J Supercomput (2016) 72:4399–4414
DOI 10.1007/s11227-016-1748-1

GPU-based parallel genetic approach to large-scale travelling salesman problem

Semin Kang[1] · Sung-Soo Kim[2] · Jongho Won[2] · Young-Min Kang[3]

Published online: 23 May 2016

Abstract The travelling salesman problem (TSP) is a well-known NP-hard problem. It is difficult to efficiently find the solution of TSP even with the large number of gene instances. Evolutionary approaches such as genetic algorithm have been widely applied to explore the huge search space of TSP. However, the feasibility constraints of TSP make it difficult to devise an effective crossover method. In this paper, we propose an improved constructive crossover for TSP. As the performance of graphics processing units (GPUs) rapidly improves, GPU-based acceleration is increasingly required for complex computation problems. Unfortunately, the constructive crossover methods cannot be easily implemented in a parallel fashion because each gene element of offspring is dependent on the previous element in the gene string. In this paper, we propose a more effective method with which large number of genes can evolve effectively by exploiting the parallel computing power of GPUs and an effective parallel approach to genetic TSP where crossover methods cannot be easily implemented in parallel fashion.

Keywords Travelling salesman problem · Genetic algorithms · Constructive crossover · GPU parallelism

This work was supported in part by ETRI R&D Program (Development of Big Data Platform for Dual Mode Batch-Query Analytics, 16ZS1410) and in part by 2016 KISTI PLSI Program.

✉ Young-Min Kang
ymkang@tu.ac.kr

1 Department of Computer Media Engineering, Tongmyong University, Busan, Korea

2 Electronics and Telecommunications Research Institute, Daejeon, Korea

3 Department of Game Engineering, Tongmyong University, Busan, Korea

1 Introduction

The travelling salesman problem (TSP) is a well-known optimization problem where the objective is to find the lowest cost of a tour path that passes all the nodes exactly once and returns back to the starting node. Although it seems simple, this optimization problem is known as 'NP-complete'. In other words, TSP has a huge search space, and it can be impossible to find the optimal solution [14,16]. Therefore, various kinds of genetic algorithms have been employed to solve this problem [6,11].

Evolutionary method is a heuristic search algorithm based on the rules of evolution like natural selection and natural genetics [4,10]. Briefly described, genetic algorithms employ selection, crossover, and mutation to improve the solutions which are expressed as genes. In this process, the crossover plays the most important role in producing better genes out of the current gene pool. However, the traditional crossover methods cannot be used for evolutionary approaches to TSP because simple exchange of substrings of genes easily violates the feasibility constraints. To avoid this problem, researchers have proposed various crossover methods which always generate feasible offsprings [5,9,17].

Among those crossover methods, 'sequential constructive crossover (SCX)' showed the best convergence compared to previous methods [1]. There are a few disadvantages when SCX is employed. First, the SCX searches only in one direction and does not take into account the circular properties of TSP tours. Another disadvantage of SCX is that the offsprings are likely to be similar to the better solution between two parents. This is inevitable because the construction process of SCX is greedy. The greedy aspect of SCX makes the gene pool rapidly converge to local minima and reduce the diversity in the gene pool. To improve the performance of the constructive crossover, bidirectional circular SCX (BCSCX) was proposed [15]. Although BCSCX improves the convergence speed, it still suffers from rapid assimilation of genes to a certain local minima.

In this paper, we propose an improved crossover that maintains the diversity of genes. This method produces offsprings which equally inherit from two parents. Since, the parents alternatingly play roles in determining the city sequence of offsprings, the method is named 'alternating recommendation crossover (ARX)'.

Evolutionary methods like the ones above search better when the gene population is large and genes are diverse. Therefore, it is necessary to use a large amount of genes [8]. This means that the necessary computation tasks also must increase along with the size and number of genes. This is a typical 'data parallelism', and GPUs are suitable for such problems. However, the constructive crossovers such as SCX, BCSCX, and ARX cannot be easily performed in a parallel fashion. In this paper, we propose efficient parallel computing techniques for constructive crossovers to solve large-scale TSPs in an efficient way.

2 Evolutionary approach to the travelling salesman problem

In this section, the difficulties in designing crossover methods for genetic approaches to TSP will be explained. Furthermore, we will introduce crossover methods suitable

for TSP. The proposed methods satisfy the constraints of TSP and unconditionally produce feasible offsprings. The basic idea is to construct a feasible offspring from the sequences of parent genes. Although the proposed method successfully generates feasible offspring, it can produce only one offspring with two parents, and the property of the offspring gene tends to depend more on one parent with better fitness. To avoid such limitations, we also propose another crossover method which can produce two offsprings and the offsprings are evenly influenced by two parents.

2.1 Gene representation and crossover for feasible offsprings

A TSP solver based on genetic algorithm requires a proper gene representation for feasible solutions. The sequence of cities in accordance with a feasible tour can be used as a gene representation.

The solutions must allow all nodes to be visited only once and there must be no missing nodes. In conventional crossover methods, an offspring is generated by exchanging some parts of parent's chromosomes. However, such a conventional crossover may easily produce duplicate nodes and missing nodes.

Crossover operators such as 'edge recombination crossover (ERX)', 'generalized n-point crossover (GNX)' and 'sequential constructive crossover (SCX)' have been proposed to guarantee the feasibility of offsprings, SCX showed better fitness convergence than other methods. SCX crossover method guarantees the validity of offsprings' chromosomes and conserves the merits of parents. This method tries to reduce the local distance between the adjacent nodes in the offspring by sequentially scanning the chromosomes of parents. The algorithm can be described as follows [1]:

1. Set the starting node 0 to be the current city p.
2. Find the two unvisited node a and b, respectively, from the chromosome of each parent by sequentially searching the first unvisited nodes (legitimate nodes) after the current city p. If the search fails, select any unvisited node from the city permutation template such as $\langle 1, 2, \ldots, n \rangle$.
3. Compare the distances from p to a (d_{pa}) and to b (d_{pb}). If d_{pa} is less than d_{pb}, add a to the offspring chromosome and set a to be the current node. Otherwise, b is added and set to be the current node. Then go back to step 2.

2.2 Improved SCX with bidirectional and circular search

To improve the performance of SCX, bidirectional circular SCX (BCSCX) was proposed. This method can search the next possible 'legitimate' nodes in the chromosomes of parents in two directions and the chromosome is regarded as circular data with no ends [15].

The BCSCX operator searches legitimate nodes in two directions. In other words, it chooses the two candidate nodes to be added to the chromosomes of offspring both before and after the currently visited node from chromosomes of each parent.

For example, assume that an uncompleted chromosome sequence $\langle 1, 5 \rangle$ has been inherited from two parents during the crossover operation. Then the current node is

city 5. If, within the chromosome sequence of one parent, city 3 is the closest unvisited node ('closest' in the aspect of the location in the sequence string) among the nodes after the current node (city 5), and city 2 is the closest unvisited node before the current node in the sequence of one parent. The proposed BCSCX takes both nodes (cities 3 and 2) as 'legitimate candidate nodes.' Two more candidates are taken from the other parent in the same way. Let us assume that the candidates from the other parent are city 2 and city 6. The candidates for the next node in the offspring's chromosome right after city 5 are the union of the candidate sets (i.e. cities 2, 3, and 6), and they are tested similarly as SCX. In this method, offspring is constructed in a greedy way so that the offspring is very similar to the better gene when the gap between the fitness values of the parents is huge.

SCX does not assume that chromosome strings are circularly concatenated. Therefore, if the last node of the chromosome string is the current node p, no candidate legitimate nodes can be obtained. To avoid this problem, SCX employed a pre-defined template.

For example, assume that the chromosome of a parent is $\langle 1, 3, 7, 6, 2, 4, 5\rangle$ and that of the other is $\langle 1, 5, 7, 2, 6, 3, 4\rangle$, and currently constructed partial chromosome of the offspring is $\langle 1, 5\rangle$. The original SCX then fails to find the legitimate node from the first parent, and the first unvisited node (in this case, city 2) from the template $\langle 1, 2, 3, 4, 5, 6, 7\rangle$ will be selected as the legitimate node. However, our method regards the chromosomes as circular data, and jumps to the first character so that city 3 will be selected as the legitimate node. It seems like a matter of course that BCSCX converges better than SCX because the tour routes are circular by nature.

2.3 Efficient search for legitimate nodes

Constructive crossovers such as SCX construct the chromosomes of offsprings by selecting the best node from the 'legitimate' candidates, and the feasibility of the offspring chromosome is guaranteed by the 'legitimacy' of the candidate nodes. However, the performance of the crossover largely depends on how to search the legitimate nodes. Moreover, the method proposed in this paper searches the legitimate nodes in two directions, and the performance of this search process affects the overall performance of the system.

The overall performance of an evolutionary TSP solver depends on three major factors: (1) k, the number of iterations needed for convergence to a reasonable solution, (2) m, the population of genes, and (3) n, the number of cities. If we denote the cost of the legitimate node search for constructing a child chromosome as $search(n)$, the performance of the system can be expressed as $O(km \cdot search(n))$. When a naïve approach such as sequential search is applied, it is obvious that $search(n)$ is $O(n^2)$ and the overall performance will be $O(kmn^2)$.

Since the genes converge to similar genes more as the number of iterations increases, the backward search used in the method proposed in this paper inevitably requires $O(n^2)$ searches for the construction of one chromosome.

To resolve this problem, we employed forward and backward jump indices. The jump indices can be described as in Fig. 1. In this figure, the visited nodes are shaded

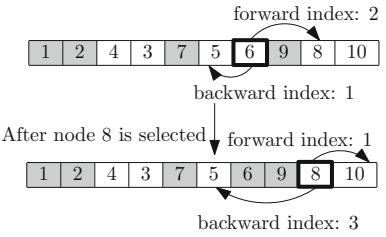


Fig. 1 Example of jump indices and their modification

Table 1 The initial setting of parent chromosome before bidirectional constructive crossover

i	0	1	2	3	4	$\cdots$	$n-1$
χ_i	1	Arbitrary city permutation with $(2, 3, \ldots, n)$					
f_i^{χ}	1	1	1	1	1	$\cdots$	2
b_i^{χ}	1	2	1	1	1	$\cdots$	1
Visited	Yes	No	No	No	No	$\cdots$	No

and the most recently visited node has a thick border. The legitimate nodes are searched from this node, and the indices make it possible to search them in $O(1)$ time.

If we consider the first chromosome state, the first five cities of offspring chromosomes have been determined, and the last city is 6. The candidates for the next city are city 8 and city 5. Therefore, the forward jump index is 2, and the backward one is 1. If the city 8 is selected as the next city, the jump indices have to be updated. The forward and backward indices of city 8, for example, become 1 and 3.

Because the solution routes of TSP must return to the starting node, we assumed all the feasible chromosomes start from city 1. Let us denote the chromosome satisfying this constraints as χ, and the ith city in the chromosome as χ_i. The forward and backward jump indices to find legitimate nodes from χ_i are denoted f_i^{χ} and b_i^{χ}, respectively. The information in the chromosomes of each parent before the construction of offspring chromosomes can be initialized as shown in Table 1.

Based on the circular property of the TSP solution, the indices restart from 0 when they become larger than $n-1$ (i.e. $i \bmod n$). Similarly, the indices come down from $n-1$ when they become less than 0. Let us denote index i satisfying this constraint as $\langle i \rangle_n$. The node 0 (i.e. χ_0) is always 1. The forward and backward indices of all nodes are initialized as 1 except for the forward one of node $n-1$, and backward one of node 1 because χ_0 will be automatically inherited to an offspring's chromosomes and regarded as already visited node. Because BCSCX takes four possible legitimate nodes from two parent chromosomes, there is no guarantee that χ_1 or χ_{n-1} will be selected as the next node. If a node χ_i is selected as the next visiting city, only two indices $f^{\chi}_{\langle i-b_i^{\chi}\rangle_n}$ and $b^{\chi}_{\langle i+f_i^{\chi}\rangle_n}$ in the chromosome must be updated. The update can be done as follows:

$$\begin{aligned} f^{\chi}_{\langle i-b_i^{\chi}\rangle_n} &\leftarrow f^{\chi}_{\langle i-b_i^{\chi}\rangle_n} + f_i^{\chi} \\ b^{\chi}_{\langle i+f_i^{\chi}\rangle_n} &\leftarrow b^{\chi}_{\langle i+f_i^{\chi}\rangle_n} + b_i^{\chi} \end{aligned} \tag{1}$$

With the assistance of the indices managed as shown in Eq. 1, the total search for legitimate nodes during the whole construction of offspring chromosomes can be done in $O(n)$. Therefore, the overall system performance is $O(kmn)$, and we have only to improve the convergence speed to reduce k when devising a better constructive crossover method.

2.4 Alternating recommendation crossover: ARX

The most important drawback of SCX and BCSCX is that an offspring is biased to a better parent. This is inevitable because of the greedy aspect of SCX and BCSCX in the construction of offspring genes by selecting the best unvisited node at every step. The greedy property makes the gene pool rapidly lose its diversity, and the evolution process easily becomes stagnant at local minima.

To avoid this problem, we propose a method that maintains the diversity of the genes by alleviating the greediness of the aforementioned crossover method and making offsprings inherit from parents with the equivalent importance. The proposed crossover was named alternating recommendation crossover (ARX) because the parents alternatingly take privilege to determine the gene sequence of offsprings. Each parent can recommend the legitimate nodes as in the BCSCX only when it is given the privilege. Therefore, the parents equally play roles in constructing the offspring gene sequence.

The actual crossover method is shown in Fig. 2. Two parents are denoted α and β, and two points where the recommendation privilege is switched are determined. The points are randomly selected for the gene pool to experience various biases to parents in the crossover process. The crossover produces two offsprings (child α and child β). In the construction of one offspring, parent α recommends the legitimate nodes until the first crossover point A, and parent β takes over the privilege after that point. At the next crossover point marked as B, the privilege is again switched to parent α. In the construction of child β, the recommendation is performed in the reverse manner. As shown in the figure, ARX can produce two different offsprings while SCX and BCSCX can produce only one offspring at each crossover. The substrings recommended by parent α and β are denoted R^α and R^β, respectively.

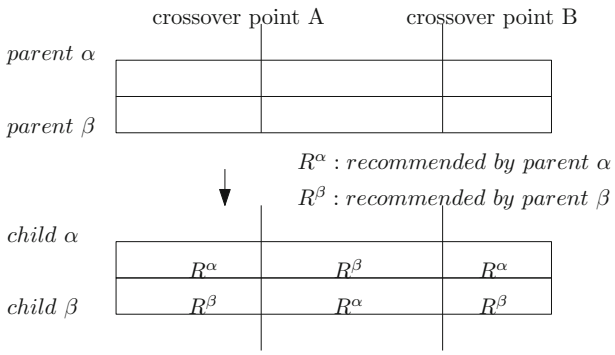


Fig. 2 Offspring construction of ARX

ARX can maintain the diversity of genes compared to SCX and BCSCX. The fitness improvement of ARX is not as rapid when compared to other constructive crossover methods because it equally takes into account the features of less fit parents. However, the diversity of the gene pool enables the genetic evolution to escape from the local minima more easily than the previous constructive crossover methods, and to find the better solution in the long term.

3 GPU acceleration

Graphics processing units (GPUs) were initially devised to transform a large amount of geometric data. Therefore, GPUs are parallel processors that perform simple and similar tasks to large data, and they are suitable for various 'data parallel' problems [7]. As the performance improvement of GPUs outperform traditional CPUs, various computation problems that are suitable to be processed by SIMD (single instruction multiple data) algorithms, such as molecular dynamics are being successfully accelerated with GPUs [2,3,18]. Moreover, GPU resources are relatively cheaper than CPUs so that more and more high-performance computing problems are accepting GPU-based parallel computing [19].

However, there are some limitations in GPU computing. First, GPUs do not allow dynamic memory allocation during tasks. As a consequence, the necessary memory for a task should be determined in advance. Moreover, memory lock is not efficiently supported. One of the most important limitations is that the communication between CPUs and GPUs is still too expensive. Therefore, the CPU–GPU communication becomes the bottleneck of the overall computation [13].

Despite the limitations, the many-core parallel processing is useful for problems dealing with large amounts of data. In this section, we present our methods to exploit parallel processing ability and to overcome limitations to implement an efficient genetic algorithm for large-scale travelling salesman problems.

3.1 Parallel computation of fitness values of genes

A single gene describes a tour visiting all the cities. The fitness of the gene is the sum of distances between every adjacent city pairs in the gene sequence. Let us denote the city in the ith place of the sequence by c_i, and the distance between two cities c_i and c_j by $d(c_i, c_j)$. φ^x, the fitness of a gene x, can be computed as follows:

$$\varphi^x = 1/\sum_{i=1}^{n} d(c_i, c_{(i+1)\mathbf{mod}n}) \tag{2}$$

It seems that we can create n threads of which index τ ranges from 1 to n to perform a task using parallel computation $d(c_\tau, c_{(\tau+1)\mathbf{mod}n})$. However, each thread τ accumulates its computation result to the same variable φ^x which should be synchronized. The synchronization nullifies efficiency obtained by the parallel processing. To solve this problem, we create threads in accordance with the number of genes. Suppose we

have m genes, the range of τ is then $[1, m]$. In the ith execution of n-sized loop, each thread τ performs $d(c_i, c_{(i+1) \bmod n})$ for τth gene, and adds the result to φ^τ. The fitness values then can be computed in an asynchronous way.

3.2 Efficient competition

Evolution is achieved by selecting better genes to produce offsprings. Each gene competes with others to survive. To implement an efficient genetic method, it is important to know which two genes are selected and compared to one another. Random selection of competing pairs in GPU implementation is not recommended because GPU does not efficiently support memory lock.

To make genes compete with each other without any need for synchronization, each gene has its fixed rival in accordance with its location in the gene pool. After the competition, the location of the winner is also fixed to not request synchronization.

Suppose we have m genes in the pool, each gene can be indexed by the integer i in the range of $[0, m-1]$. For a gene j ranging from 0 to $\lfloor m/2 \rfloor$, gene $j + \lceil m/2 \rceil$ is coupled to be compared, and the winner gene is stored at the location j. This competition process does not require any synchronization and can be efficiently performed in a parallel fashion.

3.3 Efficient parallel crossovers

Crossover methods also cannot be performed by random selection of gene pairs because of the same reason mentioned in the previous subsection describing the implementation of parallel competition. Moreover, the constructive crossover cannot be easily implemented with parallel tasks because the feasibility constraints make it impossible to independently determine each gene element in a single gene. Traditional crossover where some corresponding subsequences of two parent genes are simply exchanged and other such tasks can be performed in parallel without any difficulties. However, the constructive crossover methods can determine the kth element in the offspring gene sequence only after the $k-1$th element has been already determined. Therefore, efficient parallel crossover has two requirements: (1) asynchronous crossover pairing and offspring reproduction, and (2) parallel implementation of crossover.

For parents to be asynchronously coupled, we also fix the partner for each gene. During the competition process, we have $m/2$ elite genes in the first half of the gene pool as shown in the Fig. 4. For each elite gene which has even-number index i ranging from 0 to $\lfloor m/2 \rfloor$, the next gene $i+1$ is coupled to produce an offspring . Their offspring is computed and stored at the location $i/2 + \lceil m/2 \rceil$ as shown in Fig. 3. BCSCX produces only one offspring from two parents. Therefore, only $m/4$ offsprings are produced as shown in Fig. 4. However, ARX can produce two offsprings from two parents. Therefore, one of the offsprings is stored at $i/2 + \lceil m/2 \rceil$, and the other is stored at $i/2 + \lceil 3m/4 \rceil$.

As mentioned, the constructive crossover cannot be parallelly implemented. To solve this problem, we create threads in accordance with the number of genes m and

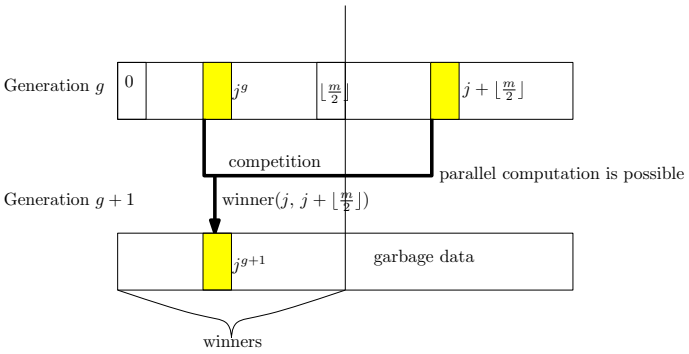


Fig. 3 Offspring construction of ARX

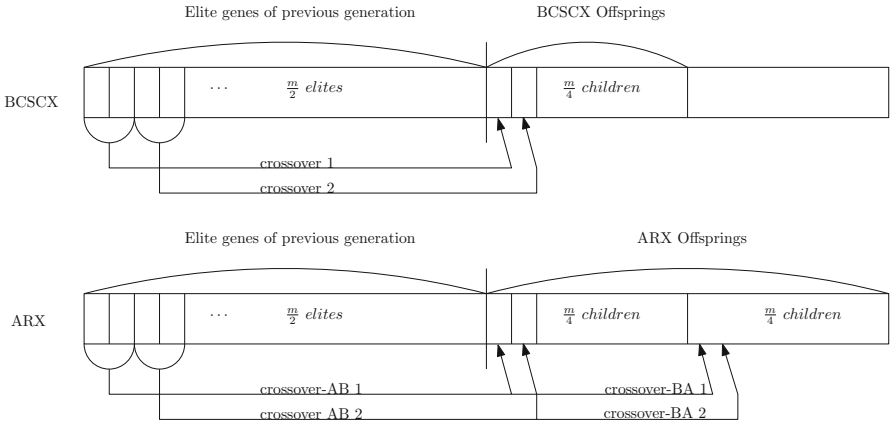


Fig. 4 Parallel crossover without synchronization

each thread independently determines only one element of its offspring gene sequence. In other words, a parallel task is denoted by $\xi(k, p_1, p_2, c)$ where ξ is a crossover task, k is the location in the offspring sequence to be determined, p_1 and p_2 are the gene indices of parents, and c is the location for the offspring in the gene pool. Therefore, the parallel crossover can be performed as shown in Algorithm 1.

Algorithm 1: Thread execution for parallel crossover

Data: n: number of cities, m: number of genes
begin
  for *each index* $(k: 1 \cdots n)$ **do**
    for *each thread*$(\tau: 1 \cdots \lfloor m/2 \rfloor)$ **do**
      do $\xi(k, \tau, \tau + 1, \tau + \lceil m/2 \rceil)$

3.4 Intergroup gene exchange

The effect of parallel crossover becomes significant when the size of the gene population is large enough in relation to the number of cities. However, simply increasing the population size does not accelerate convergence even though it results in proportional increase of computational burden. The stagnation in fitness improvement despite the increased number of genes is because of the rapid decline in gene diversity. To maintain the diversity, we divided the gene pool into several groups and genes competing with each other and produce offsprings within the specific groups they belong to.

To accelerate the convergence, the best gene of each group is periodically transferred to another group. The groups of genes which are interchanged are randomly selected. The intergroup gene exchange can be easily implemented with n threads that transfer only one element assigned to them. The communication between groups must slightly increase the computational burden. However, the fitness convergence will be accelerated by the diversity of genes.

4 Experiments

The experiments were performed on a system running on Ubuntu Linux OS with Intel Xeon 3.25 GHz CPU and NVIDIA GTX 980 GPU, and the test data were obtained from TSP data site maintained by University of Waterloo, and they are found at [20]. The project is available in a public repository (https://github.com/dknife/Proj15A_TSP).

The convergence trends of BCSCX and ARX were measured as shown in Fig. 5. In this experiment, test data contain 237 cities (xqg237), and 2048 genes were randomly generated. After 150 generations, each crossover method plotted the convergence trends shown in the figure. Although every run of genetic approach shows different convergence, they were not very different from those shown in the figure. Therefore, the graphs shown can be regarded as typical convergence trends of BCSCX and ARX. The horizontal axis represents the number of generations, and the vertical axis the error ϵ of the found solution with the cost c_b compared to the known optimal solution with the cost of c_o. The error was measured as follows:

$$\epsilon = (c_b - c_o)/c_o \tag{3}$$

As shown in Fig. 5, ARX shows slower convergence than BCSCX because BCSCX greedily selects the best one among the four candidates recommended by two parents. Although, the greedy approach rapidly converges in the early stage of the evolution, it does not produce the better solution in the long term. The crossover based on this strategy makes the gene pool homogeneous, and no more effective search will be performed after a local minima is found. This can be easily observed by checking the vertical lines under the graphs. The vertical lines are drawn when the gene pool finds a gene which is better than any other genes produced in the previous generations. After the BCSCX quickly finds a gene with little fitness, the record breaking search becomes infrequent. However, ARX, although slow, continuously updates the fitness record. For

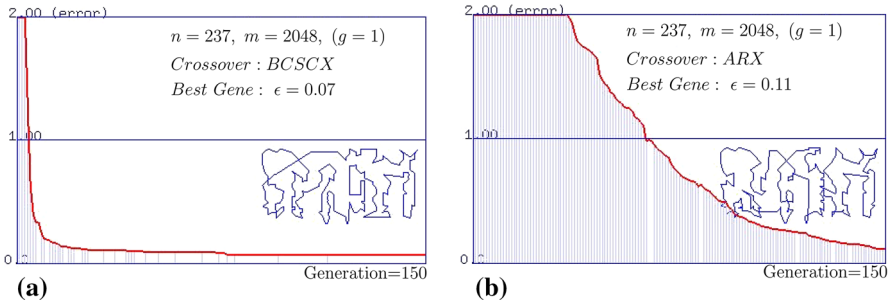


Fig. 5 Convergence trends (150 generations) of **a** BCSCX ($\epsilon = 0.07$) **b** ARX ($\epsilon = 0.11$)

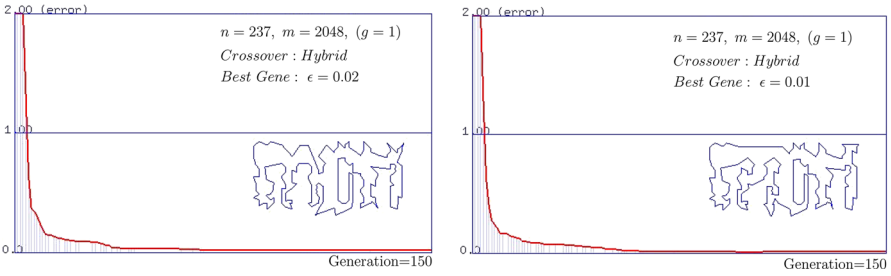


Fig. 6 Convergence trends (150 generations) of hybrid crossover

Table 2 Fitness convergence comparison

	131 cities (xqf131)		662 cities (xql662)	
	Gen. to reach $\epsilon = 0.1$	ϵ after 100 gen.	Gen. to reach $\epsilon = 0.1$	ϵ after 100 gen.
BCSCX	35.54	0.041	71.21	0.057
ARX	67.45	0.008	532.42	0.058
hybrid	26.40	0.019	54.23	0.021

mutation, we randomly selected a small fraction of genes and also randomly chose edge pairs in each selected gene. The selected edge pair $e_1(v_1, v_2)$ and $e_2(v_3, v_4)$ was then modified as $e'_1(v_1, v_3)$ and $e'_2(v_2, v_4)$ and the subsequence $v_2 \cdots v_3$ is reversed to maintain the feasibility.

To exploit the advantages of BCSCX and ARX, we implemented a hybrid method that uses both crossover methods. In the hybrid method, half of the offsprings are generated with BCSCX and the rest of them are generated with ARX. Figure 6 shows the convergence of the hybrid method with the same data used in Fig. 5. The hybrid method converged more rapidly than BCSCX by including the slower crossover ARX. This shows the significance of the gene diversity in evolutionary methods. The fitness values after 150 generations were 0.021 in average.

Table 2 compares the convergence of crossover methods with 131-city data (xqf131), and 662-city data (xql662). 2048 genes were used. To compare the initial convergence speed, we measured the number of generations required until ϵ is less than 0.1. Each case was tested 100 times and the average value is shown in the

Table 3 150-generation evolution of 128 local minima genes (662 cities, initial $\epsilon = 0.0915$)

Run	1	2	3	4	5	6	7	8	9	10	Avg.
BCSCX	0.047	0.049	0.057	0.056	0.051	0.066	0.038	0.053	0.049	0.053	0.052
ARX	0.025	0.020	0.020	0.022	0.025	0.025	0.025	0.026	0.026	0.022	0.024
Hybrid	0.045	0.039	0.048	0.046	0.022	0.044	0.051	0.053	0.042	0.041	0.043

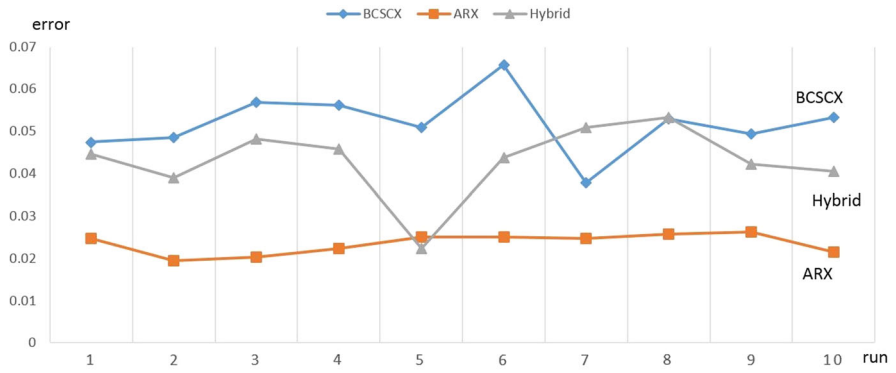


Fig. 7 Fitness after 150-generation evolution of 128 local minima genes (662 cities)

table. We also measured the error after 100 generations. Although ARX is slower in the early convergence, performance can be greatly improved by hybridization with BCSCX.

To investigate the advantage of ARX in diversity and continuous search, we applied ARX to locally optimized genes with preprocessing. We generated 128 local minima genes for a 662-city problem (xql662) by applying BCSCX and local improvement such as 2-opt [12]. The local minima genes were then used as initial genes for further evolution with BCSCX, ARX, and hybrid methods, and the convergence was measured as Table 3.

Figure 7 visually illustrates the convergence of the crossover methods shown in Table 3. ARX showed the best convergence when applied to differently evolved genes with low costs.

A similar experiment was done for a 10150-city problem (xmc10150). In this case, we increased the number of genes by multiplying the 128 local minima genes. As the size of the problem is larger than the 662-city experiment, we measured the ϵ after 500 generations with the different population sizes. The results are shown in Table 4. With this experiment, the increase of population size did not provide satisfactory convergence speed-up. To increase the diversity of genes, we divided the 2048 genes into 8 groups, and evolution was applied within each group. The result is shown in the last column in the table. The divided groups improved the performance of the crossover methods.

Performance comparison between GPU implementation and CPU implementation was also performed. We measured the performance for 4 problems which have 131,

Table 4 500-generation evolution of local minima genes with different population sizes (10150 cities)

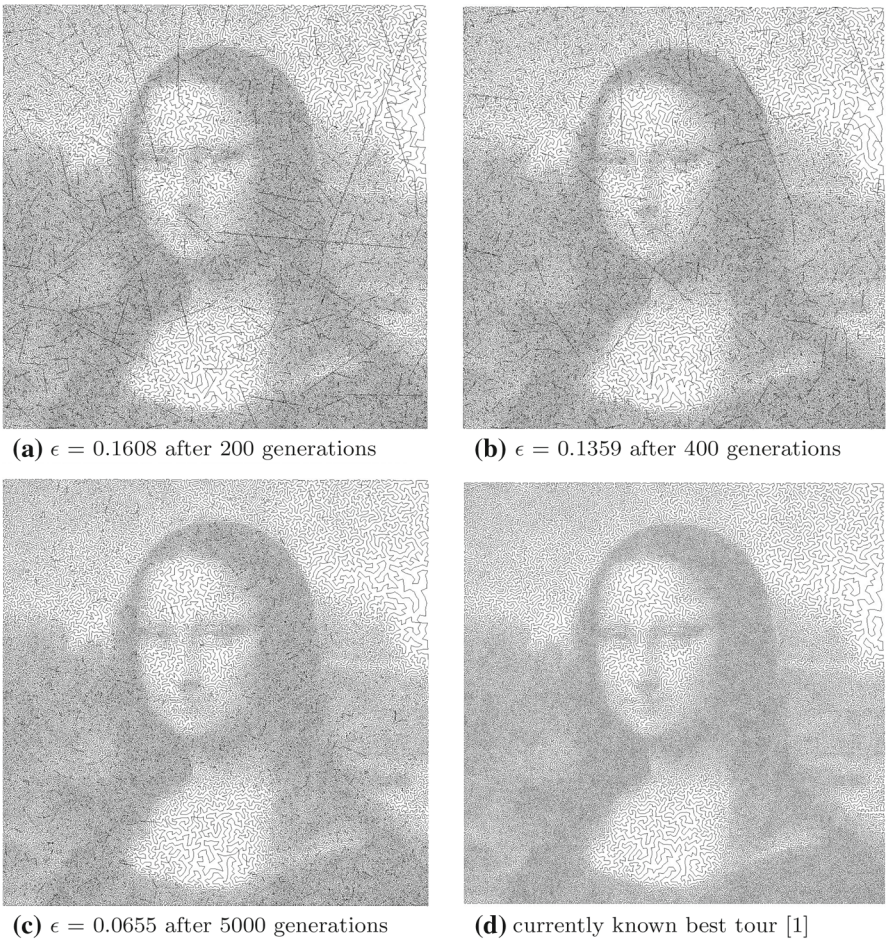
	10,150 cities (xmc10150): initial gene group precomputed to be $\epsilon = 0.1841$ Obtained ϵ after 500 generations			
	128 genes (1 group)	256 genes (1 group)	2048 genes (1 groups)	2048 genes (8 groups)
BCSCX	0.1077	0.1102	0.1107	0.0860
ARX	0.1036	0.1100	0.1019	0.0824
Hybrid	0.1038	0.1039	0.1030	0.0839

Table 5 Computation time required for one generation

Genes	Groups	Number of cities					
		131			2071		
		GPU (ms)	CPU (ms)	$\frac{\text{CPU}}{\text{GPU}}$	GPU (ms)	CPU (ms)	$\frac{\text{CPU}}{\text{GPU}}$
128	1	2.85	3.22	1.130	26.12	31.29	1.198
	8	4.17	5.32	1.276	29.01	108.71	3.747
256	1	4.03	5.33	1.323	28.32	49.47	1.747
	8	5.08	6.72	1.323	31.08	127.35	4.097
2048	1	15.22	19.15	1.258	63.25	326.35	5.160
	8	15.90	20.33	1.279	66.10	397.50	6.014
Genes	Groups	Number of cities					
		10,150			100,000		
		GPU (ms)	CPU (ms)	$\frac{\text{CPU}}{\text{GPU}}$	GPU (ms)	CPU (ms)	$\frac{\text{CPU}}{\text{GPU}}$
128	1	115.33	208.25	1.806	1160.29	5160.38	4.447
	8	127.85	976.27	7.636	1315.88	29,093.50	22.110
256	1	124.78	289.18	2.318	1308.41	6212.37	4.748
	8	132.71	1064.35	8.020	1490.96	30,559.15	20.496
2048	1	288.95	1632.42	5.649	5514.09	25,905.06	4.698
	8	340.51	2397.90	7.042	5569.41	50,387.65	9.047

2071, 10,150, and 100,000 cities, respectively. Population size was also changed to measure the performance in various environments. Genes evolved once as one group, and once as eight groups. The performance was measured with the time used to produce the next-generation genes in milliseconds. The column titled CPU/GPU shows how slow the CPU implementation is when compared with GPU version. In other words, these values indicate the speed-up of GPU implementation over the CPU version. As shown in the table, the GPU implementation showed better performance when the population size or the number of cities increases (Table 5).

The proposed method was applied to a large-scale TSP with 1,000,000 cities. Figure 8 shows the result. 2048 genes were used and divided into 8 groups. The relative



(a) $\epsilon = 0.1608$ after 200 generations

(b) $\epsilon = 0.1359$ after 400 generations

(c) $\epsilon = 0.0655$ after 5000 generations

(d) currently known best tour [1]

Fig. 8 The result of the proposed method applied to a large-scale TSP

error of the best fitness of the initial random gene pool was larger than 17.00. However, only after 200 generations, the gene with $\epsilon = 0.1608$ was found and the solution is shown in Fig. 8a. Figure 8b, c shows the errors after 400 and 5000 iterations, respectively. After 5000 generations, the genetic approach produced a gene with $\epsilon = 0.0655$. The currently known best tour is shown in Fig. 8d.

The experiments were performed on a single GPU system. Therefore, the sizes of the problem and gene pool were restricted by the capability of the GPU. To increase the problem size, a multi-GPU cluster system must be considered.

5 Conclusion

In this paper, we proposed an effective parallel approach to genetic TSP where crossover methods cannot be easily implemented in parallel fashion. The crossover

of the proposed method is a constructive approach like the previous crossover methods for TSP. Those constructive methods have serious disadvantages in that they have greedy aspects in the construction of offspring sequences and it is difficult to in parallel perform the crossovers.

The method proposed in this paper, ARX, lessens the greediness of crossover methods like SCX and BCSCX by alternating the influence of parents on offspring construction. ARX makes it possible for gene pools to maintain diversity so that the evolution-based search does not stagnate in local minima. The experimental results show that the hybridization of rapidly converging constructive method and the diversity-conserving ARX can significantly improve evolution performance.

Although the constructive crossover methods cannot be performed in parallel, GPU parallelism still can be exploited by creating threads for each gene. In this case, the genes should be sufficiently large when compared with the problem size (the number of cities). However, simple increase of population size does not guarantee better performance because the constructive methods rapidly decrease the diversity in the gene pool. We divided the genes into several groups to exploit the effect of increasing the number of genes, and the experimental results showed that evolution is improved by separating genes into groups, even with the loss of intergroup communication. GPU implementation was more efficient in such multi-population environments.

References

1. Ahmed ZH (2010) Genetic algorithm for the traveling salesman problem using sequential constructive crossover operator. Int J Biom Bioinform 3(6):96
2. Arabnia HR (1995) A distributed stereocorrelation algorithm. In: Proceedings of computer communications and networks, pp 479–482
3. Arabnia HR, Bhandarkar SM (1996) Parallel stereocorrelation on a reconfigurable multi-ring network. J Supercomput 10(3):243–270
4. Bäck T (1996) Evolutionary algorithms in theory and practice: evolution strategies, evolutionary programming, genetic algorithms. Oxford University Press, Oxford
5. Banzhaf W (1990) The molecular traveling salesman. Biol Cybern 64(1):7–14
6. Chatterjee S, Carrera C, Lynch LA (1996) Genetic algorithms and traveling salesman problems. Eur J Oper Res 93(3):490–510
7. Chitty DM (2007) A data parallel approach to genetic programming using programmable graphics hardware. In: Proceedings of the 9th annual conference on genetic and evolutionary computation. ACM, New York, pp 1566–1573
8. De Jong KA, Spears WM (1991) An analysis of the interacting roles of population size and crossover in genetic algorithms. In: Parallel problem solving from nature. Springer, Berlin, pp 38–47
9. Fogel DB (1993) Applying evolutionary programming to selected traveling salesman problems. Cybern Syst 24(1):27–36
10. Golberg DE (1989) Genetic algorithms in search, optimization, and machine learning. Addison Wesley, Boston
11. Grefenstette J, Gopal R, Rosmaita B, Van Gucht D (1985) Genetic algorithms for the traveling salesman problem. In: Proceedings of the first international conference on genetic algorithms and their applications. Lawrence Erlbaum, New Jersey, pp 160–168
12. Hasegawa M, Ikeguchi T, Aihara K (1997) Combination of chaotic neurodynamics with the 2-opt algorithm to solve traveling salesman problems. Phys Rev Lett 79(12):2344–2347
13. He B, Fang W, Luo Q, Govindaraju NK, Wang T (2008) Mars: a mapreduce framework on graphics processors. In: Proceedings of the 17th international conference on parallel architectures and compilation techniques. ACM, New York, pp 260–269

14. Hoffman KL, Padberg M, Rinaldi G (2013) Traveling salesman problem. In: Encyclopedia of operations research and management science. Springer, Berlin, pp 1573–1578
15. Kang S, Kim SS, Won JH, Kang YM (2015) Bidirectional constructive crossover for evolutionary approach to travelling salesman problem. In: 2015 5th international conference on IT convergence and security (ICITCS). IEEE, pp 1–4
16. Papadimitriou CH (1977) The Euclidean travelling salesman problem is np-complete. Theor Comput Sci 4(3):237–244
17. Pereira FB, Tavares J, Machado P, Costa E (2002) GVR: a new genetic representation for the vehicle routing problem. In: Artificial intelligence and cognitive science. Springer, Berlin, pp 95–102
18. Salomon-Ferrer R, Gtz AW, Poole D, Le Grand S, Walker RC (2013) Routine microsecond molecular dynamics simulations with AMBER on GPUs. 2. Explicit solvent particle mesh Ewald. J Chem Theory Comput 9(9):3878–3888
19. Sanders J, Kandrot E (2010) CUDA by example: an introduction to general-purpose GPU programming. Addison-Wesley Professional, Boston
20. TSP test data. http://www.math.uwaterloo.ca/tsp/data/index.html. Accessed 25 Nov 2015