NP-Hard problems in the literature. In chapter 1, we discussed the importance of Genetic Algorithm (GA) based solvers for TSP. The GA-based solver, Edge Assembly Crossover (EAX), is found to be the single best solver [16] on a wide variety of problems. We also raised research questions on the effective parallelization of GA solvers on different computer architecture. In this chapter, we describe the working of the state-of-the-art TSP solvers. Sections 2.4, 2.5 and 2.6 describes the working of state-of-the-art exact, inexact and genetic algorithm solvers, respectively. Finally, in section 2.7, we critically analyze their parallel implementations for gaps and improvements.

The TSP solver starts and terminates with a tour. A random tour permutation is an acceptable starting point. However, the convergence of the solvers depends on a good initial starting point. Section 2.3 describes some of the tour construction heuristics. Section 2.3 describes the "local search" techniques used to improve the initial tour. "Local search" techniques do not guarentee the global optima solution. However, they are used in combination with the exact and inexact solvers.

The solvers can be terminated at the global optima. However, in reality, the optimal solution is not known. Therefore, there is a need to find a "lower bound" to terminate the solvers. Section 2.1 describes the working of some of the successful "lower bound" techniques.

2.1 Lower Bounds

The TSP is an NP-Hard problem. A solution to TSP is a simple permutation of the cities. Suppose N is the problem size, the number of possible solutions is $(N-1)!/2$. It is impossible to enumerate and find the exact optimal solution. Therefore, having a good idea of the bounds of the solution can help in pruning the search space. Starting at a random vertex and just connecting to the next shortest neighbor can give a reasonable upper bound. The two standard methods to find the lower bounds are the minimum spanning 1-tree and the Held-Karp bound.

2.1.1 Minimum Spanning 1-tree

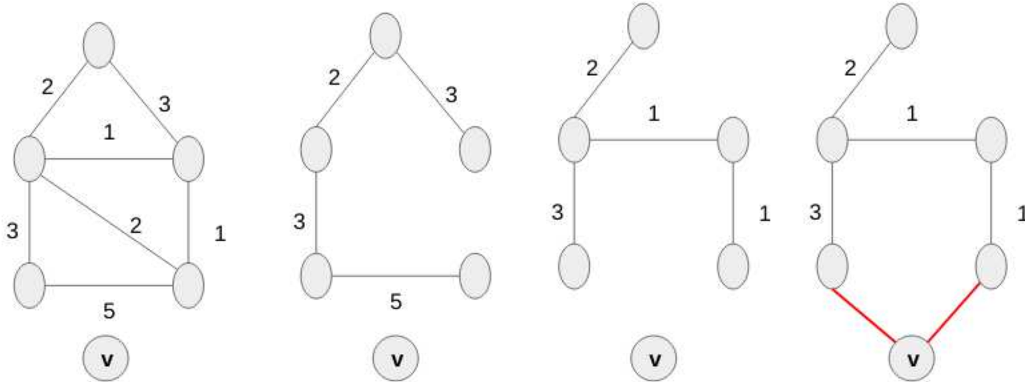


Figure 2.1: Diagrams illutrating the Minimum spanning 1-tree. The left-most graph G is connected except for the vertex v. The second graph is the spanning tree of G with a cost = 13. The third graph is the minimum spanning tree of G with cost = 7. The final, right-most graph is the Minimum Spanning 1-tree, where the vertex v is now connected to the MST using the two of its short edges, represented in red color.

The TSP can be modeled as a weighted graph G(V, E, W) with cities as vertices, edges as path between them and the weights as the corresponding distance. The lower bound of TSP can be determined by finding the "minimum 1-tree" of G as illustrated in figure 2.1. First, remove an arbitrary vertex v from the graph. Second, find the minimum spanning tree (MST) of the graph excluding this vertex. Finally, attach the vertex v to the MST using two of its shortest edges.

The minimum spanning 1-tree consists of two parts - (a) an MST and (b) a vertex attached by two edges. The degree of all vertices in a TSP tour is 2. Therefore, a TSP tour must go through this vertex v and visit all other cities in the graph. The shortest way to do this is by following the MST. Therefore, the MST excluding the vertex v is the lower bound of TSP. There can be several minimum spanning 1-tree depending on which vertex is removed. However, a better lower bound will be the largest of all such minimum spanning 1-tree.

Several algorithms have been developed to find the MST [28]. Greedy algorithms such as the Bourukva's [29], Prim's [30], Krushkal's [31] are the most commonly used implementations. A parallel implementation [32] [33] can solve the MST in $\mathcal{O}(logN)$ time using a linear number of processors on the exclusive-read exclusive-write PRAM.

2.1.2 Held-Karp bound

The Held-Karp (HK) lower bound [28, 34] is a relaxation of the Minimum Spanning 1-tree. The 1-tree can have vertices of varied degree. However, for TSP, each vertex must be a degree of 2. If we make the degree of each vertex in the 1-tree to 2, we end up in the optimal tour. However, in practice, the HK bound does not give the exact optimal solution [35]. It is usually within 2/3 of the optimal solution [36, 37] when "triangle inequality" is satisfied. For the random eucleidian instances, it is found to be within 0.8% of the optimal length [38, 39].

To find the HK bound, we need to iteratively make simple modifications to the edges of the minimum 1-tree until no significant improvements are found or if we reach the optimal tour. Suppose d_{ij} is the distance between vertex i and j. Assign arbitrary weights to each vertex. Let π_i denote the weight of vertex i. Now, the transformed edge distance is $d'_{ij} = d_{ij} + \pi_i + \pi_j$. This transformation gives a new 1-tree and the TSP bound changes. Suppose L is the length any tour, each node appears twice and thus, with the new distances, the new length is, $L + 2 * \sum_i \pi_i$. Therefore, the new bound is always a constant away from the original length.

Let deg_i^k be the degree of ith vertex of kth tree. Let T_k be the cost of kth tree from the original. The cost of kth tree is $T_k + \sum_{i \in V} deg_i^k \pi_i$. After enumerating all such k 1-trees, the minimum weight 1-tree on the transformed distance matrix is given by,

$$\begin{aligned} L^* + 2 * \sum_{i \in V} \pi_i &\geq \min_k (T_k + \sum_{i \in V} deg_i^k \pi_i) \\ L^* &\geq \min_k (T_k + \sum_{i \in V} (deg_i^k - 2)\pi_i) = w(\pi) \end{aligned} \tag{2.1}$$

The best lower bound comes from maximizing $w(\pi)$ over all values of π. Held and Karp [28, 34] use the ascent method to find the best π. If UB is the upper bound, then,

$$\begin{aligned} \pi_i^{j+1} &= \pi_i^j + t_j (deg_i^j - 2) \\ t_j &= \frac{\lambda_j (UB - w(\pi^j))}{\sum_{i=1}^n (deg_i^j - 2)^2} \end{aligned} \tag{2.2}$$

where, $0 < \lambda_j \leq 2$. Please refer to Held and Karp [28, 34] for more details on the proof.

2.2 Tour construction

The convergence of a TSP solver depends on the quality of the initial tour. Following are some of the commonly used tour construction heuristics. The time complexity of the first heuristic is $\mathcal{O}(N^2)$ and that of the remaining three algorithms is $\mathcal{O}(N^2 logN)$.

- **Nearest Neighbor:** In this method, a tour is constructed by connecting each city to one of its unvisited nearest neighbour. The first city is randomnly selected.

- **Greedy algorithm:** The TSP is viewed as a complete graph, where vertices represent the cities and the edges represent the distances between them. A TSP tour is simply a Hamiltonian cycle in this graph. The Greedy algorithm constructs the tour by starting with the shortest edge and keeps adding remaining shorter edges until a Hamiltonian cycle is formed.

- **Clarke-Wright savings:** First, a random city is selected as the central node. The salesman visits each city and returns to this central node. However, the tour length can be shortened if the salesman bypassed the central node and went directly between the cities. This savings in the tour length is calculated for every pair of non-central node cities. The TSP tour is formed in a greedy fashion by traversing this savings list from largest to the smallest.

- **Christofides algorithm:** is an approximation algorithm developed in 1976 by Christofides [40] for symmetric TSP. The algorithm guarantees that its solution is within a factor of 3/2 of the global optima. Again, consider the TSP to be a complete graph. The algorithm proceeds by first forming the Minimum Spanning Tree (MST) of the graph. Then, the Minimum-Weight Perfect Matching (MWPM) of the subgraph of MST with only the odd degree vertices is calculated. Next, an Eulerian circuit is formed from the union of the MST and MWPM. Finally, the TSP tour is created by skipping the repeated vertices. For almost 44 years, this algorithm has been the best polynomial-time approximation algorithm. However, in July 2020, Karlin, Klein, and Gharan [41] could guarantee an approximation ratio of $1.5 - 10^{-36}$. Their procedure is similar to Christofides' algorithm. However, it uses a randomly chosen tree from a carefully chosen random distribution in place of the MST.

2.3 Stochastic Local search

The initial tours can be improved using the "local search" heuristics that performs a simple exchange of edge distances. The commonly used local search methods are the 2-opt and 3-opt. In 2-opt, two edge are exchanged, whereas, in 3-opt, 3 edges are exchanged. A generalized form is called the K-opt, and it forms the integral part of the LKH [42, 43] solver. In this section, we explain the basic principle behind these heuristics, along with several optimization approaches.

2.3.1 2-opt

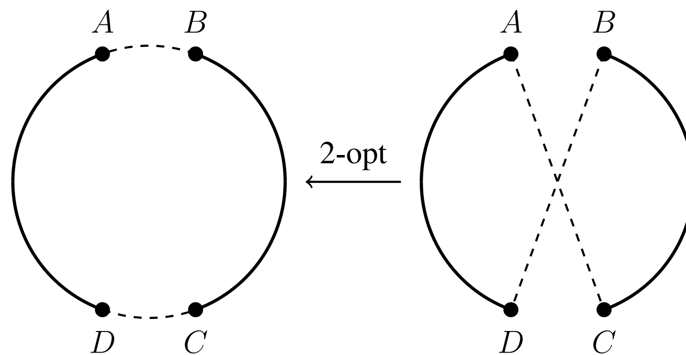


Figure 2.2: Figure showing a 2-opt move on the original tour to the left. After removing the edges AB and CD, edges AC and BD are added and BC segment is reversed. The tour to the right has a shorter length.

The 2-opt local search was first introduced in 1958 by Croes [44], whereas the basic move was given in 1956 by Flood [45]. The idea behind the 2-opt local search is to take parts of the tour that crosses itself and re-order so that there is no cross. Figure 2.2 shows the 2-opt move. Consider the cities A, B, C, and D on tour. If the sum of the edges AB and CD is greater than the sum of the edges AC and BD, then a 2-opt move is performed. The move involves three processes: (I) Delete edges AB and CD. (II) Reverse the tour between cities B and C (III) Insert edges AC and BD. Thus, the resultant tour has a shorter length. This 2-opt search is performed on all the N cities. In the worst case, the algorithmic complexity can be $\mathcal{O}(N^2)$. Techniques to reduce the complexity of 2opt such as the nearest neighbor [46] and don't look bits [47] methods have been well studied.

- **Nearest Neighbours:** The nearest neighbours concept restricts the edge exchanges to only the nearest M neighbours of a city. For instance, in figure 2.2, the 2opt swap is performed if and only if the city C is within first M nearest neighbours of city A. Similarly, we can restrict the neighbours of city B as well. Usually, M is a constant less than 100. This reduces the time complexity of 2opt search to $\mathcal{O}(NM)$, a linear time improvement.

- **Don't look bits:** Here the observation is that if for a given choice city, if we previously failed to find an improving move, and if the city's neighbors have not changed since then, then it is unlikely that we will find an improving move if we look again starting at the same location. We exploit this observation by means of special flags for each of the cities, which we call don't-look bits. Initially, these are all turned off. The bit for city is turned on whenever a search for an improving move fails and is turned off whenever a move is performed in which the city is an endpoint of one of the deleted edges. In considering neighbours for the city, we ignore all cities whose don't look bits are on. This is done by maintaining a first-in,first-out queue of cities whose bits are off.

- **Candidate edges:** The set of edges selected to perform the *K-opt* (where $2 \leq K < N$) moves are called the candidate edges. One popular way is to restrict the edges to nearest neighbours of a city. LKH [43] uses techniques such as the α-nearness and backbone-guided search to create the candidate edges. Given the cost of minimum 1-tree, the α-value of an edge is the increase of this cost when a minimum 1-tree is required to contain the edge. The α-values provide a good estimate of the edges chances of belonging to an optimum tour. The back-bone guided search [48] uses the structural information of the TSP instance to create the candidate edges. In the LKH algorithm, the edges of the tours produced by a fixed number of initial trials are used as candidate edges in the succeeding trials.

- **Parallelization:** can also help to speed up the 2opt search. The tour can be paritioned into segments. Alternatevily, if the TSP instance is a geometric, the tour can be paritioned into rectangles or clusters. The 2opt can be applied to each of these partitions in parallel.

2.3.2 3-opt

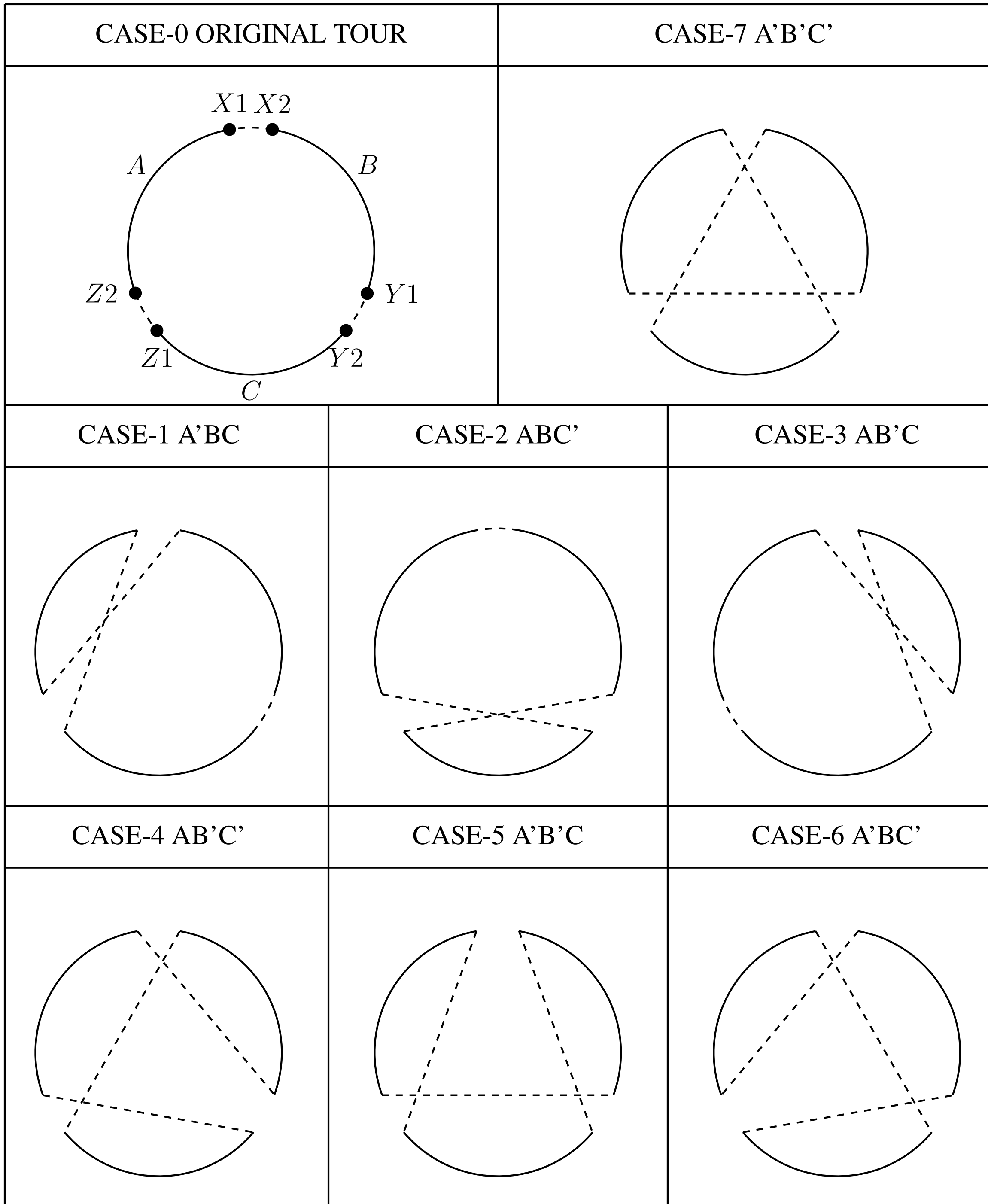


Figure 2.3: Figure showing 3-opt moves. The prime symbol means that the segment is reversed. There edges (X12, Y1Y2, Z1Z2) are modified to get a shorter tour. There are 7 possible ways of modifying these 3 edges. Case 1, 2, 3 is equivalent to performing one 2opt operation. Case 4,5,6 is equivalent to performing two subsequent 2opt operations. Casr 7 is equivalent to performing three subsequent 2opt operation.

The 3opt considers three edges at a time, finds the best out of seven different ways of reconnecting the edges to find the optimal tour. Figure 2.3 shows eight different ways of conecting the tours. The first case (case 0)is the original tour. 3/8 cases are similar to 2opt moves since one of the original edge is retained. 3/8 cases are similar to two subsequent 2opt moves. The final case (case 7) is similar to applying three subsequent 2opt moves. Therefore, one 3opt move is slower than 2opt. One full 3-opt iteration takes $\mathcal{O}(N^3)$ time. The 2opt optimizations such as the nearest neighbours, don't look bits, geometric partitioning, tour paritioning can also performed for 3-opt moves. Two notable 3-opt optimization in literature are,

- **POPMUSIC:** Partial OPtimization Metaheuristic Under Special Intensification Conditions [49] (POPMUSIC) is a framework developed specifically to tackle larger problem sizes for several optimization problems. The goal of POPMUSIC is to partially optimize these problems and merge them into the original solution. For TSP, POPMUSIC divides the tour into sub-tours and applies the 3-opt [42] local search operator. The critical point is the low empirical time complexity ($\theta(N^{1.6})$) and high quality solutions for large instances. It is tested on problems as high as 10M cities. It is integrated with the LKH [43] as a pre-processing step to produce a good set of *candidate edges*.
- **Parallelization:** Rocki and Suda [50] parallelize the 2opt and 3opt moves on the GPUs. Each thread calculates one 2-opt or 3-opt exchange. Finally, the best move is identified. Results on problems smaller than 5,000 cities show a 26$\times$ speedup over the CPU version.

2.3.3 K-opt

The K-opt moves replace K edges, where $2 \leq K < N$, to form a shorter tour. A K-opt move is a generalized form of 2-opt and 3-opt. It was first used in the Lin-Kernighan (LK) [42] local search heuristic to find the optimal TSP tour. The K-opt moves are an integral part of LKH solver as well. Some notable K-opt moves are,

- **2.5-opt** move relocates a single city from its current location to a position between two current tour neighbors elsewhere in the tour.

– **Double-bridge move** can be viewed as the combination of two illegal 2-opt moves. Consider the four tour segments are A1 A2 A3 A4, in order. The move permutes these into the new ordering A2 A1 A4 A3 (without reversing any of the segments).

– **Lin Kernighan (LK)-heuristic [42]**, one of the most-successul TSP solver, performs k-opt moves iteratively. Each iteration is adaptive and dynamically determines the number of edge exchanges. The LK heuristic restarts from a new initial tour upon stagnation.

– **Chained-LK [51, 52]** heuristic is similar to LK-heuristic except that it applies a *double-bridge* move upon stagnation instead of starting from a new tour.

2.4 Exact Solvers

Exact solvers guarantee the optimality of the solution at the expense of time and space resources. The exact solvers commonly use four types of algorithms namely, (a) Brute-force methods, (b) Dynamic Programming (c) Linear Programming and (d) Branch and bound/cut techniques. Concorde [53] is the state-of-the-art exact TSP solver. We discuss each of these in this section.

2.4.1 Brute-force Methods

The brute-force methods simply enumerate all possible solutions, evaluate each one of them and finally determine the optimized solution. Suppose N is the problem size, the number of possible solutions are $(N-1)!/2$. The time complexity of the brute-force method is $\mathcal{O}(N!)$.

2.4.2 Dynamic Programming (DP)

The DP algorithm, also called the Bellman-Held-Karp algorithm [54, 55], finds the exact solution to TSP and several related problems including the Hamiltonian circuit problem in exponential time. The time complexity is $\mathcal{O}(2^N N^2)$ and the space complexity is $\mathcal{O}(N2^N)$. If only the length of shotest path is required, then the space complexity can be reduced to $\mathcal{O}(\sqrt{N}2^N)$. However, the time complexity is exponential and is not practical for problems more than 30 cities.

2.4.3 Linear Programming

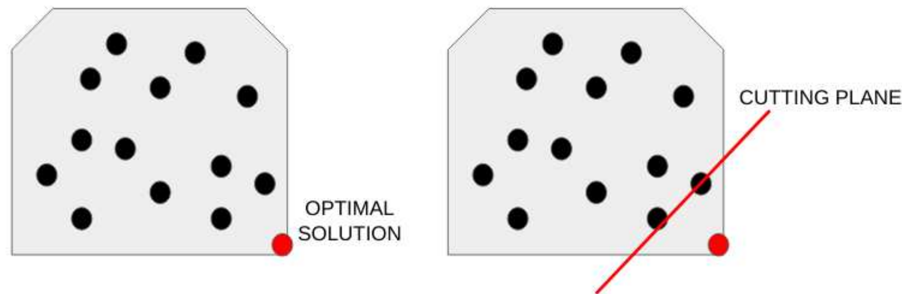


Figure 2.4: Figure illustrating a cutting plane.

The Linear Progamming (LP) formulation is the most-widely used technique. Suppose N is the number of cities, c is the cost matrix. For a TSP, a tour through the N cities can be represented as its incidence vector of length $N(N-1)/2$. Each component of the vector is set at "1" if the corresponding edges is in the tour or "0" otherwise. If x is such an incident vector, then $c^T x$ gives the tour length. So, letting S denote the set of the incident vectors of all tours, the problem is to

$$minimize \qquad c^T x, \qquad x \in \mathcal{S} \tag{2.3}$$

The breakthrough using the linear programming came in 1954, when George Dantzig, Ray Fulkerson, and Selmer Johnson [56] published a paper on the 49-city TSP problem. They wanted to solve the equation 2.3 with some suitably chosen $Ax \leq b$ linear inequalitues satisfied by all x in S. Solving this system can be viewed as a charateristic of the simplex method [57], where the optimal soltuion $x*$ is an extreme point of the polyhedron defined by $Ax \leq b$. If $x*$ is not one of the points in S, then it lies outside the *convex hull* of S. In that case, the $x*$ can be separated by a hyperplane. Therefore, there must be some linear inequality that is satified by all points in S and violated by $x*$. Such an inequality is called the *cutting plane* or simply a cut. Figure 2.4 shows an example of a cutting plane. Having found such inequalities, we can add it to the system $Ax \leq b$, solve the resulting tighter relaxation by the simplex method, and iterate this process until a relation with an optimal solution in S is found.

2.4.4 Branch and bound algorithms

The branch-and-bound (B&B) algorithms for TSP [58], divide the search space into subsets using a procedure called branching. For each subset, a lower bound on the tour is calculated. Eventually, a subset that contains the optimal solution whose length is closer to the lower bound is found. Volgenant and Jonker [59] find solutions upto 100-city problems using the branch and bound techniques. They use the 1-tree relaxation to find the lower bounds. Tschoke et al. [60] use a distribted parallel programming model to solve upto 300-city problems. They use similar B&B methods with 1-tree relaxation. In the worst case, the complexity of B&B can be as bad as the brute-force methods. Therefore, these methods are suited for smaller problem sizes. However, solution to 85,900 [61] city problem was found when combined with the branch-and-cut methods. The branch-and-cut algorithm combines the B&B and cutting plane techniques.

2.4.5 Concorde

Concorde is the state-of-the-art exact TSP solver and it is based on the LP formulation. It can solve TSP instances of sizes 1,000 to optimality within a reasonable computation time. For larger problem sizes, Concorde has been used to verify the optimality of the solution found by heuristic solvers [61]. Verification of the optimal solution still takes a very long computation time.

The core of Concorde is to reduce the LP formulation shown in equation 2.3 using techniques such as cutting-planes [62, 63], branch-and-cut [64, 65] and the Held-Karp [28, 34] bounds. In default mode, the Concorde starts with a tour locally optimized by Chained LK heuristic [66, 53]. It then uses the branch and bound algorithm to search for the optimal solution. The branch-and-cut methods embed the cutting plane methods into this branch and bound search.

The Concorde code is written in the C program and contains about 130,000 lines of code. It supports distributed parallel processing in evaluating node-and-cut tree nodes. It supports shared memory parallel processing when searching the pool of cuts. For large problem instances, often, the results produced from heuristic solvers are given as input [61]. Thus, Concorde can be used in two modes - (1) to find an optimal solution and (2) to verify if the given solution is optimal or not.

2.5 Inexact Solvers

Inexact TSP solvers [12] are fast to converge but do not guarantee optimality. These solvers are based on approximation and several heuristic approaches. Heuristic [67, 68, 6] techniques include Nearest Neighbour [46], Tabu Search [69, 70], Iterated Local Search [71] and Lin-Kernighan (LK) heuristics [42, 43]. These methods also include algorithms inspired from nature such as Simulated Annealing [72], Genetic Algorithms [73], Ant Colony Algorithms [74], Particle Swarm optimization [75], Neural Networks [76, 77] and Multi-Agent Optimization Systems [17].

For most of the last 50 years, LK [42] and LKH [43] heuristics and its variants have dominated the inexact solvers. In 2013, the genetic algorithm based TSP solver called the Edge Assembly Crossover (EAX)[78] produced competitive results. Recently, Multi-Agent Optimization Systems (MAOS) [17] based algorithm is also known to be as competitive as LKH and EAX. In this section, we describe the working of LKH and MAOS. The working of EAX, together with other genetic algorithm solvers are described in section 2.6.

2.5.1 Multi Agent Optimization Systems (MAOS)

MAOS [17] is based on the multi-agent framework-based optimization approach. The framework has four parts namely, the problem representation, the agent, the environemnt and the interaction. The problem representation contains some basic knowledge of the task. Each agent is autonomous and has a compact solving ability. The agents use the problem representation and explore possible solutions in parallel. The agents update the shared environment and communicate their acquired knowledge to the other agents and improve the search efficiency.

For the TSP, each agent starts from a set of reference structures, such as disjoint graph elements that can be combined into valid tours, and candidate sets, such as nearest-neighbor subgraphs. It then uses a Markov-chain approach to generate a set of states by applying search operators, such as local search steps, genetic algorithm crossovers, or simple completion heuristics. The number of agents can vary anywhere between 100 and 1,000. The experimental results show that MAOS is as competitive as the LKH. However, MAOS has only been tested upto 6000-cities.

2.5.2 Lin-Kernighan-Helsgaun (LKH)

The LKH algorithm is an extension of the Lin-Kernighan algorithm [42] for the TSP. The original Lin-Kernighan algorithm uses local search based on“k-opt" moves. The LKH algorithm by Helsgaun [14] also uses additional, highly efficient k-opt moves. In addition, the k-opt neighborhoods are restricted to “nearest neighbor" moves. For each vertex in the TSP, a list of it *candidate edge* is maintained. These candidate edges are selected using the α-nearness, backbone guided search heuristics. In the optimal solution, adjacent vertices in the solution are highly likely to be nearest neighbors (under some measure of nearness). Thus, most moves focus on changes to the current solution that increases the number of nearest neighbor edge in the next candidate solution.

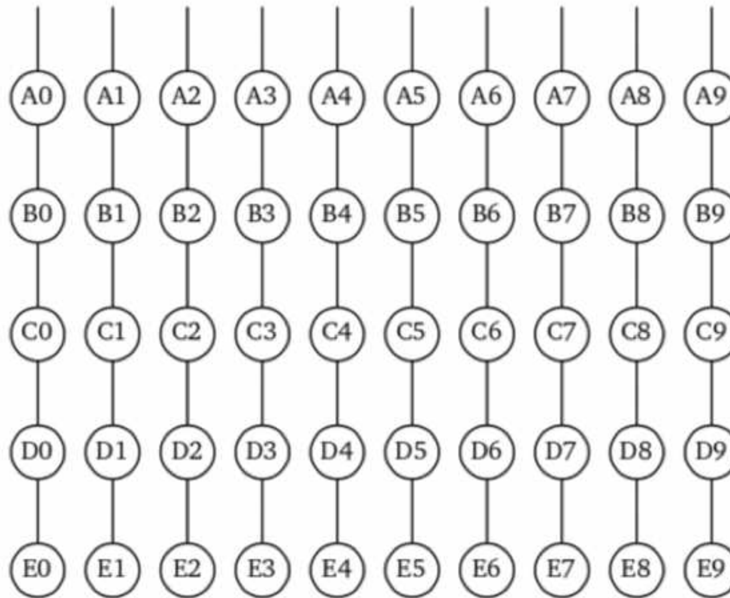


Figure 2.5: Figure showing the use of Iterative Local Search in the LKH algorithm. LKH uses a combination of "soft" and "hard" restarts in combination with efficient and selective k-opt local search. In this figure, a sequence such as A0, B0, C0, D0 represents a sequence of local optima reached using soft restarts. Search is started, then stagnates at solution E0. Then the current solution is perturbed using some random partial alternation of the solution; the soft restart is designed to move the search to a different local optima. A sequence such as A1, B1, C1 represents a different sequence after a full "hard" restart from a different random initial solution. Thus, in this figure we see 10 "hard" restarts, each which has 5 "soft" restarts. [1]

LHK is a form of iterated local search. When local search stagnates, a perturbation is made to the current solution which is designed to allow the k-opt operators to converge to a new local

optimum. (The notion of a local optimum may not be well defined in this case if the variable k-opt moves are stochastically chosen.) Thus LKH will explore multiple local optima. These "soft restarts" are also supplemented with some number of "hard restarts" which restarts the search from a new random starting point. Figure 2.5 shows an example of "hard" and "soft" restarts.

The LKH uses the two crossover operators namely, the Iterative Partial Transcription (IPT) [79] and Generalized Partition Crossover (GPX) [80] during the restarts. This version of LKH is also termed as the "multi-trial LKH." The IPT and GPX are genetic crossover operators, capable of combining two local optima and producing a different local optima. In the multi-trial version, the LKH keeps the best-so-far solution. When it reaches a new local optima, the IPT crossover operator is applied between the new local optima and the best-so-far solution. If the crossover results in an improved solution, the LKH restarts with this new solution. For example, in figure 2.5, B0 can be the result of a crossover between the best-known solution and A0.

The GPX crossover is similar to IPT in function, however, yields superior results[1] [80]. The GPX operator is used in two ways, across runs and across restarts. GPX across restarts is similar to applying the IPT crossover operator. However, GPX across runs will crossover between the local optima with same letters in figure 2.5. At the end of each run, the new found local optima is stored in a population. The best solution is crossed with every individual in the population. Finally, a genetic algorithm is used to recombine the solutions of the population. Descriptions of GPX and IPT are provided in section 2.6 of this chapter.

LKH is scalable, capable of generating near-optimal solutions for 3B-City instances. LKH employs tour partitioning and merging to handle large problem sizes. The six paritioning techniques used are the Tour segment, Karp, Delaunay, K-means, Sierpinski and Rohe paritioning. The tour segment partitioning is a general method, applied to all types of TSP. It divides the tour into equal segments, performs local improvements and merges them back together. The remaining five methods require the problem to be geometric. The Karp and Rohe methods partitions the graph into rectangles. The Delaunay and K-means methods partitions the graph into clusters. The POPMUSIC [49] heuristic is used to create the candidate edges for large problem sizes.

2.6 Genetic Algorithm Solvers

The use of Genetic Algorithms for solving TSP dates back to 1985 [81] when Goldberg and Lingle used *partially mapped crossover* operator to solve a 10-city problem. Other pioneer works in this field include the solution to 200 city problems by Grefenstette [82] and solutions to 105 city problems by Whitley [83, 84]. Since then, several efforts have been made to improve the crossover operators specific to solve TSP problems. But, none of these operators were able to outperform LK based solvers. EAX was introduced first in 1997, and over the years, it has continuously improved and, on average, outperforms other solvers [85]. To the best of our knowledge, there are no other GA based TSP solvers that can outperform EAX. However, there is one crossover operator, called the Generalized Partition Crossover (GPX) [86] developed by Whitley et. al [86, 87, 88], when used as a hybrid, can improve the performance of Chained-LK [89], EAX [90], Concorde [91] and LKH [1, 80]. The IPT crossover operator is used in the LKH solver during soft restarts [79]. In this section, we describe the working of EAX, GPX, and IPT crossover operators.

2.6.1 GA solver terminologies

The figure 1.2 from chapter 1 introduced the basic working of a GA solver. In simple words, a GA solver starts with an initial population, applies genetic operators and terminated when a desired solution is obtained. Following are some of the GA terminologies and parameters.

- **Chromosome Representation:** Each solution in the search space is encoded as a chromosome. The chromosome is usally a bit-vector. However, arbitrarily, any encoding is possible. For TSP, a real-valued vector with the city numbers is the ususal representation.

- **Evaluation function:** Each chromosome has is evaluated using a function. This evaluated value dictates the progress of the GA. For TSP, the evaluation function is the length or cost of the tour. The GA progresses until the desired tour length is obtained.

- **Initial Population:** The initial population is a set of solutions in the search space. Tour construction heuristics can be used to build the initial tour. Most of the GA based TSP

solvers randomnly initialize the population and use a local search operator to optimize the solution.

– **Local search:** The local search methods produce a local optima solution. Most of the GA based TSP solvers use 2opt as the local search method.

– **Genetic operators:** such as the crossover and mutation are applied on the initial population. The crossover modifies two or more chromosomes. The mutation operator modifies a single chromosome. The crossover operation is also referred to as recombination.

– **Parent:** is the input chromosome to the genetic operation.

– **Offspring/Child:** is the output chromosome of the genetic operation.

– **Selection strategy:** The strategy used for selecting parents to perform the genetic operation is called the selection strategy. For mutation, a single parent is selected. For crossover, two or more parents must be selected.

– **Generation:** One iteration where a genetic operator is applied to the entire population.

– **Evolution:** is the process of applying genetic operators iteratively until termination.

– **Diversity preservation strategy:** The success of the GA based solvers depends on the diversity of the population. Unlike local search solvers that moves from one solution to another in the search space, the GA solvers are capable of searching different regions of the space in a single generation. Therefore, there is a need to make the population as diverse as possible and also make sure that this diversity is not lost over the course of the evolutionary process.

– **Stagnation:** is a state where no improving moves are found.

– **Termination condition:** For TSP, the known optimal solution or the HK-bound is used for termination. A pre-defined time, number of generations can also be used to terminate.

– **Success rate:** The GA solvers are stochastic due to the use of a random seed. Therefore, there is a need to run the solver multiple times to measure the success of reaching the optimal solution. Usually, 30 independent runs without changing the parameters are sufficient.

– **Convergence rate:** is the rate at which a GA solver reaches the optimal solution. It is usually measured in terms of execution time. However, if the solver is not designed for speed, parameters such as the recombinations, generations, success rate are used.

– **Gap:** is the percentage difference between the given solution and the optimal solution.

2.6.2 Edge-Assembly Crossover (EAX)

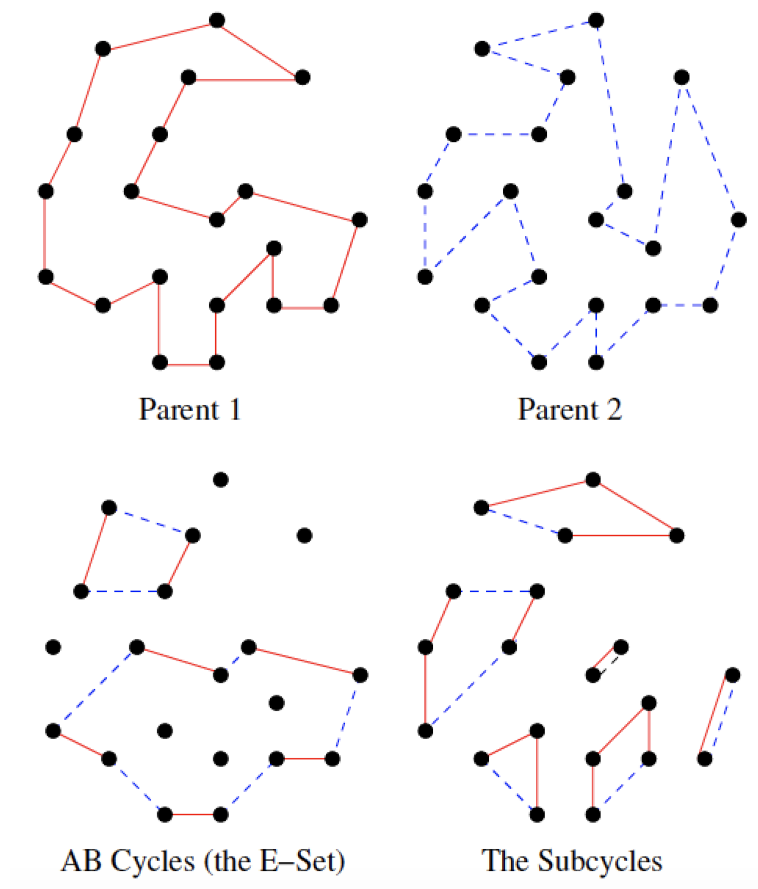


Figure 2.6: These plots show EAX crossover. The graph union of parent 1 and parent 2 yields the AB cycles. The subcycles are formed from parent 1 and E-set. The parent 1 edges in E-set are deleted from the original parent 1. The parent 2 edges in E-set are added to the original parent 1.

Figure 2.6 illustrates the key idea behind EAX [92] crossover operation. Given two parents, EAX searches for random “AB-Cycles" in a graph union of the two parents. “AB-Cycles" are subcircuits that follow an edge from Parent 1 (i.e. “A"), then an edge from Parent 2 (i.e. “B") and continues alternating (ABAB ... AB) until a loop is formed. A subset of the AB-Cycles, called the "E-set," is then used to cut Parent 1 into subcircuits. The subcircuits are composed entirely of edges from the two parents. Finally, the subcircuits are merged in a greedy fashion to create a single offspring representing a new Hamiltonian circuit. This greedy merging process introduces new edges into the offspring, and by extention, into the population.

There is *no selection* in the traditional sense. The order of the population is randomized every generation. Then the parents at location i and i+1, Parent 1 and Parent 2, are recombined; the last and first member of the population are also recombined. Thus, one chromosome takes part in two recombinations, once as parent 1 and once as parent 2.

The EAX algorithm uses a form of *brood selection* to improve the population. Each pair of parents produces 30 offspring, and the best offspring replaces Parent 1. In addition, EAX uses diversity preserving measures to maintain a more diverse set of edges in the population; thus, the *best* child is not always chosen. An edge frequency table is updated, whenever, an edge is added or deleted. The diversity metric using these table entries decides if the child will replaces the parent or not. Chapter 4 explains more on the working of this metric.

The EAX crossover operates in two stage. In the first stage, only one AB-Cycle might be used to cut Parent 1 into subcircuits. This has two consequences. First, the offspring will be much closer to Parent 1, and thus many edges are passed from Parent 1 to the offspring. This is important because this offspring will also replace Parent 1. Second, this conservative form of EAX also reduces the run time cost of EAX. If there are many subcycles then finding the optimal merge is potentially expensive. The current implementation of EAX reduces the number of subcycles and does the merging in a highly efficient way. The resulting merger may not be optimal however.

In the second stage, a more aggressive form of the EAX operator can be used, where multiple AB-Cycles are used to cut Parent 1 into more subcircuits. This only occurs after EAX stagnates

and becomes more difficult to find improved offspring. Finally, the EAX terminates when the difference between the average and best tour length of the population is less than 0.001.

The parallel version of EAX [93] can produce the best-known solutions for up to 200,000 city size problems. EAX spends 90% of the time during stage 1. Therefore, this phase is parallelized using the master-worker model. The master node maintains the whole population. It divides the population equally and sends them to the worker nodes. The worker nodes do not communicate with each other. The worker nodes generate the 30 offspring solutions and communicate the offspring solutions to the master node. The master node decides which child will replace the parent using the diversity preservation metrics. Note that EAX is parallelized at the population-level. However, there are no known parallelization of EAX crossover operator because it is sequential.

The EAX code is publicly available. The EAX uses the real-valued vector, a tour permuation for the chromosome representation. The population is randomnly initialized and optimized using the 2opt local search. The 2opt code is optimized, uses the nearest neighbour, don't look bits concepts and the two-level tree [94] data structure. By default, the EAX generates 30 children on a population of size 300. There are two versions of EAX. One for smaller problem sizes, other for larger problem sizes. The difference is in the memory required to store the data structure. For larger problem sizes, the frequency table is limited to first 100 nearest neighbours.

2.6.3 Generalized Partition Crossover (GPX)

GPX was first introduced in 2009 [86] and have been improved [88, 87, 95] over the years. Partition Crossover has been used in solving other combinatorial optimization problems such as pseudo-boolean functions [96], MAXSAT [97, 98] and NK-landscapes [99]. The time complexity of the GPX operation is $\mathcal{O}(N)$. The code is written in C++ and is publicly available [1].

GPX exploits natural decompositions of the graph unioning two parents. This decomposition also linearly decomposes the evaluation function [86]. The union graph has vertices of degree 2, 3 and 4. The degree two and three vertices share edges common to both of the parents; common

[1] https://github.com/rtinos/gpx2

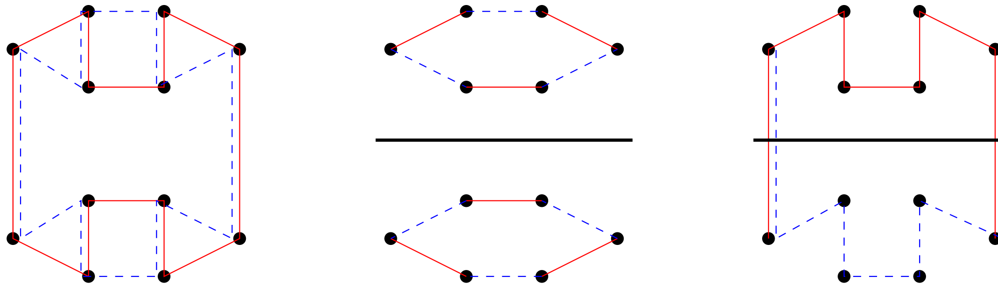


Figure 2.7: This figure illustrates a simple example of GPX. All of the vertices are of degree 3 and common edges are deleted: this guarantees the graph composes into AB-cycles. An AB-cycle is also a recombining component if the tours enter and exit the AB-cycle at the same vertices. For every recombining component, GPX calculates which parent has the shortest partial tour visiting all the vertices in the component.

edges are automatically inherited by the offspring. The degree four vertices do not have a common edge. However vertices of degree four are “split" into two vertices of degree three, connected by a common edge with zero weight. These newly introduced edges are also called as the **ghost edges**. Deleting the common edges will partition the graph into AB-Cycles [100]. Figure 2.7 shows a simple graph union of two parent tours on a twelve-city problem. All the vertices are of degree three. Deleting the six common edges results in two AB-cycles. Note that only two common edges separate the two AB-cycles. The remaining four common edges are found inside the AB-cycle.

We define each AB cycle as a **candidate component**. The common edges act as bridges between **candidate components.** This can be seen figures 2.7 and 2.8. We define a **recombining component** as a linearly independent subgraph of G_u that can be swapped between parents such that the resulting offspring are still Hamiltonian circuits. The recombining components also linearly decompose the evaluation function. A recombining component is made up of one or more candidate components. If a candidate component is connected to other components by exactly two bridges, then the candidate component is automatically a recombining component. Since these subgraphs are fragments of Hamiltonian circuits, one bridge is use to enter the recombining component and the other bridge is used to leave the recombining component. The vertices that connect to these bridges are referred to a *portal vertices*. A recombining component can be reached by more than two bridges, but there still must be an even number of bridges (clearly if a Hamiltonian circuit enters the component x times, it must also exit the component x times). Consider a candi-

date component that has four bridges and 4 portal vertices; denote these portal vertices by g_1, g_2, g_3 and g_4. If the solid (red) tour enters at g_1 and exits at g_2, and later enters at g_3 and exits at g_4, then in order to be a *recombining component* the dashed (blue) tour must also enter at g_1 and exit at g_1, then later enter at g_3 and exit at g_4. Because the parents are Hamiltonian circuits, both parent tours must visit all vertices during the two visits. Generalizing this idea, a *recombining component* must have an even number of portals, and if one tour enters at g_i and exits at g_j then the other tour must also enter at g_i and exit at g_j.

GPX first identifies the AB cycles as candidate components. It then determines which vertices are portals. (All of the other vertices can be ignored.) GPX then determines which *candidate components* are recombining components. If a candidate component is **not** a recombining component, GPX2 attempts to merge (or fuse) candidate components. First, GPX2 looks at pairs of adjacent candidate components (those with connecting bridges) and iteratively merges these pairs. Adjacent candidate components with more connections (more bridges) are merged first. GPX2 then checks to see if the new merged components are recombining components (with matched portals). This merging is repeated for a fixed number of steps.

A second type of fusion looks for *high level cuts*. For example, where there is only one bridge between two candidate components, there must exist another location somewhere in the graph G'_u where two candidate components are also connected by a single bridge. Finding pairs of single bridges and fusing the candidate components will create a new recombining component. Finally, if GPX has identified $q - 1$ recombining components, one can prove that the remaining components can always be merged to form one additional recombining component.

Figure 2.8 shows how GPX does the fusion. We ignore all of the vertices except the portal vertices. Subgraphs A and I and B are identified as recombining components, since the AB-cycles contain only two portals. The AB-cycle labeled B is nested inside of AB-cycle C. This can happen when AB-cycle B is recognized by a forward scan and AB-cycle C is recognized by a backward scan, thus yielding nested AB-cycles. While AB-cycle E and F look similar, AB-cycle E is recognized as a recombining component because the parent tours enter and exit at the same portals.

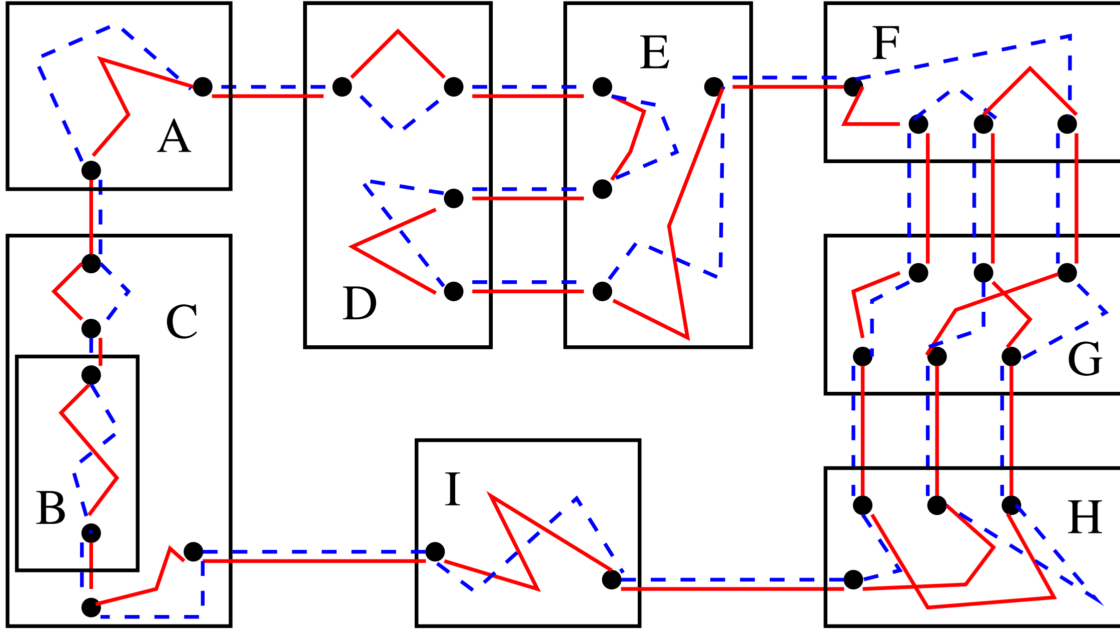


Figure 2.8: A more complex example of GPX. The boxes labeled from A to I are candidate components corresponding to AB-cycles. We can automatically identify components A, I and B as recombining components, since the AB-cycles contain only two portals (which function as an entry and an exit). After AB-cycle B is removed, AB-cycle C can now be recognized as a recombining component. AB-cycle D and E are both recognized as a recombining components because the solid tour and the dashed tour enter and exit at the same portals. The AB-cycles labeled F, G and H are not recombining components, but all three AB-cycles can be fused to create a single recombining component.

Component F is not a recombining component because the tours do not enter and exit at the same portals. The AB-cycles F, G, and H must be merged to create a feasible recombining component. After F, G, and H are fused, only two external portals remain. During recombination, GPX determines which parent tour has the shortest partial solution for each recombining component. The best offspring inherits the best partial solution for each recombining component.

GPX when used as a hybrid, improves the performce of EAX [90], LKH [1, 80], Concorde [91] and Chained LK [89]. This performance improvement is due to the highly exploitative nature of GPX and its ability to find the best out of 2^q solutions quickly. For example, if there are 30 partitions, then GPX can find the best out of 1 billion (2^{30}) possible solutions.

– **Hybrid GA using GPX versus Chained LK:** Whitley et. al [89] find that the hybrid GA using GPX outperforms chained-LK on five problems ranging from 500 to 1817. The hy-

brid GA consists of a population of 10, locally optimized by LK heuristic. Then, the GPX crossover operator is applied on the best tour and each individual (say, *tour(i)*) in the population. The GPX operator is modified to produce two children per crossover, by extension, 20 children in the population. The first child inherits the best partitions. The second child inherits all but one best partitions. The second child inherits the complement of the best partition for the longest component. By this way, two unique children are selected out of the 2^q possible solutions. A *diversity selection* metric that counts the frequency of edges in the population is used to determine which 10/20 individuals are to be retained in the population. If the GPX operator cannot produce an improving tour, i.e, no partitions are found, then the *double-bridge* mutation is performed on *tour(i)*. The best tour remains unchanged.

– **Hybrid with LKH:** The multi-trial LKH version uses the IPT and GPX crossover operator as discussed in section 2.5.2 of this chapter. Hains et. al [1] show improved results when using GPX with LKH. They experiment on six clustered TSP instances ranging between 3,000 and 30,000 cities. Tinos et. al [80] show results on a wide variety of 11 instances ranging in size between 3,000 and 115,475.

– **Hybrid with Concorde:** Concorde is the state-of-ther-art exact solver and it uses Chained-LK to improve the initial solution. Sanches et. al [91] use a hybrid of Chained-LK and GPX and show superior results on problems ranging between 1,000 and 5,000 cities. They also compare the results by replacing Chained-LK with LKH-2. For both Chained-LK and LKH, they store *m* local optima solutions either after T seconds or when a *soft restart* is made. The GPX is applied on this *m* set of population, similar to how it is used in the multi-trial LKH version. The results show that LKH with GPX improves the initial solution and reduces the runtime of Concorde on 13/15 instances.

– **Hybrid with EAX:** Sanches et. al [90] combined GPX and EAX in two ways. First, to improve the initial population along with Chained-LK. Results on ten TSP instances ranging in size between 5,000 and 13,500 cities show that population improved by GPX produces

results with a higher success rate. Second, GPX is applied on the 30 children generated by each EAX crossover operation. GPX used in this way reduces the population diversity because of its rapid improvements. Without sufficient diversity, EAX is prone to premature convergence. However, using a higher population on EAX (a size of 800 instead of the default 300) helps to preserve diversity and also take advantage of the benefits of GPX.

2.6.4 Iterative Partial Transcription (IPT)

The IPT is used as a recombination operator in the LKH solver. Given two parents, IPT first checks if a city has the same neighbor in both the tours. With this city as the starting point, a sub-sequence of length four or more is found by finding a matching end city. For example, in figure 2.9, city C has the same neighbor D in both the tours. Now, the fourth neighbor from this location is city E in blue tour and city G in the red tour. Since they don't match, the fifth location is checked. Now, the fifth city in both the subsequence match. Next, IPT checks if the cities in the sub-sequence are the same. Note that the permutation can be different. If yes, the sub-sequence lengths are compared, and the worst sub-sequence is replaced with the best sub-sequence. In figure 2.9, the blue sub-sequence replaces the red sub-sequence. This process is repeated, and finally, the better of the two tours is returned. The worst-case runtime of IPT is $\mathcal{O}(N^2)$.

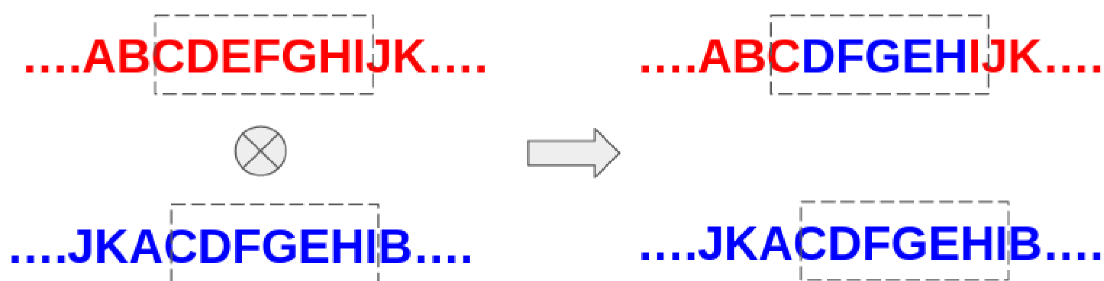


Figure 2.9: Figure showing an example of the IPT crossover operation.

2.7 Discussion

In this section, a summary of the parallelization capabilities of the TSP solvers and discussions on the implementation challenges of massively parallel GA based TSP solvers is presented.

2.7.1 Parallel TSP solvers

Table 2.1: List of state-of-the-art TSP solver and references to parallel implementations.

Solver	Category	Parallelization Level	Architecture	Scalability # Cities
Concorde	Exact	Search-space	CPU [53]	3B
2-opt	Local Search	Move-evaluation	CPU, GPU [50]	4461
		Move-evaluation, Swap-operation	GPU [101]	15,000
3-opt	Local search	Move-evaluation	CPU, GPU [50]	4461
POPMUSIC	Metaheuristic	Tour-segment	CPU [49]	10M
LKH	Iterated Local Search	Tour-segment	CPU[43]	3B
EAX	Genetic Algorithm	Population	CPU [93]	200,000
MAOS	Multi-agent Optimization	Agent	CPU [17]	6,000

Table 2.1 shows that the majority of the TSP solvers are designed for multi-core CPU architectures. The two local search solvers [50] [101] have been studied for the GPU architecture. Rocki and Sudha [50] evaluate each 2-opt or 3-opt move using a GPU thread. The best move is conveyed back to the CPU. The 2-opt or 3-opt swap is performed on the CPU. However, O'Neil and Burtscher [101] perform both the "move evaluation" and the "swap" operation using the GPU thread. Their results show a 3$\times$ speedup over the CPU implementation, and they could process 60-billion 2-opt moves per second on a single K40 GPU. Thus, the GPU implementation of O'Neil and Burtscher exposes maximum parallelization.

2opt and 3opt form an integral part of LK and LKH solvers. They are also used as a preprocessor to EAX solver. Therefore, having a highly parallel implementation of the local search solvers is useful. However, stand-alone local search methods do not guarantee the optimal solution. They need to be used in hybrid with global search solvers.

LKH is highly scalable, can find the near-optimal solution for up to 3B city instances. However, LKH works on one single tour. The parallelization techniques used for the k-opt algorithms can be applied for LKH. We also saw that LKH uses tour partitioning and merging techniques to handle larger instances. Nevertheless, the degree of parallelization of the LKH solver is limited.

Concorde uses local search heuristics in the pre-processing step and hence, exhibits some level of parallelization. The branch-and-bound techniques in Concorde also show parallelism by diving the search space across multiple CPU cores. However, inside each core, the algorithm is highly sequential. Therefore, the degree of parallelization of the Concorde is also limited.

The degree of parallelization for MAOS is directly proportional to the number of agents in the solver. The number of agents can range anywhere between 100 and 1000. The unique advantage of MAOS is its ability to combine several different solvers cooperatively. Therefore, the degree of parallelization depends on the solving ability of the agent. However, the current implementation has been tested only up to 6,000 city instances.

EAX is based on a genetic algorithm, metaheuristics inspired by Darwin's evolutionary theory. It is considered as the single best solver [16] on a wide variety of instances smaller than 2,000 nodes. There are several other parallel implementations of the population-based metaheuristic solvers, such as simulated annealing [102] and ant-colony optimization [103] methods. However, the performance is not as good as the genetic algorithm methods.

The population-based solvers exhibit higher degrees of parallelism. The search space, population, and operations on the population can be performed in parallel. For example, the multi-core CPU version of the EAX algorithm divides the population equally amongst 30 cores with ten individuals per core. The parallel EAX version can find optimal solutions up to 200,000-city instances [104]. The EAX crossover operator, however, is highly sequential and is not parallelized.

To summarize, the degree of parallelism of the existing TSP solvers is limited. The genetic algorithms exhibit higher degrees of parallelism. They can take advantage of the modern massively parallel architectures such as the GPU. In the next section, we discuss some of the existing GA-based TSP solvers and their parallelization capabilities.

2.7.2 Parallel GA solvers

There are several studies on the parallel implementations [105, 106] of GA-based TSP solvers. Most studies have divided the population among the parallel units and migrated individuals using different policies. This type of population-level parallelization is called the island model. Most of the GA-based TSP parallelization [107], [108], [20] is based on the island model.

Wang et al. [109] propose five different island model techniques. The two of the five parallel methods divide the population. However, the difference is in the migration policies. One method does not migrate individuals; whereas, the other migrates individuals. The third method is to divide the search space and perform the GA in parallel. The fourth and fifth method cuts a single tour and evolves them with and without migration respectively. Thus, they explore parallelism at the population-level, search-space-level, and operator-level. However, they tested the approach on small-size 100-city problems. So far, the parallel EAX has been the only successful GA-based solver, implemented on the CPUs, capable of solving up to 200,000 cities. Note that EAX uses the master-slave model as opposed to the famous island model parallelization.

There have been parallelization approaches on different computer architectures such as the Hypercube [22], FPGAs [23, 24] and GPUs [19, 18, 20]. The algorithms on hypercube and FPGA architecture are tested on problem sizes often less than 200. All these architectures use the island model paradigm to divide the population. However, the genetic operator used is sequential.

The GPU implementations have seen a steady rise in problem scalability over the years. Fujimoto and Tsutsui [18] implement a hybrid genetic algorithm with 2opt as their local search operator. The maximum number of cities they implemented was 500 on a population as small as 60. The authors report a 24.2$\times$ speed-up when compared to the CPU version. However, they were not able to reach the global optima. Zhang et al. [19] also use a hybrid GA and test up to 2392 city problems. But, they do not report the quality of solutions. In 2016, Kang et al. [20] report a 20$\times$ speed-up compared to their sequential counterpart on 100,000 city problems. They were also able to obtain the best-known solution to this problem. However, their report only includes seven problem instances, and the performance is not compared with state-of-the-art techniques.

Table 2.2: List of GA-based TSP solver and their properties

#	Genetic Operator	Parallelization Level	Architecture	Scalability	
				# Cities	Population
1	Edge Assembly Crossover	Population	CPU [104]	200,000	300
2	Alternating Recommendation Crossover	Population, Operator	GPU [20]	100,000	2048
3	Order-crossover	Population	GPU [18]	493	60
4	Partially Mapped Crossover, Roulette-Wheel Selection, Probability based mutation	Population	GPU [19]	225	200
5	Two-point crossover	Population	Hypercube [22]	105	1024
6	Heuristic crossover	Search-space, Population, Tour-segment	CPU [107]	100	128
7	One-point crossover, Flip-flop mutation	Population	FPGA [24, 23]	64	64

From table 2.2, we see that there are four unique parallelizations used in the literature. The population level is where the population is divided amongst the parallel units and is the most commonly practiced approach. However, the degree of parallelization for this approach is limited to the population size. The GPU implementation using the Alternating Recombination Crossover (ARX) operator [20] is parallelized at the operator level. Note that the ARX crossover operation is sequential. However, some functions such as finding the tour length, reversing the chromosome, copying parts of the chromosome are performed in parallel. Overall, the CPU implementation by Wang et al. [107] exposes the maximum level of parallelism. They could parallelize the population, divide the tour into segments as well as divide the search space.

To summarize this section, the GA solvers can explore at least three levels of parallelization. However, the current implementations do not take advantage of the massively parallel computing architectures. The EAX solver is successful; however, the degree of parallelization is just 300. The CPU implementation using the heuristic crossover operator exposes several levels of parallelism. However, the scalability is limited. Therefore, there is a need to design a solver that could expose maximal parallelism and scale and perform well. In the next section, we discuss the design challenges and properties of such a solver.

2.7.3 Design of massively parallel GA based TSP solver

Tables 2.2 and 2.1 enumerated some of the existing parallelization techniques used by the TSP solvers. Seven different parallelization techniques are proposed in the literature. However, the Genetic Algorithms are the ones capable of exhibiting maximal parallelization. Higher degrees of parallelization are favorable for NP-Hard problems mainly because the convergence rate can be fast when many solutions are searched in parallel. Also, note that the Edge Assembly Crossover (EAX) algorithm is the single-best solver on a wide variety of problems. However, the EAX crossover operation is sequential. Therefore, it is worth researching the design of GA-based TSP solvers capable of maximally utilizing parallel computer hardware capabilities.

The multi-core parallel version of EAX is best suited for the CPU architecture. However, there are several challenges when porting them to the SIMD architecture. From Chapter 1 and Appendix A, we infer that an ideal solver must satisfy both the algorithmic and architectural properties. The two major architectural features that affect the *scalability* and performance of the solver are the *memory* and *communication* costs. The main algorithmic feature to consider is the convergence and *success rate* of the solver. The convergence rate is measured in terms of *execution time* required to reach the global optima. However, suppose the solver is not designed for speed. In that case, metrics such as *number of generations*, *number of genetic operations* can be considered.

The EAX crossover is designed for speed and thus, outperforms the rest of the solvers in terms of convergence and success rate. However, it does so by maintaining large data structures. Each EAX operation takes about 100N memory. However, other genetic operators in table 2.2 are not memory intensive. We can say this based on the description of the operators. However, the actual implementation is not publicly available. From the publicly available GPX operator, we can say that it consumes 25N memory per crossover operation, a $4\times$ reduction, compared to EAX.

Let P be the population size, N be the problem size, and 30, the number of offspring per crossover operation. The worst-case communication cost per EAX generation is $\mathcal{O}(PN)$. These memory accesses are irregular and is not favourable for the SIMD machines. However, not all genetic operators in table 2.2 incur these communication costs. The two-point crossover operator

makes two cuts in the parent tours and exchanges the segments. Similarly, a one-point crossover makes a single cut and exchanges the segments. The ARX crossover is similar to a two-point crossover. However, the heuristics to make the cuts are different. The partially mapped, order crossover and the heuristic crossover introduce new edges in the offspring. The GPX crossover makes q cuts and does not introduce new edges. The IPT crossover operator exchanges subsequences and does not introduce new edges. Thus, the choice of crossover operator determines the communication costs. Note that all mutation operation must change the permutation of the tour by deleting and adding edges. Therefore, there is no way to escape the communication costs when using a mutation operator.

The operators that inherit all alleles from the parents are termed *respectful*. Based on our previous discussion, the one-point, two-point, ARX, IPT, and GPX operators are all respectful. All these operators cut the parent tours and exchange the segments. Therefore, their communication cost during crossover is zero. However, their effectiveness is directly proportional to the number of cuts. The more number of cuts, the more solutions are searched for improvements. The one-point makes one cut, two-point and ARX makes two cuts. GPX makes q cuts, and in most cases, q is greater than 2. GPX operator can make more cuts than IPT [80].

From the above discussions, we can say that the GPX operator has ideal properties that satisfy the architectural properties. GPX, when used in hybrid with LKH [80], has been tested on 115,475 city problems. GPX has also been used to improve the initial population of EAX, usually size of 300. However, as a stand-alone GA [89], the operator is tested only on problems smaller than 1,817 cities and on a population of 10. The hybrid of GPX and EAX is prone to premature convergence; however, a larger population helps to preserve diversity. A larger population is the desired property when considering massive parallelism.

To summarize our discussions, the GPX operator has the desired properties to utilize the massively parallel computer architectures. This thesis considers the GA solver using the GPX crossover operator as our ideal solver. In the following chapters, we present our results on improvements made to its scalability and parallelization capabilities.

Chapter 3

Mixing Genetic Algorithm

The Edge Assembly Crossover (EAX) is one of the state-of-the-art TSP solvers. It is found to be the single-best solver [16] for a wide variety of problems smaller than 2,000 cities. However, there is no known comparative study made for larger problem sizes. The multi-core version of EAX can solve up to 200,000 city problems. However, this version of EAX does not fully exploit the inherent parallelism. In this thesis, we want to develop a GA solver that maximizes intrinsic parallelism and is efficient for larger problem sizes.

The discussions in chapter 2 showed that the Generalized Partition Crossover (GPX) consumes $4\times$ lesser memory compared to the EAX. Furthermore, GPX is a deterministic crossover operator, inherits the edges present only in the parent tour. Owing to this, the GPX does not access the cost matrix to create new edges. GPX has been used in a hybrid mode with LKH [80], Concorde [91] and EAX [90]. The hybrid of GPX and EAX [90] is prone to premature convergence; however, a larger population helps to preserve diversity. A larger population is the desired property when considering massive parallelism. These features make GPX operator favourable for multi-core CPU machines and massively parallel SIMD machines.

The primary contribution of this Chapter is the Mixing Genetic Algorithm (MGA) for the Traveling Salesman Problem (TSP) that uses the GPX operator in a new way. The MGA is implemented on the shared memory CPU architecture using the OpenMP parallelization framework. The MGA is compared with EAX based on eight different metrics: memory costs, communication costs, execution time, number of generations to converge, number of recombinations to converge, and success rate on different TSP instances, scalability in terms of problem and population sizes. Part of this Chapter is published in the GECCO 2019 [25] conference.

The chapter is organized as follows: Section 3.1 presents the modified GPX operator. Section 3.2 explains how partitions are mixed in the population. Section 3.3 explains the working of MGA. Section 3.4 presents the experimental setup and results. Finally, section 3.5 concludes the chapter.

3.1 Generalized Partition Crossover

In Chapter 2, the basic functioning of GPX was explained. Normally, GPX is used to construct one offspring. However, in this Chapter, we modify the operator to produce two offsprings. Figure 3.1 presents the basic ideas behind the modified partition crossover that generates two offsprings.

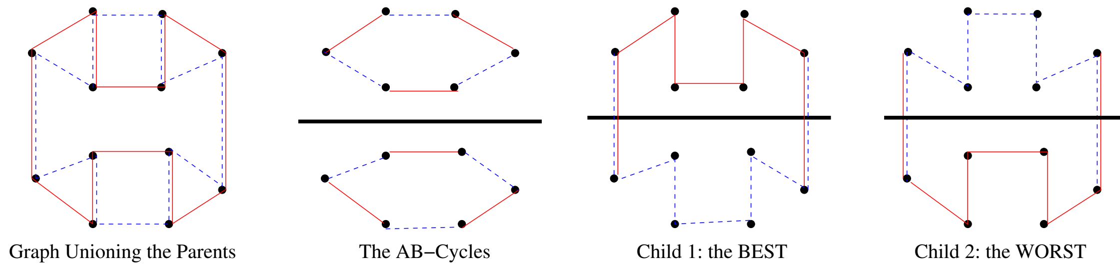


Figure 3.1: Simple GPX with one crossover opportunity and two recombining components. One parent is represented by the solid (red) edges. The other parent is represented by the dashed (blue) edges. If Child 1 represents the best possible offspring, then Child 2 must be the worst possible offspring generated by GPX.

GPX first unions two parent solutions to create a union graph G_u, which is made up of all of the edges in the two-parent solutions. The two parents are represented by the solid (red) circuit and the dashed (blue) circuit. In Figure 3.1 there exists a cut that breaks G_u into two linearly independent subgraphs. The subgraphs are found by deleting all of the common edges. We first note that the graph G_u can be transformed into a related graph G'_u where all of the vertices in G'_u have degree 3. Because the vertices in G'_u have degree 3, the subgraphs will always form AB-cycles. Assume the two parents are labeled **A** and **B**; an AB-cycle is a subcircuit of G'_u such that the vertices are ordered: $v_a, v_b, v'_a, v'_b, ..., v''_a, v''_b$ where v_a is a vertex from Parent **A** and v_b is a vertex of Parent **B**. Note in Figure 3.1, if one parent is the “red" tour and the other parent is the “blue" tour, then the edges in the AB-cycles have alternating colors. GPX can now independently pick the solid (red) or the dashed (blue) partial solution in the two subgraphs. GPX can also pick the *best* partial solutions from each subgraph that make up G'_u and G_u. This greedy strategy can be extended to cases where G'_u is broken into q independent subgraphs. By picking the best and the worst partial solution for each subgraph, GPX returns the best and the worst of 2^q reachable offspring.

3.1.1 Communication costs

The GPX code constructs two initial parent solutions. To calculate the length of the tour, the memory is accessed $2N$ times to fetch the *edge distance*. However, after the initial calcuation of the lengths, there is no need to access the cost matrix because, the edge distances are stored along with the tour permutation. Thus, the number of memory reads during the crossover is **zero**.

3.1.2 Memory costs

The memory costs of GPX operator (taken from the publicly available code [100]) remains the same as discussed in Chapter 2. However, with the modified version, the costs increases by $2N$ to store the edges of the tours. The codes are modified in the offspring generation stage to produce two offsprings. So, the total memory cost is $27N$.

3.1.3 Premature Convergence

Since the modified GPX inherits all the edges from the parent tours, there is no loss of diversity in the population. Therefore, the GPX used in this way is not prone to premature convergence.

3.2 The Mixing Population

In the previous section, GPX operation on a two-parent tour was discussed. Let us consider a population of size $popsize$. Now, the selection of the appropriate parents to recombine is a critical decision. The selection strategy was motivated by the following goals.

1. Since GPX is deterministic, the same pair of parents should not recombine more than once.

2. The best partitions in all the individuals must recombine quickly.

3. Reduce the randomization and communication costs.

The first and second goal helps to improve the convergece rate by cutting down on the number of unnecessary recombinations. The third goal helps to design a solver that would effeciently make use of the hardware architecture for performance gains.

The use of a hypercube configuration allows the population to display a form of emergent internal migration. Figure 3.2 shows the mixing of the population of size 16. The single mixing of the population consists of four steps. In general, one mixing consists of $log(popsize)$ steps. Each step is called the generation. The set of $log(popsize)$ generations is called an **epoch**.

In the first generation, every individual at location j in the first half of the population (i.e., $0 \leq j < 2^{c-1}$) is recombined with an individual at location $j + 2^{c-1}$. Thus in a population of 1024, individual 0 is recombined with individual 512, and individual 511 is recombined with individual 1023. The best child replaces the individual with index j, and the worst individual replaces the individual at location $j + 2^{c-1}$. The best offspring are now all located in the first half of the population. We can apply the standard hyperplane notation to the strings indexing the population. For a population of size 2048, let 0********** denote addresses in the first half of the population and let 1********** denote address in the second half of the population.

In the second generation, we again use the hyperplane notation to denote that individuals addressed by *0******** are recombined with individuals addressed by *1**********. Thus, individuals at locations $0 \leq j < 2^{c-2}$ and $2^{c-1} \leq j < 2^{c-1} + 2^{c-2}$ (the first quarter and third quarter of the population) are now recombined with individuals at locations addressed by $j + 2^{c-2}$.

Generalizing this hyperplane notation:

Generation 3: all individuals addressed by **0*******
are recombined with individuals **1******

Generation 4: all individuals addressed by ***0******
are recombined with individuals ***1*****

Generation 10: all individuals addressed by *********0
are recombined with individuals ********1

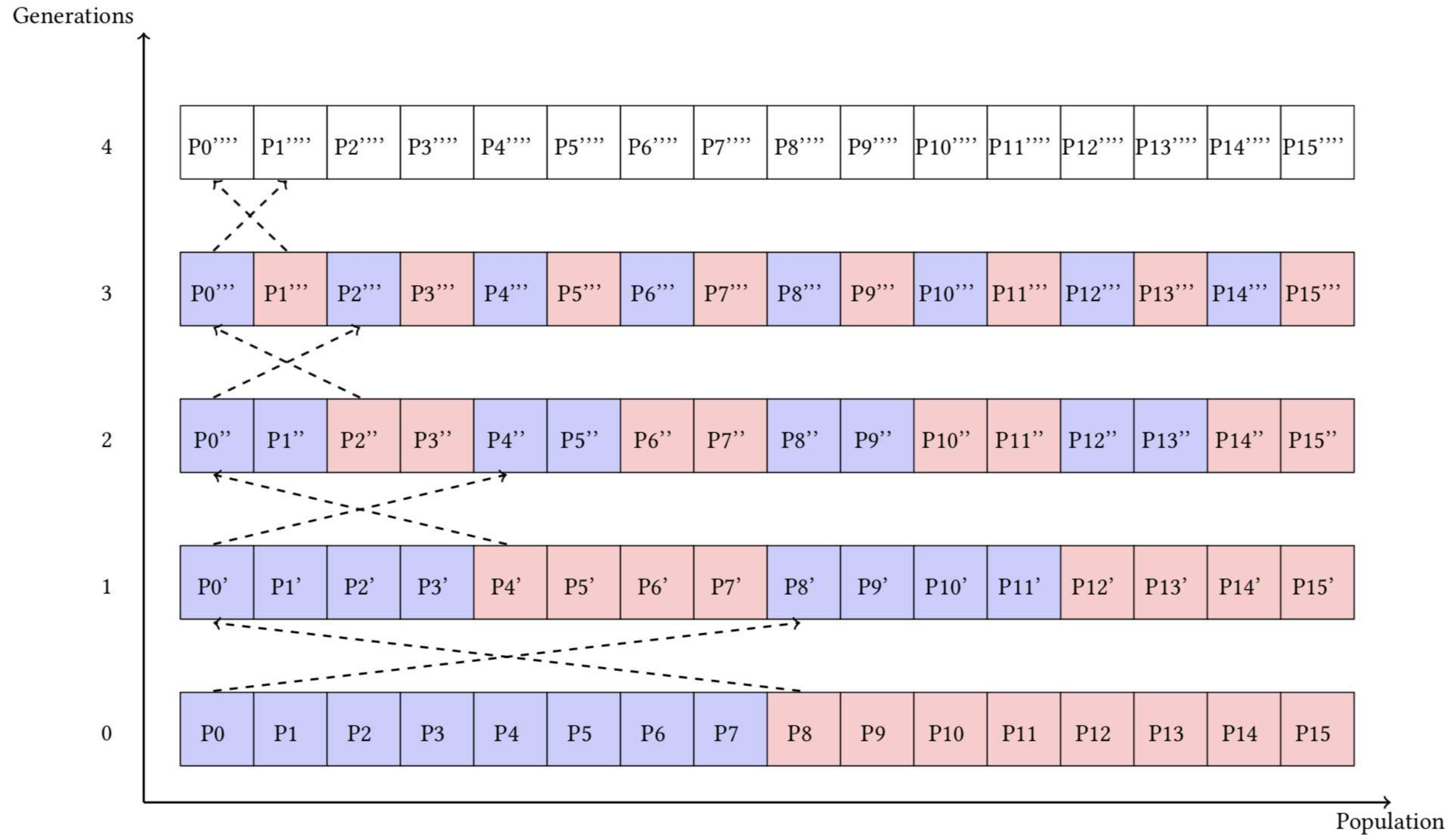


Figure 3.2: For a population size 16, there are four generations in one epoch of the Mixing Genetic Algorithm. In generation one, parents from P0 to P7 are recombined with parents P8 to P15 to produce the best offspring (P0' to P7') and worst offspring (P8' to P15'). The pattern is color-coded where blue is the first parent and red is the second parent. The crossover arrows are shown for some recombinations. At the end of an epoch, P0 is usually the best solution in the population, and P15 is usually the worst solution.

This recombination pattern moves the best offspring to locations in the population with fewer bits in the address, and also carries the best edges in the population toward location 0. This recombination pattern also uniformly mixes the population.

Consider the history of recombinations for location 0000 in a population of size 16. The population is numbered 0 to 15, and the final offspring at address 0000 after four recombinations (one epoch) is given by the following set of recombinations:

([(0x8)x(4x12)]x[(2x10)x(6x14)]) x ([(1x9)x(5x13)]x[(4x12)x(7x15)])

Note that every member of the population (from 0 to 15) is involved. In general, in every **epoch**, all individuals in the population can contribute the best components to position 0 in the population. In the first generation, edges can move from the second half of the population to the first half. In the second generation, edges can move from the second quarter of the population to the first quarter of the population.

3.2.1 Genetic invariance

At the end of the epoch, all individuals in the population have had the opportunity to contribute its best partition to P_0. Similary, all individuals in the population has had the opportunity to contribute its worst partition to P_{N-1}. The edges associated with the individuals have migrated within the population. However, none of the edges are lost. Thus, the total edges (genetic material) before and after an epoch remains unchanged, achieving **genetic invariance**. Figures 3.3 and 3.4 illustrate the genetic invariance on one epoch of att532 instance. The observation holds true for all instances and on all epochs. However, the population size must be a power of two.

For example, consider a population of size 10. In the first generation, the first and second halves are recombined. However, in the second generation, the population is divided into four. At least two individuals will not recombine because the code is written to mix the population in a hypercube topology (check listing 3.1). In the third generation, the population is divided into eight. Again, two individuals will not take part. These left out individuals may have the best evaluation. Therefore, for the best individual to migrate to the top, the population size must be a power of two.

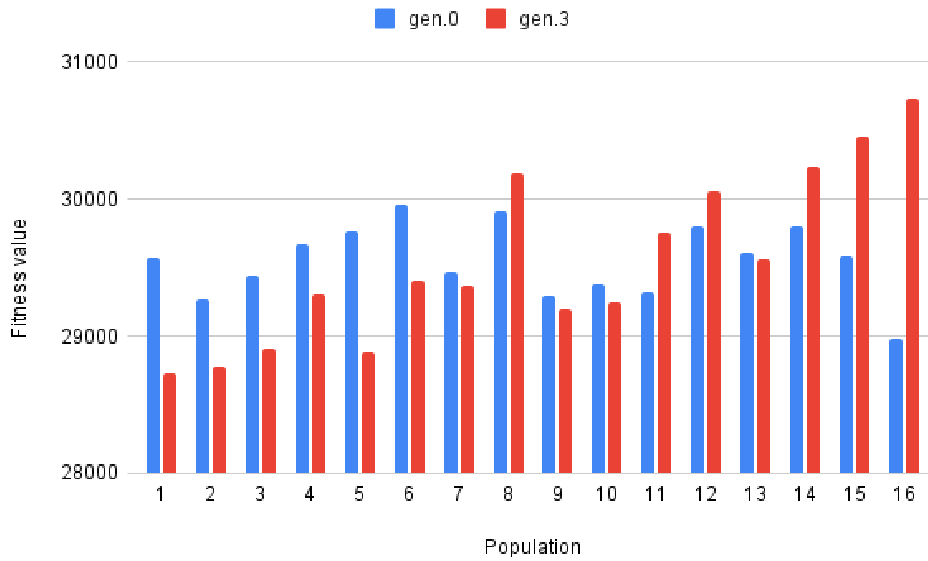


Figure 3.3: The bars show the fitness value of 16 individuals at the end of first and last generations of an epoch. It can be seen that the first individual has the best and the last individual has the worst fitness value.

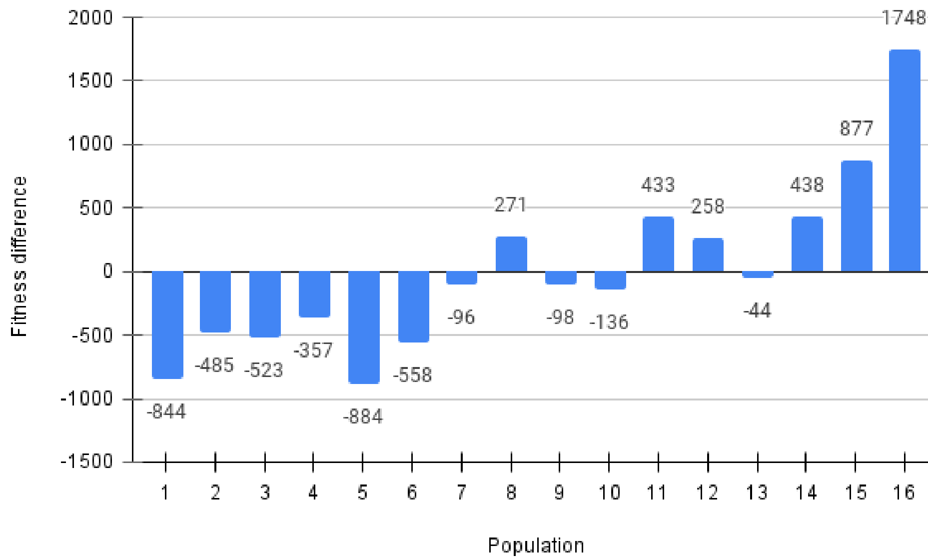


Figure 3.4: Results on running att532 on a population of size 16. The bars show the difference in fitness value of first and last generations of an epoch. Adding all the values produces **zero**. This shows that at the end of an epoch, the tour evaluations are different from that of first generation. However, the total evaluation of the popualtion remains constant, thereby achieving genetic invariance

3.2.2 The Mixing Theorem

Theorem 1 (The Mixing Theorem): *After each* **epoch** *(and c generations) of the Mixing GA, all individuals in the population have the potential to contribute their best components to the individual at location 0.*

Proof: The proof is inductive on c. For basis populations of size 2, 4, 8, 16, the property holds. Assume the property holds for a population of size $2^c/2$ (the inductive hypothesis). Double the population size to 2^c. In the first generation of the epoch, all population members in the first half of the population (locations $0 \leq j < 2^{c-1}$) are recombined with the population members in the second half of the population (each individual at location j is recombined with the individual at location $j + 2^{c-1}$). The best offspring are now in the first half of the population. The inductive hypothesis covers the remaining $c - 1$ generations. $\square$

A simple corollary of Theorem 1 is that all individuals in the population have the chance to contribute to every position in the population. The xor ($i \oplus j$) operator applied to any address in the population (denoted by i) and any fixed reference address (denoted by j) performs an affine on the address space. Thus, $\forall i, (i \oplus j)$ remaps address the entire address space relative to j. Therefore, the "mixing" property in Theorem 3.1 holds for all addresses. However, the mix of components that are inherited are different: 1) address 0 only inherited the best components, 2) address $2^c - 1$ inherits only the worst components, and all other addresses inherit a combination of best and worst components from the entire population. The ratio of zeros and ones in address j indicated how often that address inherits the best components compared to poorer components.

3.3 The Mixing Genetic Algorithm

At the end of an epoch, the best individual migrates to the first position in the population. However, this does not guarentee that the individual is the global optima solution. Therefore, there is a need to further mix the edges in the population. The epochs are repeated until the desired optimal solution is found. Since, the GA is designed to just mix the edges in the population,

Algorithm 1 The Mixing Genetic Algorithm

1. Randomly initialize the population.
2. Apply local search on the population.
3. **while** (termination is not met)
4. Apply one epoch of The Mixing Genetic Algorithm.
5. Randomly shuffle the new population.
6. **end while**

we called this as the **Mixing Genetic Algorithm**. Listing 1 shows the design of the algorithm. The population is randomly initialized (line 1). The 2-opt local search operator is applied on this population (line 2). Please refer Chapter 2 for a detailed working of 2opt.

3.3.1 Limitation of MGA

The edges in the initial population remains unchange because MGA does not introduces new edges. Therefore, it runs into risk of not being able to find the global optima if the initial population does not contain all the edges that make up the global optimal solution. As we started to generate results, we explicitly tested to see if the necessary edges are present in the initial population. Table 3.1 shows the smallest population size that contains all the edges found in the globally optimal solution. We use the notation P* to represent a globally optimal solution.

A total of 25 TSP instances in table 3.1 is considered for analysis. The first three of the cluster problems are taken from the 8th DIMACS TSP Challenge[1]. The fourth cluster instance, the Printed Circuit Board (PCB) problems, and the city problems are taken from TSPLIB database [110]. The letters in the instance tell us the type of problem and the number tells us the problem size. For example, pcb442 means, it is a Printed Circuit Board (PCB) problem with 442 cities. It can be seen from table 3.1 that for 21/25 instances, a population as small as 256 is sufficient to hold all the P* edges. However, for C3k.1, fl1577, d2103 and vm1084, a higher population is required.

[1] http://archive.dimacs.rutgers.edu/Challenges/TSP/

Table 3.1: List of problem instances and the corresponding smallest population size that has all the edges found in P* after performing EAX based 2-opt on the initial population.

Cluster Problem	Pop Size	PCB Problem	Pop Size	City Problem	Pop Size
C1k.0	256	pcb442	16	att532	64
C3k.0	256	d657	256	pr1002	128
C3k.1	512	pcb1173	256	vm1084	512
dsj1000	128	d1291	128	rl1304	64
		fl1400	128	rl1323	256
		fl1577	4096	nrw1379	128
		d1655	256	vm1748	128
		u1817	128	rl1889	128
		d2103	16384	pr2392	128
		u2319	64	fnl4461	256
		pcb3038	128		

Consider the d2103 instance. Theoretically, because d2103 is a symmetric TSP instance, a population of size 1,052 is sufficient to hold all the edges in the tour. This is because, the number of unique edges for a symmetric instance is $(N^2/2) + N$. Therefore, the number of unique edges for a 2,103 sized problem is $((2103*2103)+(2103)/2 = 2,212,356$. The number of edges present in a population of size 1,052 is $1052 * 2103 = 2,212,356$. However, a population of size 16,384 was needed to contain all the global optima edges. This is because there is a bias in the 2-opt code that impacts which edges are likely to appear in the initial population.

The 2-opt code we use is based on the nearest neighbours concept. On further inspecting the population, we found out that the global optima edges for 11 instances are found in the first 50 nearest neighbors. For the remaining 14 instances, more than 99% of their global optima edges are in the 50 nearest neighbor list. These observation demonstrates that 2opt using nearest neighbor heuristics produces a population with abundant global optima edges.

It has to be noted that the instances we consider have known global optimum. However, in reality, the instance need not have a known global optima. In such cases, we cannot predict the population size. Thus, the limitation of MGA is that, the convergence of the solver is possible if and only if the initial population contains all the P* edges.

3.3.2 On Convergence and improving moves

Now, since the initial population is rich in P* edges, the goal of MGA is to simply keep mixing until termination. Thus, an epoch is repeated in a while loop. See line 4 in listing 1. However, by after applying four or five epochs, the solver might not produce an improving move. An improving move is found when the child tour has a better evaluation value in comparison with the parent tour. The GPX operator is deterministic and it cannot find a new improving move when the same two parent solutions are recombined. Again, figure 3.5 demonstrates this aspect on 25 problems for a population of size 256. It can be seen in figure 3.5 that for all problems, the solver finds an improving move upto five epochs. For some instances, improving moves are found upto 8 epochs. However, beyond that, there are no improvements. This trend is true for all population sizes.

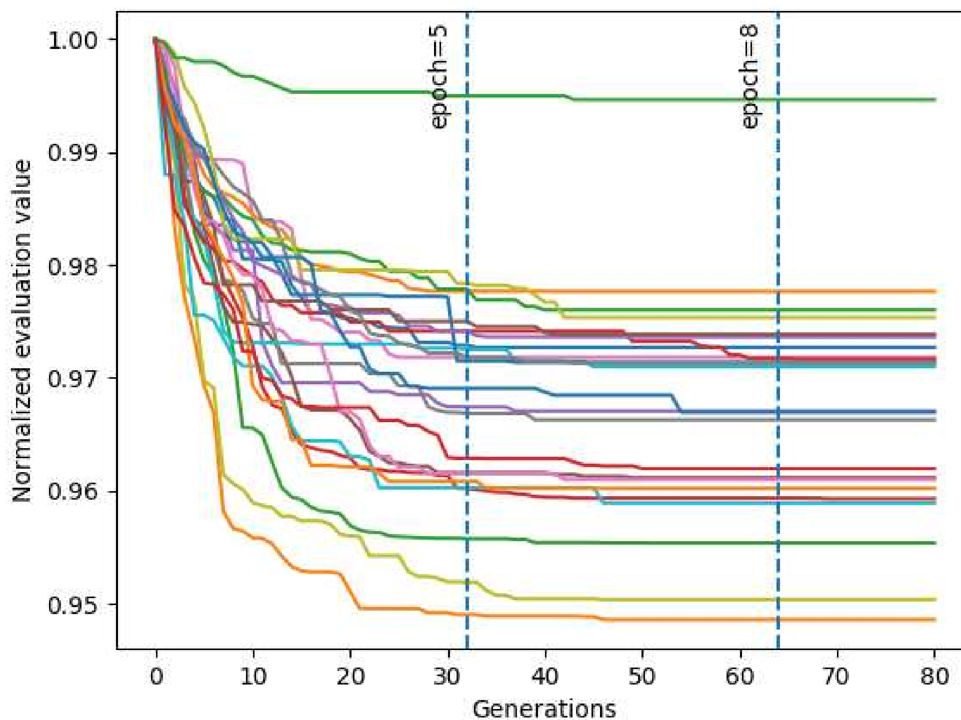


Figure 3.5: MGA does not produce an improving move after continuously applying four or five epochs without any randomization of the population.

Table 3.2: Percentage of improving moves found by running the MGA on a population of size 1024.

Instance	%	Instance	%	Instance	%
rl1323	51.16	pcb3038	54.19	C3k.0	69.31
u2319	51.2	fnl4461	56.55	att532	71.78
fl1577	51.32	nrw1379	57.23	d1291	75.86
pcb1173	51.84	rl1889	57.97	vm1084	77.86
u1817	52.95	pcb442	59.29	d657	79.04
pr1002	53.07	vm1748	60.42	C1k.0	82.21
C3k.1	53.74	dsj1000	60.55	fl1400	88.34
d2103	53.81	d1655	67.99	rl1304	89.39
		pr2392	68.92		

To overcome the stagnation, we randomize the population at the end of every epoch. Check lines 3,4,5,6 in listing 1. Randomization decreases the possiblity of the same parent tours recombining. However, it does not guarentee that the solver will not stagnate at a later stage. Table 3.2 tells the percentage of improving moves. It can be seen that for 13/25 instances, the improving move is less than 60%. However, for 3/25 instances, the percentage is higher than 80%. The improving moves is a direct function of the edges in the tour. Table 3.2 demonstrates that the performance of the solver varies with the problem type and problem size.

Note that MGA can be designed to randomize the population once every two or more epochs too. The default is to randomize at the end of every epoch. Thus, the resulting MGA using the GPX operator does not lose any diversity in the population. Unlike the previous implemenations, the GPX introduced in this thesis does not suffer from premature convergence.

3.3.3 Termination condition

MGA terminates when it reaches the earliest of two criteria (a) known global optima or, (b) x number of generations. Here, we chose x=10,000, an arbitarily value, so that the solver does not run for a long time. For problems, for which the global optima is not known, the Held-Karp (HK) bounds can be used for termination. Please refer chapter 2 for a review on calculating HK-bound.

Listing 3.1: The Mixing Genetic Algorithm – Parallel Loops

```
1 for(int i=1;i<= steps;i++){ //steps = Generations/epoch
2   int twoPowerI = pow(2,i);
3   #pragma omp parallel for if(twoPowerI>5)
4   for(int j=1;j<= (twoPowerI/2);j++){ // Hyperplane loop
5    #pragma omp parallel for if(twoPowerI<6)
6    for(int k=1;k<= N/twoPowerI; k++){ // Crossover loop
7     .....  }}}
```

Figure 3.6: The Mixing Genetic Algorithm – Parallel Loops

3.3.4 Parallelization

Population-level parallelization using the openMP parallelization paradigm is applied to the "for loop" that encloses the crossover operation. However, since MGA mixes the population in an hybercube topology, the "openMP pragmas" are placed appropriately.

Figure 3.6 shows a code snippet from the MGA code. The outer loop is the generation loop. The number of steps denote the number of generations per epoch. The "j" loop is the hypercube dimension. Consider an example, when $i = 1$ (generation=1), $j = 1$. There is one hyperplane, diving the space into two. However, inside this hyperplane, we have $k = 1, 2, ..., N/2$, a total of crossover $N/2$ operations. The Hamming distance between the parents is $N/2$. When $i = 2$, $j = 1, 2$. Therefore, in the second generation, there are two hyperplanes. Inside each hyperplane, we have $k = 1, 2, ..., N/4$, a total of $N/4$ crossover operations. The Hamming distance between the parents is $N/4$. Thus, with generations, the hamming distance reduces.

The openMP paradigm is used for a "Shared Memory (SM)" architecture. A typical configuration of a shared memory is shown in figure 3.7. Here, the parallel units (a.k.a, processors) share a single main memory. The population is divided amongst the processors. There is no explicit dividing of the population. Each core has one or more threads associated with it. In listing 3.1, line 3, the "pragma omp parallel" asks the threads to compute the operations in parallel. The number of threads is fixed at runtime. The default "static" scheduling with "no chunck size" is used. This means that, the population is approximately equally divided amongst the given number of threads.

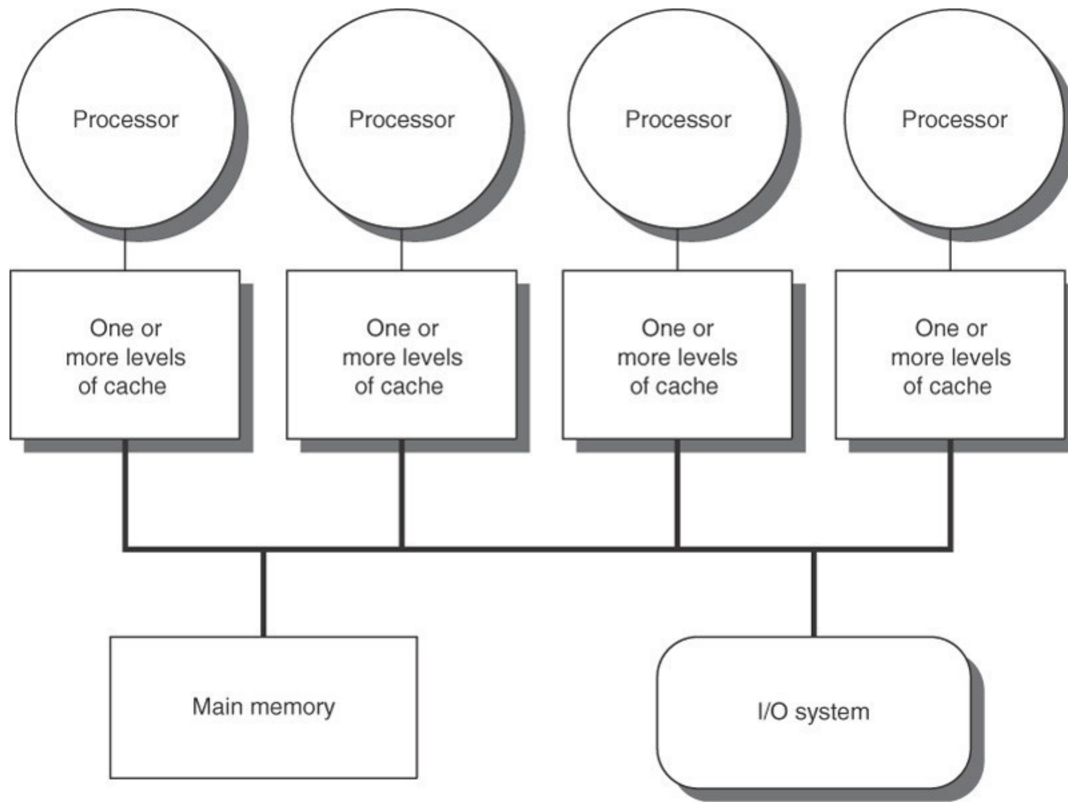


Figure 3.7: Figure illusrtating an example of a shared memory architecture.

In listing 3.1, note that the k loop is parallelized only when the Hamming distance is greater than $N/6$. The number of individuals in the k loop is equal to the Hamming distance. The machine we used has 6 processors and parallelization makes sense only when the number of individuals per processor is large enough. However, when the Hamming distance is lesser than $N/6$, the parallelization happens at hyperplane level. Therefore, two levels of parallelization are explored (a) hyper-plane level and (b) points inside hyperplane. At any given point, only one level of parallelization is realized depending on the Hamming distance.

This population-level implementation has no “migration” in the traditional sense. However, at the end of every epoch, the population is randomized. If the individuals are stored in the local “cache”, the memory system automatically transfers them to the appropriate processor. The programmer need not specify these communication commands.

3.3.5 Summary

Table 3.3 summarizes the parameters used in the design of MGA discussed in this section.

Table 3.3: Design parameters of Mixing Genetic Algorithm with openMP parallelization

Parameter	Comments
Initialization	2opt local search. Refer chapter 2 for 2opt parameters Time is used as a random seed
Randomization	once every epoch One epoch = $log(popsize)$ generations
Genetic operator	Generalized Partition Crossover (GPX) Selection of parents for mating follows hypercube topology
Termination	Earliest of the below three: (a) known global optima (b) 10,000 generations (c) HK lower bound when global optima is not known
Population-level Parallelization - Shared Memory architecture	
Parallel Programming model	OpenMP
Parallel architecture	Single multi-core machine
Number of parallel units(P)	6 processors and 12 threads
Sub-population size(I)	$popsize/P$, determined at run-time
Topology	Hypercube
Migration Policy	none.

3.4 Experimental setup and results

The MGA in listing 1 is implemented in C++ with openMP parallelization. It uses the modified version of the publicly available GPX implementation taken from R. Tinós [100]. We use the EAX solver as the standard benchmark to compare and evaluate the performance of MGA. The EAX code is from Nagata and Kobayashi [15]. The machine used is Intel Xeon CPU E5-1650 v4 running at 3.60GHz. It has 6 cores and 12 threads with hyperthreading. EAX and MGA codes are compiled using icpc with -O3 flag. For the parallel MGA version, -qopenmp flag is used. The experiments are run for 30 trials. Tables 3.4 summarizes the results.

Table 3.4: Results on applying MGA to 25 problem instances to two population sizes (1024, 16,384). A population of 16,384 on MGA is comparable to EAX on a population of 300 with 30 children per recombination. Note that for fl1577, the minimum population required to store all P* edges is 4096. Therefore the results reported under MGA-1024 column is for a population of 4096.

Problem	MGA-1024			MGA-16384			EAX		
	Generations		S.R	Generations		S.R	Generations		S.R
	median	avg		median	avg		median	avg	
C1k.0	171	181.33	30/30	57	58.86	30/30	264.5	265.16	30/30
C3k.0	2466	3821.33	24/30	323	319.26	30/30	652.5	650.36	30/30
C3k.1	10k	9.2k	5/30	57	1583	30/30	690.5	691.43	30/30
dsj1000	186	1167	27/30	57	64.93	30/30	237	236.83	30/30
pcb442	696	3095.64	21/30	71	66.8	30/30	57	60.9	30/30
d657	186	209	30/30	64	64	30/30	171	170.63	30/30
pcb1173	10k	9100	4/30	554	572.2	30/30	230	230.2	30/30
d1291	71	85.67	30/30	43	41.13	30/30	153	153.6	30/30
fl1400	51	64.33	30/30	29	27.55	30/30	194	186.86	30/30
fl1577*	3721	3757	30/30	624	878	30/30	196	180.46	12/30
d1655	591	693.66	30/30	183	184.86	30/30	232	231.86	30/30
u1817	10k	9780	2/30	1296	1526.06	30/30	225.5	229.2	18/30
d2103				1289	3074.93	27/30	174	174	30/30
u2319	10k	10k	0/30	10k	10k	0/30	238	236.26	0/30
pcb3038	10k	10k	0/30	5104	5085.33	30/30	468.5	468.33	30/30
att532	281	308.67	30/30	71	75.67	30/30	164	163.6	30/30
pr1002	381	428	30/30	99	104.6	30/30	229.5	229.86	30/30
vm1084	466	2078.66	26/30	57	58.4	30/30	181	180.73	30/30
rl1304	61	58.33	30/30	29	28.06	30/30	164	163.33	30/30
rl1323	10k	9037	3/30	113	442.46	29/30	172.5	172.9	30/30
nrw1379	10k	8263.66	10/30	1177	1341.73	30/30	301	299.86	28/30
vm1748	531	580.66	30/30	148	144.26	30/30	238.5	237.36	30/30
rl1889	1121	1701.66	30/30	239	265.6	30/30	213	212.46	30/30
pr2392	10k	10k	0/30	8191	6821.41	19/30	295	298.2	30/30
fnl4461	10k	10k	0/30	10k	10k	1/30	785.5	796.1	28/30

3.4.1 Initial population

The default population size of EAX is 300. Two random parents, say parent A and parent B are recombined to produce 30 offsprings. All parents participate as parent A in one recombination and parent B in another recombination. Thus, for a population of 300, we have 300 recombinations per generation. In total, EAX evaluates 9,000 recombinations (300 X 30) per generation. One recombination of MGA produces two children. However, the two children are complementary to each other. Therefore, we consider it as a single recombination. Therefore, for a population of size N, the number of recombinations per generation of MGA is $N/2$.

The closest number of hypercube nodes to 9,000 is 8,192. For MGA, a population of 16,384 yields 8,192 (16,382/2) recombinations per generation. Note from table 3.1, the minimum population required to hold the P* edges for all the problems is 16,384. Thus, chosing 16,384 for MGA makes sense. It is known that EAX converges slow with larger population size. Therefore, choosing a higher population size for EAX is not recommended. Therefore, population of 300 for EAX and 16,384 for MGA are comparable. However, there can be bias in the performance of the solver because of the properties of the population. We discuss this aspect in chapter 4.

3.4.2 Population size scalability

We compare the performance of MGA on two population sizes. For fl1577, we present the result on a population of size 4,096 instead of 1,024 because 4,096 is the minimum population size to hold all P* edges. Similarly, the 1,024 results for d2103 is skipped. From table 3.4, the number of generations decreases with increasing population size. This is a desirable property when implementing a genetic algorithm on a massively parallel machine. For EAX and for most genetic algorithms, a larger population typically results in slower convergence.

Note from table 3.4, that higher population size also increases the success rate on 12/25 instances, seven of which achieve 100% success. For one instance, u2319, the success rate is the same (=0) on two population sizes. Both the population sizes achieve 100% success rate on 12/25 instances. Therefore, increasing the population size either improves or maintains the performance.

3.4.3 Problem size scalability

From table 3.4, the 16,384 MGA population achieves 100% success rate for all problems smaller than 2,000 nodes. However, on four instances (fnl4461, pr2392, u2319, d2103), the success rate is low. The u2319 is a special case and we discuss more in chapter 4, section 4.3.4. For the 1,024 population size, the convergence rate is slow with increasing problem size. This is expected because, MGA does not introduce new edges in the population. Having a higher population increases the number of P* edges, thereby MGA can converge fast.

3.4.4 Recombinations to global optima

The number of recombinations required bv the MGA to reach the global optimum is sometimes much less than EAX, particularly for problems with less than 2000 vertices. We can count the recombinations by multiplying the population size, generations and number of children. For instance, consider C1k.0. The MGA requires a median of 171 generations whereas EAX requires 264 generations. Thus, the MGA performs $1024 * 171 * 1/2 = 87552$ recombinations which is 27 times lesser than the number of recombinations used by EAX ($300 * 264 * 30 = 2376000$).

On a few problems EAX is clearly better (fnl4461, pr2392, rl1323, d2103) but on other problems the MGA is clearly better, or finds the global optimum more often (fl1577, u1817, nrw1379). Both MGA and EAX fail on u2319 instance. From figure 3.8, we can see that that MGA outperforms EAX on 14 out of 16 instances where both MGA and EAX achieve 100% success.

3.4.5 Success Rate

The success rate of the MGA becomes more stable and reliable with increasing population size. Consider the following cases: C3k.1, pcb1173, u1817, rl1323, nrw1379. The success rate of these five instances is less than 33% for 1024 population size. For a population size of 16384, the success rate is 100%. For these cases, the MGA gets stuck at some particular local optima and takes a long time to escape even though it has all the P* edges in the population.

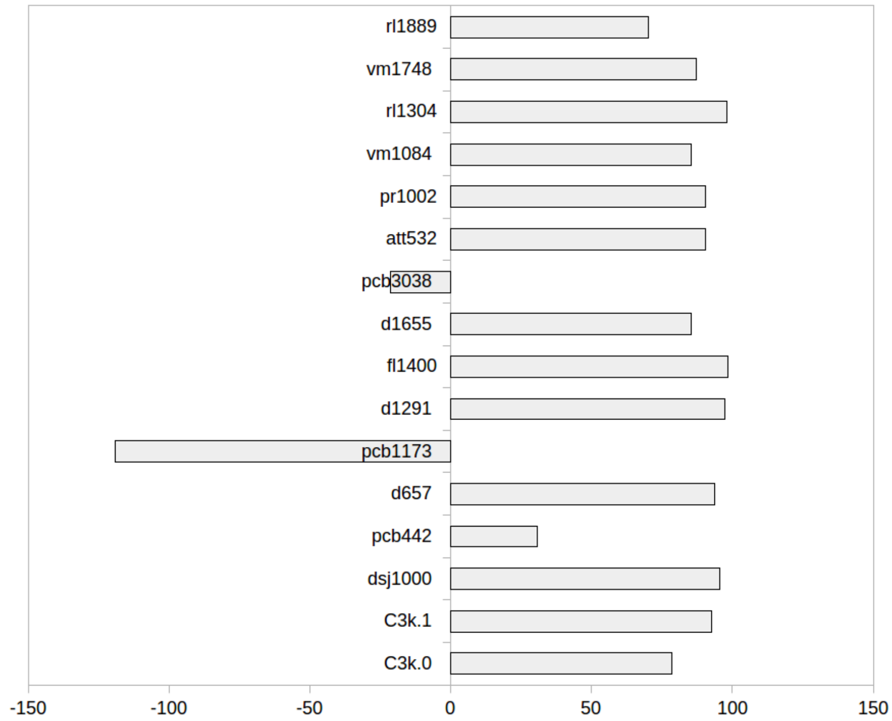


Figure 3.8: Percentage improvement of Mixing GA over EAX in terms of median recombinations for instances where both EAX and Mixing GA achieve 100% success

For the rl1323 instance, the MGA gets to a local optima with an evaluation value of 270226 reasonably quickly. However, the number of generations it takes to jump from this local optima to the global with an evaluation of 270199 is high and unpredictable in the current implementation. The problem is that the local optimum with evaluation value 270199 does not contain 51 edges that are in P* and the frequency of these (misleading) edges in the best solutions in the population is high. With the 1024 population size, the Mixing GA was stuck in this local optima for 29 of the 30 trials. This suggests that the search space induced by the MGA is still sometimes a “multi-funnel" landscape [3] and that the MGA can still become stuck for a significant amount of time.

3.4.6 Execution time

The execution time of MGA is slow compared to EAX. The number of recombinations per generation of EAX with a population size 300 and children size 30 is $300 * 30 = 9000$. For MGA with a population of 16,384, the number of recombinations per generation is $16,384/2 = 8192$.

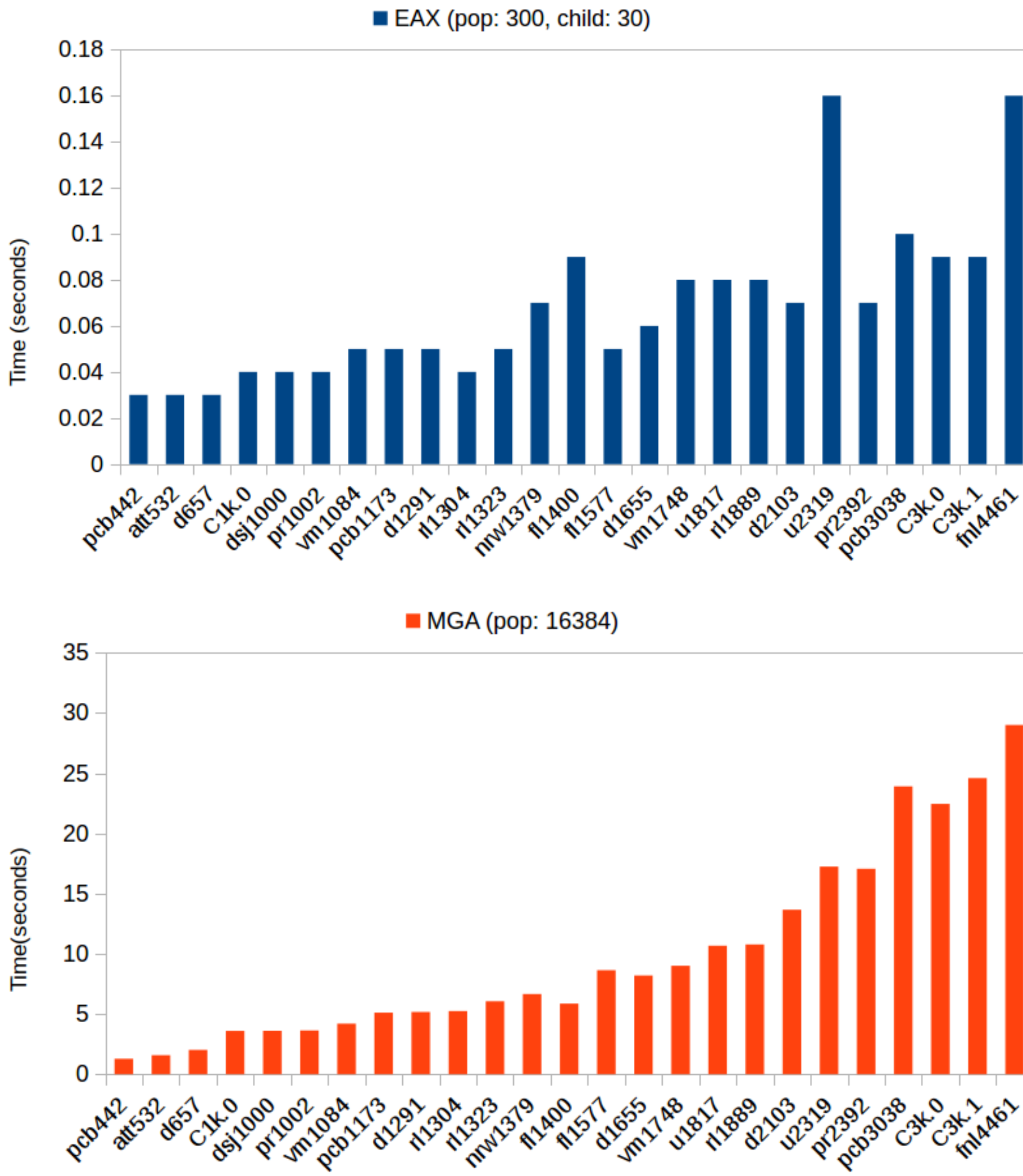


Figure 3.9: Although the number of recombinations per generation of EAX and MGA are comparable, there is a huge difference in their execution times.

Although the number of recombinations per generation is comparable, their execution times are not. Figure 3.9 shows the average execution time per generation. It can be seen that execution time is directly proportional to the problem size except on two instances (fl1400, u2319).

The MGA is on average $132\times$ slower than EAX. But, this observation is not surprising because (a) EAX uses highly sophisticated data-structures [94] (b) EAX consumes more memory to retain

information. When two parents recombine, the information regarding the AB cycle are stored in these additional memory structures. Thus, the 30 children per crossover operation is highly local. (c) EAX uses $54\times$ lesser population size when compared to MGA. Multiplying 30(EAX children) and 54 gives 1620 recombinations. But, MGA is run on 12 thread machines. Thus, dividing 1620 by 12 gives 135. From these calculations, MGA can be expected to be $135\times$ slower when compared to EAX. To summarize, the slower execution time is justifiable and needs improvement.

3.5 Conclusion

The Edge Assembly Cossover can be efficiently implemented on the multi-core CPU machines. However, there are two major challenges when porting to massively parallel SIMD machines. One, the memory EAX crossover operation takes is 100N. Second, the communication costs per crossover operation is irregular and random. In this Chapter, we address these two challenges by developing the Mixing Genetic Algorithm using the Generalized Partition Crossover operator. GPX consumes 4x less memory and has zero communication during the crossover operation.

The performance of MGA is compared to EAX with respect to six metrics. First, the MGA requires fewer recombinations to reach global optima on all 25 instances considered. Second, MGA converges fast with higher population sizes. Third, the success rate of MGA on a higher population size is comparable to EAX for 18/25 instances, outperforms EAX on 3/25 instances. Fourth, in terms of number of generations, the MGA converges fast on 9/25 instances. Fifth, in terms of problem scalability, MGA has slower convergence on higher problem sizes. However, higher population size helps to overcome this issue. Finally, in terms of execution time, the implementation of MGA is slow when compared to EAX.

The immediate future steps to study the convergence rate and the execution time of the solver. Chapter 4 studies and compares the runtime behaviour of EAX and MGA. Chapter 5 studies and presents results on execution time improvements and problem scalability on multi-core CPU architecture. Finally, Chapter 6 presents priliminary results on execution time improvements and problem scalability on the GPU architecture.

Chapter 4

On Runtime behaviour and Convergence

The Edge Assembly Crossover (EAX), is one of the state-of-the-art TSP solver. It is found to be the single-best solver [16] for a wide variety of problems smaller than 2,000 cities. However, there is no known comparitive study made for larger problem sizes. The multi-core version of EAX can generate best known solutions upto 200,000 city problems. However, this version of EAX does not fully exploit the inherent parallelism of the problem. In this thesis, we want to develop a GA solver than can maximally utilize the inherent parallelism and is efficient for larger problem sizes. To this end, in Chapter 3, we developed the Mixing Genetic Algorithm (MGA) using the Generalized Partition Crossover (GPX).

The GPX crossover operator consumes 4x less memory, and does not incur communication costs to the distance matrix. These features make the GPX operator favourable not only for multi-core CPU machines, but also for massively parallel SIMD machines. The MGA using the GPX has been tested on 25 instances smaller than 5,000 cities. The results show that MGA converges faster using fewer number of recombiantions and on higher population sizes. The success rate is 100% for problems smaller than 2,000 nodes. However, it achieve a 100% success only for 2/7 problems larger than 2,000 nodes. The convergence rate is extremely slow in terms of the execution time.

In this chapter, we analyze the run time behaviour of the MGA and EAX algorithm. The two solvers start with a population initialized by the same 2opt code. However, evolution of the population is different. To study more on this aspect, first, in section 4.1 we provide a background on the run-time behaviour of EAX and MGA. Previous research work on the analysis of TSP solvers is provided. Next, in section 4.2, we find that the two properties - *edge-frequency* and *multiple-global optima* help in understanding the working of the solvers. Visualization methods to understand the solvers are introduced. Finally, in section 4.3 we show the complementary nature of the two solvers. A simple hybrid of EAX and MGA solves some hard TSP instances. These

observations help to understand the slower convergence and provides insight on how to improve the convergence of the solver. Part of this chapter is published in GECCO 2020 [26].

4.1 Background

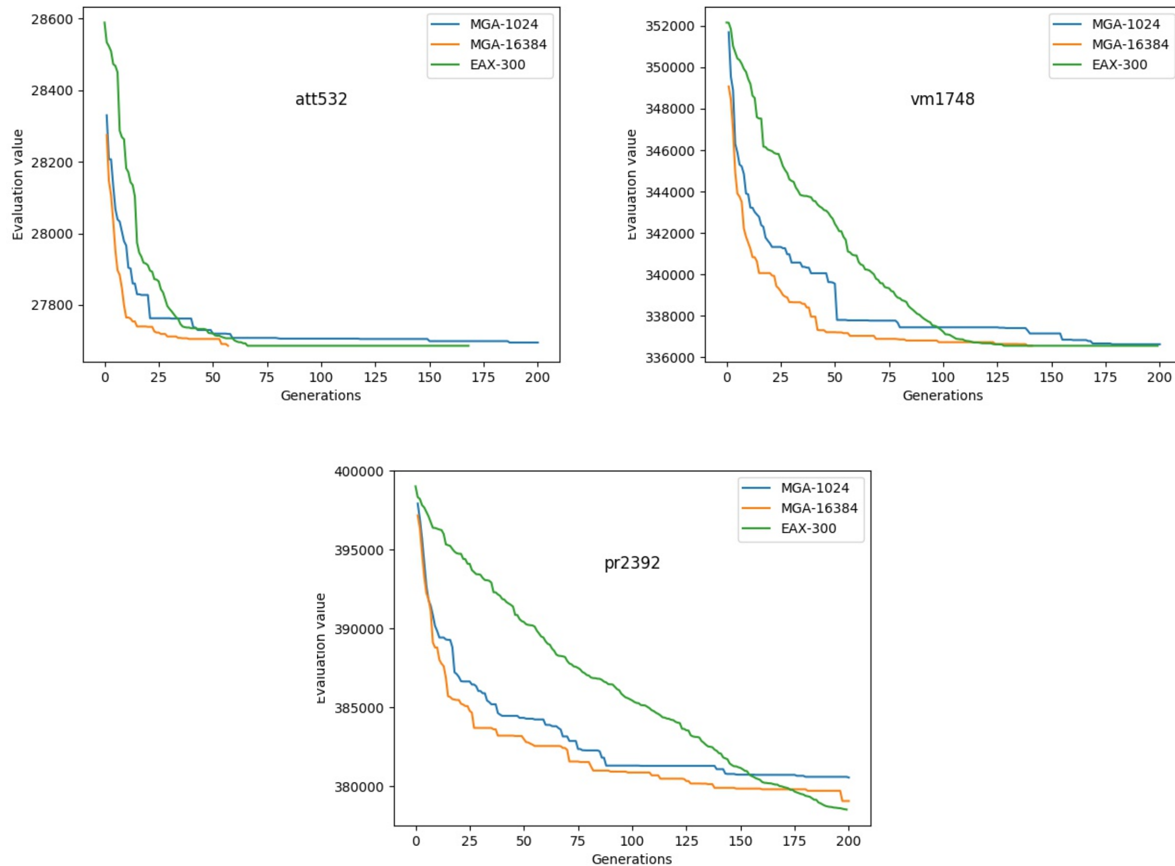


Figure 4.1: Figure showing the first 200 generations of running MGA and EAX. EAX overtakes the MGA trace between 25th and 175th generations. MGA converges faster on higher population sizes.

EAX and MGA starts with a population initialized by the same 2opt local search. However, the population size of the two algorithms are different. The default population size for EAX is 300. For MGA we need a population as large as 16,384. The main reason is that MGA does not introduce new edges in the population and thus needs a larger population to hold all possible global

optima edges. The question here is, can we do a fair convergence comparison when the two solvers start with different population sizes?

The EAX algorithm converges slower for larger population size. The reason is because of the diversity preservation mechanisms. We will discuss this aspect a little bit later in this section. On the other hand, the MGA converges fast on larger population size. The reason is because of the mixing properties. The MGA mixes the population in a hypercube topology. In log(popsize) steps, the best partitions of the entire population migrates to the first individual. If the population is larger, more good partitions migrate, leading to faster convergence.

Clearly, the two solvers have different properties and hence, starting with a population that favors convergence is a valid argument. However, the bias here will be in the properties of these populations. The population is merely made up of certain permuations of the edge distances. Thus, with the variation in the population sizes, the number of occurences of these edge distances differ. The order in which these edges occur inside a tour also differ. Thus, the performance of the solver comes down to the properties associated with the edge distances. Having these biases in mind, let us look at the run time behaviour of the two solvers.

Figure 4.1 shows the plot of EAX and MGA on three different instances. The results from chapter 3 are used to generate these traces. The MGA trace with a population of 16,384 is comparable with EAX trace with a population of 300. EAX produces 30 children per recombination. With 300 as the population size, EAX performs $300 * 30 = 9,000$ recombinations per generation. MGA takes two parents and produces two offprings. However, since the two offsprings are complementary to each other, we consider it as a single recombination. Thus, for a population of 16,384, MGA performs $16,384/2 = 8,192$ recombinations per generation. Therefore, the number of recombinations performed per generation is comparable. However, the two operators explore different regions of the search space.

Figure 4.1 shows that the MGA trace with a population of 1,024 and 16,384 follow a similar trend; except that MGA-16384 converges faster. In comparison with the EAX trace, MGA is fast in the intial stages. This observation is because the GPX operator is greedy and finds the best of

2^q possible offspring solutions, q being the number of partitions. However, the point at which the EAX trace intercepts MGA traces are different based on the problem type and size. It intercepts the MGA trace at an earlier stage for att532 (say, 30th generation), mid-way for vm1748 (say, 100th generation) and later stage for pr2392 (around 175th generation). On all instances, EAX trace overtakes the MGA trace betwen the 25th and 175th generation. The question here is, why does MGA slow down after initial stages?

MGA does not lose any diversity in the population. The edges are mixed in the population. However, there is no loss of edges. EAX, on the other hand, preserves diversity by using the "edge-entropy metric". Edge-entropy is the measure of the probability distribution of the edges in the population. For each vertex $i(i = 1, 2, ..., N)$, let $V_i = V_{ij}(j = 1, 2, ..., N)$ be the distribution of the vertices linked to vertex i in the population. i.e, the number of individulas including edge (i, j) in the population divided by $2popsize$. The edge entropy H of the population is defined as,

$$H_i = -\sum_{j=1}^{N} V_{ij} log(V_{ij})(i = 1, 2, ..., N), \tag{4.1}$$

$$H = \sum_{i=1}^{N} H_i = -\sum_{i=1}^{N}\sum_{j=1}^{N} V_{ij} log(V_{ij}). \tag{4.2}$$

Here, $H_i(i = 1, 2, ..., N)$ represents the entropy with respect to the distribution V_i, and H is the approximate entropy of the population, defined under the assumption that the distributions $V_i(i = 1, 2, .., N)$ are independent of each other.

The edge entropy information is updated at the end of every crossover operation. Let L the average tour length of the population. Let y be an offpring solution generated by a set of parents. Let $\triangle(L)_y$ and $\triangle(H)_y$ be the change in average tour length and edge-entropy due to this offspring. When $\triangle(L)_y$ is a positive value, an improving move is not found and so, the offspring never replaces the parents. When $\triangle(L)$ is a negative value, an improving move is found. However, the EAX algorithm also checks the $\triangle(H)_y$ values. A positive value of $\triangle(H)$ indicates that the diversity is not lost due to the replacement of this offspring. Note that EAX generates 30 offspring solution. Thus, if mulitple offsprings have a negative $\triangle(L)$ value, and a positive $\triangle(H)$ value,

the one with the smallest $\triangle(L)$ replaces the parent. For cases where there is no positive $\triangle(H)$, the offspring with smaller improvement in the average tour length per unit loss of the population diversity ($\triangle(L)/\triangle(H)$) is selected.

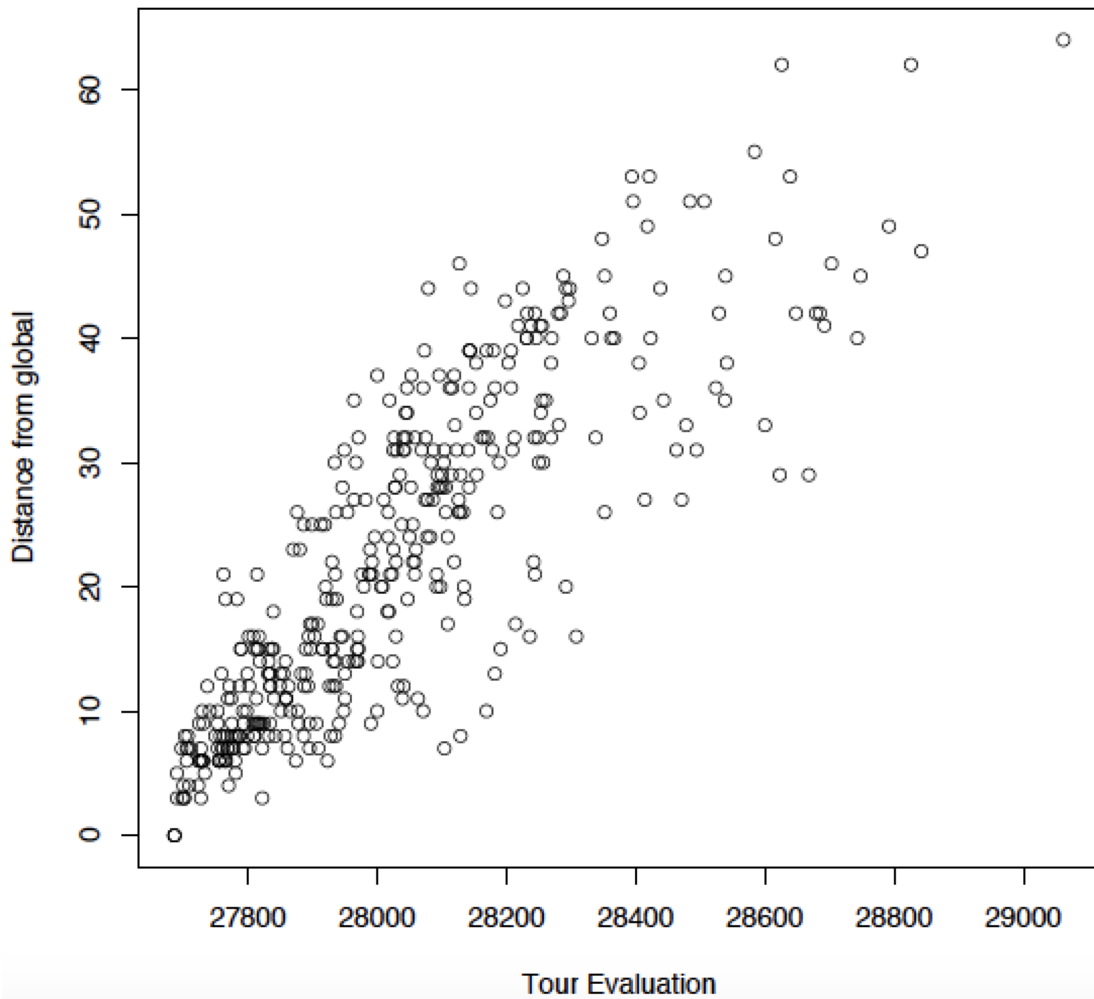


Figure 4.2: A classic example of the "Big Valley" distribution. Each circle is a randomly generated local optima under 2-opt for TSP instance ATT532. The x-axis is the objective evaluation $f(x)$, and the y-axis is the number of edges in each local optima that are **not** shared with the global optima. [2]

Boese et al. [2] originally introduced the concept of the “Big Valley" distribution of local optima, as illustrated in Figure 4.2. Briefly, the “Big Valley hypothesis" states that local optima that are closer to the global optimum in evaluation are also closer to the global optimum in terms of the number of shared edges (or some other reasonable distance metric). The definition of a local

optimum depends on the choice of a local move operator. Operators with larger neighborhoods will induce fewer local optima in expectation.

The EAX termination strategy is based on the "big valley" hypothesis. As L decreases, the H decreases (loss of diversity) and eventually converge. The EAX algorithm terminates when the difference between the average and the best tour length is less than 10^{-3}. However, the "big valley" hypothesis does not hold true for all TSP instances. The TSP landscape can have "big valley" that represents the local optima and not necessarily the global optima. The EAX solver does not check the gap of the optimal solution and the lower bounds. Therefore, the EAX solver runs into the risk of converging to this local optima.

Hains et al. showed that a classic TSP instance, ATT532, displays not only one large valley or "funnel," but additional funnels, although these were relatively small [111]. A funnel is nothing but a region in the TSP landscape where several local optima converge to one single point. This point need not be the global optima. Figure 4.3 depicts a simple two funnel structure. Ochoa et al. [3] also documented a multi-funnel structure in some TSP instances.

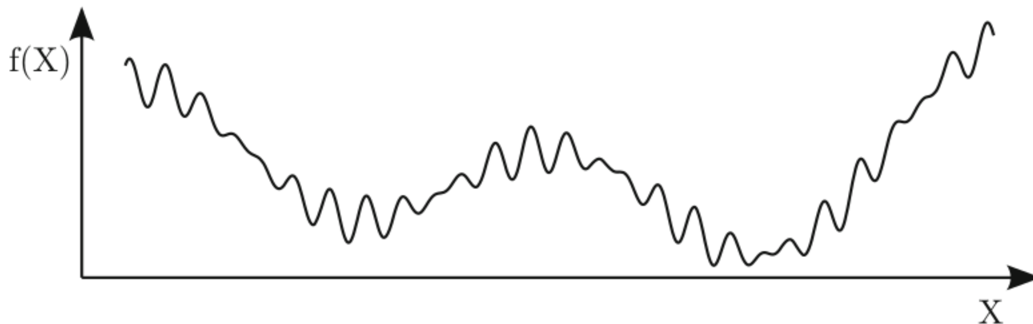


Figure 4.3: Figure illustrating a two funnel landscape [3].

Ochoa et al. [3] develop visualization techniques for mapping the global structure of TSP landscape. To construct the global landscape, they use the "local optima network"(LON). LON is a graph that has the local optimas as the vertices, and the edges are called "escape edges".

These escape edges represent the transition probabilities from one local optima to the next. These probabilities are collected when performing 10,000 iterations of Chained-LK [51].

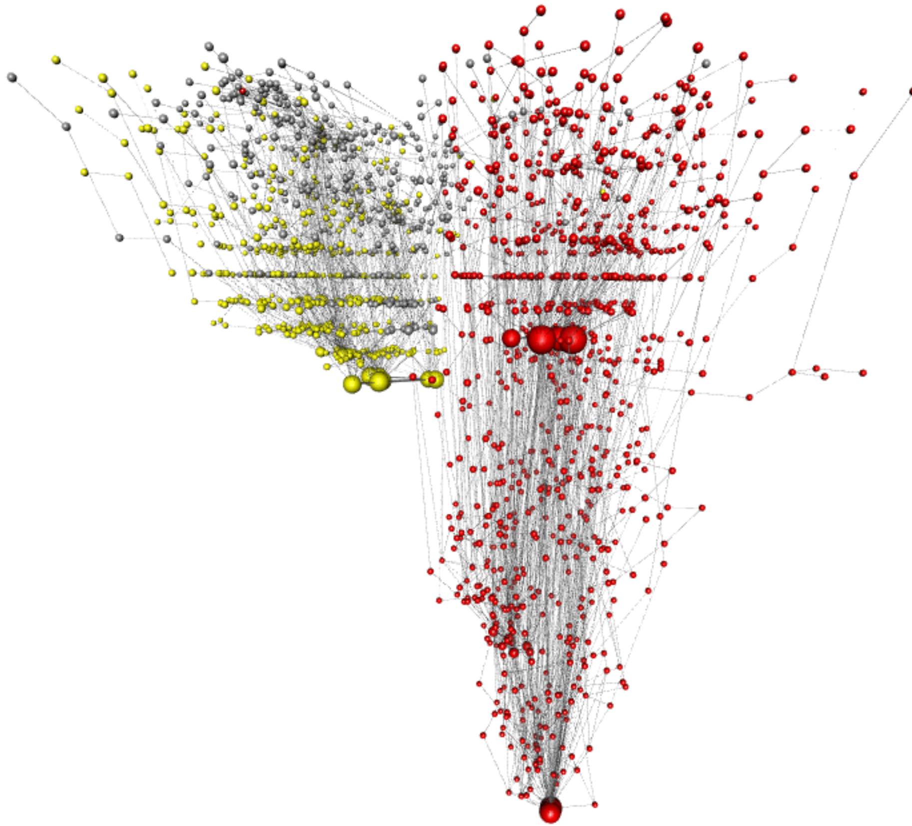


Figure 4.4: 3D Visualization of att532 funnels [3]. The red funnel shows the presence of global optima. The second funnel is colored yellow to indicate that the cost of the local optima is higher than those in the red funnel. for att32, the two funnels are over-lapping.

After extracting the LON, they systematically devise an approach to identify the "funnel" structures. Figure 4.4 shows the two funnels in att532. These study conclude that the *big valley* structure of the landscape is not typical, and in fact, the global structure exhibits *multi funnel* landscape.

Landscape analysis is very relevant to Iterated Local Search methods such as Chained-LK and LKH. But it is less clear that it can explain the behavior of population based methods, such as the EAX genetic algorithm [78]. The problem with the Big Valley hypothesis is that it only provides a "macro-level" landscape analysis that looks at the distribution of local optima. In this chapter

presents a “micro-level" analysis that counts the frequency of edges found in the global optimum. We analyze how the global and non-global edges are distributed across the population.

4.2 Micro-level analysis

Let us consider the 25 instances from chapter 3. The global optima edges for 11/25 instances are found in the first 50 nearest neighbors. For the remaining 14/25 instances, more than 99% of the global optima edges are in the 50 nearest neighbor list. This observation demonstrates that 2opt using nearest neighbor heuristics produces a population with abundant global optima edges.

EAX takes advantage of this nearest neighbour concept. EAX stores the edge frequency information of the population (for N^2 edges). For larger problems, EAX updates edge frequency for the 100 nearest neighbors. The edge-entropy defined in section 4.1 is a measure of these edge frequencies. Therefore, edge frequency is an important metric that determines the convergence rate. In this section, we discuss how edges change when applying the EAX and GPX operators.

4.2.1 Edge Frequency Distribution

We inspect the known global optimal solution of the 25 problems from chapter 3 . Table 4.1 shows what percentage of the population is made up of edges found in the global optima. It can be seen that for 20/25 problems, more than 70% of the initial population is made up of global optima edges. Furthermore, it can also be seen that this percentage is fairly constant with increasing population sizes. These observations suggest that the population created by the 2opt code, irrespective of the size, is rich in global optima edges. Table 4.2 gives the number of missing global optima edges when the population size is increased from 1 to 16,384. For 21/25 cases, a population as small as 256 is sufficient to hold all the global optima edges.

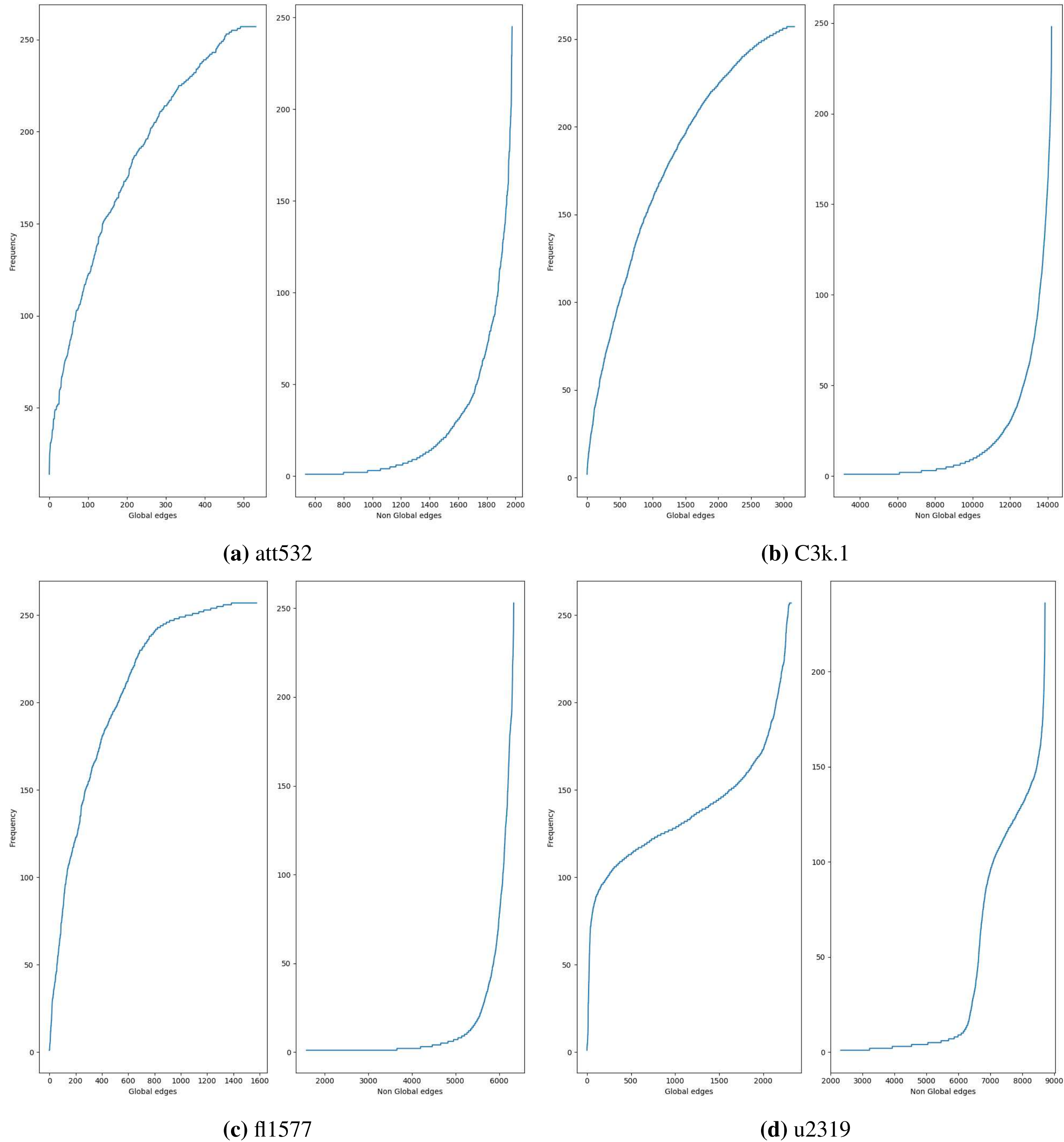
Next, we examined the frequency of each global optima edge. For att532, the total number of edges found in this particular population is 1,460. Of these, 534 are global optima edges. In figure 4.5a, edges that appear in 50% or more of the individuals in the population make up more than 400 of the 532 edges that appear in the global optimum, but include only 20 edges that are not in the

Table 4.1: Rank of instances sorted by the percentage of globally optimal edges found in the initial population. Instance u2319 had the fewest edges; instance rl1304 had the most edges from the global optimum. For 20 of the 25 instances, more than 71% of the edges in the population come from the global optimum. This percentage is almost constant with increasing population size

Instance	Population Size														
	1	2	4	8	16	32	64	128	256	512	1024	2048	4096	8192	16384
u2319	54.9	54.2	54.0	54.5	53.7	53.6	53.7	53.7	53.7	53.7	53.7	53.7	53.7	53.7	53.7
fl1400	56.6	56.2	55.6	55.7	55.8	55.8	55.8	55.9	55.9	55.9	55.9	55.9	55.9	55.9	55.9
fnl4461	69.5	70.3	70.2	69.9	69.7	69.7	69.6	69.6	69.6	69.6	69.6	69.6	69.7	69.6	69.6
pr1002	69.9	69.8	69.7	70.0	70.2	70.1	70.4	70.4	70.6	70.5	70.5	70.5	70.4	70.4	70.4
d657	70.2	71.8	72.1	73.0	73.1	73.2	73.2	73.2	73.0	73.0	73.0	73.0	73.0	73.0	73.0
dsj1000	70.3	70.9	70.3	70.8	71.4	72.0	72.1	72.1	72.0	72.0	72.0	72.1	72.0	72.0	72.0
pcb442	70.6	71.2	71.8	70.7	70.1	69.8	69.5	69.2	69.2	69.3	69.2	69.3	69.3	69.3	69.4
C3k.1	70.7	70.9	70.8	70.7	70.6	70.6	70.8	70.8	70.8	70.8	70.8	70.8	70.8	70.8	70.8
pcb1173	71.3	71.6	73.7	73.9	74.2	74.1	74.4	74.6	74.6	74.5	74.5	74.5	74.5	74.5	74.5
C3k.0	71.8	72.1	71.4	71.4	71.4	71.4	71.5	71.6	71.7	71.6	71.6	71.6	71.6	71.6	71.6
nrw1379	71.9	70.9	70.8	70.8	70.3	69.9	70.0	70.0	70.0	70.0	70.0	70.0	70.0	70.0	69.9
att532	72.0	71.6	70.7	70.9	70.7	71.4	72.0	71.8	72.0	72.0	71.9	71.8	71.8	71.8	71.8
pcb3038	72.0	71.9	72.5	72.4	72.5	72.4	72.6	72.7	72.6	72.7	72.7	72.7	72.7	72.7	72.7
C1k.0	72.5	71.7	71.5	72.3	72.4	72.1	71.4	71.4	71.5	71.4	71.4	71.5	71.5	71.5	71.5
pr2392	74.6	75.1	74.6	75.2	75.5	75.1	75.0	75.0	75.1	75.1	75.1	75.1	75.1	75.1	75.1
u1817	76.4	77.4	77.8	78.1	78.3	78.6	78.7	78.6	78.6	78.5	78.6	78.6	78.6	78.6	78.6
d1655	79.4	79.6	80.1	80.0	79.8	79.5	79.4	79.4	79.5	79.5	79.6	79.6	79.6	79.6	79.6
fl1577	79.7	79.9	79.3	79.3	79.6	79.8	80.0	80.0	80.0	80.0	79.9	79.9	79.9	79.9	79.9
vm1084	82.8	81.8	81.1	80.8	80.5	80.4	80.5	80.6	80.6	80.6	80.5	80.5	80.5	80.5	80.5
vm1748	83.0	82.2	82.0	82.0	82.1	82.0	82.0	81.9	81.9	81.9	81.9	81.9	81.9	81.9	81.9
rl1889	83.7	84.1	84.1	84.0	84.3	84.4	84.4	84.4	84.5	84.5	84.5	84.5	84.5	84.5	84.5
d1291	85.2	85.9	86.6	86.4	86.1	86.3	86.5	86.5	86.4	86.3	86.3	86.3	86.2	86.3	86.3
rl1304	87.1	86.8	87.3	87.3	87.2	86.9	87.1	87.2	87.3	87.3	87.2	87.2	87.2	87.2	87.2
rl1323	87.2	87.7	87.9	87.8	87.4	87.4	87.1	87.1	87.1	87.1	87.1	87.1	87.1	87.1	87.1
d2103	87.4	86.6	86.9	86.4	86.2	86.1	86.3	86.3	86.3	86.2	86.3	86.2	86.2	86.2	86.2

Table 4.2: Number of missing global optima edges

Instance	Population Size														
	1	2	4	8	16	32	64	128	256	512	1024	2048	4096	8192	16384
att532	149	77	32	10	2	1	0	0	0	0	0	0	0	0	0
C1k.0	275	133	70	29	16	3	1	1	0	0	0	0	0	0	0
C3k.0	891	466	224	96	30	13	3	1	0	0	0	0	0	0	0
C3k.1	927	480	207	91	32	11	5	1	1	0	0	0	0	0	0
d1291	191	76	30	15	8	2	1	0	0	0	0	0	0	0	0
d1655	341	158	71	42	23	12	7	5	0	0	0	0	0	0	0
d2103	265	102	67	57	30	14	12	8	5	5	5	3	3	1	0
d657	196	93	50	17	6	3	1	1	0	0	0	0	0	0	0
dsj1000	297	141	79	26	5	1	1	0	0	0	0	0	0	0	0
fl1400	608	346	180	77	33	17	7	1	0	0	0	0	0	0	0
fl1577	320	157	75	30	12	7	6	4	2	1	1	1	0	0	0
fnl4461	1359	671	268	108	37	13	4	2	0	0	0	0	0	0	0
nrw1379	388	193	74	32	10	3	1	0	0	0	0	0	0	0	0
pcb1173	337	185	72	32	10	4	1	1	0	0	0	0	0	0	0
pcb3038	851	397	155	68	16	6	2	0	0	0	0	0	0	0	0
pcb442	130	70	15	5	0	0	0	0	0	0	0	0	0	0	0
pr1002	302	167	76	29	12	3	1	0	0	0	0	0	0	0	0
pr2392	607	294	142	58	16	6	1	0	0	0	0	0	0	0	0
rl1304	168	89	42	23	8	2	0	0	0	0	0	0	0	0	0
rl1323	170	88	52	23	10	4	1	1	0	0	0	0	0	0	0
rl1889	308	160	97	51	19	6	2	0	0	0	0	0	0	0	0
u1817	428	188	90	36	13	7	2	0	0	0	0	0	0	0	0
u2319	914	445	139	20	7	4	0	0	0	0	0	0	0	0	0
vm1084	186	104	61	38	19	15	6	4	4	0	0	0	0	0	0
vm1748	297	160	80	34	15	7	2	0	0	0	0	0	0	0	0



(a) att532

(b) C3k.1

(c) fl1577

(d) u2319

Figure 4.5: Plots showing the frequency of each global and non-global edges in the population after applying the 2opt local search. For instance, the graph to the right is the frequency of global optima edges. The graph to the left is the frequency of non-global optima edges.

global optimum. More than 70% of the **total** edges in the population are also found in the global optimum; however, more than 35% of all the **distinct** edges in the population (i.e., 532 of 1460 edges) are found in the global optimum. We see this trend to be true with cluster instances C3k.1

in figure 4.5b. For fl1577, in figure 4.5c, we see that the frequency of global optima edges are even much higher. However, the trend is a bit different for u2319 in figure 4.5d. The reason is explained in the following section.

4.2.2 Multiple Global optima

After further investigation, we found that only seven instances (rl1323, vm1748, C3k.1, pr1002, C1k.0, dsj1000, rl1304) have unique global optimum (i.e., that we could find). This explains the lower edge-frequency numbers on u2319, fl1400 and fnl4461. Thus, the presence of multiple global optima can change the statistics we discussed. However, the goal here is to observe the edge-frequency change and how the presence of multiple global optima edges affect the convergence.

4.2.3 Unknown Global optima

In this chapter, we analyze only the problems that has a known global optima solutions. However, for problems that do not have a known global optima, Held-Karp (HK) [35] bound is considered. Please refer to chapter 2, section 2.1 on steps to calculate the HK-bound.

4.2.4 Visualization methods

We use "heatmaps" to visualize the tours in the population. In the heatmap, the global edges are colored from red to yellow and non-global edges are colored black. This representation tells the pattern and exact location of the edges inside a given tour. The heatmap in figure 4.6 shows the initial att532 population of size 512. To better understand the edge frequency distribution, each tour is sorted. The non-global edges are accumulated to the front, followed by the global edges. The heatmap in figure 4.7 demonstrates the sorted heatmap version. Since we are interested in the edge frequency information, we use the sorted version for further analysis.

The initial att532 population in figure 4.6 shows that the percentage of non-global and global edges is approximately constant across the population. This confirms the observation made in table 4.1. Irrespective of population size, the percentage of edges remain approximately constant because of the bias created by the 2opt local search.

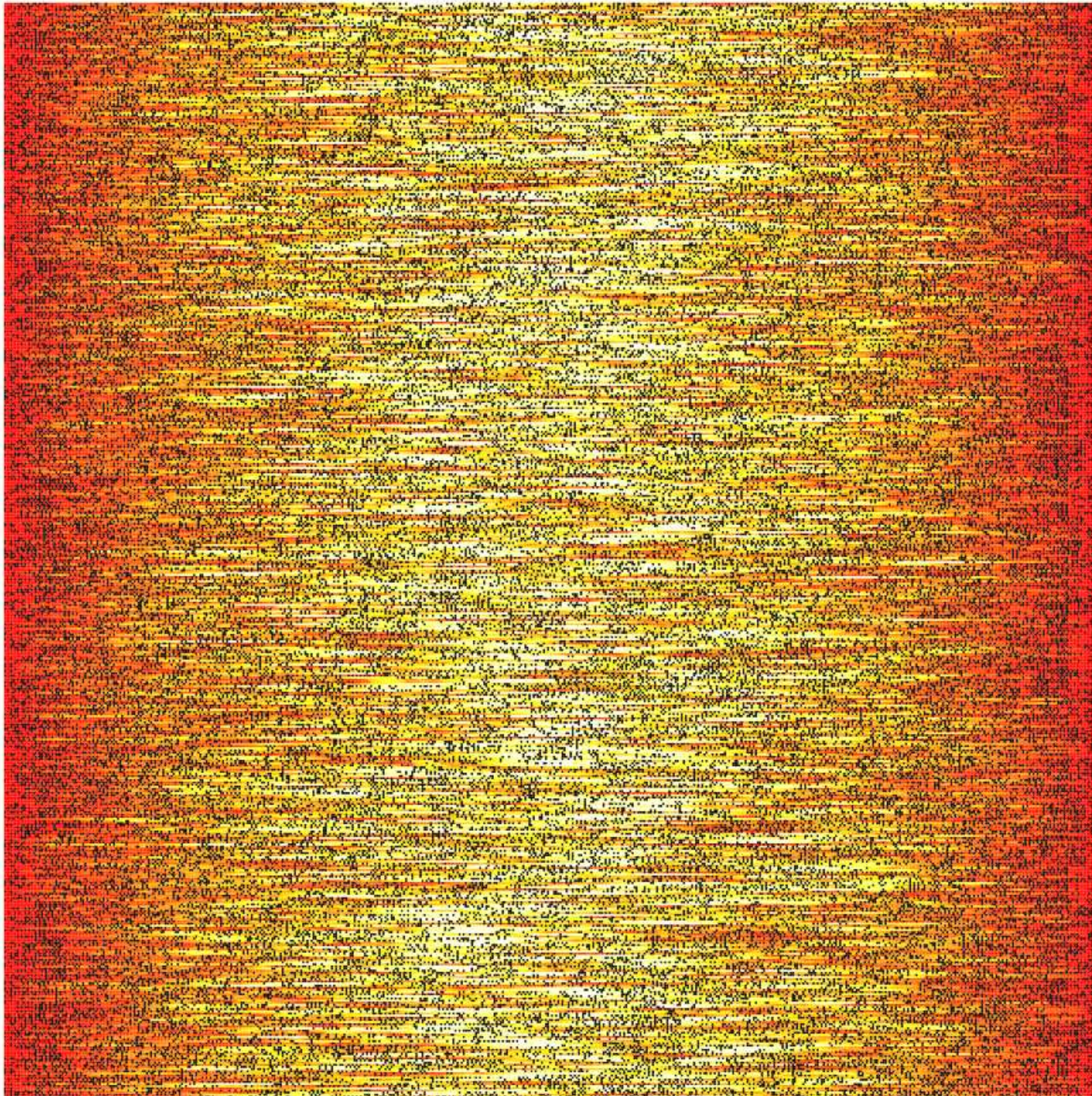


Figure 4.6: Figures showing heatmaps of the intial population (size=512) of att532. The heatmap has 512 rows and 532 columns. Each row represents a tour. Black color represents the non-global edges. The non-black edges represent the global edges.

Figure 4.8 shows the heatmaps for every 10 generations of EAX on att532. It can be seen that EAX gradually removes the non-global edges and increases the global edges in the population. EAX finds the global optima at generation 60 yet, it continues and terminates only at generation 160. This is because, the EAX algorithm terminates iff the difference between the average and the best tour length is less than 10^{-3}.

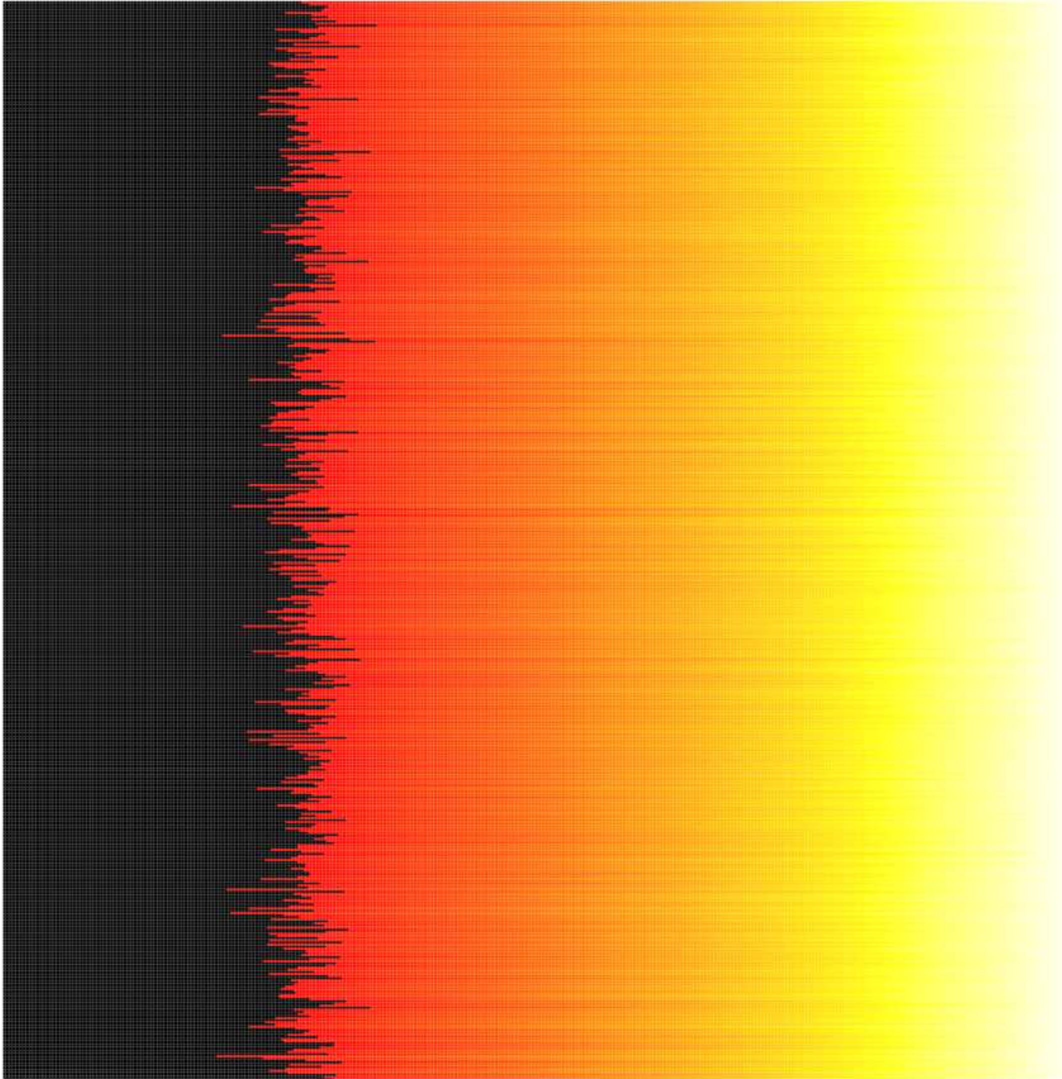


Figure 4.7: Figures showing heatmaps of the intial population (size=512) of att532. The heatmap has 512 rows and 532 columns. Each row represents a tour sorted according to the edge number. The black color represents the non-global edges and have a number 0. The non-black edges represent the global edges and are numbered from 1 to 532. Hence, the gradient in color from red to white.

Figure 4.9 shows the heatmaps at the end of every epoch. It can be seen that, with the progression of the epoch, the non-global edges are pushed to the bottom of the population, global edges ride to the top. The feather-like distribution of edges is because of the mixing pattern of MGA. It corresponds to the total number of best and worst partitions accumulated over the generations. However, the first position is guarenteed to always receive the best partition. Similarly, the last position is guarenteed to always receive the worst partition.

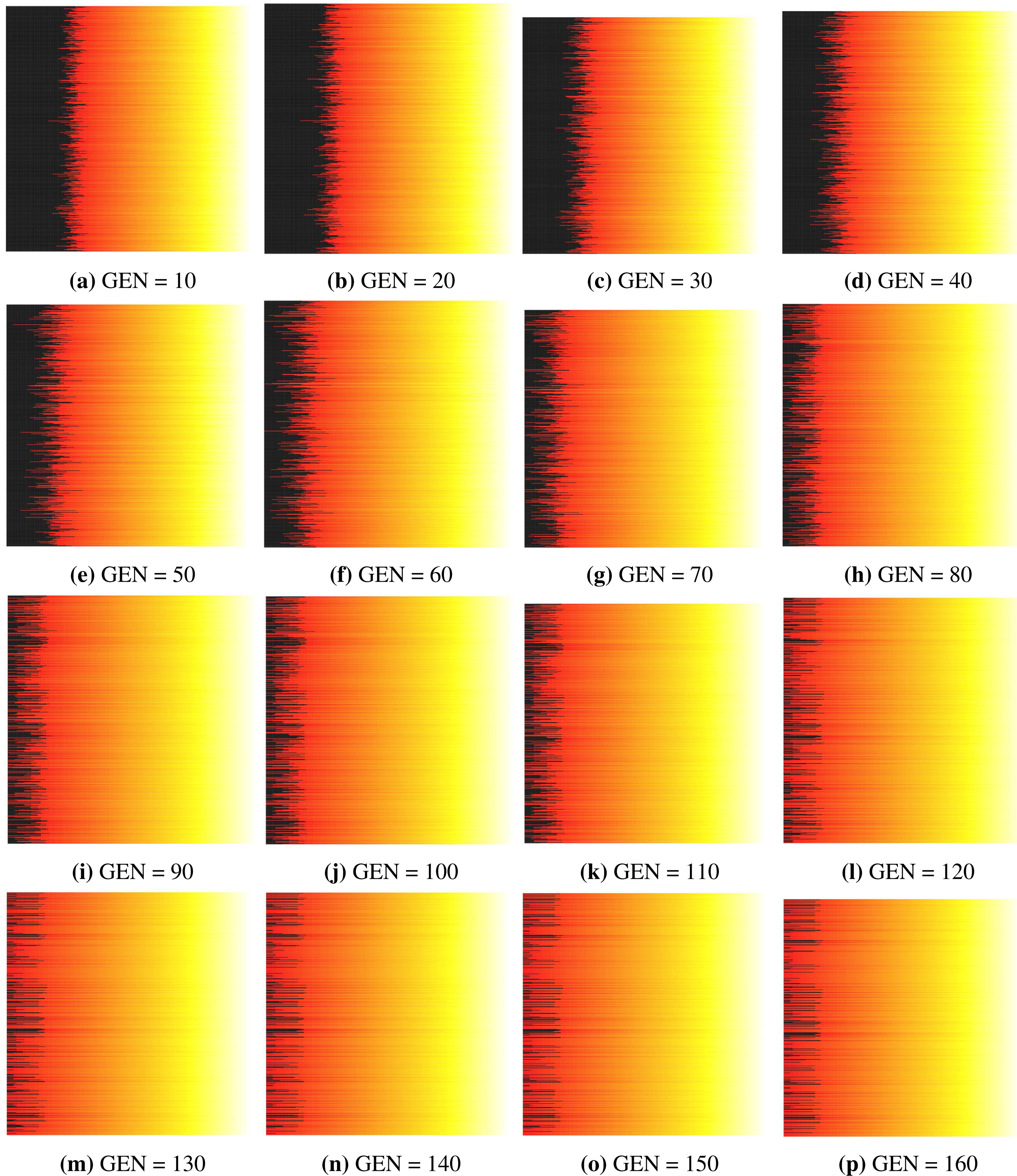


Figure 4.8: The heatmaps on EAX population for every 10 generation on att532 with a population of size 512. Two individuals reach global optima at the 60th generation.

4.2.5 Summary

While EAX and MGA use different recombination operators, both EAX and GPX are exploring a very restricted space that is densely populated with (often all of) the edges that appear in the

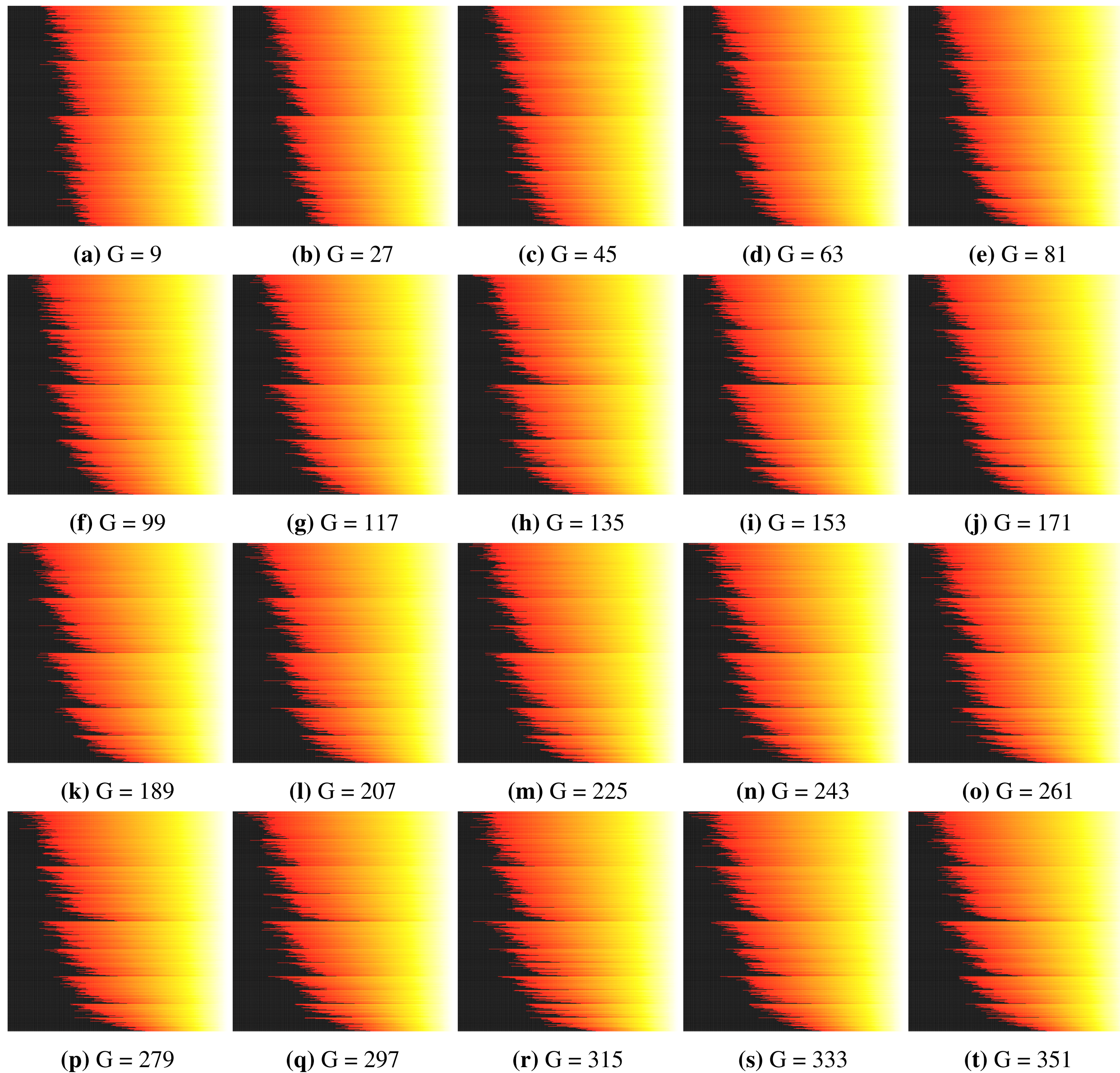


Figure 4.9: The heatmaps on MGA population for every other epoch on att532 with a population of size 512. One epoch consists of 9 generations (G).

global optimum. In the next section, we introduce simple hybrid MGA+EAX algorithm. The main goal is to examine how the solvers exploit edge frequency information in the population.

4.3 Hybrid algorithms

The limitaion of MGA is that it doesn't create any new edges. On the other hand, EAX creates new edges in the population. We want to study the complementary nature of these two solvers.

Algorithm 2 The MGA+EAX for TSP – Hybrid 0

1. Randomly initialize the population.
2. Apply 2opt on the population.
3. **while** (termination is not met)
4. Randomly shuffle the population.
5. Apply one **epoch** of The Mixing GA.
6. Apply one generation of EAX
7. **end while**

The listing 2 shows the simple hybrid of EAX and MGA. Let us called this version as hybrid-0. We reduced the population size of the MGA to 300 to be the same as EAX. In generation 1, the 150 individuals in the first half of the population recombine with the 150 individuals in the second half of the population. In general, adjacent *groups* recombine. In generation 2 there are 4 groups of 75: group 1 recombines with group 2, and group 3 recombines with group 4. Then the population involved in recombination is reduced: In generation 3 the population is broken into 8 groups of 37. In generations 4 and 5, there are 16 groups of 18, and 32 groups of 9 respectively. In generations 6, 7 and 8 the population involved in recombination is reduced to 256 (a power of 2). There are 8 generations in an epoch. In the 9th generation, the EAX recombination operator is applied for 1 generation, with the normal procedure of generating 30 offspring.

Table 4.3: Results on applying MGA, MGA+EAX, EAX-default and EAX-optimal to 25 problem instances. *The results shown for fl1577 under last 3 columns are on a population of size 4096. Here S.R means success rate. Each instance is run for 30 times.

Instance	MGA+EAX, pop=300			EAX-default, pop=300			EAX-optimal, pop=300		MGA pop=16384		
	Avg. Gen.	Avg. Rec.	S.R	Avg. Gen.	Avg. Rec.	S.R	Avg. Gen.	Avg. Rec.	Avg. Gen.	Avg. Rec.	S.R
pcb442	64.7	71773.87	30/30	60.9	548100	30/30	28.3333	254999.7	66.8	547225.6	30/30
att532	117.53	130379.95	30/30	163.6	1472400	30/30	62	558000	75.67	619888.64	30/30
d657	146.33	162328.75	30/30	170.63	1535670	30/30	69.2667	623400.3	64	524288	30/30
C1k.0	147.5	163626.67	30/30	265.16	2386440	30/30	156.633	1409697	58.86	482181.12	30/30
dsj1000	132.5	146986.67	30/30	236.83	2131470	30/30	131.733	1185597	64.93	531906.56	30/30
pr1002	248.67	275857.92	30/30	229.86	2068740	30/30	127.9	1151100	104.6	856883.2	30/30
vm1084	126.3	140108.8	30/30	180.73	1626570	30/30	78.1333	703199.7	58.4	478412.8	30/30
pcb1173	309.8	343671.47	30/30	230.2	2071800	30/30	126.1	1134900	572.2	4687462.4	30/30
d1291	99	109824	30/30	153.6	1382400	30/30	51.4	462600	41.13	336936.96	30/30
rl1304	56.76	62965.76	30/30	163.33	1469970	30/30	61.0667	549600.3	28.06	229867.52	30/30
rl1323	133.1	147652.27	30/30	172.9	1556100	30/30	68.1667	613500.3	442.46	3624632.32	30/30
nrw1379	574.43	637234.35	30/30	299.86	2698740	28/30	208.133	1873197	1341.73	10991452.16	30/30
fl1400	93.73	103977.81	30/30	186.86	1681740	30/30	94.1333	847199.7	27.55	225689.6	30/30
fl1577	6028.5	6687616	12/30	180.46	1624140	12/30	190	1710000	878	7192576	30/30
d1655	363.23	402943.15	30/30	231.86	2086740	30/30	133.2	1198800	184.86	1514373.12	30/30
vm1748	294.86	327098.03	30/30	237.36	2136240	30/30	135.8	1222200	144.26	1181777.92	30/30
u1817	762.13	845456.21	30/30	229.2	2062800	18/30	172.6	1553400	1526.06	12501483.52	30/30
rl1889	174.43	193501.01	30/30	212.46	1912140	30/30	110.433	993897	265.6	2175795.2	30/30
d2103	251.93	279474.35	30/30	174	1566000	30/30	70.7	636300	3074.93	25189826.56	27/30
u2319	7516.67	8338492.59	22/30	236.26	2126340	0/30	258.7	2328300	10000	81920000	0/30
pr2392	519.5	576298.67	30/30	298.2	2683800	30/30	232	2088000	6821.41	55880990.72	19/30
pcb3038	1039.76	1153440.43	30/30	468.33	4214970	30/30	365.3	3287700	5085.33	41659023.36	30/30
C3k.1	610.43	677170.35	30/30	691.43	6222870	30/30	522.933	4706397	2583	21159936	30/30
C3k.0	587.6	651844.27	30/30	650.36	5853240	30/30	558.833	5029497	319.26	2615377.92	24/30
fnl4461	2436.83	2703256.75	30/30	796.1	7164900	28/30	738.633	6647697	10000	81920000	1/30

4.3.1 Experimental setup

The experimental setup is the same as that followed in Chapter 3. The hybrid MGA+EAX algorithm terminates after 10,000 generations or when it finds the global optima. The MGA and EAX results are taken from previous chapter for comparison purposes.

4.3.2 Recombination, Generation, Success Rate

Let *P* denote the population size and let *G* denote the number of generations. Each run of MGA generates $P/2 * G$ recombinations. Let *C* denote the children produced by EAX in each "brood" (i.e., each pair of parents is recombined C times). Each run of EAX generates $P * C * G$ recombinations. For the hybrid MGA+EAX algorithm, there are $G/((log_2 P) + 1)$ epochs, CP recombinations using EAX per epoch and $(log_2 P)P/2$ recombinations using GPX per epoch for a total of $G/((log_2 P) + 1)\,[(log_2 P)P/2 + CP]$ recombinations. However, note that due to the hybercube topology, for a population of 300, some of the individuals may not participate in the recombination. The numbers we report are calculated during run-time and not using the formula. The idea here is to use the same initial population and see how that impacts the performance of the solver. In general, it is always advised to use a population that is a power of 2 for MGA solver.

Comparative results are shown in Table 4.3. In terms of number of generations to global optima, on 14/25 instances, the EAX optimal is fast. MGA-16384 is fast on 9/25 instances. For the remaining 2/25 instances (fl1577, u2319), the EAX-default seems to be superior. However, note that EAX does not achieve 100% success rate on fl1577. On u2319, it achieves 0% success rate. Therefore, comparing the number of generations on u2319 does not make sense. To summarize, the hybrid algorithm is not superior to EAX or MGA in terms of number of generations to global optima. However, in terms of number of recombinations to global optima, the hybrid is superior to EAX and MGA on 23/25 instances. On the remaining two instances (fl1577, u2319), the EAX-default seems to dominate. Note that for u2319, the hybrid version alone finds the global optima. Therefore, comparing the generations with other versions does not make sense. Overall, the hybrid is superior in terms of number of recombinations.

In terms of success rate, all three solvers achieve a 100% on 17/25 instances. On remaining 8 instances (u2319, fl1577, ,d1655, u1817, d2103, pr2392, C3k.0, fnl4461), we find a mixed behaviour. Clearly, the hybrid is superior on u2319. MGA is superior on fl1577. The d1655 be an outlier (since EAX-optimal achieves 29/30 and EAX-default achieves 30/30). EAX has trouble solving u1817. MGA and the hybrid gets 100% on u1817. MGA has trouble solving pr2392, fnl4461 and C3k.0. For d2103, MGA achieves 90% (27/30) success rate. This is because, the d2103 requires a huge population to contain all P* edges. Therefore, the reduced rate can be because of the absence of abundance of the rare and important edge. The reasons for these different behaviour is explained in the subsequent sections. Table 4.5 summarizes the reasoning.

Figures 4.10, 4.11, 4.12 and 4.13 shows the run time bahviour of the three algorithms on four different instances. Four distinct trends can be observed. For C3k.0, the MGA finds the solution fast, followd by EAX and then the hybrid. For C3k.1, it is the hybrid, followed by EAX and then MGA. For pcb3038, the EAX, followed by hybrid and then MGA. Finally, for fl1400, it is MGA, followed by hybrid and then EAX. In all the cases, the MGA algorithm is fast in the initial stages (at least upto 30 generations), followed by the hybrid and then the EAX. This is because of the presence of the GPX operator.

4.3.3 Final Populations Edge Frequencies

Table 4.4 gives the frequency of global optima edges in the final populations. The frequencies for the standard MGA are the same as those for the initial population. For most instances the hybrid MGA+EAX algorithm only slightly increased the frequency of global optima edges in the final population. This is because, when EAX is combined with MGA, we use only the localized version with a single selection strategy. This ensures that only very few edges are introduced.

On most instances, EAX is highly converged. EAX is less converged for two of the instances, u2319 and fl1400. When examining these two instances more closely, we found that u2319 has two global optima that are far apart from each other, whereas fl1400 has at least 30 different global

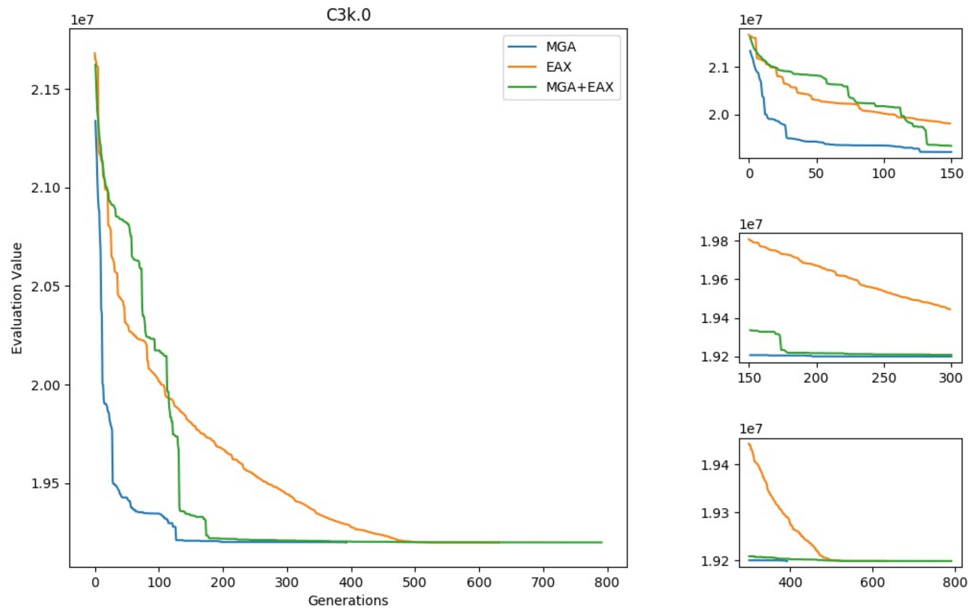


Figure 4.10: Progress of EAX and MGA and the hybrid MGA+EAX on C3k.0 for popsize=300.

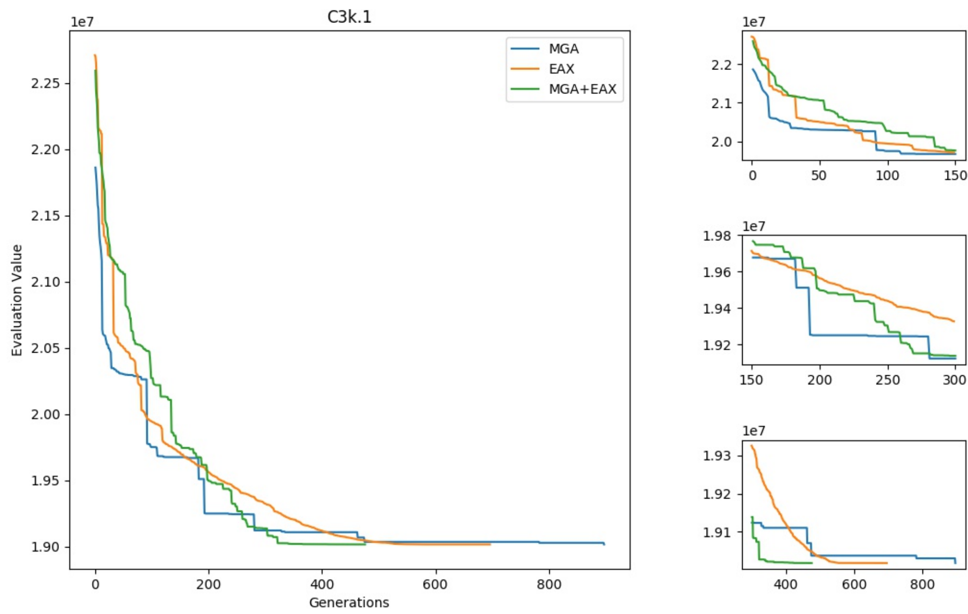


Figure 4.11: Progress of EAX and MGA and the hybrid MGA+EAX on C3k.1 for popsize=300.

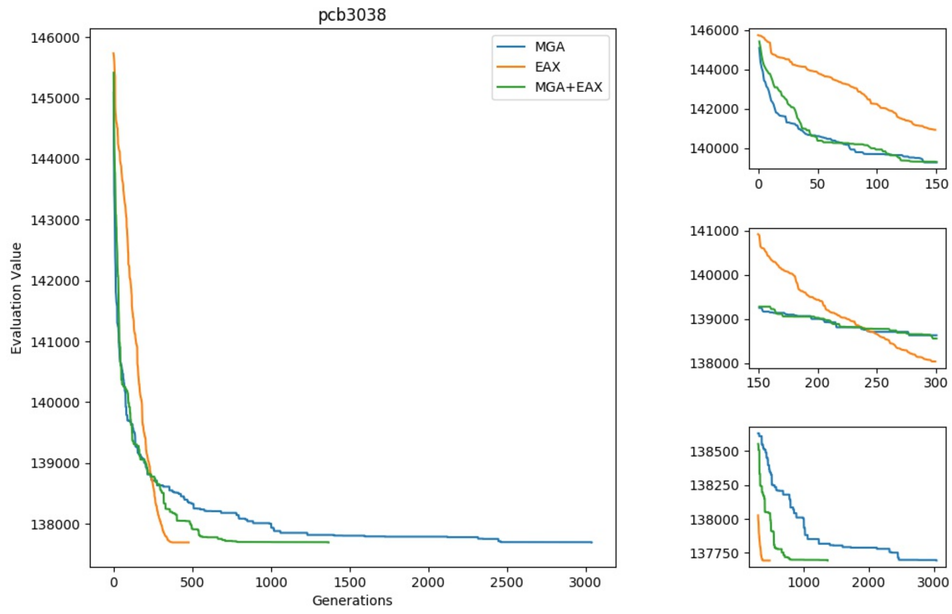


Figure 4.12: Progress of EAX and MGA and the hybrid MGA+EAX on pcb3038 for popsize=300.

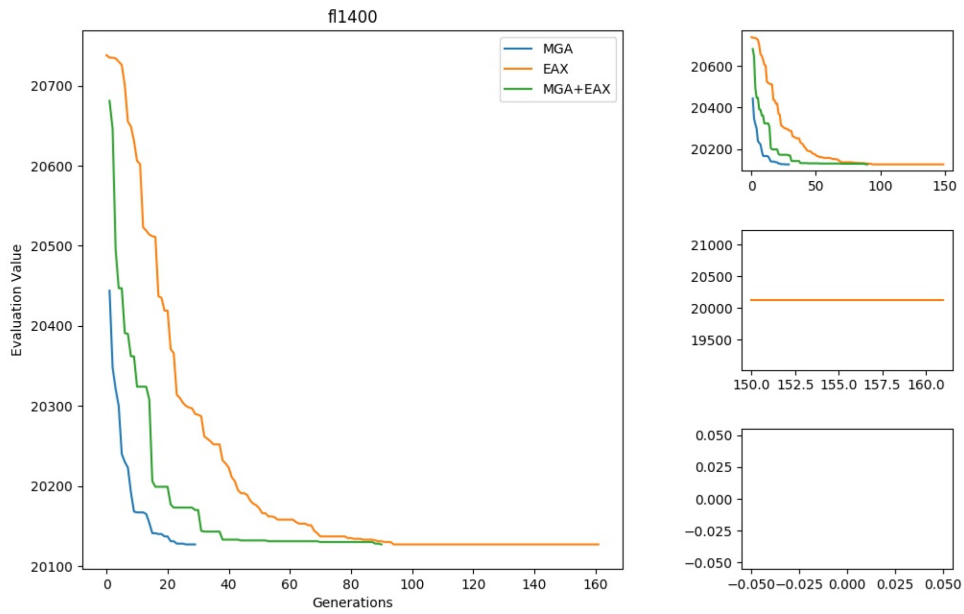


Figure 4.13: Progress of EAX and MGA and the hybrid MGA+EAX on fl1400 for popsize=300.

Table 4.4: Frequency of global edges in the final population, sorted by EAX results. The particular global optima were obtained using the LKH algorithm. For over half the instances, edges in the global optimum make up over 98% of all edges in the EAX population. This is good if EAX converges, but also can lead to premature convergence. MGA+EAX is only slightly more converged than MGA. The pop. size is 300.

Instance	EAX	MGA	EAX+MGA	Instance	EAX	MGA	EAX+MGA
u2319	62.88	53.72	62.36	rl1323	98.09	87.12	87.98
fl1400	66.29	55.88	56.58	d1291	98.44	86.56	86.77
pcb442	90.83	69.15	70.7	vm1748	98.69	81.9	99.43
fl1577	92.93	80.39	89.15	pr2392	98.74	75.07	78.73
vm1084	93.48	80.54	81.36	d1655	98.82	79.48	83.03
att532	94.75	71.97	73.47	C3k.0	98.9	71.61	71.99
u1817	95.59	78.62	86.03	C3k.1	98.91	70.78	71.26
nrw1379	96.71	69.97	75.42	fnl4461	98.96	69.62	78.06
pcb1173	97.26	74.57	77.52	d657	99.19	73.04	74.96
pcb3038	97.67	72.64	77.81	pr1002	99.55	70.53	72.77
rl1889	97.69	84.48	85.4	C1k.0	99.56	71.48	72.14
d2103	97.75	86.25	88.59	dsj1000	99.61	71.96	72.6
				rl1304	99.79	87.24	87.5

optima. When we include edges from *all* of the global optima, the percentage of global edges for fl1400 increases to 88.55%. For u2319, it grew to 71.32%.

4.3.4 Multiple Global Optima and Frequencies

On average, the problems with multiple global optimal are more challenging to solve for evolutionary algorithms. However, some problems are also easy if the multiple global optima are closer together. Let us consider four instances: fl1400, fl1577, u2319, and att532.

The fl1400 has at least 30 global optima. The total number of distinct global edges is 2821. Out of these, 469 of them are shared amongst all 30 global optima. Given any two global optima, they share 946 (67.57%) of their edges. Nevertheless, this is an easy instance. This may be because the global optima are not too far apart. Thus, the search can converge to any of the 30 solutions.

Instance u2319 is a compelling case because it is a complex problem to solve. From Table 3.1, a population as small as 64 is sufficient to contain all the global edges. Yet, MGA was not able to solve u2319 when using a population size of 16,384. Moreover, EAX in its default configuration also fails to solve u2319. But, the hybrid MGA+EAX was able to achieve a 73% success rate.

There are two global optima for u2319. The number of edges common between the two global optima is 1484. However, each global optima has 835 edges that are not common (where $1483 + 835 = 2319$). Thus the total number of distinct edges found in both global optima is 3154 ($1484 + 2 * 835$). This means these two solutions are pretty far away in the search space, which can prevent a population-based search method from converging. It can also cause a problem for branch and bound methods. However, we would expect an iterated local search methods such as LKH would not have the same problem, and indeed the empirical results support this hypothesis [112].

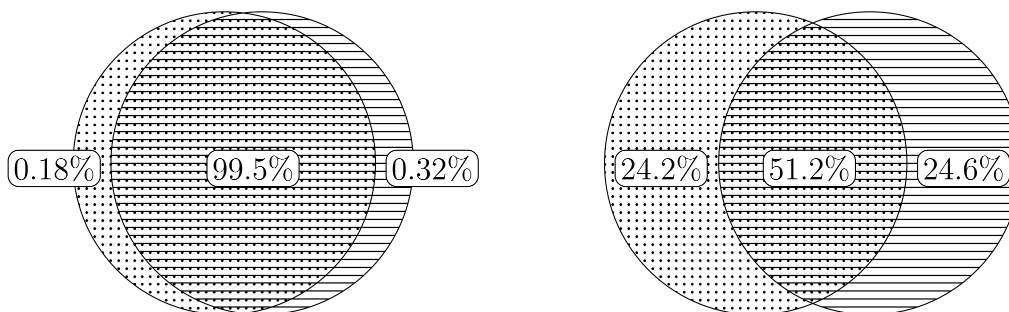


Figure 4.14: Overlap percentage of the two global optima edges for a population of 300 on (a) att532 (the top venn diagram) and (b) u2319 (the botton venn diagram) instances

Figure 4.14 shows a Venn diagram demonstrating the overlap of the two global optima edges of att532 and u2319. Both att532 and u2319 have two global optima. However, att532 is an easy instance, but u2319 is hard. This is because the edges common between the two global optimal solutions for att532 is 99.5% and only 51.2% for u2319. Thus, the distance between the two global optima affects the ease of convergence.

The fl1577 is a difficult instance for the hybrid and EAX to solve. However, MGA achieves a 100% success. The presence of EAX is not favoring the search because the EAX gets stuck at some local optima. Ochoa et al. [3] make a similar observation for fl1577 using the Chained LK solver. They observe that the local area network for fl1577 does not have shared local optima, and the landscape has large “plateaus”. A plateau comprises of a group of local optima with the same objective function. They observed that the search often gets stuck in a non-optimal plateau.

To summarize, the presence of multiple global optima affects the performance of the solver. All three solvers achieve a 100% success rate for 17/25 instances. However, for eight instances, the behavior is different. Table 4.5 summarizes the rationale behind it. For 5/8 of these instances, the reason is due to the presence of multiple global optima. For 2/8 insances, hybrid is superior. Overall, the hybrid solver is helpful for 7/8 instances.

Table 4.5: Summary of mixed success rates and behaviors of the three algorithms on 8 instances

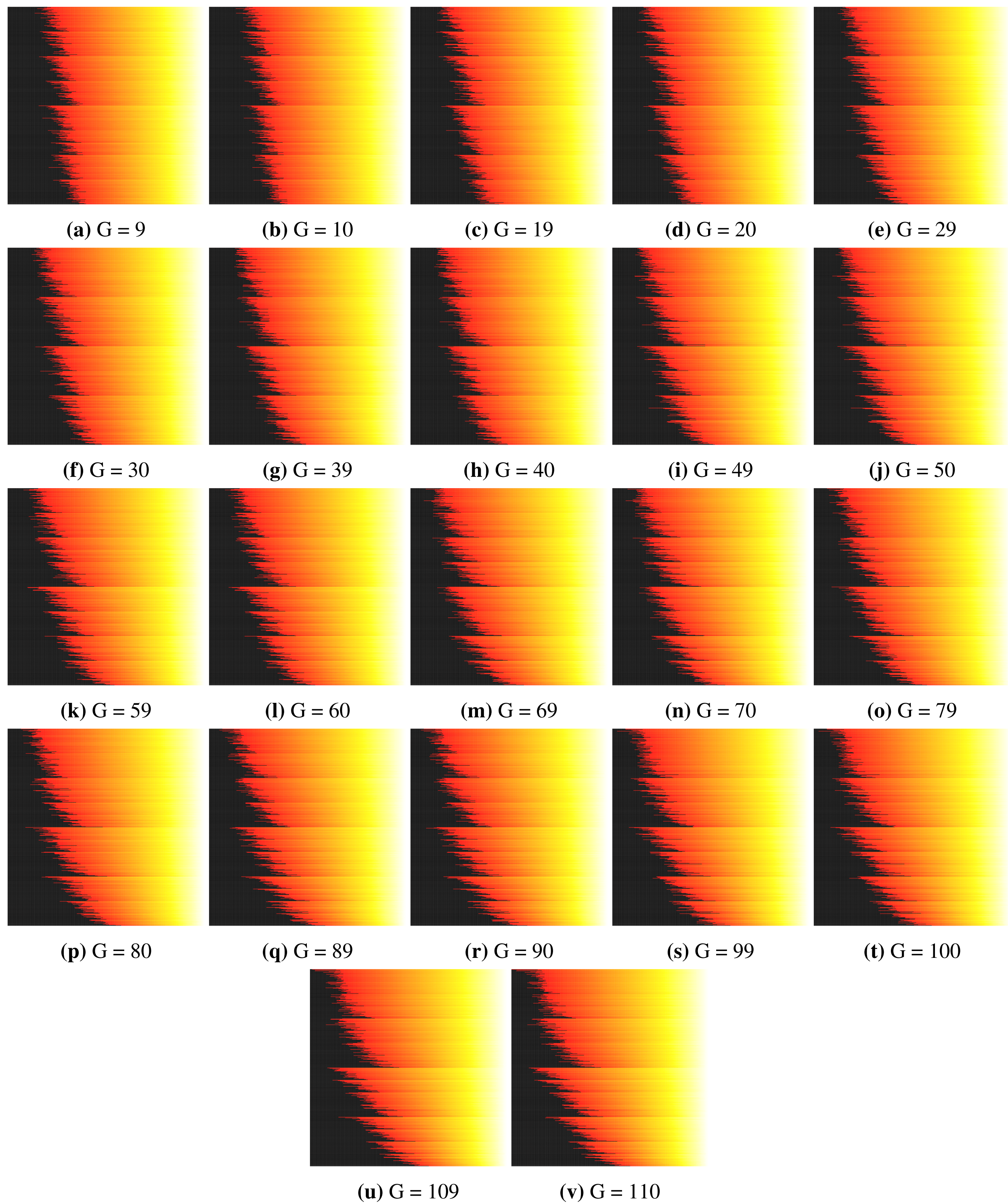
Instance	Best solver	Comments – why other solver did not work?
u2319	Hybrid	multiple global optima
fl1577	MGA	mulitple global optima
d1655	all	97% success rate on EAX-optimal is considered as an outlier.
u1817	Hybrid and MGA	multiple global optima
d2103	Hybrid and EAX	MGA obtains 90% success rate. There was a need to use a large population (16,384) to contain all P* edges Hence, the reason is the low frequency of important P* edges.
pr2392	Hybrid and EAX	multiple global optima
C3k.0	Hybrid and EAX	higher problem size, leads to slow convergence for MGA
fnl4461	Hybrid	multiple global optima, higher problem size, leads to slow convergence for MGA

4.3.5 Final population visualization

Figure 4.15 shows the population for the hybrid MGA+EAX algorithm. We have similar results for other problems instances. The global optimum is found in the first row of figure 4.15v. The best edges are concentrated in the “top" of the population.

EAX recombination was applied to this population for only 22 generations. On average, this increased the number of global edges from 72% to 73.5% of the population. This is only a 2% increase compared to the heatmap presented in Figure 4.6. EAX running alone will not create this concentration at exactly generation 22, but it will achieve this concentration early in the search.

The empirical data suggests that both EAX and the hybrid MGA+EAX could potentially converge much faster. For EAX, the entire population converges to a configuration that very close to the global optimum; this happens relatively early in the search. But given that the EAX pop-



(a) G = 9 (b) G = 10 (c) G = 19 (d) G = 20 (e) G = 29
(f) G = 30 (g) G = 39 (h) G = 40 (i) G = 49 (j) G = 50
(k) G = 59 (l) G = 60 (m) G = 69 (n) G = 70 (o) G = 79
(p) G = 80 (q) G = 89 (r) G = 90 (s) G = 99 (t) G = 100
(u) G = 109 (v) G = 110

Figure 4.15: The heatmaps on MGA+EAX population on att532 with a population of size 512. Here, one epoch of MGA (9 generations) is followed by one generation of EAX.

ulation is quickly “close to optimal" in terms of the high concentration of global edges in the population, it takes EAX a surprising amount of time to actually reach the global optimum. Again, the concentration of global edges in figure 4.15 resulted from only 22 generations of EAX. The EAX algorithm, on average, required 62 generations to converge to global optima. The default termination conditions took 163 genrations to converge.

Given the way that EAX does recombination, it must always “construct" the global optimum by merging subcircuits. EAX has no way to just recombine the global edges already in the population. Nevertheless, EAX still requires diversity for crossover to be effective: if the parents are identical, there are no AB-Cycles for cutting Parent 1 into subcycles. The EAX algorithm terminates when the difference between the average and best tour length is less than 0.001, which means there is very little diversity left in the population. This is why the restarts used by Mu et. al. [113] help.

While the EAX algorithm runs the risk of being too greedy, the hybrid MGA+EAX does not appear to be greedy enough: it does not spend enough time exploiting the best solutions, and it is not discarding very poor solutions. This also slows down convergence. More work is needed to better understand what these population based algorithms are actually doing in order to continue to improve population based search methods for the TSP.

4.3.6 Execution time

While the hybrid MGA+EAX algorithm is able to solve problems using fewer evaluations than EAX, the EAX code is still approximately ten times faster than the hybrid MGA+EAX algorithm on average. While that is not important for understanding problem structure, it might have implications for constructing a viable hybrid algorithm.

The EAX code was obtained from Y. Nagata [92] and the GPX code was obtained from R. Tinós [100]. Building extremely fast code takes considerable time and resources. The EAX code is clearly much more optimized for speed. EAX uses data structures that are between 3 to 4 times larger than the space required by GPX. The current GPX code spends additional time looking for recombining components using “fusions" of AB-Cycles. Many recombining components are

found without these fusions. A order of magnitude speed-up might be possible just by avoiding fusions. The current GPX code also does not exploit the fact that children replace parents. If Child 1 replaces Parent 1, Child 1 can be constructed in place, only replacing the components in Parent 1 that are inherited from Parent 2 to yield Child 1. Finally, the EAX crossover operator also contributed to the additional runtime because EAX cannot pass its data structures from one generation to the next generation in the hybrid MGA+EAX algorithm.

4.4 Conclusions

The limitation of MGA is that it assumes the initial population will contain all "global optimal" edges. However, in reality, one cannot guarantee this assumption. In this chapter, we studied this assumption more in detail. We found out that the initial population created by the 2opt local search often contains all the global optima edges. 99% of these edges are found within the first 50 neighbor lists of each city. Thus, using a 2opt local search that uses the nearest neighbor list concept is an excellent way to initialize MGA.

Next, we studied how these edges affect the performance of the solver. We saw that the presence of multiple global optima edges could make the search easy or hard. We measured the distance between these global optima in terms of the number of edges shared. We saw that the difficulty of the search is directly proportional to the distance between the multiple global optima.

Next, we introduced a simple hybrid of EAX and MGA. We saw that the hybrid solver is superior to EAX and MGA regarding the number of recombinations to global optima. It also helps to solve 7 of 8 problematic instances. It achieves 73% on u2319, the one example on which both EAX and MGA failed. The hybrid solver, however, calls EAX once every epoch of MGA. Thus, the ratio of EAX and MGA influence on the search is $1 : log(pop.size)$. Furthermore, the visualization helps to observe how these solvers manipulate the edges. The MGA filters and pushes the global optima edges to the top of the population. EAX filters and removes the non-global edges from the population. These observations suggest that with careful analysis, the properties of EAX and MGA can be combined to solve a wide variety of TSP instances.

Finally, the MGA and hybrid solver is superior in terms of the number of recombinations. However, the execution time is slow. MGA also has slower convergence for larger problem sizes. We can improve the execution time by optimizing the code and by using parallel programming techniques. We discuss some parallel programing approaches in chapters 5 and 6. Additional research is needed to improve the convergence. In this thesis, we explore three directions.

The first direction is to explore different hybrid versions of EAX and MGA. It will be worthwhile to find the appropriate balance between the two solvers. In chapter 5, we introduce yet another hybrid solver for larger problem sizes. The second direction is to develop some sophisticated selection mechanisms for MGA recombinations. Currently, random individuals recombine in the hypercube topology. But, we do not decide whether the recombination will be helpful or not. Understanding the usefulness of the recombinations could help build a faster MGA. The third direction to improve convergence is to use different diversity preservation techniques. The MGA, in its current form, does not lose any diversity in the population. However, this can lead to slower convergence. In the next chapter, we introduce some "replacement" techniques that lose diversity.

Chapter 5

Distributed memory parallelization of MGA

The Edge Assembly Crossover (EAX), is one of the state-of-the-art TSP solver. It is found to be the single-best solver [16] for a wide variety of problems smaller than 2,000 cities. However, there are no known comparitive study made for larger problem sizes. The multi-core version of EAX can solve upto 200,000 city problems. However, this version of EAX does not fully exploit the inherent parallelism. In this thesis, we want to develop a GA solver than can maximally utilize the inherent parallelism and is efficient for larger problem sizes. To this end, in Chapter 3, we developed the Mixing Genetic Algorithm (MGA) using the Generalized Partition Crossover (GPX).

The GPX crossover operator consumes 4x less memory, and does not incur communication costs to the distance matrix. These features make GPX operator favourable not only for multi-core CPU machines, but also for massively parallel SIMD machines. The MGA using the GPX has been tested on 25 instances smaller than 5,000 cities. The results show that MGA converges fast fewer number of recombiantions and on higher population sizes. The success rate is 100% for problems smaller than 2,000 nodes. However, it achieve a 100% success only for 2/7 problems larger than 2,000 nodes. The convergence rate is extremely slow in terms of the execution time.

One way to improve the exectution time of the solver is by population-level parallelization. In chapter 3, we saw the population-level parallelization on the shared memory architecture. Even with the parallelization on the shared memory architecture, the execution time was about 300$\times$ slower compared to the EAX. In this chapter, we increase the degree of parallelism by using both the shared memory and distributed memory parallelization.

The parallel units in a distributed memory architecture do not share the memory. Figure 5.1 shows a typical distributed memory architecture, where the parallel units are connected via the interconnect system. Note that each parallel unit, in turn, has its own memory system. On a distributed memory system, there is a need to explicitly migrate the individuals between the processors. We explore two different communication protocols namely, the “migrate” and the “replace”.

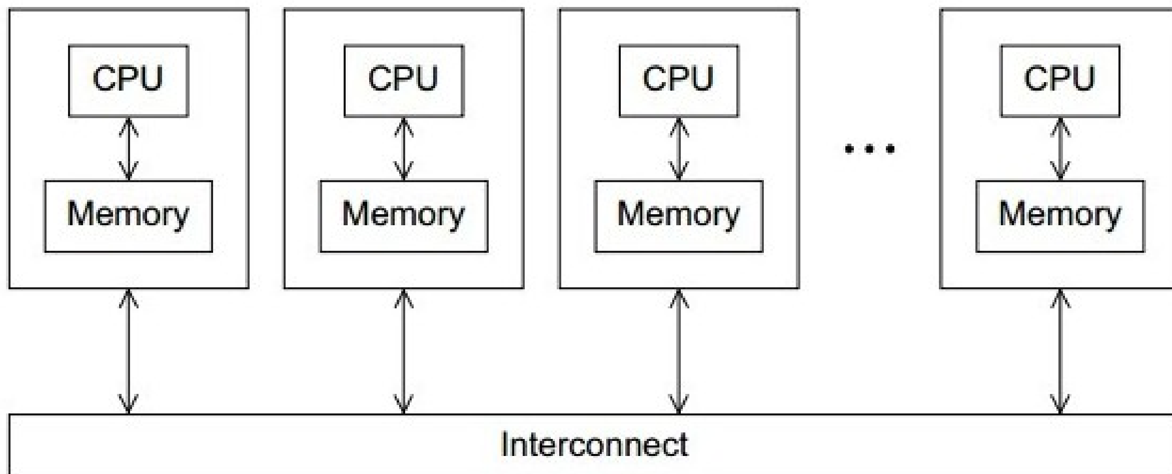


Figure 5.1: In a distributed memory system, the parallel units do not share memory. An interconnect system is used for communication. Each parallel unit has its own memory and input/output system.

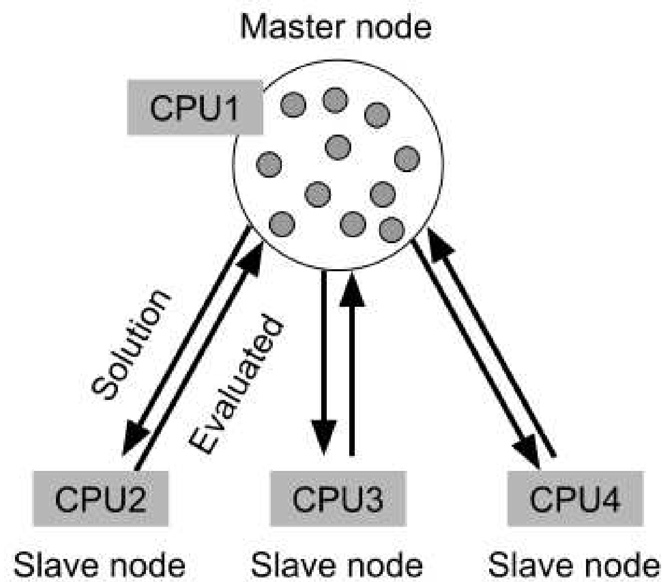
We also explore two different topologies of the processors namely the "ring" and the "hypercube". The topology and communication protocol determines the performance of the solver. Overall, we develop four different parallel MGA models.

From chapter 4, it is clear that MGA is fast in the initial stages because of the GPX operator. Also, a hybrid of EAX and MGA can solve some hard TSP instances with a higher success rate. The hybrid takes fewer recombinations than MGA to converge. Thus, the hybrid versions are better than the individual solvers. We leverage these properties and use MGA as a pre-processing step for EAX on problems as large as 85,900 cities.

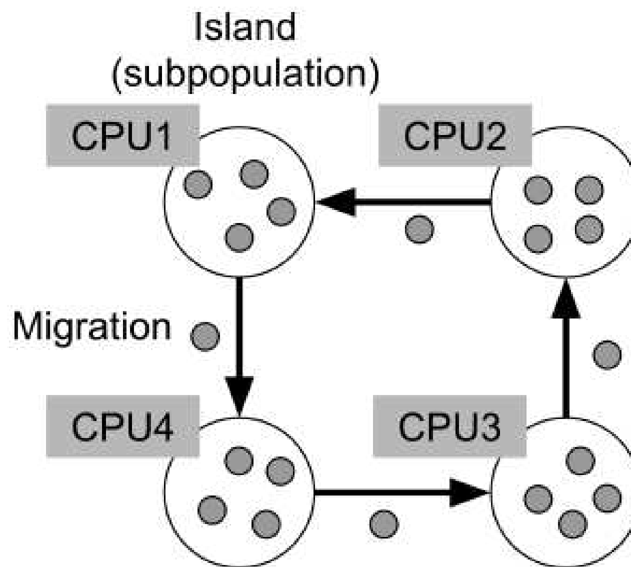
The results indicate that the hybrid versions have their pros and cons. There isn't one best implementation. However, when combined in an ensemble setting, executing all solvers in parallel, the ensemble helps solve some difficult TSP instances. We explore an ensemble of solver in combination with EAX and LKH solvers. A part of this chapter is published in GECCO 2021 [27].

The rest of the chapter is organized as follows. Section 5.1 presents the background and related work. The parallel MGA introduced in this chapter is discussed in section 5.2. The experiments and results on hybrid algorithms are presented in section 5.3. The usefulness of the solver in an ensemble setup is presented in section 5.4. Finally, section 5.5 concludes the chapter.

5.1 Background



(a)



(b)

Figure 5.2: Figure showing examples of two Parallel GA models [4] . (a) Master-slave model: The master node contains the population. It sends an individual to the slave node to perform a genetic operation. The slave node performs the genetic operation and sends the individual back to the master node. The slave nodes do not communicate with each other. (b) Island Model: The population is divided amongst the four islands (CPUs). The islands communicate with each other by migrating the individuals in the population.

The design of population-level parallelization for distributed memory architecture depends primarily on the parallel programming model of the GA. Figure 5.2 shows two parallel models. The master-slave model is where the entire population is managed by the master node. The master node divides the population equally and sends them to the slave nodes. The slave nodes do not communicate with each other. The slave nodes, perform the designated operations in parallel. For example, the slave nodes of the multi-core EAX algorithm [104] generates the 30 offspring solutions and communicate it back to the master node. The master node decides which child will replace the parent using the diversity preservation metrics.

The second popular model is the island model. Here, a single population is divided into sub-populations. In each processor, sub-populations evolves separately. In figure 5.2, the sub-population is divided among four CPUs. The individuals may or may not migrate between islands. Most of the GA-based TSP parallelization [107], [108], [20] is based on the island model.

The master-slave model is suitable for EAX algorithm because, to replace a single parent, the EAX depends on the edge-entropy of the population. As discussed in chapter 4, section 4.1, the edge entropy depends on the distribution of the edges of the entire population. Therefore, the slave nodes generate the offsprings, but, they do not replace the parent. Instead, the master node collects all the offsprings from the slave nodes, calculates the edge-entropy and then updates the population. For MGA, there are no such dependencies. The sub-populations can evolve independently. Therefore, we select island model implemenation for the MGA.

The design involves four significant parameters namely, (a) number of parallel units, (b) sub-population size, (c) processor topology and, (d) migration policy. Here, the MGA population is divided equally amongst the given parallel units. However, unlike the shared memory model, there is a need to design the communication protocols explicitly. The communication protocol depends on the topology and migration policy of the islands.

5.1.1 Island model topologies

The interconnect of the Island model in figure 5.1 must have a topology associated with it. We are not interested in the “hardware topology”. We are interested in the design of a “software topology”. Software topology policies determines the efficient performace of the solver. Processors connected in the hardware topology may not be connected via the software topology. However, it is necessary for the processors connected in the software topolgy to be connected via hardware. Figure 5.3 illustrates some examples of island model toplogies.

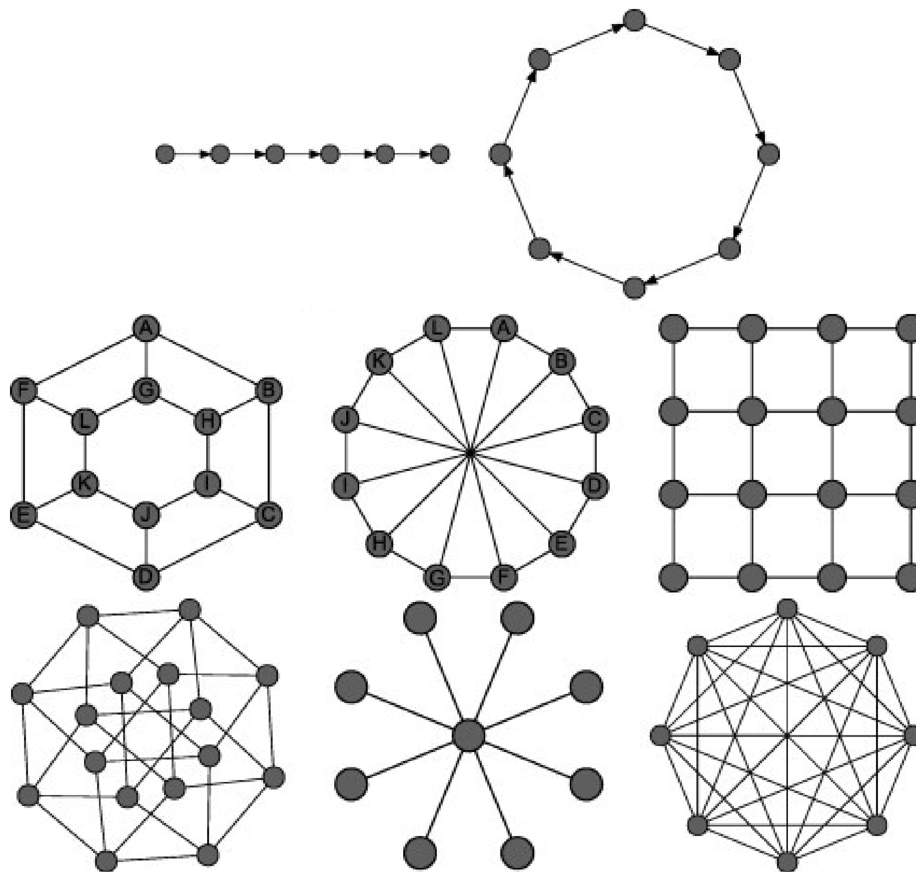


Figure 5.3: Figure showing examples of different island model topologies [4]. Top row (left to right): chain, ring. Middle row (left to right): toroid, cartwheel, mesh. Bottom row (left to right): hypercube, star, fuly-connected

5.1.2 Island model migration policies

The migration or movement of individuals between the islands usually depends on the type of topology. For an island model, the frequency of migration and the selection/replacement policies must be configured. The frequency of migration determines when to migrate the solutions. The selection/replacement policy determines which individual to migrate and replace in the target island. Each island can be designed to follow a homogenous policy. However, to promote diversity, heterogenous policies are used. In the homogeneous structure, all islands have the same search strategy. Whereas, in the heterogeneous structure, each island has different strategies. For example, islands can have different population sizes, different probability of genetic operators [114] and in some heterogeneous model, each island performs different evolutionary algorithms [115].

Wang et al. [107] study five different parallel island model approaches to TSP. Some of their models evolve independently without migration. However, they find that the models with migration is the best approach. Kang et.al [20] use the island model implementation on the GPUs; wherein, they divide the population into eight groups and exchange the best individual at random. However, the state-of-ther-art TSP solver, EAX [78] does not employ any migration techniques.

5.1.3 Other parameters

To summarize, the two main design parameters for the island model are the topology and migration policy. Dynamics of configuration is another aspect of the island model. In the static configuration, a configuration of search is constant during the optimization process, while in the dynamic configuration, a configuration may change during the optimization process depending on the pre-defined manner. For example, the islands can use the same genetic operator or change operators depending on solution quality. In the next section, we describe the parameters used in the island model design of Mixing Genetic Algorithm.

5.2 Island Model Mixing GA

The goals of parallelizing the Mixing GA is threefold

1. to quickly mix the alleles in all the islands.

2. to have a runtime strictly less than its sequential implementation.

3. to produce the same or better quality individuals for a given number of generations.

One epoch of MGA is sufficient to equally mix the alleles in the population. Consider a population of size 256, divided into eight islands with 32 individuals per island. As per the sequential MGA, the individuals in the population are selected for crossover based on the hypercube topology. Therefore, in the first generation, individuals in the first four islands must be recombined with the individuals in the last four islands. However, migrating all the individuals across islands will incur unnecessary communication overhead. To avoid this overhead, we use the mixing properties of MGA. Recall the Mixing theorem 1 in Chpater 3, section 3.2.2. The best individual migrates to the first position in the population at the end of every epoch. We can leverage this property. Let each island perform a local epoch. At the end of the local epoch, the first individual in each island will probabilistically have the best partitions. This locally best individual can migrate to other islands.

5.2.1 Mixing island sub-populations

Let us consider a population divided among 8 islands as shown in figure 5.4. As a first step, each island performs one epoch of MGA, represented by the block E_i. At the end of the local epoch, the best individuals are probabilistically found at the first position. The green and the red circles represent the best and worst individuals at the end the local epoch. Now, we want to mix these best individuals amongst the islands. We follow the same hypercube communication pattern amongst the islands. The communication steps are represented by C_i block.

The first communication pattern happens between the two halves of the hyperplane. The first half of the hyperplane consists of islands 1,2,3 and 4. The second half consists of islands 5,6,7 and 8. The best individual in the first half of the islands rare exchanged respectively with the best

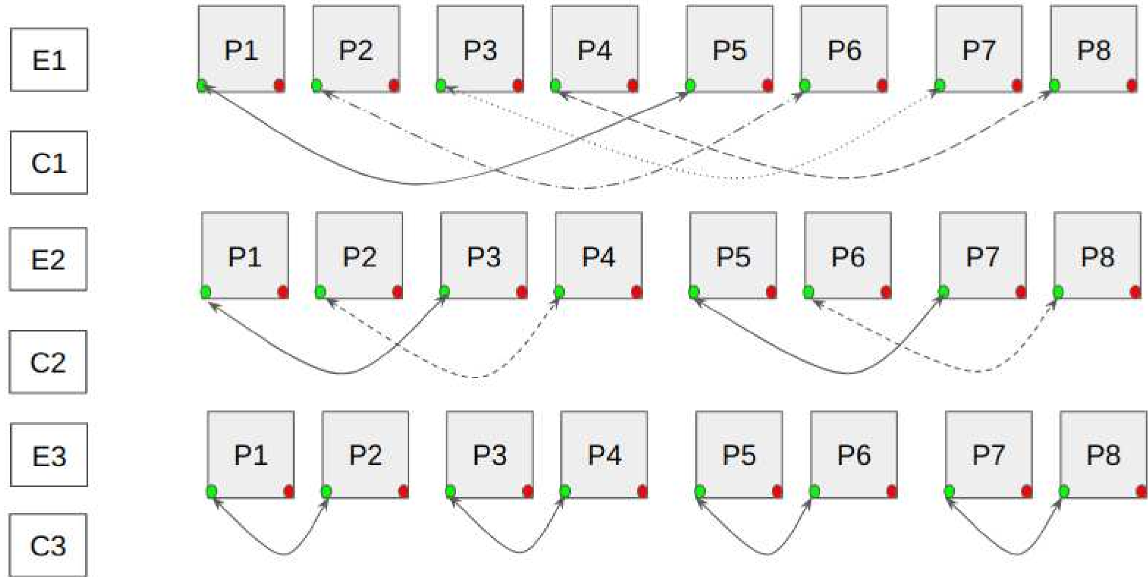


Figure 5.4: Hypercube topology - Migrate policy

individuals from the second half of the islands. Here, we do a simple exchange so that there is no loss of diversity in the population. Thus, the properties of sequential MGA is preserved.

After this local epoch and communication step, each island performs another local epoch, represented by the E_2 block. In this local epoch, every island has the opportunity to mix the best partitions from two islands (itself and from its partner). At the end of the second local epoch. the second hyperplane communication takes place. Now, the hyperplanes divide the islands into four groups. The best individual from the first quadrant (island 1,2) is exchanged with the indivudals from the third third quadrant (island 5,6). Similary, the best individuals from the second quandrant (island 3,4) is exchanged with the individuals from the fourth quadrant (island 7,8). This communication is represented by the C_2 block.

Similar to an epoch, consisting of $log(N)$ generations, the migration pattern consists of $log(P)$ local epochs and communication. We call this set as one **batch**. Thus, at the end of one batch, the individuals are guarenteed to have the opportunity to equally mix amongst the islands. For the population in figure 5.4, one batch consists of three local epochs and three communication steps.

5.2.2 Replacement policy

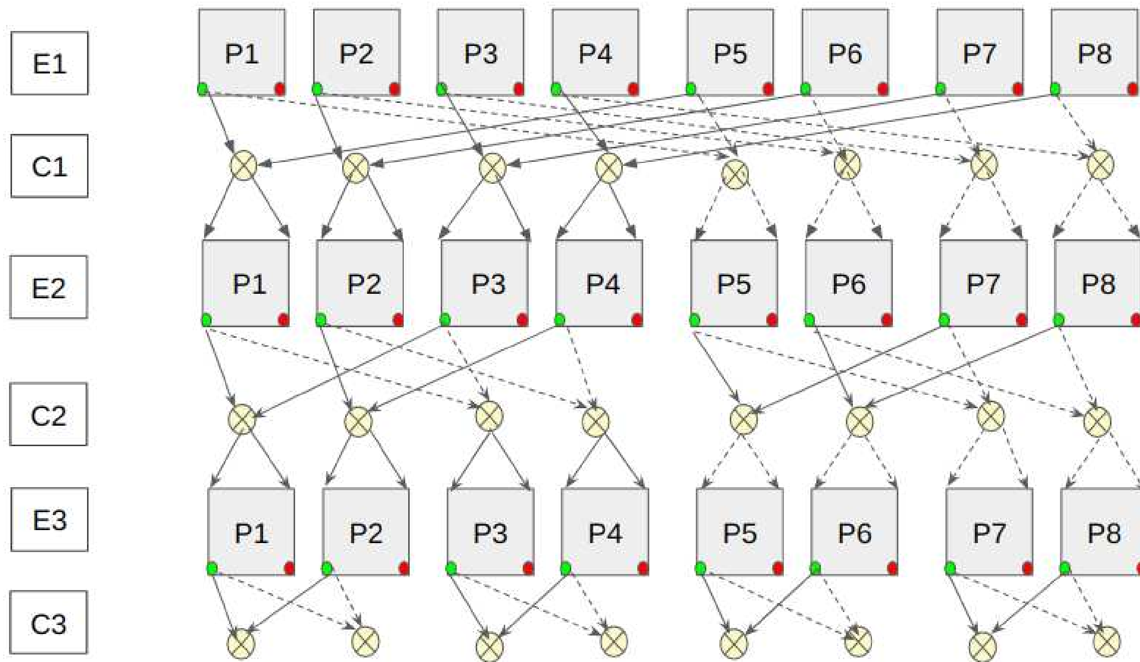


Figure 5.5: Hypercube Topology - Replace policy

The mixing of the sub-populations does not lose any diversity. However, based on observations from chapter 4, MGA can converge more quickly by systematically removing the unimportant edges. In this section, we introduce the replacement policy, to remove some unwanted edges.

Consider the same eight islands in hypercube topology. A batch will now have three local epochs and three communication steps. However, instead of a simply exchange, the individuals are recombined. The best and worst of this recombination replaces the best and worst individuals in the island. Therefore, the worst individuals in the islands are replaced at the end of every communication step. Figure 5.5 ilustrates this concept. Here, the crossover operation is represented by the yellow circles. The communication step, represented by C_i includes a crossover operation and replacement of the best and the worst individuals in the population.

5.2.3 Ring Topology

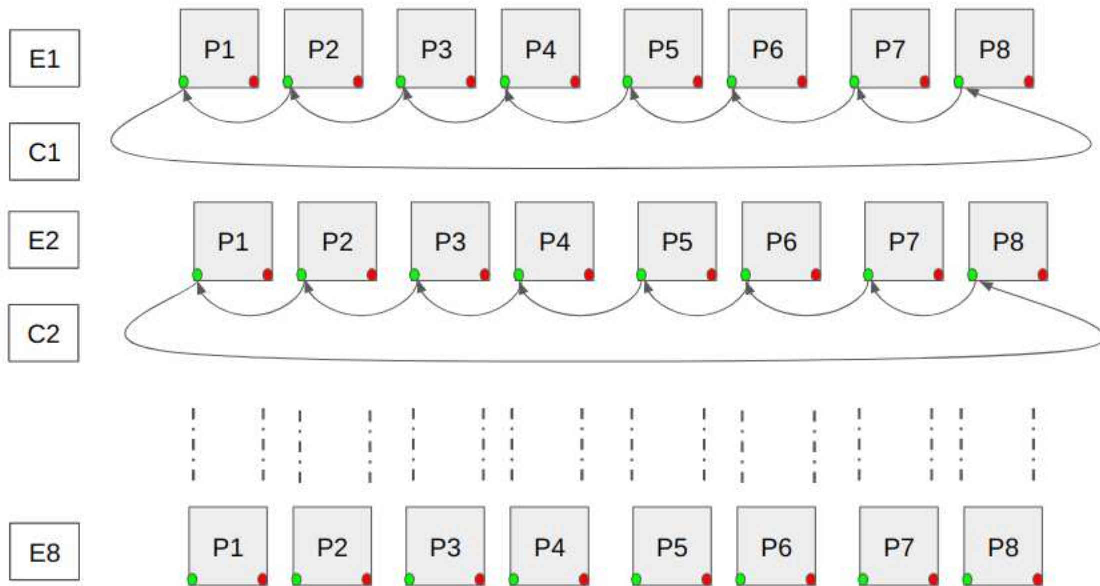


Figure 5.6: Ring topology - Migrate policy

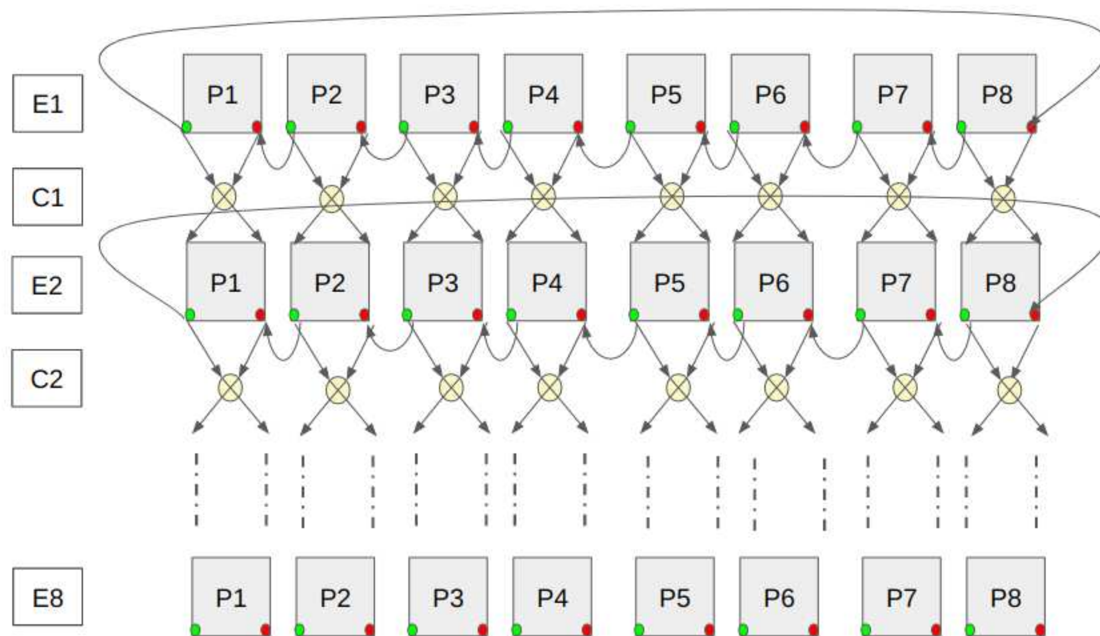


Figure 5.7: Ring Topology - Replace policy

The hybercube topology takes $log(P)$ steps to achieving one full mixing. However, if the hardware topology does not match with the software topology, the communication cost required to migrate the individuals in a hypercube topology may be unnecessarily high. To have an alternate solution in such cases, we consider a simple ring topology. Here the communication happens only between the left and right neighbour of the processor.

Figures 5.6 and 5.7 illustrates the migration and replacement policies. The islands are numbered P1 through P8. The label E1 says that each island performs one local epoch in parallel. The red and green circles denote the worst and best individuals at the end of an epoch. In the migrate policy, the best individuals are migrated to the left neighbour in a ring fashion.

With the replacement policy, the best individual replaces the worst individual of its left neighbour. Then, the best local individual and best migrated individual are recombined. This communication step is denoted as C1. It is followed by the next parallel epoch. At the end of 8 parallel epochs and 7 communication steps, we achieve the mixing of individuals across islands.

5.2.4 One batch of island model MGA

Algorithm 3 shows the steps involved in one batch of MGA. Note that there are two communication protocol depending on the topology of the islands. There are also two mixing policies. Therefore, there are four different versions of parallel MGA namely, (a) hypercube - migrate, (b) hypercube-replace, (c) ring-migrate and, (d) ring-replace.

Algorithm 3 One batch of parallel MGA

1. Apply one epoch of MGA
2. Apply the **communication protocol** (Hypercube/Ring)
3. Apply the **mixing policy** (Migration/Replacement)

The number of epochs and communication *per batch* is given by,

$$\begin{aligned} gen_{ring} &= P * log_2(popsize/P) \\ epochs_{ring} &= P \\ comm_{ring} &= P - 1 \end{aligned} \tag{5.1}$$

$$\begin{aligned} gen_{hypercube} &= log_2(P) * log_2(popsize/P) \\ epochs_{hypercube} &= log_2(P) \\ comm_{hypercube} &= log_2(P) \end{aligned} \tag{5.2}$$

where $popsize$ is population size and P is the number of processors (a.k.a, islands)

5.2.5 Experiments on small problem sizes

Algorithm 4 shows the complete steps of the island model MGA. The population are initialized in parallel. The local search is performed in parallel. The sub-population is locally randomized at the end of every batch. Note that there are four versions of parallel MGA. Thus, what happens inside a batch depends on the type of topology and migration policy chosen.

Algorithm 4 Parallel island model MGA

1. **do in parallel**
2. Randomly initialize the population.
3. Apply 2opt on the population.
4. **while** (termination is not met)
5. Apply one **batch** of parallel MGA.
6. Randomly shuffle the population.
7. **end while**
8. **end do**

Termination condition

MGA and EAX terminate if the algorithms find the optimal solution or if the run takes more than 24 hours. For instances that do not have a known optimal solution, we use the best known solution to terminate. In case these conditions are not met, the algorithms use their default condition to terminate. By default, MGA ends if it reaches 10,000 generations. The EAX code ends when the difference between the best and the average tour length of the population is less than 0.001.

Program setup

The proposed GA is tested on 10 TSPLIB [116] instances ranging from 442 to 1655. The EAX code is from Nagata and Kobayashi [15]. The MGA code is openMP [1] parallelized, to run the crossover operations in parallel within a single core. For the island model versions, we use open MPI [2]. For parallel MGA versions, we use the *MPI_SendRecv* command for communication. The parallel EAX code [104] is not publicly available. We use the sequential version of EAX.

Machine and time settings

The experiments were run on the general nodes of the Rocky Mountain Advanced Computing Consortium (RMACC) [3]. Each node has 2 CPUs, and each CPU has 12 cores. Therefore, it is possible to run $450 * 2 * 12 = 10,800$ jobs in parallel. The codes are written in C++, compiled using icc $17.0.4$ with *fopenmp* flag. The MGA code is parallelized using OpenMP pragmas. This parallelization exposes fine-grain parallelism within a single core. EAX code is sequential. Each run is performed 30 times, average and standard deviation are reported. The timing routines in the OpenMPI library are used to measure the time. We do not report the runtime cost of using 2-opt to improve the initial populations; this cost is identical across all algorithms.

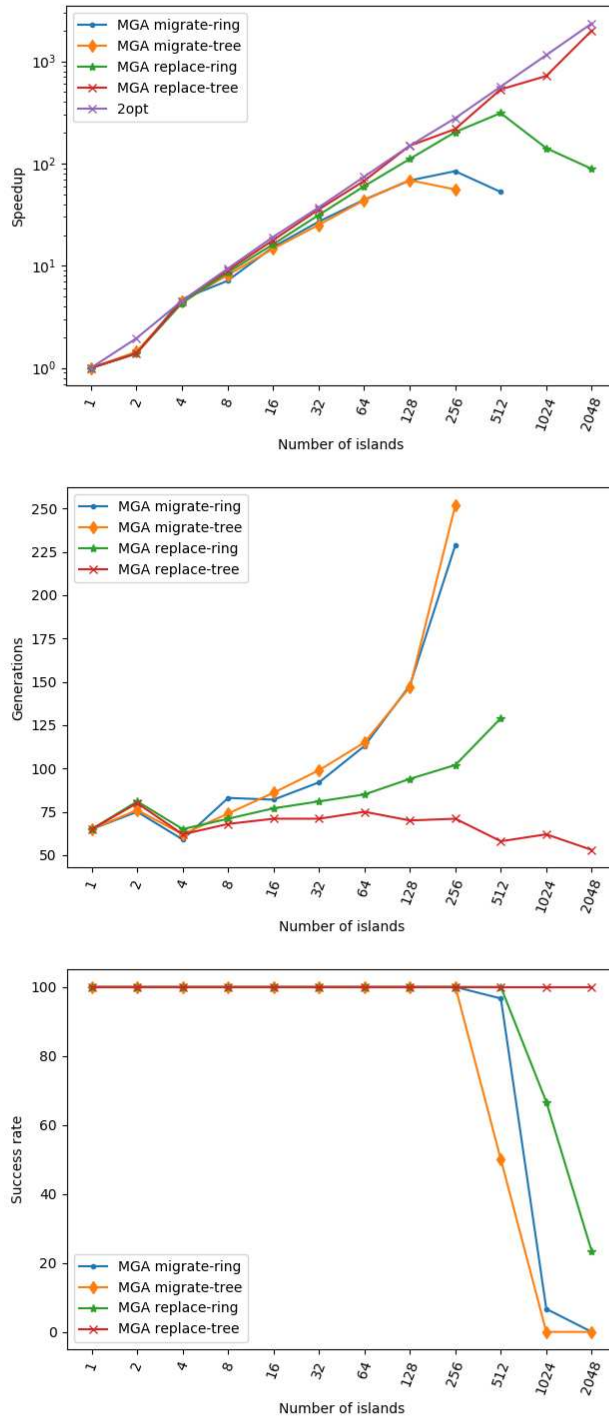


Figure 5.8: Plots showing performance of att532 on various parallelization techniques. The base of comparison is the MGA sequential code. The population size is fixed to 16384. The top graph shows the linear time speedup of 2opt and parallel MGA algorithms over the sequential version. Note that the y-axis is in log scale. The middle graph shows that the convergence rate in terms of number of generations to global optima. Note that both the replace strategy converges at a faster rate with increase in island . The bottom graph shows that success rates of all parallel strategies is 100% for upto 256 islands.

Effects on population scalability

Figure 5.8 shows the results on parallelizing att532 instance for a population of 16384. The number of islands is varied from 1 to 2048. The corresponding population per island ranges from 16,384 to 8. All strategies achieve linear speedup up to 128 islands. Beyond that, the migration strategies start to decline because migration does not remove the population's unwanted edges. With the decrease in the population per island, there is a prolonged period of stagnation. Therefore, migration strategies are not good for smaller island size. The replace strategy is able to give a speedup up to 512 islands. Now, the population per island is 32. Beyond this, the ring topology fails. The hypercube topology succeeds up to 2048 islands because the hypercube topology requires only $log(I)$ steps to achieve full mixing. The ring topology requires I steps and there is a drop in the convergence rate owing to loss of diversity and slow mixing.

The variation in the number of generations is interesting to note here. For ring topologies, it takes $I - 1$ transfers for the alleles (edges) in all islands to combine. This trend is valid with the migrate-ring topology. However, with the replace-ring topology, we see that the algorithm converges fast. It suggests two things (a) MGA could lose a small diversity to converge faster (b) A smaller population size is sufficient for att532 instance. This trend is observed clearly in the replace-hypercube strategy as well. The takeaway message here is the linear speedup with increasing islands.

Effects on problem size scalability

Table 5.1 shows the result of running small size instances in range 442 to 2103. The success rate does not reduce for all the versions. The parallel versions are much faster when compared to the sequential MGA version. For problems, less than 1000 nodes, the replace strategy achieves near-linear speed-up. For problems greater than 1000, migrate policies achieve super-linear speed-up. In terms of the number of generations, the replace-hypercube strategy outperforms every other

---

[1] https://www.openmp.org/

[2] https://www.open-mpi.org/doc/v4.0/

[3] https://www.colorado.edu/rc/resources/summit/specifications

Table 5.1: Results on applying parallel MGA versions. Results are compared with sequential MGA and EAX. MGA population size = 16384, No. of islands = 64. The EAX population size is 300, No. of children = 30. All algorithms achieve 100% success. (Exception: The Success rate of EAX on fl1577 is 3/30)

Instance	MGA seq.		EAX seq.	
	Gen.	Time	Gen.	Time
pcb442	61.38 ± 11.96	595.71 ± 99.71	27.19 ± 1.31	2 ± 0
att532	64.25 ± 7.01	878.26 ± 151.54	60.16 ± 2.28	5.78 ± 0.41
d657	56.31 ± 9.94	862.73 ± 313.85	69.72 ± 0.87	6.53 ± 0.5
dsj1000	60.5 ± 24.73	1465.87 ± 632.77	132.25 ± 1.98	14.28 ± 0.45
pr1002	99.81 ± 10.63	2527.75 ± 300.31	231.53 ± 2.89	19.25 ± 0.5
vm1084	48.19 ± 14.83	1281.27 ± 449.2	180.44 ± 2.95	16.84 ± 0.44
d1291	31.94 ± 5.57	962.52 ± 235.51	299.69 ± 8.97	43.56 ± 1.52
fl1400	22.91 ± 4.75	868.23 ± 171.66	229 ± 0	96.33 ± 0.47
fl1577	495.42 ± 108.36	25029.39 ± 6206.79	234.25 ± 3.76	36.94± 5.37
d1655	210.22 ± 57.65	12141.35 ± 3303.94	367.94 ± 5.84	109.69 ± 37.95

Instance	Migrate - ring		Replace - ring	
	Gen.	Time	Gen.	Time
pcb442	129.29 ± 44.71	15.5 ± 6.25	124.27 ± 48.17	18.41 ± 7.03
att532	116 ± 21.75	21.43 ± 3.68	113.1 ± 20.12	19.76 ± 1.91
d657	113.03 ± 20.4	24.35 ± 3.62	122.93 ± 22.95	27.91 ± 5.26
dsj1000	135.58 ± 34.09	21.29 ± 26.18	117.33 ± 31.78	12.73 ± 20.49
pr1002	265.14 ± 56.23	33.76 ± 47.51	264.89 ± 61.33	28.56 ± 46.14
vm1084	97.04 ± 27.03	35.59 ± 10.46	104.89 ± 17.46	11.6 ± 18.54
d1291	67.11 ± 16.47	12.03 ± 14.91	72.89 ± 12.19	9.37 ± 14.64
fl1400	31.27 ± 4.11	7.91 ± 8.4	32 ± 7.63	5.26 ± 7.95
fl1577	2477.33 ± 1166.23	244.72 ± 732.92	4608 ± 2274	893.45 ± 2034.12
d1655	343.27 ± 76.99	87.82 ± 141.46	328 ± 65.21	67.26 ± 104.05

Instance	Migrate - hypercube		Replace - hypercube	
	Gen.	Time	Gen.	Time
pcb442	81.25 ± 25.07	11.05 ± 2.73	66.25 ± 15.65	10.35 ± 2.84
att532	84.13 ± 18.94	12.91 ± 5.54	72.25 ± 17.03	12.75 ± 2.67
d657	79.5 ± 14.55	16.88 ± 3.59	63.75 ± 8.6	13.44 ± 1.72
dsj1000	75.75 ± 17.77	25.57 ± 5.84	69.75 ± 15.52	26.04 ± 7.04
pr1002	172.75 ± 42.61	65.67 ± 13.31	129.75 ± 43.48	49.09 ± 17.38
vm1084	74.5 ± 19.84	26.5 ± 8.31	73.87 ± 19.99	26.39 ± 11.58
d1291	51.75 ± 9.59	21.8 ± 3.63	48.27 ± 6.36	17.98 ± 6.69
fl1400	25.55 ± 4.73	11.66 ± 4.61	25.29 ± 4.11	13.37 ± 4.99
fl1577	652 ± 437.69	479.26 ± 344.89	742.61 ± 497.23	512.91 ± 443.54
d1655	209.25 ± 85.44	126.81 ± 39.24	180.27 ± 92.37	88.52 ± 42.29

strategy on 9/11 instances. Figure 5.9 shows the time speed-up of the parallel versions in comparison with the sequential version. Next, let us compare with the EAX versions in table 5.7. For three problems, the runtime of the parallel algorithm is comparable with EAX. However, in terms of the number of generations, the parallel MGA versions converge faster, suggesting that with a larger number of islands, the parallel MGA can outperform EAX.

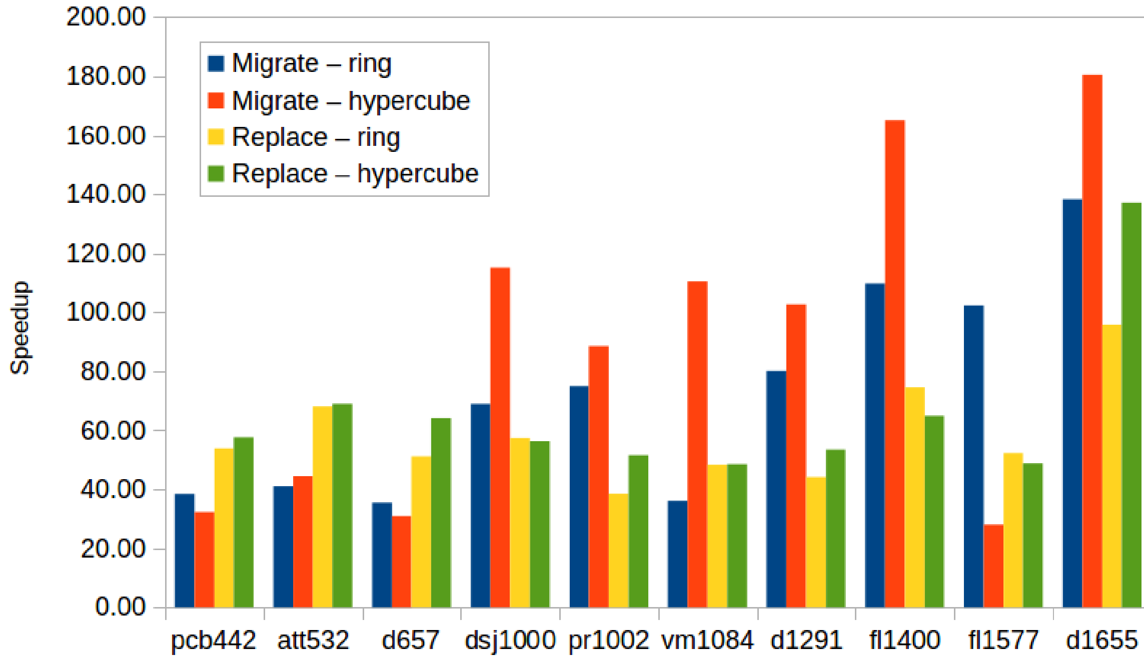


Figure 5.9: Plots showing the time speedup of the island model versions over the sequential code.

5.3 Hybrid algorithms

In Chapter 4, we observed that a hybrid of EAX and MGA (Hybrid-0) is better than the standalone version in terms of number of recombinations. We also observed that the MGA algorithm is highly exploitative in the initial stages. Pascal et al. [16] also suggest having a fast solver in the initial stage and switching to EAX is a promising future direction. Using these observations, we introduce a new hybrid algorithm wherein we run parallel MGA for **x** number of batches, followed by EAX. MGA is parallelized. Also note that the different parallel versions of MGA have differ-

ent mixing properties. Therefore, we have four hybrid versions namely, (a) hypercube - migrate (Hybrid-2), (b) hypercube-replace (Hybrid-3), (c) ring-migrate (H ybrid-4) and, (d) ring-replace (Hybrid-5).

The corresponding sequential version is to run the MGA for y epochs, followed by EAX. Let us call this sequential hybrid version as Hybrid-1. the number y is determined by the batch size selected for the parallel implementation. Refer equations 5.1 and 5.2 for relationship between epochs and batches.

Algorithm 5 Sequential MGA+EAX - Hybrid 1

1. Randomly initialize the population.
2. Apply 2opt on the population.
3. Apply **y epochs** of sequential MGA
4. **while** (termination is not met)
5. Randomly shuffle the population.
6. Apply one generation of EAX
7. **end while**

5.3.1 Experimental setup

We test the hybrid algorithms on problems ranging from 2,000 to 85,900. For hybrid versions, the EAX is called from within the MGA code. The data structures such as the distance matrix and nearest neighbor matrix are shared between the two codes. The parent tours are copied when switching between the algorithms because they use different data structures. MGA stores the tour as a permuation. EAX uses the next and previous arrays to represent the tour. Thus, there is some overhead involved in switching between the algorithms. The experiments are run on the same RMACC summit supercomputer.

Algorithm 6 Parallel MGA+EAX - Hybrid 2,3,4,5

1. **do in parallel**
2. Randomly initialize the population.
3. Apply 2opt on the population.
4. Apply **x batch** of parallel MGA
5. **end do**
6. Gather all individuals to the root node.
7. **while** (termination is not met)
8. Randomly shuffle the population.
9. Apply one generation of EAX
10. **end while**

5.3.2 Choice of batch size (x)

Figure 5.10 shows the trace of att532 on all versions. The population size is 16,384 for MGA algorithms. For EAX, it is 256. With 30 children, the number of recombinations per generation will be 7,680. This number is comparable with 8,192 recombinations per generation of MGA. It can be seen that MGA algorithms are highly exploitative in the initial stages because of the GPX operator. Between 40 and 60 generations, EAX trace intersects with MGA traces. This suggests that the choice of x should be somewhere in this range. This trend is observed for other problems as well. For our experiments, we consider 40 generations. With 256 as the population size, the x value will be $40/log(256) = 5$ epochs.

In tables 5.2 and 5.3, we show the results on sequential MGA+EAX for two values of x (5 and 2). These two versions are equivalent to one batch of ring and hypercube topologies, respectfully. Hence, it helps with comparing the parallelization effects. Comparing the sequential versions, we see that on three large problems - d15112, d18512, and pla85900, $x = 5$ converges fast. For these problems mixing the population more is helping. For the rest of the problems, the EAX is spending more time to remove the unwanted edges. Thus, there is a slight delay in the convergence rate.

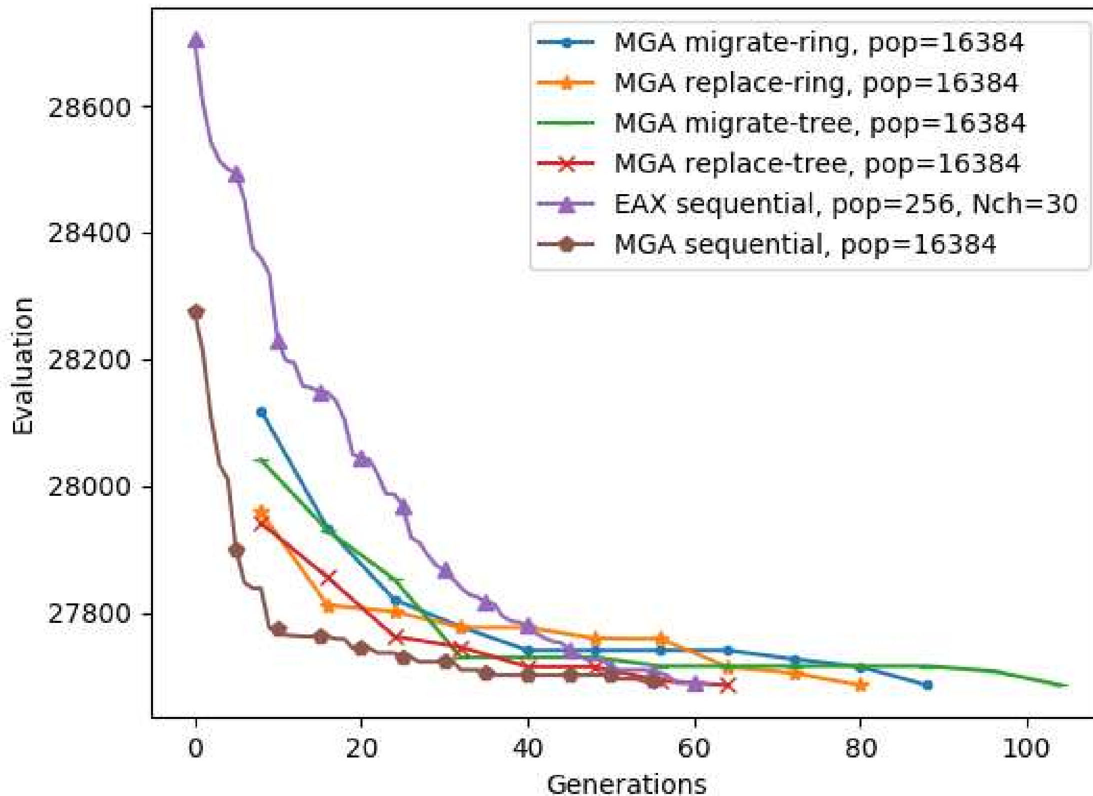


Figure 5.10: Plots showing a single run of att532 on all the four hybrid (MGA+EAX) island model versions, sequential EAX and sequential MGA. The MGA traces are quicker in the initial stages because of the GPX operator. EAX cuts the traces between 40 and 60 generations.

5.3.3 Results on hybrid algorithms

Tables 5.4 and 5.5 shows the results on ring and hypercube topology. Table 5.7 shows results for EAX, as well as the EAX and hybrid ensembles.

Effects of topology

One significant difference between the two topologies is the number of generations it takes to mix the entire population. The ring topology takes I epochs, and the hypercube topology takes $log(I)$ epochs. We next look at the speed of convergence for the migration policy. The hypercube topology is faster on 12 out of 15 instances. We gain significant speed up (by 1,000 generations) on pla85900 instance. The “migrate policies" do not lose any edge in the population. However, the

Table 5.2: Sequential MGA+EAX (hybrid-1) x=2

Instance	MGA time	EAX time	Generations	Success Rate
u2319	16.61 ± 0	91.58 ± 0	276 ± 0	1/30
pr2392	14.93 ± 5.56	30.23 ± 11.27	207.19 ± 5.11	30/30
pcb3038	22.61 ± 10.31	85.3 ± 33.77	361.94 ± 4.57	30/30
fnl4461	28.71 ± 1.52	246.35 ± 22.07	750.91 ± 30.96	30/30
rl5915	38.23 ± 26.38	106.35 ± 49.48	285.96 ± 11.42	26/30
rl5934	34.43 ± 1.9	104.36 ± 10.72	295.14 ± 13.31	14/30
pla7397	±	±	±	0/30
rl11849	95.18 ± 23.64	651.96 ± 156.35	1045.33 ± 47.11	30/30
usa13509	115.15 ± 6.25	1635.13 ± 140.6	2674.67 ± 86.92	6/30
brd14051	125.88 ± 5.35	2119.29 ± 330.58	3109.12 ± 188.5	30/30
d15112	166.54 ± 38.43	3874.53 ± 996.22	4139.34 ± 391.99	29/30
d18512	184.58 ± 6.74	4119.88 ± 471.61	4989.94 ± 490.82	30/30
pla33810	313.11 ± 10.58	4236.13 ± 588.61	6910.93 ± 871.41	29/30
pla85900	1398.17 ± 377.4	30041.34 ± 3641.31	21730.34 ± 1926.52	30/30

Table 5.3: Sequential MGA+EAX (hybrid-1) x=5

Instance	MGA time	EAX time	Generations	Success Rate
u2319	56.95 ± 22.37	117.28 ± 46.41	305 ± 0	3/30
pr2392	40.39 ± 14.29	25.65 ± 9.47	207.19 ± 4.4	30/30
pcb3038	58.85 ± 25.24	77.23 ± 28.31	366.59 ± 8.53	30/30
fnl4461	74.22 ± 4.6	240.88 ± 22.43	771.73 ± 28.7	26/30
rl5915	82.71 ± 4.49	91.43 ± 10.58	295.86 ± 14.75	30/30
rl5934	84.68 ± 6.07	96.24 ± 12.09	297.41 ± 15.71	17/30
pla7397	±	±	±	0/30
rl11849	228.5 ± 25.63	631.64 ± 92.77	1035.72 ± 75.9	30/30
usa13509	274.05 ± 14.7	1767.55 ± 313.58	2830.22 ± 272.62	9/30
brd14051	305.57 ± 18.74	2207.45 ± 304.89	3271.62 ± 161.03	30/30
d15112	417.96 ± 99.31	3707.66 ± 924.8	4130.17 ± 382.66	23/30
d18512	469.35 ± 100.91	3999 ± 626.08	4765.16 ± 352.46	30/30
pla33810	826.4 ± 422.77	4672.78 ± 1562.38	7308.45 ± 935.93	29/30
pla85900	2673.36 ± 596.78	28736.98 ± 6017.05	21521.84 ± 3078.93	30/30

“migrate policies" cause EAX to require more time to converge for the ring topology; this appears to be because it takes EAX longer to remove unnecessary edges from the population.

On three instances - u2319, pla7397, pla33810, the ring topology converges faster. Additional mixing is helpful for these cases. In Chapter 6, we note the existence of multiple global optima for

Table 5.4: MGA+EAX - Ring topology

Instance	MGA+EAX, migrate-ring				MGA+EAX, replace-ring			
	MGA time	EAX time	Gen	S.R	MGA time	EAX time	Gen	S.R
d2103	3 ± 0	14 ± 3	107 ± 5	30/30	3 ± 0	11 ± 2	96 ± 11	26/30
u2319	4 ± 0	91 ± 27	294 ± 46	4/30	$\pm$	$\pm$	$\pm$	0/30
pr2392	4 ± 0	21 ± 1	210 ± 6	30/30	4 ± 0	16 ± 1	160 ± 12	30/30
pcb3038	5 ± 0	55 ± 4	364 ± 20	30/30	5 ± 0	46 ± 4	293 ± 21	29/30
fnl4461	8 ± 0	214 ± 14	786 ± 46	28/30	8 ± 0	183 ± 11	645 ± 34	19/30
rl5915	10 ± 2	92 ± 20	295 ± 10	29/30	10 ± 0	74 ± 4	240 ± 13	30/30
rl5934	10 ± 3	97 ± 35	298 ± 8	18/30	10 ± 0	71 ± 5	234 ± 14	14/30
pla7397	13 ± 0	185 ± 0	945 ± 0	1/30	14 ± 0	243 ± 0	938 ± 0	1/30
rl11849	25 ± 0	555 ± 50	1078 ± 95	18/30	25 ± 1	515 ± 58	944 ± 85	30/30
usa13509	36 ± 1	1590 ± 128	2788 ± 78	7/30	35 ± 1	1331 ± 109	2233 ± 153	7/30
brd14051	35 ± 3	1900 ± 315	3215 ± 232	30/30	35 ± 2	1567 ± 188	2703 ± 196	30/30
d15112	41 ± 1	2811 ± 344	3996 ± 359	24/30	42 ± 2	2608 ± 482	3623 ± 416	12/30
d18512	48 ± 2	3441 ± 378	5162 ± 447	30/30	48 ± 2	2986 ± 317	4298 ± 326	30/30
pla33810	74 ± 3	3168 ± 362	6880 ± 893	29/30	75 ± 1	2765 ± 250	5844 ± 741	17/30
pla85900	279 ± 53	22289 ± 2396	22606 ± 2442	30/30	274 ± 56	19307 ± 2940	19139 ± 2660	30/30
AVERAGE:				22.53/30				18.33/30

Table 5.5: MGA+EAX - Hypercube topology

Instance	MGA+EAX, migrate-hypercube				MGA+EAX, replace-hypercube			
	MGA time	EAX time	Gen	S.R	MGA time	EAX time	Gen	S.R
d2103	1 ± 0	14 ± 1	86 ± 3	30/30	1 ± 0	12 ± 1	79 ± 5	30/30
u2319	2 ± 0	78 ± 2	300 ± 17	2/30	2 ± 0	62 ± 0	213 ± 0	1/30
pr2392	1 ± 0	24 ± 1	207 ± 5	30/30	1 ± 0	21 ± 1	186 ± 8	30/30
pcb3038	2 ± 0	57 ± 2	355 ± 9	30/30	2 ± 0	55 ± 2	333 ± 11	30/30
fnl4461	3 ± 0	213 ± 14	759 ± 41	27/30	3 ± 0	205 ± 12	718 ± 34	30/30
rl5915	3 ± 0	95 ± 4	290 ± 11	27/30	4 ± 0	89 ± 4	267 ± 12	28/30
rl5934	3 ± 0	95 ± 3	297 ± 9	15/30	4 ± 0	88 ± 4	266 ± 12	20/30
pla7397	5 ± 0	197 ± 0	1029 ± 0	1/30	5 ± 0	231 ± 28	1126 ± 158	2/30
rl11849	10 ± 0	568 ± 40	1080 ± 64	30/30	10 ± 0	538 ± 31	984 ± 41	30/30
usa13509	11 ± 0	1227 ± 44	2545 ± 52	2/30	13 ± 1	1512 ± 169	2682 ± 158	11/30
brd14051	14 ± 4	1827 ± 351	3177 ± 282	30/30	13 ± 1	1714 ± 123	3045 ± 169	30/30
d15112	16 ± 1	2770 ± 280	3972 ± 288	23/30	16 ± 1	2731 ± 376	3833 ± 352	27/30
d18512	18 ± 1	3338 ± 365	4903 ± 446	30/30	18 ± 1	3209 ± 281	4771 ± 440	30/30
pla33810	28 ± 1	3207 ± 337	7104 ± 957	28/30	28 ± 2	3036 ± 365	6568 ± 858	26/30
pla85900	109 ± 22	22905 ± 2821	21650 ± 2452	30/30	120 ± 30	23430 ± 4187	22384 ± 3141	30/30
AVERAGE:				22.33/30				23.67/30

u2319, which appears to result in a more difficult search space. For pla33810, there is no known optimal solution. For pla7397, it is unknown if there are multiple global optima.

The migrate-ring topology is better than migrate-hypercube topology on 7 of the 15 instances in terms of success rate. Both topologies achieve 100% success on 6 of 15 instance, and solve pla7397 on one run. However, for one instance - rl11849, the hypercube topology is better. When examined closely, this particular instance gets stuck at one specific local optima and cannot escape.

We next look at the replacement policies. The replace-ring topology often displays fast convergence, particularly on larger problems. However, this topology has the lowest overall success rate. This suggests that replacing poor individuals can limit exploration.

To summarize, the migrate-ring and replace-hypercube topology resulted in the highest success rates, but the migrate-hypercube and replace-ring topology often converged faster.

Effects of the mixing policy

The migration policies help to retain all the edges in the population. The replacement policy, however, loses some edges in the population. There is a second factor that impacts these results. MGA runs longer in the ring configuration. This is because it take I epochs for the best individual to migrate across all of the islands. Using the hypercube configuration, it takes $log(I)$ epochs for the best individual to migrate across the islands. Thus one "batch" takes longer in the MGA ring configuration than the MGA Hypercube configuration. In the "migrate configuration" the ring has longer to make progress. But in the "replace configuration" the ring also has more opportunities to remove edges from the population. This appears to explain why the replace-ring topology is sometimes very fast, but it has a lower overall success rate. The difference in the generations increases with problem size. On pla85900, we get a speedup of 3467 generations. These observations do suggest that sometimes more aggressive selection (by removing edges) can improve the convergence speed. The replace-hypercube and migrate-hypercube topologies are somewhat similar in terms of runtime and success rate. The replace-hypercube topology has a better success rate, except in two cases - u2319 and pla33810. As noted in the previously section, this is perhaps because these instances contain multiple optimal solutions.

To summarize, migration helps to preserve diversity, but may slow down convergence speed. Replacement can speed up fast convergence, but critical edges can potentially be lost. Because MGA runs for fewer generations under the hypercube topology, there are less opportunities to lose edges. Thus, different algorithm configurations display difference performance on different instances. When properly leveraged, this can be an advantage.

5.4 A Parallel Ensemble of Solvers

The hybrid solvers performs differently on different problem instances. This is no suprise because, in chapter 4, we saw similar results. The MGA and EAX did not find the solution for u2319 instance. However, the hybrid found the solution with 73% success rate. Thus, to find one single best solver for all TSP instance is an open challenge.

A similar study by Kerschke et al. [16] shows that different instances may be solved effectively by different solvers. They build an algorithm selector based on three supervised learning techniques. Their results on a wide variety of TSP instances showed that **EAX with an added restart mechanism is the single best solver**. They also suggest that a hybrid solver using a fast solver like LKH in the front and switching to EAX might hold significant promise. Their study is, however, limited to problems that are 2,000 city and smaller.

The two studies suggest that we can improve the EAX solver by combining it with LKH or MGA. Instead of finding the best solver, we can use the different hybrid configurations as a **parallel ensemble**. With the rapid growth in parallel architectures, we now have an abundance of computing resources. We would argue that running an ensemble of solvers in parallel is an alternative to using machine learning to select solvers based on the instance type, particularly when we are selecting from a small number of solvers (e.g. less than 8).

5.4.1 Ensemble setup

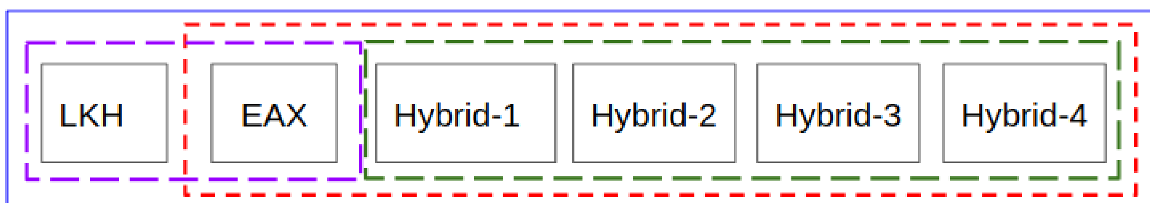


Figure 5.11: Figure showing the nested ensemble layers. The inner layer consists of three ensemble (a) all four Hybrid MGA (b) The four hybrid MGA+EAX algorithms. (c) LKH and EAX. The outer layer consists of all six inexact solvers, also called the Full ensemble.

Figure 5.11 shows the ensemble setup consisting of six inexact solvers. Each algorithm is run in parallel. In this study, when run as an ensemble all solvers are terminated when one finds the known optimal solution or reaches the HK bound. The EAX and LKH solvers are sequential. The hybrid versions consists of parallel MGA and sequential EAX.

Table 5.6: Levels of parallelism

Level	Degree	Comments
1	6	Number of solvers
2	8	Number of islands(CPUs)/solver
3	12	Number of cores/island

Table 5.6 shows the degree of parallelism of the ensemble setup. The degree of parallelism of each hybrid solver is 12. The degree of parallelism of the hybrid ensemble is $12 * 4 = 48$. The degree of parallelism of the GA ensemble is 49. The degree of parallelism of the entire ensemble is 50. The experiments are run on the RMACC Summit supercomputer[4].

The EAX code is from Nagata and Kobayashi [15] and is publicly available [5] The LKH version 3 is also publicly available. [6]. The MGA and EAX use a population of 256. The number of EAX children is 30. The number of islands for parallel MGA is 8. For LKH, we use POPMUSIC to generate the candidate set. We use 5-opt moves. The patching A and C parameters are set to 2 and 3, respectively. Other default values are not altered. The algorithms terminate when it finds the known optimal solution or the best known solution. Otherwise, the algorithms terminate using their default settings. For EAX, the default termination condition is stagnation after a pre-determined period. For MGA, it is 10,000 generations. LKH terminates when there are no further improving moves. Each instance is run for 30 trials, the average time is reported in table 5.8. For the ensemble setup, the solver which terminates early also terminates the other algorithms.

[4]https://www.colorado.edu/rc/resources/summit/specifications

[5]https://github.com/sugia/GA-for-TSP

[6]http://webhotel4.ruc.dk/ keld/research/LKH-3/

5.4.2 Comparisons with EAX

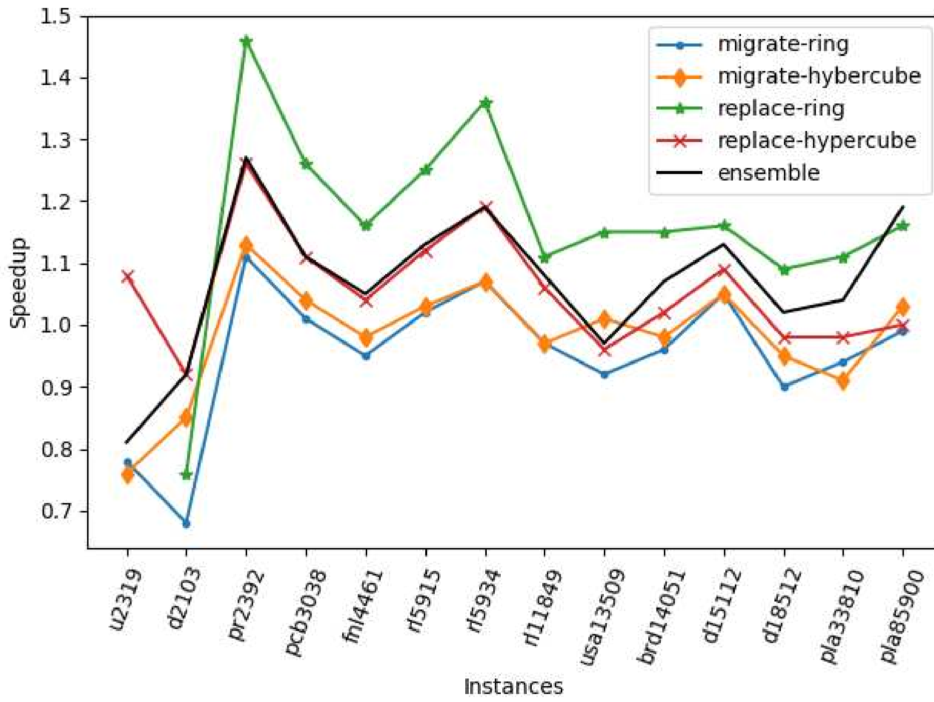
If one only consider "total successes" EAX is slightly better than all of the MGA+EAX hybrids, but only slightly so. However, EAX completely fails on instance pla7397; we ran EAX 120 times, and it failed every time on this instance. This is an unfortunate, and thus one might argue that the replace-hypercube MGA+EAX is preferred since it solves every instance at least once.

All of the MGA+EAX hybrid algorithms converge faster than EAX on all instances. Figure 5.12 shows the hybrid versions' speedup for the number of generations and execution time. In terms of time, we usually get a speedup of $1.1\times$ to $2.5\times$. This speed-up of the MGA-EAX hybrids is not only due to parallelization. We separate the runtime of MGA and EAX for a reason. The EAX part of the hybrid algorithms is run under exact the same compute configuration as EAX. Thus the speed-up of the MGA+EAX hybrids is due to MGA accumulating the best edges in the best individuals. EAX is then able to exploit this concentration of good edges to speed up its own convergence. Thus, standard EAX runs for 8.1 hours on pla85900, while the replace-ring MGA+EAX hybrid runs for 5.3 hours on average on pla85900. Both are 100% successful.

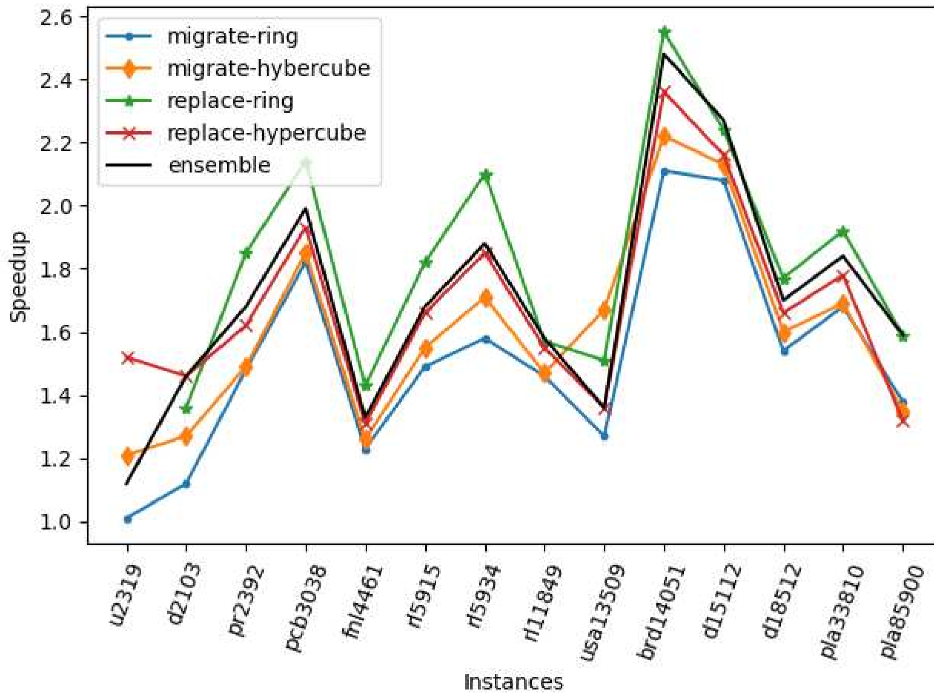
5.4.3 Comparisons with LKH

We explore three levels of parallelism - (a) within a single multi-core, (b) across multiple machines, and (c) the ensemble setup. The first two levels of parallelisms only parallelize the initial 2opt and the MGA generations. Nevertheless, this yields a $2.5\times$ speedup over EAX. This speedup is mostly consistent with problem sizes. Figure 5.13 (plot to the left) shows a wide range of speedup differences over the LKH sequential code. On 9 of the 15 instances, the hybrid MGA+EAX ensemble setup is faster compared to the LKH. LKH finds the solution for four of these nine instances but fails on the remaining five instances.

The LKH does not find the solution on d2103, a drilling TSP problem. In Chapter 4, we show that d2103 has multiple global optima. Although multiple global optima for d2103, do not affect the GA algorithms, it is hard for LKH to find a solution. This particular instance is hard for Concorde as well. The ensemble obtained global optimal solution with an execution time $17.5\times$

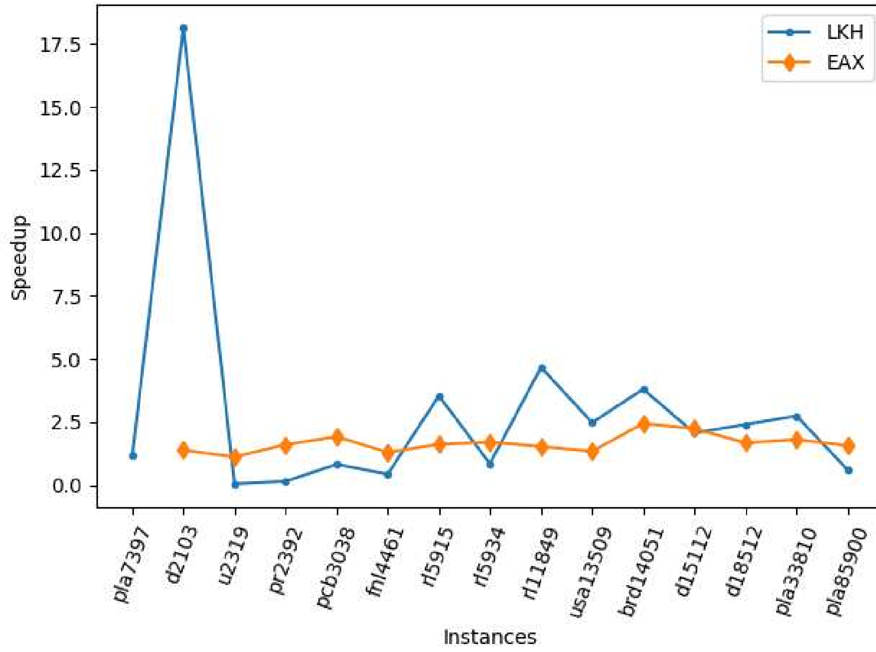


(a)

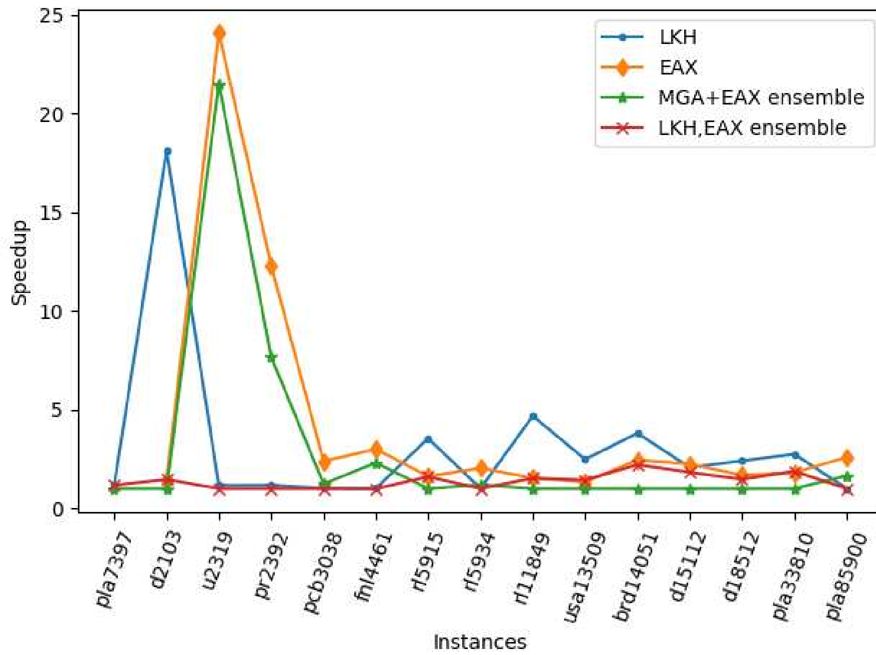


(b)

Figure 5.12: Speedup of the hybrid versions over EAX with respect to (a) generations (b) time



(a)



(b)

Figure 5.13: The speedup of (a) Hybrid MGA+EAX ensemble over sequential LKH and EAX. The hybrid MGA+EAX is parallelized to run within a single-multicore and across multiple CPUs. (b) Full ensemble over the two sequential algorithms and two ensemble setups.

faster then LKH. Thus, the speedup of the hybrid ensemble over LKH is not proportional to the problem size but depends on the instance.

Figure 5.13 (plot to the right) shows the speedup of the *Full Ensemble* over the two sequential algorithms and two smaller ensembles. The Full Ensemble improve convergence speed when compared to individual solvers. On 9 of the 15 instances, the *Full Ensemble* speedup is enhanced by the performance of the hybrid MGA+EAX ensemble. The MGA algorithm improves the runtime performance of EAX by providing EAX much better starting solutions in the population. It also does this without lose of any edges. On the remaining 6 of the 15 instances, the *Full Ensemble* speedup is due to the LKH solver. Here, we can see a trend in the timings. The GA solvers are fast on instances greater than 5,000 city problems. The LKH solver is fast for some smaller problem sizes that are nevertheless difficult for the GA solvers. We have already noted that d2103 is difficult for LKH. The pla85900 is a programmable logic array TSP instance. The POPMUSIC heuristic used by LKH divides the initial tour into segments and optimizes each segment using the 3opt local search. The structure of the pla85900 has clusters in it.[7] Creating the candidate edges using POPMUSIC has given LKH an advantage, because of the structure of the problem.

The main take-home message is that the ensemble exploits the fact that some "hard" problems for one algorithm may be easy for another algorithm. The different solvers complement one another. Table 5.8 compares the average time required to solve all the instances. The EAX is slower, followed by LKH. The hybrid MGA+EAX ensemble is faster than the ensemble using just LKH and EAX. Thus, the *Full Ensemble* needs more than just LKH and EAX to achieve the best results.

5.4.4 GA Ensemble

Table 5.7 illustrates the advantages to using an ensemble of solvers in parallel. There are three sets of results given in this table. One set of results are for the "single run" sequential EAX. As previously noted, instead of adding restarts to EAX, which results in longer run time, we created an ensemble of 4 EAX solvers (which in effect may be better than having 4 restarts). The EAX

[7] http://www.math.uwaterloo.ca/tsp/pla85900/index.html

Table 5.7: EAX versus MGA+EAX Ensemble

Instance	EAX		Sequential	Ensemble	MGA+EAX		Ensemble
	Time	Gen.	S.R.	S.R.	Time	Gen.	S.R.
d2103	16 ± 7	73 ± 3	30/30	30/30	13 ± 4	79 ± 0	30/30
u2319	92 ± 1	229 ± 0	3/30	3/30	86 ± 45	283 ± 23	7/30
pr2392	32 ± 5	234 ± 4	30/30	30/30	22 ± 7	185 ± 1	30/30
pcb3038	99 ± 34	368 ± 6	30/30	30/30	55 ± 11	331 ± 2	30/30
fnl4461	256 ± 63	747 ± 37	30/30	30/30	206 ± 33	714 ± 13	30/30
rl5915	135 ± 22	300 ± 3	30/30	30/30	91 ± 15	266 ± 6	30/30
rl5934	151 ± 3	317 ± 6	11/30	30/30	90 ± 22	266 ± 9	30/30
pla7397	$\pm$	$\pm$	0/30	0/30	235 ± 157	1125 ± 28	5/30
rl11849	752 ± 36	1046 ± 28	30/30	30/30	538 ± 40	968 ± 35	30/30
usa13509	1943 ± 44	2579 ± 21	17/30.	29/30	1523 ± 196	2654 ± 198	17/30
brd14051	3878 ± 1780	3102 ± 284	30/30	30/30	1647 ± 200	2900 ± 149	30/30
d15112	5633 ± 2235	4189 ± 197	30/30	30/30	2612 ± 303	3702 ± 297	30/30
d18512	5128 ± 1510	4668 ± 332	30/30	30/30	3160 ± 297	4576 ± 293	30/30
pla33810	4972 ± 598	6466 ± 495	30/30	30/30	2962 ± 629	6213 ± 282	30/30
pla85900	29294 ± 5575	22281 ± 2204	30/30	30/30	19531 ± 2005	18745 ± 2530	30/30
AVERAGE:	3741.5		24.06/30	26.13/30	2184.7		25.93/30

ensemble improves it performance on usa13509 and rl5934. But the ensemble of EAX ran with 4 different seeds does not improve on u2319. (Indeed our first "Sequential EAX" seems have been somewhat lucky at having such a high success rate compared to other members of the EAX ensemble, which had average success rates closer to 23/30.) In table 5.7, the reported runtime is for the Sequential EAX, but the results are almost identical when all four versions of EAX are run in parallel, since there is no communication costs. EAX failed to solve pla7397 on all 120 runs of the ensemble. This is one potential danger of using an ensemble composed of just one solver.

Table 5.7 presents the result of using the four topologies of the MGA+EAX hybrids as an ensemble of 4 solvers (with a combined population size of 1024). The average success rate increases to 25.93 out of 30. But the runtime also improves relative to EAX and the EAX ensemble. The average runtime decreased by more than 58% from 62.35 minutes to 36.41 minutes when using the MGA+EAX hybrids. The MGA+EAX ensemble solved u2319 on 7 of 30 runs compared to 3 of 30 for EAX and the EAX ensemble. And the MGA+EAX ensemble solved pla7397 on 5 of 30 runs. As noted, EAX and the EAX ensemble never solve this problem given 120 attempts.

5.4.5 Full Ensemble

We consider six algorithms for the *Full Ensemble* Table 5.8 shows the results on the three ensembles (a) LKH and EAX (b) Hybrid MGA+EAX algorithms (d) the *Full Ensemble*. The Hybrid MGA+EAX ensemble outperforms LKH on 5 of 15 instances. LKH outperforms the Hybrid MGA+EAX ensemble on 2 of 15 instances. On the remaining 8 of 15 instances, both the solvers achieve 100% success. This observation is also true if we compare LKH and EAX. EAX outperforms LKH on the same five instances. However, LKH outperforms EAX on 3 of 15 instance. The Hybrid MGA+EAX ensemble outperforms EAX on the rl5934 instance.

The u2319 is a difficult instance identified by Varadarajan and Whitley [26], and Dubois et al. [117]. There are two global optima, and the distance between the two global optima makes it difficult for the GA solvers to converge. In contrast, the LKH can solve it with ease. This is because

Table 5.8: Results on ensemble setup. S.R refers to Success Rate. Time is in seconds. The average success rates at the last row is out of 30 trials.

Instance	Sequential				Ensemble					
	LKH		EAX		LKH+EAX		MGA+EAX		Full Ensemble	
	S.R	Time	S.R	Time	S.R	Time	S.R	Time	S.R	Time
d2103	0/30	236	30/30	18	30/30	19	30/30	13	30/30	13
u2319	30/30	5	3/30	96	30/30	4	7/30	86	30/30	4
pr2392	30/30	4	30/30	37	30/30	3	30/30	23	30/30	3
pcb3038	30/30	47	30/30	110	30/30	46	30/30	57	30/30	46
fnl4461	30/30	92	30/30	273	30/30	90	30/30	210	30/30	91
rl5915	0/30	337	30/30	154	30/30	152	30/30	95	30/30	95
rl5934	30/30	80	11/30	161	30/30	79	30/30	94	30/30	79
pla7397	30/30	283	0/30	NA	30/30	283	5/30	241	30/30	241
rl11849	0/30	2600	30/30	853	30/30	849	30/30	556	30/30	556
usa13509	0/30	3844	17/30	2078	17/30	2223	17/30	1549	23/30	1549
brd14051	0/30	6367	30/30	4093	30/30	3693	30/30	1673	30/30	1673
d15112	30/30	5477	30/30	5936	30/30	4822	30/30	2646	30/30	2646
d18512	30/30	7705	30/30	5383	30/30	4716	30/30	3204	30/30	3204
pla33810	30/30	8334	30/30	5484	30/30	5695	30/30	3034	30/30	3034
pla85900	30/30	12129	30/30	31318	30/30	12099	30/30	19969	30/30	12099
AVERAGE	20	3169	24.06	3999	29.13	2318	25.93	2230	29.53	1688

the k-opt moves direct the search towards only one of the global optima. For instance pla7397 LKH solve this problem every time, but it is a difficult for the population based GA solvers.

To summarize, table 5.8 shows the average success rates on all the solvers. On average, LKH is the least successful, followed by EAX. This observation supports the observation made by Pascal et al. [16]. They claim that EAX is the best single solver on problems lesser than 2,000 cities. We show that EAX is better than LKH on larger problems as well. Looking at the ensemble setup, the Hybrid MGA+EAX solvers are better than EAX alone. This observation supports our previous studies [26], [27]. Having a hybrid of MGA and EAX is better than the single EAX solver. The ensemble of LKH and EAX has a score of 29.13. The *Full Ensemble* has a score slightly greater of 29.53. These observations suggest that LKH and EAX are the best solvers. But, all the hybrid MGA+EAX has an advantage on certain hard problems.

5.5 Conclusions

This chapter makes two fundamental contributions. 1) It constructs a family of hybrid MGA and EAX algorithms that run faster than a standard configuration of EAX; the hybrid algorithms also competitive in generating globally optimal solutions. 2) It introduces the use of an ensemble of solvers ran in parallel to improve the successful convergence.

The MGA using the GPX operator can effectively mix the population and reach the global optima in relatively few generations using large population sizes. However, the time it takes to process a large population is expense, even if the number of generations needed to converge decreases. We introduce four parallel hybrid versions of MGA and EAX. The ring and hypercube topology effectively bring out the mixing properties of the MGA, which are not guaranteed with any other standard GA. Under a migration policy, MGA concentrates the best edges in the best individuals without losing any edges from the populations. A replacement policy was also introduced that removes the worst individuals from the island populations after migration. This can speed up convergence, but it also loses edges.

Some of the future research directions are to improve the rate of convergence. It should be possible to better balance the mixing properties and migration policies and replacement polices. A second research direction is to use the existing topologies on other massively parallel architectures such as the GPU. Since we can efficiently mix the population for a smaller island size, the parallel MGA will form an ideal candidate for architectures having a small memory footprint. The third research direction is to explore new way to control population diversity. There may be more intelligent ways to decide when to allow MGA to remove edges from the population, perhaps by considering a metric such as the edge frequencies used by EAX. Finally, we also note that other researchers have recently developed a new form of GPX [118] that finds even more recombining components and also finds them much faster, but we have not implemented or tested this new form of GPX. Improvements in finding more recombination components faster will also improve our the ability of genetic algorithms to yield excellent results on large instances of the TSP.

Chapter 6

Preliminary findings on operator-level parallelization

The MGA designed so far is superior to EAX in five key aspects. First, the memory consumed per crossover operation is $4\times$ lesser compared to EAX. Second, there is no communication to the cost matrix during the crossover operation. Third, it reaches the global optima solution with fewer recombinations. Fourth, the convergence rate on higher population size is fast, the desired property when using parallel architectures. Fifth, the population-level parallelization gives significant speed-up without the loss of performance. Despite these changes, the execution time of one crossover operation is high. The main reason is, the GPX operator is not optimized for speed. In this chapter, we revisit the GPX operator and look for operator-level parallelization opportunities.

The GPX crossover operation is divided into two major phases, namely, partition and recombination. The partition phase is where the union graph is split into recombining components. In figure 6.1, the GPX operator finds the union of the two-parent tours. The solid red and the blue dashed lines represent the two-parent tours. The GPX operator, then, splits vertices of degree four. In figure 6.1, vertex 1 and 6 are degree 4 vertices. These vertices are split by introducing an imaginary common edge (1-1' and 6-6') with zero weight. Finally, the GPX operator finds the candidate AB cycles by deleting the common edges. These steps complete the partition phase.

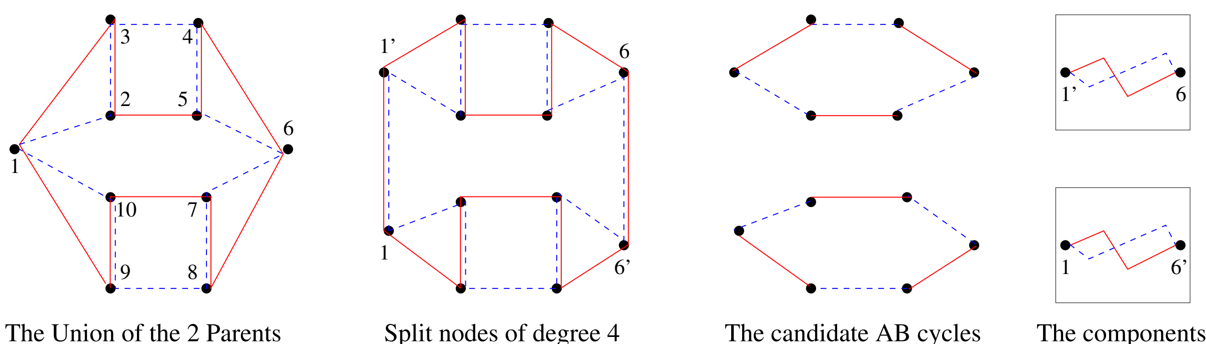


Figure 6.1: Partition Phase of GPX operator finds the union of the two parent tours, splits the degree 4 vertices, deletes the common edges to form the two candidate AB cycles. The path of the two tours inside each component can be replaced by a single red and blue edge.

The recombination phase first, tests the feasibility of candidate components. Chapter 2, section 2.6.3 gives examples on the feasibility of recombination. This phase also includes the fusing of multiple candidate components to find more recombination opportunities. Finally, each recombining component's best and worst partitions are chosen to form the best and worst child, respectively.

The time profiling of the two phases is measured for instances ranging in size from 11,849 upto 2-million cities. Note that these numbers are measured on two random input tours. The time reported is an average of 30 random recombinations. From 6.2, it can be seen that for problems higher than 160K, more than 50% of the time is spent in the partition phase. In this chapter, we focus only on the parallelization of the partition phase.

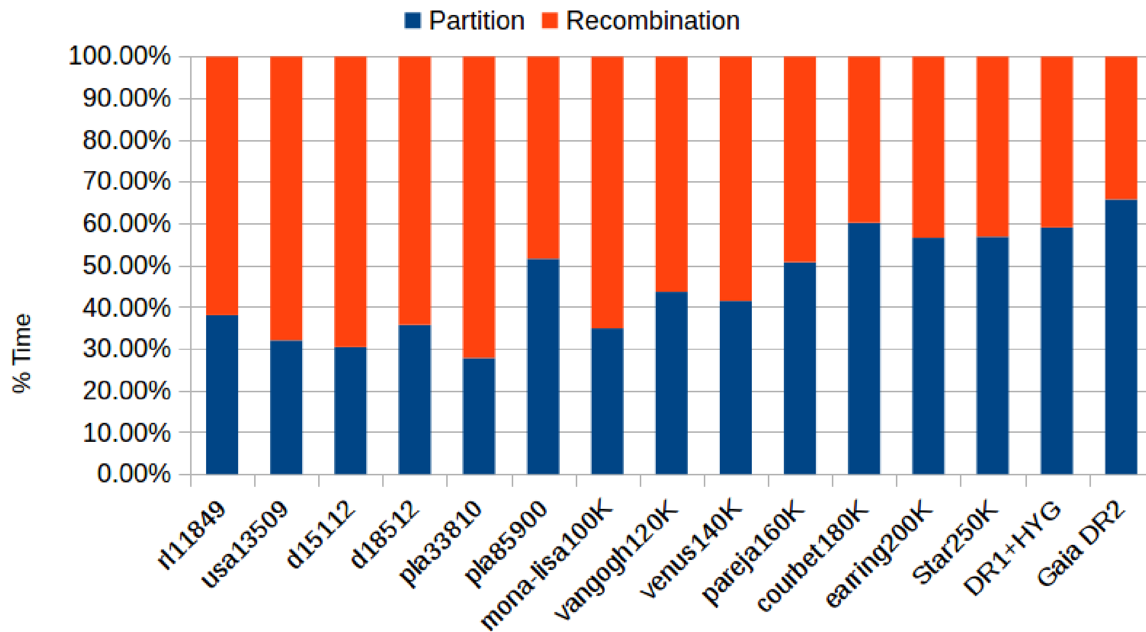


Figure 6.2: Figure showing the time profiling of the GPX operator. The DR1+HYG and GaiaDR2 instances are 1M and 2M city problems, respectively. For problems smaller than 160,000 recombination phases is dominating. For problems higher than 160,000, the parititon phase is dominating.

The chapter is organized as follows. In Section 6.1, we provide a brief background. Section 6.2 provides the parallel implementation of the partition phase. Section 6.3 presents the experimental results. Finally, section 6.5 concludes the chapter with discussions on future work.

6.1 Background

The operator-level parallelization makes sense when using massively parallel architectures (example: GPUs). Typically, GPUs use 1000s and 10,000 threads and are highly efficient on larger data sets. Whereas, for the CPU parallelism, there will be a limit on the number of parallel units. Typically, CPUs have 10s or 100s of parallel units. To the best of our knowledge, the GPU-based TSP solver by Kang et al. [20] contains some form of operator-level implementation. Operations on the tour such as calculating the tour length, revesing the tour, copying the tour are performed in parallel. However, the entire crossover operation is performed sequentially.

Population-level parallelization on the GPUS can limit the solver's scalability because of the memory required per crossover. Fujimoto and Tsutsui [18] use Ordered Crossover operator and could scale only up to 500 cities. Zhang et al. [19] use the Partially Mapped Crossover (PMX) operator and could scale up to 2392 cities. We observe similar scalability concerns for MGA.

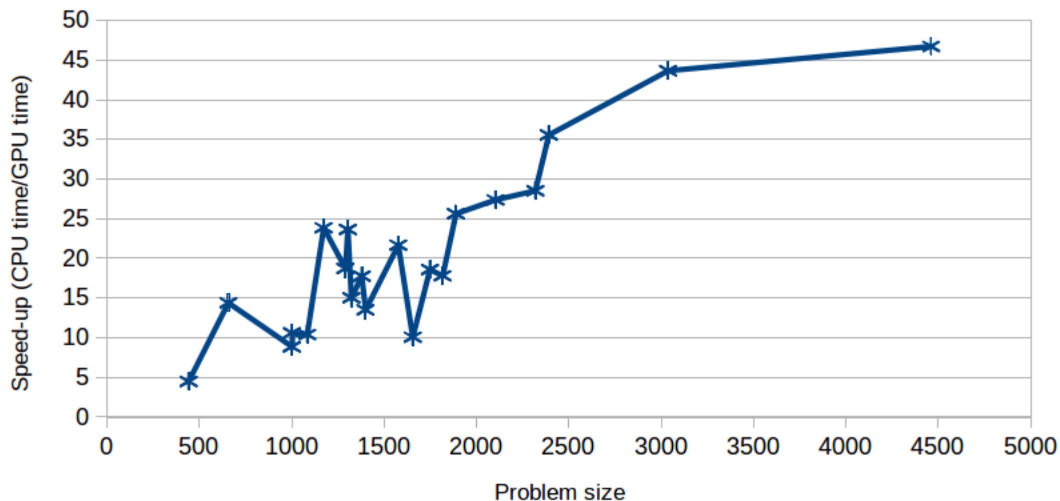


Figure 6.3: Plot showing the speedup of GPX crossover on the GPU with respect to the CPU code.

The population-level implementation of the MGA algorithm on the GPUs could scale only up to the 4000-cities. Beyond that, the hardware memory becomes the limitation. Figure 6.3 shows a simple speed-up of the crossover operation over the sequential CPU implementation. One thread was responsible for one crossover operation. The population-level GPU results are not encour-

aging from scalability perspective. However, the GPU implementation of Kang et al. [20] using the Alternating Recommendation Crossover (ARX) operator could scale up to 100,000 city problems. Note that the crossover operation is sequential. However, some operations are performed in parallel. Therefore, the degree of parallelization is proportional to the population size.

Wang et al. [107] explore another form of operator-level parallelism on the CPUs. The TSP tours are divided into segments. Local search and crossover operations are performed on each segment. Finally, the segments are recombined to form the original tour. They report that segmenting the tour and migrating the best sub-tours between the segments provide better results.

The LKH solver [43] uses tour partitioning techniques on large TSP instances. It uses six different techniques, namely, Tour segment, Karp, Delaunay, Rohe, K-means, and Space-filling Curve partitioning. The tour segment partitioning is a general method, applied to all types of TSP. It divides the tour into equal segments, performs local improvements and puts them back together. The remaining five methods require the problem to be geometric. The Karp and Rohe methods partitions the graph into rectangles. The Delaunay and K-means methods partitions the graph into clusters. The tour merging techniques merge these partitions into a single tour.

Partial OPtimization Metaheuristic Under Special Intensification Conditions [49] (POPMUSIC) is a framework developed specifically to tackle larger problem sizes for several optimization problems. The goal of POPMUSIC is to partially optimize these problems and merge them into the original solution. For TSP, POPMUSIC divides the tour into sub-tours and applies the 3-opt [42] local search operator. The critical point is the low empirical time complexity ($\theta(N^{1.6})$) and high quality solutions for large instances. It is tested on problems as high as 10M cities. It is integrated with the LKH [43] as a pre-processing step to produce a good set of candidate edges.

Tour segmentation and tour merging techniques although, efficient, do not fully utilize the benefits of massively parallel architectures. In this chapter, we demonstrate our preliminary findings on the operator-level parallelization of the partition phase of the GPX operator. To the best of our knowledge, there are no other fine-grain implementation of genetic algorithm operators for TSP.

6.2 Parallelization of the partition phase

The partition phase of the GPX code consists of three significant steps, namely, (1) Finding the union of two-parent tours, (2) Splitting nodes of degree 4 and, (3) Identifying the recombining components. In this section, we describe the parallel implementation of each step.

We provide the implementation details for GPUs and compare how they would differ with the shared memory CPU and distributed memory CPU architectures.

6.2.1 Union of two-parent tours

The input to the GPX operator is two vectors, representing the tour permuatation. The GPX operator uses the "edge-table" representation to find the union of these two tours. Graphically, when two tours are combined, each vertex will have at least two and at most four neighbours. The edge table captures this concept by using a two-dimensional matrix with N rows and 4 columns. Each row represent a vertex and its neighbours. Thus, the edge-table is a form of adjacency list. However, the number of columns are fixed to four because we are dealing with just two graphs. Table 6.1 demonstrates this concept on a 10-city problem built based on the tours in figure 6.1.

Table 6.1: Example edge table of a 10-city problem

Vertex	Neighbour-1 (prev-red)	Neighbour-2 (next-red)	Neighbour-3 (prev-blue)	Neighbour-4 (next-blue)
1	9	3	10	2
2	3	5	1	3
3	1	2	2	4
4	5	6	3	5
5	2	4	4	6
6	4	8	5	7
7	8	10	8	9
8	6	7	7	9
9	10	1	8	10
10	7	9	9	1

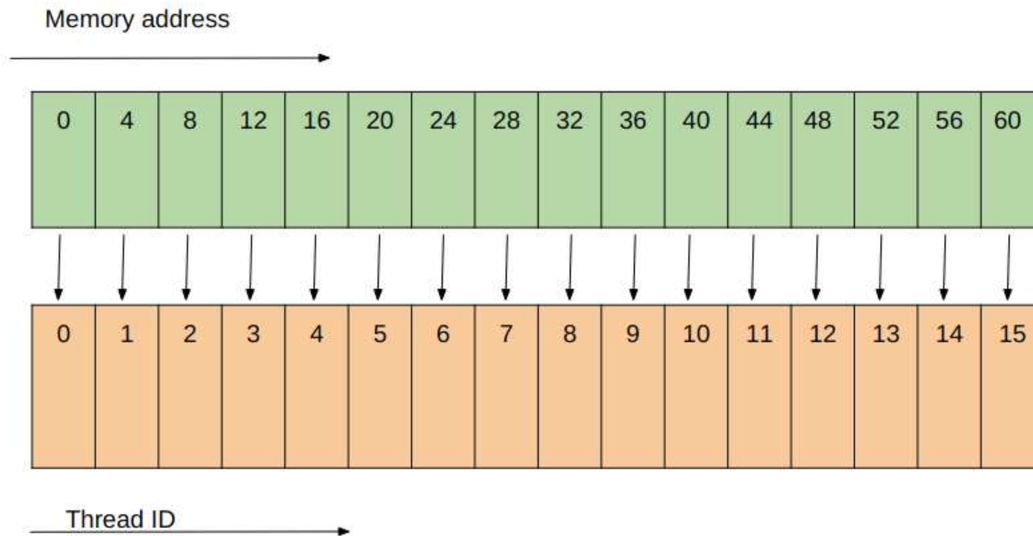


Figure 6.4: Figure showing the thread and memory mapping for a 16-city problem. There are 16 threads corresponding to each city. Each thread access four memory consecutive memory locations. Thus, a total of 64 memory locations are accessed. Threads are coalesced since memory access is sequential and aligned.

Let us consider the example from figure 6.1. The tour permuation of the parents are **1-3-2-5-4-6-8-7-10-9** and **1-2-3-4-5-6-7-8-9-10**. The corresponding "edge-table" is shown in table 6.1. Note that the order in which the neighbours are entered into the edge-table is important. The first neighbour is the previous city in the red tour. The second neighbour is the next city in the red tour. Similarly, the third and fourth neighbours are the previous and next cities in the blue tour.

The GPU implementation creates the edge table in parallel. Each thread operates on one city and fills the edge table. Thus, the degree of parallelization is equal to the number of cities (N). However, the edge-table on the GPU is a flattened array of size $N * 4$. The flattening is done such that the memory access are serialized and coalesced. Coalescing means combining multiple memory accesses into a single transaction. For efficient GPU implementation, the memory access must be coaslesced. Figure 6.4 demonstrates how coalescing is achieved.

The shared memory CPU implementation uses the same edge-table datastructure. However, the number of threads on the shared memory architecture is limited. One thread will be responsible for a group of cities. For example, the GPU architecture can have as many as 2048 threads running

in parallel. However, the shared memory architecture in this thesis has only 24 threads running in parallel. Therefore, ideally, the GPU implementation must show about $100\times$ speedup.

The distributed memory architecture uses the same edge-table datastructure. However, here, there is a need to broadcast the tour to the parallel units. Ideally, a master-worker parallel model will be suitable here. The master node can manage the main GPX operation and the each slave node can work on parts of the edge-table. The number of slave nodes decides the degree of parallelism. In this thesis, we use the RMACC Summit Supercomputer [1] . The maximum possible number of CPUs that can be run in parallel is 900. However, the GPU can have thousands and ten thousands of threads. Therefore, the GPUs are ideal for operator-level parallelization. Note that the supercomputer has multiple GPUs. Therefore, mulitple GPUs can also be cascaded in a distributed memory fashion, act as worker nodes, thereby increasing the degrees of parallelization.

6.2.2 Splitting nodes of degree four

The next step is to split nodes of degree 4. The original GPX code identifies vertices of degree 4 and splits them by introducing imaginary nodes called the ghost nodes. For our example, vertex 1 and 6 are degree 4 vertices. The two new tour will be, **1-1'(11)-3-2-5-4-6-6'(12)-8-7-10-9** and **1-1'(11)-2-3-4-5-6-6'(12)-7-8-9-10**. Note that the 1' and 6' are numbered 11 and 12 respectively. These vertices are called as "ghost" vertices since they do not exist in the original tour. The ghost vertices, in the original GPX code, are numbered starting from N. In the worst case, if all the vertices are of degree 4, the new tour will be of length $2N$.

Let v be the current vertex. Let A and B be the previous and next vertices of red tour. Let C and D be the previous and next vertices of blue tour. As seen in figure 6.5, the blue tour can be aligned in forward or reverse direction. The reason to align the tours in different direction is to maximize the number of partitions. For example, consider the second city (vertex number 2) in our example. The left and right neighbours of red tour are 3(=A) and 5(=B) respectively. The left and right neighbours of forward blue tour are 1(=C) and 3(=D) respectively. Now, even though,

[1]https://www.colorado.edu/rc/resources/summit/specifications

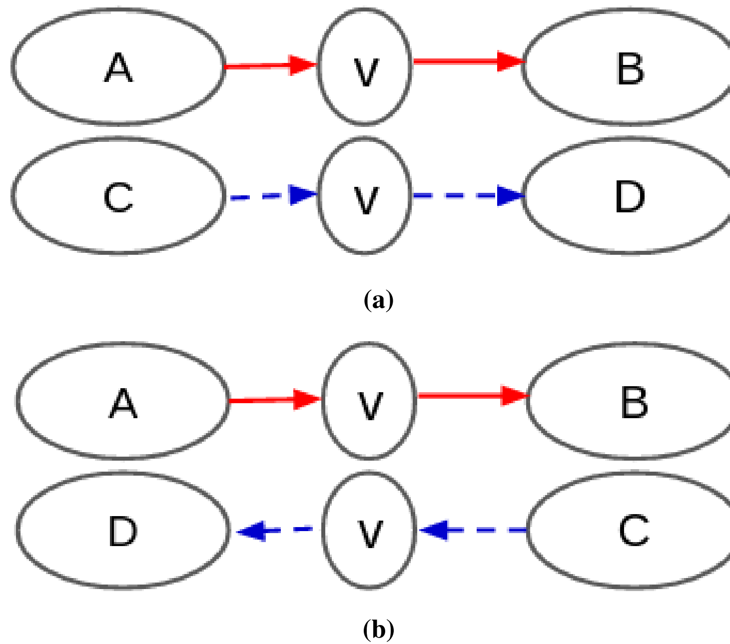


Figure 6.5: Figure showing a common vertex (v) of the two parents aligned in (a)forward (b) reverse directions. Vertices A and B are left and right neighbors of the vertex v in the red parent. Vertices C and D are the left and right vertex v in the blue parent.

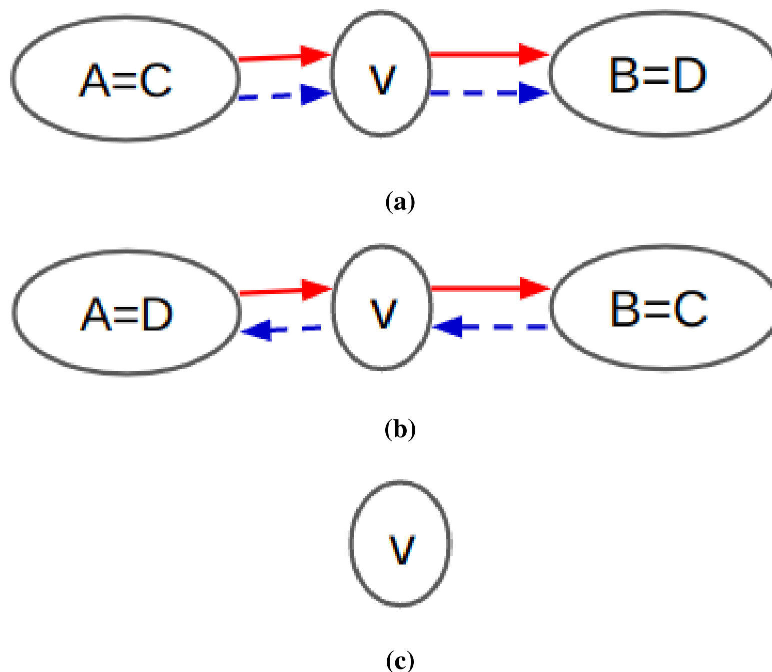


Figure 6.6: Figure showing parents aligned in (a)forward (b) reverse directions. The common vertex v is of degree 2. After removing the common edges, the resultant is (c) a single vertex with no edges.

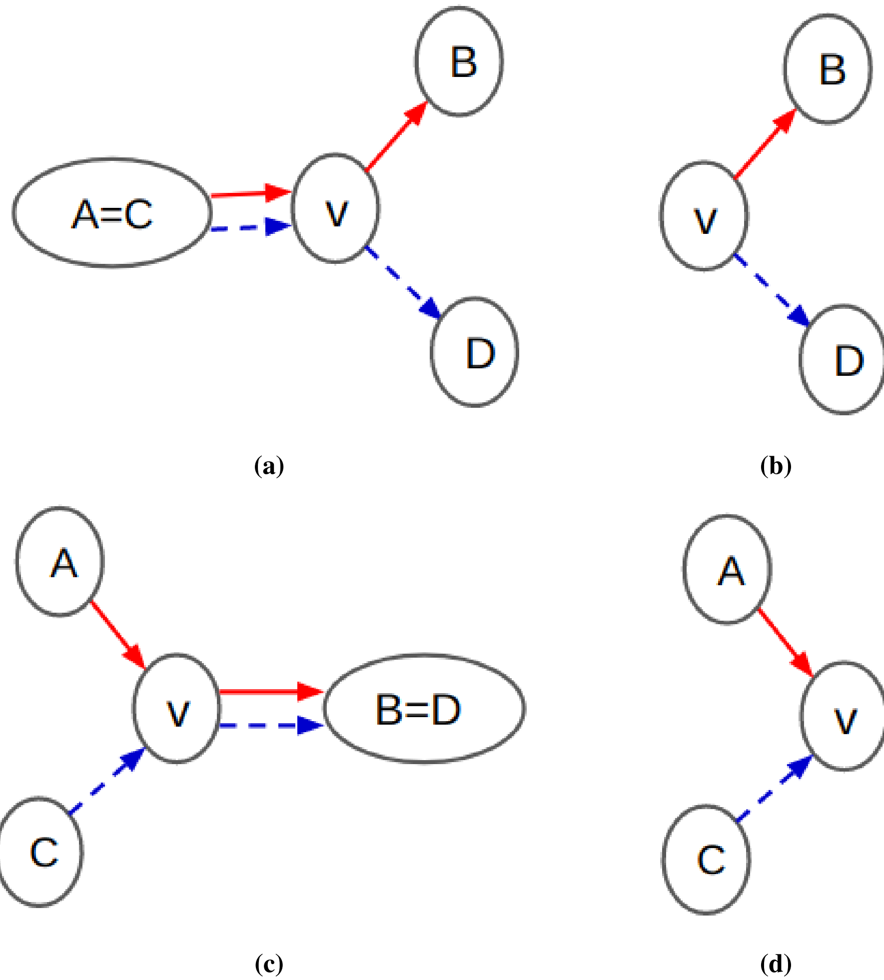


Figure 6.7: In this figure, the common vertex v has degree 3. The parents are aligned in forward direction. Figures (a) and (c) show the two possible cases of vertices before partition. Figures (b) and (d) show the vertices after deleting the common edges.

vetex 2 has a common edge with city 3, due to the forward alignemnt, this edge will not be deleted. However, when we align the blue tour in reverse direction, the the common edge (2-3) is deleted. Therefore, direction of the tours, help in maximizing the number of partitions. In the following paragrahps, we will discuss these rules in detail.

From figure 6.6, it can be seen that, irrespective of the directions, the degree 2 vertices, after deleting the common edges do not take part in recombination. They are inherited by both the children.

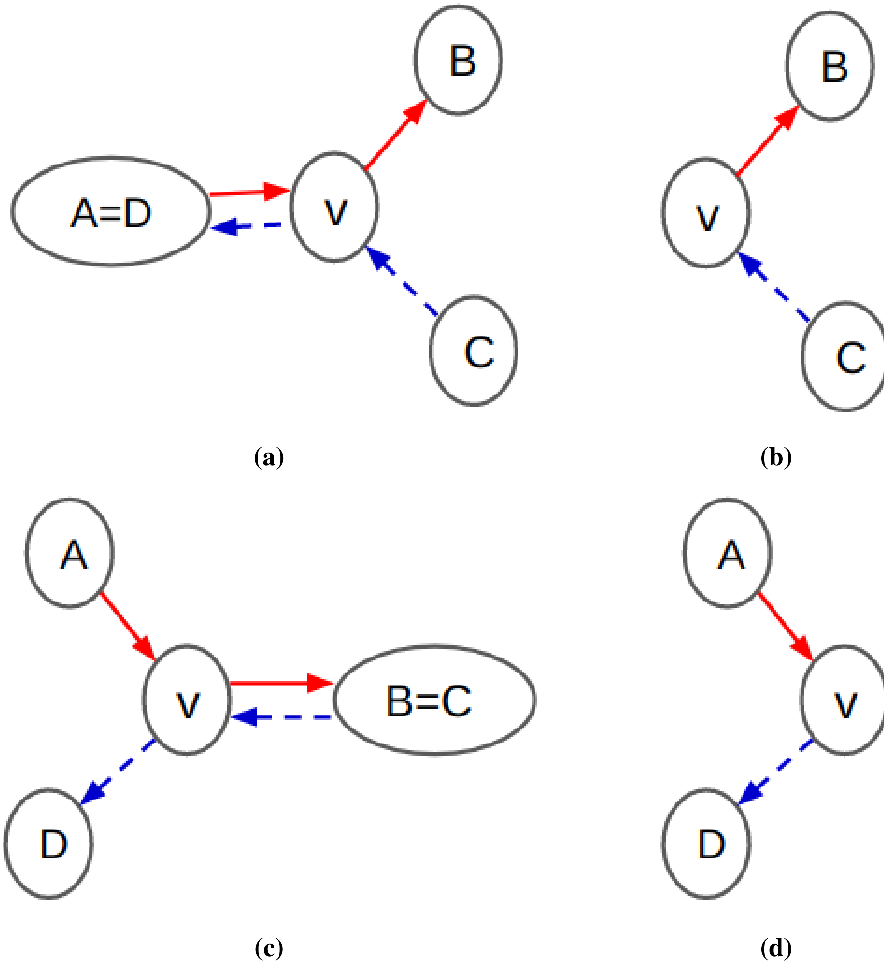


Figure 6.8: In this figure, the common vertex v has degree 3. The parents are aligned in reverse direction. Figures (a) and (c) show the two possible cases of vertices before partition. Figures (b) and (d) show the vertices after deleting the common edges.

Figures 6.7 and 6.8 illustrate the splitting of degree 3 vertices in the forward and reverse directions, respectively. There are two possible scenarios for degree 3 vertices in the forward direction. Either the previous neighbours of the tours are same (A=C) or the next neighbours are the same (B=D). Similarly, there are two possible scenarios for degree 3 vertices in the reverse direction. Previous and next neighbours of the tours are same (A=D)and (B=C). In all cases, after deleting the common edges, the degree 3 vertices become degree 2.

Figures 6.9 and 6.10 illustrate the splitting of degree 4 vertices in the forward and reverse directions, respectively. The degree 4 vertices are split, by introducing a common edge with zero

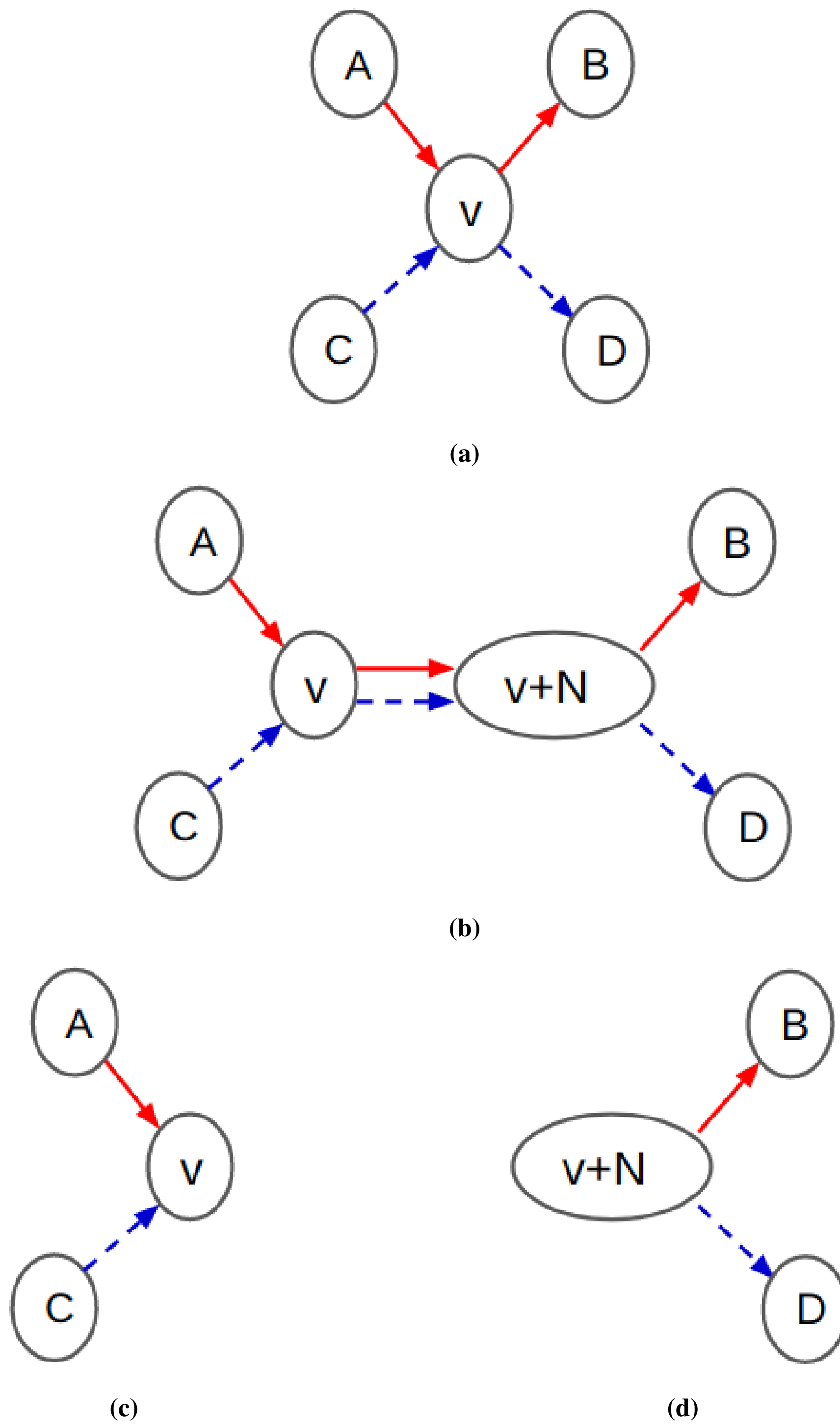


Figure 6.9: Figure (a) shows a degree 4 common vertex v. The parents are aligned in forward direction. In figure (b), the degree 4 vertex is split by introducing a ghost vertex. Note that the numbering of the ghost node is v+N. The common edge is removed and the vertex is split into (c) and (d) partitions.

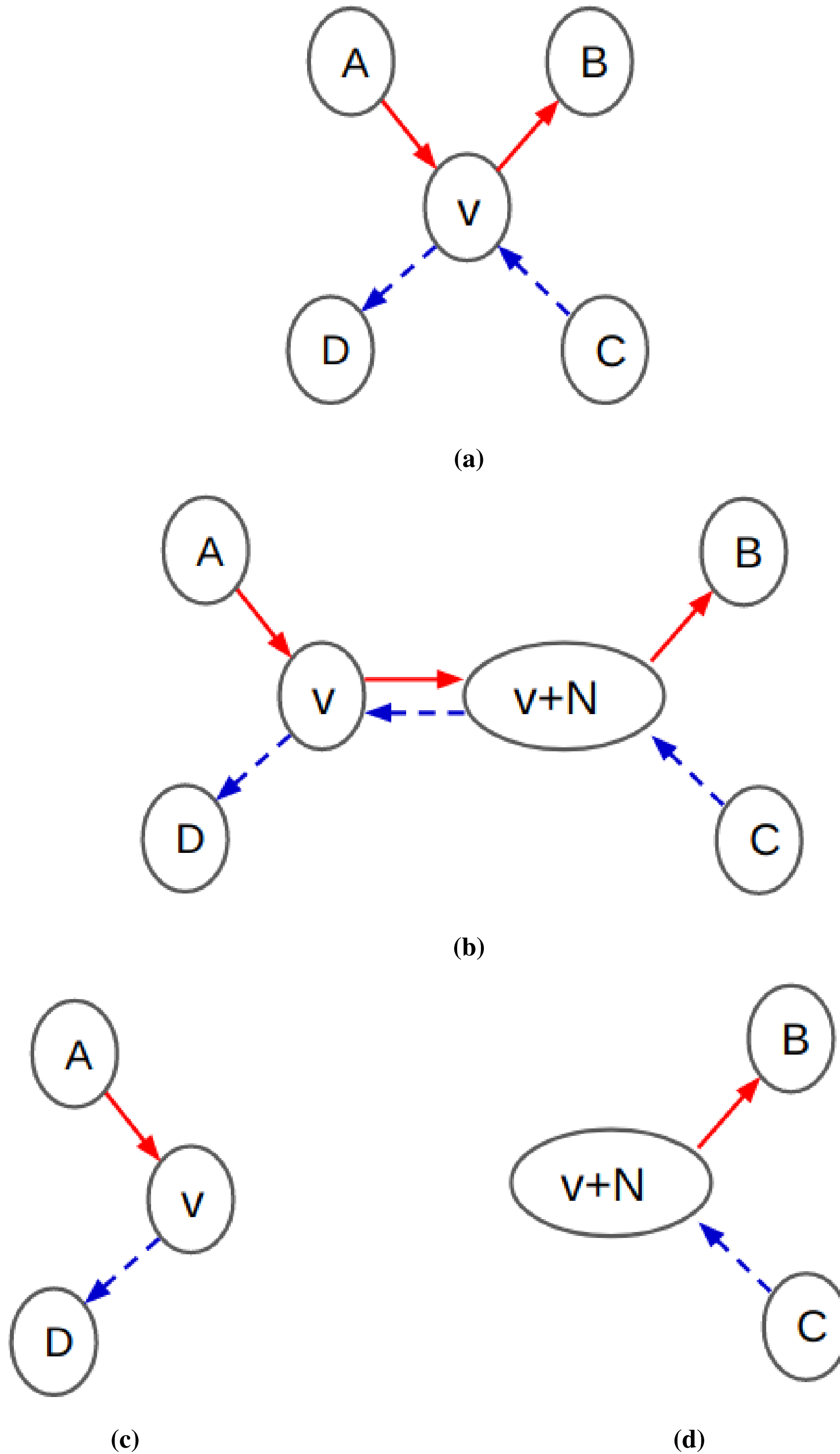


Figure 6.10: Figure (a) shows a degree 4 common vertex v. The parents are aligned in reverse direction. In figure (b), the degree 4 vertex is split by introducing a ghost vertex. Note that the numbering of the ghost node is v+N. The common edge is removed and the vertex is split into (c) and (d) partitions.

Table 6.2: Look up table. "X" denotes the dont't care condition.

Degree	Common edge	Direction	Is ghost node?	Old node degree		New node	
				Previous	Next	Previous	Next
2	X	X	No	X	X	-	-
3	B=D	Forward	No	3	3	A	C
				4	4	A+N	C+N
				3	4	A	C+N
				4	3	A+N	C
	A=C	Forward	No	X	X	B	D
	A=D	Reverse	No	X	3	B	C
				X	4	B	C+N
	B=C	Reverse	No	3	3	A	D
				4	4	A+N	D
				3	4	A	D
				4	3	A+N	D
4	v=v+N	Forward	No	3	3	A	C
				4	4	A+N	C+N
				3	4	A	C+N
				4	3	A+N	C
		Forward	Yes	X	X	B	D
		Reverse	No	3	3	A	D
				4	4	A+N	D
				3	4	A	D
				4	3	A+N	D
		Reverse	Yes	X	3	B	C
				X	4	B	C+N

weight. Again, the degree 4 vertices, after splitting and deletion of the common edges, become degree two vertices.

In figures 6.9 and 6.10, note that the ghost node (v+N) has a different numbering. The original GPX code starts numbering from $N, N+1, ..., N+$(total ghost nodes). The GPX does this because, it creates new data structures depending on the number of ghost nodes. However, in our implementation, we do not use any additional data structures. We simply find the degrees of all vertices. A simple look-up to this degree vector and the "edge-table" will tell the previous and next cities in the graph. There is no need to create tours with ghost nodes [87]. This look-up concept is introduced to reduce the memory and computational resources.

Table 6.2 shows the look up entries. Applying only the forward direction to our example tours will result in table 6.3. Note that in table 6.3, there are still nodes of degree-3 whose common edges are not deleted. This is because, those edges will be common if the blue tour is aligned in the reverse direction. Table 6.4 captures the scenario where the blue tour is reversed. Here all the common edges, except one is deleted. Therefore, to maximize the number of partitions, we need to consider both the forward and reverse directions.

Table 6.3: Example edge-table for a 10-city problem after splitting degree 4 vertices and deleting the common edges. - Forward direction

Node	Neighbour-1 (prev-red)	Neighbour-2 (next-red)	Neighbour-3 (prev-blue) forward	Neighbour-4 (next-blue) forward
1	9	-	10	-
1’	-	3	-	2
2	3	5	1	3
3	1	2	2	4
4	5	6	3	5
5	2	4	4	6
6	4	-	5	-
6’	-	8	-	7
7	-	10	-	9
8	6	7	7	9
9	10	1	8	10
10	7	9	9	1

To find the degrees of the vertices, each parallel unit operates on one or group of cities, does a simple comparison of the “edge table” entries. The CPU code uses 9N memory for this operation. However, the parallel code uses only one vector of length N. The memory savings is 8N.

6.2.3 Identifying recombining components

Finding the recombining components is similar to finding the connected components (CC) in a graph. Therefore, we modify one of the state-of-the-art GPU based CC algorithms, namely, the ECL-CC. There are several other GPU-based CC algorithms [119]. However, the ECL-CC code

Table 6.4: Example edge-table for a 10-city problem after splitting degree 4 vertices and deleting the common edges. - Reverse direction

Node	Neighbour-1 (prev-red)	Neighbour-2 (next-red)	Neighbour-3 (prev-blue) reverse	Neighbour-4 (next-blue) reverse
1	9	-	2	-
1’	-	3	-	10
2	-	5	-	1
3	1	-	4	-
4	-	6	-	3
5	2	-	6	-
6	4	-	7	-
6’	-	8	-	5
7	-	10	8	9
8	6	-	9	-
9	-	1	-	8
10	7	-	1	-

is publicly available [2] and easy to access and understand. The ECL-CC algorithm is based on the label propagation concept. Each vertex is associated with a label. The algorithm progresses so that all vertices in a component have the same label at the end.

There are three major steps to the algorithm. The first step involves the initialization of the labels. The label of each vertex is initialized to the smallest neighbor of each city. At the end of the algorithm, all vertices in the component will have a label equal to the smallest vertex. This type of initialization ensures that each component has a unique label. Note that a node is a neighbor to itself. Figure 6.11 demonstrates this concept for the example considered in figure 6.1. Here, we show the label propagation for only one partition (**1’(11)-2-5-6-4-3**). The smallest vertex number of this component is 2. In figure 6.11, the subscripts are the labels associated with each vertex. It can be seen that only 3 out of 6 vertices are labeled as 2 in the initialization phase. The three vertices correspond to vertex two and its neighbors (1’(11) and 5).

Listing 6.1: The intermediate pointer jumping

[2]https://userweb.cs.txstate.edu/ burtscher/research/ECL-CC/

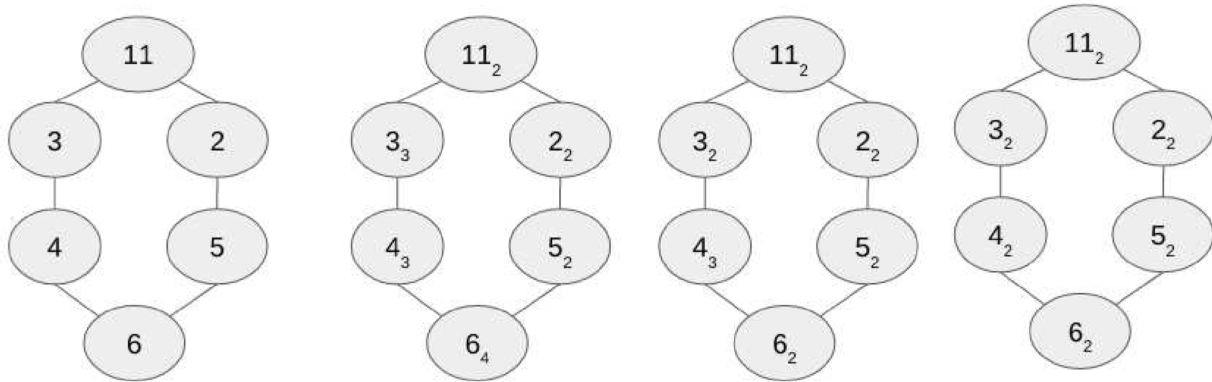


Figure 6.11: Figure showing the steps in ECL-CC. The four graphs from left to right: (a) Initial graph (b) Initialization (c) Intermediate Pointer Jumping and (d) Hooking. The subscripts represent the label.

```
1  /* intermediate pointer jumping */
2  static inline __device__ int representative
3      (const int idx, int* const __restrict__ label){
4    int curr = label[idx];
5    if (curr != idx) {
6      int next, prev = idx;
7      while (curr > (next = label[curr])) {
8        label[prev] = next;
9        prev = curr;
10       curr = next;}}
11   return curr;}
```

The second step is to propagate the smallest label to each of the neighbors. The ECL-CC code uses "intermediate pointer jumping" and "hooking" concepts to achieve this. The connected components, in our case, are all cycles. However, the depth of the vertices from the smallest vertex is different. The pointer jumping concept is used to reduce the depth to the root node. In figure 6.11, vertex two is considered as the root node since it is the smallest vertex. The vertices 11 and 5 are at depth one relative to vertex 2 - similarly, vertices 3 and 6 at depth two and vertex four at

depth 3. The "intermediate pointer" jumping concept reduces the depth of the vertices to the root node by a factor of 2. For each vertex, the intermediate pointer jumping is applied in parallel. The corresponding code is shown in listing 6.1.

Following the code in listing 6.1, the labels of vertices 3 and 6 are updated to 2 at the end of the "intermediate pointer jumping" step. Note that line-8 has potential race conditions. If vertex four is processed just after vertex 3's label is updated, then vertex 4's label will be 2. But, if vertex four is processed just before vertex 3's update, the label of vertex four will be 3. The authors [120] ignore these race conditions stating that the cost of synchronization is much higher compared to the few race conditions. The presence of race condition does not give wrong results, instead, can result in slow updates of the labels. For our discussion, let us consider that vertex four still has the label 3 to it. After applying the intermediate pointer jumping to each vertex, the neighbors are inspected. The "hooking concept" makes sure that the vertex and its neighbors all have the same label. Thus, at the end of the hooking step, the label of vertex four is updated to 2.

The example we considered is simple. However, there can be more complex partitions. In such cases, at the end of the hooking stage, the vertices may indirectly point to the root. To overcome these challenges, the algorithm employs iterative pointer jumping to make sure that each vertex has the appropriate component ID. To summarize, we make two major modifications to the ECL-CC:

1. The original ECL-CC code uses the Compressed Adjacency List (CAL) representation. We use the edge-table representation. The edge table is nothing but the CAL with a fixed number of neighbors. The CAL sorts the neighbor's list. However, we do not sort. We want to preserve the order of the vertices entered in the edge table.

2. The original ECL-CC code uses the "waitlist" data structure to process higher degree vertices. However, since the degree of vertices in our case is at most 4, we use only the codes required for processing "low-degree" nodes. The total memory savings is N.

6.2.4 Integrating with the recombination phase

The partition phase takes the two tour (each of length N) permutations as input. It produces the label vector (of length 2N) as the output. Note that, in the worst case, we can have a tour with all degree 4 vertices. In the best case, the two tours may not have degree 4 vertices. However, to avoid dynamic memory allocation, we fix the length of label vector to 2N.

The label vector is used to create the data structures required for the recombination phase. Note that we do not generate ghost tours for the partition phase. However, the CPU version of the recombination phase requires the modified tours with the ghost nodes and the label vector as input.

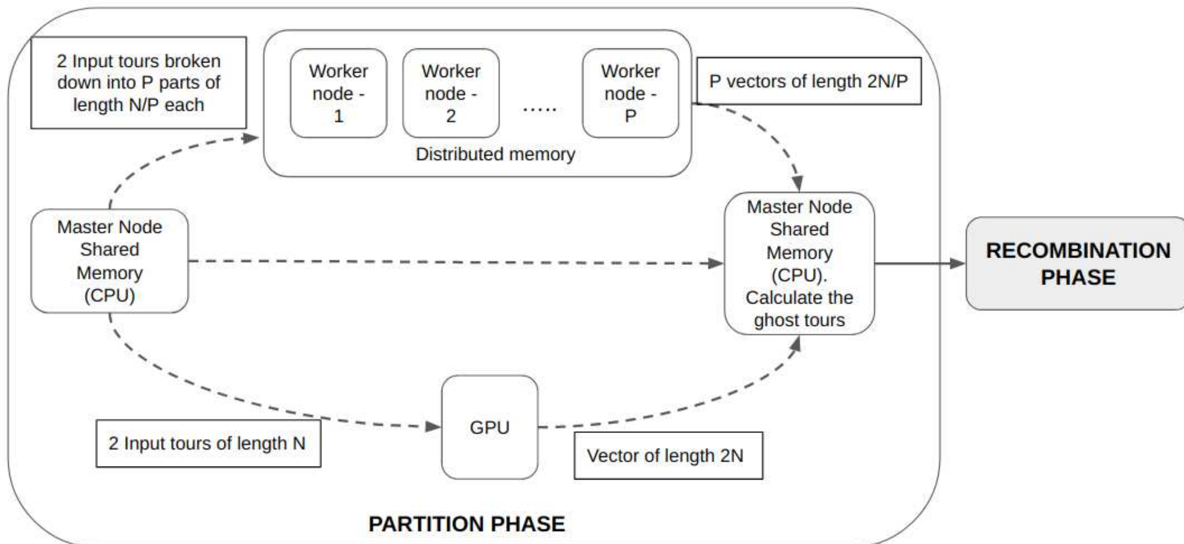


Figure 6.12: Figure showing the integration of partition phase with the recombination phase.

Figure 6.12 illustrates the integration process. For the shared memory architecture, there is no copying of the vector. For the distributed memory architecture, there is a need to gather parts of the label vector from each worker node to the master node. For the GPUs, there is a need to copy this vector to the CPU. Therefore, the two overheads associated with integrating the partition and recombination phases are (1) copying the label vector and (2) creating the tours with ghost nodes. The second step requires at most 7N memory and O(N) time.

6.2.5 Integrating with MGA

The original GPX takes two parents as input, produces two children as output. There is a need to do the forward and reverse partition phases to create the same two children. The original naive CPU code does both the phase in a single traversal. However, to explore operator-level parallelism, the implementation developed involves doing the forward and reverse phases separately.

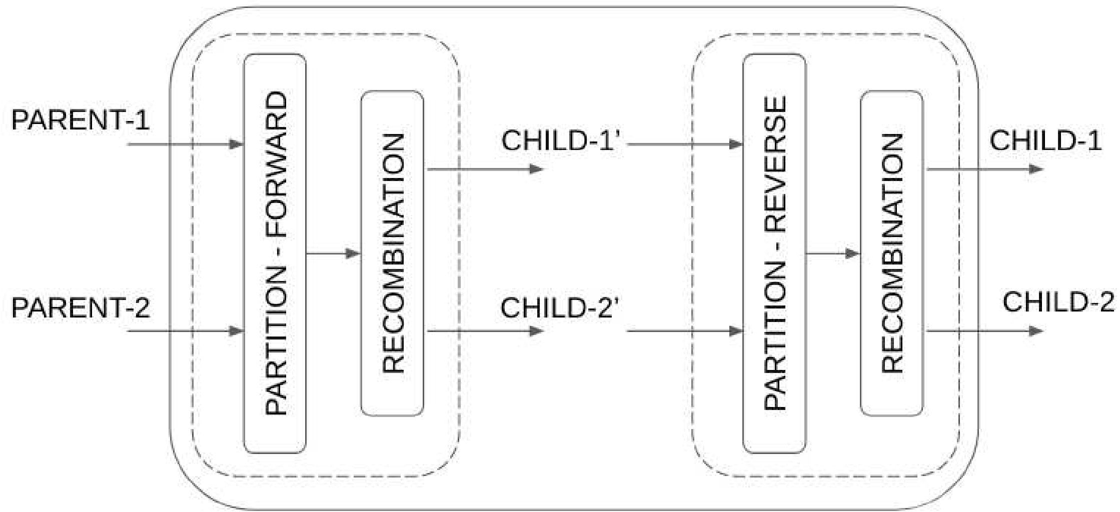


Figure 6.13: Figure showing the block diagram for integrating the fine-grain partition phase with MGA.

In figure 6.13, the two-parent tours are considered in the forward direction. They produce two children, namely, child-1' and child-2'. These two children are not the same as those that the original GPX code would produce. When the child-1' and the reversed child-2' are again recombined, they create the same child-1 and child-2 tours. However, the overhead of doing the recombination twice is huge and diminishes the benefits of the accelerated partition phase.

To overcome this overhead, we can randomly choose to do a forward or a reverse partition phase. Since MGA keeps mixing the population, there won't be any actual loss of performance. Note that the number of partitions of forward and reverse phases will always be less than the original GPX code. Because of this, we predict that the number of fusions will also be less. Therefore, separating the forward and reverse phases reduces the overhead of fusions in the recombination.

6.3 Experimental setup

– **Dataset:** The dataset consists of 14 TSP instances raning in size between 10K and 2M cities. Six instances (10,000 to 85,900 cities) are from TSPLIB [110]. Six large instances (100,000 to 200,000 cities) are Art TSP [3] problems. The remaining are 3D Star TSP [4] problems.

– **Machine used:** The experiments were performed on the RMACC Summit Supercomputer.

– **GPU specifications:** The GPU used is a NVidia Tesla K80. It has 4,992 NVIDIA CUDA cores with a dual-GPU (GK210) design. However, for our experiments, we use only one GPU. The 2,496 cores are distributed over 13 multiprocessors that can hold 26,624 threads. We set the threads per block to 256. We use all the 26,624 threads, and thus, we have 104 blocks (= 26,624/256). Each multiprocessor has L1 Cache and is configurable from 16 KB up to 48 KB. We prefer a larger L1 cache and smaller shared memory. So, the "cudaFuncCachePreferL1" option is set to 1. The multiprocessors share a 1.5 MB L2 cache and 12 GB of global memory with a peak bandwidth of 240.6 GB/s.

– **CPU specifications:** CPU used is Intel Xeon E5-2680v3 @2.50GHz with 12 cores and 24 threads. However, we use sequential execution for our experiments.

– **Codes:** The GPX CPU code is taken from our previous experiments. The partition phase is replaced with the modified GPU code. The TSP tours are randomnly initialized. The current time is used as the random seed. The codes are compiled using g++ and nvcc compilers.

– **Evaluation:** We use metrics such as the execution time, number of partitions, number of fusions and the best solution for evaluation. The execution time for each phase (partition, recombination) is provided. Each instance is run for 30 times and the average is reported.

The scope of these experiments is limited to preliminary comparison of the naive sequential CPU and the proposed GPU implementations of one single GPX crossover operation.

---

[3]http://www.math.uwaterloo.ca/tsp/data/art/

[4]http://www.math.uwaterloo.ca/tsp/star/

6.4 Results

Table 6.5: Results on applying the naive CPU-based GPX operation without fusions

Instance	Execution time			Number of		Best Solution
	Part.	Recomb.	Total	Partitions	Fusions	
rl11849	0.03	0.04	0.07	11.50	0.00	86,988,761.19
usa13509	0.02	0.04	0.08	9.17	0.00	2,149,561,571.26
d15112	0.03	0.04	0.08	10.06	0.00	126,622,708.50
d18512	0.04	0.06	0.10	9.47	0.00	59,251,067.71
pla33810	0.12	0.15	0.29	12.68	0.00	9,409,773,098.00
pla85900	0.42	0.39	0.85	13.83	0.00	28,802,029,304.17
mona-lisa100K	0.58	0.43	1.06	13.82	0.00	932,050,394.76
vangough120K	0.50	0.51	1.28	12.57	0.00	1,232,959,497.00
venus140K	0.96	0.73	1.97	15.00	0.00	1,370,440,388.00
pareja160K	1.12	0.80	1.99	13.77	0.00	1,674,142,147.77
courbet180K	0.90	0.77	1.76	9.64	0.00	1,525,679,504.82
earring200K	1.27	0.96	3.27	17.60	0.00	2,190,701,839.60
star250K	0.98	1.20	2.85	14.17	0.00	525,489,891.25
gaia2079471	25.20	12.99	40.83	19.05	0.00	26,308,411,447.48

Table 6.6: Results on applying the GPU-partition phase of GPX operation without fusions

Instance	Execution time			Number of		Best Solution
	Part.	Recomb.	Total	Partitions	Fusions	
rl11849	0.00046	0.04	0.11	8.07	0.00	86,778,488.73
usa13509	0.000489	0.05	0.12	9.17	0.00	2,148,903,067.37
d15112	0.000496	0.05	0.12	8.60	0.00	133,668,652.10
d18512	0.000542	0.06	0.20	8.40	0.00	59,368,123.07
pla33810	0.000763	0.12	0.20	9.90	0.00	9,395,107,125.80
pla85900	0.001424	0.39	0.50	10.83	0.00	28,825,312,523.67
mona-lisa100K	0.001632	0.48	0.59	10.87	0.00	989,750,836.80
vangough120K	0.001913	0.58	0.71	10.86	0.00	1,233,029,426.55
venus140K	0.002188	0.69	0.82	10.92	0.00	1,369,750,862.20
pareja160K	0.002459	0.80	0.95	11.61	0.00	1,674,070,200.46
courbet180K	0.00272	0.90	1.06	13.35	0.00	1,864,955,140.73
earring200K	0.002948	1.03	1.20	13.27	0.00	2,189,681,466.42
star250K	0.003276	1.31	1.71	14.50	0.00	525,231,014.04
gaia2079471	0.041593	16.13	18.58	49.71	0.00	26,309,721,374.86

Table 6.7: Results on applying the naive CPU-based GPX operation with fusions

Instance	Execution time			Number of		Best Solution
	Part.	Recomb.	Total	Partitions	Fusions	
rl11849	0.03	0.05	0.08	12.21	2.32	87,017,257.05
usa13509	0.03	0.06	0.09	9.30	4.10	2,148,757,196.50
d15112	0.03	0.06	0.09	9.47	2.05	126,982,162.11
d18512	0.05	0.08	0.13	9.79	4.74	59,265,926.53
pla33810	0.13	0.21	0.36	13.16	3.53	9,411,080,553.21
pla85900	0.45	0.49	1.02	13.00	2.31	28,806,066,955.23
mona-lisa100K	0.52	0.51	1.10	13.00	2.45	873,784,700.88
vangough120K	0.45	0.65	1.37	14.43	4.57	1,233,094,126.43
venus140K	1.03	1.09	2.28	15.11	5.33	1,370,384,067.78
pareja160K	0.88	1.19	2.26	11.33	5.22	1,675,566,356.44
courbet180K	1.10	1.18	2.37	12.55	3.91	1,694,994,304.09
earring200K	1.73	1.14	3.29	17.86	3.86	2,190,793,333.29
star250K	1.01	1.22	2.53	10.95	3.18	429,946,845.91
gaia2079471	25.53	17.78	45.95	18.20	2.60	26,308,762,203.50

Table 6.8: Results on applying the GPU-partition phase of GPX operation with fusions

Instance	Execution time			Number of		Best Solution
	Part.	Recomb.	Total	Partitions	Fusions	
rl11849	0.00046	0.05	0.18	7.93	2.00	86,821,504.17
usa13509	0.00049	0.06	0.19	9.30	1.50	2,147,242,632.27
d15112	0.000498	0.06	0.19	8.23	1.60	133,722,619.87
d18512	0.000545	0.09	0.22	9.70	1.43	59,324,795.33
pla33810	0.000761	0.15	0.29	8.69	2.10	9,397,400,864.97
pla85900	0.001433	0.48	0.65	11.71	1.43	28,833,856,854.50
mona-lisa100K	0.001629	0.56	0.73	10.97	0.93	990,025,677.70
vangough120K	0.001917	0.66	0.84	10.46	1.12	1,233,331,280.12
venus140K	0.00218	0.79	0.99	11.48	0.96	1,369,838,129.00
pareja160K	0.002431	0.95	1.15	12.77	1.00	1,674,061,540.03
courbet180K	0.002603	1.05	1.27	11.92	0.75	1,864,951,850.92
earring200K	0.002849	1.21	1.44	13.37	1.19	2,189,589,956.26
star250K	0.003114	1.47	1.88	13.67	0.85	525,163,694.37
gaia2079471	0.040828	16.14	18.62	48.50	0.50	26,311,532,443.50

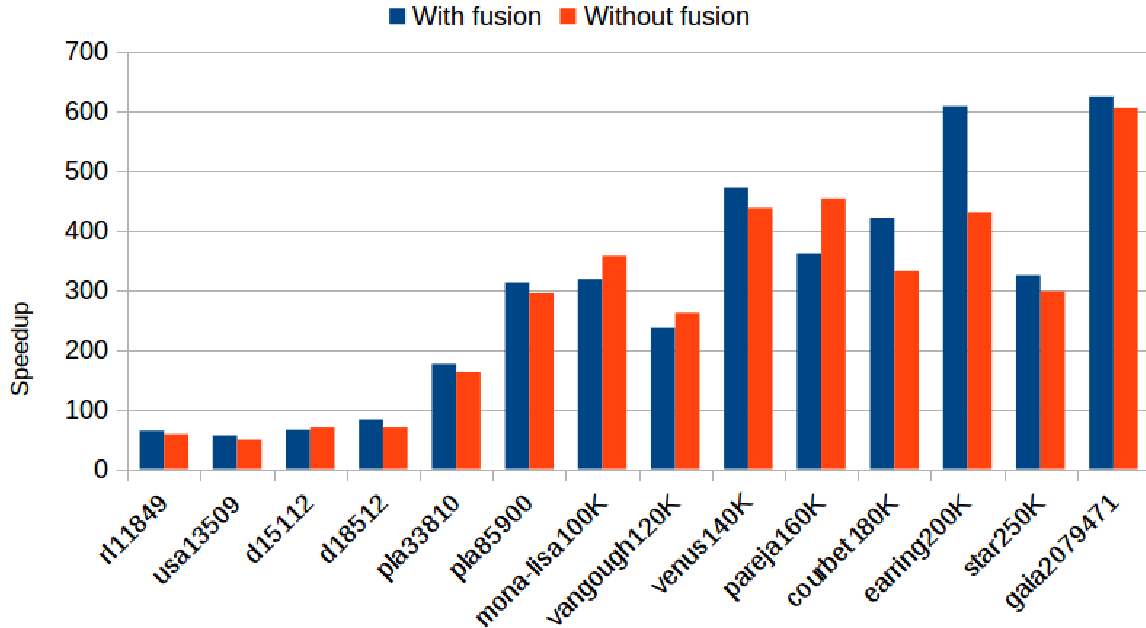


Figure 6.14: Average Speedup of the GPU-based partition phase over the naive sequential CPU codes. The number of GPU threads=26,624 (104 blocks, 256 threads per block).

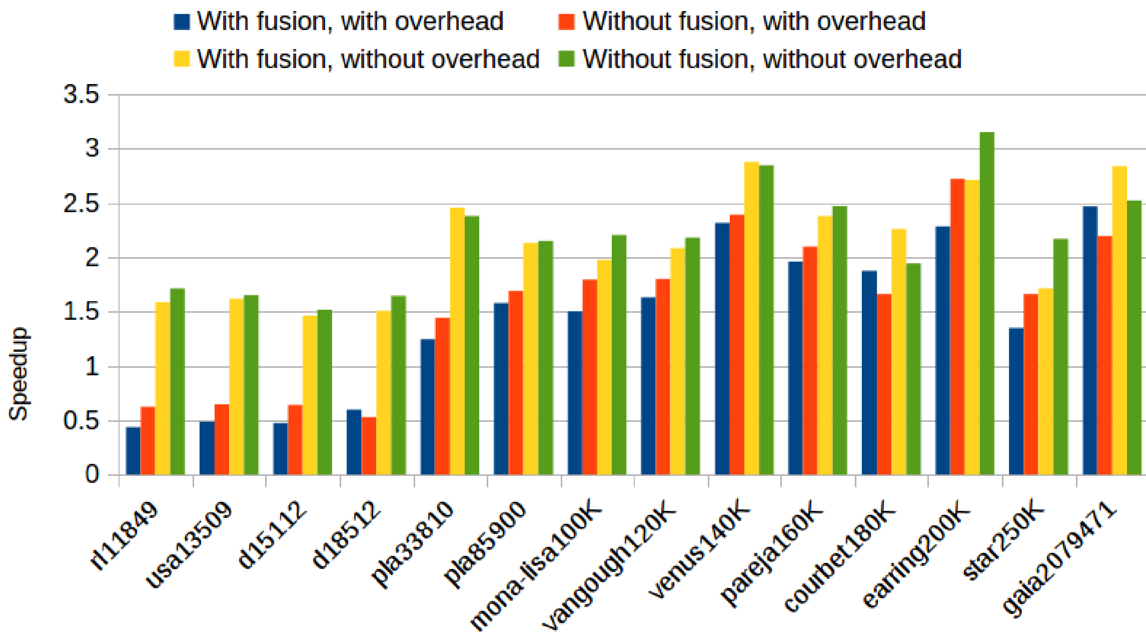


Figure 6.15: Speedup of the GPU+CPU based GPX operation over the naive sequential CPU codes. The number of GPU threads=26,624 (104 blocks, 256 threads per block). The overhead includes memory allocation, deallocation, transfers between CPU and GPU.

6.4.1 Execution time speedup

Tables 6.7 and 6.5 shows the results on applying the naive CPU code on two arbitrary inputs. The execution time is similar for the fusion and non-fusion versions. The version without fusions is on average 1.14$\times$ fast when compared to the version with the fusions.

Tables 6.8 and 6.6 shows the results on applying the GPU+CPU based GPX code on two arbitrary inputs. The execution time is similar for the fusion and non-fusion versions. The version without fusions is, on average 1.29$\times$ fast when compared to the version with the fusions.

Figure 6.14 shows the speedup of the GPU-based partition phase over the CPU version. The two traces correspond to the fusion and non-fusion versions. Ideally, there shouldn't be any different because the fusions occur only in the recombination phase. However, in figure 6.14, we can see that for 10/14 instances, the version with fusions shows a slightly higher speedup. The difference is much higher for the instance "earring200K". The only explanation here has to do with the initialization. The two codes use time as the seed to initialize the tours.

The GPU version is extremely fast. The speedup ranges from 48$\times$ to 625$\times$. The number of GPU threads is fixed to 26,624 spread across 104 blocks with 256 threads per block. The speedup is also because the GPU version does only the forward phase partition. In contrast, the CPU code does both the forward and reverse phases. However, the performance does not significantly differ when we use just the forward phase. We discuss performance differences in the section 6.4.2.

Figure 6.15 shows the speedup of the entire crossover operation. It has four traces to show the differences due to fusion operations and the "overhead". The overhead includes memory allocations and transfers between the GPU and CPU. The versions without overhead shows the maximum speedup of 1.5$\times$ to 3$\times$. The versions with overhead show a speedup of 1.2$\times$ to 2.5$\times$ for instances higher than 33,810 cities. On smaller size instances, we see a slow-down. This observation makes sense because we are using a fixed number of threads of 26,624. Notice that the overhead reduces with problem size. Thus,to achieve appropriate performace gains, in the future, the number of threads has to be modified according to the problem size.

6.4.2 On partitions, fusions and performance

The GPU versions perform only the forward phase of the partition. The CPU version, however, performs both the forward and reverse-phase in a single tour traversal. Owing to this difference, the number of partitions found by the GPUs must be lesser than the CPU counterpart. However, on 3/15 instances (courbet180K, star250K, and gaia2079471), the GPU version makes more partitions. For gaia2079471, we get a massive difference in the number of partitions. The reason simply being in the order the tour is traversed. The reduced number of GPU partitions is also an advantage because the number of fusions is correspondingly reduced.

Tables 6.7, 6.5, 6.8 and 6.6 show the average best solutions. It can be seen that for 6/14 instances, the CPU version without the fusions performs the best. The GPU fusion-less versions perform the best on 3/14 instances. 5/14 versions with the fusion are found to be better. It is surprising to observe that the codes without fusions perform much better. Therefore, with the appropriate tuning of the code, there are opportunities to reduce the overhead associated with fusions. It is also surprising to note for 6/14 instances, the GPU code performs better than the CPU versions. This performance difference shows that breaking the partition stage into forward and reverse phases do not necessarily decrease the performance.

The preliminary data discussed here shows the variation of performance with fusions and partition phases. Another issue is the random initialization of the tours. However, when integrating with the MGA, a local search solver will improve the initial tour. Therefore, a clear understanding of various GPX parameters has to be performed in the future.

6.4.3 Memory savings

The CPU code allocates temporary memory for the partition phase. However, we do not use such a data structure for the GPUs. We save somewhere between 17N and 28N memory. Note that the range is because of the introduction of "ghost nodes". In the best case, there are no ghost nodes. In the worst case, all vertices can have a ghost node. With the GPU implementation, there is no need to dynamically change the memory because of the look-up table concept.

6.5 Conclusions

Overall, the central message of this chapter is the substantial performance benefits of performing operator-level parallelization. The parallel implementation of the partition phase of the GPX crossover operator has been introduced in this chapter. With the GPU implementation, we get a range of 48× to 625× speedup over the naive sequential CPU implementation. Furthermore, the GPU implementation reduces the GPX memory consumption by 17N. This GPU implementation is the first of its kind to be tested on problems as big as 2 million cities. To the best of our knowledge, this is the first fine-grain implementation of Genetic algorithm (GA) operators for the Traveling Salesman Problem (TSP).

Numerous future works can stem from this preliminary finding. First, parallelizing the recombination phase should yield higher speedup. The recombination phase consists of two primary operations. The first operation is to test if each partition is a recombining component. This test should involve a modified Hamiltonian pathfinding problem. A partition is a recombining component if each partitioned parent tours form a Hamiltonian sub-path. The second operation of the recombination phase is to find the best and worst partition. At the end of the recombination phase, the best child replaces the first parent (say, parentA). The worst child replaces the second parent (say, parentB). To find the best partition, we compare the lengths of the partitioned tours. If parentB's partition has a smaller length, the parentA and parentB sub-tours are swapped. In terms of arithmetic operation, this step involves adding, comparing, and exchanging data. The functions (finding Hamiltonian path, addition, comparison, and swapping) on each partition can be parallel. However, some steps inside a partition are sequential. Thus, the fine-grain parallelism of the recombination phase is a challenge. It can produce valuable results and insights to the Genetic Algorithm (GA) and High-performance computing (HPC) community.

The future work is to increase the degrees of parallelization by cascading multi-CPU and GPU. Finally, integrating these in the MGA framework, exploring several parallel implementations is guaranteed to improve execution time.

Chapter 7

Future work

The Mixing Genetic Algorithm (MGA) developed in this thesis can be improved in several ways. In this chapter, we divide the construction of the solver into four major categories. Sections 7.1, 7.2, 7.3 and 7.4 describe the possible future work in each of those categories. Finally, section 7.5, discusses the generalization and use of MGA beyond Traveling Salesman Problem (TSP).

7.1 Initialization techniques

For the TSP, the initial population of MGA is assumed to have all the global optima edges. In our experiments, we showed that the population, usually after 2opt operation contains more than 99% of the global edges. However, this assumption is far too ideal. To have a realistic solver, there is a need to (a) introduce new edges in the population or (b) create a population with all the edges.

The time complexity of the 2opt is $\theta(N^{2.29})$, prohibitely high on large problems. Therefore, low complexity heuristics ($\theta(N^{1.6})$) such as the POPMUSIC [49] can be explored.

The MGA population must be large to hold global optima edges. MGA converges fast with higher population sizes. However, the initialization cost is huge for a larger population. Several ways to overcome this challenge include (a) **Duplicate the population created by 2opt**. Since the 2opt produces a population with an equal proportion of global and non-global edges (check table 4.1), we can say that duplication helps maintain the edges' proportion. (b) **Use frequency of edge information to construct tours**. 2opt produces a population with a higher frequency of global optima edges. New tours using these higher frequency edges can help reduce the initialization cost and create optimized tours. (c) Use a combination of several local search heuristics.

Finally, the initial population size is different for each problem type. In chapter 5, we saw that some problems require a larger population to contain all global optima edges. Some require only a smaller population. The user cannot determine this in real-time. Therefore, in the future, one can explore techniques to select the population size dynamically [121].

7.2 Convergence techniques

The next challenge of MGA is the slower convergence with larger problem sizes. One way to combat this issue is to develop techniques to increase the number of partitions [118]. With more partitions, the number of solutions explored is high; thereby, MGA can converge faster.

The convergence depends on several parameters such as the population size, randomization after epochs, and size of an epoch. The movement of individuals in the population determines the mixing rate and pattern. Therefore, an appropriate study to explore a deterministic movement of individuals can help understand how the global edges are mixed to make up the final solution.

The convergence rate also depends on the individuals that are selected for recombination. Currently, individuals are chosen randomly. Some recombination does not produce an improving move. A correlation between the improving move and parent properties such as the fitness value, number of common edges can understand the usefulness of the recombination [122] .

Finally, in Chapter 5, we saw that the replacement techniques produce better individuals. Although replacement violates the genetic invariance property of MGA, it can help understand the convergence rate of the solver. Future work on studying this aspect can help with developing appropriate diversity preserving mechanisms.

7.3 Execution time improvements

The island model introduced in Chapter 5 has several paramaters such as the migration policy, migration topology and replacement policy. These parameters should be studied in more detail for further execution time improvements. The operator-level methods introduced in Chapter 6 has several future directions. Parallelization of the recombination phase, integrating the whole setup in the MGA framework are some of the immediate next steps. Combining population-level and operator-level parallelism can improve the execution time as well.

In this thesis, we explored the MGA implementation on shared memory, distributed memory, GPUs and Mesh architectures. A future study could include a detailed analysis of MGA and GPX for a heterogenous system and on other hardware accelerators.

7.4 Performance and usefulness of the solver

The presence of multiple global optima helped understand the working of MGA. However, several other problem parameters include, the structure of the problem, the number of global optima and the distance between the global optima. A machine learning technique [16] that analysis the performance of the solver based on several of its feature is a viable future direction.

The MGA was shown to be helpful as a hybrid solver. The different hybrid algorithms use a different ratio of GPX and EAX operations. Future work on the detailed analysis of this ratio can help understand the benefits of the complementary nature of the two solvers. Furthermore, GPX has been used to improve Chained-LK [89], LKH [80] and Concorde [91]. Therefore, a hybrid of MGA and these solvers can also be explored in the future.

The MGA was used in an ensemble setup and improved the solver's performance on some challenging TSP problems. However, the default parameters of the EAX and LKH were used. Changing the parameters can produce different results. Thus, in the future, parameter tuning in an ensemble setup can help understand these solvers better. Furthermore, the ensemble setup can be explored on various problems, solvers, and solver parameters.

7.5 Generalization and beyond TSP

The Partition Crossover crossover operator has been used to solve other NP-Hard problems such as the Psuedoboolean optimization [96], MAXSAT [97] [98] and NK- landscapes [99]. The MGA can be used with those problems as well. The visualization techniques developed in chapter 4 can be applied to other population based TSP sovlers. For TSP, we used the edges to understand the mixing properties. However, for other NP-hard problems, a different metric can be used in a similar analysis. For example, consider the MAXSAT problem. It can be modelled as a graph problem with variables as vertices and interaction between them as the edges. This graph is called the Variable Interation Graph [97][98]. By using the visualization techniques, one can understand how the interaction between variables change so as to maximally satisfy the clauses.

Chapter 8

Conclusion

The genetic algorithms are embarrassingly parallel. However, the successful parallelization and mapping of the algorithms to parallel architectures highly depends on the genetic operators. This thesis looked into the parallelization of GA solvers for the traveling salesman problem (TSP). The TSP is the most studied problem in the literature, primarily because of its simplicity. Yet, the study of TSP has contributed to several discoveries and innovations in computing and optimization.

The TSP representation for genetic algorithm poses a challenge for parallelization because of the constraints on the chromosome order. For TSP, each chromosome is a real-valued vector, representing a tour permutation. Each number in the permutation is a city. A genetic operation on a city depends on other cities. Owing to these dependencies, the genetic operators designed for TSP are highly sequential. However, each genetic operation on a population can be performed in parallel. Parallelization of one single genetic operation is called operator-level parallelization. The parallelization of the population is called the population-level parallelization.

Parallelization has been studied on three different computer architectures - shared memory CPUs, distributed memory CPUs, massively parallel Single-Instruction Multipe-Data (SIMD) machines (Mesh, GPU). The state-of-the-art GA-based TSP solver is the Edge Assembly Crossover (EAX). The crossover operation is sequential. However, the population-level parallelism of EAX can find the best-known solution to the 200,000 city problem. Porting the same algorithm on to SIMD machine poses a significant challenge because of two main reasons. First, the memory required per crossover operation limits the problem scalability. Second, the communication pattern per crossover operation is random, which is not favorable for SIMD machines.

In this thesis, we address the issues related to developing a GA solver for TSP that can make use of the parallelism available in the modern computer architectures. Note that addressing these issues is not a simple software engineering activity. It involves significant changes to the existing CPU-based algorithm. To summarize, we make the following contributions:

- We find that the Generalized Partition Crossover (GPX) is the best crossover operator for SIMD machines for two major reasons. It consumes 4x lesser memory compared to EAX. Second, it is a deterministic crossover operator. It does not introduce any new edges in the population. Therefore, there are no communication costs during the crossover operation.

- We modify the GPX not to lose any diversity in the population. The previous form of GPX finds only the best child out of the two given parent tours. The revised form finds both the best and the worst children. Suppose 2N edges make up the two-parent tours. With the previous version, we lose N edges per crossover operation. However, with the revised version, none of the edges are lost.

- The Mixing Genetic Algorithm (MGA) using the GPX crossover operator reaches global optima solution quickly for problems smaller than 2,000 cities. It achieves faster convergence on larger population sizes.

- The MGA progresses in epochs. Within an epoch, the individuals are mixed in a hypercube topology. One epoch consists of log(population size) generations. At the end of an epoch, the unique property of this epoch progression is the best individual migrates to the top of the population.

- We introduce visualization techniques to understand the performance of the population-based solvers. These techniques are helpful to understand population-based solvers, especially when there are multiple global optima solutions.

- We introduce several hybrid versions of MGA and EAX solvers. The hybrid variations exploit the two solvers' complementary nature and help solve some of the hard TSP instances.

- The shared memory parallelization using OpenMP is explored on problems ranging from 442 to 85,900 cities. The distributed memory parallelization using the MPI model is explored on problems ranging from 5,000 to 85,900 cities. A combination of openMP and

MPI parallelization is examined on problems ranging between 5,000 to 85,900 cities. We show near-linear speedup (proportional to the number of parallel units) on these instances.

- Preliminary results on GPU parallelization of the partition phase of GPX crossover operator show a 48x to 625x speedup over the naive sequential implementation. This is the first step towards the fine-grain parallelization of GA operators for TSP. The results are tested on problems ranging from 10,000 to 2M cities.
- An ensemble of solvers, running the state-of-the-art solvers (EAX, Lin-Kernighan-Helsgaun heuristic), and the hybrid version of MGA and EAX is introduced. The ensemble setup is tested on problems ranging between 5,000 and 85,900 cities. Results show the ensemble setup is cost-effective in solving a wide variety of problems. The MGA, again, is shown to improve the success rate of some hard TSP instances.
- Finally, this thesis has potential for several future research directions discussed in Chapter 9.

Following are the publications that came out of this thesis:

1. Varadarajan, S., & Whitley, D. (2021). A Parallel Ensemble Genetic Algorithm for the Traveling Salesman Problem. In Proceedings of the 2021 Genetic and Evolutionary Computation Conference (pp. 636-643). ACM.
2. Varadarajan, S., Whitley, D., & Ochoa, G. (2020). Why Many Travelling Salesman Problem Instances Are Easier than You Think. In Proceedings of the 2020 Genetic and Evolutionary Computation Conference (pp. 254-262). ACM.
3. Varadarajan, S., & Whitley, D. (2019). The Massively Parallel Mixing Genetic Algorithm for the Traveling Salesman Problem. In Proceedings of the 2019 Genetic and Evolutionary Computation Conference (pp. 872-879). ACM.

Bibliography

[1] D. Hains, D. Whitley, and A. Howe. Improving Lin-Kernighan-Helsgaun with crossover on clustered instances of the TSP. In *Proc. of PPSN XII*, pages 388–397. Springer, 2012.

[2] Kenneth D. Boese, Andrew B. Kahng, and Sudhakar Muddu. A new adaptive multi-start technique for combinatorial global optimizations. *Operations Research Letters*, 16(2):101 – 113, 1994.

[3] G. Ochoa and N. Veerapen. Deconstructing the big valley search space hypothesis. In *European Conference on Evolutionary Computation in Combinatorial Optimization*, pages 58–73, 2016.

[4] Tomohiro Harada and Enrique Alba. Parallel genetic algorithms: A useful survey. *ACM Comput. Surv.*, 53(4), August 2020.

[5] Thomas Weise, Raymond Chiong, Jorg Lassig, Ke Tang, Shigeyoshi Tsutsui, Wenxiang Chen, Zbigniew Michalewicz, and Xin Yao. Benchmarking Optimization Algorithms: An Open Source Framework for the Traveling Salesman Problem. *Comp. Intell. Mag.*, 9(3):40–52, August 2014.

[6] Ibrahim H. Osman and Gilbert Laporte. Metaheuristics: A bibliography. *Annals of Operations Research*, 63(5):511–623, Oct 1996.

[7] J. K. Lenstra and A. H. G. Rinnooy Kan. Some Simple Applications of the Travelling Salesman Problem. *Journal of the Operational Research Society*, 26(4):717–733, 1975.

[8] R. Agarwala, D. L. Applegate, D. Maglott, G. D. Schuler, and A. A. Schäffer. A fast and scalable radiation hybrid map construction and integration strategy. *Genome research*, 10(3):350–364, Mar 2000.

[9] J. Grey Monroe, Zachariah A. Allen, Paul Tanger, Jack L. Mullen, John T. Lovell, Brook T. Moyers, Darrell Whitley, and John K. McKay. TSPmap, a tool making use of traveling

salesperson problem solvers in the efficient and accurate construction of high-density genetic linkage maps. *BioData Mining*, 10(1):38, Dec 2017.

[10] Olin Johnson and Jing Liu. A traveling salesman approach for predicting protein functions. *Source Code for Biology and Medicine*, 1(1):3, Oct 2006.

[11] Large travelling salesman problems arising from experiments in X-ray crystallography: A preliminary report on computation. *Operations Research Letters*, 8(3):125 – 128, 1989.

[12] Gilbert Laporte. The traveling salesman problem: An overview of exact and approximate algorithms. *European Journal of Operational Research*, 59(2):231 – 247, 1992.

[13] David L. Applegate, Robert E. Bixby, Vasek Chvatal, and William J. Cook. *The Traveling Salesman Problem: A Computational Study (Princeton Series in Applied Mathematics)*. Princeton University Press, USA, 2007.

[14] Keld Helsgaun. An effective implementation of the lin-kernighan traveling salesman heuristic. *European Journal of Operational Research*, 126(1):106 – 130, 2000.

[15] Y. Nagata and S. Kobayashi. A powerful genetic algorithm using edge assembly crossover for the traveling salesman problem. *INFORMS Journal on Computing*, 25(2):346–363, 2013.

[16] Pascal Kerschke, Lars Kotthoff, Jakob Bossek, Holger H. Hoos, and Heike Trautmann. Leveraging TSP Solver Complementarity through Machine Learning. *Evolutionary Computation*, 26(4):597–620, 2018.

[17] X. Xie and J. Liu. Multiagent Optimization System for Solving the Traveling Salesman Problem (TSP). *IEEE Transactions on Systems, Man, and Cybernetics, Part B (Cybernetics)*, 39(2):489–502, 2009.

[18] Noriyuki Fujimoto and Shigeyoshi Tsutsui. A Highly-Parallel TSP Solver for a GPU Computing Platform. In Ivan Dimov, Stefka Dimova, and Natalia Kolkovska, editors, *Numerical*

Methods and Applications, pages 264–271, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[19] Kai Zhang, Siman Yang, Li li, and Ming Qiu. Parallel Genetic Algorithm with OpenCL for Traveling Salesman Problem. In Linqiang Pan, Gheorghe Păun, Mario J. Pérez-Jiménez, and Tao Song, editors, *Bio-Inspired Computing - Theories and Applications*, pages 585–590, Berlin, Heidelberg, 2014. Springer Berlin Heidelberg.

[20] Semin Kang, Sung-Soo Kim, Jongho Won, and Young-Min Kang. GPU-based parallel genetic approach to large-scale travelling salesman problem. *The Journal of Supercomputing*, 72(11):4399–4414, Nov 2016.

[21] Aryaf Al-Adwan, Basel A. Mahafzah, and Ahmad Sharieh. Solving traveling salesman problem using parallel repetitive nearest neighbor algorithm on OTIS-Hypercube and OTIS-Mesh optoelectronic architectures. *The Journal of Supercomputing*, 74(1):1–36, Jan 2018.

[22] Ranieri Baraglia and Raffaele Perego. Parallel Genetic Algorithms for Hypercube Machines. In Vicente Hernández, José M. L. M. Palma, and Jack J. Dongarra, editors, *Vector and Parallel Processing – VECPAR'98*, pages 691–704, Berlin, Heidelberg, 1999. Springer Berlin Heidelberg.

[23] L. Guo, C. Guo, D. B. Thomas, and W. Luk. Pipelined Genetic Propagation. In *2015 IEEE 23rd Annual International Symposium on Field-Programmable Custom Computing Machines*, pages 103–110, May 2015.

[24] Liucheng Guo, Andreea Ingrid Funie, David B. Thomas, Haohuan Fu, and Wayne Luk. Parallel Genetic Algorithms on Multiple FPGAs. *SIGARCH Comput. Archit. News*, 43(4):86–93, April 2016.

[25] Swetha Varadarajan and Darrell Whitley. The Massively Parallel Mixing Genetic Algorithm for the Traveling Salesman Problem. In *Proceedings of the Genetic and Evolutionary Computation Conference*, GECCO '19, pages 872–879, New York, NY, USA, 2019. ACM.

[26] Swetha Varadarajan, Darrell Whitley, and Gabriela Ochoa. Why many travelling salesman problem instances are easier than you think. In *Proceedings of the 2020 Genetic and Evolutionary Computation Conference*, GECCO '20, pages 254–262, New York, NY, USA, 2020. Association for Computing Machinery.

[27] Swetha Varadarajan and Darrell Whitley. A parallel ensemble genetic algorithm for the traveling salesman problem. In *Proceedings of the Genetic and Evolutionary Computation Conference*, GECCO '21, pages 636–643, New York, NY, USA, 2021. Association for Computing Machinery.

[28] Michael Held and Richard M. Karp. The traveling-salesman problem and minimum spanning trees: Part ii. *Mathematical Programming*, 1(1):6–25, Dec 1971.

[29] Jaroslav Nešetřil, Eva Milková, and Helena Nešetřilová. Otakar borůvka on minimum spanning tree problem translation of both the 1926 papers, comments, history. *Discrete Mathematics*, 233(1):3–36, 2001. Czech and Slovak 2.

[30] R. C. Prim. Shortest connection networks and some generalizations. *The Bell System Technical Journal*, 36(6):1389–1401, 1957.

[31] Joseph B. Kruskal. On the shortest spanning subtree of a graph and the traveling salesman problem. *Proceedings of the American Mathematical Society*, 7(1):48–48, January 1956.

[32] Ka Wong Chong, Yijie Han, and Tak Wah Lam. Concurrent threads and optimal parallel minimum spanning trees algorithm. *J. ACM*, 48(2):297–323, March 2001.

[33] Seth Pettie and Vijaya Ramachandran. A randomized time-work optimal parallel algorithm for finding a minimum spanning forest. *SIAM Journal on Computing*, 31(6):1879–1895, 2002.

[34] Michael Held and Richard M. Karp. The traveling-salesman problem and minimum spanning trees: Part ii. *Mathematical Programming*, 1(1):6–25, Dec 1971.

[35] Christine L. Valenzuela and Antonia J. Jones. Estimating the held-karp lower bound for the geometric tsp. *European Journal of Operational Research*, 102(1):157–175, 1997.

[36] Laurence A. Wolsey. *Heuristic analysis, linear programming and branch and bound*, pages 121–134. Springer Berlin Heidelberg, Berlin, Heidelberg, 1980.

[37] Michael Ian Shamos and Dan Hoey. Closest-point problems. In *16th Annual Symposium on Foundations of Computer Science (sfcs 1975)*, pages 151–162, 1975.

[38] D. S. Johnson, L. A. McGeoch, and E. E. Rothberg. Asymptotic experimental analysis for the held-karp traveling salesman bound. In *Proceedings of the Seventh Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '96, pages 341–350, USA, 1996. Society for Industrial and Applied Mathematics.

[39] David S. Johnson. Local optimization and the traveling salesman problem. In Michael S. Paterson, editor, *Automata, Languages and Programming*, pages 446–461, Berlin, Heidelberg, 1990. Springer Berlin Heidelberg.

[40] Nicos Christofides. Worst-case analysis of a new heuristic for the travelling salesman problem. 1976.

[41] Anna Karlin, Nathan Klein, and Shayan Oveis Gharan. An improved approximation algorithm for tsp in the half integral case, 2019.

[42] S. Lin and B. W. Kernighan. An Effective Heuristic Algorithm for the Traveling-Salesman Problem. *Oper. Res.*, 21(2):498–516, April 1973.

[43] K. Helsgaun. An effective implementation of the Lin-Kernighan traveling salesman heuristic. *European Journal of Oper. Res.*, 126(1):106–130, 2000.

[44] G. A. Croes. A Method for Solving Traveling-Salesman Problems. *Operations Research*, 6(6):791–812, 1958.

[45] Merrill M. Flood. The Traveling-Salesman Problem. *Operations Research*, 4(1):61–75, 1956.

[46] Jon Jouis Bentley. Fast Algorithms for Geometric Traveling Salesman Problems. *INFORMS Journal on Computing*, 4(4):387–411, 1992.

[47] Bruno Codenotti, Giovanni Manzini Luciano Margara, and Giovanni Resta. Perturbation: An efficient technique for the solution of very large instances of the euclidean TSP. *INFORMS Jounal on Computing*, 8(2):125–133, 1996.

[48] Weixiong Zhang and Moshe Looks. A novel local search algorithm for the traveling salesman problem that exploits backbones. In *Proceedings of the 19th International Joint Conference on Artificial Intelligence*, IJCAI'05, pages 343–348, San Francisco, CA, USA, 2005. Morgan Kaufmann Publishers Inc.

[49] Éric D. Taillard and Keld Helsgaun. POPMUSIC for the travelling salesman problem. *European Journal of Operational Research*, 272(2):420 – 429, 2019.

[50] Kamil Rocki and Reiji Suda. Accelerating 2-opt and 3-opt local search using gpu in the travelling salesman problem. In *2012 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (ccgrid 2012)*, pages 705–706, 2012.

[51] David Applegate, William Cook, and André Rohe. Chained lin-kernighan for large traveling salesman problems. *INFORMS Journal on Computing*, 15(1):82–92, January 2003.

[52] Olivier Martin, Steve W. Otto, and Edward W. Felten. Large-step markov chains for the traveling salesman problem. *Complex Systems*, 5:299–326, 1991.

[53] David L. Applegate, Robert E. Bixby, Vasek Chvatal, and William J. Cook. *The Traveling Salesman Problem: A Computational Study (Princeton Series in Applied Mathematics)*. Princeton University Press, USA, 2007.

[54] Richard Bellman. Dynamic programming treatment of the travelling salesman problem. *J. ACM*, 9(1):61–63, January 1962.

[55] Michael Held and Richard M. Karp. A dynamic programming approach to sequencing problems. In *Proceedings of the 1961 16th ACM National Meeting*, ACM '61, pages 71.201–71.204, New York, NY, USA, 1961. Association for Computing Machinery.

[56] G Dantzig, R Fulkerson, and S Johnson. Solution of a large-scale traveling-salesman problem. *Operations Research*, 2:393–410, 1954.

[57] J. A. Nelder and R. Mead. A Simplex Method for Function Minimization. *The Computer Journal*, 7(4):308–313, 01 1965.

[58] John D. C. Little, Katta G. Murty, Dura W. Sweeney, and Caroline Karel. An algorithm for the traveling salesman problem. *Oper. Res.*, 11(6):972–989, December 1963.

[59] Ton Volgenant and Roy Jonker. A branch and bound algorithm for the symmetric traveling salesman problem based on the 1-tree relaxation. *European Journal of Operational Research*, 9(1):83–89, 1982.

[60] S. Tschoke, R. Lubling, and B. Monien. Solving the traveling salesman problem with a distributed branch-and-bound algorithm on a 1024 processor network. In *Proceedings of 9th International Parallel Processing Symposium*, pages 182–189, 1995.

[61] David L. Applegate, Robert E. Bixby, VaÅąek ChvÃątal, William Cook, Daniel G. Espinoza, Marcos Goycoolea, and Keld Helsgaun. Certification of an optimal tsp tour through 85,900 cities. *Operations Research Letters*, 37(1):11–15, 2009.

[62] Gilbert Laporte and Yves Nobert. A Cutting Planes Algorithm for the m-Salesmen Problem. *The Journal of the Operational Research Society*, 31(11):1017–1023, 1980.

[63] P. Miliotis. Using Cutting Planes to Solve the Symmetric Travelling Salesman Problem. *Math. Program.*, 15(1):177–188, December 1978.

[64] Manfred Padberg and Giovanni Rinaldi. A Branch-and-Cut Algorithm for the Resolution of Large-Scale Symmetric Traveling Salesman Problems. *SIAM Review*, 33(1):60–100, 1991.

[65] Hipólito Hernández-Pérez and Juan-José Salazar-González. A branch-and-cut algorithm for a traveling salesman problem with pickup and delivery. *Discrete Applied Mathematics*, 145(1):126 – 139, 2004. Graph Optimization IV.

[66] O. Martin, S. Otto, and E. Felten. Large-step markov chains for the traveling salesman problem. *Complex Syst.*, 5, 1991.

[67] Daniel J. Rosenkrantz, Richard E. Stearns, and Philip M. Lewis. *An analysis of several heuristics for the traveling salesman problem*, pages 45–69. Springer Netherlands, Dordrecht, 2009.

[68] David S. Johnson and Lyle A. McGeoch. The Traveling Salesman Problem: A Case Study in Local Optimization. 2008.

[69] Michel Gendreau, Gilbert Laporte, and Frédéric Semet. A tabu search heuristic for the undirected selective travelling salesman problem. *European Journal of Operational Research*, 106(2):539 – 545, 1998.

[70] Miroslaw Malek, Mohan Guruswamy, Mihir Pandya, and Howard Owens. Serial and parallel simulated annealing and tabu search algorithms for the traveling salesman problem. *Annals of Operations Research*, 21(1):59–84, Dec 1989.

[71] Helena R. Lourenço, Olivier C. Martin, and Thomas Stützle. *Iterated Local Search*, pages 320–353. Springer US, Boston, MA, 2003.

[72] E.H.L. Aarts, J.H.M. Korst, and P.J.M. Laarhoven, van. A quantitative analysis of the simulated annealing algorithm: A case study for the traveling salesman problem. *Journal of Statistical Physics*, 50(1-2):187–206, 1988.

[73] P. Larrañaga, C. M. H. Kuijpers, R. H. Murga, I. Inza, and S. Dizdarevic. Genetic Algorithms for the Travelling Salesman Problem: A Review of Representations and Operators. *Artificial Intelligence Review*, 13(2):129–170, Apr 1999.

[74] Marco Dorigo and Luca Maria Gambardella. Ant colonies for the travelling salesman problem. *Biosystems*, 43(2):73 – 81, 1997.

[75] Kang-Ping Wang, Lan Huang, Chun-Guang Zhou, and Wei Pang. Particle swarm optimization for traveling salesman problem. In *Proceedings of the 2003 International Conference on Machine Learning and Cybernetics (IEEE Cat. No.03EX693)*, volume 3, pages 1583–1585 Vol.3, 2003.

[76] Bernard Angéniol, Gaël [de La Croix Vaubois], and Jean-Yves [Le Texier]. Self-organizing feature maps and the travelling salesman problem. *Neural Networks*, 1(4):289 – 293, 1988.

[77] F. Favata and R. Walker. A study of the application of Kohonen-type neural networks to the Travelling Salesman Problem. *Biological Cybernetics*, 64(6):463–468, Apr 1991.

[78] Y. Nagata and S. Kobayashi. A powerful genetic algorithm using edge assembly crossover for the traveling salesman problem. *INFORMS Journal on Computing*, 25(2):346–363, 2013.

[79] A. Möbius, B. Freisleben, P. Merz, and M. Schreiber. Combinatorial optimization by iterative partial transcription. *Phys. Rev. E*, 59:4667–4674, Apr 1999.

[80] Renato Tinós, Keld Helsgaun, and Darrell Whitley. Efficient Recombination in the Lin-Kernighan-Helsgaun Traveling Salesman Heuristic. In Anne Auger, Carlos M. Fonseca, Nuno Lourenço, Penousal Machado, Luís Paquete, and Darrell Whitley, editors, *Parallel Problem Solving from Nature – PPSN XV*, pages 95–107, Cham, 2018. Springer International Publishing.

[81] David E. Goldberg and Robert Lingle. Alleles Loci and the Traveling Salesman Problem. In *Proceedings of the 1st International Conference on Genetic Algorithms*, pages 154–159, USA, 1985. L. Erlbaum Associates Inc.

[82] John J. Grefenstette, Rajeev Gopal, Brian J. Rosmaita, and Dirk Van Gucht. Genetic Algorithms for the Traveling Salesman Problem. In *Proceedings of the 1st International Conference on Genetic Algorithms*, pages 160–168, USA, 1985. L. Erlbaum Associates Inc.

[83] L. Darrell Whitley, Timothy Starkweather, and D'Ann Fuquay. Scheduling Problems and Traveling Salesmen: The Genetic Edge Recombination Operator. In *Proceedings of the 3rd International Conference on Genetic Algorithms*, pages 133–140, San Francisco, CA, USA, 1989. Morgan Kaufmann Publishers Inc.

[84] Darrell Whitley, Timothy Starkweather, and Daniel Shaner. The Traveling Salesman and Sequence Scheduling: Quality Solutions Using Genetic Edge Recombination. In *In Handbook of Genetic Algorithms*, pages 350–372, 1990.

[85] Pascal Kerschke, Lars Kotthoff, Jakob Bossek, Holger H. Hoos, and Heike Trautmann. Leveraging TSP Solver Complementarity through Machine Learning. *Evolutionary Computation*, 26(4):597–620, 2018.

[86] Darrell Whitley, Doug Hains, and Adele Howe. Tunneling between Optima: Partition Crossover for the Traveling Salesman Problem. In *Proceedings of the 11th Annual Conference on Genetic and Evolutionary Computation*, GECCO '09, pages 915–922, New York, NY, USA, 2009. Association for Computing Machinery.

[87] Renato Tinós, Darrell Whitley, and Gabriela Ochoa. A New Generalized Partition Crossover for the Traveling Salesman Problem: Tunneling between Local Optima. *Evolutionary Computation*, 0(0):1–34, 0. PMID: 30900928.

[88] Renato Tinós, Darrell Whitley, and Gabriela Ochoa. Generalized Asymmetric partition crossover (GAPX) for the Asymmetric TSP. In *Proceedings of the 2014 Annual Conference*

on Genetic and Evolutionary Computation, GECCO '14, pages 501–508, New York, NY, USA, 2014. Association for Computing Machinery.

[89] Darrell Whitley, Doug Hains, and Adele Howe. A Hybrid Genetic Algorithm for the Traveling Salesman Problem Using Generalized Partition Crossover. In Robert Schaefer, Carlos Cotta, Joanna Kołodziej, and Günter Rudolph, editors, *Parallel Problem Solving from Nature, PPSN XI*, pages 566–575, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

[90] D. Sanches, D. Whitley, and R. Tinós. Building a better heuristic for the traveling salesman problem: combining edge assembly crossover and partition crossover. In *Proc. of GECCO'2017*, 2017.

[91] D. Sanches, D. Whitley, and R. Tinós. Improving an exact solver for the traveling salesman problem using partition crossover. In *Proc. of GECCO'2017*, 2017.

[92] Y. Nagata and S. Kobayashi. Edge Assembly Crossover: A high-power genetic algorithm for the travelling salesman problem. In Thomas Bäck, editor, *Proc. of the Seventh International Conference on Genetic Algorithms (ICGA97)*, 1997.

[93] K. Honda, Y. Nagata, and I. Ono. A parallel genetic algorithm with edge assembly crossover for 100,000-city scale TSPs. In *Proc. of the 2013 IEEE Congress on Evolutionary Computation*, pages 1278–1285, 2013.

[94] M.L. Fredman, D.S. Johnson, L.A. Mcgeoch, and G. Ostheimer. Data Structures for Traveling Salesmen. *Journal of Algorithms*, 18(3):432 – 479, 1995.

[95] R. Tinós and D. Whitley. A Fusion Mechanism for the Generalized Asymmetric Partition Crossover. In *2018 IEEE Congress on Evolutionary Computation (CEC)*, pages 1–8, 2018.

[96] Renato Tinós, Darrell Whitley, and Francisco Chicano. Partition Crossover for Pseudo-Boolean Optimization. In *Proceedings of the 2015 ACM Conference on Foundations of Genetic Algorithms XIII*, FOGA '15, pages 137–149, New York, NY, USA, 2015. Association for Computing Machinery.

[97] Wenxiang Chen, Darrell Whitley, Renato Tinós, and Francisco Chicano. Tunneling Between Plateaus: Improving on a State-of-the-art MAXSAT Solver Using Partition Crossover. In *Proceedings of the Genetic and Evolutionary Computation Conference*, GECCO '18, pages 921–928, New York, NY, USA, 2018. ACM.

[98] Francisco Chicano, Gabriela Ochoa, Darrell Whitley, and Renato Tinós. Enhancing Partition Crossover with Articulation Points Analysis. In *Proceedings of the Genetic and Evolutionary Computation Conference*, GECCO '18, pages 269–276, New York, NY, USA, 2018. Association for Computing Machinery.

[99] Francisco Chicano, Darrell Whitley, Gabriela Ochoa, and Renato Tinós. Optimizing One Million Variable NK Landscapes by Hybridizing Deterministic Recombination and Local Search. In *Proceedings of the Genetic and Evolutionary Computation Conference*, GECCO '17, pages 753–760, New York, NY, USA, 2017. Association for Computing Machinery.

[100] R. Tinós, K. Helsgaun, and D. Whitley. Efficient Recombination in the Lin-Kernighan-Helsgaun Traveling Salesman Heuristic. In *Proc. of PPSN XV*, pages 95–107. Springer, 2018.

[101] Molly A. O'Neil and Martin Burtscher. Rethinking the parallelization of random-restart hill climbing: A case study in optimizing a 2-opt tsp solver for gpu execution. In *Proceedings of the 8th Workshop on General Purpose Processing Using GPUs*, GPGPU-8, pages 99–108, New York, NY, USA, 2015. Association for Computing Machinery.

[102] D.Janaki Ram, T.H. Sreenivas, and K.Ganapathy Subramaniam. Parallel simulated annealing algorithms. *Journal of Parallel and Distributed Computing*, 37(2):207–212, 1996.

[103] Antón Rey, Manuel Prieto, J. I. Gómez, Christian Tenllado, and J. Ignacio Hidalgo. A cpu-gpu parallel ant colony optimization solver for the vehicle routing problem. In Kevin Sim and Paul Kaufmann, editors, *Applications of Evolutionary Computation*, pages 653–667, Cham, 2018. Springer International Publishing.

[104] K. Honda, Y. Nagata, and I. Ono. A parallel genetic algorithm with edge assembly crossover for 100,000-city scale TSPs. In *Proc. of the 2013 IEEE Congress on Evolutionary Computation*, pages 1278–1285, 2013.

[105] Enrique Alba, Antonio J. Nebro, and José M. Troya. Heterogeneous Computing and Parallel Genetic Algorithms. *Journal of Parallel and Distributed Computing*, 62(9):1362 – 1385, 2002.

[106] T. Starkweather, D. Whitley, and K. Mathias. Optimization using distributed genetic algorithms. In Hans-Paul Schwefel and Reinhard Männer, editors, *Parallel Problem Solving from Nature*, pages 176–185, Berlin, Heidelberg, 1991. Springer Berlin Heidelberg.

[107] L. Wang, A. A. Maciejewski, H. Siegel, V. Roychowdhury, and Bryce D. Eldridge. A study of five parallel approaches to a genetic algorithm for the traveling salesman problem. *Intell. Autom. Soft Comput.*, 11:217–234, 2005.

[108] L. Guo, C. Guo, D. B. Thomas, and W. Luk. Pipelined Genetic Propagation. In *2015 IEEE 23rd Annual International Symposium on Field-Programmable Custom Computing Machines*, pages 103–110, May 2015.

[109] L. Wang, A. A. Maciejewski, H. J. Siegel, and V. P. Roychowdhury. A comparative study of five parallel genetic algorithms using the traveling salesman problem. In *Proceedings of the First Merged International Parallel Processing Symposium and Symposium on Parallel and Distributed Processing*, pages 345–349, 1998.

[110] Gerhard Reinelt. *The Traveling Salesman: Computational Solutions for TSP Applications*. Springer-Verlag, Berlin, Heidelberg, 1994.

[111] D. Hains, D. Whitley, and A. Howe. Revisiting the big valley search space structure in the TSP. *Journal of the Oper. Res. Soc.*, 62(2):305–312, 2011.

[112] Thomas Stützle and Holger H. Hoos. *Analysing the Run-Time Behaviour of Iterated Local Search for the Travelling Salesman Problem*, pages 589–611. Springer US, Boston, MA, 2002.

[113] Zongxu Mu, Jérémie Dubois-Lacoste, Holger H. Hoos, and Thomas Stützle. On the empirical scaling of running time for finding optimal solutions to the TSP. *Journal of Heuristics*, 24(6):879–898, Dec 2018.

[114] Yiyuan Gong and Alex Fukunaga. Distributed island-model genetic algorithms using heterogeneous parameter settings. In *2011 IEEE Congress of Evolutionary Computation (CEC)*, pages 820–827, 2011.

[115] Hossein Rajabalipour Cheshmehgaz, Habibollah Haron, and Abdollah Sharifi. The review of multiple evolutionary searches and multi-objective evolutionary algorithms. *Artificial Intelligence Review*, 43(3):311–343, Mar 2015.

[116] Gerhard Reinelt. *The Traveling Salesman: Computational Solutions for TSP Applications*. Springer-Verlag, Berlin, Heidelberg, 1994.

[117] Jérémie Dubois-Lacoste, Holger H. Hoos, and Thomas Stützle. On the empirical scaling behaviour of state-of-the-art local search algorithms for the euclidean tsp. In *Proceedings of the 2015 Annual Conference on Genetic and Evolutionary Computation*, GECCO '15, pages 377–384, New York, NY, USA, 2015. Association for Computing Machinery.

[118] Ozeas Quevedo de Carvalho, Renato Tinós, Darrel Whitley, and Danilo Sipoli Sanches. A new method for identification of recombining components in the generalized partition crossover. In *2019 8th Brazilian Conference on Intelligent Systems (BRACIS)*, pages 36–41, 2019.

[119] Laxman Dhulipala, Changwan Hong, and Julian Shun. Connectit: A framework for static and incremental parallel graph connectivity algorithms. *Proc. VLDB Endow.*, 14(4):653–667, December 2020.

[120] Jayadharini Jaiganesh and Martin Burtscher. A high-performance connected components implementation for gpus. In *Proceedings of the 27th International Symposium on High-Performance Parallel and Distributed Computing*, HPDC ’18, pages 92–104, New York, NY, USA, 2018. Association for Computing Machinery.

[121] K.C. Tan, T.H. Lee, and E.F. Khor. Evolutionary algorithms with dynamic population size and local exploration for multiobjective optimization. *IEEE Transactions on Evolutionary Computation*, 5(6):565–588, 2001.

[122] Joseph Culberson and Joseph C. Culberson. Genetic Invariance: A New Paradigm for Genetic Algorithm Design. Technical report, University of Alberta, 1992.

[123] L. Darrell Whitley, Timothy Starkweather, and D’Ann Fuquay. Scheduling Problems and Traveling Salesmen: The Genetic Edge Recombination Operator. In *Proceedings of the 3rd International Conference on Genetic Algorithms*, pages 133–140, San Francisco, CA, USA, 1989. Morgan Kaufmann Publishers Inc.

[124] Martin Grötschel. *Polyedrische Charakterisierungen kombinatorischer Optimierungsprobleme*, volume 36. 1977.

[125] Martin Grötschel and Manfred W. Padberg. On the symmetric Travelling Salesman Problem: Theory and Computation. In Rudolf Henn, Bernhard Korte, and Werner Oettli, editors, *Optimization and Operations Research*, pages 105–115, Berlin, Heidelberg, 1978. Springer Berlin Heidelberg.

[126] M. Padberg and G. Rinaldi. Optimization of a 532-city symmetric traveling salesman problem by branch and cut. *Operations Research Letters*, 6(1):1 – 7, 1987.

[127] David L. Applegate, Robert E. Bixby, Vaŝek Chvatál, and William J. Cook. *Computation*, pages 489–530. Princeton University Press, 2006.

[128] Keld Helsgaun. General k-opt submoves for the Lin–Kernighan TSP heuristic. *Mathematical Programming Computation*, 1(2):119–163, Oct 2009.

Appendix A

Case Study on Singular Computing Machine

The Singular Computing machine (also called S1) [1] is a massively parallel machine with 34,000 cores arranged in a 4×4 grid of chips. Each chip, in turn, has about 2,112 cores arranged in a 48×44 two-dimensional grid. The peak performance of the machine is 8.5 TFLOPS, and the peak throughput is 8.5Tb/s. The key cool idea of this machine is that it uses approximate computing and thereby achieves less power and area consumption. This machine also belongs to the SIMD parallel architecture, making it ideal for deep learning and other sorts of compute-intensive applications. It can produce results as good as a modern GPU, but with $30 - 50\times$ better compute per watt. It uses a low-level programming language called the Nova.

In this case study, we were tasked to implement the GA based TSP solver on this massively parallel machine. From Chapter 2 we know that the EAX applies the local search operator 2-opt on the initial population. The EAX algorithm employs the edge-assembly crossover, and the best offspring replaces the parent in the next generation. The pair of parents to participate in the crossover operation is selected randomly. We also know that the parallelization of EAX is performed at the population level. That is, operations on every pair of individuals in the population are performed in parallel. But, there were three main challenges in this implementation, described in detail in the following sections.

A.1 Challenge 1: Memory costs

The recommended parallelization suffers severely due to the hardware limits on the S1 machine. This challenge is because each core in the machine has only 512 bytes of memory. This gives a total of 34.6 KB of total memory per machine. The EAX consumes about $100N$ ints per individual in the population, where N is the number of cities. Assuming a minimum of two

[1] http://www.singularcomputing.com/

individuals in our population, the maximum number of cities that we can implement on the S1 machine would be 2800. Therefore, the memory occupied per individual in the population is a severe bottleneck.

To overcome the memory constraints, we used a lightweight crossover operator called the Edge recombination (EX) [123] together with the 2opt local search operator. This entire setup occupies only $16N$ memory per individual in the population. Since the memory per core of the S1 machine is only 512 bytes, a group of 64 cores was combined to represent one genome. With this setting, for a population of 1024, we were able to accommodate a population of size 528. This setting is reasonable but yet had two other challenges.

A.2 Challenge 2: Communication costs

First, the 2opt and the EX crossover operator introduces new edges in the population. Therefore, there is a need to read data from the cost matrix. These reads to the cost matrix are irregular, and we called them as Random Access Reads (RAR). The S1 machine employs mesh architecture, and so, each core can communicate only with it's North, South, East, and West neighbors. The cost matrix is distributed across the core, and so, there is no central global memory where the RAR can fetch the data. There is a need for all the cores to participate in the RAR, and it involves communication across chips. One communication across a chip takes 180,000 cycles, and so poses a severe communication challenge.

To overcome the communication challenge, we employed several heuristic techniques. The RAR occurs in both the 2opt and EX algorithms. Listing 7 shows the steps involved in the EX algorithm. The algorithm starts by creating an edge table consisting of the neighbors of a given city. Thus, a city can have a minimum of two and a maximum of four neighbors. It then creates the child tour by randomly picking and chaining the edges from the table. If there is a broken chain, new edges are introduced in the population. This step corresponds to the seventh step.

To avoid the communication incurred during the seventh step, we examined four different versions of the EX algorithm as described below:

Algorithm 7 EX

1. Create EdgeTable
2. Randomly pick a city and add it to the child tour. Say this is the current city.
3. **for** (numOfCities)
4. Delete the current city from EdgeTable.
5. **if** (current city has a neighbour)
6. Pick one of current city's neighbour with least number of neigbours. If there is a tie, the first one is always picked.
7. **else**
8. Pick the unvisited city. This is done in lexicographic order of the city number.
9. **end for**

---

1. When a new edge is introduced, pick it from the nearest neighbor list (from the matrix) of the "current last city" of the child tour. If this fails, do RAR
2. if the above method fails, pick the new edge from the neighbor list (from the given tour) of the "first city" of the child. Here, we will reverse the tour once. If this fails, do RAR
3. if the above two fails, pick the new edge from the nearest neighbor list (from the matrix) of the "first city" of the child tour. Here, we will reverse the tour once. If this fails, do RAR
4. Let us maintain a boolean array (checkNear) of size N: checkNear[i] is True: If all the near M neighbors of the city i is picked in the tour. If version 3 fails, Pick the vertex already in the child tour and whose checkNear is false. Reverse the tour between the *currentLastCity* and this vertex.

These steps require us to maintain M nearest neighbor matrix and eliminate the $\mathcal{O}(N^2)$ cost matrix. Figure A.1 shows the reduction in RAR count on a 1000-city instance. Here, version 0 is the standard EX algorithm. It can be seen that RAR must be performed for versions 1, 2, and 3.

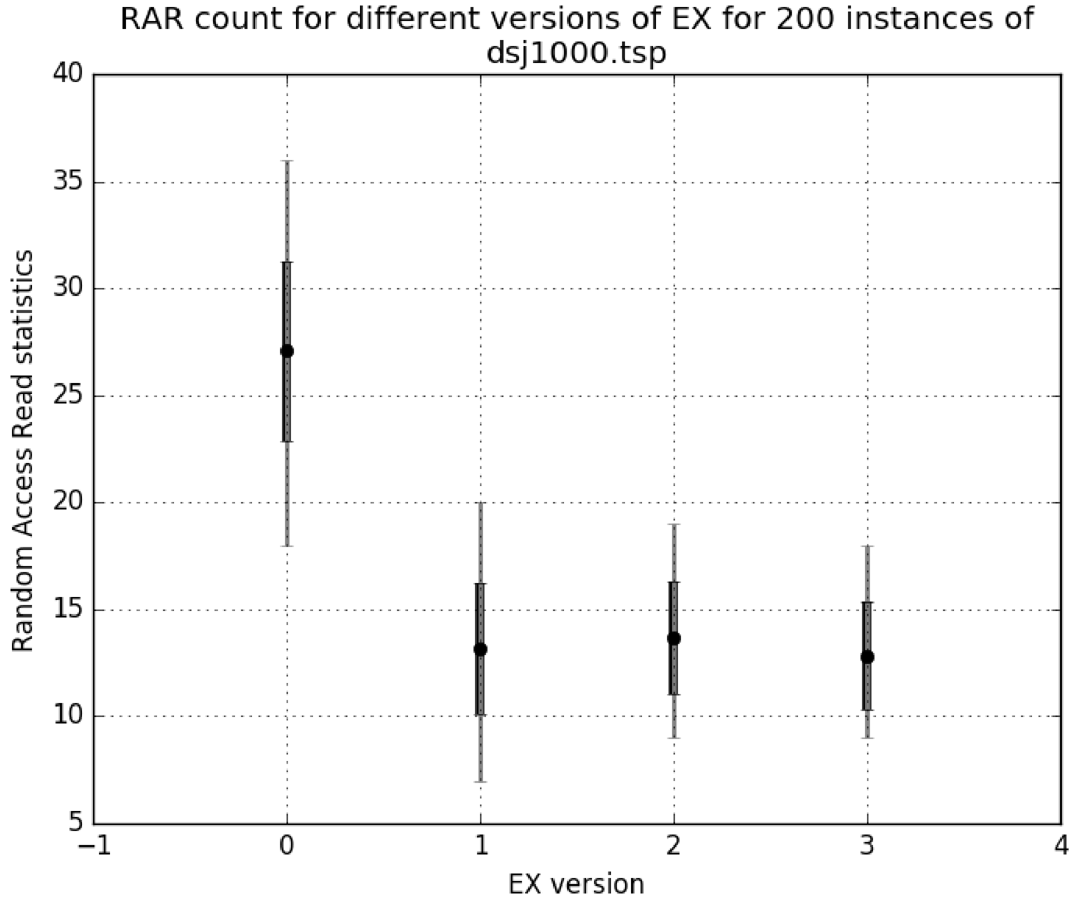


Figure A.1: Random Access Reads (RAR) count for different EX versions on dsj1000 city problem collected over 200 individual runs

The fourth version relies only on the neighbor matrix for the new edges. Since we have about $M * N$ edges in the neighbor matrix, this guarantees that there is no RAR with this version. But, this version of EX increases the computational cost of the algorithm. Because of these reasons, we developed a new heuristic to eliminate RAR. In the version-1 of EX, if you must pick a random edge, assign it an estimated cost twice as the largest edge in the tour. It is large enough that performing 2-opt on the resultant tour should remove it. This heuristic ensures that the communication cost is restricted only to the neighbor matrix. The computation cost is $O(NM)$.

The RAR in 2opt occurs at every swap 2. We make sure that this RAR is also read-only from the nearest neighbor list. Each chip in the S1 machine is loaded with the MN neighbor matrix. Thus communication across chips is avoided in this way. Also, only the first $M/2$ entries in each

row of the neighbor matrix are similar across the chips. The second $m/2$ entries are randomly selected to maintain the diversity in the population. In the case of asymmetric TSP instances, both the forward and reverse edges are stored. Thus, increasing the memory costs by MN. Overall, these heuristic methods helped reduce the communication costs of the system.

A.3 Challenge 3: Accuracy of Solution

Now, the proposed GA starts with a random initial population, followed by the 2opt local search and the EX crossover operator. At the end of each generation, the 2opt and EX are iteratively applied until the global optima is reached. The population is not randomized across the chips. But, they are migrated within the chip. The number of nearest neighbors chosen was 50. With these settings, the proposed algorithm reaches up to 2$\%$ optimal solution on the att532 problem in 7 minutes using a population size of 528. In contrast, EAX can reach global optima in 11 seconds. Furthermore, the proposed algorithm on the S1 machine takes more than a day to reach the global optima, which is not desired.

This study showed that implementing GA based TSP solvers on massively parallel machines has several challenges, such as (a) Memory per individual in the population (b) Communication costs and patterns (c) Quality of solution. Therefore, to design an efficient TSP solver on massively parallel machines with limited resources, there is a need to modify the existing algorithms.

Appendix B

Dataset and Codes

In this appendix, references to the dataset and codes used in this thesis are discussed.

B.1 TSP File format and calculations

The TSP tours are stored in TSPLIB [110] format. Each file consists of a specification part and of a data part. The specification part contains information on the file format and on its contents. The data part contains explicit data. The files are stored with a .tsp extension.

Figure B.1 gives an example of a TSP instance. The number in the file name denotes the number of cities. As seen in the figure B.1, the first six lines contains information of the file. The remaining 280 lines correspond to the 2-dimensional eucledian coordinates of each of the cities. Finally the keyword *EOF* is used to end the file.

The TSP tours are randomly initialized. The tour distance calculations differ depending on the "edge weight type." The dataset considered in this thesis fall under four different types, namely, EUC_2D, EUC_3D, CEIL_2D, and ATT. Suppose, (x,y,z) are the co-ordinates of two points (say, i and j), the corresponding distance calculations are shown in figure B.2.

B.2 Dataset and references

The references to the five TSP instances in Figure 1.1 from Chapter 1 are,

1. gr120 - It is a 2-dimensional TSPLIB problem [1], solved to optimality by Grötschel and Padberg [124, 125] in 1978.

2. pr1002 - It is a 2-dimensional TSPLIB problem, solved to optimality by Padberg and Rinaldi [126] using the LP relaxation of the problem in 1987.

[1] http://www.math.uwaterloo.ca/tsp/history/tspinfo/gr120_info.html

```
1  NAME : a280
2  COMMENT : drilling problem (Ludwig)
3  TYPE : TSP
4  DIMENSION: 280
5  EDGE_WEIGHT_TYPE : EUC_2D
6  NODE_COORD_SECTION
7     1 288 149
8     2 288 129
9     .....
10    280 280 133
11 EOF
```

Figure B.1: Example of a TSP file format

```
1 double t1 = x[i] - x[j], t2 = y[i] - y[j], t3 = z[i] - z[j];
2 EUC_2D: D = round(sqrt(pow(t1,2) + pow(t2,2)));
3 EUC_3D: D = (int) (sqrt(t1 * t1 + t2 * t2 + t3 * t3) + 0.5);
4 CEIL_2D: D = (int)ceil(sqrt(t1 * t1 + t2 * t2));
5 ATT: D = (int)(sqrt((t1 * t1 + t2 * t2)/10.0));
```

Figure B.2: Calculation of TSP edge distances for various input formats

3. usa13,500 - It is a 2-dimensional USA instance, 13,500 cities having a population of 500 or more. Concorde [127] found the optimal solution in May 1998 on a network of Digital Alphaservers, Intel Pentium II workstations, and Sun Microsystems Sparc workstateions at Rice University.

4. 1,904,711 - It is a 2-dimensional World TSP instance [2], created in 2001. The best tour with a 0.0058% close to HK bound was found by the LKH solver. However, recently, in February 15, 2021, a new best solution is found by Helsgaun starting with a tour found by Xavier Clarist on February 11, 2021.

5. E10,000,000 - It is a 2-dimensional *E-instance*, the largest TSP problem in the 8th DIMACS TSP Challenge [3]. The *E-instances* consists of points uniformly distributed in the 1,000,000 by 1,000,000 square under the Euclidean metric. The optimal solution is unknown. The LKH solver found the best solution[128], 0.588 close to the HK bound.

6. 1,331,906,450 - It is a 3-dimensional Star TSP instance, the largest TSP problem in the world (as of 2021). The distances are measured in parsec. The description of the entire project is in the Star TSP website [4]. The LKH solver is used to find the best known solution, 0.0038 close to the HK bound.

The rest of the dataset used in this thesis comes from DIMACS TSP Challenge [5], Star TSP instances [6], TSPLIB [110] and ART TSP instances [7].

---

[2]https://www.math.uwaterloo.ca/tsp/world/

[3]http://dimacs.rutgers.edu/archive/Challenges/TSP/index.html

[4]http://www.math.uwaterloo.ca/tsp/star/gaia2.html

[5]http://archive.dimacs.rutgers.edu/Challenges/TSP/

[6]http://www.math.uwaterloo.ca/tsp/data/art/

[7]http://www.math.uwaterloo.ca/tsp/star/

B.3 Software codes

The software codes developed in this thesis are uploaded onto the Github private repository. The codes are available upon request. Please email swesri91@gmail.com. I hope to release the codes along with the journal articles.