Computing and Informatics, Vol. 31, 2012, 675–692

A DISTRIBUTED EVOLUTIONARY ALGORITHM WITH A SUPERLINEAR SPEEDUP FOR SOLVING THE VEHICLE ROUTING PROBLEM

Krunoslav Puljić, Robert Manger

Department of Mathematics, University of Zagreb
Bijenička cesta 30, 10000 Zagreb, Croatia
e-mail: {nuno, manger}@math.hr

Communicated by Vladimír Kvasnička

Abstract. In this paper we present a distributed evolutionary algorithm for solving the capacitated vehicle routing problem. Our algorithm consists of autonomous processes that create heterogeneous evolutionary environments, perform evolution on separate populations of chromosomes, and communicate asynchronously through occasional migrations of chromosomes. The paper also presents experiments where the algorithm has been tested on some benchmark problem instances. By measuring the effects of distribution on solution quality and on computing time, the experiments confirm that the algorithm achieves a superlinear speedup.

Keywords: Vehicle routing problem, evolutionary algorithms, distributed algorithms, superlinear speedup, experiments

Mathematics Subject Classification 2010: 90C27, 90C35, 90C59, 68W15, 68W40

1 INTRODUCTION

The *vehicle routing problem* (VRP) [18] is an interesting combinatorial optimization task, which occurs frequently in real-world applications. The problem deals with scheduling a fleet of vehicles to distribute goods between depots and customers. A set of routes for vehicles should be determined, which are in some sense optimal, e.g. the shortest or the cheapest. Certain constraints should be taken into account, such as customer demands or vehicle capacities.

Evolutionary algorithms (EAs) [12] are a popular metaheuristics, which tries to solve optimization problems by imitating processes observed in nature. An EA maintains a population of "chromosomes" where each of them encodes a feasible solution to a particular problem instance. Evolution of those chromosomes takes place through application of evolutionary operators such as selection, crossover, mutation, etc.

Distributed evolutionary algorithms (DEAs) [3] are a special kind of EAs, where the population of chromosomes is divided into subpopulations placed on isolated "islands". The evolutionary processes on islands run autonomously, and they interact only occasionally by some form of migration of chromosomes.

Since the VRP is computationally very hard, its large instances cannot be solved to optimality, but only approximately by metaheuristics. Thus it makes sense to consider applications of EAs to the VRP. There have been many such attempts in recent years, but most of them have been restricted to the conventional non-distributed form of evolution.

The aim of this paper is to explore capabilities of DEAs to solve the VRP. More precisely, the aim is to investigate how the evolutionary process used for solving the VRP is influenced by distribution. We expect that, apart from speeding up the whole computation, distribution can bring some additional gains, such as more diversity of chromosomes and better quality of solution.

The main contribution of this paper is the design of a concrete DEA for the VRP, which has been obtained after a detailed experimental evaluation of various design options. In accordance with our aim, the obtained algorithm is purely evolutionary, i.e. it is based only on evolutionary operators and migration, and it does not incorporate any other heuristics such as local search. The paper also presents the final series of experiments, which confirm that the algorithm achieves a superlinear speedup.

The paper is organized as follows. Section 2 lists preliminaries about the VRP and EAs. Section 3 describes the overall design of our EA for the VRP and specifies its building blocks. Sections 4 and 5 report on experiments where the implemented algorithm has been tested on a well known library of benchmark problem instances. Thereby Section 4 presents the results dealing with quality of solution, while Section 5 presents measurements of computing time and speedup. Section 6 contains a discussion that explains why our method of computing speedup is appropriate in spite of some possible criticisms. The final Section 7 gives conclusions.

2 PRELIMINARIES

There are many variants of the VRP found in literature. In this paper we consider the standard *capacitated* VRP [18], which is described as follows. Let $G = (V, A)$ be a complete directed graph, where $V = \{0, 1, 2, \ldots, n\}$ is the vertex set and A is the arc set. Vertices $i = 1, 2, \ldots, n$ correspond to the customers, and vertex 0 corresponds to the depot. A nonnegative cost c_{ij} is assigned to each arc $(i, j) \in A$,

and it represents the travel cost spent to go from vertex i to vertex j. Each customer vertex i is associated with a nonnegative demand d_i to be delivered, and the depot 0 has a fictitious demand $d_0 = 0$. A set of K identical vehicles, each with the capacity C, is available at the depot. The CVRP consists of finding a collection of $\leq K$ elementary cycles in G with minimum total cost, such that:

- each cycle visits the depot vertex 0,
- each customer vertex $i = 1, 2, \ldots, n$ is visited by exactly one cycle,
- the sum of the demands d_i of the vertices visited by a cycle does not extend the vehicle capacity C.

Obviously, the cycles constituting the solution to a VRP instance specify optimal routes for the vehicles delivering goods from the depot to the customers. Thereby the demand of each customer is satisfied and no vehicle is overloaded.

Note that the VRP can be considered as a generalization of the *traveling salesman problem* (TSP) [8]. Indeed, the TSP is a special case of the VRP where the number of vehicles K is equal to 1, and the capacity C of the vehicle is infinite.

As already mentioned before, an EA is a randomized algorithm which maintains a population of chromosomes. Each chromosome represents a feasible solution to a given instance of an optimization problem. The population is iteratively changed, thus giving a series of population versions usually called *generations*. It is expected that the best chromosome in the last generation represents a near-optimal solution to the considered problem instance.

An EA consists of many building blocks, which can be chosen and combined in various ways. Consequently, there is a wide variety of possible EAs for the same optimization problem. Here follows a list of most important building blocks and design options.

- The *data structure* used to represent a chromosome.
- The *initialization procedure* that produces the initial population of chromosomes.
- The *evaluation procedure* used to evaluate a chromosome to give some measure of its "goodness" or "fitness". The goodness measure is related to the objective function of the original optimization problem.
- The genetic operators that produce new chromosomes from old ones. There exist unary genetic operators, called *mutations*, which create new chromosomes (mutants) by a small random change in a single chromosome. There also exist higher order operators called *crossovers*, which create new chromosomes (children) by combining parts from several (usually two) chromosomes (parents).
- The *selection procedure*, used to find "good" chromosomes for crossover, or "bad" chromosomes that will be discarded from the population.
- The *termination condition*, which determines when the whole evolutionary process should be stopped.

Since the VRP is so much related to the TSP, it is not surprising that most EAs for solving the VRP are assembled from components that have originally been designed for the TSP. Indeed, it has been quite common to represent the chromosome for the VRP as a permutation of customer vertices $1, 2, \ldots, n$, which can be visualized as a big cycle spanning the whole graph [11]. With such representation, it is also possible to reuse a variety of genetic operators that transform permutations into permutations. Here are some of those operators with brief explanations given.

- *Order crossover* (OX) [9, 15]. One part of the child chromosome is directly copied from the first parent, and the remaining part is constructed by following the vertex ordering from the second parent.
- *Alternating edges crossover* (AEX) [14]. The child cycle is formed by choosing in alternation arcs from the first and from the second parent, with some additional random choices in case of infeasibility.
- *Heuristic greedy or random crossover* (HGreX or HRndX) [17]. The child cycle is formed by choosing from each vertex the shorter of the two respective parent arcs; in case of infeasibility some additional greedy or random choices are made.
- *Mutation by inversion* (IM) [9]. A part of the chromosome between two randomly chosen positions is reversed.
- *Mutation by reinsertion* (RM) [11]. A randomly chosen vertex is taken out from the chromosome and reinserted at a new randomly chosen position.
- *Swap mutation* (SM) [19]. Two randomly chosen vertices in the chromosome are swapped.

Note that, contrary to the TSP, a chromosome in form of a permutation does not determine uniquely a solution to the VRP. Namely, such chromosome must be interpreted as a concatenation of vehicle routes, and many different combinations of routes can produce the same concatenation. Consequently, using permutations as chromosomes makes evaluation procedure much more complicated, since prior to computing the objective function the permutation must be splitted into feasible individual routes. If we insist on optimal evaluation, i.e. on such splitting that minimizes the objective function, then the well known split procedure from [15] must be used whose complexity per evaluation is $\mathcal{O}(n^2)$. Another possibility is to use a greedy splitting procedure, which is much faster but not optimal.

Apart from the mentioned components that originate from the TSP, EAs for the VRP may also contain more general components that are applicable to any kind of problem. Here are some examples.

- Selection is usually accomplished by *tournament selection* [12], where a predefined number of chromosomes is picked up randomly, and then the best or the worst of them is selected.
- The initial population is often formed by some heuristics or randomly.
- The termination condition is usually based on the elapsed time or on the number of evaluations.

- Most algorithms support *elitism* [6], i.e. the best chromosome is never discarded.

In this paper we are dealing with DEAs, i.e. such EAs where more processes run concurrently, each of them maintains its own subpopulation of chromosomes, and communication is accomplished by migration of chromosomes. Obviously, DEAs belong to a broader class of parallel EAs (PEAs) [3], i.e. EAs that use some kind of parallel or concurrent computing.

There is a lot of papers on PEAs or even DEAs found in literature. However, we are aware of only four papers [2, 4, 5, 13] that describe PEAs specifically for the VRP. The algorithms from [2, 4, 5] cannot be called distributed since they use other paradigms of parallel computing. The algorithm from [13] is not a proper EA since it incorporates also two other heuristics. Thus we are not aware of any previous work that completely fits in our focus of interest and describes a distributed and purely evolutionary algorithm for the VRP.

3 ALGORITHM DESIGN

Our DEA for the VRP has been obtained after a careful experimental evaluation of various design options. We have tried different evaluation procedures, different crossover operators, and exclusion or inclusion of mutation or migration. Altogether we have tested 24 variants of the algorithm in order to choose the best one. The finally chosen variant is shown if Figure 1. Note that this pseudocode corresponds to one process (one island), and each process executes its own copy of the same code.

In our algorithm, the chromosome is represented as a permutation of customer vertices. The initialization procedure simply produces 30 random permutations. All tournament selections are performed with 2 participants within a tournament. The crossover operator is one of the following: OX, AEX, HGreX or HRndX. Mutation is either IM, SM or RM. Evaluation is always accomplished according to the optimal split procedure. One process executes only one crossover operator, and combines all types of mutations. However, if there are more processes, then each of them chooses a different crossover, thus creating a different environment for evolution. A process is stopped after a given number of evaluations.

Note that our algorithm needs at least four processes to be fully operational. Indeed, with that number of processes each crossover operator is performed by at least one process. Still, in our experiments we will also use the sequential version of the algorithm that runs with only one process. Since one process can execute only one crossover operator, there in fact exist four different sequential versions, each using a different crossover. We do not consider combining more crossovers within the same process since according to our preliminary testing such version would not produce better results.

In course of the algorithm, the population P changes due to insertions of new chromosomes. All insertions rely on the concept of similarity. We say that two chromosomes are *similar* if their goodness values differ in less than 0.5 % of the better value. Insertion "by taking into account similarity" means the following.

```
DistributedEvolutionVRP( ) {
   input the VRP instance and the EvaluationLimit;
   initialize the population P with 30 chromosomes;
   evaluate the whole population P;
   EvaluationCount = 0; GenerationCount = 1;
   while (EvaluationCount < EvaluationLimit) {
      // crossover
      Mother = a good chromosome from P selected by tournament;
      Father = a good chromosome from P selected by tournament;
      Child = crossover of Mother and Father;
      evaluate Child; EvaluationCount += 1;
      insert Child into P by taking into account similarity;
      // mutation
      Rnd = a random integer between 1 and 100;
      if (Rnd == 1) { // probability 1%
         Mutant = a randomly chosen chromosome from P;
         if (Mutant is not the best in P) {
            mutate Mutant;
            evaluate Mutant; EvaluationCount += 1;
         }
      }
      // migration
      if (GenerationCount % 50 == 0) {
         Emigrant = a good chromosome from P
         selected by tournament;
         S = a randomly chosen population in another process;
         send a copy of Emigrant to S;
         scramble Emigrant;
         evaluate Emigrant; EvaluationCount += 1;
      }
      if (Immigrant received) {
         evaluate Immigrant; EvaluationCount += 1;
         insert Immigrant into P by taking
         into account similarity;
      }
      GenerationCount += 1;
   }
}
```

Fig. 1. Pseudocode of our DEA for solving the VRP

- If there exists another chromosome in P that is similar to the new one, then the better of those two "twins" is retained in P and the other one is discarded.
- If there is no similar chromosome, then the new one is retained in P, and some other "bad" chromosome from P is selected by tournament and discarded.

It is easy to check that according to our rules the size of P always remains the same, i.e. 30. Indeed, whenever a chromosome is inserted, another one is discarded. Similarly, when an emigrant leaves P, it is replaced by its "scrambled" version, i.e. by a new random chromosome.

Note also that our algorithm does not necessarily preserve the best chromosome in P. Namely, that chromosome may migrate to another population S. Still, if the migrant is the best within the whole algorithm, then it will surely survive in S. Thus there is no strict elitism within one population, but there exists a form of elitism when all populations are considered together.

As we can see, our algorithm performs a relatively intensive migration policy, where one emigrant is sent every 50 generations, and an immigrant is received whenever it becomes available. The implemented migration frequency has been chosen since it produced the best results in preliminary testing. This is in contradiction with recommendations made by some other authors [3, 13] who claim that migrations should be less intensive.

Putting it all together, our algorithm is characterized by the following properties.

- It is *distributed*, because it consists of more processes that work on separate data and interact only by exchanging messages.
- It is *asynchronous*, namely processes do not wait one for another when they exchange messages.
- It is *purely evolutionary* or non-hybrid. Indeed, it is built of evolutionary procedures and operators, and it does not incorporate elements of any other heuristic or metaheuristic.
- It is *heterogeneous*, in the sense that each process runs a different combination of evolutionary procedures and operators.

4 ASSESSING THE QUALITY OF SOLUTION

In order to perform experiments, we have developed a C++ implementation of our DEA for the VRP. Communication among processes has been realized with the MPI library [16]. The implemented algorithm has been evaluated on 7 benchmark VRP instances from the well known Christofides-Mingozzi-Toth collection [7]. Some basic parameters of those instances are shown in Table 1. Experiments have been performed on the Isabella computer cluster [10]. Up to 64 truly parallel processes have been used.

Since our algorithm is randomized, its repeated execution on the same input data with the same number of processes usually produces slightly different results.

Problem instance	Number of customers (n)	Number of vehicles (K)	Vehicle capacity (C)	Cost of the best known solution
CMT01	50	5	160	524.61
CMT02	75	10	140	835.26
CMT03	100	8	200	826.14
CMT04	150	12	200	1 028.42
CMT05	199	17	200	1 291.45
CMT11	120	9	200	1 042.11
CMT12	100	10	200	819.56

Table 1. Benchmark VRP instances from the CMT collection

To amortize this effect of randomization, each experiment from our agenda has been repeated 30 times. Consequently, all measured values reported in the forthcoming Tables 2–5 and Figures 2–3 are in fact averages obtained over 30 repetitions. The corresponding standard deviations are all below 0.03, thus assuring that the reported averages are accurate and stable.

In our first round of experiments, we have solved each of problem instances with different numbers of processes. As the termination condition, we have fixed the number of evaluations *per process* to 1 000 000. It means that a run with more processes was in position to execute more evaluations. Note that the number of evaluations per process roughly corresponds to the computing time. Thus the experiments model a situation where one wants to improve solutions within a fixed time deadline by engaging more computing resources and doing more work in parallel.

Problem instance	Number of processes (p)							
	1	4	8	12	16	24	32	64
CMT01	541.62	531.40	535.21	532.55	532.55	529.97	530.62	529.28
	3.2 %	1.3 %	2.0 %	1.5 %	1.6 %	1.0 %	1.1 %	0.9 %
CMT02	871.65	862.47	858.09	857.24	857.78	854.24	855.29	853.48
	4.4 %	3.3 %	2.7 %	2.6 %	2.7 %	2.3 %	2.4 %	2.2 %
CMT03	857.91	848.27	846.02	841.68	840.86	840.44	840.66	837.81
	3.8 %	2.7 %	2.4 %	1.9 %	1.8 %	1.7 %	1.8 %	1.4 %
CMT04	1 104.34	1 074.14	1 073.04	1 067.62	1 062.55	1 068.14	1 062.53	1 058.43
	7.4 %	4.4 %	4.3 %	3.8 %	3.3 %	3.9 %	3.3 %	2.9 %
CMT05	1 403.60	1 368.52	1 365.52	1 359.06	1 360.35	1 352.58	1 352.51	1 348.23
	8.7 %	6.0 %	5.7 %	5.2 %	5.3 %	4.7 %	4.7 %	4.4 %
CMT11	1 090.55	1 064.37	1 058.49	1 056.60	1 054.83	1 053.28	1 054.46	1 052.02
	4.6 %	2.1 %	1.6 %	1.4 %	1.2 %	1.1 %	1.2 %	1.0 %
CMT12	828.42	821.51	820.68	820.60	820.80	820.82	820.57	820.62
	1.1 %	0.2 %	0.1 %	0.1 %	0.2 %	0.2 %	0.1 %	0.1 %
Average	4.7 %	2.9 %	2.7 %	2.4 %	2.3 %	2.1 %	2.1 %	0.8 %

Table 2. Solutions obtained with a fixed number of evaluations per process

The results of the first round of experiments are summarized in Table 2. The table is organized so that each of its rows corresponds to a particular problem instance, and each column to a particular number of processes p. Each table entry refers to the solution obtained for the corresponding combination of problem instance and p. The solution is described by its cost and as the relative error with respect to the best known solution. The average error for a chosen p over all instances is also given.

Figure 2 comprises a more detailed presentation of results for one chosen problem instance. Each of the three graphs corresponds to a particular number of processes and shows how the current solution for that number of processes depends on the number of evaluations already executed per processor. The labels on the left border of the figure denote the absolute solution cost, while the labels on the right border refer to the relative cost compared to the best known solution.

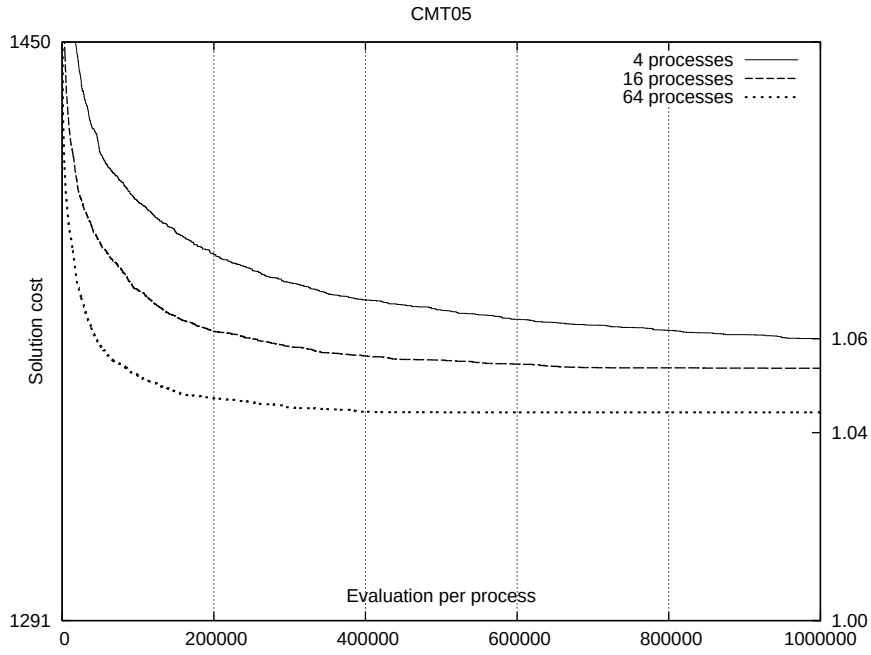

Fig. 2. Details for CMT05 – solution cost vs. number of evaluations per process

Note that Table 2 and Figure 2 include results where the number of processes is 1. As we explained before, there are in fact four sequential versions of the algorithm. Still, there is no confusion since any shown value refers to the version that happens to be the best for that particular problem instance. In this way, our evaluation of the algorithm is more strict. Namely, the results of distributed computing are compared to the best available results obtained sequentially.

The results presented in Table 2 and Figure 2 show clearly that the quality of solution improves with more processes. This is quite plausible, namely with more processes the algorithm executes more evaluations and is therefore able to find better solutions. Still, we cannot be sure whether the observed improvement can only be attributed to more computing.

In order to better understand how distribution affects the solution quality, we have also performed a second round of experiments. Again we have solved each problem instance repeatedly with more and more processes. But now we have fixed the *total* number of evaluations in all involved processes to 4 000 000. Thereby each of processes has been assigned roughly equal number of evaluations. Such experiments model a situation where one is satisfied with solution quality but wants to speed up computation by dividing the same amount of work to more computing resources that work in parallel. The results of the second round of experiments are summarized in Table 3. Organization of data is analogous to Table 2.

We can see from Table 3 that even in the second round of experiments the quality of solution improves with more processes. This improvement is smaller than in the first round, but it still exists. Since with more processes the total number of evaluations is not increased but only redistributed, the observed improvement cannot be attributed to more computing any more. Thus it must be a positive effect of distribution by itself. It means that even in the first round of experiments the improvement was only partially accomplished by more computing, while the remaining part was due to distribution.

Problem	Number of processes (p)							
Instance	1	4	8	12	16	24	32	64
CMT01	537.59	531.40	535.46	533.26	533.18	530.98	531.49	533.32
	2.5 %	1.3 %	2.1 %	1.6 %	1.6 %	1.2 %	1.3 %	1.7 %
CMT02	862.67	862.47	859.15	859.89	861.24	860.21	859.66	860.65
	3.3 %	3.3 %	2.9 %	2.9 %	3.1 %	3.0 %	2.9 %	3.0 %
CMT03	848.42	848.27	848.12	845.01	844.79	843.09	844.82	842.01
	2.7 %	2.7 %	2.7 %	2.3 %	2.3 %	2.1 %	2.3 %	1.9 %
CMT04	1 083.77	1 074.14	1 075.77	1 072.11	1 070.36	1 072.72	1 070.07	1 068.11
	5.4 %	4.4 %	4.6 %	4.2 %	4.1 %	4.3 %	4.0 %	3.9 %
CMT05	1 382.29	1 368.52	1 369.77	1 364.45	1 368.48	1 362.00	1 362.14	1 363.21
	7.0 %	6.0 %	6.1 %	5.7 %	6.0 %	5.5 %	5.5 %	5.6 %
CMT11	1 072.22	1 064.37	1 060.84	1 061.87	1 059.97	1 057.69	1 057.99	1 056.36
	2.9 %	2.1 %	1.8 %	1.9 %	1.7 %	1.5 %	1.5 %	1.4 %
CMT12	823.79	821.51	821.13	821.17	821.03	821.21	821.15	821.20
	0.5 %	0.2 %	0.2 %	0.2 %	0.2 %	0.2 %	0.2 %	0.2 %
Average	3.5 %	2.9 %	2.9 %	2.7 %	2.7 %	2.5 %	2.5 %	2.5 %

Table 3. Solutions obtained with a fixed total number of evaluations

5 MEASURING THE COMPUTING TIME

Apart from solution quality, in our experiments we have also measured computing time. With such measurements it has been possible to determine the *speedup* of our DEA, i.e. how much faster it runs with p processes than with 1 process.

Problem	Number of processes (p)							
instance	1	4	8	12	16	24	32	64
CMT01	53.25	15.97	7.98	5.68	4.11	2.82	2.10	1.17
		3.33×	6.67×	9.38×	12.95×	18.91×	25.36×	45.66×
CMT02	67.85	22.53	11.38	7.72	5.57	3.74	2.86	1.56
		3.01×	5.96×	8.79×	12.19×	18.13×	23.72×	43.49×
CMT03	126.63	39.58	20.02	13.10	9.51	6.41	4.85	2.60
		3.20×	6.33×	9.67×	13.32×	19.77×	26.10×	48.79×
CMT04	193.65	60.37	30.21	20.34	15.00	10.03	7.56	4.04
		3.21×	6.41×	9.52×	12.91×	19.31×	25.62×	47.92×
CMT05	243.32	81.37	40.41	26.58	19.71	13.16	9.89	5.18
		2.99×	6.02×	9.16×	12.35×	18.50×	24.62×	47.01×
CMT11	177.79	56.37	27.58	19.32	13.84	9.29	7.00	3.69
		3.15×	6.45×	9.20×	12.85×	19.13×	25.38×	48.21×
CMT12	108.63	37.93	19.79	12.72	9.25	6.26	4.71	2.54
		2.86×	5.49×	8.54×	11.74×	17.37×	23.08×	42.69×
Average	138.73	44.87	22.48	15.06	11.00	7.38	5.57	2.97
		3.11×	6.19×	9.18×	12.62×	18.73×	24.84×	46.25×

Table 4. Computing times for a fixed total number of evaluations

The following Table 4 shows the computing times and speedups for the second round of experiments, thus the same round whose solutions have previously been presented by Table 3. Organization of data is similar as in Tables 2 and 3. In each table entry, the upper value is the total computing time in seconds, while the lower value denoted with × is the speedup. The last row contains again average values for a chosen p over all instances.

As we can see, the average speedup values range from 3.11 for 4 processes, to 46.25 for 64 processes. The corresponding *efficiency* values [3], i.e. speedups divided by numbers of processes, range from 78 % to 72 %. Thus the efficiency is well below 100 %. Indeed, because of communication costs, the algorithm running with p processes is less than p times faster than with 1 process.

Note that the values from Table 4 are based on exact computing times measured under condition that the algorithm with p processes must execute the same total number of evaluations as with 1 process. One can say that such measurement is unjust since it does not take quality of solution into account. Indeed, it is true that the time with p processes is longer than the sequential time divided by p, but in that longer period the algorithm also obtains better solutions. Consequently, it would

Problem	Number of processes (p)							
instance	1	4	8	12	16	24	32	64
CMT01	53.16	2.50	2.90	1.22	0.36	0.18	0.12	0.04
		21.24×	18.30×	43.72×	148.24×	296.75×	433.46×	1 424.42×
CMT02	67.54	18.75	5.77	3.83	3.59	2.51	1.56	1.15
		3.60×	11.70×	17.63×	18.81×	26.89×	43.41×	58.83×
CMT03	125.77	37.22	19.41	6.50	4.24	2.60	3.12	0.88
		3.38×	6.48×	19.35×	29.69×	48.34×	40.31×	142.18×
CMT04	193.28	21.35	12.01	5.83	4.27	2.92	2.15	0.95
		9.05×	16.09×	33.16×	45.24×	66.11×	89.77×	203.35×
CMT05	241.91	22.64	16.58	7.47	7.49	3.50	2.78	1.82
		9.08×	14.59×	32.38×	32.30×	69.03×	87.15×	132.77×
CMT11	177.04	24.72	10.05	6.53	3.18	2.10	1.69	0.66
		7.16×	17.62×	27.10×	55.71×	84.22×	104.63×	267.92×
CMT12	108.41	10.89	5.70	3.48	2.27	2.29	1.31	0.70
		9.96×	19.01×	31.13×	47.80×	47.28×	82.72×	155.73×
Average	138.16	20.30	10.35	4.98	3.63	3.57	1.82	0.89
		9.07×	14.83×	29.21×	53.97×	91.23×	125.92×	340.74×

Table 5. Computing times needed to reach target solution quality

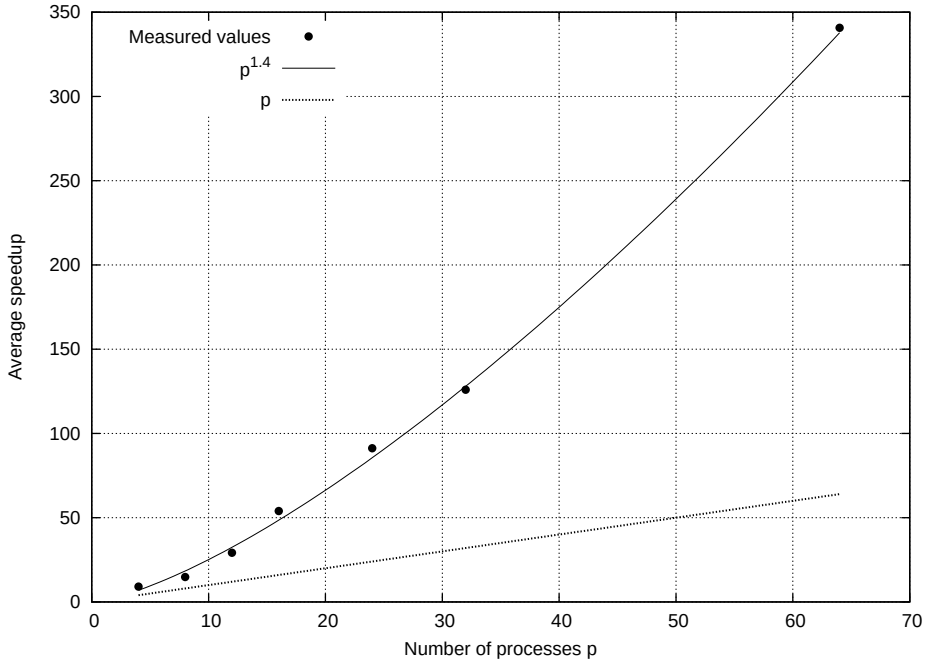

Fig. 3. Reaching a target solution quality – average speedup vs. number of processes

be more appropriate to compare versions of the algorithm that produce the same solution quality.

To enable such more appropriate assessment of speedup, we have monitored the execution of our algorithm in more detail, and recorded the exact moments when it obtains certain solutions. In other words, we have collected the data needed to simulate a different termination condition, which stops the algorithm when it reaches a desired solution quality. By scanning the detailed recordings, it has always been possible to determine the exact time when the algorithm with p processes obtains the solution with the same relative error as the sequential algorithm performing all 4 000 000 evaluations. It has indeed always been possible thanks to the fact that the distributed algorithm finally achieves a better solution than the sequential algorithm. The times determined in this way and the corresponding speedups are given in Table 5. The same data are graphically presented by Figure 3.

As we can see from Table 5, the average speedup computed by taking into account solution quality ranges from 9.07 for 4 processes, to 340.74 for 64 processes. The equivalent efficiency values range from 226 % to 532 %. Thus, we deal here with a superlinear speedup since its value for p processes is well above p, and the efficiency is well above 100 %. Figure 3 shows that the values from Table 5 interpreted as a function of p can quite closely be approximated by the superlinear function $p^{1.4}$.

6 DISCUSSION

In the previous section we have presented some results dealing with the speedup of our distributed evolutionary algorithm. Since the considered algorithm is fairly complex, measuring its speedup is certainly not a straightforward task. Consequently, it would not be a surprise if some readers come up with their own ideas how the speedup should be computed. In this section we analyze some possible criticisms regarding our calculations and explain why we still believe that our approach is the most appropriate.

First of all, some people would not be pleased with our definition of speedup. Indeed, we measure the so-called *weak* speedup [1, 3] where a distributed algorithm is compared to the sequential version of the same algorithm (i.e. to the same algorithm running with the number of processes set to 1). There are some other definitions found in literature: e.g. the *strong* speedup [1] where the "best known" sequential algorithm is involved, or the *orthodox* speedup [1] where a distributed algorithm is run for comparison as a set of concurrent (quasi-parallel) processes on one processor.

Obviously, the weak speedup is the most appropriate for our purposes. Namely, our aim was to study the effects of distribution to an evolutionary process. Thus we have to compare our distributed algorithm to its sequential version where evolution is still present but distribution is missing. The strong speedup does not make sense because the "best known" sequential algorithm is probably not evolutionary. Also, the orthodox speedup is not suitable since it is based on a sequential algorithm that produces exactly the same results as the distributed algorithm – thus

only load-balancing capabilities are measured and not any algorithmic improvements.

As mentioned before, there is a slight problem with the weak speedup in our particular case. Namely, there are in fact four versions of the sequential algorithm, each with a different crossover operator. To overcome this problem, we have always used the results of the version that is the best for a particular problem instance. In this way, our assessment of speedup is more stringent, and it has a flavor of "strongness".

Some readers may feel that our method of computing speedup is not fair, because the distributed algorithm can use all four crossover operators and the sequential algorithm only one. Indeed, on the first sight it seems plausible that the sequential algorithm, if allowed to run all four crossovers, would be able to produce better results, thus cutting down the advantages of the distributed version.

To address this issue, let us repeat again that prior to our main experiments we have done a lot of preliminary testing. In addition to the "plain" sequential versions of the algorithm using one crossover each, we have also tested the "mixed" version that picks up among four crossovers randomly with equal probability. It turned out that for each particular problem instance one plain version is the best (not always the same one!), while the mixed version is somewhere between the best and the worst plain version. Thus if we based our calculations on the mixed version, the speedups would be even larger than they are now. Consequently, our approach with best plain versions is in fact more rigorous.

Another possible reason why our way of computing speedup may look unjust is the following. The population size in the sequential algorithm is 30, while in the distributed algorithm each process has a population with size 30, thus making the total number of involved chromosomes much larger. So the sequential version seems to be in a bad position since it must choose its final solution from a much smaller set of available solutions.

To address the last criticism, we have done some additional testing. We have run the sequential algorithm on our problem instances with a fixed number of evaluations but with a varying population size. It turned out that, in general, increasing the population size above a certain threshold has a negative effect on the solution quality; or, differently speaking, all that really counts is the number of evaluations, not the number of chromosomes. Consequently, our approach to computing speedup is appropriate, and the chosen population size 30 per process seems to be just right.

The observed behavior of the sequential algorithm regarding the population size can be explained in the following way. If the population size is large, then the given number of evaluations is spread among many chromosomes and each chromosome receives a small portion of processing in average; so there is a big choice of solutions, but all of them are immature. On the other hand, if the population is smaller, then the same number of evaluations is applied to less chromosomes and some chromosomes are improved several times. Thus there is a choice among a smaller number of well developed solutions. So putting it all together, we can expect similar results if the processing effort is the same, no matter how large is the set to be processed.

Problem Instance	Population size							
	1×30	4×30	8×30	12×30	16×30	24×30	32×30	64×30
CMT01	541.48	540.69	542.11	544.21	547.20	551.87	553.24	565.77
	3.2 %	3.1 %	3.3 %	3.7 %	4.3 %	5.2 %	5.5 %	7.8 %
CMT02	865.95	867.49	878.49	885.86	888.20	890.30	893.27	910.26
	3.7 %	3.9 %	5.2 %	6.1 %	6.3 %	6.6 %	6.9 %	9.0 %
CMT03	861.12	855.61	865.61	879.34	879.85	895.72	902.88	928.63
	4.2 %	3.6 %	4.8 %	6.4 %	6.5 %	8.4 %	9.3 %	12.4 %
CMT04	1 107.55	1 098.94	1 106.34	1 107.31	1 124.29	1 138.22	1 154.99	1 194.28
	7.7 %	6.9 %	7.6 %	7.7 %	9.3 %	10.7 %	12.3 %	16.1 %
CMT05	1 415.05	1 391.60	1 404.35	1 425.95	1 443.09	1 459.34	1 469.02	1 521.85
	9.6 %	7.8 %	8.7 %	10.4 %	11.7 %	13.0 %	13.7 %	17.8 %
CMT11	1 076.57	1 084.33	1 094.40	1 103.07	1 111.23	1 119.64	1 126.01	1 143.71
	3.3 %	4.1 %	5.0 %	5.8 %	6.6 %	7.4 %	8.1 %	9.7 %
CMT12	826.14	827.85	832.28	835.01	836.85	837.63	841.48	856.07
	0.8 %	1.0 %	1.6 %	1.9 %	2.1 %	2.2 %	2.7 %	4.5 %
Average	4.6 %	4.3 %	5.2 %	6.0 %	6.7 %	7.6 %	8.4 %	11.1 %

Table 6. More solutions – one process, mix of crossovers, fixed number of evaluations

In reality, the performance with a larger population becomes even worse. This is because the algorithm spends more time for the same number of evaluations since it has to manipulate larger data sets.

Our claims about the sequential version of the algorithm are documented in more detail by Table 6. The shown results have been obtained with the mixed sequential version, but analogous results are also available for plain versions. The table shows the solutions for a fixed number of evaluations and varying population size. Each row corresponds to a particular problem instance, and each column to a particular population size. Each solution is again described by its cost and as the relative error versus the best known solution. The number of evaluations is fixed to 4 000 000, i.e. it is the same as in the previous round of experiments described in Sections 4 and 5. The population size ranges from 30 up to 64×30, thus mimicking the total number of chromosomes in the previous experiments with 1 or up to 64 processes.

By comparing the first column in Tables 3 and 6, we can see that the mixed sequential version of the algorithm really produces worse results than the best of plain versions. Mutual comparison of columns within Table 6 clearly shows that increasing the population size does not improve the solution quality, quite contrary.

7 CONCLUSIONS

In this paper we have demonstrated that EAs for the VRP can be considerably improved by distribution. Indeed, we have presented a distributed algorithm that, according to very strict measurements, achieves a superlinear speedup. Thus our algorithm not only runs faster due to parallel computing, but it also produces better

results for the same total amount of computing. Such improvement is possible because a set of heterogeneous and fairly isolated evolutionary processes can assure more diverse search for solutions than a single process.

In this paper we were only interested in exploring the potential of evolutionary algorithms to solve the VRP, and how that potential can be enhanced by distribution. We were not aiming to design the most competitive algorithm for the VRP so far. Consequently, we restricted only to evolutionary operations and migration. Otherwise it would not be possible to determine whether the obtained quality of solution is really assured by evolution and distribution, or by other paradigms.

Of course, if we need even better quality of solution, we could still upgrade our algorithm with additional heuristics. For instance, a better initial population could be created by some constructive heuristics; or the subsequent populations could be optimized by some form of local search. Then we would obtain solutions comparable to those produced by the best available hybrid algorithms.

REFERENCES

[1] Alba, E.: Parallel Evolutionary Algorithms Can Achieve Superlinear Performance. Information Processing Letters, Vol. 82, 2002, No. 1, pp. 7–13.

[2] Alba, E.—Dorronsoro, B.: Solving the Vehicle Routing Problem by Using Cellular Genetic Algorithms. In: J. Gottlieb and G. R. Raidl (Eds.): Proceedings of the 4th European Conference on Evolutionary Computation in Combinatorial Optimization (EvoCOP 2004), Coimbra, Portugal, April 5–7, 2004. LNCS Vol. 3004, Springer Verlag, Berlin 2004, pp. 11–20.

[3] Alba, E. (Ed.): Parallel Metaheuristics: A New Class of Algorithms. John Wiley and Sons, Hoboken (NJ) 2005.

[4] Alba, E.—Dorronsoro, B.: Computing Nine New Best-so-far Solutions for Capacitated VRP with a Cellular Genetic Algorithm. Information Processing Letters, Vol. 98, 2006, pp. 225–230.

[5] Berger, J.—Barkaoui, M: A Parallel Hybrid Genetic Algorithm for the Vehicle Routing Problem with Time Windows. Computers and Operations Research, Vol. 31, 2004, pp. 2037–2053.

[6] Choi, I.—Kim, S.—Kim, H.: A Genetic Algorithm with a Mixed Region Search for the Asymmetric Traveling Salesman Problem. Computers and Operations Research, Vol. 30, 2003, pp. 773–786.

[7] Diaz, B. D.: The VRP Web. Languages and Computation Sciences Department, University of Malaga, 2010. Available on: http://neo.lcc.uma.es/radi-aeb/WebVRP.

[8] Gutin, G.—Punnen, A. P. (Eds.): The Traveling Salesman Problem and Its Variations. Kluwer Academic Publishers, Dordrecht (NL) 2002.

[9] Hwang, H.-S.: An Improved Model for Vehicle Routing Problem with Time Constraint Based on Genetic Algorithm. Computers and Industrial Engineering, Vol. 42, 2002, pp. 361–369.

[10] Isabella Computer Cluster. University Computing Centre, Zagreb 2010. Availaible on: `http://www.srce.hr/isabella`.

[11] Larranga, P.—Kuijpers, C. M. H.—Murga, R. H.—Inza, I.—Dizdarevic, S.: Genetic Algorithms for the Travelling Salesman Problem: A Review of Representations and Operators. Artificial Intelligence Review, Vol. 13, 1999, pp. 129–170.

[12] Michalewicz, Z.: Genetic Algorithms + Data Structures = Evolution Programs. Third Edition, Springer, New York (NY) 1995.

[13] Ochi, L. S.—Vianna, D. S.—Drummond, L. M. A.—Victor, A. O.: A Parallel Evolutionary Algorithm for the Vehicle Routing Problem with Heterogeneous Fleet. Parallel and Distributed Processing, LNCS Vol. 1388, Springer Verlag, Berlin 1998.

[14] Pongcharoen, P.—Stewardson, D. J.—Hicks, C.—Braiden, P. M.: Applying Designed Experiments to Optimize the Performance of Genetic Algorithms Used for Scheduling Complex Products in the Capital Goods Industry. Journal of Applied Statistics, Vol. 28, 2001, No. 3 and 4, pp. 441–455.

[15] Prins, C.: A Simple and Effective Evolutionary Algorithm for the Vehicle Routing Problem. Computers and Operations Research, Vol. 31, 2004, pp. 1985–2002.

[16] Quinn M. J.: Parallel Programming in C with MPI and OpenMP. McGraw-Hill, New York (NY) 2003.

[17] Tan, K. C.—Lee, L. H.—Zhu, Q. L.—Ou, K.: Heuristic Methods for Vehicle Routing Problem with Time Windows. Artificial Intelligence in Engineering, Vol. 15, 2001, pp. 281–295.

[18] Toth, P.—Vigo, D. (Eds.): The Vehicle Routing Problem. SIAM Monographs on Discrete Mathematics and Applications, SIAM, Philadelphia 2002.

[19] Zhong, Y.—Cole, M.H.: A Vehicle Routing Problem with Backhauls and Time Windows: A Guided Local Search Solution. Transportation Research Part E: Logistic and Transportation Review, Vol. 2, 2005, pp. 131–144.

Krunoslav Puljić received his B. Sc., M. Sc. and Ph. D. degrees in mathematics from the University of Zagreb in 1999, 2004 and 2009, respectively. He is now a research assistant at the Department of Mathematics, University of Zagreb. His research interests include combinatorial optimization, evolutionary algorithms and object-oriented software engineering. He has published two papers in scientific journals and two more papers in conference proceedings. He is a member of the Croatian Mathematical Society.

Robert Manger received the B. Sc. (1979), M. Sc. (1982), and Ph. D. (1990) degrees in mathematics, all from the University of Zagreb. For more than ten years he worked in industry, where he gained experience in programming, computing, and designing information systems. He is presently a Professor at the Department of Mathematics, University of Zagreb. His current research interests include combinatorial optimization, parallel and distributed algorithms, and soft computing. He has published 20 papers in international scientific journals, over 30 scientific papers in conference proceedings, 10 professional papers, and 4 course materials. He is a member of the Croatian Mathematical Society, Croatian Society for Operations Research and of IEEE Computer Society