Genetic Algorithms Using Parallelism and FPGAs: The TSP as Case Study

Miguel A. Vega-Rodríguez, Raúl Gutiérrez-Gil, José M. Ávila-Román,
Juan M. Sánchez-Pérez, Juan A. Gómez-Pulido
Univ. Extremadura. Dept. Informática
Escuela Politécnica. Campus Universitario s/n. 10071 Cáceres. SPAIN
E-mail: mavega@unex.es FAX: +34-927-257202

Abstract

In this work a detailed study about the implementation of Genetic Algorithms (GAs) using parallelism and Field Programmable Gate Arrays (FPGAs) is presented. Concretely, we use the Traveling Salesman Problem (TSP) as case study. First at all, the TSP is described as well as the GA used for solving it. Afterwards, we present the hardware implementation of this algorithm. We detail 13 different hardware versions, searching that each new version improves the previous one. Many of these improvements are based on the use of parallelism techniques. Finally, the found results are shown and analysed: Hardware/software comparisons, resource use, operation frequency, etc. We conclude indicating the parallelism techniques that obtain better results and stating FPGA implementation is better when the problem size increases or when better solutions (nearer to the optimum) must be found.

1. Introduction

Genetic Algorithms (GAs) have been satisfactorily applied for solving many optimisation problems. However, a GA used for solving a great size problem may be running during several days, even when it is run on a high-performance machine [1]. For this reason, methods for increasing the algorithm speed-up are necessary. We believe that implementing a GA directly on hardware may increase its efficiency (decreasing the running time) in such a way that we can face up to more complex problems as well as to perform more detailed researches about the operation mode of that GA. In this work, we describe the hardware implementation, using FPGAs and parallelism techniques, of a GA for solving the TSP (Traveling Salesman Problem).

In the literature another studies related with the FPGA use in evolutionary computing may be found. In [2] a full review about these subjects is performed, distinguishing between the researches devoted to implement on FPGAs exclusively the fitness function, the ones that implement the full genetic or evolutionary algorithm, and those related with evolving hardware.

In this work the following steps have been given: Using Visual C++ a genetic algorithm has been implemented for solving the TSP. From this software implementation many experiments have been made in order to study the influence and importance of the different parameters (population size, crossover operator,…) on the quality of the solutions found by the used GA. After obtaining the optimal values for the GA parameters, we have developed the hardware version of the GA (using a Virtex-E FPGA and the Handel-C language). Then, the hardware version has been optimised using internal arrays (FPGA resources instead of external memory banks) and parallelism techniques. In this way, the execution time has been considerably reduced. Finally, we have analysed and compared the time measures on the different implementations and we have conclude that the more work volume the more efficient/better the hardware version is, overcoming the software version if the work amount is sufficiently high.

2. The TSP

In the TSP we have a set of nodes, which represent cities or, simply, places. Starting in a node, we have to visit all the nodes, visiting each node exactly once, and stopping in the initial node. Each arc/union of two nodes is characterized by a C_{ij} value, which indicates the distance or cost to go from the *i* node to the *j* node. To solve the problem, it is necessary to find the order in which the nodes must be visited with the goal that the travelled distance/cost is the minimum. There are several TSP variants: symmetric, asymmetric, Hamiltonian, etc. In this work, we use the more classical and popular version: The symmetric TSP.

At first sight, the TSP seems an easy problem, and it could be thought that an exhaustive search could be enough for finding all possible tours. However, for N nodes, the number of possible tours is (N-1)!/2, and consequently, the total time to find all the tours is proportional to N!. For example, for only N=10 nodes, N! is equal to 3,628,800, a quite high number [3]. In fact, within the complexity theory, the TSP is considered as a NP-complete combinatorial problem, being necessary to use a heuristic search. In the literature may be found many alternatives for solving this problem. In particular, in [4] there is a detailed review of the literature related with this problem. Among the possible alternatives the genetic algorithms have an important place.

The TSP has a great number of practical applications: It is the base for many applications of transportation, logistics and delivery of merchandise. It is also used in the scheduling of machines to drill holes in circuit boards or other objects, in the construction of radiation hybrid maps as part of the work in genome sequencing, in the optimisation of scan chains in integrated circuits, in the design of fiber optical networks, etc. [5]

3. The genetic algorithm used

We use a genetic algorithm based on the classical GAs. In our case, the first step of the GA consists in creating randomly the initial population that has 197 individuals. These individuals will be ordered according to the fitness function. Then, the following actions are made in each iteration of the GA:

- Three members of the population are randomly selected.
- These three members/parents are crossed among them for generating three new offsprings.
- The population individuals suffer mutations with an 1% probability.
- The three new offsprings are added to the population ordered according to their fitness function. In this time the population has its highest size, 200 individuals.
- The three worst adapted members are removed, and therefore, the population has again 197 individuals.

In our case, the algorithm stop condition is reached either when the maximum iteration number (50,000) is finished or when the population is stabilized. We consider the population is stabilized when the best adapted individual is not modified during 1,000 iterations.

It is important to emphasize that the election of all these values (population of 200 individuals, crossover among 3 parents, mutation probability of 1%, maximum number of iterations 50,000, stabilization at 1,000 iterations, etc.) has been made after performing many experiments. These values have been chosen because they lead to the best results.

An individual consists in a cyclical tour, for this reason, we have used the *path representation* [4], which is the most natural representation of one tour. Furthermore, this is the most used representation for the TSP with GAs, since it is the most intuitive and has also lead to the best results [4]. The crossover operator used is the ER (*genetic Edge Recombination crossover*). This operator was developed by Whitley et al. [6-7], and according to [4], it is one of the crossover operators better adapted to the TSP. The mutation operator used in this work is the IVM (*InVersion Mutation*) [8-9], which is the most suitable for the TSP with the ER crossover operator [4]. The IVM operator is also called *Cut-Inverse Mutation Operator* [10] in other works. Finally, the fitness function selected is the sum of all the weights (distances or costs, C_{ij}) of all the arcs/unions in the cyclical tour. The best adapted individuals will have the minimum value fitness function.

4. Hardware implementation using parallelism and FPGAs

For the hardware implementation of the genetic algorithm a Xilinx Virtex-E FPGA has been used. Concretely, the XCV2000E FPGA, included in the Celoxica RC1000 board [11]. The hardware description has been performed by means of Handel-C language and the DK1 design suite [11].

For generating the random numbers in the GA (i.e. for creating randomly the initial population, controlling the mutation, etc.) a *Linear Feedback Shift Register* (LFSR) of 16 bits has been used.

As starting point we have designed, using Handel-C, a GA similar to that of the software version in Visual C++, without adding parallelism and making an extensive use of RC1000 board memory banks. In successive hardware versions of the genetic algorithm, improvements to the hardware programming have been added for comparing their results. We summarize the modifications performed in each of the different hardware versions of the GA:

- *Version 1 (V1)*: Original version, similar to the software version in Visual C++.

- *Version 2 (V2):* Modification of the individual creation. With the new method, it is not necessary to examine completely the node array (for checking if the node already exists within the tour/individual) each time a node of the new individual is generated. The *Create_Individual()* method is modified, allowing the access in parallel to different nodes of an individual (use of parallelism).
- *Version 3 (V3):* Parallelism sentences are introduced along all the code (*par* sentences in Handel-C). In Handel-C, all the sentences included in a *par* code block will operate in parallel.
- *Version 4 (V4):* Creation of the internal RAMs (FPGA resources instead of external memory banks) for *DistanceArray* and *MappingArray*, saving in each iteration many accesses to the board memory banks (they are slower).
- *Version 5 (V5):* Maximal reduction of the sizes in bits of the variables and structures used in hardware. The internal RAM *Individual* is created in the *Create_Individual()* method.
- *Version 6 (V6):* Parallelism is added to the *Sorting()* method, and in all the code where *DistanceArray* and *MappingArray* are used. Now, *DistanceArray* and *MappingArray* are different internal RAMs and we can access to them at the same time. Before version 4, it was not possible because they were stored in the same memory bank (only one access to the memory bank at a time).
- *Version 7 (V7):* Improvement in the ER crossover algorithm and in the neighbour map [4]. This map is implemented as an internal RAM (*EdgeMap* structure). In the *Create_Offspring()* method the *FlagArray* structure is added as internal array, and the *CandidateArray* structure is implemented as internal array instead of internal RAM for using it with parallelism (multiple accesses in parallel). An internal array, like an internal RAM, uses FPGA resources instead of memory banks, having faster accesses. Furthermore, an internal array allows multiple accesses to different array positions in parallel (an internal RAM only allows one access to one array position at the same time). An internal array consumes more FPGA resources than an internal RAM.
- *Version 8 (V8):* Improvement in the mutation. In the *Mutate_Individual()* method, the auxiliary structures *Individual* and *Aux_Population* are incorporated by means of internal arrays (FPGA resources). In this way, we can apply parallelism to these structures (multiple accesses in parallel).
- *Version 9 (V9):* The *Sorting()* method is removed, and it is substituted by a method of insertion in order (*Insert_Order()*). This implies that the process of initial creation of the population (it is the only one that uses the sorting) may be placed outside the FPGA. All this also involves the removal of the methods *Extend_Population(), Create_Initial_Population()* and *Create_Individual(),* because they are not necessary now.
- *Version 10 (V10):* Data replication. The distance triangular matrix is stored in the four memory banks of the board for accessing to memory four times at the same time when the individual distances are computed. That is, making the memory accesses in parallel, four times quicker.
- *Version 11 (V11):* Data compactation. The node data have a 7-bit size. They are stored from four in four in each memory location (32 bits). So, four nodes are read in only one access. This allows us to include more parallelism. In the previous versions, by simplicity, each memory location only stored one node.
- *Version 12 (V12):* This version is equal to the version 8, but with *Stabilization* at 16,000 iterations. In this way, we can check if the efficiency of the hardware implementation improves when we increase the stabilization.
- *Version 13 (V13):* This version is equal to the version 11, but with *Stabilization* at 16,383 iterations (maximum value with 14 bits). Furthermore, we have removed the *Annul_Individual()* method, because it is not necessary now.

As we can observe, many of the improvements included in the successive hardware versions are based on the use of parallelism techniques. Concretely, the hardware versions 2, 3, 6, 7, 8, 10 and 11 include improvements based on the use of parallelism.

5. Results

In order to perform our experiments we have used the TSPLIB library [12]. TSPLIB is a library of sample instances (benchmarks) for the TSP collected from multiple sources. These instances cover the different TSP types (symmetric, asymmetric, Hamiltonian cycles,…). All the files belonging to this library are in a specific format.

Table 1. Ratio between the HW/SW average execution times

Instance	V1	V2	V3	V4	V5	V6	V7	V8	V9	V10	V11	V12	V13
Burma14	38.627	36.918	36.773	25.003	24.444	22.899	22.396	22.383	19.382	19.317	19.204	14.024	14.919
Ulysses16	33.277	31.085	30.902	20.126	19.546	18.241	17.723	17.704	17.258	17.198	17.078	12.160	12.672
Ulysses22	23.358	24.020	23.893	15.589	14.962	14.157	13.594	13.581	11.662	11.602	11.474	8.640	8.723
Gr24	23.914	22.060	21.928	14.318	13.685	12.992	12.423	12.416	12.599	12.538	12.403	7.539	7.861
Bayg29	19.252	18.897	18.775	12.524	11.889	11.396	10.827	10.815	9.769	9.706	9.573	5.861	6.574
Bays29	17.558	18.255	18.133	11.858	11.220	10.727	10.155	10.143	10.032	9.970	9.837	5.832	6.285
Gr48	10.801	10.870	10.780	7.435	6.863	6.690	6.182	6.175	7.050	7.001	6.870	2.961	3.489
Eil51	13.774	9.685	9.597	6.670	6.111	5.961	5.465	5.458	5.354	5.305	5.180	2.914	3.216
Berlin52	8.905	9.340	9.255	6.256	5.702	5.557	5.065	5.059	4.692	4.643	4.523	2.876	2.975
Brazil58	7.402	8.905	8.826	6.310	5.786	5.676	5.211	5.206	4.219	4.175	4.060	2.460	2.598
St70	6.622	7.075	7.009	5.229	4.761	4.693	4.281	4.277	3.712	3.674	3.562	2.055	2.244
Pr76	5.825	5.985	5.928	4.342	3.906	3.852	3.468	3.464	3.385	3.352	3.245	1.567	2.137
Rat99	4.070	3.759	3.721	2.838	2.507	2.482	2.193	2.191	2.280	2.257	2.172	1.312	1.747
Rd100	3.749	4.124	4.077	3.134	2.787	2.761	2.457	2.455	2.297	2.269	2.183	1.424	1.493

In conclusion, the TSPLIB library contains a great number of TSP instances from around the world in a normalized format, but also it contains the best solution known for each instance until the date. This fact converts this library in a reference for everyone that works with the TSP.

Table 1 shows the ratio between the average execution time of the different hardware versions (XCV2000E FPGA of the RC1000 board) and the corresponding software versions (1.7-GHz Pentium-IV with 768 MB of RAM). These data are shown for multiple instances of the TSPLIB library. Each software implementation has the same improvements than its corresponding hardware implementation. For every instance (TSP problem), both implementations (hardware and software) find solutions of identical quality (in fact, due to the pseudo-random number generation both implementations find identical solutions). Although we do not show these solutions for space reasons, we indicate that in all the cases we have found the optimal solution or a near solution. Note that the instance names include a suffix that represents the instance size/difficulty, that is, its number of nodes.

In order to make easier the interpretation of the data in table 1, table 2 indicates the differences in the HW-time/SW-time ratio between a version and the previous one. In this table, the last row shows the average of the differences for each column. Using these averages we can estimate overall the improvement of a version with respect to the previous one.

From tables 1 and 2, we conclude that the advantage of the software version decreases when the problem size (number of nodes) increases, that is, when more time is necessary to solve the TSP problem. On the other hand, when hardware versions are improved (by the use of parallelism, reduction of accesses to memory banks by internal arrays, etc.) the HW-time/SW-time ratio decreases, doing the hardware comes closer more and more to the software. Assuming an improvement is significant when it surpasses the average 0.5 in the last row of table 2, the more significant changes/improvements in this ratio are produced in versions 4, 5, 9, 12 and 13 (comparing version 13 with respect to version 11, from where version 13 comes). The improvement of version 4 indicates that the use of internal resources of the FPGA for data storing allows us to reduce the number of accesses to memory banks, improving the hardware implementation efficiency. Version 5 shows us that the use of the exact number of bits required for each datum is important in the hardware implementations. In version 9, where the individual sorting is changed by an insertion in order, an important time improvement is obtained in the hardware version, which has more penalization than the software version in the data access. This improvement will be more pronounced while the problem size is smaller, because it will have more weight in the algorithm. The improvements in versions 12 and 13 display that increasing the *Stabilization* value (that is, the genetic algorithm must do more iterations) is reached a point from which the software version is slower than the hardware version, because the more iterations the more advantages for the FPGA implementation.

From the point of view of the parallelism, these are the hardware versions that include improvements based on the use of parallelism, and the parallelism techniques applied:

- *V2:* Data parallelism due to multiple accesses to different array positions in parallel.
- *V3:* Functional parallelism [13], performing several sentences at the same time.
- *V6:* Data parallelism because of accessing to different arrays in parallel.
- *V7* and *V8:* Data parallelism due to both multiple accesses to different array positions and several accesses to different arrays at the same time.

Table 2. Differences in the HW-time/SW-time ratio between a version and the previous one

Instance	V2-V1	V3-V2	V4-V3	V5-V4	V6-V5	V7-V6	V8-V7	V9-V8	V10-V9	V11-V10	V12-V11	V13-V12
Burma14	1.709	0.145	11.77	0.559	1.545	0.503	0.013	3.001	0.065	0.113	5.18	-0.895
Ulysses16	2.192	0.183	10.776	0.58	1.305	0.518	0.019	0.446	0.06	0.12	4.918	-0.512
Ulysses22	-0.662	0.127	8.304	0.627	0.805	0.563	0.013	1.919	0.06	0.128	2.834	-0.083
Gr24	1.854	0.132	7.61	0.633	0.693	0.569	0.007	-0.183	0.061	0.135	4.864	-0.322
Bayg29	0.355	0.122	6.251	0.635	0.493	0.569	0.012	1.046	0.063	0.133	3.712	-0.713
Bays29	-0.697	0.122	6.275	0.638	0.493	0.572	0.012	0.111	0.062	0.133	4.005	-0.453
Gr48	-0.069	0.09	3.345	0.572	0.173	0.508	0.007	-0.875	0.049	0.131	3.909	-0.528
Eil51	4.089	0.088	2.927	0.559	0.15	0.496	0.007	0.104	0.049	0.125	2.266	-0.302
Berlin52	-0.435	0.085	2.999	0.554	0.145	0.492	0.006	0.367	0.049	0.12	1.647	-0.099
Brazil58	-1.503	0.079	2.516	0.524	0.11	0.465	0.005	0.987	0.044	0.115	1.6	-0.138
St70	-0.453	0.066	1.78	0.468	0.068	0.412	0.004	0.565	0.038	0.112	1.507	-0.189
Pr76	-0.16	0.057	1.586	0.436	0.054	0.384	0.004	0.079	0.033	0.107	1.678	-0.57
Rat99	0.311	0.038	0.883	0.331	0.025	0.289	0.002	-0.089	0.023	0.085	0.86	-0.435
Rd100	-0.375	0.047	0.943	0.347	0.026	0.304	0.002	0.158	0.028	0.086	0.759	-0.069
Average	0.440	0.099	4.855	0.533	0.435	0.475	0.008	0.545	0.049	0.117	2.839	-0.379

- *V10:* Data parallelism because of data replication. For very large arrays (for example, the distance triangular matrix), we have to store them in the memory banks of the board because there are not enough FPGA resources for implementing them using internal RAMs or internal arrays. In these cases, we have created several copies of the same array in different memory banks. Therefore, we can perform several accesses to the same array in parallel, although it is in memory banks.
- *V11:* Data parallelism due to data compactation. Now, each memory bank location (32 bits) does not store one node (7 bits) of an individual, but it stores four compacted nodes in the same memory word. In this way, in only one access we can read four times more data, and thus, we can apply more parallelism.

In summary, we have used very different parallelism techniques, applying the two possible kinds of parallelism [13]: functional parallelism and data parallelism. In this case, data parallelism obtains greater improvements in the hardware implementations. On the one hand, in the code of our GA there are many data accesses, and on the other hand, this high number of accesses limits the parallelization of other code portions. In particular, from table 2 we conclude that the parallelism techniques that obtain better results are the ones applied to the versions 7, 2 and 6, in this order. Version 8 generates the worst improvements because this version optimises the mutation, and even though this is a good optimisation, it is used very little (mutation probability is 1%).

Table 3 shows the comparisons of FPGA occupation (resource use) and maximum frequency for the different hardware versions. The resource use of the FPGA increases while we apply more and more improvements to the hardware versions, because the improvements imply the parallelism exploitation, the use of internal arrays for data storing, etc. We highlight versions 5 and 9. In version 5, due to we have reduced the size in bits of the variables and data structures, the FPGA occupation has been decreased in more than 50%. In version 9, the substitution of the sorting by the insertion in order also releases FPGA resources. Finally, remember that version 12 is identical to version 8, but with *Stabilization* at 16,000 iterations (for this reason, both versions have the same results).

As for the frequency (and the period), this is practically stable in all the hardware versions (varying slightly between 11.157 MHz and 15.588 MHz).

Table 3. Resource use, frequencies and periods for the different hardware versions

Version	Resource Use (Slices)	FPGA Occupation (% of 19,200)	Maximum Frequency (MHz)	Minimum Period (ns)
V1	4696	24%	14.641	68.301
V2	6316	32%	15.108	66.188
V3	6294	32%	15.588	64.152
V4	5575	29%	14.426	69.321
V5	2728	14%	12.793	78.167
V6	4007	20%	11.157	89.629
V7	7821	40%	12.194	82.007
V8	10411	54%	13.008	76.874
V9	8984	46%	13.927	71.805
V10	11954	62%	11.881	84.167
V11	16858	87%	13.224	75.620
V12	10411	54%	13.008	76.874
V13	16796	87%	13.807	72.428

6. Conclusions

In this work we have performed a study about the implementation of GAs by means of parallelism and FPGAs, using the TSP as case study. Figure 1 summarizes graphically some of the more important conclusions we have found.

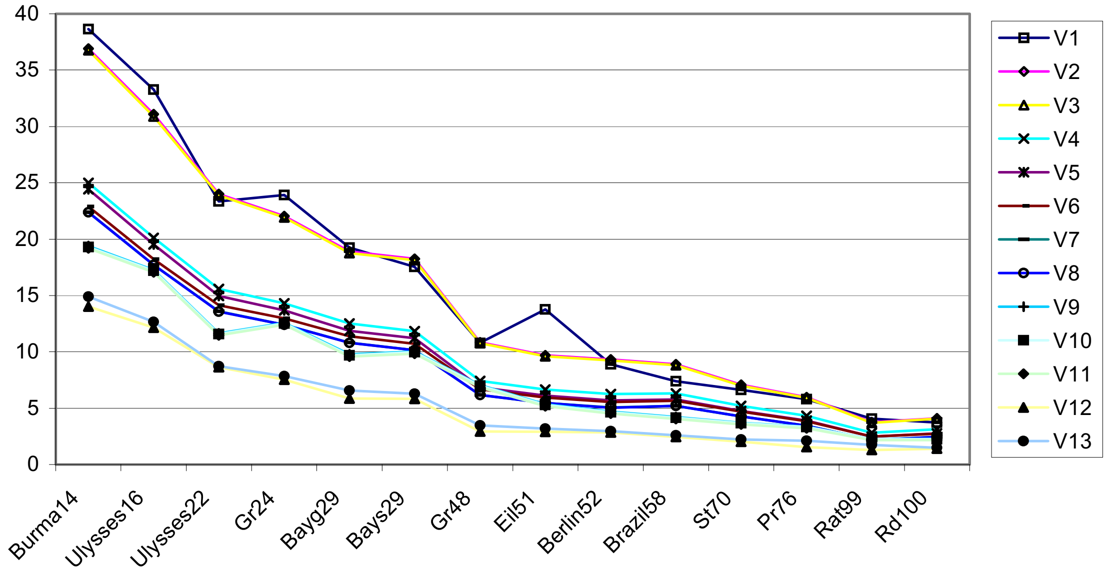

Figure 1. Graphical representation of the HW/SW ratios from table 1

The evolution of the HW/SW ratio, in the 13 versions, indicates clearly that, for problems whose size is greater than 100 nodes, the FPGA implementation will become more efficient than the software implementation, increasing this difference when the problem has more size/complexity (more time for solving it). The evolution in the 13 versions also displays that each new version improves the previous one.

It is clear that after fixing the best parameters and the best GA for the problem, the quality of the obtained solutions will depend on the time we employ in finding the solution, that is, the number of iterations used for obtaining the solution. In our case, it will depend on the *Stabilization* parameter. As it can be seen in this figure and in table 1, the higher the stabilization is, the more efficient the hardware (FPGA) version is with respect to the software version. In conclusion, the hardware implementation also gathers strength in this aspect, since it will be more necessary when more work has to be done.

From the point of view of the parallelism, we have studied very different parallelism techniques in order to improve the efficiency of the FPGA implementation of GAs. From this work, using the TSP, we conclude that we obtain greater improvements using the data parallelism. Particularly, the data parallelism techniques that include multiple accesses to different array positions or several accesses to different arrays at the same time.

Finally, another important conclusion is that FPGAs do not adapt in the same way to all the GAs. In this case, the TSP, the fitness function implies a great number of memory accesses for reading both the individuals and the node list of each individual (nodes in each tour), or even the distances between different nodes. A number so great of memory accesses penalizes the efficiency of the hardware version in relation to the software one. For this reason, we believe that other GAs, whose fitness functions have few memory accesses and more computation, would be more suitable for FPGA implementation. In the TSP case, the computation is minimal, because the fitness function only has to add the different distances between the nodes in the tour/individual.

7. Acknowledgments

This work has been supported in part by the Spanish Government under Grant TIC2002-04498-C05-01, TRACER project.

8. References

[l] Aporntewan, C., Chongstitvatana, P.: A Hardware Implementation of the Compact Genetic Algorithm. In Proc. Congress on Evolutionary Computation (CEC2001), Seoul, Korea (2001) 624–629

[2] Martin, P.: A Hardware Implementation of a Genetic Programming System using FPGAs and Handel-C. Genetic Programming and Evolvable Machines, 2(4) (2001) 317-343

[3] Rich, E., Knight, K.: Artificial Intelligence. McGraw-Hill, 2nd edition (1991)

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

[5] Princeton University: Traveling Salesman Problem. www.math.princeton.edu/tsp/ (2005)

[6] Whitley, D., Starkweather, T., Fuquay, D.: Scheduling Problems and Traveling Salesman: The Genetic Edge Recombination Operator. In Proc. 3rd International Conference on Genetic Algorithms, Los Altos, CA, USA (1989) 133–140

[7] Whitley, D., Starkweather, T., Shaner, D.: The Traveling Salesman and Sequence Scheduling: Quality Solutions Using Genetic Edge Recombination. Handbook of Genetic Algorithms, Van Nostrand Reinhold, New York, USA (1991) 350–372

[8] Fogel, D.B.: A Parallel Processing Approach to a Multiple Traveling Salesman Problem Using Evolutionary Programming. In Proc. 4th Annual Parallel Processing Symposium, Fullerton, CA, USA (1990) 318–326

[9] Fogel, D.B.: Applying Evolutionary Programming to Selected Traveling Salesman Problems. Cybernetics and Systems, 24 (1993) 27–36

[10] Banzhaf, W.: The "Molecular" Traveling Salesman. Biological Cybernetics, 64 (1990) 7–14

[11] Celoxica Ltd: www.celoxica.com (2005)

[12] Reinelt, G.: TSPLIB. www.iwr.uni-heidelberg.de/groups/comopt/software/TSPLIB95/ (2004)

[13] Sima, D., Fountain, T., Kacsuk, P.: Advanced Computer Architecture: A Design Space Approach. Addison-Wesley (1998)